Load Balancing for Power Minimization in Networks on Chips

MATTHIAS BAUHOFER

Master of Science Thesis
Stockholm, Sweden 2006

ICT/ECS-2006-20

Master of Science Thesis

Load Balancing for Power Minimization in Networks on Chips

Matthias Bauhofer

Electronic, Computer, and Software Systems
Royal Institute of Technology
Stockholm

ROYAL INSTITUTE
OF TECHNOLOGY

Supervisor: Prof. Axel Jantsch
April 27, 2006, Stockholm

Abstract

To satisfy the communication needs of multiple cores in a System On Chip design the Network On Chip solution has been introduced in the past. Using global allocation and local scheduling of tasks this master thesis investigates the possibility to save energy by using different allocation algorithms.

The predecessor for this master thesis [Tho05] developed a simulator that allows a task to make use of processing and communication resources and it uses a global allocator to assign tasks at runtime to one of the equal processing elements. Neither the network nor the processing element models simulate any dynamic behavior during runtime.

By adding a network simulation (SEMLA, the NOSTRUM network core) this thesis develops an extended simulator that adds those dynamic behaviors and some more system details to the previous one. Further a decentralized load balancing system gets introduced.

As result the thesis shows the qualities and usability of the different allocation algorithms regarding the energy consumption and it compares those influences to the power needs of the newly introduced simulation details.

The thesis concludes that given the developed simulator the influence of the allocation algorithms on the needed energy is not very dominant compared to influences from the added simulation details and that the effectiveness of the allocation strategies for saving energy strongly depends on the used system configuration.

Keywords: System on Chip, Network on Chip, global task allocation, decentralized allocation algorithm, energy consumption, low power, DVS

Abstrakt

För att tillfredställa kommunikationsbehovet hos multipla processorer i ett System on Chip kommunikations nätverk har introducerats tidigare. Med hjälp av global allokering och lokal scheduling undersöker denna avhandling möjligheterna till energibesparing genom användandet av alternativa allokerings algoritmer.

Föregångare till denna avhandling utvecklade en simulator som tillåter en uppgift att använda sig av behandling och kommunikationsresurser och använder sig av global allokering för att distribuera uppgifter under processens gång till en av de likvärdiga processor elementen. Varken nätverket eller processor elementen simulerar något dynamiskt beteende under processens gång.

Genom att lägga till en nätverkssimulering (SEMLA, NOSTRUMs nätverkskärna) utvecklar denna avhandling en utökad simulator som lägger till de dynamiska beteendena och några ytterligare systemdetaljer till den föregående simulatorn, dessutom introduceras ett decentraliserat avlastningssystem.

Som resultat visar denna avhandling kvaliteterna och användbarheten av de alternativa allokerings algoritmerna beträffande energikonsumptionen. De tillagda simulations detaljerna visar på en stor påverkan över resultatet i simulationen.

Avhandlingen avslutas med slutsatsen att, givet den utvecklade simulatorn, allokerings algoritmerna inte är särskilt dominanta i jämförelse med influenserna från de tillagda simulationsdetaljerna på den nödvändiga energiåtgången och att effektiviteten hos allokeringsstrategierna för energibesparing påverkas kraftigt av den använda systemkonfigurationen.

Acknowledgments

This master thesis was developed and written at the Electronic, Computer, and Software Systems department at Royal Institute of Technology in Stockholm during the academic year 2005/2006.

I would like to thank my supervisor Professor Axel Jantsch for his assistance as the project did go along. Zhonghai Lu provided me with the SEMLA code and help in understanding it.

Thanks also go to Bjarke Thorman who introduced me to his work and the whole energy-saving-by-allocation world. Sandro provided me with support for the integration of his network power model in my simulation. Thank you guys!

Then there was my buddy Mathias Meyer, who helped with moral support and nice layout tips and I have to thank Magnus for helping me with the Swedish abstract.

Finally I would like to thank my girlfriend Sabine for going with me through all the tough and stressful times, during the period of this work!

Contents

1. Introduction

As companies recently have announced, the race for higher processor frequencies is over. It became more important and more feasible to produce processing units with multiple processing cores to increase the computation power instead of pushing the processor's clock frequency. In other words the long predicted use of multiple processing elements on a single chip has reached the every days desktop computer.

When using more and more independent components on a single chip instead of multiple chips the means of communications between the parts becomes crucial. The traditionally used bus architecture cannot fulfill the requirements anymore, instead a whole network on chip (NoC) can help out to connect all different processing components as shown in [BDM02].

Such components together form a system on chip (SoC) and can be all different, dedicated to solve special problems, or they can be identical so that they can support each other in computing a solution.

In the case that a problem can be solved by different components of the chip, the choice of the component that solves the problem can influence the actual power consumption of the chip, depending on the current resource utilization (see figure 1.1).

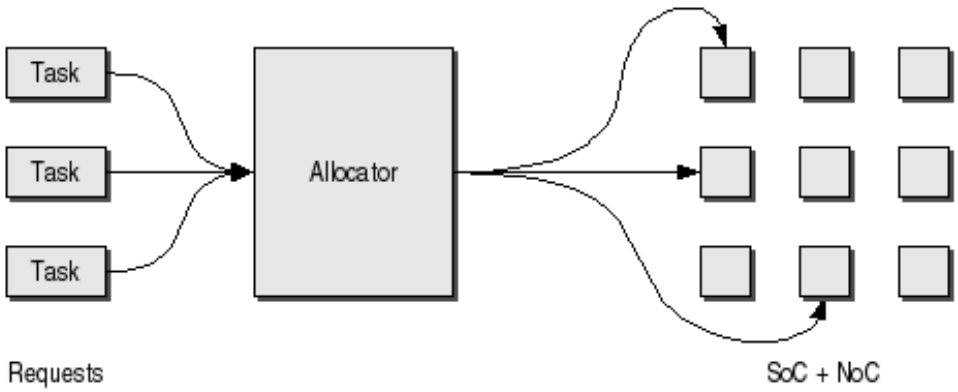

Figure 1.1.: During runtime a task creation request can occur. An allocation system decides then on which SoC component the task should be executed.

With this knowledge the predecessor of this master thesis [Tho05] developed a simulator that investigates the effect of different load balancing strategies on the power usage. Heuristics where used to model the components and the network on chip. To continue that work, this master thesis tries to refine the network and processing models to obtain a more sophisticated simulator that can show the impact of such additional simulator implementation details.

The term *load balancing* of the title of this master thesis refers to the allocation of a task to a processing element on the chip at the tasks creation time during run time. So, my task is to build an advanced SoC with NoC simulator that allows experiments of different task allocation strategies and shows the resulting power usage and energy consumption readings. I will call this simulator *SoC with NoC simulator* - SNS.

The targeted abstraction level of SNS includes the simulation of the traveling of each network message through the used NoC, the time consuming behavior of a task and the simulation of the operating system (or at least parts of it) on each processing component of the SoC.

To actually keep the scope of this master thesis within certain limits, SNS will not include the simulation of different SoC components (different types of processing elements). No memory management or usage will be taken into account and the instructions of a task exist only as counter for clock cycles that have to be consumed. Contrary to previous work all simulated software components will make use of the simulated hardware resources.

Of course all this is necessary to get a more realistic behavior of the simulation. If for example the on each processing component local scheduling of tasks does suddenly consume processing time as well, the overall execution of the tasks takes more time then without the scheduler's time consuming simulation detail.

Another important simulation detail that is worth mentioning here is the resource requirement of the central allocation component. By scaling the size of the used SoC components the central component of the allocation algorithm makes more and more use of its local resource until it reaches the limits of the local resources. Thus it is necessary to think of a different load balancing strategy at runtime, that does not need a single central component.

Besides developing a more realistic and detailed simulation, this master thesis will therefore introduce an additional allocation system and this new allocator will be compared with the central component based allocation algorithms.

1.1. Thesis layout

The document is divided into four parts. Part I will mention three other master theses which together build the theoretical basis for this work. Part II contains then extensions to the basic work and new features and III refers to the implementation of those additional simulation details. IV finally presents the experiments, results and conclusions of the work.

Chapter 2

It presents the corresponding master thesis and describes which parts of it are important and which parts are going to be reused by this work. In this chapter the master based allocation algorithms and the means for energy saving by DVS are described, too.

Chapter 3
Contains the description of the network simulator used by this master thesis. Its design and code are used to build the network simulation part of SNS.

Chapter 4
Since the previous chapter lacks the possibility of calculating the network's power this chapter covers that part. The underlying master thesis does not refer to the same network, nor to all parts of it, so the solution for that issue will be presented as well.

chapter 5
This chapter introduces the first new major feature from SNS. A work scenario is similar to a task graph and it is also used to generate tasks pile for SNS. What more it can do, you can find in the chapter.

Chapter 6
Since SNS is basically an extension of two other master theses the content of this chapter describes the new major parts of the developed simulator. These are the advanced processing element model and the operating system service module.

Chapter 7
The new non-central based load balancing technique is described. It will be shown how task allocation and the task's address management can work without a master component.

Chapter 9
The network simulator SEMLA was described in 3 and here now it is time to show how it was modified and extended to be usable within SNS. Basically the whole SEMLA-code was rewritten and adapted to be fully functional.

Chapter 10
The previous presenter processing element (in 2) does not support any dynamic resource usage simulation during runtime. This major feature and others are described in this chapter and it will be explained how this makes the overall simulation more realistic.

Chapter 11
To allow other components such as tasks and allocators the access to network and processing resources the operating system service offers the needed means. It all will be described in this chapter.

Chapter 12
After knowing what parameters a task has and what services are offered to it, this chapter describes how its behavior can be simulated and how it is done in SNS.

Chapter 13
What would be the best simulator if we could not look at what happens during the simulation? This chapter describes all the results that a run of SNS can offer.

Chapter 14
To perform experiments with SNS it is necessary to configure all the used components. The description of them and the why are presented in this chapter of the thesis.

Chapter 15
Finally we can take a look at the results of the simulations. Which experiments where done and what for, gets explained in this chapter.

Chapter 16
What the results actually mean and why they are like they are is then explained in this conclusion chapter.

Chapter 17
Last but not least some ideas of how to continue this work will be presented in this last chapter.

Appendix A
It shows additional results from the previous experiments. With that the obtained energy readings can maybe be understood better.

Appendix B
The nearly constant network power consumption of the performed experiments is shown again in this appendix by a different example.

Appendix C
The thesis ends with a short glossary.

Part I.

Related Work

2. Dynamic resource allocation

The work of Bjarke Thorman [Tho05] is the starting point of this master thesis. Based on his work a new simulator has been created.

The dynamic resource allocation considers the following scenario. We have:

- packet switched network with mesh architecture
- uniform network resources, called processing units, that represent state-of-the-art microprocessors
- real time multiprocessor operating system
- real time periodic tasks with one for each task defined communication partner (one way communication, no other dependencies between tasks)

And we want: *load balancing* that saves as much energy as possible.

To show that the proposed solution actually works Thormann wrote a simulator and used it to perform his experiments with different types of tasks and different types of load balancing algorithms.

2.1. Energy consumption

Reducing the power needs of a *system on chip* (SoC) with a *network on chip* (NoC) can be done at many stages during the development of a product. Lots of hardware and software solutions have been invented to reduce the overall energy consumption.

It also is easier to do energy optimizations for a particular application then to do so by inventing general energy saving rules. An effective way of saving energy regarding any system[1], is to take a look at the actual power model and ask the question which values have the most influence?

The power consumption is mostly influenced by the power quota used by the switching activity of the hardware (dynamic power of the logic) and the power quota of the hardware in the no-activity state (leakage power). The later one is directly related to the kind of technology used for the solution and cannot be influenced at run time. It is therefore not relevant for an at runtime load balancing consideration[2].

The dynamic power of the hardware can be described by the following formula:

$$P = \alpha \cdot \left(C_L \cdot V_{dd}^2 \cdot f_{clk}\right)$$

[1] Using CMOS logic.

[2] The leakage power is also influenced by the used voltage; lower voltage is better. But since this is valid for the dynamic power use as well, we still can ignore the leakage power reduction at runtime.

P is the resulting power value in watt [W], α describes the current switching activity (how many gates did change their state) and its range lies between 0 and 1. The technology has its influence in the C_L parameter, that is the load capacitance of the logic represented in farad [F]. The V variable represents the used voltage and is given in volt [V]. The last parameter f_{clk} is the frequency with which the gates switch and its unit is hertz [Hz].

Looking at this formula, the best way to reduce the power reading would be to reduce the voltage as much as possible. Of course that is not possible. The used technology defines a lower limit for it and there is not much room for optimization. The capacitance is also determined by the used technology. Maybe in particular applications the switching activity can be reduced but generally it cannot be influenced much to save energy.

This indicates that a change on the frequency might be the way for reducing the power. In fact there exists the possibility of changing the frequency and with it the voltage too. It is called *dynamic voltage scaling* (DVS). This technique is based on the fact that reducing the voltage requires the use of a lower frequency as well. Since the power costs per (clock) cycle are not depending on the used frequency, but depend only on the used voltage, the reducing of the voltage is what reduces the energy consumption (regarding the computation of a task) and the lower frequency is just a necessary side effect.

Bottom line, for my master thesis I assume that the used processing elements can reduce there power consumption by reducing there execution speed and the power is described by the following formula:

$$P = P_{max} \cdot S^3$$

Where P_{max} represents the technology parameter and S (its range goes theoretically from 0 to 1) represents the actual speed of the processing element, relative to the maximal speed. Using DVS, the frequency is proportional to the used voltage and so we get the S^3 polynomial characteristic of the power to speed relation (more detailed explained in Thormann's thesis [Tho05]).

2.1.1. Reducing network energy

The use of dynamic voltage scaling requires the knowledge of the current utilization of any controlled resources. Since it is rather difficult to measure the overall network usage at any time and the measurement alone would require additional network resources, choosing DVS to reduce the network power usage is not a practical choice.

We need other methods to reduce the network power. And we need to reduce the network power because it has a significant influence on the overall power consumption (depending on the size of the NoC and the used resource that can be up to 40% and more).

So, instead of trying to change the speed of the network we go one step back and remember that also the (switching) activity has a direct (proportional) influence on the network's power readings. Generally we observe the less the activity per communication, the more power we can save.

One variable that has a big influence on the networks activity is the architecture of the network. The type of the network determines how the network nodes (locations) are connected with each other and how they can communicate with each other. With that the architecture influences the average energy per communication cost.

For example a torus architecture (figure 2.1-left) allows shorter connections between two nodes than a mesh network (figure 2.1-right) and with that it can reduce the overall power consumption.

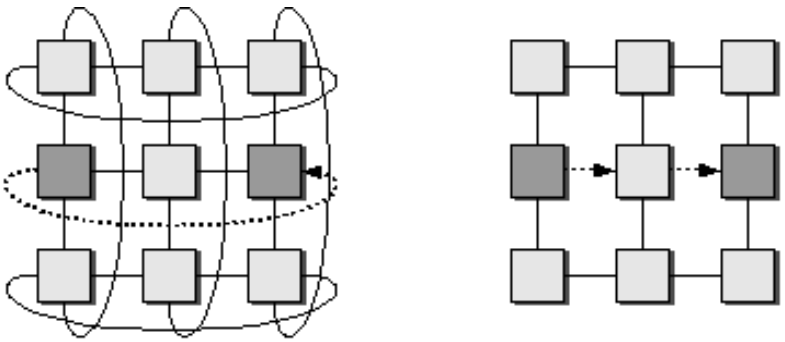

Figure 2.1.: Torus (left) and mesh (right) network architecture. Sending the specific messages involves 1 hop in the torus network and 2 hops in the mesh network.

Since the NOSTRUM project (involving my supervisor) [Homa] recommends the use of a mesh architecture (because of among other things the regular, simple to use layout) I too will use that architecture for SNS and so the power savings, derived through another technology, do not influence my work.

There is though another component that influences the path of network messages. For the packet switched network each message gets first split into smaller fragments (with a fixed maximal size) and then those fragments get routed through the network. The used routing algorithm influences directly the amount of energy, used for delivering a network packet. However not only the designate paths of the network messages influence the power but the network switches (the logic that decides where to send a packet) itself does so, too.

Regarding this issue NOSTRUM makes the suggestion to use a deflection routing algorithm (which I will not explain a this point yet) that uses a fairly minimal amount of logic and with that comes at a low power cost.

Apart from the switches there exists more logic, that is part of a network, but that logic too follows the above described idea.

More complex logic might reduce the overall communication related switching activity a bit but comes at a higher power cost (especially the use of big buffers is not advisable).

For example the use of channels (assigning one special route between a source-destination pair to a certain type of packets for a certain amount of time) can reduce the needed complexity of the logic and with that reduce the power per message costs but if channels get created and revoked to often (short bursts of communication) the management of the opening and closing of channels might

use all the saved power or more likely even consume more power.

All in all again, if we have a particular application we can use one or another optimization and reduce the power significantly and reliably but otherwise there is no real killer-method to save power for the network used in SNS.

One possibility however, we do have. And that is the one that is of most interest for Thormann's master thesis [Tho05]. It is the observation that, knowing with whom the task communicates, it is possible to place each two communication partner as close as possible (at runtime) and so create mostly local traffic. This traffic that involves a minimal amount of switches can help saving power and the next section discusses how such a load balancing can be done.

2.2. Allocation algorithms

The basic unit, considered at this point, is a *task*. It represents a set of program instructions that have to be executed somewhere on the SoC (assuming all processing elements are equal). A task can have certain properties like a maximal and minimal boundary regarding the amount of instructions that it uses, or like the requirement to send a message to another task.

In any case within the context of this master thesis (and Thormann's work) at some point of time the execution of a task starts and some time later the execution ends and there exists a limit for how much time is allowed to pass between the start end the end time (real time requirements). Further it is assumed that the system has to execute more tasks then it has processing elements, which requires a processing element to be capable of handling more then one task at a time.

When it is time to start the execution of a task two questions turn up:

- where to execute a task (on which location)
- how does the new task affect the execution of the already existing tasks

The issue of those two questions can be answered in many different ways. For once we could use one global instance that knows about all tasks and each time a new task comes up (or an existing task finishes) it distributes all tasks in an optimized way across the SoC. But this system requires the possibility to move the execution of a task from one processing element to another during the execution time of a task. And that would require to transfer all relevant task states to another processing element which creates additional network traffic and energy needs.

We try to avoid that and use another solution that does not include task migration. We separate the issues of the two questions and get two problems to solve. By using a local *scheduler* for each processing element, which decides how much and at which time each task can make use of the local processing resources, we can solve the second issue (how the execution of one task affects others). As for the question where to put a task, we still might have to use a global system. We let an *allocator* make that decision. The resulting solution is illustrated in figure 2.2-right.

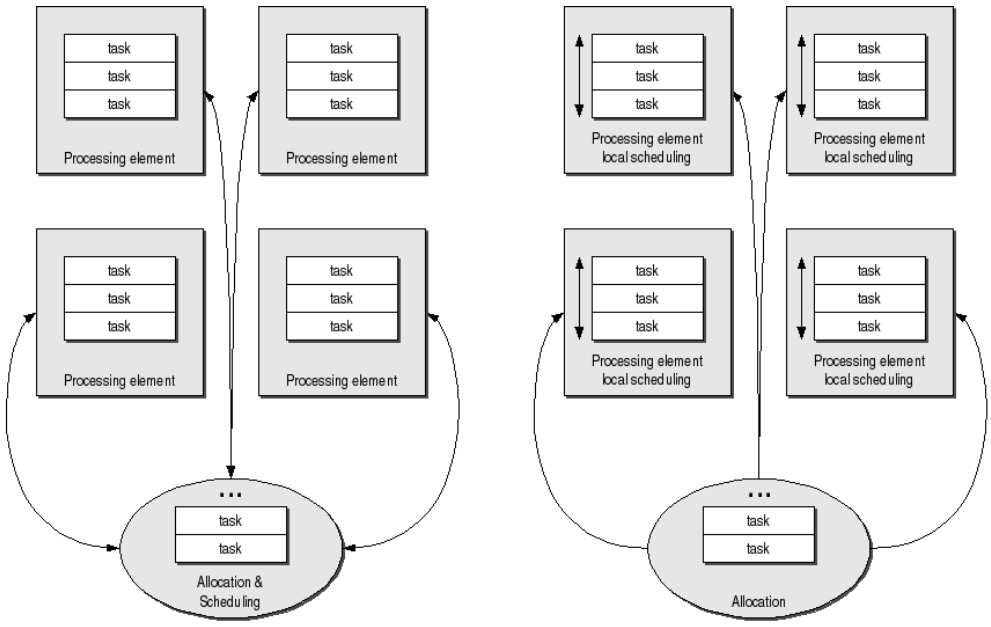

Figure 2.2.: Left we have a global scheduling and allocation schema with task migration. The right system uses a global allocation and a local scheduling, without task migration

The goal of an allocation algorithm is to decide during run-time which task to execute on which network location. The information on which the algorithm has to base its decisions is not always the same. Thormann has collected and described some allocation algorithms(the most common and useful ones) including a new by Thormann invented one, that make their decisions based on the following informations:

- the worst case execution time (how much processing resources are needed)
- the worst case injection rate (how much network communication resources are needed)
- the communication partner (to which network location does the generated traffic go)

Based on this information the following algorithms place each task to a processing element. All of them are simple enough for using at run-time and all of them come originally as solutions for the bin-packing problem (fill up k identical bins with a collection of different weights).

First-Fit

If a new task has to be created this algorithm searches through all processing elements and the first one that has enough computation resources left to be able to handle the new task, gets the new task.

This algorithm concentrates the tasks at a very local selection of processing elements and as more and more tasks come along it fills up all remaining elements with work, too.

This strategy is not all bad. It might keep the traffic local at some level and if some processing elements have nothing to do at all the algorithm even might save some energy.

Next-Fit

This algorithm is much like the first-fit algorithm, but at the point where the current processing element cannot handle the new task, it never gets another task(until the current period is over) even if it would be able to handle it.

With this modification it cannot happen that a task is put somewhere very far away from a previous (or further) created task.

Like the first-fit algorithm this one also might save energy with the assumption that tasks which are created one after another are more likely also the communication partners for each other and with that mostly local traffic would by generated.

Best-Fit

This allocation algorithm and the next one try two complementary strategies in distributing the tasks. The best-fit algorithm places a task at that processing element which achieves the highest utilization with the new task.

By using this strategy the algorithm leaves as much processing elements for as long as possible without anything to do. But if the system's load is high(which is most likely the case) this strategy might not save much energy.

Worst-Fit

The worst-fit algorithm tries to do the opposite then the best-fit solution. Each task gets assigned to the processing element which has the lowest utilization (with the new task included).

With that this algorithm keeps the maximum utilization as low as possible. Thinking of the polynomial rise of the power from a processing element regarding the used speed, this strategy might indeed help saving energy. If it is better to have some processing elements at high speed and the rest idle or having all processing elements working at nearly the same (lower) speed, depends highly on the actual (power influencing) properties of the used processing elements.

2.2.1. Energy aware allocation

The previous described allocation algorithms do not make use of the two communication related informations that a task provides at creation time. Instead they just use the processing utilization information about a task to decide where to put it. Together with the use of a dynamic voltage scaling this can already achieve considerable power consumption reduction.

To reduce not only the computation power but also the communication power Thormann introduces in his work [Tho05] a new allocation algorithm called *minimum-fit*. This algorithm estimates the network power costs by calculating the power used by the given communication amount for the given communication partner. Together with the given computational utilization the algorithm calculates a power cost factor for each (possible) network location and picks the one with the lowest cost:

```
Minimum_Fit()
{
  float pe_cost; # PE power cost
  float net_cost; # network power cost
  pe_type opt_pe; # variable for optimal PE.
  float min_cost; # variable for minimal cost.
  for every pe
  {
    total_cost := pe_cost + net_cost;
    if ((total_cost < min_cost) || first_iteration)
    {
      min_cost := total_cost;
      opt_pe := pe;
    }
  }
  return opt_PE;
}
```

Since this algorithm is the only allocator that considers both the processing utilization and the communication costs it is the favorite for energy saving.

The problem with that algorithm is to calculate the communication costs. Not every task might be reducible to the required task model and the information where the communication partner actually will be executed might not be available as well.

Altogether the first four algorithms do not provide any particular energy saving (they just have some characteristic energy behavior) and this last algorithms has very tight assumptions. The next sections will show what to do and what is further more needed.

2.3. Implementation

As mentioned before the simulator used in Thormann's master thesis is the base for SNS, developed during this master thesis. Therefore I will explain now in short how his (original) simulator works and which components are reused.

Generally speaking Thormann's work allows a task to be executed on a certain network location and the task can make use of processing and network communication resources. That means a task can claim a certain amount of the available resource. It does not mean that a task gets actually executed, or that a task actually sends network messages.

At the top of Thormann's simulation we have the allocation algorithm (the master) which decides where to put a new task. The resulting structure can be seen in figure 2.3.

The experiments use tasks that request different amounts of each resource (for each they use more communication or more computational resources) and the experiments are done with different allocation algorithms. During a simulation the amount of generated tasks gets increased continuously, until all processing elements are fully occupied.

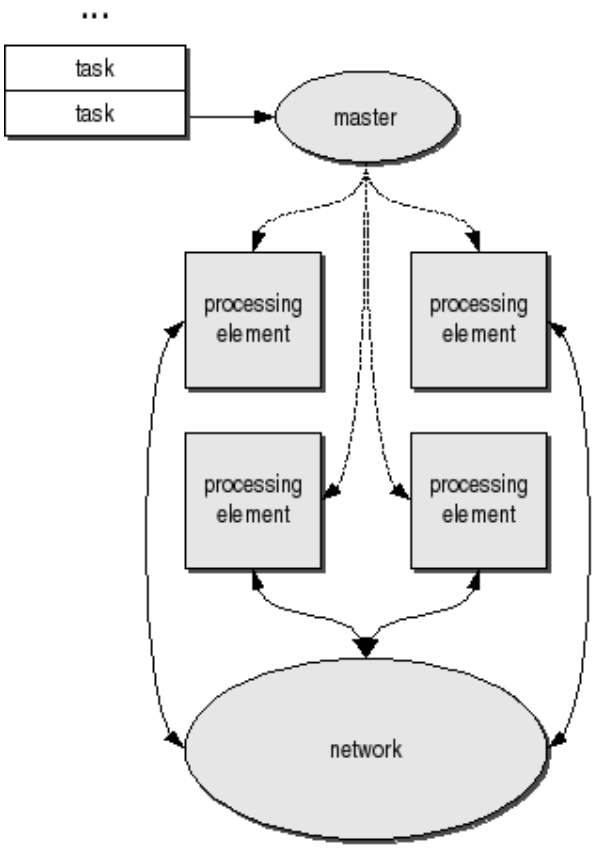

Figure 2.3.: Each task creation request gets handled by the master. It assigns the task to one of the processing elements. All processing elements are connected to the network.

Process

The process represents the set of instructions that can be executed on each processing element. Each process has its own set of the following properties, that control the process's claims for resources:

- pi: the period
- ci: the worst case execution time
- ri: the worst case injection rate
- cpi: the communication partner

Actually a process and a task are not exactly the same. A process is usually considered to represent a whole application whereas a task just is used to solve parts of an application. Since this difference it not important for my master thesis I will use *task* as the term for the entity that gets executed on a processing element.

I will reuse the idea of the parameterized task model in my simulation as well, although since the single communication partner and the injection rate

parameter are very tight restrictions in the task model, the communication qualities of a task will be probably represented in a different way.

Processing element

The processing element consists mainly of an utilization factor. 0 speed means the processing element has nothing to do and 1 means that the processing element cannot handle more tasks. Each task has for itself a processing resource request and the utilization of a processing element gets determined by those request, summed up from all local currently executed tasks.

I will continue to use this idea to simulate the use of a processing resource and expand the model with some more details. But we leave that for the later chapters of this thesis.

Master

Before a task gets created this central component decides where to put the task. The algorithm that makes that decision can be one of the above described allocation algorithms.

The component itself does not consume any network resources for this action. The task creation is delayed by the amount of time that it takes to reach the decision to which processing element the task should be assigned. Since the master does not operate on a specific processing element it does not consume any processing resource, either.

The master controls the overall energy consumption of the simulator by deciding on which processing element each task shall be executed. The decision is made during runtime and each time the current system state gets involved in the decision.

I will of course make use of this idea as well and let the allocation algorithm influence the energy consumption and I will try to estimated the network and processing resources that such a central component might use when placed on a particular processing element.

Network

Each task has exactly one communication partner. At the time that a task gets created the system knows where the destination for the task's communication is and how much of the network resource that task will use. With this informations (for each task) the overall network utilization can be estimated and consequently the power and energy consumption as well.

Since the task's communication goes in one direction, the counter part which receives the data does not necessarily have to exist. And based on the fact that the task does not actually "send" data, the system can estimate the network's utilization by just the communication amount and distance.

One reason for my master thesis is in fact to not use this kind of network abstraction. Instead I'll try to use yet another simulator (a network simulator) to analyze the communication effects and power costs.

2.4. Results

The results of Thormann's thesis [Tho05] clearly show that the energy aware minimum-fit allocation algorithm achieves the best power minimization during all the different experiments (variations of the network size and the communication and processing resource use). This can be explained by the fact that the minimum-fit algorithm is the only algorithm that actually considers the (power) cost of the different possible placements of a task and creates a far better power usage than the other algorithms.

All the same, I will not make use of this algorithm because it requires to tighter communication assumptions regarding the task (process) model. To make the minimum-fit allocator work each task can only have one communication partner and it must be known how much it will communicate with it and no dependencies between tasks are allowed.

Since I plan to allow more than one communication partner and let the task actually really send and receive network messages I cannot estimate the required informations and so I cannot use this particular allocator.

Between the remaining allocation algorithms I choose to use two different algorithms;

- the first-fit algorithm
- and the worst-fit algorithm

The reason why exactly those two is for once the difference between the two used strategies. The first-fit algorithm tries to fill up the processing elements utilization, processing element by processing element and could cause a certain amount of local traffic. The worst-fit algorithm on the other hand tries to hold the processing power as low as possible by distributing the tasks evenly between all processing elements. With this two different strategies I hope to show what differences in terms of energy consumptions are possible by using different allocation algorithms.

Then there is another reason for why just use two algorithms. Besides that they can show the (worst and best) possibilities I just selected two for not getting to many results. I will add one more (new) allocation algorithm and the simulation will also offer a lot of other possibilities to influence the energy consumption such that the use of three different allocation algorithms will be enough.

3. SEMLA - A network simulator

The SEMLA (*S*imulation *E*environ*M*ent for *L*ayered *A*rchitecture[1]) network on chip simulator it the result of Rikard Thid's master thesis [Thi02].

This simulator simulates a packet switched mesh network and is capable of simulating the actual way of a message's data through a mesh network. The simulator is based on the different implementations of selected layers from the ISO's OSI model [fSI94]. Each layer can be exchanged and with that virtually any behavior of a mesh network from different switching algorithms till faulty physical connections can be integrated and simulated.

The SEMLA simulator is also the core of the NOSTRUM environment [ZLNJ05]. The project adds a lot of features to the SEMLA core, but since I am just interested in a (simple) network simulation I will just use the SEMLA core (or at least its idea) and not all of the NOSTRUM extensions. Some advice or configurations however I will reuse for SNS(like the idea to use a mesh network and not another network architecture).

3.1. Network on chip

A *network on chip* (NoC) has to do all transfers of network messages (information data) between two resources of a *system on chip* (SoC), like in figure 3.1.

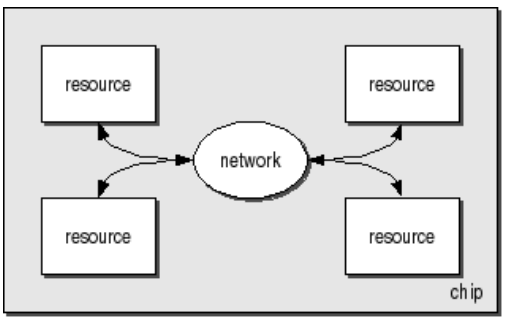

Figure 3.1.: Network on chip in a system on chip. The network has to connect all resources and provide communication services.

We have:

- a certain amount of modules (in context of this master thesis they are processing elements)
- the need to transfer data (packets of bytes) from one module to another one

[1] Curiously "semla" also is the Swedish equivalent for the Austrian "Faschingskrapfen"

From the view of the modules it does not matter how a message gets transfered, as long as the message gets transfered at all and within a certain amount of time. This point of view is typical for a certain layer in the context of a network and it is the application layer.

To design a fully functional network the *open systems interconnection* (OSI) reference model [fSI94] defines seven layers (views). Such a stack of layers exists on both communication sides and on top sits each time an (possibly different) application, as seen in figure 3.2.

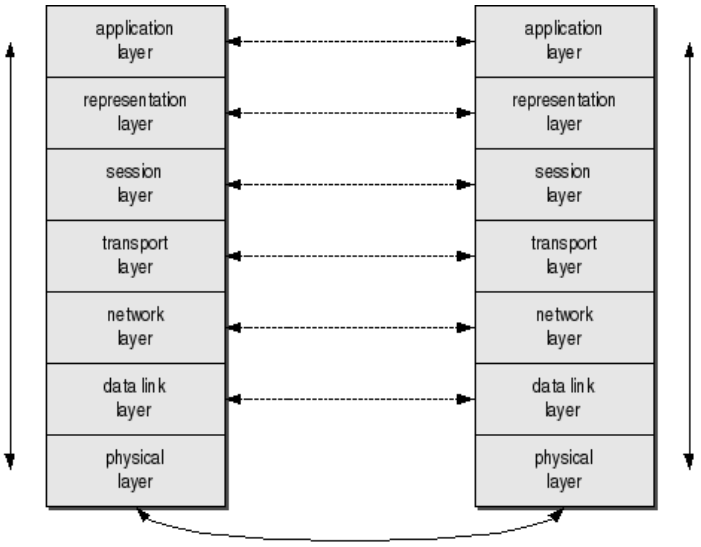

Figure 3.2.: The layers of the OSI reference model. Each layer virtually communicates with its counter part on the other side, but in fact the messages are passed vertically from layer to layer. Only the physical layers do have a real connection to each other.

When a message gets handled through the layers, each layer just takes the message from it's top layer, processes the message and forwards it to the bottom layer and to the other side; the same happens in the other direction (from the bottom to the top) until the restored original message reaches its destination.

Each layer has its own work to do and the layers are designed in such a way that no two tasks which are alike, are in different layers. But not all layers may be implemented. As for the SEMLA simulator the presentation and session layer are not needed and the application layer is reduced to an interface for accessing the transport layer. This leads to the remaining four layers that are in use:

- transport layer
- network layer
- data link layer
- physical layer

I now describe shortly each layer as how they are used in the network simulator (and will be also used in SNS).

Transport Layer

As the name says, the transport layer ensures the safe transport of the data. The messages that are handed over to this layer are first broken down into smaller fragments and then they get the necessary information for the reconstruction on the other side, before they are passed on to the network layer. This process is displayed in figure 3.3.

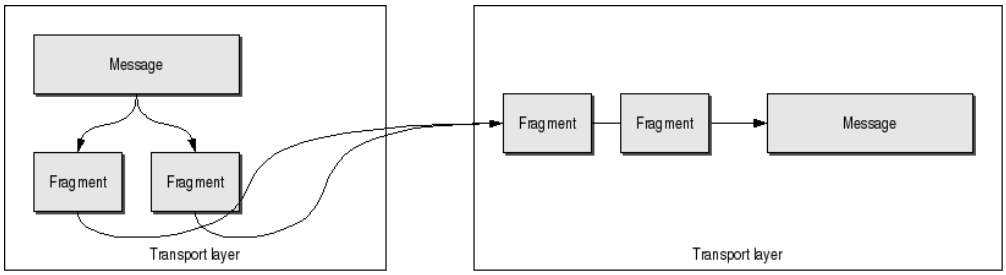

Figure 3.3.: The transport layer splits a message up into smaller fragments, suitable for the next layer. The receiver's transport layer must be able to reconstruct the message.

Fragments coming from the network layer get reassembled and the transport layer has to ensure that no part is missing and that the different fragments do not get mixed up.

In the original SEMLA code the transport layer actually just buffered the incoming messages and passed them on to the next layer without splitting them up.

Network layer

The network layer takes the fragments coming from the transport and data link layer and knowing what network actually looks like (how each node is connected to the network) this layer tries to divert the fragments accordingly such that each fragment reaches it's destination (after possibly passing through more then one network node), see figure 3.4.

There exists different strategies (for different network topologies) that determine how to handle all incoming fragments and where to send them. One particular algorithm, used in the simulator for the mesh network, will be presented in the next section 3.2.

Data link layer

This layer has the (not trivial) task to ensure that the data is still the same when it arrives at the layer's counter part. It is possible that the lower physical layer transfers the data not correctly and in that case the data link layer must be able to detect and possibly correct the errors.

Normally the data link layer (also called just link layer) uses data detection (and correction) codes to either sort out the packets that contain errors or to correct them.

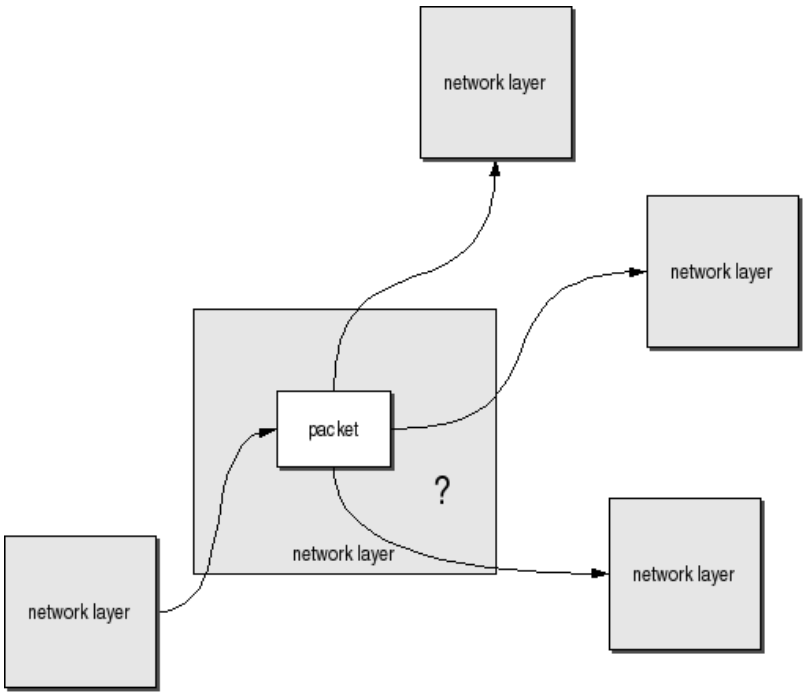

Figure 3.4.: The network layer knows about the topology of the network and has to decide whereto direct each incoming packet.

As this is all just a simulation, the physical layer does not create faulty data transfers. There exists however the possibility to program the physical layer to do so, which is of course not done so in SNS.

So the data link layer has in fact nothing to do. It just passes the data from one component to the other without changing anything.

Physical layer

The lowest layer does the actual transfer of the data between two hardware connected network locations. This layer represents the actual wires of a network connection.

In the simulator the maximal amount of data that can be transfered with one clock cycle is 128 bits. This results in 256 wires in the physical layer.

It would be possible to actually use physical models to simulate the 256 interconnection wires (and the possible errors), but that would go too far (and the SEMLA simulator does not have one either) and so the physical layer is like the data link layer empty and just passes on each received packet, without doing anything else with it.

3.1.1. Mesh network

The topology used by the SEMLA simulator is the two dimensional mesh network architecture. A mesh network contains n times n network nodes that are usually addressed like the elements within a matrix. In figure 3.5 you can see an example for an 3 by 3 mesh network. Each network node is connected to

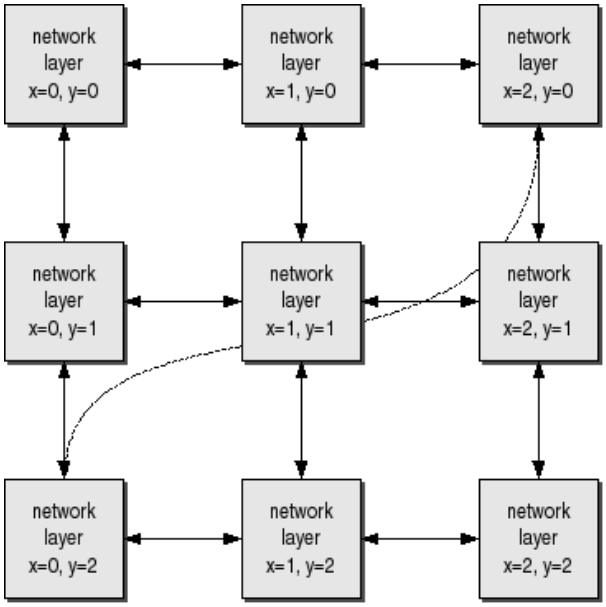

Figure 3.5.: The topology of an mesh network. Each node has to address values, the x (horizontal) and y (vertical) component. If a message has for example to travel from 0,2 to 2,0 it has to pass through other network nodes, before reaching its destination.

its vertical and horizontal next neighbors. Of course the network nodes on the borders do not have all four connections, as seen in the figure.

Each node is connected further on to a resource via a network interface. The resource can use the interface to send and receive network messages. A sent message might have to travel through more then one network node before it reaches its destination (as in figure 3.5).

This network topology is highly regular which is actually the reason why it is used.

Deflection routing

An algorithm is needed to decide how each incoming packet should get forwarded to which neighbor. In the trivial case an incoming packet is addressed to the node itself and can be delivered. In any other case the simulator uses the *deflection routing* algorithm to determine where a packet should go.

The deflection algorithm does not buffer packets that come from other nodes. It decides which packet should be forwarded to which node (that includes the node where it did come from) and sends the packet away within one clock cycle.

This makes the (hardware/software) implementation of this switching algorithm very small and that is the reason why it is used.

It also stresses the links between nodes more then necessary, because if a packet cannot be forwarded in the direction where it should go(because already another packet has to go there too), then it has to be sent in the wrong direction, possibly one step back and stays longer in the network than necessary (displayed in figure 3.6).

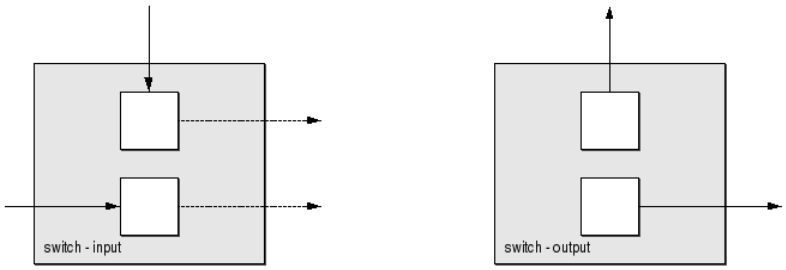

Figure 3.6.: Deflection routing: Each input packet gets diverted immediately to an output port, no internal buffers are used. If two ore more packets want to go in the same direction, just one wins, the other ones are getting send somewhere else, possibly also to where they did come from - right picture.

3.2. Simulation model

The implementation of the SEMLA network simulator makes use of a hardware modeling library called SystemC [Pan01]. This software library makes it possible to use a clocked synchronous simulation model and it allows the use of modules and channels as described in figure 3.7.

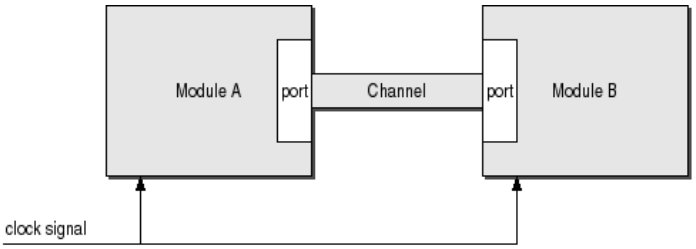

Figure 3.7.: SystemC model. In a clocked synchronous model each module (collection of logic and or other modules) uses the same clock signal. The modules in SystemC use ports and interfaces for module-to-module connection. A channel implements an interface and can then connect two or more modules.

The simulator uses a common base for all layers which defines the variable amount of input and output ports which then are used to connect the different layers and it defines also the used data type.

The physical layer has one input and on output port to transfer the data over the wires. The link layer has two input and one output ports (one pair for each direction, up-down and down-up) and also just passes the data along.

The network layer has four input ports for its neighbors and one for the transport layer and respective five outputs. It uses the deflection routing algorithm (described in the previous section 3.1.1).

To not cause an immediate packet loss given the case that the network switch gets five input packets and none of them is addressed to the local node, the network layer has an internal buffer to store packets (coming from the transport

layer) that cannot be handled at the moment.

The transport layer itself has no tasks to do in SEMLA. It just pases the data on to the next layer, but it serves also as the interface between an application and the network and it has therefore to buffer the packets in both directions.

By connecting all the modules together as in figure 3.8 we get a complete, ready to use mesh network.

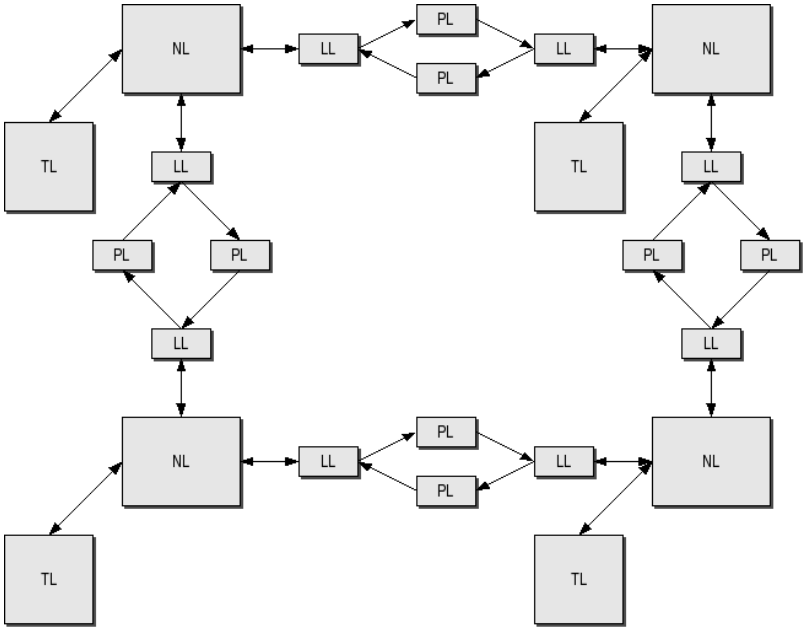

Figure 3.8.: The SEMLA model of a 2x2 mesh network, using the implementations of each layer: transport layer (TL), network layer (NL), data link layer (LL) and physical layer (PL).

Automatic network generation

To make the creation and connecting step more flexible the SEMLA simulator uses an automated mesh network generator. It takes the desired network size as an input and generates the whole network with all modules and connections.

The network generator will be reused later for creating all simulation components and connections not just for the network part but for the entire SNS.

3.3. Results

The concepts introduced by Thid's master thesis [Thi02] are flexible and easy to use. Each layer implementation can be easily exchanged, to use for example a different routing algorithm; even a different network topology can be implemented using the same framework. Finally with the automatic mesh network generator it is possibly to create mesh networks of different sizes and use them by just changing the actual network size parameter.

Alltough I will not make use of the possibilities to change the network topology or routing algorithm I still will make use of the presented network simulation

framework. Unfortunately the SEMLA code that I got was not very (re)usable so I had to rewrite the entire SEMLA simulator before I could continue.

The next step in building the SNS design is to combine Thormann's simulator (presented in chapter 2) whit the SEMLA simulation as in figure 3.9 and to extend the resulting simulation with more features to achieve in the end a more realistic and complete simulator.

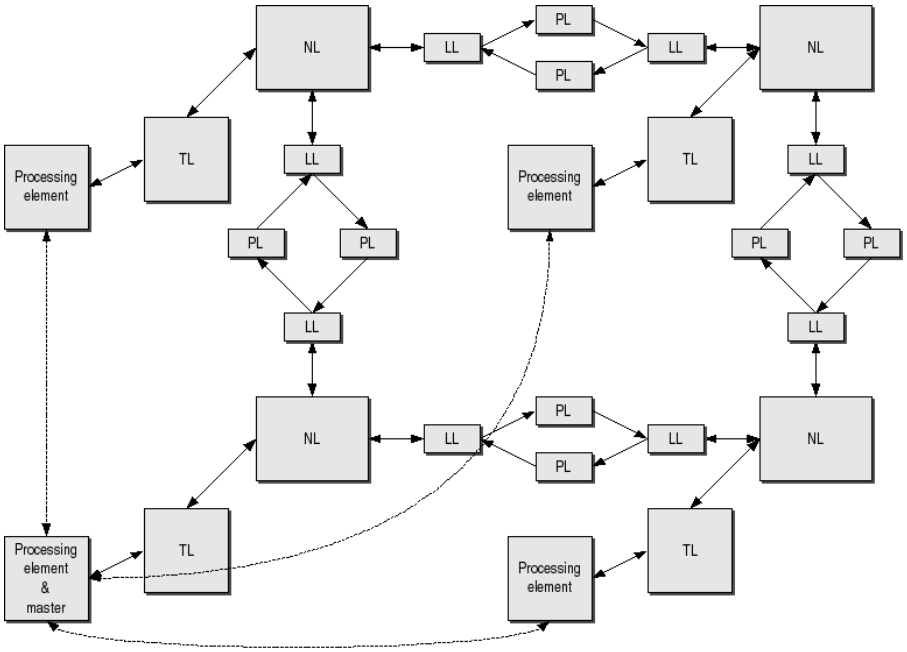

Figure 3.9.: A SEMLA mesh with additional processing element extension.

4. Empirical network power model

Most of the used concepts are already introduced. What remains, to conclude the related work part is the issue of how to calculate the network's power use.

The aim of my master thesis is to examine the energy consumption of different load balancing strategies and not to develop a network power model. I therefore will adapt the results from Sandro Penolazzi's master thesis [Pen05] to estimate the power usage of the network that is used by SNS.

Penolazzi basically tried to develop a power formula that depends on input activity, used technology and frequency and represents the real power usage as good as possible. For the comparison part he used Synopsys Power Compiler [Syn04] to get his reference power readings.

4.1. Switch power model

The power model developed in Penolazzi's thesis [Pen05] offers a mathematical formula to compute the power usage of a 2D mesh network switch (seen in figure 4.1) with deflection routing, as currently used in the NOSTRUM project [Homa].

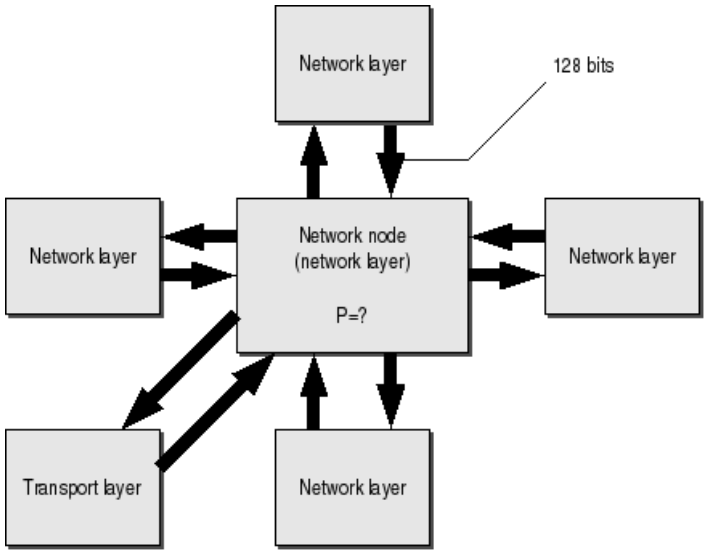

Figure 4.1.: A mesh network node's inputs and outputs. How to model a power formula?

How did he get there?

To be able to create a mathematical model of the power usage, first an adequate reference model has to be found. Obviously this cannot be the hardware

itself, although that is in fact the only real reference. Instead a VHDL model of the switch was used as input for a program called Synopsys Power Compiler, which is able to simulate the actual power usage behavior sufficiently accurate.

With the obtained reference values the development of an empirical power model could begin. The inputs for the power formula are the (in the NOSTRUM project) used settings (technology voltage and frequency) and the for each clock cycle different switch inputs. This makes it possible to get the power usage reading for each clock cycles depending on the actual load of the switch.

Initial experiments showed that the separation of static power (all input bits are zero and do not change between two clock cycles) and dynamic power (some input changes between two clock cycles) is possible and the first power model looked like this:

$$P = P_{static} + \frac{P_{dynamix,max} - P_{dynamic,min}}{bits_{tot}} \cdot bits_{switching}$$

Comparing this power results with the reference showed that this power model results in much lower power readings then the real ones.

Further tests showed then why: The state of the non switching bits seemed also to affect the used power. Switching half of the input bits and leaving the rest at 1 generates a lower power reading then doing the same but setting all non-changing bits to 0.

Considering this fact a new power model was designed that takes into account the different power usage when the non-switching bits are 0 or 1. This new formula still generated power results that are too low. But an inspection of the actual power graph showed that the power model followed the reference power reading quite well, regarding the shape, it was just to low.

With even more tests Penolazzi could determine that the distribution of the switching bits along the inputs have an influence on the used power, too. To cover this new fact a correction function is introduced which basically elevates the power a little if just a few bits are switching and it raises the power nearly by nothing if mostly all bits are switching. With that addition finally a power model was developed that is accurate within a few percentages compared to the reference power usage in all tested situations.

4.2. Results

The final power formula uses the information about all input bits (are they changing or not, which value do they have) to calculate the power. Alltough the accuracy of the formula is also very good (it is only off by a few percentages) there are two reasons why I cannot use it directly as it is for SNS:

- first the NOSTRUM network switch implementation is not the same as used in SNS
- secondly providing the information about each bit state for each clock cycle is too much of a simulation effort

Assuming that the different switch implementations are not too different I decided to use the power model anyway. But the problem of knowing all the bit states remained. I therefore asked Penolazzi for a little help and together we came up with a modified power formula, that just considers the case when an entire input switches or not (instead of watching over each single bit switch):

$$P = (7.55 + in_s \cdot 0.15 \cdot (4.764 \cdot (in_s \cdot 128)^{0.33} - 0.0645 \cdot in_s \cdot 128) + 0.064 \cdot in_s \cdot 128) \cdot f_{mod}$$

in_s is the count of how many inputs did switch regarding the last clock cycle and f_{mod} is the factor for different network clock frequency.

I did not talk about the power consumption that is caused by switching the connection wires between two network nodes yet, because I already presented that power formula. It is the common formula used to calculate the power of any CMOS logic(see section 2.1). I just used the same setting for the load capacitance and the voltage as mentioned in Penolazzi's master thesis [Pen05] and I set the switching parameter to 0.5. The result is this power formula, that calculates the power for all outgoing links from one network node to other nodes.

$$P = \alpha \cdot (C_L \cdot V_{dd}^2 \cdot f_{clk}) = C \cdot f_{clk} \cdot out_s \cdot 128$$

C is the constant that contains the capacitance C_L, the voltage V_{dd}^2 and α parameter. out_s is the count of how many outputs are switching.

For one component Penolazzi's thesis does not give a power model at all and that is the transport layer. In SNS the transport layer exists and does also do some work and therefore consumes power, too. But how to estimate that power usage?

After some advice from my supervisor I decided to reuse the power formula from the network layer. Instead of five inputs the transport layer has only one and this needed a further adjustment of the formula before it was ready to use.

Clearly this is just an estimation. Since I already used a different network switch than required and considering the complexity of the transport layer implementation, used in SNS I guessed at the end that the generated power readings can representatively be used as the networks power usage.

By adapting Penolazzi's empirical power model for the network and transport layer used in SNS and using his link configurations I now have the possibility to compute the power usage of the simulator's network and the actual development of the simulator can begin.

Part II.

Method

5. Work scenario

So far we have:

- network, for transporting data
- processing elements that consume computing time
- tasks that determine how much computing time gets used on a processing element
- different strategies for allocating a task to a processing element at runtime

With that we can simulate the execution of some tasks and measure the consumed energy. What we need more is to define a set of tasks, that we can use through different experiments, to determine the influence from changed parameters. E.g. investigate how a different allocation strategy affects the energy consumption, or witch configuration of the simulation allows all tasks to be completed in time. I call such a set of tasks a *work scenario*. Once defined it can be used to compare the results of different simulations.

Now let's take a look at what we need for a complete work scenario.

5.1. Task parameters

The simulator needs to know how to treat each single task from the work scenario. Therefore the scenario defines a set of parameters for each task and those parameters influence the behavior of a task during the simulation. So what is it, that needs to be influenced in a task's behavior? Let's take a look to what a task does in the simulation:

- consume network resources
- consume computing resources

To make use of the network resources a task sends a message to another task, let's call that other task it's communication partner. To generate always the same amount of traffic at the same time for different simulation runs, a task needs two traffic parameters: How *much* data gets *when* sent. So we have the first two parameters for the task: The size of one outgoing message and the time interval between two of those messages. To simplify the simulation let's say that the communication partner does not care about the content of the message. It is enough that the message gets received.

These two parameters ensure network communication from one task to its communication partner in that one direction. If we want the communication

partner to replay something, we have to assign some traffic parameters for the communication partner as well.

Just to clear things up, let me say that of course one task can have more then one communication partner. Each task gets a list: for each communication partner the size of the messages and the time intervals can be different[1].

Using computing resources is essentially described by how much *computing* steps a task wants to use and when the task has it's deadline. This implies actually that we are restricted to a real time system. But that is not the case. I just want to use the information about how much computing resources a task is using to get information about how the simulation did its job(e.g. if a lot of tasks miss there deadline the system might be overloaded...).

Then again we need to know about the needs of a task for the task-allocate algorithms, otherwise we have no basis on which we could decide how to balance the load of the system[2].

So back to the task's parameters. We now have the parameter to determine the maximal amount of the computing steps that a task is going to need and we need to determine a deadline for a task. That is actually tricky. We could use a fixed time, e.g the task has to be finished at 40 milliseconds simulation time. But that would restrict us to use for each task a start time and a deadline, without the possibility to create tasks dynamically (out from other tasks). So instead we still can give each task the possibility of a *start* time, but to determine the deadline we give each task an amount of *time* within the task has to be completed.

Now we have three new parameters for the task: a possible start time, the maximal amount of computing steps and the amount of time within the task has to do all its computing steps.

To make the work scenario a little bit more flexible (e.g. have a limited set of tasks with limited execution time, but still get an infinite long simulation) we can add to each task a possible *period*. After the task has completed we start it again, when the period time is passed. In that way we can create a scenario in which tasks are recreated as long as the simulation lasts and we can see how the system reacts during the different situations (in the case that tasks have different periods, of course).

With that we have completed the list of parameters considering the use of network and computing resources. What remains now is the following problem: We can define tasks that calculate for some time and send some data to some other tasks, but they do not care about others yet, which is of course not what we want. A brief look at for example a multimedia application tells us why. If we assume that one task decodes a video fragment and another task decodes an audio fragment then we need a third task that synchronizes those actions, such that on the outside audio and video appear synchronized and not as a video

[1] e.g. send 300 bytes after each 500 computing steps to task A and 200 bytes after each 1200 computing steps to task B

[2] That is not entirely true. We could also determine the load of the system be measuring the actual use of a processing elements(e.g. how much idle time it has), but nevertheless an algorithm cannot look into the future, so we still could not apply the same allocator algorithms, at least not the ones from subsection 2.2.1

stream with an apparently too late or too early arriving sound.

In other words each task must have the possibility to wait for other tasks and to tell other tasks when it is finished, like in figure 5.1. This means two more parameters are necessary. One parameter tells the task for whom it has to *wait*. That means we have for each task a list, which contains other tasks, that have to be finished, before this task can start working.

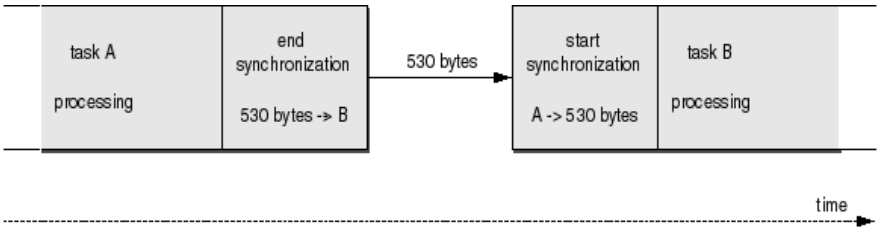

Figure 5.1.: As soon as task A finishes to process, it sends the synchronization message (consistent of 530 bytes in this example) to task B. Task B does start processing as soon as the synchronization message from A has arrived.

We can use a special marked network message to synchronize two tasks[3]. The task that has finished sends a special message to the waiting task. The waiting task must know from whom the message should come and which size it should have, so that the message can be identified. To assume that the size is know may not cover all situations (e.g. what if the synchronization message is the result of the task's computation and the result is a list with variable length...) but since the main purpose is to provide a synchronization mechanism between tasks, a limited size will do.

And of course at the end of each tasks life time the second parameter tells the task to whom it must *send* a synchronization message and how big it should be.

With those two parameters, the start and end synchronization list, we introduce dependencies between the tasks. This aspect opens a lot of possibilities for modeling a work scenario. Together with the parameter for the periodic re-start of a task we can now model any possible task situation (including those where tasks without deadline exits. They always can be split down into periodic partial tasks with deadlines)

Now we are nearly there. The last parameter gives a task the ability to create other tasks. Actually we need two parameters to make that possible. Obviously one parameter tells the task when to create a new task, but not just any other task. To extend the possibilities further we can give each task an identification (that we need anyway already to identify a communication partner). With that we can tell a task *when* to create *which* task.

To make the scenario more flexible we can also add a *type* parameter to the task properties, such that it is possible to use different task models during the

[3]This means that non-synchronization messages are marked as well. They are marked as non-synchronization messages

simulation or to allow to adjust the execution speed for a certain task type on each processing element.

This parameter is primary intended to inspire future work. Anyway one practical use of this parameter in the simulation will be discussed at the end of this chapter. For now we have the task parameter, visualized in figure 5.2.

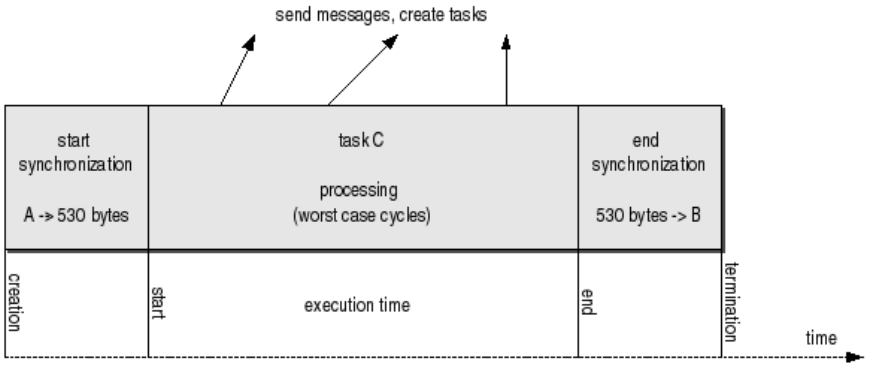

Figure 5.2.: A task's life cycle, defined by its parameters.

5.2. Types of task sets

Now that we have determined which task parameters we are going to use, let us play with them! In real applications, the creation and synchronization follows always a certain individual pattern. Some of those patterns can be put together to families and described by a model, other patterns are simply random (or have some random characteristics, caused due events from the outside, e.g. user inputs). What we need is to define some of those models for the use within our work scenario and within the possibilities of the above defined task parameters.

Pipeline model

A typical pattern for heavy data processing (multimedia encoding, decoding and picture processing) consists of a start task an end task and multiple (parallel and serial) stages in between (shown in figure 5.3).

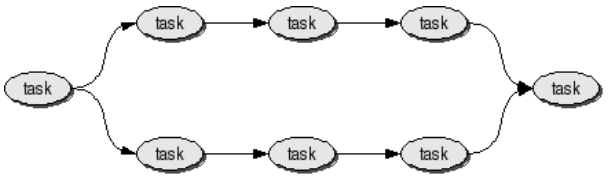

Figure 5.3.: The work scenario pipeline model.

We can realize this with our task model by using a start task that creates and synchronizes all tasks for the next stage. Each task within a stage creates and synchronizes its successor and the last stage synchronizes with an end task. Of course one of the tasks in the last stage has to create the end task.

And so we can define our first model, that represents different kinds of multimedia and heavy data processing applications. Since the model is similar to a pipeline we call it the pipeline model[4]. The synchronization data represents in this case the intermediate results that are passed between the stages. As important parameters for this model we have the width and the length of the pipeline model. And in giving just the start task a period, the pipeline can be repeated over and over again.

Star model

Since we covered now data processing lets take a look at a process that creates a task, waits for it's result then creates another task, waits for it's results and so on. Such a situation happens for example in a client-server application. On the server side the server task gets a client request and creates a task that processes the client's request. If another request comes in the server task creates yet another task to process this new request and so on. This behavior is displayed in (figure 5.4, top)

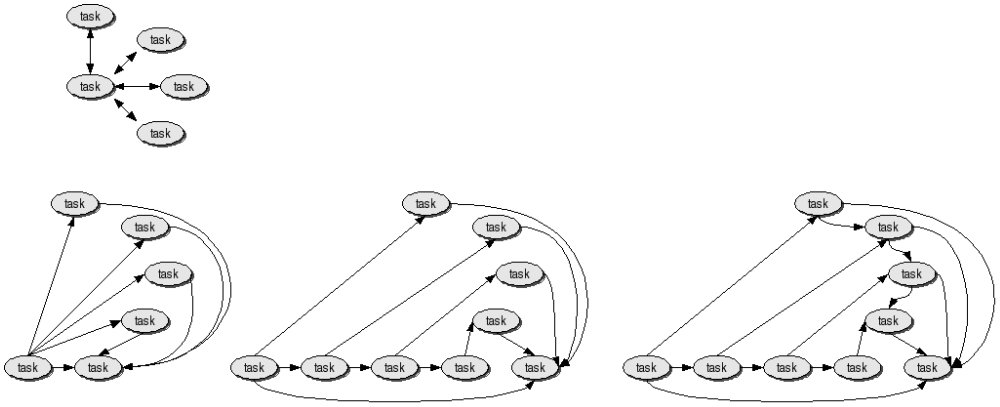

Figure 5.4.: Star models: The top model is the underlying star model, the bottom models are the real star models. From left to right type A, B and C

To be more flexible lets explore the possibilities of this model. In the simplest case (figure 5.4 type A) we have one central task that creates all surrounding tasks at once, sends them all a synchronizing message and they all start to consume computation resources at (theoretically) the same time. An end task waits till all tasks are finished and the model is completed. This model represents a massive parallel computation situation or any other situation where lot's of things have to be done simultaneously (like decoding all the MP3-ID-tags of a play list)

Type B is more complicated in its construction. Since we want it to be like a server task model we have one central task and several other tasks, that get created with a certain interval of time between the creations of the tasks. Still each created task has to send some synchronize message back to the center when it has finished and that represents a problem.

[4] Although it cannot represent a real pipeline in the sense that each time every stage processes data, it is still a processing pipeline regarding the different stages of a processing procedure.

We can only send synchronization messages between the end of one task and the beginning of another task. Once a task has started it cannot receive synchronization data anymore. So we have to split the one central task in lots of smaller central tasks, one additional for each generated client task. That makes the model bigger, but the principle remains the same (see 5.4). Again we have the end task that waits till all tasks are finished.

Type C is an extension of type B, in the sense that now the client tasks are synchronizing each other, too. That ensures a certain order of execution between the client tasks (see figure 5.4, the client tasks get executed one by one). This model covers situations where each computation step has to wait for the previous one and has to report to the central task (like when we want to parse just one MP3 file at a time).

Since all this models have the shape of a star (with the central task in the center and the client task all around it) I call this model the star model. The important parameters for this model are the amount of the client tasks and of course the (sub) type of the star.

Tree model

To cover all other behavior (and to not to have to many models) I construct a model, where one task randomly creates some other tasks and sends them synchronization messages. Each of the created tasks does the same and the next created one too. With that we get sort of a tree structure (see figure 5.5) and that is why I call this model the tree model.

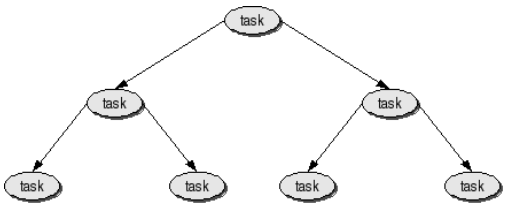

Figure 5.5.: The work scenario tree model.

It basically covers all behavior where tasks are spawned more or less randomly and it is not predictable how much and when tasks get created (like when reacting to user inputs or other IO events). The important prameters for this model is the depth of the tree and how much tasks each task should create.

Common parameters

All of those models have a start task with special parameters. Each start task gets a start time and a period time, so that the whole model starts at some time and repeats itself after completion (the task gets not created again before the period time is not over). All other tasks of the models do not get any period time, since the start task has it, and they do not get a start time either, because they are create by some other tasks.

5.3. Creation of a work scenario

Now that we have different models of how to chain up tasks with dependencies we have to create a whole bunch of tasks, that then are fed to the simulation and produce a lot of results. The question is, how many tasks should the scenario contain and how to combine the different models.

To determine how many tasks should be contained in a work scenario we must answer the question what effect the work scenario should have. Depending if we want to stress the whole system or just overload the network or the processing elements the scenario needs to be configured differently. I decided to determine the size of the scenario by experiments and to reach a point where as much as possible of the resources is in use without causing any overload. In other words, create a work scenario that causes no packet loss in the network and lets all processing elements operate at maximum speed.

The bottom line is, creating a work scenario by hand is simply too complicated. Maybe some small scenarios for special purpose can be written manually, but mostly the scenarios are too big to assign all task properties task by task, so an automated creation of the work scenario is needed.

One problem coming along with that decision is that we cannot combine the different task models with each other, at least not within the scope of this master thesis. Instead I gave the automated work scenario generator a whole set of parameters, to make the resulting task graph as much configurable as possible.

For each model parameter (like the depth of the tree model) it's possible to define a minimum and a maximum value. In addition to that it is possible to determine how many instances of each model should be generated (for example: create between 4 and 9 pipeline task models). But that its not all. To extend the configurability further and to make each task model instance different it is also possible to define min and max values for each task property for each task model.

In that way we can get a work scenario with really different task parameters and models, as you can see in figure 5.6.

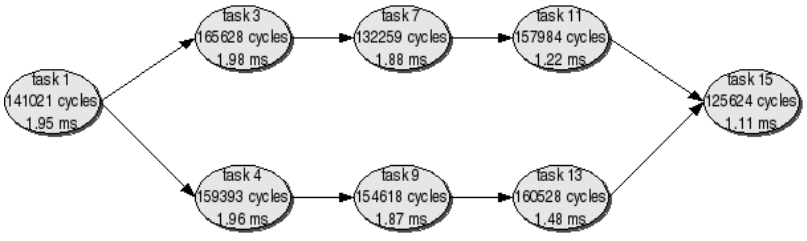

Figure 5.6.: A work scenario pipeline model example.

Still, the use of the synchronization data as only source for network traffic does not really generate a lot of network load(except if we use a lot of tasks, like a hundred or more per processing element at a time). To get more traffic and to make the scenario more realistic it is desirable that a task sends data to other tasks during its processing time, not just at the end.

To realize that feature, the work scenario generator tries to estimate the time during which a task is alive using the start time and execution time parameter and the type of the underlying task model.

With that information at hand the generator checks for each task the possible communication partners and adds them to the communication parameter. Of course all of those parameters can be randomized as well such that we get different traffic patterns.

With that addition the above task model looks like in figure 5.7.

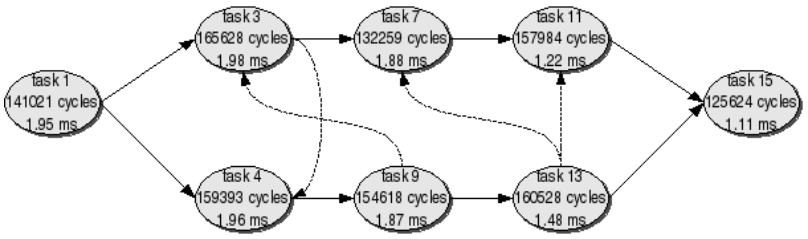

Figure 5.7.: A work scenario pipeline model example with extra communication.

5.4. Memory task

By now tasks can make use of computing time (processing resources), they can send and receive network messages, they can be synchronized between each other and they can create each other. Considering the network traffic there is another effect, caused by a task. That is when a task gets created the executable code for it has first to come from somewhere and that may cause additional network traffic, that has to be considered in the simulation.

Lets take a closer look at this:

- One of the resources has to simulate the access to of-chip memory and can therefore not execute any tasks. Lets call this resource the memory controller.
- Before a task can be created, its executable code has to be obtained first[5].
- This executable code gets transfered from the memory controller to the processing element, where the task gets executed.
- At the time when all the executable code arrives the task can start.

Why do I explain this in such great detail anyway? Well since my simulation does not have any special purpose resources technically I cannot simulate a situation like the above one (I cannot use a memory controller). Instead I can use a trick to simulate the loading of a task. That is, we use an additional type of tasks, that is allowed only to be executed on one of the processing elements

[5]Of course we do not have real executable code, we just send some random bytes, to simulate the use of network resources.

(our memory controller), all other tasks of course are then not allowed to use this resource any more. I call the additional task type the memory task.

Now for each task we create a memory task first, as displayed in figure 5.8. The memory task does not use any processing resource (has no computing cycles to consume) it just sends synchronization data to the task, that has to be created. With that action we simulate the actual loading of the task.

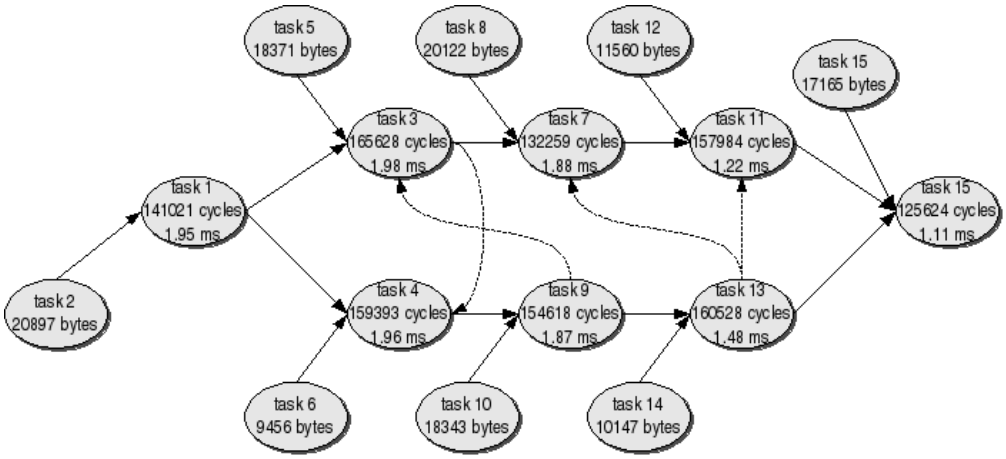

Figure 5.8.: A work scenario pipeline model example with extra communication and memory task emulation.

Of course this procedure is not very accurate. No real memory access is done and no local memory, that a processing element may have is considered. Since we do create some additional network traffic at the start of each task we can still say that the simulated loading of a task is sufficiently accurate. The most important result of the simulation is in fact the energy consumption of the whole chip.

Considering that, another problem presents itself. The processing element, that simulates the of-chip memory access does not really consume an accurate amount of energy. It just goes from suspend power to minimal power and back, each time a task gets created. Maybe a real input-output would draw a more constant power level, but without specialized resource, this cannot be simulated and so this issue remains unsolved[6]

[6]It is possible to assign each memory task some amount of computing steps, that have to be done, and in that way elevate the power level a bit, but that would also limit the amount of memory tasks per time.

6. Simulation method

My master thesis is mostly based on the master thesis of Thormann [Tho05]. His simulation makes use of a network model (transport and energy consumption) via some formulas based on such heuristic network models where the network actually does not exist as such. In other words his simulation does not include the actual transporting of data through a network. Between a send and a read of a network message stands the one function that calculates the approximated delay of the communication. In the same way his simulation deals with the computing resources.

I am going to change exactly those tow aspects[1]. Why? Well, for once it is the logical step in extending the work and on the other hand using a more detailed model promises to produce more accurate energy readings and more possibilities in configuring the simulation.

But that is not all. A more complex model also opens up the possibility to integrate further solutions into the model, that have to be solved in a "real" system eventually.

The use of a global master allocation algorithm for example requires some communication between the global and the local(on each processing element) instance of the algorithm. This communication needs a protocol, which has to be elaborated at some point, too. In Thormann's simulation this whole issue is solved by simply ignoring it and using some magic instead (from the system's point of view).

Some of the refinements that I made and why you can find in the next sections. The detailed description of all extensions (including those not relevant to the simulation method per se) you can find in the next part of my master thesis (chapter 9 till chapter 13).

6.1. Simulation duration

My first shot was to make all simulations use the same (time) length. If the duration of the energy graphs is the same, I can compare them easily with one and another and see what the differences are, or so I thought.

But what happens if we consider only one period (of the tasks) and therefore make it possible for a simulation to finish earlier then another one. In that case the faster simulation has some time, where the simulation does not do anything, but consumes (idle) power and energy, as shown in figure 6.1.

What to do in such a case? The faster simulation has the possible advantage to not be charged with the extra energy due to the extra (idle) time. Considering

[1] I am going to change much more, but mainly those two aspects. For example I already changed the general task model, by introducing the work scenario in 5 and the extended task parameters in 5.1

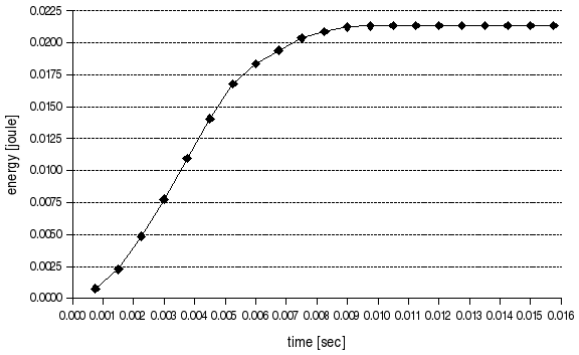

Figure 6.1.: An energy readings example. The simulation time was too long and in the last part the system does not do any work. Even if not visible in the graph, the energy usage still grows.

an energy per task cost point of view it makes actually sense not to count the extra energy, because in that case we just want to know how fast a simulation can execute all tasks. At the completion of the last task the simulation actually ends. In fact, that way we do not even have an equal simulation duration and we do not have extra idle time at the end of the simulation.

How do we compare then the energy reading of simulation with different simulation times? We do not. We do not compare the energy values directly, because if one simulation has a longer simulation time, then the extra time adds also extra (idle) energy, that another simulation does not have and that would change the results.

Instead we use the consumed energy to express a cost factor of the simulation. For example: To solve the work scenario using a particular algorithm the simulation uses a certain amount of time x and energy y. The cost factor is the consumed energy divided by the consumed time($\frac{y}{x}$). And the we compare the simulations by using the cost factor and not the energy alone.

With that decision I do get the following simulation duration policy:

- each simulation has a fixed time amount at its disposal
- in the case of an one-period simulation the simulation end gets recorded and accounted for in the evaluation

6.2. Simulation components

As compared to Thormann's simulation, the simulation of the network and processing resources is refined in SNS in order to get more accurate simulation results.

For the network part SNS mainly makes use of the SEMLA simulation (described in 3). Which means that we have now a network model composed of four layers at our disposal. In each layer we can use a model with as many details as we want.

That means for the lowest two levels, the model of the physical wires and the above error correction and detection layer, that they do not exist in SNS, not entirely at least.

Since this thesis does not include any physical error phenomenas (bit changes on network links) our physical layer does nothing but pass on the received data and the next higher level, the data link layer, has also nothing to do, since the data transfer is optimal and no errors occur.

Since the SEMLA simulation did not work when I got the corresponding code I created my own versions of the rest of the layers (the network switch and the transport layer). As for using the network as a transport medium, things where getting interesting. The SEMLA simulator stops at the transport layer. The problem is or rather was to make the next step in creating a more detailed simulation, as shown in figure 6.2.

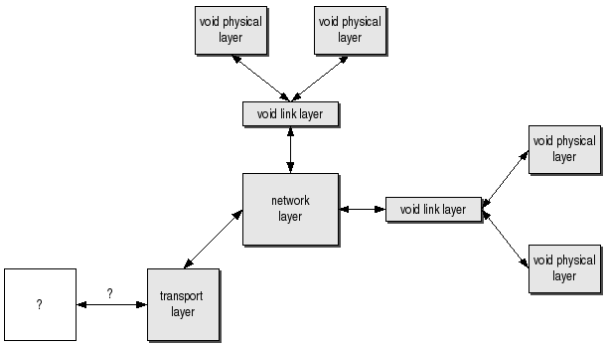

Figure 6.2.: Coming from a SEMLA mesh from section 3.2, how should we proceed?

In other words we need some way to introduce the idea of processing resources and the possibility to make use of network resources from within a task.

6.2.1. Processing element

Which properties does a processing element have? Well, for a start it offers the possibility to execute a task's code. In Thormann's simulation the processing element also has an *utilization* property, that indicates how many tasks are executed on a particular processing element (and how much use of the processing element resources each task makes).

This method of simulation ignores many facts. For example, if one task has to wait for a network message it cannot be processed at the moment, but later on it needs more processing resources as it actually has at its disposal. This leads to the situation where each task does not use its computing resource at a constant rate(figure 6.3-left), but makes sometimes more or less use of it, as seen in figure 6.3-right.

If too many tasks make too much use of the processing element within one time interval they might try to use more computing resources than the processing element can actual offer. That means that some of the tasks are not executed and get delayed(figure 6.4).

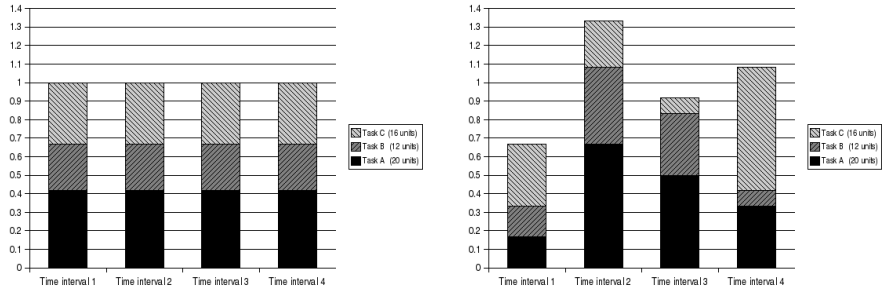

Figure 6.3.: Constant and possible actual processing resource use

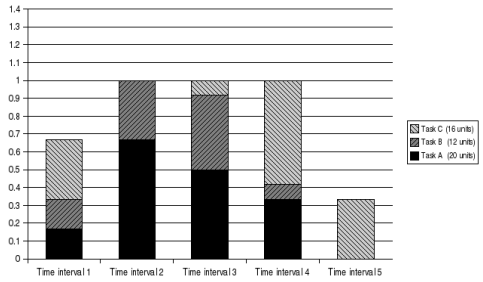

Figure 6.4.: Delay example(task C gets delayed), for using a processing resource(assuming the same task requirements as in figure 6.3-right)

To model such a behavior a processing element needs to execute tasks, step by step. With that ability we can then know at any time how much of its work each task has done (the amount of steps is limited by the tasks *worst case cycles* parameter, see section 5.1) and we also know for each time interval how many steps each task did consume.

But that is not everything what our processing element must be able to do. Another important aspect is the connection to the network resource. We can access the read and write functionality by accessing the top layer of the network.

One question remains: How do we know that the top network layer has data for us? We could try to read at every possible time, but that is very inefficient (and has also some other issues). Or we could give the processing element the possibility to be notified. We extend the processing element with an interrupt mechanism, like in figure 6.5.

Earlier in section 5.4 I spoke about different task types and how it should be possible to forbid the execution of a certain task type on certain processing elements. For that purpose each processing element gets an execution factor for each task type. Meaning the execution of a task takes longer or shorter on different processing elements, depending on the type of the task.

This property of a processing element can be seen as the behavior of different types of processing elements. For example between a processing element that supports integer operations and a processing element that supports floating point operations, a task that uses floating point commands might be much faster on the processing element that supports them.

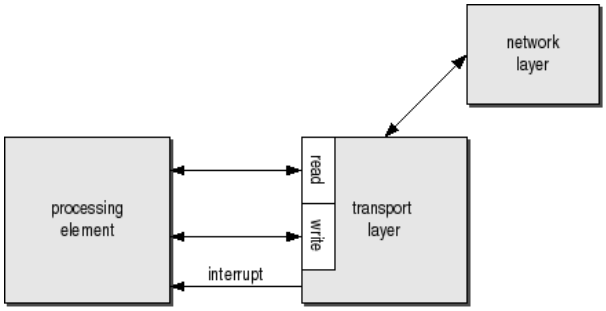

Figure 6.5.: Resource's read and write access and interrupt signal.

In the context of this master thesis the only use of this property is to reserv exactly one processing element for the execution of exactly one type of tasks (the memory tasks). All other tasks can be executed on all other processing elements at the same speed. Corresponding to that, the processing elements in the simulation are called *general purpose processing elements*.

6.2.2. Operating system

From time to time I spoke about how processing elements or tasks send messages over the network to each other. Clearly the messages are nothing more then random data and sometimes they do not even mean anything(for example in section 5.1 - start and end synchronization list). But someone hast to do the actual talking. Tasks cannot do that by them selfs and processing elements do not do any actual talking at all.

As you can guess from the title of this section, such services as described above have to be provided by the operating system. Therefore I give each processing element (more specific each general purpose processing element) an operating system that is capable of providing some services that have yet to be defined.

One type of service we already know and that are the services for the tasks, such as sending and reading data to and from the network. But tasks are not the only clients for the operating system.

Another group of clients are the allocation algorithms. Technically they are a part of the operating system, but since we are using different algorithms during the experiments I had to separate them from the operating system.

Then there is a third group that uses the operating system's services and I have not mentioned them, yet. I speak about the per processing element local scheduling algorithms. They are also part of the operating system (in the same way as the allocation algorithms), but to make it possible to experiment with different ones (although that is not part of this master thesis) we have to offer them separate operating system services as well.

So, in general the operating system in my simulation does nothing else then provide certain services. Because of that I usually speak of the operating system service that operates somehow like a task on each general processing element (you can read more about all that in the next part of the master thesis).

Services for a task

The most important services for a task is the possibility to send and read data from the network. And with that comes along the request to know if any data is available for a task or not.

Another important issue for a task is the creation of another task and the possibility to wait, until the other task is created (successfully or not created at all). With that comes along the ability of the operating system to send messages (let's call them events) to a task.

For example such a message could tell if the task creation request was completed successfully or not. That opens also the possibility for a task to make the request for waiting until network data is available for it.

All in all, in SNS the operating system service can offer the following to a task:

- Send and receive network data
- Check if and how much network data is available for the task
- Create a new task
- Receive events from the operating system
- Wait for the completion of the create-task request
- Wait till network data is available for the task
- Terminate itself

Services for the allocator and scheduler

The needs of an allocator or scheduler strongly depend on the actual used algorithm. For example usually a scheduler does not communicate with other schedulers or an allocator does not need to know the current power utilization of the network.

However, through experience and the examples from other simulations I tried to put together a set of useful operating system services for the allocator and scheduler operating system extensions..

Like all of the master based allocator algorithms mentioned in Thormann's master thesis [Tho05] an allocator needs the possibility to send and receive network messages. This requirement is actually a bit tricky to handle. There is the access to read and write functions, that is not the problem.

However there is the need for an another mechanism, which actually notifies the allocator (or scheduler) about incoming network data. This is not a problem either, but there is the question about how to realize network communication involving timeouts.

The problem hereby is to give an allocation algorithm some sort of call-me-at-some-time possibility, so that the algorithm can make checks for timeouts. I did choose to not add the timeout handling to the responsibility of the operating system, because not every algorithm might use it and because not every algorithm might use it the same way.

To solve this timeout problem, the operating system provides means to identify each message by an unique identification number and it allows the allocator (or scheduler) to ask for a possibility to resend messages on a certain point in time. With those two services, it is possible for any algorithm to resend messages, when the first sending of the message was not successful.

Of course an algorithm does not necessarily have to make use of this extra mechanism, but when someone tries to stress the network and provokes the loss of messages, this is a useful feature[2].

Besides the network service the operating system has to allow the allocator (and scheduler) to manipulate the speed of the processing element.

Why is that? Mostly the allocator will be in the position to make use of this service, because when an allocator places a task on a processing element to be executed (or is informed that the task is finished) it might want to adjust the processing speed of the processing element to conserve energy.

In some cases the scheduler might want to do the same, for example based on the fact that a task finished earlier then expected and freed with that some processing resources.

With that service we came across the most important service that the operating system has to provide to the allocator and that is the service for creating a task (on the local processing element). Then we have some other minor functionality, like check if a given task is executed on the local processing element or the ability to get to know the address of the local network node and so on.

Regarding the scheduler I have to say that I do not actually use any operating system service from within any scheduler algorithm, used in my experiments, because in my master thesis the scheduler algorithms do not change the energy consumption by manipulating the speed of the processing element and they do not communicate which each other either.

At the end we have the following operating system services for an allocator or scheduler:

- Send and receive network messages
- Make use of a per packet identification and time call-back service
- Change the speed of the local processing element
- Check if a given task is executed on the local processing element
- Ask the network address of the local network node

In case of an allocator the operating system offers two more services:

- Advice the operating system to execute a task on the local processing element
- Advice the operating system about the success or failure of a task-creation request from a given task

[2] And it helped me a lot during the developing of SNS.

6.3. Power and energy measurement

In this section I am going to say a few words about how the power demand of the network and the processing elements are measured in SNS and how the energy graphs are constructed.

To collect data the simulator uses an event system. That means that whenever somewhere an object needs to tell something (does not matter to whom) it wraps the data into a corresponding event and gives it to a central unit.

This central unit checks who wants to have the data and gives it to them. The consumers of the events are actually report generators. They collect data, put the entries together in a table and write them to a file.

Each line of the generated power and energy tables corresponds to a certain simulation time and each value is the new power or energy reading that was registered at that time.

Presumably the power reading remains constant between two changes and the energy graph is simply the integration of the power graph over time, e.g. old power value by time interval from since the old power value was measured.

So far that is all nice and if we have enough power readings we get some more or less nice looking, smooth graphs to show. Unfortunately, the tool, which I use to produce the graphs, does not support some 10000 or more entries in a table. Therefore I had to use a different method to display my results and that is why I wrote this section.

The graphs that you will see later on, are all based on the same creation process:

1. Register all power changes
2. Divide the simulation time into x (for example 500) equal intervals
3. Generate the average over each interval and write that value into the result table
4. Optionally create for each table entry two entries, to mark the begin end the end of the time interval

With that I do get graphs where you can see the average of the power/energy values that where registered for each time interval. You can see an example of how that looks like in figure 6.6, where I used two intervals: the first average value is $\frac{3 \cdot 2 + 6 \cdot 3}{5}$ and the second one is $\frac{6 \cdot 1 + 2 \cdot 2 + 4 \cdot 2}{5}$.

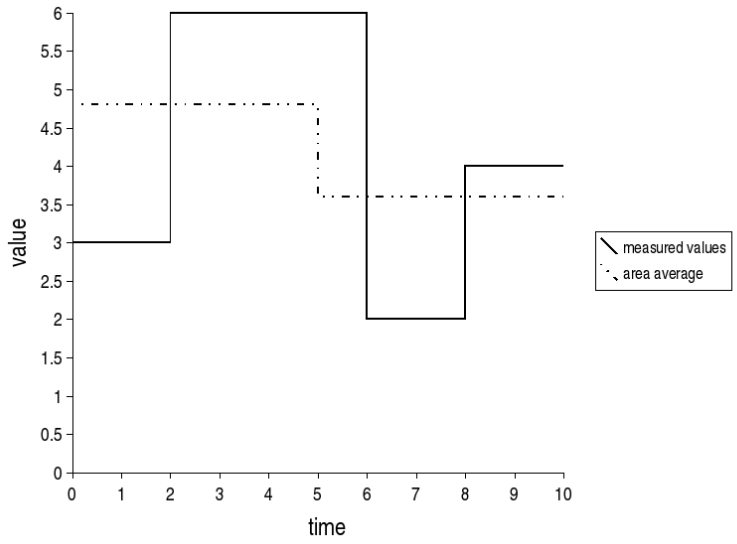

Figure 6.6.: Average over intervals from the measured values(1-5, 5-10).

7. Neighbor Allocation

All allocation algorithms presented in section 2.2 from [Tho05] (described in chapter 2) are based on a central component (the master) which makes all the decisions about which processing element should handle which task.

All of those different masters have the same services to offer (at least regarding SNS):

- Assign a task, that has to be executed, to a processing element, which can handle the extra load.
- Be able to tell which network address a task currently has if the task currently exists.

Those services result in considerable extra network traffic and have one point in the network that handles all requests, meaning they generate a lot of local traffic for that particular (master) network node.

You see where this leads to. If the network size is small this might be working fine, but with increasing network size the amount of traffic for the master node increases as well. The traffic amount can grow until it reaches the maximum of the network node's capacity and then we get delayed responses for the masters services. If things get worse the network traffic rises too high and the master services might break down and the whole system may become unstable.

An allocator system with a central component is therefore not really scalable. With this in mind I tried to design an allocation algorithm which would work without a central component and which would eliminate with that the bottleneck problem.

7.1. Decentralized allocation

The main task of an allocator is of course to allocate processing resources for a given task. Doing that is easy[1], if we know all about the processing resources currently in use.

If we do not have a central node which knows about the current use of all processing resources, then there are two options for obtaining that information:

1. Each processing element tells[2] all others about any changes about it's load.

[1] Obviously just doing it is not hard, but optimizing the process for some parameters, like the energy consumption, is the tricky part.

[2] To go into some details here, let me say, that of course it is not the processing element that does the talking, but some operating system service, running on the processing element.

2. At the time we need the information, we ask all processing elements about it.

Considering the first option we can get again the situation that is caused by increasing network size. If too many processing elements need to be notified, we risk to overload the processing elements' network node and this could cause some data loss or inconsistency. In this case that means the information about any processing element's utilization could be incorrect and tasks could be assigned to processing elements, which cannot handle them.

The second option hits the same problem, only from the other end. If we overload the network node of the processing element by sending all the request the answers might not arrive at all, arrive too late or in the wrong order.

Apart from that, the quality of both solutions depends highly on the current network load. If the network traffic is high, the packets do need longer to reach there destinations causing some (considerable) delay in the response of the allocator's services.

This argument and the fact that both options still have the bottleneck problem made me think of another solution.

There is a third option and that is to simply not know about the current status of all processing elements. We just make the decision where to put a task without that information.

So if a request for a task creation happens, the allocator algorithm tries first to create the task locally, if that does not work (because the local processing element is too occupied, to handle the new task as well) the algorithm simply forwards the request to one of it's neighbors and let them worry about the problem.

Because of this step I call this type of allocation algorithm the *Neighbor Allocation*.

Okay, this step to forward the request works fine if we do it only once. If the next Neighbor Allocator cannot handle the request either, it has to forward it again, but of course not back to where it came from. And that thought leads us to how allocation of a task with the Neighbor Allocation actually works.

For each task allocation request do the following (an example of those rules can be seen in figure 7.1):

1. Try to allocate the task local. If the resources are sufficient, we are done.

2. If we cannot handle the request ourselves add the own address to the request.

3. Try to select one of our neighbors for forwarding the request that is not contained in the request address list.

4. If no neighbor fits that requirement, select any other processing element, that is not contained in the request list, and forward the request.

5. If that fails too, the system cannot handle the task at this time. The task's creation fails.

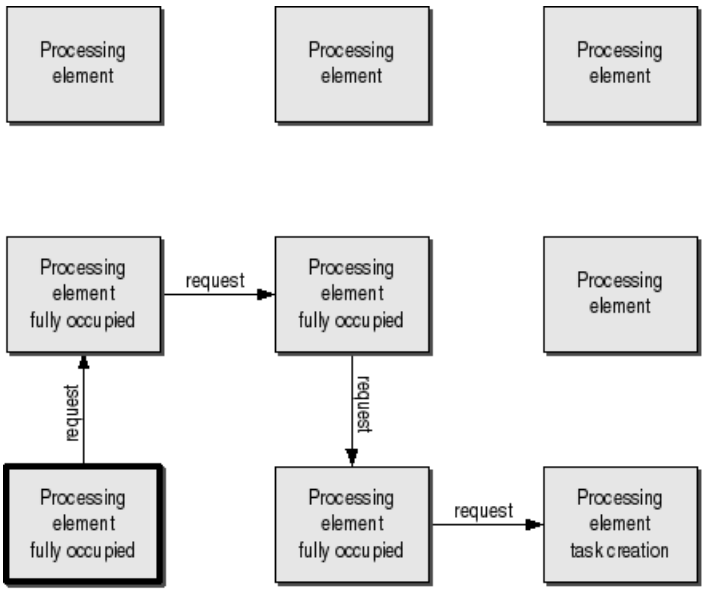

Figure 7.1.: A processing element gets asked to create a task, since its capacity is fully in use the request gets passed on, until it reaches a processing element with enough free resources.

7.2. Decentralized task address management

If a task wants to send a message to another task, someone has to figure out which destination the message has.

If we use a system with a central allocation algorithm, that central component can also tell us the addresses of all tasks. So if we want to know the address of a task we just ask the master for it.

Since with the Neighbor Allocation we do not have any central component anymore, we have to figure out another way to get the address of a task. Again, we have the same two options as above. Meaning we can either tell every one every time a task is created or terminated and so every one knows all the time all addresses, or if someone needs a task address it asks all others about it. Either way the traffic for such an action increases proportionally with the size of the network. But that is exactly what we want to avoid with the decentralized Neighbor Allocation.

Unfortunately in this case I could not think of a third option, that would not make use of a centralized component and would not create a bottleneck situation.

So lets take a closer look at the two option we have and which one is best to use.

In the first case we have to notify each other allocator about a task twice. That comes to $2 \cdot t \cdot (n-1)$ requests during one simulation. t is the number of *tasks* that are executed during the simulation time and n is the *number of the nodes* in the network.

In the second case each time that we have to know an address we have to ask all others about it. Each processing element asks only once for the address(the

second time, we already know the address). That results in $(n-1) \cdot \sum^{t} m_t$. m_t is the *number of nodes* that asked for the address of task t during the simulation. If a tasks gets terminated all processing elements that know about it have to be notified, so the number of requests becomes $(n-1) \cdot \sum^{t} m_t + \sum^{t} m_t$. The worst case is $(n-1) \cdot (n-1) \cdot t + (n-1) \cdot t$ and the best case is 0.

In my simulations I am going to try to use as much as possible of the available resources. It is therefore possible to assume that the number of request in the second case will be more likely near the worst case and not near zero. Therefore I make use of the first option.

That means the task address management for the Neighbor allocation works as follows:

1. If a tasks gets created (locally) all others nodes are notified about the tasks address.

2. If a tasks gets terminated all other nodes are notified about that, too.

In that way each processing element knows all the time if a task exists and if it does, where to find it.

Still, the fact that one processing element has to send a network message to all others at once, generates a lot of local traffic and could overload the network node of the processing element.

There exists a way to evade this situation. The idea is to split the notifying of all other processing elements from one big step into lots of smaller steps and limiting by that the increasing amount of local traffic.

Let's look at it step by step. First our processing element discovers that it must send a message to all other processing elements and so it sends the message to all its neighbors (horizontally and vertically). In the second step the vertical neighbors just forward the message in the opposite (vertical) direction from what the message came from. The horizontal neighbors do the same, but in the horizontal direction. But they also forward the message to their vertical neighbors (which then in the next step do the same as the other vertical neighbors). This steps are repeated until in each direction the end of the mesh is reached and so we reached all processing elements, without a local traffic amount (per node) that is proportional to the network size.

The process can be seen in an 5 by 5 network example in figure 7.2.

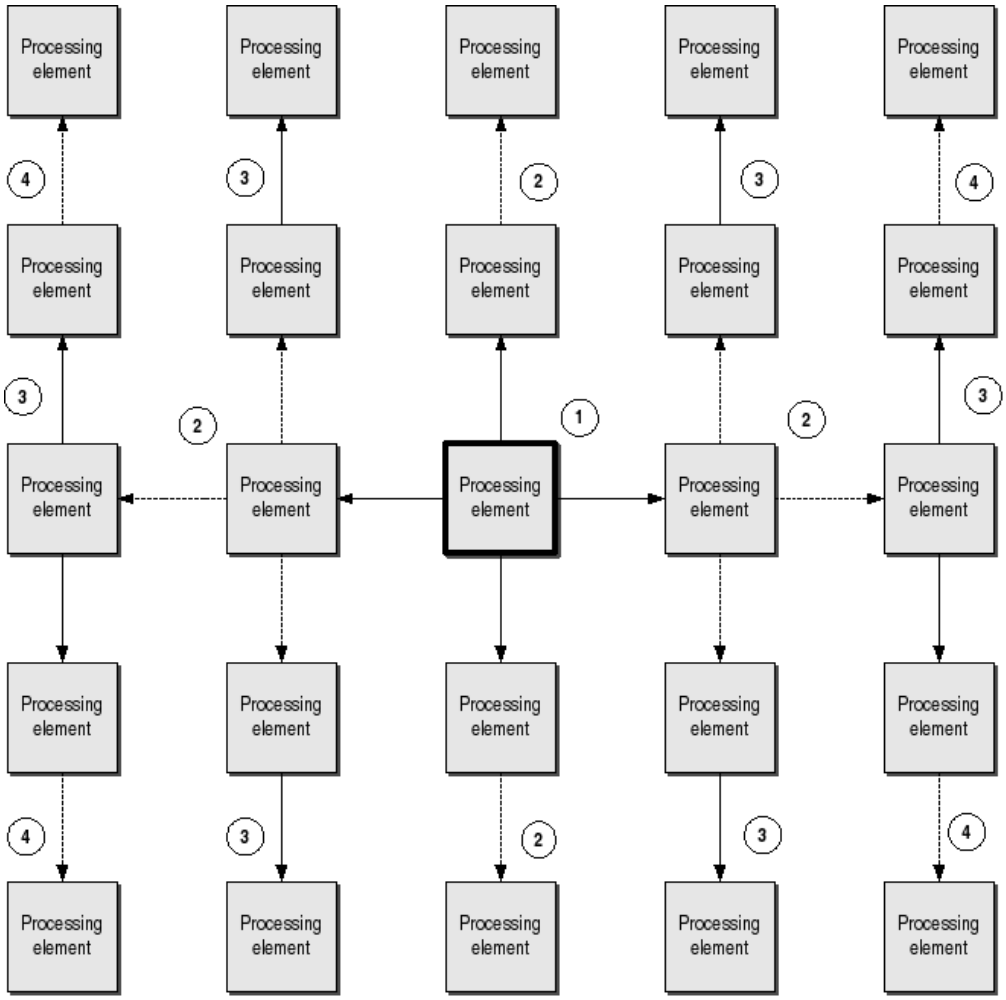

Figure 7.2.: If a task gets created or deleted, this information is spread across the whole network. In this example the origin of the information is the processing element in the center and the numbers represent the (time) steps by which the information is sended.

8. Expectations

Why discuss expectations regarding SNS at this point?

All the previous chapters of part II have introduced new concepts, used in the simulation. Now its time to discuss the expected effects of them on the results.

For once the simulator gets a lot bigger then it is supposed to be. Hand in hand with that comes the problem of errors in the implementation. I invested about 5 months into writing and optimizing this software, but still if you try a configuration that was not intended, the simulator will probably react badly or even crash. Especially if you try to stress the system, unpredictable behavior will be the result.

Since I am using SystemC [Pan01] for the simulation, there is a certain time limitation to the experiments. Elevating the frequency of the network past the 1GHz mark, slows the simulator down a lot. The size of the network can make bigger simulation infeasible as well.

In other words, the actual configuration for an experiment has much influence on the outcome regarding to whenever we even get useful results within acceptable time.

To check on that I added several *control*-reports, just to see what happens during the simulation. For example one of those reports tells if during the simulation network messages where lost.

I will mention some of those extra infos in the results, most of them however I will not show; it might just be too much and more confusing then beneficial.

Compared to the previous simulations (described in part I) I introduced important new concepts and they all will improve the quality of the simulation in a different way.

Work scenario expectations

The concept of the work scenario(chapter 5) allows the definition of tasks (use of communication and computation resource) and the interaction between them (one task waits for another, one task creates another one). It also adds the important possibility to create big work scenario (with thousands of tasks), store them in a file and reuse them for several experiments.

This makes the actual experimenting a lot easier and adds realism to the simulation.

It is not the case that a work scenario can recreate the situation of lets say for example 10 seconds in a multimedia mobile phone, but it can at least create a lot of activity, that might come pretty close to a "real" sitation. How close, I cannot say, since I did not have the chance to analyze such a "real" environment and rebuild it with SNS.

The improvements by using the work scenario are therefore mainly in auto-

matically creating task scenarios of variable size, that can be reused for different experiments.

It would be possible to just create very different scenarios and see how the different algorithms can handle the different situations, but initial tests showed that it does not make that much of a difference what actually happens, as long as lots of it happens. The fact of how much a resource is used seems to be more important then the how the resource is used, as long as limits of the resources are not breached.

Neighbor Allocation expectations

This new method of allocation (and task address management) system should come in handy as the network size increases. I might not be able to show that in my results, because first I try to adjust the configuration of an experiment such that nothing goes wrong (no loss of network messages and such) and secondly the point where the master based allocation algorithm gets into serious troubles might be somewhere (beyond 10 by 10 network size) where I cannot do any simulations anymore.

Anyway, the important thing for the new allocation algorithm is that it is not necessary to hold back one processing element regarding the execution of tasks and with that the system has increased capacities at its disposal, compared to the master-based allocator.

In the end this fact should result in use of less time for the same work scenario then the master based allocator experiments would use.

New simulation method expectations

As for the rest of the new concepts I expected that they make it easier to use the simulator and also easier to interpret and present the results[1]. I have in mind the format of the power and energy readings by means of time tables. I use the old but simple *comma separated values*(CSV) file format to store those tables and with that anyone can perform a new simulation and with help of any office-suite create a value-time graph to inspect the results.

Considering how highly configurable the simulator is, it will be difficult to pick the right experiments to present. I will try to stick to the actual effects of the different allocation strategies. However as I tried to configure the constants for the power use of the processing elements I did not come around as to notice that with some usual power usages of like 1 or more watts per processing element, the power used by the network became irrelevant(the whole 3 by 3 network used less power then one processing element). Therefore I will try to mention in the results how the situation does look like for the situations where the processing elements consume about the same amount of power as the network and the situations where they consume much more or much less.

Seeing how much things play a role for the energy consumption of a SoC with a NoC I am not sure how much sense it makes to just look a different allocation strategies, but the experiments will probably show that, too.

[1] Of course SNS contains implementation details that should make the whole simulation more realistic, too.

Part III.

Actual work

9. SEMLA extensions

The SEMLA construct as described in chapter 3 defines the four network layers and comes with a mesh network as an example how to make use of the layers. Further it offers a basic deflection routing network switch (network layer) and a channel based transport layer.

The source code that I did get in the beginning was not working very well, so I decided to reimplement this part by keeping the idea of the network layers.

I started with the base class *Layer* which provides the variable amount of input and output ports for all other layers. But which kind of data type to use?

Since mostly all kinds of networks can make use of a header and data differentiation I choose to use a container (*ProtocolDataUnit*) for both a header and a data object to be the data type for the input and output ports. The important thing regarding the header and data object is that they can be converted to and from a byte array, such that it is possible to get the whole network message as bytes.

I kept the implementation open for any network architecture, but since the 2D mesh architecture seems to be the most common and useful network SNS too uses a packet switched mesh network (see NOSTRUM [ZLNJ05]).

Next I constructed all the other layers on top of the base classes, implemented the two "empty" physical and data link layers and used the deflection routing algorithm based network switch in SNS as well.

Network layer

Special for the network layer is actually how I defined the header of each network packet.

To know where a packet should go I defined an address object for the mesh network that contains two bytes: the first is the horizontal count of the node x and the second one is the vertical count of the node y whereby the origin ($x = 0$, $y = 0$) is to be found in the left topmost corner.

As far as the network layer is concerned, those four bytes (two source, two destination) are enough to transport a network message from its source to its destination.

In addition to the logic that switches the packets, the network layer needs a buffer for the packets coming from the transport layer.

Transport layer

Since I decided to use a fixed link width (like used in the NOSTRUM project [Homa]) of 256 bits for both directions, SNS limits a network packet to 128 bit. Each packet, coming to a transport layer, has to be broken down into 128 bit pieces. To reconstruct the fragments on the other site some additional information is needed.

First an additional byte in the header has to uniquely identify each fragment of a packet and it must also reflect in which order they are. Then to be able to distinguish between the fragments of different packets an additional identification number is needed.

With those two bytes of information we still need to know how much fragments there actually are and with that we get the resulting header for a packet fragment as shown in figure 9.1.

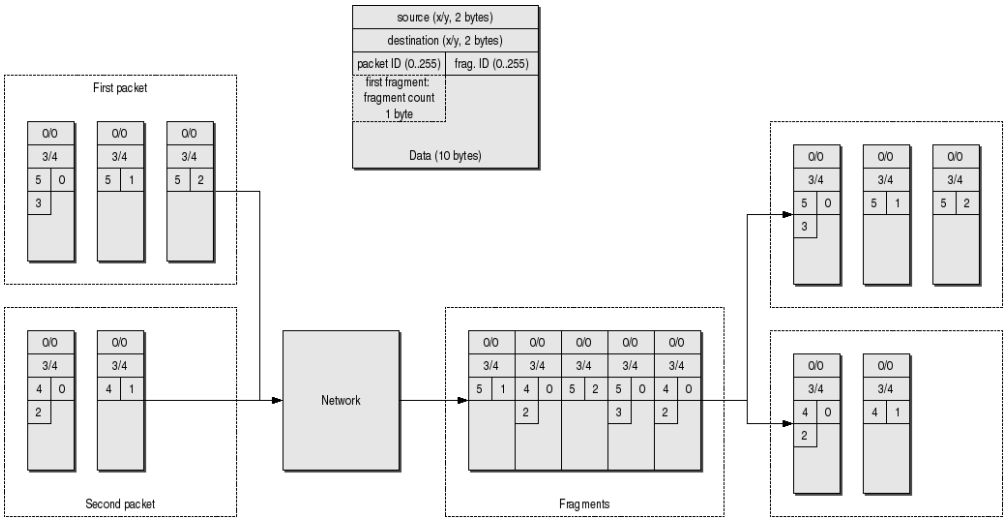

Figure 9.1.: Header of a packet fragment and example of sending and reconstructing two packets

This example shows that each packet gets split into several fragments and the order in which they arrive is not determined. Each fragment header consists of the source and destination address (4 bytes), of the identification number(1 byte) for each bunch of fragments and of the fragment count(1 byte). With that the size of the header sums up to 6 bytes, leaving 10 bytes in each fragment for the transport of the network data.

The identification is unique between each source and destination address and allows no more then 256 packet to be on the way from one source to one destination point at once. The fragment count (also fragment identification) indicates the order between the fragments.

To know how many fragments are needed to reconstruct the whole packet, we use the fragment count information from the first fragment (fragment count equals 0). Sorting by the fragment count and packet identification, from each set of fragments the whole packet can be reconstructed.

Be aware, this kind of mechanism, provided by the transport layer, sets two important limitations:

1. The maximal size of a packet is $256 \cdot 10 - 1 = 2559$ bytes.

2. Maximal 256 packet can be circulating in the network for each source-destination pair. More packets can lead to an unsolvable packet mix up[1].

[1] And further to a network error, in the best case; worst case: segmentation fault.

Besides the addition to the header, the transport layer needs two kinds of buffers. One kind holds all packets that have to be sent and the second kind of buffer has to take care of the incoming (fragment) packets. The size of those buffers in SNS is big enough for any situation.

In a real implementation the size of the buffers might be very limited, to reduce the overall chip area cost of a transport layer module.

9.1. Resource interface

To access the network resource a processing resource needs an interface. Since the original SEMLA architecture did not define one I define it here.

First we need obviously a read and write access. Since I defined already a flexible data container for the data exchange between the layers the same container (protocol data unit, *PDU*) gets reused as the data type for those two access methods as well.

Then there is actually also the need for a more interactive mechanism. Each time whole packet can be reassembled in the transport layer and is ready to be read, the processing resource gets a signal. With that the (simple) interface between the processing and the network resource is complete. The result can be seen in figure 9.2.

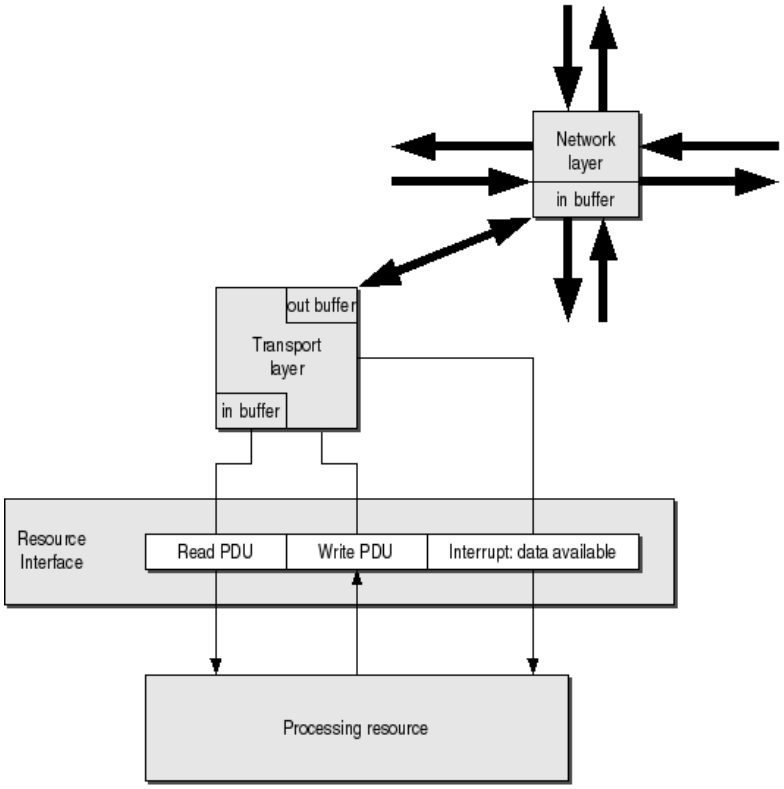

Figure 9.2.: Interface between the processing resource and the network resource (transport layer).

9.2. Inactive clock cycles optimization

This section describes speed optimizations used for the mesh network and transport layer from SNS. If you are not interested in this kind of stuff, skip this section, it is not that important for the end results, although in a sense it made them possible.

Tests about the execution time of the simulation showed that the most (time) expensive part of the simulation is in fact the SystemC bulk behind the network layers.

To operate the network with a high clock frequency means a big slowdown of the simulation. To decrease this effect SNS tries to separate the network and transport layer from the clock signal as much as possible.

This can be done by three steps, instead of activating the layer for each clock edge:

1. Wait for any change on the input ports.
2. Wait till the next clock edge.
3. Activate the layer (e.g. perform the layer's work) and return to step one.

With those three steps the layers react to changes on the input side only. Sometimes the internal buffer states have to be considered as well, to keep the integrity of the simulation. Like, if the transport layer still has packets in the outgoing buffer, the layer has to be activated, too, even if no change on the input ports has happened.

The optimization works especially well during low network utilization. The optimization factor was tested only subjectively.

9.3. Mesh network generator

The SEMLA [TMJ03] simulator uses a *topology* generator to create and connect (mesh) networks of variable sizes. In order to reuse the same idea for SNS some modifications had to be made.

First of course the generator had to be made aware of the additional simulation elements like the processing element and the operating system with its extensions.

With that the mesh network generator does not just create the network alone but it creates and connects all necessary simulation components and is called *mesh simulator* (generator).

To make the generator more configurable, too, SNS offers the possibility to determine the size of the network and the used components by entries in the configuration file for the simulator.

For example it is possible to change the used scheduler or allocator by just changing the corresponding configuration file entries. The whole process is shown in figure 9.3.

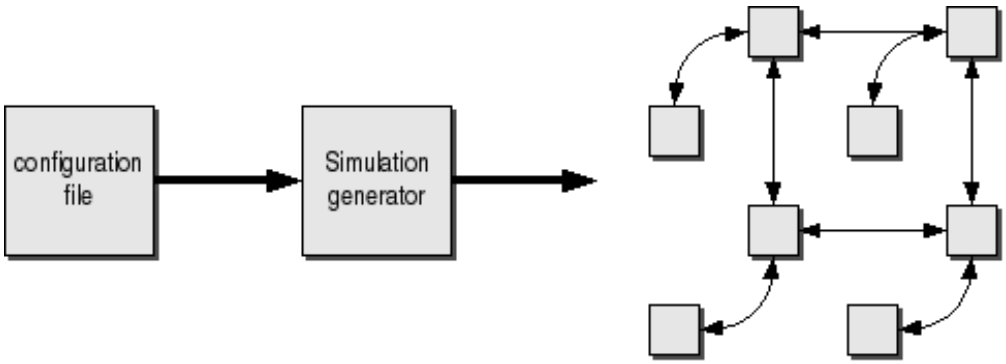

Figure 9.3.: Using the information from the configuration file (such as for example the desired network size) the generator creates all necessary objects and connects them to get a ready-to-use simulator.

10. Processing element

A processing element can represent any kind of logic, which is connected to the resource interface of the network. It can be a signal processor, a graphical processing unit a mathematical processor, a sound processor and so on. In the context of this master thesis however the processing elements are altogether of the same type and that is the *general purpose processing element*.

I introduced a processing element interface for the simulator, so that in the future it might be possible to implement other processing element types and with that build a simulator that makes use of specialized hardware. Of course this requires a lot of other (designing and programming) work, too. The allocators have to be aware of the new resources and more important, adapted versions of the operating system and possibly the task model are required as well.

10.1. General purpose processing element

This type of processing element is currently the only type in use for SNS. It is supposed to be able to handle all execution situations during the simulation (that is why it is called the general purpose processing element). It does not represent a specific kind of hardware it rather stands for any central processing unit, that can execute programs.

However, as in Thormann's master thesis [Tho05] mentioned(and explained in section 2.1), one important requirement for the logic is the possibility to vary the execution speed (and with that the power consumption). Thormann uses for that purpose one single parameter that indicates the current execution speed (and utilization) of a processing element.

SNS uses this same concept to simulate the computation resource use and power consumption with the difference that not any value between zero (no utilization) and one (full utilization) can be used. Instead only certain predefined values are possible.

For example a processing element can operate at 0.2, 0.5 and 1.0 efficiency. In other words the frequency of the processing element cannot be reduced to any variable speed but only to defined steps.

Those possible clock periods can be defined in the configuration file and each time the speed of a processing element gets changed (most likely by an allocation algorithm) the processing element jumps to the next higher possible execution frequency as desired. For example an allocation algorithm registers the completion of a task and because of that the execution speed can now be reduced to 500 MHz, but the processing element only supports 400 MHz or 700 MHz, so the new execution speed is set to 700 MHz.

Until now I have not made any major changes on the processing element (except the limitation to steps of the execution speed), that means the power formula $power = constant \cdot speed^3$ is still valid for the general purpose processing element.

I have two reasons now to add an additional property to the model. First the above power formula is restricted to the dynamic power consumption only and secondly I do not have a solution for how to handle an idle situation (no tasks are executed at a time), yet.

To cover those issues a general purpose processing element gets an additional sleep state. Instead of reducing the speed to zero the processing element can enter a state that is marked by a minimal power consumption (the sleep power) and the fact that no code is executed (see figure 10.1)

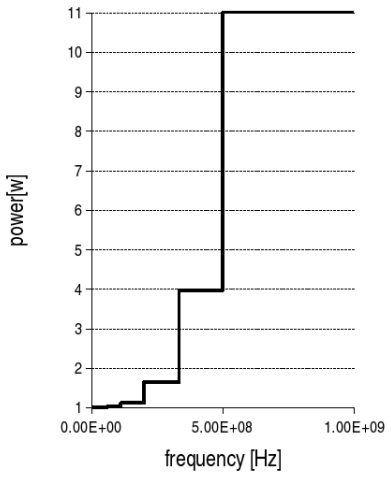

Figure 10.1.: Example for different speed steps and the corresponding power consumption. In this example the sleep power is 1 W.

To include this new state in the power calculation the sleep power gets simply added to the previous presented power formula.

For the determination of the processing model's power parameters I just took a look at currently available processors (for SoC) and used similar values (for sleep and max power) in the simulator, but since I need to shift the processing element to network power ratio in the experiments (as explained in chapter 8) I rather guessed the parameters, then using real values.

Speed change time and suspend time

The changing of the execution speed and the entering or exiting of the sleep state do not happen immediately. The change of the frequency (and the voltage too) requires some time for the logic to adjust. The same applies to changing to or from the sleep state (see example in figure 10.2.

To simulate that detail we can just use two more parameters and simply consume the required time (make a SystemC *wait*) each time such an event happens. As for the amount of those two time parameters, I just try to guess the valuse like I do with the power consumption.

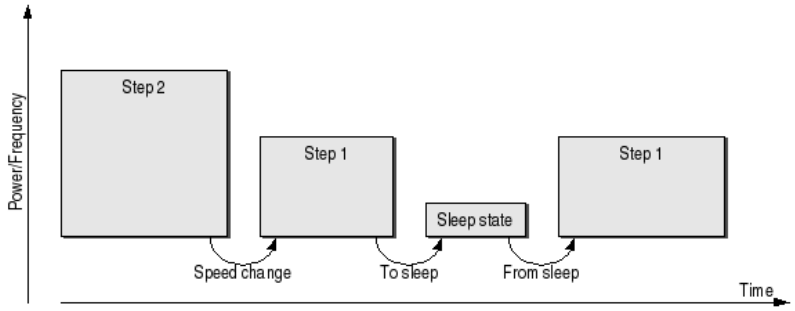

Figure 10.2.: Example for different speed (and power) steps and the time that is needed to change the processing element's operation state.

10.2. Task execution

Until now we can modify the speed step of the processing element and or send it into the sleep mode to reduce to power consumption to a minimum. The main purpose of a (general purpose) processing element is however the execution of tasks.

Since the simulator has now knowledge of the command set of a processor we cannot execute a task command by command. Instead it is enough if we know how much time is needed (at the current execution speed) to execute a certain amount of them.

Thinking like that we can define the "execution" of a tasks by three simple steps:

1. Let the task do whatever it wants to (like sending some network data).

2. The task tells us how much commands its actions are approximately worth.

3. The processing module waits the amount of time, that those commands would consume.

Note that the time which I speak of here is the SystemC [Pan01] simulation time. As seen in figure 10.3 this means that all the actions of the task occur in fact within the first clock cycle that the task gets executed. The rest of the time the task just consumes processing time without doing anything. This may seem a bit inappropriate, but the model is not complete, yet.

If the task does perform actions that do not involve other components (meaning the task just changes it's internal states), the exact point of time when such an action happens is not relevant for the simulation.

This is true, because we do not simulate the access to any memory during a task's processing time (only at the tasks creation time we have one access to off-chip memory, simulated by the memory task, see section 5.4).

Actions of a task that do influence other components of the simulation are restricted to operating system service calls (see subsection 6.2.2).

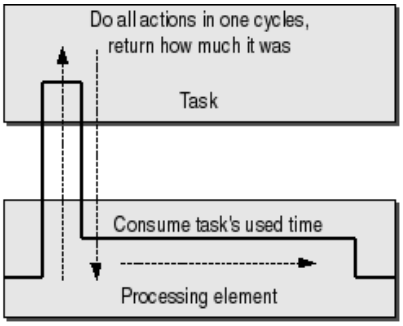

Figure 10.3.: Task execution simulation. First the processing element lets the task perform its actions, the task returns how much cycles the action where worth and the processing element consumes the time for those cycles.

With our model a task would simulate the execution of any system call within the first clock cycle, used by the task's execution. To change this behavior (and make it more real) we let the task make the request for any operating system service but we do not execute the system call immediately. Instead we wait until the operating system gets some execution time and then we perform the requested operation.

That is about it. The task's execution model does not get more detailed as this and we have the model as presented in figure 10.3.

Given this mechanism for the use of operating system service from within a task we have to deal with two (acceptable) consequences:

1. After the execution of each task the operating system service must have a chance to do it's job (e.g. at least select the next task to execute).

2. The task makes just one system call for each time that it gets to consume processing resources.

Switch cycles

The switch from one executing task to the execution of another task is not timeless. It requires the change of the internal state of the processing element, like the modification of some registers.

To take this time consuming action into account during the simulation, the processing element gets an additional parameter. This *switch cycles* parameter determines the amount of clock periods that are needed to switch between the execution of one task to the execution of another task.

The processing element consumes (waits) the time required for this action according to the current execution speed each time the task to execute gets changed.

Cycles factor

Behind this cryptic name stands the mechanism as described in section 5.4, which is the possibility to speed up or slow down the execution of a certain task type.

For each processing element we can define how fast each task type can be executed. Thereby we simply multiply the the time that a task wishes to consume by the given *cycles factor*.

For example to make the execution of a task 50% slower on a particular processing element we simply set the corresponding cycles factor to 0.5.

This allows the distinction of different types of processing elements, which is currently not used by this master thesis. It would add the possibility to first define different tasks (like tasks that need a set of special graphic acceleration instructions and tasks that do not) and define what the slow down or speed up for different processing elements is (like how much slower is a task that needs the extended instructions set but gets executed on a processing element that does not have those instructions).

Why use this parameter then at all?

Because it also makes it possible to define for a processing element which type of task cannot be executed at all (by setting the factor to zero). In SNS this later possibility is used for the memory task mechanism in section 5.4.

10.2.1. Operating system task

Since the (general purpose) processing element only knows about how to execute tasks and since we have operations that are restricted to be used only be the operating system service, SNS makes it possible to define one of the tasks as the operating system for each processing element.

This means that the operations

- changing the execution speed
- changing the task to be executed

can only be done by the task, marked as the operating system.

This should prevent any misuse of the processing element interface from any task implementation and it also forces the use of a specialized task model, the operating system service.

If looked at it the other way around, the operating system service is nothing else then a special kind of task and has therefore the possibility to consume processing time, like any other task, as displayed in figure 10.4.

This last fact is a very important detail of the simulator. If the operating system can consume processing resources as well, then we have to pay attention to the complexity of the used allocation and scheduling algorithms (and the various other actions of the operating system service as well) and the tasks get slowed down if the operating system has too much to do.

Interrupt handling

The network interface resource defines an interrupt signal (see section 9.1). This demands some kind of interrupt handling mechanism for the processing element. Actually we let the operating system handle the interrupts but the processing element has at least to tell the operating system about an interrupt.

Each time an interrupt occurs, the processing element interrupts the execution of the current task and gives the control to the operating system service, so

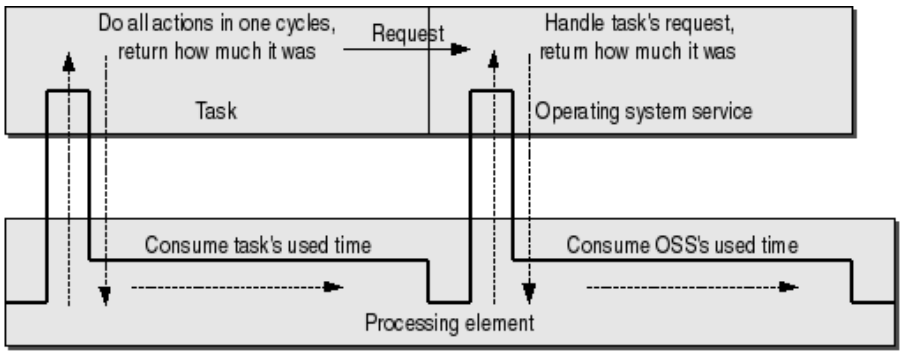

Figure 10.4.: Task and operating system execution simulation. First the processing element lets the task perform it's actions, the task returns how much cycles the action where worth and the processing element consumes the time for those cycles. The next time the operating system can make use of the processing resource it and performs the task's request.

that the operating system can take care of the interrupt. The operating system itself on the other hand cannot be interrupted and any interrupt that occurs during that period of time, has to wait until the end of it.

Boot operation

One small detail, that I would like to mention here, is the problem of initialization of the processing element. That is, since we have defined the special marked operating system, we need to tell the processing element about which task of all the existing ones, the special one actually is. To do so is easy and is covered by a simple function call. But there is more to it than that.

Somehow we have to start the execution of the first task and since the only known task (at least at the simulation's begin) is the operating system task, the first action of the processing element are this:

1. Set lowest processing speed.
2. Generate first power reading.
3. Set the operating system task as the one to be executed next.
4. Start processing (executed the operating system task for the first time).

As usual this initial actions are the boot functionality of the processing element.

11. Operating system service

Why is it called operating system *service* and not just operating system? Because the simulator does not have an actual operating system, it just has some parts of it, a small selection of services and so it is just called the operating system service.

What is it anyway? *It* is the necessary component that links tasks, allocator and scheduler together and makes it possible for them to consume processing time and network resource in form of sending and receiving messages. In other words, on each processing element runs (from the begin until the end of the simulation) a task that controls the access to resources.

Determined by the previous chapter 10 the operating system service is nothing more then an extension of a task. That means the each use of an service (defined in 6.2.2) consumes some amount of processing resource and with that delays the execution of other tasks.

Operating system service routine

Each time the operating system service task gets to consume processing resources it follows the same steps:

- Perform the requested service (like send a network message; only one service at a time is possible).
- Handle any occurred interrupts (like accept data from the network resource and possibly notify the scheduler or allocator about it).
- Ask the scheduler for the next task to run, if no task is left put processing element into the sleep mode.
- During those actions, sum up the consumed amount of required computing resources and pass that value on to the processing element.
- Let the processing element do it's work.

To perform all the different system services (defined in 6.2.2) like sending a network message or handling an interrupt the operating system in SNS does a lot of work and makes use of many different programming concepts.

It does not make sense too explain them all in details at this point, because as long as the simulation works the inner working of the operating system code is not a concern for this master thesis. Interested readers can instead take a look at the actual code of the simulator and its documentation.

Timer interrupt

Most of the relevant facts are already described in previous parts of this document (mostly in part II). The offered service like access to the network and

processing resource are already explained and the needed components defined. One important mechanism however was not explained, yet.

Sometimes an allocator or scheduler algorithm might want to use a time limit (timeout) for an answer to a sent network message. Let's not consider the why at this point but the how. The operating system offers the possibility to notify the extension about the when of certain point in time has passed. For doing so the operating system itself needs someone that measures time.

The solution to that lies in the use of an additional (hardware) component (lets call it the *timer*) that accepts as input a point of time information and creates as soon as that point of time is reached an interrupt. The operating system then gets called from the processing element, because an interrupt occurred, and can handle this *time interrupt*.

According to the operating systems internal state, the operating system can then notify any task or scheduler or allocator about the time event.

11.1. Allocator extension

Previously (in subsection 6.2.2) I wrote about what services an allocator might need, now it is time to look the other way around and see what service an allocator actually has to provide for the operating system.

The allocator is actually just an extension of the operating system and in the simulator an allocator has to provide all functionality that is directly dependent on different master systems and has to be implemented separately for each allocation method.

Let's start from what an allocator actually does: Assign a task to a processing element, where it mus be executed.

At the point when the operating system service gets the request to create a task it has therefore to call the allocator for actually putting the task somewhere. In return the allocator has to notify the operating system of the completion (or failure) of this action.

If the allocator knows about the task it might also want to know when the task gets terminated, so the operating system has to tell the allocator that, too.

Then there is the situation that network data arrives and the data is in fact addressed to the allocator (by a special information in the header of the message; explained in a following section 11.3) and the allocator must be notified about that. In association with it the allocator mus be advisable about a timer interrupt has well (to be able to handle timeouts).

But that is not the only function of the timeout. As described in section 5.1, a task can have a period, after which it has to be executed again. To mark the next repeated creation of the task the timeout mechanism is used, too.

As last functionality that cannot be performed by the operating system directly, we have the issue of determining the address of the processing element where a particular task is currently executed. This, too, has to be done by the allocator. And so we have the functionality that we separated from the operating system and give it to an allocator implementation:

- Allocate a task to a processing element (advice the corresponding operating system to create the task locally) and possibly change the processing element's speed.
- Revoke the allocation of a task and possibly adjust the processing speed.
- Handle incoming network data.
- Handle timeouts and possibly restart some tasks, from which the period is over (depending on the used configuration).
- Determine the address of where a task is currently executed, if at all.

Booting of an allocator

As with the description of the timeout I mentioned already that an allocator can create a task without any request coming from the operating system. This is necessary to recreate a task, after its period is over and so to simulate the periodic execution of tasks.

But someone has also to create the first task(s). To do so each allocator system must select one allocator that creates the first task(s).

It is not necessary to create any task at the very start of the simulation since each task can have a start time (like described int section 5.1), instead the selected allocator obtains a list of the start tasks at boot time and creates for each task's start time a timer interrupt.

When the the start time of a task has passed the task gets created automatically be the corresponding allocator and the simulation rolls.

11.2. Scheduler extension

The requirements to the scheduler extension are not so many as for the allocator from the previous section 11.1. Still some functionality changes with the use of different schedulers and must therefore be implemented by each of them.

The most important job of a scheduler is to determine at any point of time which task should be executed. Each time the operating system gets called the scheduler must be able to tell which task to execute next.

The operating system can be called because an interrupt occurred, the previous task is finished or, if it exists, the period of execution time for the previous task was over. Of course it is the scheduler who determines this time period for each task as well.

Of course there is also the possibility for a scheduler algorithm to make use of the network and communicate with others and this involves schedulers being capable of handling there own network data (and timeouts) like the allocators do, too. So the functionality that a scheduler has to provide is this:

- Add a task to the scheduling list.
- Remove a task from the scheduling list.

- Determine the next task to be executed and its maximal processing time (can be unlimited).

- Optionally, handle network messages and timer interrupts (timeouts).

11.2.1. Round-Robin and Earliest-Deadline-First scheduler

It is not explained anywhere else, yet, so I write now a few lines about the scheduling in SNS. Although the scheduling can influence the system a lot I do not observe it's effects on the energy consumption within the scope of this master thesis.

Anyway, to ensure that most of the tasks do not miss there deadlines I tried first to use a scheduler that just looks at the deadlines of all tasks and picks each time the task with the earliest deadline (Earliest-Deadline-First scheduling). This algorithm in conjunction with the tasks synchronization parameter (see section 5.1) can create a deadlock situations as in figure 11.1.

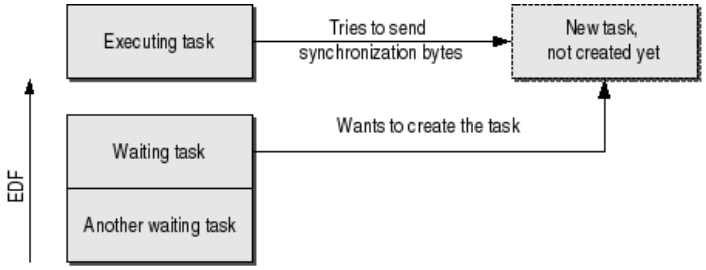

Figure 11.1.: The waiting task wants to create the new task, but is blocked by the currently executing task which waits actually for the new task to be created. Deadlock situation with using the earliest deadline first scheduler at runtime.

A tasks that tries to send synchronization data to another task (which does not exist yet) can block the actual creator of this other task and the system is stuck.

To avoid this SNS uses a halve Round-Robin half Earliest-Deadline-First scheduling.

The Round-Robin scheduling allows the execution of each task for a certain amount of time (the execution period) and when all tasks where executed once, it just starts over again. Combined with the other scheduling method it means that we first executed the task with the earliest deadline, then we take one task from the Round-Robin scheduling and then we return to the selected task and so one, see figure 11.2.

The execution period for each type of scheduling can be set independently (to give one or another kind of scheduling more execution time). With this algorithm we still ensure that as much as possible tasks can be executed within there deadlines, but we avoid the described deadlock situation, too.

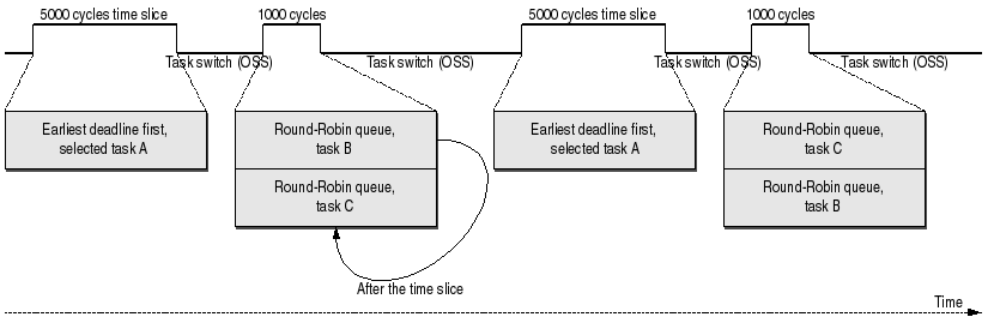

Figure 11.2.: The first period we do normal earliest deadline first scheduling, but the next time we do a Round-Robin scheduling to give other tasks a chance to do something too. Then it is back to the EDF scheduling schema and so on.

11.3. Mesh network operating system service

At this point the operating system service does provide the functionality for using the network resource, but in the SNS implementation I did separate the actual implementation and put it into another module. The reason for this is to make it possible for future work to implement the operating system service for different network topologies.

As for now the simulator just has a mesh network option and so the *Mesh network operating system service* implements the operating system services for the mesh network. The network extension of the operating system has to grant each other extension and the tasks access to the network's read and write functionality.

The scheduler and allocator extensions can send network messages to others (network addresses) but it is the operating system's network extension who must determine if a network message is for one of them or for a task.

To do so we give each network message an additional header (the *operating system header*) in which we define the type of the network message, as seen in figure 11.3.

The message can be directed to either the scheduler or allocator or an user task. For the later case we have to add the task's source and destination identification, too (how else should we know for whom the messages is or from where it did come from).

When a message should be written to the network resource interface the network extension adds first the operating system header and then on the other side with this information it can handle the packet accordingly. For each read request the operating system fetches one message (and only one, until it has been consumed by someone) and if it is a message addressed to a local task, the message gets stored in the local buffer for the corresponding task. With the next read request a task can obtain this stored network message.

If the allocator or scheduler extension tries to read, they get the message immediately (if it was addressed to them) or the reading fails and they have to try it again, later.

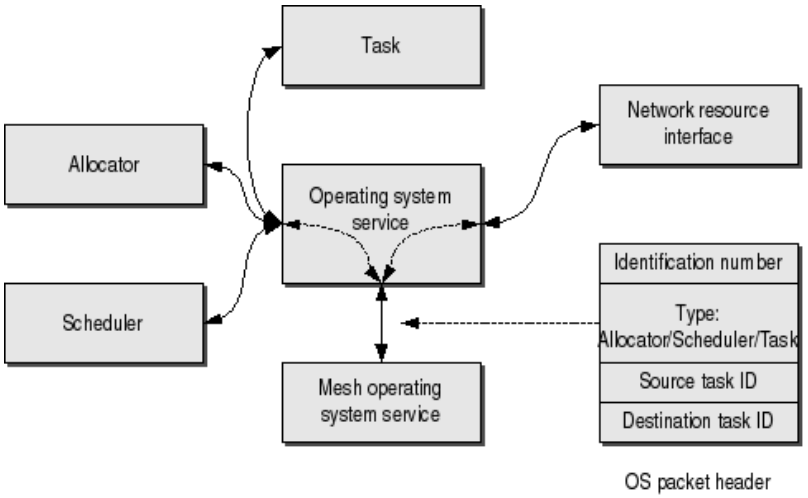

Figure 11.3.: All the actual network functionality is not done directly by the operating system itself. It is done instead by it's extension, the mesh OSS. It does handle all the network jobs and to do so it uses a special header for each network message to be able to distinguish between different kind of messages (addressed to a task the scheduler or the allocator).

12. Task model

The execution of a task needs many different components. All of them have been presented by now and so in this chapter I will concentrate on how a task is actually defined and handled within SNS. When I speak of a task I mostly mean the actual behavior of a task or the task model.

A task model of the simulator describes how a task behaves regarding the use of resources and service. It does not described what the task actually does, or what it's purpose is.

For example SNS does not requires tasks that have, well, a specific task. Meaning if we have a task that can decode a part of an audio stream we do not do the "decoding process" in the simulator, but we create (or configure) a task model that makes use of the resource and service in the same way as the actual decoding task would do.

For that kind of simulation of a task I already defined the parameters in section 5.1. The task model itself now, has to make use of the parameters and consume resources and services accordingly.

From the processing element's point of view the task model has already received the need of telling something about the used computing steps. In figure 12.1 it is possible to see, what I mean.

The processing element can simulate the time consumed by the (imaginary) execution of a certain amount of computation commands. At the time that a task can make use of processing resources the task model can do first the actual work (like prepare a network message and make the request for a network send request) and then it has to tell the processing element how much clock cycles (processing commands) the actions where worth.

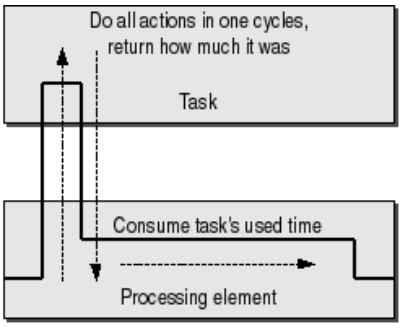

Figure 12.1.: Task execution simulation.

The processing element then consumes the time according to that result, but it does not do that necessarily all at once. For example if the "execution" of a task gets interrupted by an incoming interrupt signal, the processing element will eventually resume the consuming of time for that particular task.

At the point where the requested amount of computation cycles is done (or an operating system event happened) the task model can again do some work and it starts all over again.

Used data type

Depending on the actual implementation of a task model, different data types for network messages will be in use. To cover this face the abstract task model does not specify the used data type. A later implementation can make use of any data type, as long it is possible to convert the data into a byte array.

The variable data type opens the possibility to define any data protocol between instance of the same or different task models and with that it would actually be possible to implement also a real "decoding" task.

12.1. Operating system messages

Any task model can make use of the operating system service (described in 6.2.2). To make a request, like a network send request, a task (model instance) fills out the argument buffers, such as the destination task identification and the actual data, and then tells the operating system that it wishes to make use of the network send service. The next time that the operating system can do some work it grabs the parameters and fulfills the request.

By the time a task's request is done (or the request failed) the operating system has to tell the corresponding task what happened.

To do so, the operating system can send messages to a task. Such messages just tell the task if the corresponding request did succeed or fail.

In the case of a success the task can find a possible response in the argument buffer. For example if a task was created successfully the operating system tells the creating task the new task's identification by writing the information to the argument buffer of the request (displayed in figure 12.2).

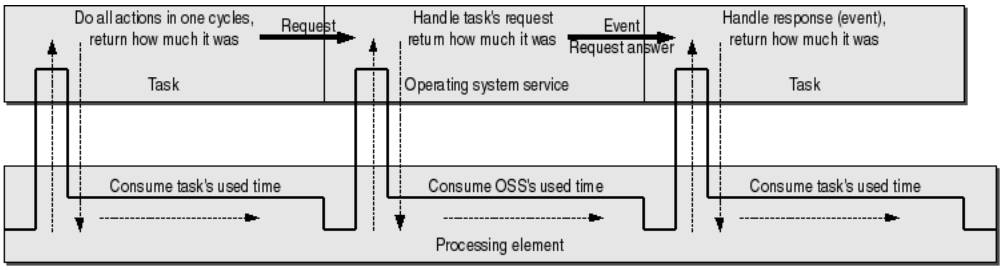

Figure 12.2.: As the tasks makes an operating system service request, the OSS will handle the request as soon as it can consume more cycles. The answer for the request gets passed on via an operating system event and the next time, that the task has its turn, it can handle the response (event).

All of the requests that are offered to a task model are non-blocking. When a task makes a request the corresponding message will be available eventually. At which time exactly cannot be said for all requests.

All of the services regarding the use of the network return a message for the next time that the task can do some more work.

The wait-for services send the corresponding event at the time that the task resumes it's work and the task termination request does not of course return any event.

Just the task creation request returns it's event some time later which cannot be predicted when that is. So, it can happen that a task has more then one event at its disposal when it gets the opportunity to do some more work.

For that reason the task model gets the possibility to receive more then one operating system events at a time. The latest message if existent gets passed on directly to the task model, at the time that the task can do more work.

The other messages have to be accessed by the corresponding mechanism, provided by the abstract task model (the base for all other task models).

12.2. General task

The only implementation of an abstract task model used in SNS is the *general task model*. This task model considers all task properties discussed in section 5.1 and behaves accordingly.

The task parameters contain the task's worst execution cycles amount. Every instance of the model determines at creation time the actual amount of commands, that the execution of the task is going to need in total. With that adjustment we get tasks with variable execution length (and variable processing resource utilization).

For example if the worst case should be 50000 then the actual amount of cycles consumed by the task in total could be just 400000. To limit this variable adjustment the general task model gets an additional parameter that tells the task about how much the lower boundary is, regarding the maximum amount.

In the previous example this could be 0.5 and each instance of the task model wold calculate the actual amount of cycles to consume like this: $cyclesToConsume = random(0.5 \cdot worstCaseCycles, worstCaseCycles) = random(25000, 50000)$.

Besides the modification of the worst case processing resource use parameter each task model creates for each communication partner a list that tells the task when to send how much data to that particular task. The when is actually given in cycles.

For example if the communication parameter contains the following information (table 12.1):

comm. task id	*min interval*	*max interval*	*min size*	*max size*
4	200	300	20	40
7	100	500	5	8
11	130	130	10	10

Table 12.1.: Traffic parameter example. For each communication partner the parameter contains the minimal and maximal boundaries for the interval between two messages and the size of the message.

Then the result for the actual communication plan for the corresponding task model instance could look something like this (table 12.2):

comm. task id: 4	*cycles*	123	368	624	864	...
	size	38	27	34	29	...
comm. task id: 7	*cycles*	425	587	1067	1412	...
	size	5	7	5	6	...
comm. task id: 11	*cycles*	130	260	390	520	...
	size	10	10	10	10	...

Table 12.2.: Traffic list example. Each communication partner gets a list of when (after how much cycles, counting from the creation of the task) and how much bytes to send to each of the tasks.

It is easy to see that we can adjust the amount of how much communication a task does with those parameters and with that we can determine if a task does more (input-output) network resource or computation resource.

Actually we cannot determine the input amount directly (because at this point we do not know who is going to send something to the task), so I decided to give the general task model an additional parameter that tells the task model to send a message back to the origin of a network message, except in the case that the origin is on of the task's communication partner.

By that we can assume that each task from the traffic parameter sends also some network data back to the task and we have established a simple two way communication.

12.2.1. Execution states

After the determination of the above parameters the execution of a general task model instance can begin.

Start synchronization state

According to the the tasks parameters each task has first to wait to receive all synchronization messages specified in the start synchronization list.

During this *start synchronization* state the task just waits till it receives a network message and does consume a constant minimal amount of cycles each time it checks for a message.

The cycles consumed during this state are not considered as the actual work of the task and therefore not counted for the fulfilling of the to-consume-cycles.

Progressing state

If all start synchronization messages have arrived the task enters the next state in which it tries to consume all the cycles that the task should consume. This state i call the *progressing* state.

Also, the task does send network messages (according to its traffic list) and it creates other tasks according to the task's task-creation parameter.

Each time the task can do some work it does just one action and it calculates how much cycles have to be completed until the next message should be sent or the next task created. It returns then this value to the processing element,

such that after this amount it can continue it's work (send the next message or create the next task).

A network send request is guaranteed to be completed until the next call for the task (the next time that the task is allowed to work, there will be a corresponding operating system message waiting for the the task). However we do not know how much time it takes to process a task creation request.

It can therefore happen that at the time when the task has has consumed all its must-consume cycles there are still tasks left to create.

Last task creation state

The next state of a task is also the *last task creation* state. Sometimes we even want the task to create other tasks not before it has done it's amount of cycles, so we just put the point of creation for such tasks in the task's creation parameter, at the end of the life time of the task.

Anyhow in this state the task creates all other task, left to create, according to the corresponding parameter. When all tasks are created (each request was a success) the task has one more thing left to do.

End synchronization state

In the beginning the task waited for other tasks to send it synchronization messages. Now it is time for the task to do the same for other tasks.

It enters the *end synchronization* state, where it sends synchronization messages according to the tasks end synchronization parameter.

After all synchronization messages have been sent ssuccessfully, the task terminates itself. The complete diagram about all states can be seen in figure 12.3.

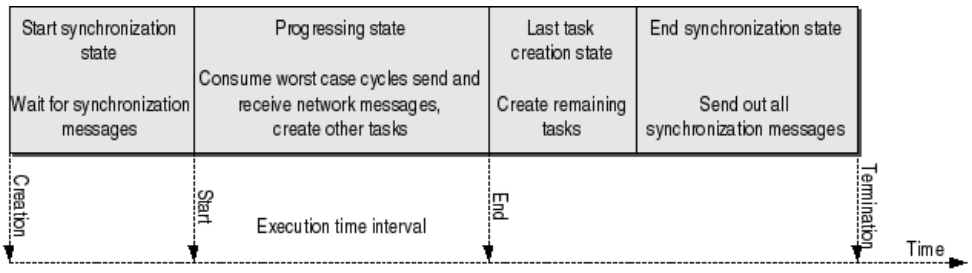

Figure 12.3.: The different states, that each general task model works through.

12.2.2. Memory task

The memory task is used to simulate the transfer of some data between off-chip memory and a task which is about to start processing via a special memory controller (see section 5.4).

This mechanism does not require a new implementation of the abstract model, instead we can reuse the general task to do the job.

A memory task has no start synchronization state no progressing state and no last task creation state. The only thing that the memory task has to do is to send some data to the task that is about to start processing.

This other task has then to add the memory task to its start synchronization list and everything works like it should.

To force the general task model to skip the states that are not needed, we can simply let the corresponding parameters be empty or set them to zero. This parameters are:

- Start synchronization list: *empty.*
- Worst case execution cycles: *zero.*
- Traffic patterns: *empty.*
- Tasks to create list: *empty.*

The start time and period parameter of the memory task have to be the same as for the pending task.

13. Reports

SNS has not only to simulate something, but more important it has to generate some results that we can use to speculate about what the simulated experiment actually has to teach us.

To make every measurement as flexible as possible I decided to use events that are created each time something of interest happens. Such "interesting" things can be anything and will probably become more and more as the simulator grows.

For example if a processing element changes its processing speed, an event that contains the information about the value of the changed speed and about who's speed was changed gets created and delivered to a central component that collects all events in SNS.

The events do not have anything to do with the actual SoC with NoC simulation and therefore they do not change any result of an experiment. No events about events exist.

The decision of what is actually of interest and should be marked by a corresponding event was not made by evaluating the simulation procedures and choosing the most important ones.

Instead I examined first which results might be interesting and inserted the the needed events. In the same way it is of course possible to implement new reports and events and to get even more results.

One nice aspect of using events to obtain results is that once an event gets created and delivered to the main distributer, any report can make use of it. Each report just has to register itself for the kind of events that it wants, before the simulation starts.

This solution makes it possible to reuse events for new reports and of course to use the same event within more then one report at once. The in this way during the simulation by each report collected data gets usually written to a report file.

It is possible to write the results from different simulations into one file and mark each of them with a different identification (at least by now all of the reports support that feature).

To keep the number of reports created by SNS low, some of the reports generate more then one report file. Also take into account that some of the report file formats can be changed by corresponding configuration file entries.

Most of the reports display the collected data in the form of a table and all report files are comma separated values(CSV)-format [Sha05] conform with the addition that the "#" character introduces a comment line (which does not contain data) and the commas are in fact replaced by tabulator signs.

Such a report table usually looks as follows:

#time of event[sec]	*element 1 [some unit]*	*element 2 [some unit]*	...
0.0000000	45	33	...
0.0004509	45	33	...
0.0004600	20	50	...
0.0005009		50	...
0.0005100		24	...
0.0005509	20		...
0.0005600	76		...
...	...	...	...

Table 13.1.: Report table example. Each line corresponds to a certain point of time. Not for every point of time all of the element values might be available. In this case the report also uses the *rectangle* style: For every entry there exists a another entry 0.0000001 seconds before the actual event to mark the old value and generate a kind of steps-graph, that gets visible if all values are connected by lines.

13.1. Kinds of reports

Although I might not mention every result obtained by the different reports in the result chapter 15, I still will need all reports to check if the experiment was within acceptable parameters or something odd happened.

For example I will use the report for lost PDU's to check if any network message was lost during the simulation and might have stopped some task from being created (indicated by another report).

Or we can take a look at the bytes-traffic-per-location report, that tells whenever the traffic caused by the operating system services is much higher then the actual task generated traffic (in which case an adjust of the corresponding work scenario parameter would be required to generate more task traffic) and so on.

In the rest of this section we will therefore take a closer look at which reports exists and what they can tell us.

13.1.1. Bytes traffic report

This report can generate two result files: *ReportBytesTraffic.csv* and *ReportLocationBytesTraffic.csv*.

Bytes traffic

The first file contains informations about which task was executed on which location and the amount of bytes that where sent. The operating system service is represented by the task identification number 0 (on each location it occurs only once). If a task gets executed more then once it can show up on more then one location, too.

With this report it is possible first to tell how much traffic was created by each individual task and operating system service and where (the task) has made use of the network resource.

The amount of traffic created by each task gives a possibility to check how

much each task did actually communicate with others and it is therefore an indicator of how good or badly balanced the work scenario is.

Location traffic

The second report contains the information about how much for each location the sum of the traffic generated by all local tasks is and how much of the traffic was caused by the operating system service.

The traffic value corresponds to the sent bytes and the report also displays for each network location the operating system traffic per task generated traffic ratio.

The information about how much the task generated outgoing traffic on each location is interesting but by it alone not very useful.

The information about the operating system traffic on each network node can tell us if for example in the master allocation system the traffic at the master node was way higher then on the other nodes or if that influence was not noticeable.

The third information about the ration between the two kind of traffics is the important information from this report. It tells us about how much traffic overhead was generated by the allocation algorithms (since no other part of the operating system service makes use of the network resource) and that is a indicator for how good or bad the used allocation system is.

13.1.2. PDU lost report

This report creates just one file and that is: *ReportPDULost.cvs*.

It contains informations about which (named) SystemC module lost a protocol data unit and what the time of the event was.

This is mainly a check if all messages where delivered during the simulation as requested. If some data got lost unexpectedly it could lead to some misbehavior of SNS and to inappropriate results.

Therefore any time that I got some lost network messages I reconfigured the network, so that the situation did not occur anymore.

There are (currently) three components of SNS that can loose a network message (because of a buffer overflow) and they are:

- Network layer: PDU queue, equals in-buffer for the transport layer.
- Transport layer: In-buffer; if the out-buffer is full a write request would return a failure.
- Operating system service: Any task's in-buffer.

13.1.3. Consumed cycles report

This report also produces just one file: *ReportConsumedCycles.cvs*.

The report contains informations about how much cycles where consumed on each location and by whom. For each location we have the information about how much the operating system made use of the processing resource and how

much in total the tasks did use the local processing element. All values are given in clock periods count (cycles).

The third information contained in the file is about the ration between clock cycles used by the operating system and all the tasks for each network location.

This report gives us a clue about whenever some processing elements where busy just consuming processing time for the operating system or could do some more useful things.

The ration operating system versus task should preferably be very low. If it gets to high it means either there were not enough tasks to be executed and an adjustment of the work scenario is required or it means that the processing element was indeed mostly used for operating system operations, which is rather bad and needs a very god excuse.

13.1.4. Packet fragment traffic report

This report creates the biggest amount of results and it writes six files :*ReportNetworkEnergy.csv*, *ReportTrafficPerTime.csv*, *ReportNetworkPower.csv*, *ReportTrafficPerTimeNLToTL.csv*, *ReportTrafficPerLocation.csv* and *ReportTrafficPerTimeTLToNL.csv*

Network energy report

The energy report contains the energy consumed by each network location and the total sum till the given time. In the energy consumption is included the transport layer, the network layer (switch) and the link layer (just the half link for each connection).

This is of course one of the essential reports. It indicates the tendency of the networks energy consumption and this graph will be discussed in the next part IV of the master thesis.

Traffic per time report

Each time a network message reaches the transport layer, coming from or going to the network layer, a corresponding event gets created and the information is used to create three different reports that contain each the same kind of data.

For each location the report records the passing through of a network packet (actually just the 16 bytes fragment of a bigger message) at a given time. Hereby does the *NLToTL*-report record all fragments that are passed from the network layer to the transport layer for each location (and the sum of them) and the *TLToNL*-report records the other way of the fragments. The third report finally documents both kinds at the same time.

The traffic per time reports are actually just to see how much traffic was handled on each location at any time and that is about it. It is though the only report the tells the exact amount of traffic that occurred during the simulation.

Network power report

This report contains the power readings for each network location for a given time. It just contains the actual changes in the power reading of an location and it contains also the summed up power of overall the network.

The power report is actually just a sort of by-product, since the power values

are used altogether to create the network energy report, but what we can see from this report's graphs is still useful.

And what we can see from this graph is an indicator of how much the network was used at a given time during the simulation. A network's location power consumption is (not an exactly proportional) indicator for the local amount of traffic and it can be used to determine whenever a network location was particularly stressed at some point of time.

Traffic per location report

The traffic amount per time report we do already have. This report also records the packet's fragment traffic, but it sums the traffic up and displays it for each location.

The report distinguishes between traffic coming from or going to the network layer and it also sums up both of them to the total traffic values.

This report gives a quick overview if some processing elements where not reached at all by any network traffic, which would probably require an adjustment of the work scenario.

In addition to that this report too gives an overview over any kind of *odd* behavior of an experiment(like the bytes traffic report in 13.1.1 does, too).

13.1.5. GPPE cycle length change report

If any general purpose processing element(GPPE) changes its execution speed (equals clock period, equals frequency) this report catches the event and generates three files: *ReportGPPECycleLengthChanged.csv*, *ReportGPPEEnergy.csv* and *ReportGPPEPower.csv*.

Cycle length change report

This report contains every speed change of every processing element and it documents when those changes happened. The value of the speed can be given in the frequency of the element or in form of its clock period (equals $\frac{1}{frequency}$).

The report shows us directly how much occupied each processing element was at any time during the simulation.

Odd behavior of an experiment, like if one processing element was not used at all or if one element did change it's speed way to often, can be seen very well in the data of this report. Otherwise there is not much else that we can learn from it.

GPPE energy report

This is the second energy report (the other one is the network energy report from subsection 13.1.4) and it records the consumed energy of each and all processing elements till the given time.

This report too is of the uppermost importance and it's meaning will be discussed in chapter 15.

GPPE power report

The power report is a by-product of the energy report and contains the power use of each processing element at any time during an experiment. It also contains the total by all the processing elements used power.

As the speed report does already do, this report too is an indicator of how much a particular processing element or all elements together where used during the simulation. If some processing element was used to less or in a strange way, it is possible to see this fact in the graphs, generated out of this report's data.

13.1.6. Task start-end report

A general task model has certain states during its life time (see subsection 12.2.1) and the point of time at which any of this states of any task is reached gets recorded by this report. With that data the report generates this two files: *ReportTaskStartEnd.csv* and *ReportTasksLocation.csv*.

Task start-end report

This is the more interesting of both the reports and it records in fact the point of time at which a certain state of a task is reached. Those recordings include for each period and task:

- Creation time: Entering the start synchronization state.
- Start time: Eentering the progressing state.
- End time: Entering the last task creation state.
- Difference between end time and deadline: Equals start time plus execution time parameter minus end time.
- Termination time: Task is done and it's instance gets revoked.

With those informations it is possible to check if every task of the work scenario was executed properly and we can also see how many task did not make it to finish processing before the deadline (which should be zero tasks, but it might not always be possible).

If not every task completed at least once, then there is something wrong or with the configuration of the experiment or with the used work scenario.

What it is exactly can hopefully be guessed through the report's data.

Tasks location report

This last report records the location of a task for each time a task gets created. The order in which the tasks got created during the simulation can be left in its original order or it can be sorted for each processing element.

The first option just shows us on which network location each task was executed and in which time order the tasks where created.

The second option makes it easier to see how many times each task was executed (if more the one period is allowed) and thus makes it possible to see which task was not executed during all periods(if an overall global period was used).

Part IV.

Analysis

14. Configuration

Before we can take a look at the results of this thesis it is important to know how those results where obtained. By now SNS and how a simulation works has been discussed. It also was mentioned that most of the simulator's components are highly configurable. Therefore I am going to present and explain the most important configuration details for the following experiments (in chapter 15) in this chapter.

I will not name all of the possible configuration properties (there are several hundreds of them), instead I will follow the tracks of how to reach a fully valid configuration and name only the relevant properties.

The point from where I started to create a basic configuration for the simulator was the configuration of the processing elements.

A processing element can have a minimal and a maximal frequency (clock period cycle equals execution speed) and some steps in between. To choose a set of speed steps including the minimal and maximal processing speed means to choose how much computation resource the simulator will offer for the (simulated) execution of tasks. It affects therefor directly the size of the possible work scenario. It also influences the possibilities to save energy. More steps allow a more detailed adaption of the processing element and might keep the power usage lower then fewer step could.

To make this decision I reviewed the data of available microprocessors and then decided to use 5 steps between 50 MHz to 500 MHz.

Clock period (speed) steps [ns]	2	3	5	9	17
Corresponding frequency [Hz]	500	333	200	111	58.8
Speed step change duration [ns]	30 (15-2 clock cycles)				
Power usage: dynamic / sleep / max [W]	1	0.1	1.1		
Suspend to sleep and resume from sleep [ns]	50 (25-3 clock cycles)				
Task switch cycles	20				

Table 14.1.: Configuration of the general purpose processing element.

Similar to the speed steps, there is the possibility to determine the time duration of a speed step change and the time that it takes the processing element to reach the sleep state (low power usage) and return from it.

The overall used configuration can be seen in table 14.1. The cycles that are needed to make a task switch are not really part of a processor specification, but SNS performs that operation within the processing element module, so I added that value to this chart, too.

The power configuration is used directly by the power formula presented in section 10.1.

With hose setting I continued to specify the task parameters. To make the simulation duration not too long (more simulation time is equal to more memory usage of the simulator program) I set the range of the execution time parameter to one till two milliseconds. In order to determine the worst case computation cycles for a task I had to balance the operating system cycles, the scheduler's period for a task and the utilization of a processing resource.

The scheduler assigns each time the next-to-run task gets selected a processing period to the processing element. When this period time is over the execution of the task is stopped by the processing element and the operating system takes over. Each time the operating system holds on to the processing element it consumes a certain amount of cycles. Those cycles had to be minimal regarding the task's execution period to ensure that the task makes the most use of the processing element and not the operating system service[1].

On the other hand if the worst case cycles of a task are set to low, the task gets not switched enough and since it can execute just one system call (like send a network message) within one, scheduler assigned execution time period, it would not be possible for a task to perform all its duties. In other words the task would create less network traffic. If the worst case cycles are set to high to few task can be executed on one processing element.

After a some amount of test runs I decided to use the settings shown in table 14.2 for all experiments.

Operating system cycles	200	
Scheduler task execution period	1500 EDF	500 RR
Task worst case execution cycles min/max	120000	180000
Task execution time [ms]	1	2
tasks per processing element min/max	1	8

Table 14.2.: Configuration for the task's length and related settings. EDF and RR are both part of the Round-Robin and Earliest Deadline First scheduler, described in subsection 11.2.1.

Of course each task decides (randomly) by itself how much cycles to consume and the worst case can be reduced down to 70% which then allows more tasks on a processing element then indicated in table 14.2.

By now we know the execution parameters of a task and we go on by determining the parameters for the work scenario. The important parameters here are the settings for each scenario model. To make each model not to long in its overall execution time i gave for example the pipeline model a length range of two to maximal 4 tasks. The settings for all scenario models can be seen in table 14.3.

A scenario's size determines how much of the processing elements resource is maximal used at a time. For each network size used during my experiments I did run a series of tests to determine the amount of models (for each model type) and with that I determined the overall amount of tasks contained in

[1]The system has a preemptive scheduling, but in the SNS's implementation it is the processing element who controls the execution of a task and not the operating system

Model	*Parameter*	*Min*	*Max*
Pipeline model	*length*	2	4
	width	2	4
Star model	*nodes*	2	4
Tree model	*depth*	3	4
	nodes children	1	3

Table 14.3.: Configuration of the work scenario's model parameters.

the scenario, too. I adapted the size of each scenario such that all processing elements got used during an experiment[2].

With the size of the work scenario comes along the network resource request. Besides the sizes of the network's buffers the networks frequency is the key to grant a successful run of a simulation. But the use of the network is also influenced by each task's communication parameter. Each task creates at creation time a list of to-send messages. The list contains all messages sizes and send times (see section 12.2). The min and max values that each task uses are determined for each task individually with the boundaries of table 14.4.

Parameter	*Min*	*Max*
Min interval [cycle]	1500	2500
Max interval [cycle]	5000	8000
Min message size [byte]	50	250
Max message size [byte]	300	900
Communication partners	0	5

Table 14.4.: The min and max value for each task's communication parameters.

As this table indicates, I choose the size of the interval between the sending of two messages such that it is most of the time two to four times bigger then an execution period that a task has at its disposal. That way a task will probably make a system call (as for example send a network message) each time it can make use of processing resources.

If you want to know more about the configuration possibilities (like the configuration of the reports) you will have to take a look at the simulator's source and the used configuration files. In the next section I am going to present the experiment related configuration settings.

14.1. Experiments specific configuration

I cannot discuses the used configuration without mention how the configuration differs from experiment to experiment, but it does not make sense to explain the experiments within this chapter either. So I have to mention which experiments

[2] At leas I tried too. For the bigger network sizes it did not work out so well, meaning in some experiments some low amount of processing elements is nevertheless not in use.

I am going to perform in the next chapter and I will explain now how they are influenced by the configuration settings.

The important settings which were not mentioned yet are the power and frequency (and size) configurations of the used network.

As it is for the network power configuration I will use the power formulas of Penolazzi's master thesis [Pen05] presented in section 4.2. The frequency influences the power consumption as well, but more important it defines failure or success of the used work scenario. If the frequency is not high enough the network cannot deliver all messages (fast enough) and the simulation does not complete with all tasks done. Turing the determination of all the used work scenarios I found the frequency of 2GHz most suitable for all experiments.

Well, not all experiments. The ones that show the influence of the memory task and operating system service computation time needed a slightly lower frequency (which I am not sure why, but probably due some message timing issues). Those experiments that show simulation detail influences use a network size of 3 by 3 and the corresponding work scenario is composed of 60 tasks. As for the memory task, I placed the controller simulation in the top left corner (location 0,0) regarding the off chip connection that it theoretically has.

The memory task (see section 5.4) itself creates an additional network load between 7500 and 22000 bytes per task and the cycles that are consumed each time the operating system service gets executed are restricted to exactly 200.

I use the memory task only during the special experiment, whereas the operating system execution cycles are used in all (but the one special one) experiments. The reason why I did not make use of the memory task emulation during all simulation is the resulting, bigger work scenario. In fact some of the simulations that included the memory task tried to consume more memory then my computer has.

The experiments which show the allocation algorithm energy behavior are using three network sizes from 3 by 3 to 5 by 5 to 7 by 7. I choose those particular sizes to be able to place the master (and the first executed tasks) in exactly the middle of the network. Bigger networks are not part of the experiments, simply because bigger network sizes required too much memory, so I had to stop at the 7 by 7 network size. In table 14.5 you can see the different network and work scenario sizes.

Network size	*Tasks*	*Pipeline models*	*Star models*	*Tree models*
3x3	63	3	3	2
5x5	220	7	7	6
7x7	390	13	13	15

Table 14.5.: The used network sizes and related work scenarios stats.

Finally the variation of the processing element to network power usage ration should make it possible to see the influence of different configurations on the allocation algorithm capabilities, but due the high frequency (and with that low usage) of the network this configuration variation becomes obsolete as will be discussed in the following chapters.

15. Results

This chapter presents the results of the experiments. As in the introduction mentioned the main purpose of SNS is to introduce more simulation details. So one of the first experiments shows if the more detailed simulation of the processing and network resource has its influence on the energy consumption. I cannot make a direct comparison between my results and for example the results from Thid's master thesis [Tho05] (described in section 2.4) though, because the simulation settings (processing capacities, network clock frequency and so on) are different and do not allow a direct comparison.

So, the experiments concentrated on showing the influence of the simulation's new features and of course the energy per time costs from the three different allocation algorithms (first-fit, worst-fit see 2.2 and neighbor allocation see 7).

Thid's results show further on that the variation of the by the tasks preferred resources usage (input-output bound or computation bound) has a significant influence on the energy consumption, too. Since in the following experiments about 60 to 400 tasks are executed within 15 ms and less, I assume that for a start the average of the resource preference is about even between processing resource and network resource and furthermore the used work scenario is that big, that local extreme cases do not influence the overall results to much. In other words the IO versus computation effect is not shown by my master thesis.

Instead lets take a look at the first experiment that provides us with an important result.

15.1. Network energy influence

As far as the energy contribution from all system components goes, the network has a big influence on the used energy. An experiment with a 3 by 3 network and a periodic repeated work scenario shows in fact that the total network energy has about the same order of magnitude as the total processing element energy.

A network node (transport layer, network layer plus link power) can reach a maximum power of about the same value as a single processing resource as well. This means that if the network power and the processing element power is about the same then so is the energy consumption, which is not trivial, considering the possible different usage.

Anyway the important conclusion can be seen in figure 15.1, which shows the network energy consumption for all three different allocation algorithms. To clear it up immediately, yes all three results are shown in the figure together. The thing is, they are (not exactly exactly but very close) all the same.

This very same behavior can be seen in all simulations, all tough the allocation algorithms change the nature of the network traffic, the energy consumption

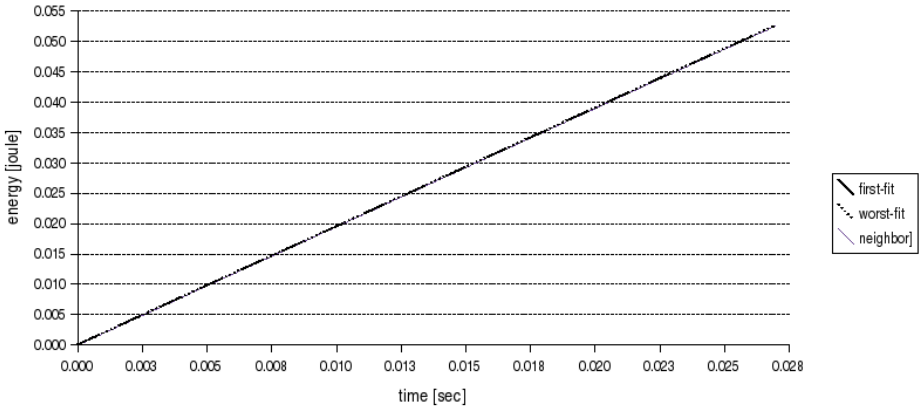

Figure 15.1.: Different allocation algorithms: network energy consumption, consumed by a 3x3 network(2GHz) and periodic task repetition (3 periods).

graph (in end value and progression) is always nearly the same. There is no visual difference visible at all.

The reason for this behavior lies in the next graph. Figure 15.2-top shows the power use of the network and with that the activity of the network. From the form of that graph we should not get such a continuous energy graph as in 15.1.

But if we look at the bottom graph of 15.2 we see the same power readings, this time with proper scaled power axis. And now the energy figure becomes clear. Obviously the 2GHz network is nearly idle all the time. At least from a global point of view.

Since there is so much unused network capacity, I tried to reduce the network clock frequency (and with that the power need), but then the simulation did not complete properly. Some tasks are not finishing any more because of missing synchronization messages.

This leads to the conclusion that in spite of the fact that there is a lot more network resource at disposal then needed, locally (in time and location) the full capacity of the network is needed to fulfill the communication requests of the simulation.

Besides those facts the above example has another important consequence. The network energy consumption is not influenced by the used allocation algorithm in my experiments. Therefore I want consider the network energy in the following experiments.

Let me rephrase this last fact: The network energy in my experiments depends only from the used network's clock frequency and size. The network frequency remains the same throughout all the experiments.

15.2. Memory task influence

The memory task emulation (from section 5.4) introduces the possibility to use one processing element as a sort of a memory controller for off-chip memory

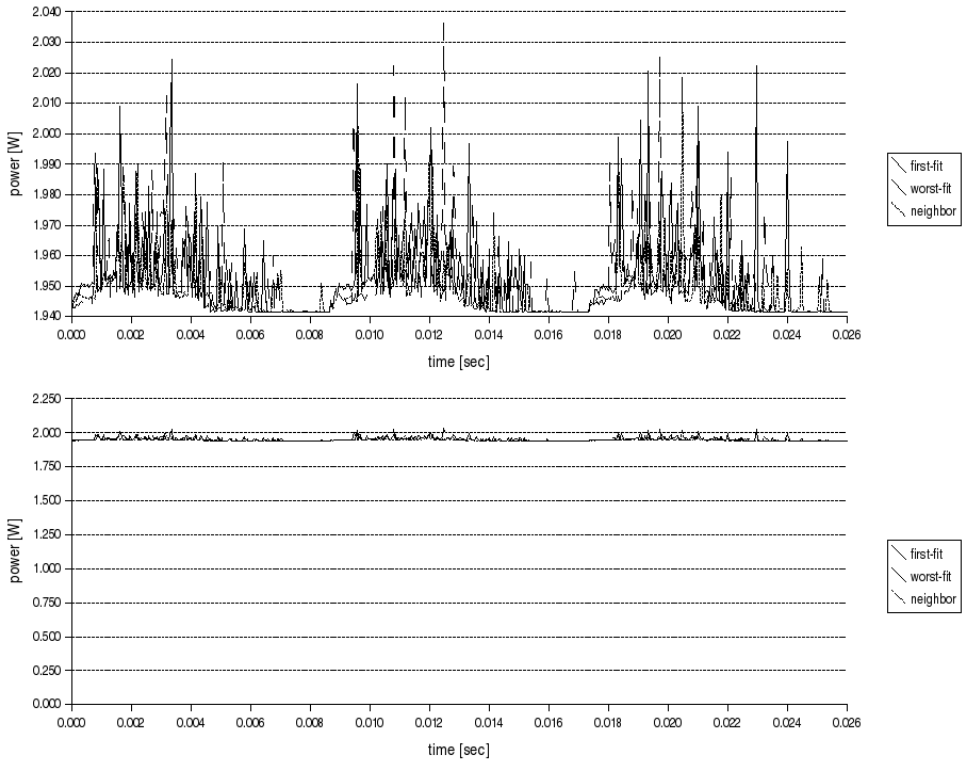

Figure 15.2.: Different allocation algorithms: network power consumption, consumed by a 3x3 network (2GHz) and periodic task repetition (3 periods). The top figure uses a scaled version of the bottom graph!

and the task's code loading from that external memory.

With that extra network usage the energy consumption raises, as expected. The comparison can be seen in figure 15.3. The top graph is the energy consumption using the memory task emulation and the lower graph is the same experiment without using the memory task.

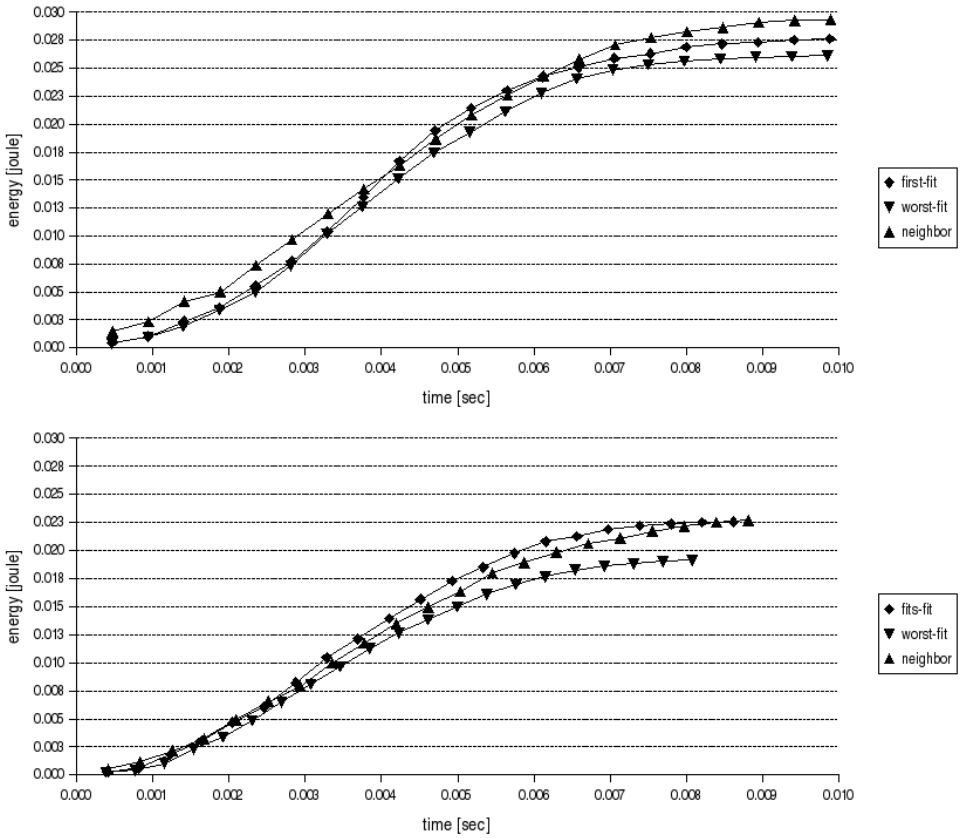

Figure 15.3.: The top graph's experiment makes use of the memory task (from section 5.4) and the lower graph is the same scenario without a memory task emulation. Both graphs display the total energy consumed by all general purpose processing elements.

But the higher energy consumption is not the only effect of the simulation detail. It also causes the experiments to take longer to finish. In figure 15.4 this is expressed by using the energy per time factor ($\frac{\text{total consumed energy}}{\text{total consumed time}}$).

Clearly the memory task effect adds considerable baggage to the system, but other then in this example I want make use of those feature because with growing network size the effect grows too and that leads to the situation where my computer runs out of memory and so I had to remove the memory task emulation from my experiments.

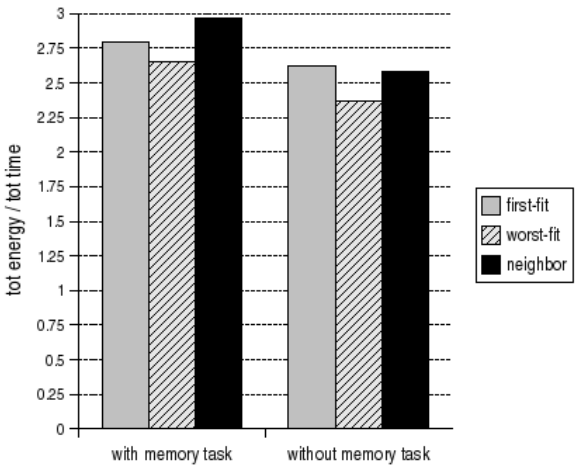

Figure 15.4.: The difference between the left and right values is caused by the added memory task emulation.

15.3. Operating system computation time influence

Similar to the previous section, this experiments handles the influences caused by the extra computation resources cost of executing the operating system service.

Each time a task has used all of its designated processing time the operating system service gets a chance to handle requests (system calls like sending network data or handling of an interrupt). Usually we exclude those cycles that the operating system needs for doing it's work, but the simulator allows the definition of the needed cycles for each operating system action.

Due to complexity and time issues I had to reduce that functionality to a simpler model: Each time the operating system gets activated it consumes 200 clock cycles on the local processing element regardless the action it is performing.

The resulting growth of energy need can be seen in figure 15.5-top. Compared to the energy consumption without that extra cycles cost (same figure, bottom) the difference is about 20%.

That is a considerable amount of energy and it I will use this effect in all experiments further on. Another consequence of this result is that the operating system service seams to use a significant amount of processing resources (see 15.6 for energy per time cost).

Therefore I am going to limit the maximal use of a processing element by tasks to 75% regarding the maximal capacity and the master in the first-fit and worst-fit allocation algorithm gets its own processing element. No other tasks are executed on the master's processing element.

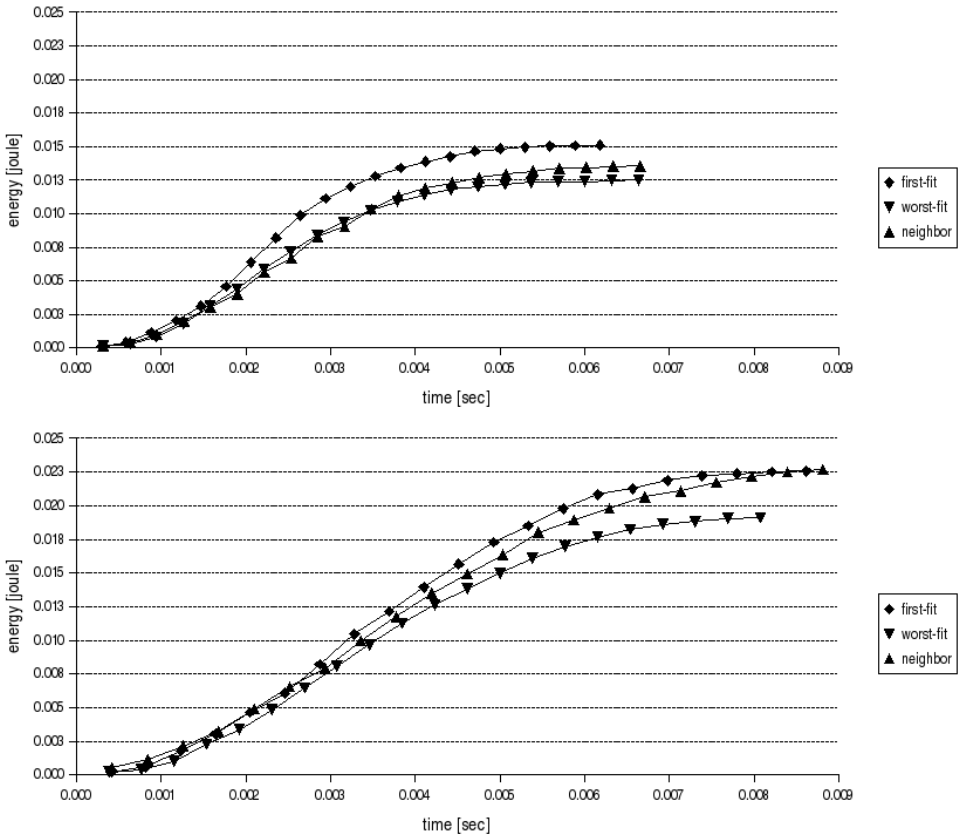

Figure 15.5.: The top graph's experiment does not include the costs for the operating system service execution and the lower graph is the same scenario with the costs included (200 cycles each call). Both graphs display the total energy consumed by all general purpose processing elements.

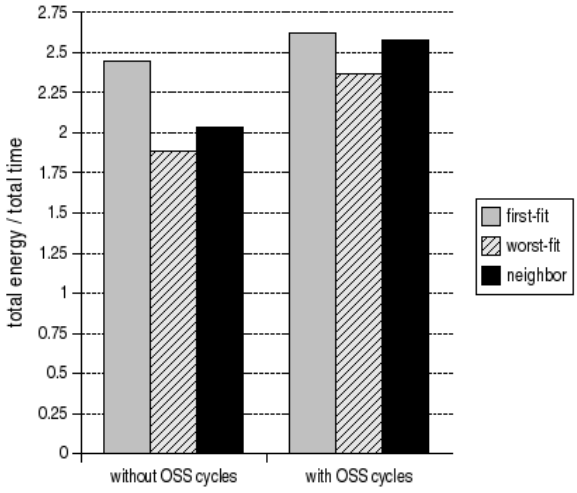

Figure 15.6.: The difference between the left and right values is caused by the added operating system computation cycles.

15.4. Allocation algorithm influence

Finally we are now taking a look at the different allocation algorithms. Two of them (the first-fit and worst-wit algorithm) use a centralized component (the master) and the third algorithm, the neighbor algorithm, comes without a central management.

To see now which algorithm is better I performed nine experiments. For each of the three network sizes 3x3, 5x5 and 7x7 I searched for a suitable work scenario and then let each allocation algorithm try to solve the simulation.

The outcome of the energy readings can be viewed in figure 15.7. There we have the three network sizes (from top to bottom: 3x3, 5x5 and 7x7) and for each network the three outcomes of the different algorithms.

From this figure we can easily see (and the cost factor in figure 15.8-top confirms it) that the worst-fit algorithm has the lowest energy readings and costs, followed immediately by the first-fit algorithm. The benefit of the worst-fit master can be explained by the strategy which the algorithm uses. It keeps the overall processing usage as low as possible. Given the configuration used by the experiments this pays of.

The neighbor algorithm on the other hand has by far the highest energy readings. This is not totally unexpected, considering the fact that without a central management the single allocators need to do more communication to gather all needed informations. And this results obviously in a considerable higher energy consumption.

Looking at the per node cost in figure 15.8-bottom we can see even better that the worst-fit and first-fit costs per node remain more or less the same. In this figure it is even more obvious that the neighbor allocation algorithm causes the use of a lot more energy then the others do.

To go a little into details here I show the sent bytes diagrams in 15.9. For each location the (by the operating system and the tasks) sent network messages are recorded and the final byte count is shown in the diagram. In each diagram we see two matrices (representing all the network nodes from 0,0 to max,max). The matrix more to the front represents the bytes sent by the operating system and the matrix more in the back show the task related sent bytes. The first row of diagrams has a network size of 3 by 3. The middle row has 5x5 and the bottom row a 7x7 network.

What we can see from this figure are two main results. First the master from both the first-fit and worst-fit allocator cause way more traffic then the other nodes (considering just the operating system part, of course). It seams that this tends to get more sever as the network size grows. At least the rest of the nodes have nearly no traffic compared to the tasks traffic volume.

For both master based algorithm we can see that with bigger networks the amount of traffic caused by the allocation communication protocols at the master's location gets more then the traffic caused by the tasks themselves at the rest of the network nodes. This result is not nice at all.

The neighbor allocator shows instead what it's decentralized system does cost. It does not have a single point of traffic concentration but it has a lot of overall distributed traffic that also seems to grow with the network size.

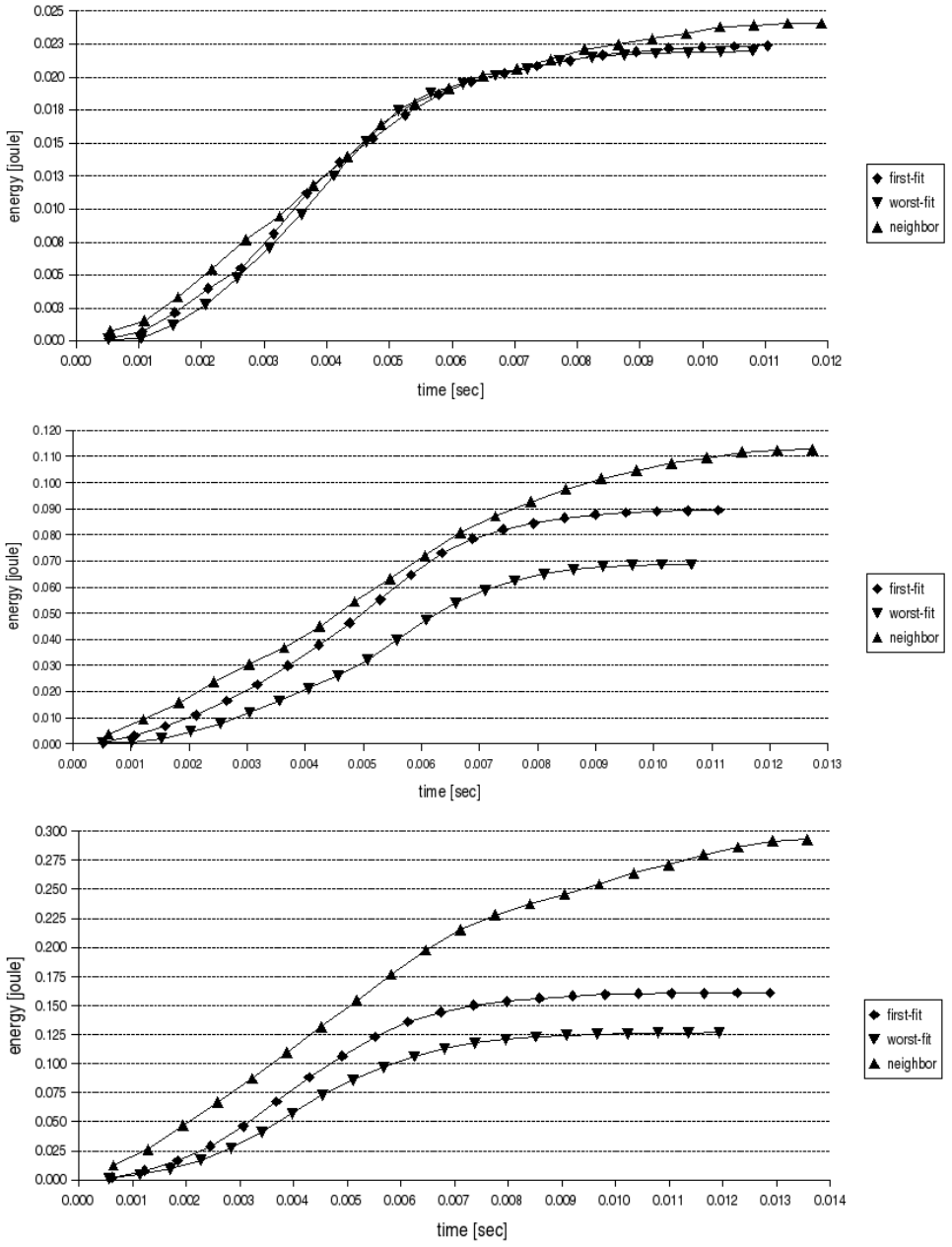

Figure 15.7.: The top graph shows a 3 by 3, the middle graph a 5 by 5 and the bottom graph a 7 by 7 network energy consumption for each different allocation algorithm.

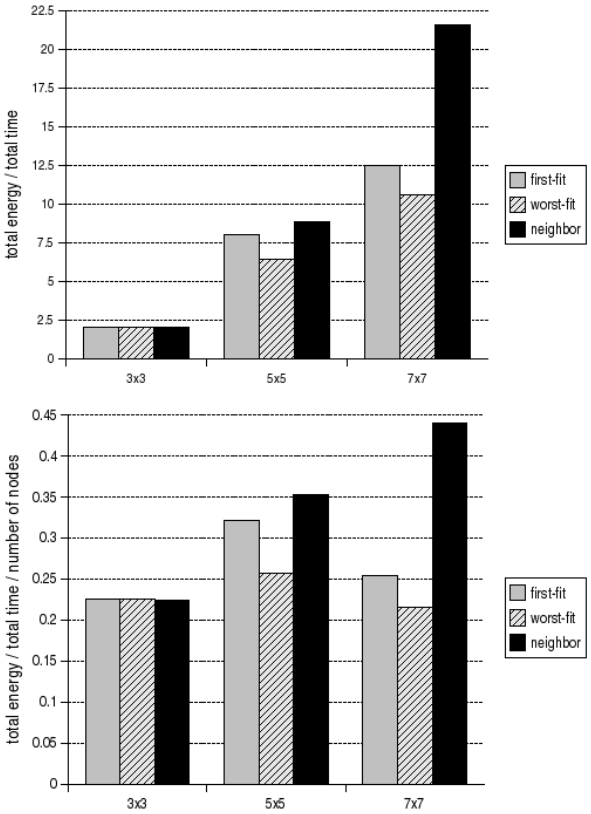

Figure 15.8.: The top graph shows the total energy per total time cost for each allocation algorithm, the bottom graph does the same, but this time the costs are also divided by the count of nodes of the corresponding network.

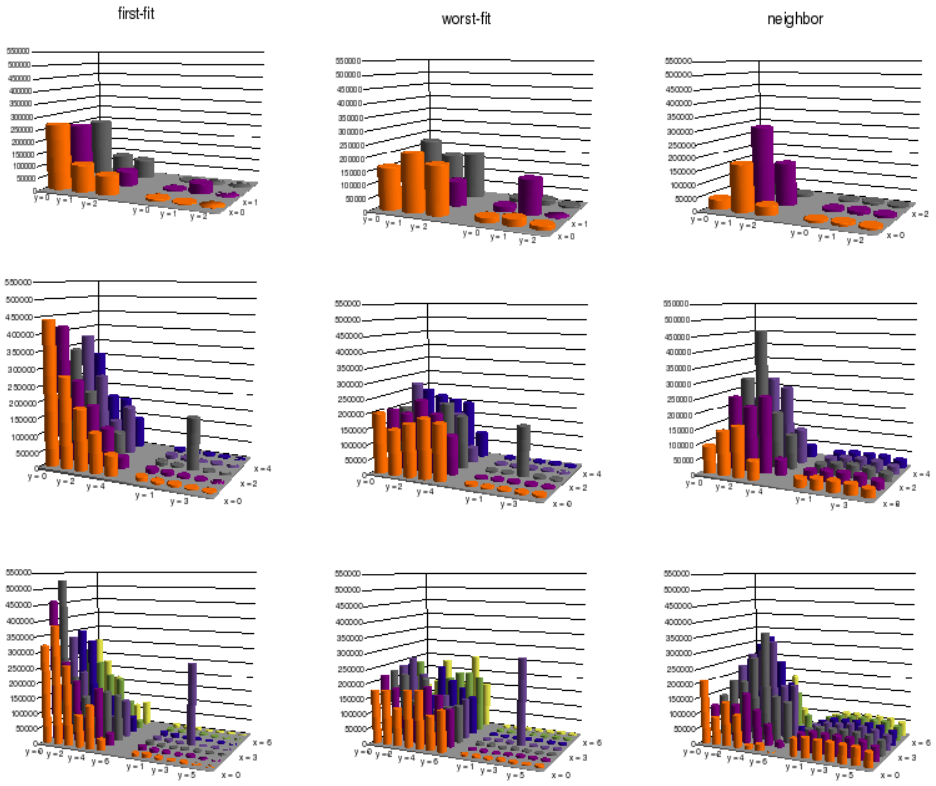

Figure 15.9.: Each pillar shows the total amount of traffic, emitted by the corresponding component of the specific network location. The pillars are organized as a matrix corresponding to the mesh network. Each diagram contains two times the same network. The matrix more to the front always shows the outgoing traffic of the operating system components (the allocator communication) and the matrix more to the back shows the outgoing traffic of all tasks together. The value of each pillar is the summary of all message sizes during the simulation. The unit is byte. Mind the same scaling of all diagrams.

To see what happens with the processing resources I show the corresponding diagrams in figure 15.10. The diagrams are organized in the same manner as the previous ones, only this time they show how much processing cycles the operating system (allocator, scheduler and service request handling) and how much cycles the tasks consumed on each processing element.

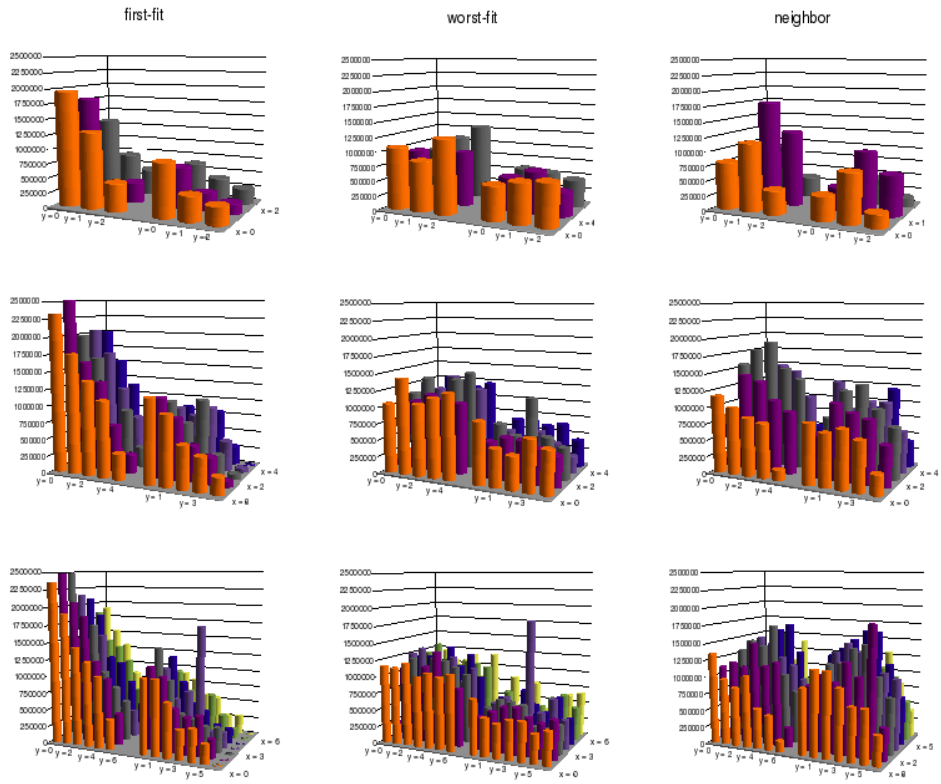

Figure 15.10.: Each pillar shows the total amount of cycles, consumed by the corresponding component of the specify network location. The pillars are organized in the same manner as in figure 15.9, meaning the pillars form the matrix of a mesh network. Each diagram contains two times the same mesh network. The matrix more to the front represents the cycles consumed by the operating system on each processing element and the matrix more to the back shows for each same processing element the amount of consumed task cycles. The unit of each pillar are the consumed processing element's clock cycles during the simulation, in other words the amount of executed instructions if the simulator could simulate that. Mind the same scaling of all diagrams.

It shows that the neighbor allocator really uses a lot of resource and with that a lot of processing element energy as well, but the other two algorithms are not so far behind, at least in using processing resources. And in this figure we can also see very well that the master component of the first-fit and worst-fit algorithm really needs its own processing element to function properly.

15.5. Periodic task execution

To sort of verify the previous seen results I performed one last experiment. I used the 3 by 3 network and its work scenario and this time activated the periodic repetition of the tasks. This means each time that the work scenario is completed the execution of all task starts over again.

The resulting energy reading can be found in figure 15.11. This reading show what we already guessed, the worst-fit algorithm has clearly the lowest energy expense. Surprisingly the first-fit is very close to the neighbor solution. This can be explained by the small network (and work scenario) size, used for this experiments. As the results from the previous section show, the first-fit algorithm's tendency goes more in the direction of the worst-fit allocator.

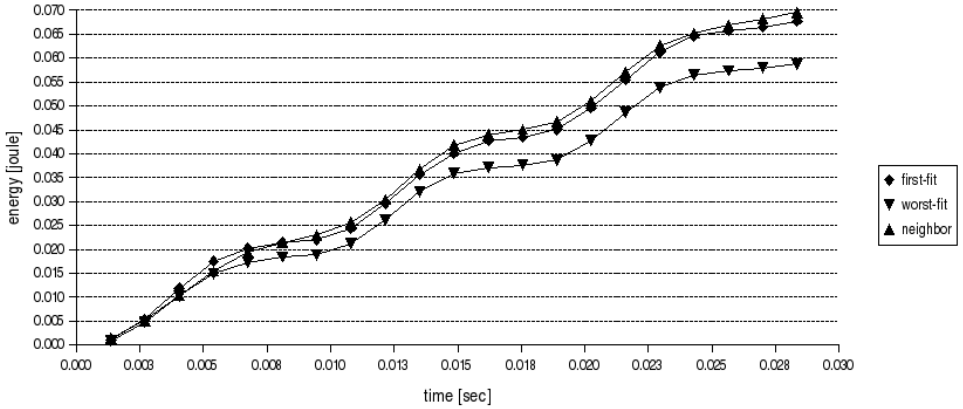

Figure 15.11.: Three periods where completed during this experiment on a 3 by 3 network. The graph shows the by all processing element together consumed energy, depending on the used allocation algorithm.

All in all this last experiment confirms the previously obtained results. The worst-case algorithm has the lowest energy readings and the neighbor algorithm the highest, due to too much resource use. However the relative differences between the algorithms are less significant then the influences created by for example the memory task (see section 15.2) or operating system cycles consume (see section 15.3). This is a really important fact since the allocation algorithms where originally designed to preserve energy. More about it will be explained in the next chapter 16.

16. Conclusion

I think the most important conclusion of this master thesis is the fact that the various influences from different sides (simulation realism) can change the energy consumption considerably. That means if we make too much (simplifying) assumptions about the system the so developed load balancing methods might in the end not save that much energy as predicted or even increase the energy usage.

In fact the results from the previous chapter show that for example alone the resources that are consumed by an eventually needed operating system can use more energy then any allocation algorithm can save.

Another interesting fact is the energy consumption by the network, which grows at a constant (but not exactly constant) rate. Initially the overall low usage of the network made me do a double and triple recheck on my simulator's implementation, before I realized that it is really just the local use of the network that requires such a fast clock period[1].

It is because of this behavior that the network energy consumption can in fact not be influenced by a different allocation of tasks. At least it is so in my simulations.

The influence of the systems dimensions (how fast processing elements and the network are, etc.) is very high and by using energy saving components or by using just slower components more energy then with a better allocation algorithm can be saved.

This might also require the use of smarter software (that makes for example a better use of the network capacity), but in terms of energy saving it is worth it.

Lowering the overall system requirements, and make it more tolerant regarding delays, can also have more effect then energy saving through load balancing.

Some words to the success of the worst-fit algorithm during the experiments. Its low energy readings are due to the strategy to keep the overall processing element usage as low as possible, which in the case of the used processing element properties payed off. But already the use of processing elements that have different power levels for the speed steps than the ones used in the experiments or even less speed steps to begin with can take away the small advantage of the allocator's savings.

On the other hand the neighbor allocation system makes too much use of system resources (both network and computation resources) to be a good substituted for a master based system[2].

[1] I started from 100 MHz and arrived at 2GHz, until the simulation worked.

[2] At least with the algorithm implementation that I used. Other implementations might improve the results. Especially the communication protocol can be made more efficient, I guess.

Considering the biggest shown network of 7 by 7 processing elements the results indicate that the master based load balancers reached there limit. The network usage was already as high as the tasks network usage and the processing resource usage was even higher.

This means with the settings used during the experiments of this master thesis the master based allocation algorithms will become a bottle neck in networks with sizes above 7 by 7.

Then an algorithm such as the neighbor allocator might be the only solution, because a too occupied master will slow the system down, and then when the usage goes up even more make it impossible for the system to work properly.

The big conclusion: Provided the level of simulation details used by this master thesis different allocation strategies have not that much effect on energy saving than previous studies might have indicated and with growing network size the master based solutions become unusable, such that decentralized algorithms such as the neighbor allocator are necessary.

17. Future work

Of course at this point it is possible to say what could have been done better and what possibilities show up now that the work is done. Well, some possibilities offered them selfs already during the implementation and were included in SNS but then never got used, because there was not enough time. Others where planed from the beginning, but again there was not time to try them, so we will take a look at it now; what is left to do and what can be possibly done to continue this work.

One thing needs to be mentioned here and that is that the aim of SNS was to be as realistic as possible, but at the end it comes down to choose the right parameters. So for any future work it is recommended to first check the realism of the used configuration before changing the simulator's implementation.

17.1. Possible Experiments

The simulator as it is offers lots of possibilities for new experiments. The important ones are those, which consider additional effects from simulation details on the overall energy consumption. So far the memory task, operating system cycles and of course the network size and different allocation algorithms have been presented.

But there are more components that influence the energy readings. For example one interesting experiment could be to vary the amount of the different work scenario models. Just use pipeline models and see if it is different to just using tree models. Maybe different allocation algorithms are affected differently by different combinations of scenario models as well.

Then the size of the work scenario (the amount of tasks) as well changes the energy consumption considerably. It would be interesting to see if the difference between the allocation systems changes when the scenario size gets reduced. As more and more processing elements are left without work the energy behavior of the allocation algorithms might change a lot.

Another interesting experiment could bee to see how the whole task parameter configuration together with the scheduler and operating system configuration can influence the experiment's outcomes. Meaning what happens if different kind of application requirements or (operating) system settings are applied. Are the characteristics for the different allocation algorithms still the same?

You probably can guess where this leads too. Nearly each configuration possibility allows a whole new set of experiments, provided we ask the right questions.

17.2. Inhomogeneous processing elements

Till now all processing elements had the same properties. They all have the same speed steps the same power consumption and can execute all tasks within the same time. In a real System on Chip (SoC) environment that is of course not always the case. In fact it is most common that not all resources can even execute all tasks.

To simulate those circumstance each processing element can be configured differently. The speed steps can be changed and even the execution of a certain type of task can be slowed down or be speeded up (as already used for the memory task, section 5.4). Such a possible inhomogeneous processing element configuration can be seen in figure 17.1.

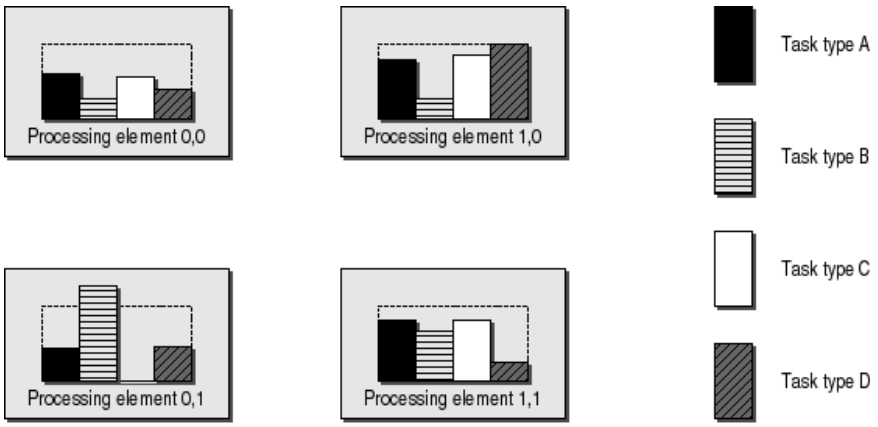

Figure 17.1.: Each processing element can execute a task type at a different speed compared to the possible standard execution time. The different factors are represented by the bars within the processing element blocks, corresponding each to a different task type. Processing element 0,1 shows the case where a type of task cannot be executed at all (type C) and the case where tasks (type B) can be executed even faster then usual.

The four task types of this example cannot be executed on just any processing resource, nor are the energy costs the same. The used allocation system has to be aware of the later issue to still be able to allocate each task to the right processing element and minimize the overall energy consumption. In fact as [Mey06] shows, with different types of processing elements the energy aware static scheduling of tasks can save a huge amount of energy. In the dynamic case the savings might not be that significant[1] but for a start the allocator must be aware of the inhomogeneous of the system such that an allocation is possible, even if it is just to ensure that a task can at least be executed at all.

SNS already supports the configuration of different task type execution speeds and also different speed steps for each processing element. The support for different power usage or power models is however not included in the current implementation.

[1] Or might they even be more?

Anyway I think that the consideration of an inhomogeneous SoC is the most significant way to continue this work. It adds a whole deal of realism to the simulation and it requires a whole new set of allocation algorithms with new important possibilities to save energy.

17.3. Hierarchical allocation system

The results showed that the master based allocation system is in fact the best solution. Its only draw back is the resource bottle neck at the central component. The solution is not very scalable.

To overcome this issue it is thinkable to use several master components, instead of a single one. Each hierarchical lower master controls just a fraction of the network. If requests for other network parts should be handled, a local master just forwards the request tho the next hierarchical higher allocation system which in turn can also be master controlled, but that is not compulsive.

An example of a combination of the master based and the neighbor allocation system can be seen in figure 17.2. There each local master controls a 3 by 3 part of the network and if a master wants to communicate with other masters it uses the neighbor allocation system.

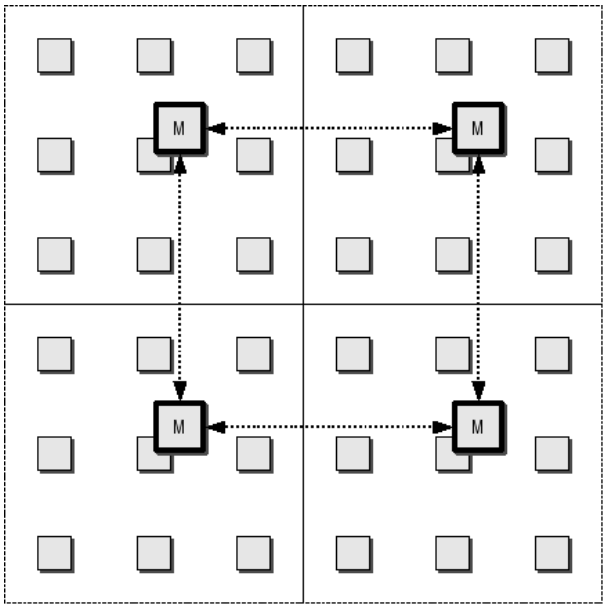

Figure 17.2.: Instead of using one single master this example uses for master components. Each master controls a corresponding 3x3 section of the whole 9x9 network. The master components itself can be organized via the neighbor allocation system, as shown in the figure. Of course they can also be controlled by an additional global master (not shown).

The combination of different allocation systems to cancel out the disadvantages from each one, certainly is a very important way for getting better allo-

cation algorithms. The hierarchical arrangement of allocation systems is just one example what might be possible.

The simulator itself does not provide any special means for a hierarchical allocation structure. Instead it is possible to configure SNS in such a way that for each processing element a different allocator gets used. As long as the different allocators can communicate with each other and organize them selfs, any allocation system is possible.

17.4. SNS refinements

Besides all the above mentioned possibilities the simulator alone could need some improvements as well. Considering that SNS tries to be as much realistic as possible, there are still lots of issues to improve.

And besides the realism the simulator implementation needs a better memory management as well. For example a $7x7$ network simulation with about 600 tasks and a network frequency of 2 GHz needs with all reports activated more then 2 GB of computer memory. The main memory consumption is all the data that is stored during the simulation and processed only when the simulation is over. With small changes this data could instead be cached to a file and the memory usage would be reduced to around 150 MB which then allows the simulation of bigger networks.

The only limitation then would be the time, which it takes to complete a simulation. For the above mentioned $7x7$ network it takes for example about 15 minutes to complete a simulation.

However to improve the realism of SNS two main changes (extensions) are recommendable:

Simulated memory management

Besides the simulation for an initial (off-chip) memory access now memory usage is simulated. For once each processing element might have some local amount of memory. The use of this local memory does not have an immediate effect on the task simulation, but considering that the local memory amount is limited it would make sense to also limit the amount of tasks for that processing element.

Each task would then make use of a certain amount of local memory and when it is all used up, no more tasks can be executed on the processing element, even if there are enough processing resources for another task.

The second memory related simulation details is about the off-chip memory access during a tasks simulated execution and at it's end. A task might want to access some information or data that is not stored in the local memory nor has it been fetched during the tasks initial memory access. So the task accesses off-chip memory during its execution time, which can be simulated similar to the memory task mechanism (see section 5.4).

Of course at a tasks termination, the tasks might want to send some data to an off-chip memory first, which also can be simulated by a mechanism similar to the memory task's one.

More system calls during a task's execution period

As it is now each time a task can make use of the processing element only one system call can be made. That is of course highly unrealistic. The solution to that is probably a bit tricky.

A simple way to overcome this issue would be to buffer system calls in the operating system service and at the time that the operating system can process it executes them all at once. Still this would not be very realistic. The operating system service should handle each request immediately when it is made.

This however would probably require to first change the task model to be able to do actions more then ones during a tasks execution period. Then the operating system must be reprogrammed as well such that it can handle an request immediately. All in all a significant big part of the SNS's implementation must be changed to achieve that kind of realism.

Implementation optimization

As last point of the further work chapter I would like to mention that the implementations of the different simulated parts of SNS also significantly influence the energy readings. Components like the network layer, transport layer or allocation system communication protocol influence the behavior of the overall simulation a lot.

For example currently the transport layer generates a communication overhead of 6 bytes versus 10 bytes data load. Reducing this ration to for example $4 : 12$ would shorten the average communication delay, speed up the work scenario execution and reduce the energy consumption. It might even change the the properties of the allocation algorithms.

Alone this small example shows what potential the simulated components implementation might withhold.

With this hopefully inspiring words I conclude the master thesis.

Stockholm, March 2006

A. Additional results

As the report chapter (see chapter 13) mentions, SNS generates a lot of results. Until now only the energy reports, the consumed-cycles-per-location reports and the bytes-per-location-traffic reports were shown.

To judge the quality of the experiments from the presented results you might want to take a look at the other results of the same experiments. Unfortunately the overall data is just too much to be presented in the report. So if you want to see all results you might take a look at the result tables, attached to the simulator code (result directory).

As for now, the following section present for each report an example, how the data from one of the experiments looks like.

A.1. Bytes traffic example

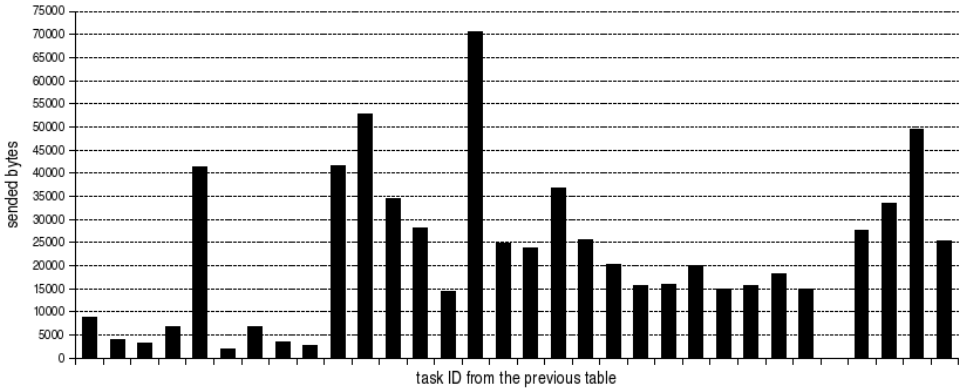

Figure A.1.: Sent bytes from each operating system service and task. The bars correspond to the entries from the next table A.1.

Task's ID	*X*	*Y*	*Total sent bytes*
0	0	0	8711
0	0	1	4076
0	0	2	3266
0	1	0	6869
0	1	1	41295
0	1	2	1879
0	2	0	6675
0	2	1	3501
0	2	2	2700
1	0	0	41497
2	1	0	52900
3	1	2	34627
4	0	0	28098
5	2	1	14293
7	0	0	70683
8	2	2	24735
9	0	2	23801
10	2	0	36844
11	0	1	25599
12	0	0	20202
13	1	0	15618
14	1	0	16063
15	2	1	20015
16	0	1	14896
17	0	0	15723
19	1	0	18350
21	1	0	14896
23	0	0	54
25	0	0	27756
26	1	0	33510
27	2	0	49588
28	2	1	25443

Table A.1.: The bytes, sent by the corresponding task during the simulation. The X and Y columns represent the network location of each task. The tasks with the identification 0 are the operating system services on the corresponding location. The data comes from the 3x3 network size experiment using the first-fit allocator.

A.2. GPPE cycle length changes

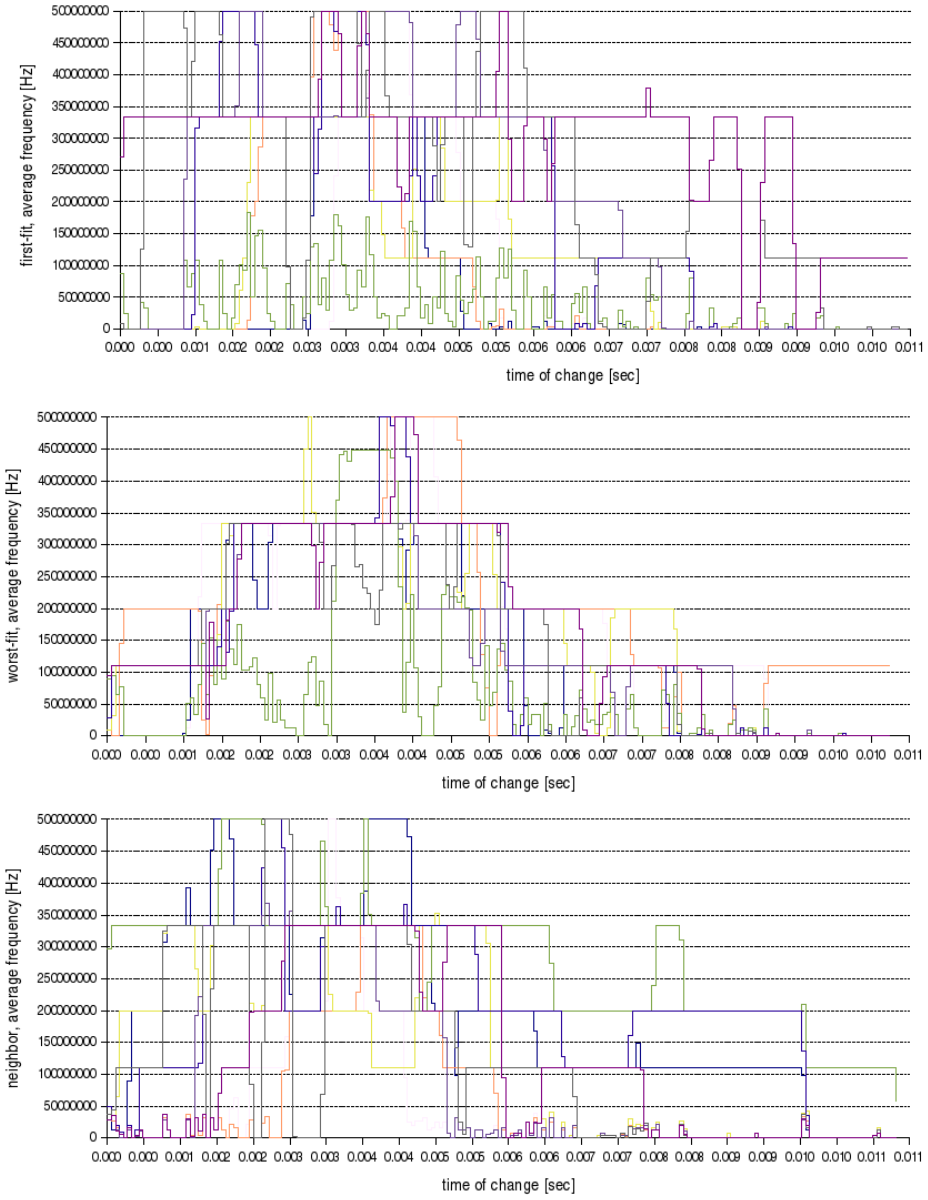

Figure A.2.: Average frequency (per interval) for each processing element from the 3x3 network size experiment.

A.3. GPPE power

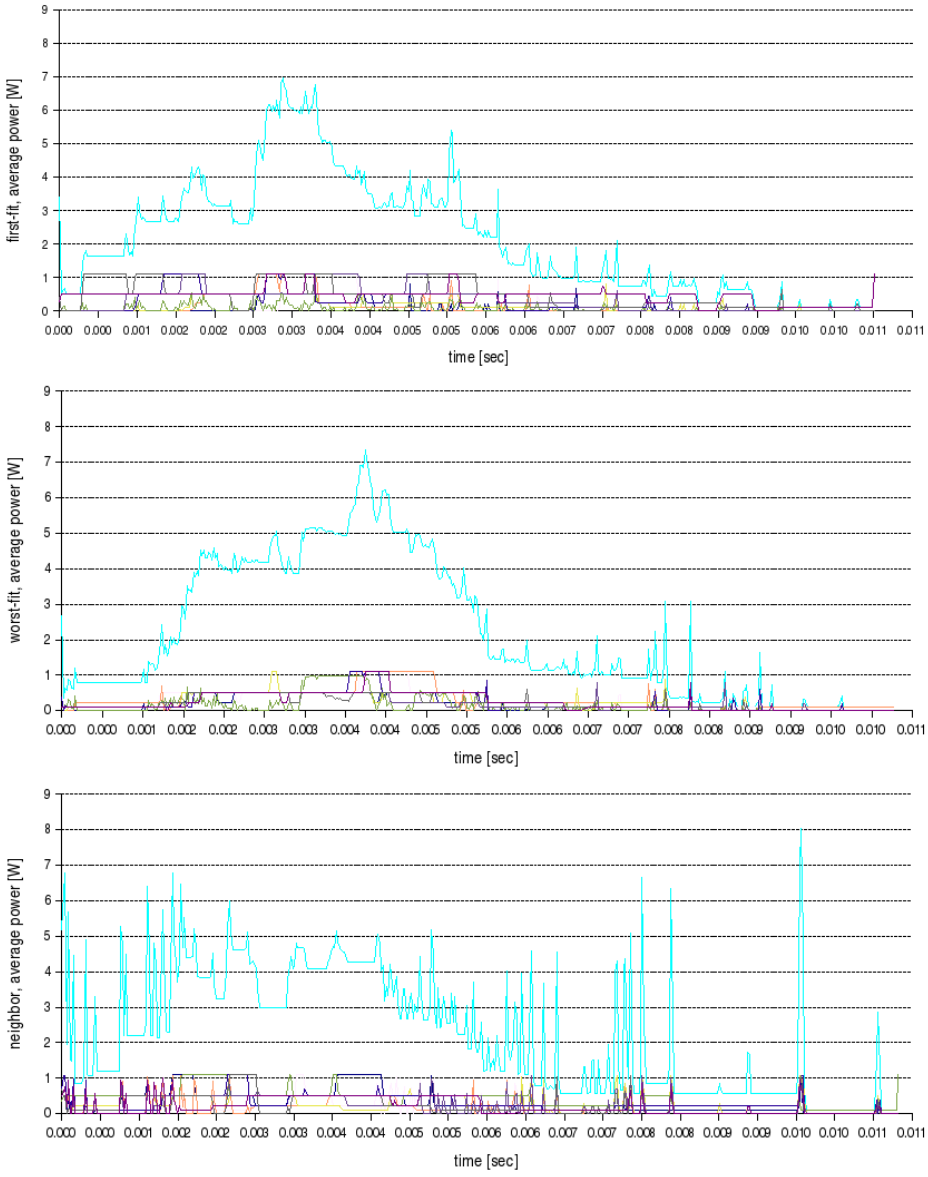

Figure A.3.: Power readings of all processing elements from the 3x3 network size experiment. The 10th (highest) power reading is the overall GPPE power usage.

A.4. Network power

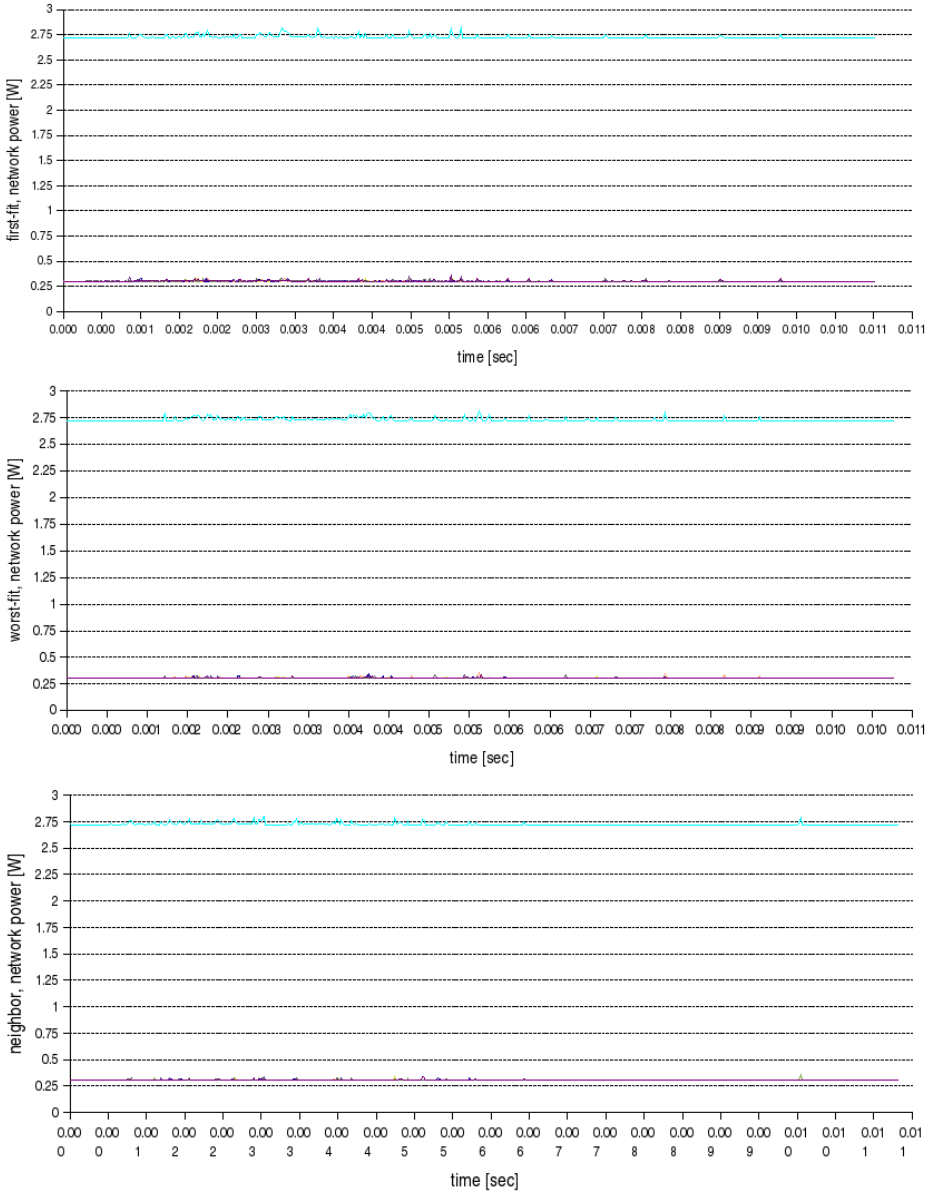

Figure A.4.: Power usage of all network nodes from the 3x3 network size experiment. The highest reading is the overall network power usage.

A.5. Traffic per time

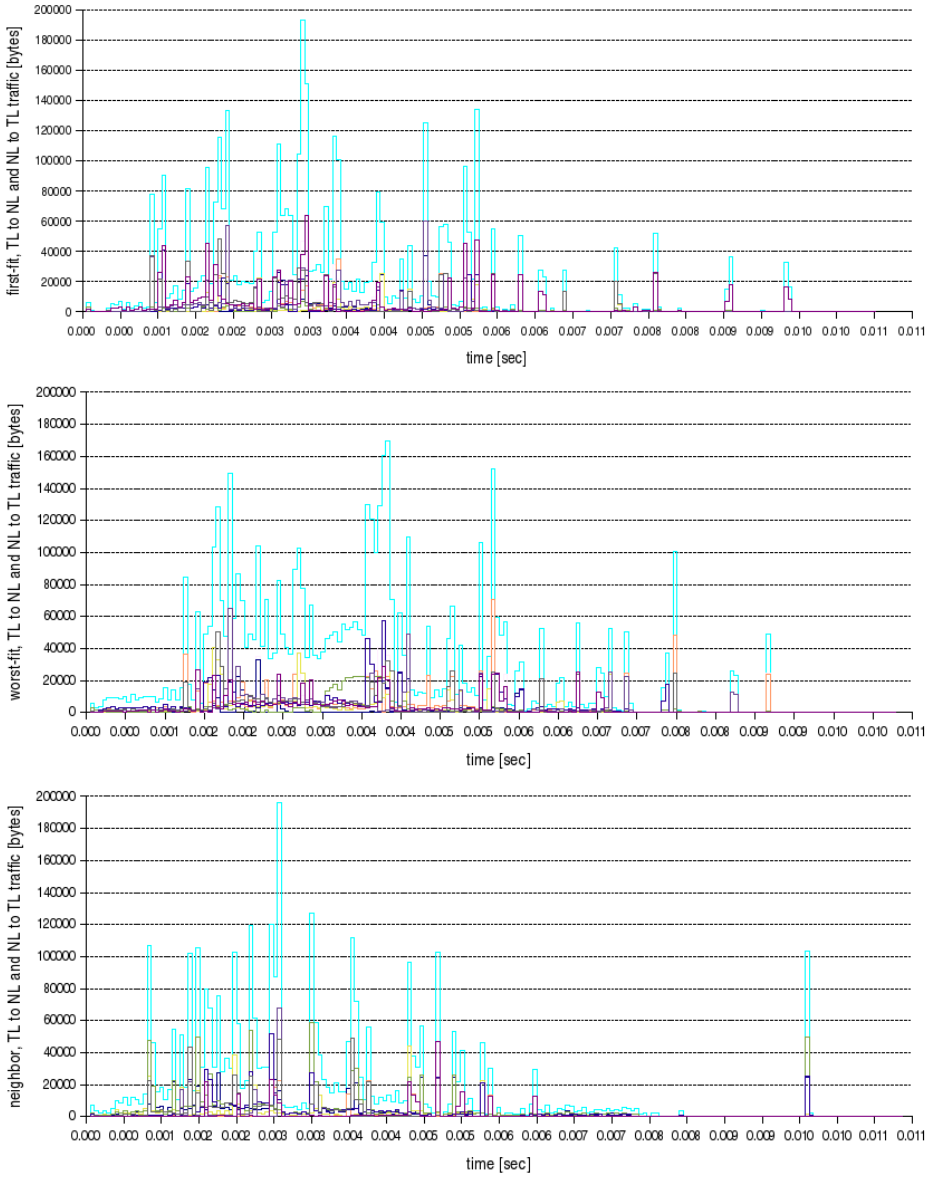

Figure A.5.: TL to NL and NL to TL traffic for each network node. The 10th and highest traffic reading is the sum over all nodes. The data comes from the 3x3 network size experiment.

A.6. Traffic per location

Algorithm		$x = 0$	$x = 1$	$x = 2$	*traffic type*
first-fit	$y = 0$	467680	330768	292560	*NL to TL*
first-fit	$y = 1$	241984	68464	148400	*NL to TL*
first-fit	$y = 2$	177104	126128	161024	*NL to TL*
first-fit	$y = 0$	448560	403568	396832	*TL to NL*
first-fit	$y = 1$	184768	73040	146080	*TL to NL*
first-fit	$y = 2$	130608	100912	129744	*TL to NL*
first-fit	$y = 0$	916240	734336	689392	*total*
first-fit	$y = 1$	426752	141504	294480	*total*
first-fit	$y = 2$	307712	227040	290768	*total*
worst-fit	$y = 0$	315200	314832	308800	*NL to TL*
worst-fit	$y = 1$	324144	232400	251104	*NL to TL*
worst-fit	$y = 2$	504960	171968	307904	*NL to TL*
worst-fit	$y = 0$	296352	335632	385152	*TL to NL*
worst-fit	$y = 1$	404896	224992	300384	*TL to NL*
worst-fit	$y = 2$	329536	159376	294992	*TL to NL*
worst-fit	$y = 0$	611552	650464	693952	*total*
worst-fit	$y = 1$	729040	457392	551488	*total*
worst-fit	$y = 2$	834496	331344	602896	*total*
neighbor	$y = 0$	175520	265840	80944	*NL to TL*
neighbor	$y = 1$	244000	359344	229232	*NL to TL*
neighbor	$y = 2$	92992	263280	84864	*NL to TL*
neighbor	$y = 0$	76464	238928	96000	*TL to NL*
neighbor	$y = 1$	311360	513584	173296	*TL to NL*
neighbor	$y = 2$	70192	281904	34288	*TL to NL*
neighbor	$y = 0$	251984	504768	176944	*total*
neighbor	$y = 1$	555360	872928	402528	*total*
neighbor	$y = 2$	163184	545184	119152	*total*

Table A.2.: Traffic in bytes for each location. The data comes from the 3x3 network size experiment. For each type of algorithm the traffic readings from NL to TL, TL to NL and the sum of both are displayed.

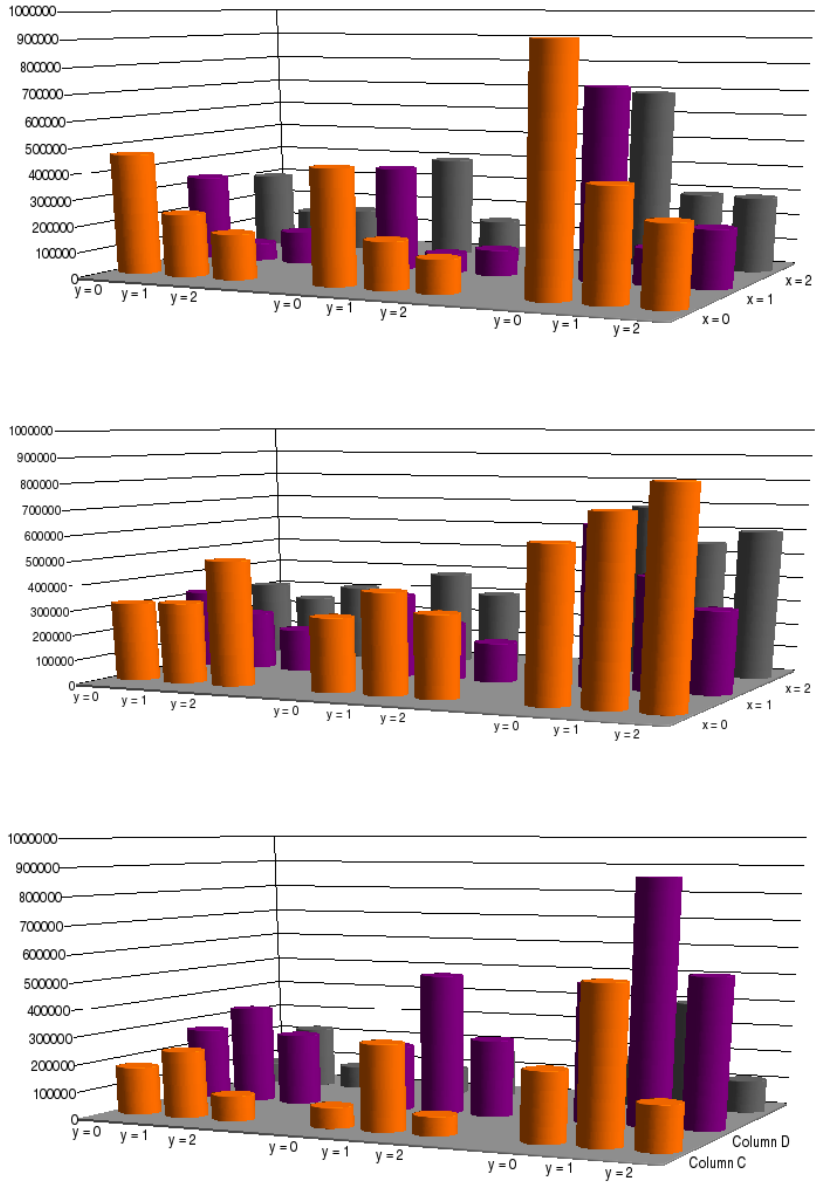

Figure A.6.: TL to NL and NL to TL traffic for each network node. The pillars form up the data presented in the previous table A.2. Each diagram contains the three matrices (TL to NL, NL to TL and the sum of both the traffic types) for each allocation algorithm.

A.7. Task life cycle example

Task's ID	*Creation*	*Start*	*End*	*Deadline*	*Termination*
1	0.01	0.01	2.46	-0.49	2.56
2	2.47	2.53	3.17	0.64	3.19
3	2.47	2.56	3.91	-0.12	3.94
4	3.17	3.21	5.15	0.03	5.17
5	3.91	3.95	4.94	0.13	5.17
6	5.15	5.21	7.19	-0.35	7.19
7	0.01	0.01	2.85	-0.86	2.93
8	2.86	2.89	3.83	0.14	3.85
9	2.86	2.93	4.67	-0.4	4.7
10	2.86	2.95	4.75	-0.19	4.77
11	2.87	2.99	5.04	-0.47	5.07
12	3.83	3.85	5.15	-0.17	5.16
13	4.67	4.71	6.03	0.17	6.06
14	4.75	4.77	5.37	0.58	5.39
15	5.05	5.07	7.02	-0.11	7.05
16	5.15	5.17	5.76	0.9	5.77
17	6.03	6.18	7.54	0.19	7.57
18	5.37	5.4	7.01	0.17	7.03
19	7.02	7.07	8.5	0.02	8.55
20	5.76	5.86	7.86	-0.07	7.88
21	7.54	7.59	9.28	-0.04	9.31
22	7.01	7.07	8.22	0.16	8.24
23	8.51	8.55	8.94	0.92	8.95
24	7.86	9.32	10.51	0.36	10.51
25	0.01	0.01	1.57	0.1	1.61
26	1.57	1.59	2.21	0.88	2.22
27	1.57	1.62	3.87	-0.26	3.88
28	1.57	1.62	4.27	-1.13	4.29
29	2.21	2.26	4.19	0.07	4.21
30	3.87	3.95	5.17	0	5.18
31	4.27	4.3	6.32	-0.05	6.34
32	4.19	6.34	7.6	0.14	7.6
33	0.3	0.3	1.7	0.53	1.77

Table A.3.: Absolute values of each task's creation, start, end, and termination time. The deadline time is calculated by $t_{start}+t_{exec}-t_{end}$. All time values are in milliseconds. The data comes from the 3x3 network size experiment with the first-fit allocator.

A.8. Tasks locations

Algorithm	*0,0*	*1,0*	*2,0*	*0,1*	*1,1*	*2,1*	*0,2*	*1,2*	*2,2*
first-fit	25	33	51	47		28	36	3	43
first-fit	1	58	40	48		35	55	62	8
first-fit	7	50	52	59		54	61	42	37
first-fit	45	46	27	53		5	9		
first-fit	60	26	29	56		15			
first-fit	34	49	41	11					
first-fit	57	2	10	30					
first-fit	4	63	38	32					
first-fit	12	31	44	16					
first-fit	39	13	6						
first-fit	18	14							
first-fit	20	19							
first-fit	17	21							
first-fit	22								
first-fit	24								
first-fit	23								
worst-fit	25	1	7	45		33	58	50	40
worst-fit	28	60	27	47		48	26	35	59
worst-fit	34	36	8	2		3	46	61	10
worst-fit	9	56	51	11		43	52	62	29
worst-fit	63	5	57	12		31	42	32	41
worst-fit	4	44	18	37		53	14	15	49
worst-fit	38	20	21	19		30	6	55	13
worst-fit	16					22	17		54
worst-fit							24		39
worst-fit									23
neighbor	42	33	59	40	25	58	37	50	49
neighbor	29	46	35	36	1	47	54	48	55
neighbor	38	34	61	41	7	60	31	51	63
neighbor	32	26	5	2	45	27		28	
neighbor	39	3		10	52	62		53	
neighbor	6	11		4	43	44		8	
neighbor		57		14	9	30		56	
neighbor				18	15			13	
neighbor				22	12			17	
neighbor					19			21	
neighbor					16				
neighbor					23				
neighbor					20				
neighbor					24				

Table A.4.: The location where a task (ID) was executed. The order of the tasks within a location represents the creation order. The data comes from the 3x3 network size experiment.

B. Network Energy at 100 MHz

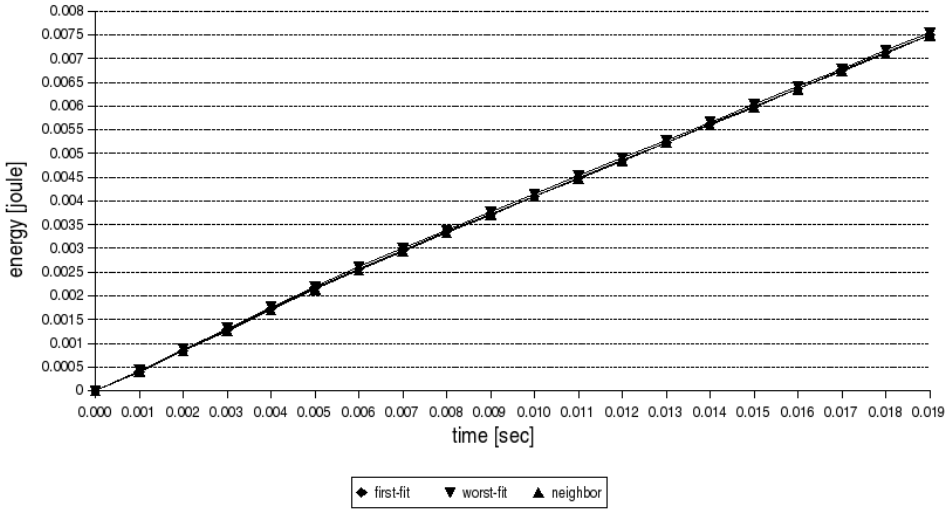

Figure B.1.: Network energy of an unsuccessful 5x5 network experiment. The diagram shows all three allocation algorithms. The network clock speed of the experiment is 100 MHz. Even with lower frequency the communication energy influence is not exactly noticeable (see section 15.1).

C. Glossary

Term	*Acronym*	*Explanation*
Allocator		Algorithm/module which assigns a task to a PE.
Comma **S**eparated **V**alues	**CSV**	File format (RFC 4180), used for all result files in SNS.
Cycles		Clock cycles, clock period or speed. Represents (one) instruction that is executed on a GPPE within one clock period.
Data **L**ink **L**ayer	**LL**	The network layer which has to correct and detect possible transmission errors from PL.
Dynamic **V**oltage **S**caling	**DVS**	Adjustment of the voltage (and frequency) to operate a processor at a different power level.
Earliest **D**eadline **F**irst	**EDF**	Scheduling algorithm which selects the task with the nearest deadline to be executed.
First-**F**it Allocator	**FF**	Allocation algorithm which assigns a task to the first suitable PE.
General **P**urpose **P**rocessing **E**lement	**GPPE**	A Processing element which can execute all possible kinds of tasks.
Identification	**ID**	Usually a number that identifies an object uniquely within it's kind.
Input-**O**utput	**IO**	
International **O**rganization for **S**tandardization	**ISO**	
Moving **P**icture **E**xperts **G**roup	**MPEG**	
MPEG-1 Audio Layer **3**	**MP3**	
Neighbor Allocator	**N**	Allocation algorithm which has no single global component.

Network **L**ayer	**NL**	Also network switch. Layer within the network which knows about the topology and does the message delegating work.
Network Layer(s)		Refers to all different layers within a network, or to one of them. Each layer plays a specific role for the network's work.
Network **O**n **C**hip	**NoC**	A Fully functional stand-alone network on a single chip, connecting different components of the chip.
Network Resource		1) Traffic/Load capacity of a network. 2) Resource, which is connected to the network (e.g. a PE).
Open **S**ystems **I**nterconnection	**OSI**	Network and communication reference model, defined by ISO.
Operating **S**ystem **S**ervice	**OSS**	Operating system parts which are executed on each GPPE in SNS.
Physical **L**ayer	**PL**	Hardware interconnection of a network.
Processing **E**lement	**PE**	It It is a part of a SoC, which can execute tasks. For example a microprocessor.
Processing Resource		Also amount of processing cycles, instructions or clock periods. It refers to the amount of instructions per time that can be executed.
Round-**R**obin	**RR**	Scheduler which assigns each task a certain (equal) amount of execution time.
Simulation **E**environ**M**ent for **L**ayered **A**rchitecture	**SEMLA**	NoC simulator.
SoC with **N**oC **S**imulator	**SNS**	Simulator, developed within this master thesis.
System **O**n **C**hip	**SoC**	A SoC denotes the integration of all the system's components on a single chip.
SystemC		Hardware modeling language, based on C++.
Task		Set of instructions/part of a program. Basic execution unit in SNS.
Transport **L**ayer	**TL**	Top NoC layer in SNS. Splits a message into suitable fragments.
Very-**H**igh-**S**peed **I**ntegrated **C**ircuit	**VHSIC**	
VHSIC **H**ardware **D**escription **L**anguage	**VHDL**	
Worst-**F**it Allocator	**WF**	Allocation algorithm which tires to keep the overall PE usage as low as possible.

Bibliography

[AMD] AMD.
http://www.amd.com.

[AY03] Hakan Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. Computer Science Department George Mason University, 2003.

[BDM02] L. Benini and G. De Micheli. Networks on chip: a new paradigm for systems on chip design. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 418–419, 2002.

[BVC04] Nilanjan Banerjee, Praveen Vellanki, and Karam S. Chatha. A power and performance model for network-on-chip architectures. Department of CSE, Arizona State University, 2004.

[Con] Embedded Microprocessor Benchmark Consortium.
http://www.eembc.org.

[fSI94] International Organization for Standardization (ISO). Iso 7498-1:1994(e) open systems interconnection - basic reference model, 1994.

[Homa] NOSTRUM Project Homepage.
http://www.imit.kth.se/info/fofu/nostrum.

[Homb] SystemC Project Homepage.
http://www.systemc.org.

[Ini02] Open SystemC Initiative. *SystemC User's Guide*. Open SystemC Initiative, 2002.

[Int] Intel.
http://www.intel.com.

[JT03] Axel Jantsch and Hannu Tenhunen. *Networks on Chip*. Kluwer Academic Publishers, 2003.

[LJ05] Zhonghai Lu and Axel Jantsch. Traffic configuration for evaluating networks on chips. Laboratory of Electronics and Computer Systems, Royal Institute of Technology, Sweden, 2005.

[Lu05] Zhonghai Lu. *A User Introduction to NNSE*, 2005.

[Mey06] Mathias Meyer. Energy-aware task allocation for network-on-chip architectures. Master's thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2006.

[Pan01] Preeti Ranjan Panda. *SystemC A modeling platform supporting multiple design abstractions*. Synopsys Inc., 2001.

[Pen05] Sandro Penolazzi. An empirical power model of the links and the deflective routing switch in nostrum. Master's thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2005.

[Sha05] Y. Shafranovich. *RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Network Working Group, 2005.

[Syn04] Synopsys. Synopsys power compiler - user guide, release v-2004.06, 2004.

[Thi02] Rikard Thid. A network on chip simulator. Master's thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2002.

[Thi03] Rikard Thid. *SEMLA tutorial*, 2003.

[Tho05] Bjarke Thormann. Modeling of dynamic resource allocation in a network on chip. Master's thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2005.

[TMJ03] Rikard Thid, Mikael Millberg, and Axel Jantsch. Evaluating noc communication backbones with simulation. Technical report, Royal Institute of Technology (KTH), 2003.

[ZLNJ05] Mikael Millberg Zhonghai Lu, Rikard Thid, Erland Nilsson, and Axel Jantsch. Nnse: Nostrum network-on-chip simulation environment. Technical report, Royal Institute of Technology, Sweden, 2005.

List of Figures

List of Tables