Eötvös Loránd University

Department of Information Systems

DISTRIBUTED SURVIVABLE PIPELINE COMPUTATION AND COMMUNICATION

PhD Dissertation

Zsolt Palotai

Supervisor: Dr. habil. András Lőrincz

Head Senior Researcher

ELU, Department of Information Systems

PhD School of Informatics

Dr. János Demetrovics

PhD Program of Information Systems

Dr. András Benczúr

Budapest, 2007.

Abstract

The increasing availability of mobile and/or intelligent sensors with computation and wireless communication capabilities makes distributed computer networks more and more heterogeneous and complex. At the same time, in distributed systems several units can work on the same task simultaneously, making execution quicker and more reliable. In high level synthesis (HLS), there are well-tried concepts and procedures to accomplish a complex pipeline task on the immobile network of various hardware units. In my dissertation I am introducing a distributed self-organizing protocol along with its main components. The protocol is able to set up and maintain the execution of pipeline tasks in heterogeneous dynamic networks. The protocol has four special features which originate from the fact that I've implemented the concepts and procedures over the hardware-software co-design on networks, combining them with the newest achievements of machine learning: i) scalability, applicable also in heterogeneous dynamic networks; ii) natural ability to execute hierarchical tasks; iii) the units can process and forward data in the pipeline way; iv) survivability, that is it adapts to the constantly changing units of the network and maintains the functions while there are enough resources in the network.

Thanks

I met András Lőrincz in the spring of 1999. He has been trustingly patronizing my aims and patiently masterminding my researches ever since. I am grateful to him for helping, supporting, encouraging and directing me through these years. I am beholden to the former and present members of the research team: Ákos Bontovics, Bálint Gábor, Viktor Gyenes, György Hévizi, Melinda Kiszlinger, Katalin Anna Lázár, Sándor Mandusitz, Barnabás Póczos, Zoltán Szabó, Botond Szatmáry, Gábor Szirtes, István Szita and Bálint Takács. It was very pleasant and inspiring to work with them. We helped each other many times. Thank you all. Special thanks to Gábor Szirtes for commenting on my dissertation on a really constructive way. I am grateful to Péter Arató, Tibor Kandár, Zoltán Ádám Mann, Zoltán Mohr, András Orbán, Tamás Visegrády and Gábor Ziegler for supporting my work. I am much obliged to Attila Meretei for helping me set feasible aims for myself. I have to thank Pázmány Péter Foundation and John von Neumann Computer Society for supporting my work with their scholarships.

I am grateful to my grammar school teachers. I especially learned a lot from Andrea Gyengéné Beé and Tamás Sas.

I would like to thank my family for supporting me.

And last but not least, I thank Eszter for being by my side, loving and supporting me.

Contents

1 Introduction **1**

2 High Level Synthesis and Pipeline Operation **7**

2.1 Formulation of the HLS problem with IPs 9

2.1.1 Scheduling of elementary operations 10

2.1.2 Pipeline data processing 15

2.1.3 Satisfying restart time constraints 16

2.1.4 Allocation with IPs 18

2.1.5 Pipelined Operations 20

2.1.6 PO scheduling 21

2.1.7 PO restart time considerations 22

2.1.8 PO allocation using IPs 22

2.2 Conclusions 24

3 Weblog Algorithm in Scale Free Small Worlds **25**

3.1 Related work 27

3.2 Forager architecture 29

3.2.1 Algorithms 30

3.2.2 Reinforcing agent 33

3.2.3 Foragers 34

3.3 Experiments 35

3.3.1 Data collection on the Web 35

3.3.2 Simulations 37

3.4 Discussion 42

3.4.1 General weblog algorithm 46

3.5 Conclusions 47

4 Survivable Pipeline Protocol **48**

4.1 P2P basics 49

4.2 Survivable Pipeline Protocol 50

4.2.1 Downscaled version for RF-MEMS devices 56

4.3 Experimental results 57

4.3.1 JXTA SPP 57

4.3.2 TinyOS SPP 58

4.4 Outlook and prototype application 62

5 Summary **64**

List of Figures

1 **The main steps of High-Level Synthesis (HLS).** 10

2 **Timing diagram of Extendend Elementary Operations.**
Inputs of the i^{th} EO, e_i, are stable during its working (execution time = τ_i cycles in this example). Output is stable after the working cycle. 13

3 **ASAP and ALAP concepts**
Illustration of As Soon As Possible and As Late As Possible greedy timing algorithms. Latency of pipeline computation is set to 6 in this example 14

4 **Concepts of restart time and latency time in pipeline operation.**
Operations can be started more frequently than their computation takes. (l=6) of the Control Data Flow Graph 17

5 **Resource occupancy time (ROT)**
The resource occupancy time q_i of EO e_i is equal to the execution time plus the maximum of the execution times of the successor EOs. 18

6 **Four types of concurrencies**
Two EOs can be concurrent in four different ways. Compatibility classes are defined as the **maximal sets** of non-concurrent EOs. . 19

7 **Definition of Intellectual Property (IP)**
An IP is defined by the EOs that it can execute and by its cost. The illustrative example shows the first IP (Π_1) with two EOs (op_1, op_2) and cost c_1. 20

8 **Scale-free property of the Internet domain**. Log-log scale distribution of the number of (incoming and outgoing) links of all URLs found during the time course of investigation. Horizontal axis: number of edges ($\log k$). Vertical axis: relative frequency of number of edges at different URLs ($\log P(k)$). Dots and dark line correspond to outgoing links, crosses and gray line correspond to incoming links. 36

9 **Degree distribution of the environments.**
Dots and dark line correspond to outgoing link distribution. X-s and gray line correspond to incoming link distribution. (a) Upper: degree distributions of SF1 environment. Lower: degree distributions of SF2 environment. (b) : degree distributions of SFRandom environment. 40

10 **Measured parameter values.**
The subfigures show the investigated parameters. Each subfigure contains the parameter values for the four type of foragers in four columns as shown below the bottom subfigures. The 4 different markers correspond to the measured parameter values in the 4 environments as shown in the top middle figure legend. On each marker an error bar shows the standard deviation of the corresponding parameter values for the 3 simulations. Mean age is in hours in the upper left subfigure. 44

11 **Example SPP task** 55

12 **Simulation of 30 motes** 59

13 **15 motes of real experiment in a circle** 60

14 **Calculated positions of real experiment with 15 motes in a circle** 61

List of Tables

1 **Extended Elementary Operation attributes in HLS** 12
2 **Investigated parameters** 41
3 **Quantitative results for algorithm–structure pairs** . . . 46
4 **JXTA SPP results** 57
5 **Startup and working times** 62

1 Introduction

The increasing availability of mobile and/or intelligent sensors with computation and wireless communication capabilities makes distributed computer networks more and more heterogeneous and complex. In these networks the high performance components, consuming high power, can communicate reliably with low latencies through high bandwidth channels. On the other hand the low power and/or low performance components, like small sensors and mobile units, can communicate unreliably with widely varying latencies through lower bandwidth channels. Also the tasks which have to be solved by these networks have varying bandwidth, latency, and quality of service requirements which must adapt to the capabilities of the networks. At the same time, in distributed systems several units can work on the same task simultaneously, making their execution quicker and more reliable. To do so, we have to decide what the certain processing units should work on separately to achieve the required processing speed and reliability jointly. There are two common resolutions used in distributed systems [10]: i) each unit complete the whole task but with different data or parameters; ii) if the task is easily divisible into independent subtasks that can be dealt with simultaneously, the units work on these separately. Both resolutions presume that the units are similar and can complete the given task with approximately the same efficiency. In the second case, the number of units that can work on the task is determined by the number of its parallel subtasks.

In high level synthesis (HLS), there are well-tried concepts and procedures to accomplish a complex pipeline task on the immobile network of various hardware units [3]. Here, the task is also divided into independent subtasks, operations. But these subtasks can be not only parallel but linear also, that is they can follow one another and the output of one operation can provide

the input of the next one. This enables different hardware elements to do the operation that suits them best and parallel processing does not restrain splitting the task into parts. This kind of procedure of operation division enables the so-called pipeline data processing. This way the system starts processing the next data before having finished processing the previous one. Accordingly, the throughput of the system improves (it can process more data in a given period of time) but its latency (the time it needs to process an item completely) does not change.

I've developed the so-called Survivable Pipeline Protocol (SPP), with the help of which the units of the network can implement the tasks in a self-organizing way. A task has to be described with a graph of operations along with the task's parameters of confidence (e. g. the task requires 1.5 times more resources than it should) and timing (maximum restart time, minimum restart time, maximum latency). The edges of the graph determine which operations take the input and give the output and which operation provides input for which one. With the protocol then units in the network can decide which unit should implement which operation without having to appoint central units in advance. In addition, with the protocol, the units that are involved in the implementation of the task adapt to the constantly changing network. For instance, if an operational unit falls out of the network or the computational/ communication capacity drops, the operating units will search for new units in the network until the task has enough resources or until the network runs out of resources. From now on I call the constantly changing physical network communication network.

I have implemented the protocol over the JXTA P2P network [33] and I present here my first results. I have found that the network's computational capacity scales linearly with the available computational power. This imple-

mentation can be used over the internet and as a grid software with an easy programming interface. I have downscaled the protocol to RF-MEMS units and implemented it for the Cricket Motes in NesC over the TinyOS event driven operating system [19]. The particular application I have developed for the motes is to calculate their relative positions to each other. I have found that at least 15 motes can work together in this task. In simulations I have shown that even 30 motes can work together.

The protocol needs an algorithm with the help of which the units can find and memorize the units they can effectively co-operate with on certain tasks. Over the communication network this establishes a new network which I'm calling trust network. The vertices of the trust network are units from the communication network. There is an edge between two units in the trust network if the above mentioned algorithm has found that they can efficiently work together on a task.

The problem of organizing task execution in a communication network can be seen as a resource constrained optimization problem. The measures to be optimized are the cost of used resources, the task execution speed and reliability. The resource constraints are, e.g. the finite number of units in the network, the finite communication channel bandwidth, the finite processing capacity of units. Studying more and more complex systems researchers find that certain systems create scale-free small world (SFSW) network among the units of the system [6, 1]. Such systems are for example: the Internet, the web, an electric network. Common features of these systems are: robust function, they optimize certain standards with limited resources. In these networks, selection effect influences connections and units. Connections or units that are not necessary any more, or do not co-operate efficiently with the other units of the network, cease in the end. On the other hand, the effec-

tive units establish more connections and efficient connections gain strength.

Using this experience (and assuming that self-organizing systems that I am examining establish scale-free (SF) worlds by the above mentioned constraints) from the start I've looked for such an algorithm that is effective in these networks. Furthermore, I have been examining the efficiency of the algorithm in scale-free networks (SF) that do not meet the small world (SW) criteria and in random networks as well. For developing and testing the algorithm, I looked for a real problem in which I can change the structure of the environment freely enough without changing other parameters of the problem. Such problem is when web crawlers have to find as many new items of news on the web from that day as they can [39]. After saving the data of a few weeks' execution, the link structure is easy to change without changing the availability, appearance and update of the documents. In this task it is not trivial to find the optimal solution, that is, the order of documents and the paths between them on the links of the pages that the crawlers have followed. Nevertheless, documents are easy to evaluate according to their contents and to the previously found documents. Under certain conditions, reinforcement learning finds an optimal solution for such tasks [48]. Experience shows that reinforcement learning achieves good results even on such tasks that do not fully accomplish the terms. This task does not accomplish the terms either but executions showed earlier that reinforcement learning has just as high or even higher efficiency than other crawler algorithms [24, 26]. One of the characteristics of small worlds is that they are clustered [54]. It means that in the network the neighbors of a vertex are very likely to be connected, so they are each other's neighbors as well. In scale-free small worlds I examined the operation of the crawlers that use reinforcement learning and I found that they get stuck in the clusters or can leave the cluster with difficulty. That

is why I developed the weblog update algorithm [36, 35, 39] that searches and memorizes the right start-up URLs for the search. Based on the experience mentioned above I used a simple selection method. I found that in scale-free small environments the simple selection algorithm performs better in itself than the others, while in scale-free and in random environments the algorithms which are combined with reinforcement learning are the most efficient and the efficiency of such algorithm depends the least on the structure of the environment. In the crawler experiments the valuable resources are the URLs from which the crawlers can find much new that day information. In the task execution problem the resources are the processing units. A processing unit is a good resource for an other processing unit if those can efficiently work together on a task. This algorithm enables the units to find the suitable partners to complete the tasks, just like the crawlers found the right start-off URLs.

The first obstacle of the thesis was the fact that the operation concept does not include pipeline operation in typical high level synthesis (HLS) [3]. Consequently, the already planned pipeline systems can be used restrictively in designing more complex systems in typical high level synthesis. Moreover, in scale-free systems it is practical if the task is describable in different scales with operations of various difficulties. To achieve this, we need to be able to describe the operation of any complex pipeline system with a single operation. That is why I have extended the operations of high level synthesis to pipeline operations and have shown how to use the algorithms of high level synthesis for these pipeline operations [38].

The organization of the dissertation is the following. In chapter 2 I introduce the concepts of High-Level Synthesis (HLS) and I describe the extension of elementary operations to pipeline operations. In chapter 3 I propose a se-

lection based resource finding algorithm, which is efficient in scale-free small world environments, and its extension with reinforcement learning is efficient in non small world environments, too. In chapter 4 I introduce a novel peer-to-peer protocol to organize and maintain computation and communication within a distributed, heterogenous, continuously changing network of computing, sensing, and communicating units. Finally, in chapter 5 I summarize the dissertation.

2 High Level Synthesis and Pipeline Operation

In general, the first step of any kind of design is to determine the components that will not be decomposed any further. I shall call such components extended elementary components, or extended elementary operations (EOs) [40, 12]. The word 'extended' is added to emphasize that the algorithms utilized are not limited to addition, subtraction, and multiplication, but higher order units, such as FFT, ciphering, etc., units can be considered as EOs. Restrictions on the properties of EOs define the type of design under consideration. One may say that the design concerns 'the compilation of EOs'. This formulation corresponds to 'silicon compilation', SICOM [18], when the HLS description is translated into layout in one step. Combinatorial explosion of the design phase, however, deprives SICOM at more complex design tasks. Time constraints and considerations on cost also direct HLS research and development to consider reconfigurable off-shelf elements that are already optimized. An element, that can perform different operations, is called Intellectual Property (IP) [46, 16, 20]. The number of available IPs is increasing on the market, and thus HLS should consider the optimization of IPs in the design phase. This is the 'allocation problem' that determines which EO will be executed by which IP.

The operation concept does not include pipeline operation in typical high level synthesis (HLS). Consequently, the already planned pipeline systems can be used restrictively in designing more complex systems in typical high level synthesis. Moreover, in scale-free systems it is practical if the task is describable in different scales with operations of various difficulties. To achieve this, we need to be able to describe the operation of any complex pipeline system with a single operation. That is why I have extended the operations of high level synthesis to pipeline operations and have shown how

to use the algorithms of high level synthesis for these pipeline operations. In turn, the method can deal with pipelining IPs as sub–units.

The chapter is organized as follows. First, an overview of HLS is given. I formulate scheduling, pipelining, restart time and the allocation problems in Subsections (2.1.1- 2.1.4). Pipeline operations are introduced in Subection 2.1.5. Conclusions are drawn in Section 2.2.

2.1 Formulation of the HLS problem with IPs

HLS, the process of transforming abstract (high-level) hardware descriptions into silicon, is widely practiced by the CAD community. The obvious method is called 'silicon compilation' [18], when a high-level system descriptionis directly transformed to transistor or layout-level. A direct transformation from a high-level description to silicon may have the unique advantage of easy testing and verification. However, in case of complex systems, heuristics guided support for verification can be substantial. Alas, silicon compilation leads to designs of larger silicon area and this overhead may be too high for practical usage, especially in large systems (see e.g., [53] and references therein).

Beyond silicon compilation, HLS is the automated design from an abstract high-level description to an advantageous intermediate-level connection-oriented RTL description of a system. The intermediate-level RTL is a description of structures, defined in terms of storage (registers), elementary functional units, and interconnects between storage elements, with the necessary control logic.

Most of the time, HLS problems are given as algorithms, or sample implementations in a sufficiently high-level programming language describing dataflow graphs in standard programming languages, typically C, C++, or, increasingly, Java [56]. Practical hardware–software implementations exist using C++ and Java descriptions of communication protocols, where the actual protocol representations remain invisible to the designer beyond class methods (expanded only at compile time from libraries) [52], [15].

HLS, which I will detail in the next subsection, starts with a dataflow–like description. The vertices of this dataflow–like graph correspond to particular EOs, whereas the edges correspond to data connections between the

EOs. The graph describes the propagation of the data between EOs. It also contains information that enables one to derive the control structure. This graph is called the Control–Data–Flow (CDF) graph. One starts from adapting EOs and defining the HLS properties of the EOs. The second step is the scheduling of EOs of the CDF graph. In turn, processor should be rendered to each EO. This last step is called allocation. Allocation, however, is strongly influenced by scheduling, whereas scheduling is restricted by the possibilities of allocation. The optimization problem is known to be NP–complete (see, e.g., [3] and references therein). Figure 1 depicts the main steps during HLS.

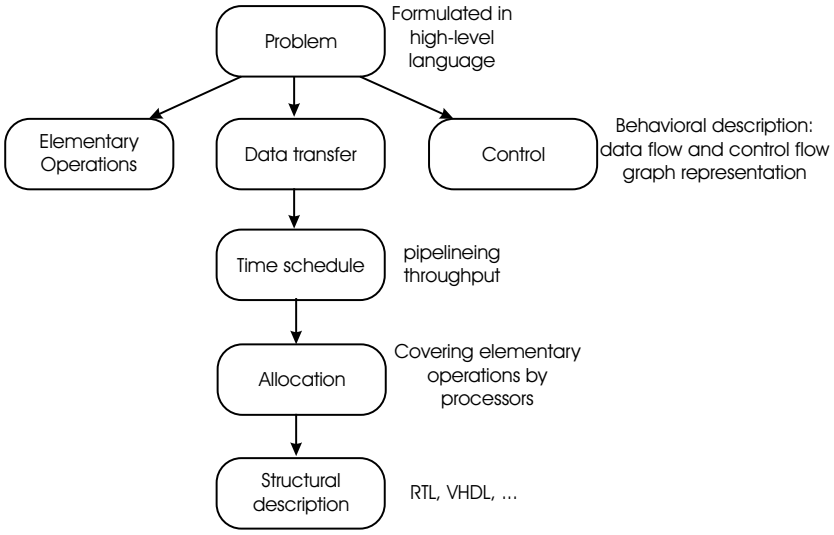


Figure 1: **The main steps of High-Level Synthesis (HLS).**

2.1.1 Scheduling of elementary operations

Elementary operations are functional operations that may be realized directly with one register-transfer level primitive. The operation model in traditional HLS assigns the following properties to the i^{th} elementary operation e_i (see

Table 1):

- **Timing** information is generally expressed in dimensionless units, time cycles, making designs scalable with technology [41]. Time cycles are numbered from 0, as usual.

- **Start time**, t_i, is the time (or time cycle) when operation e_i receives all its input data and may start processing it. By time cycle t_i all inputs are assumed to be available and stable.

- **Successor set**, S_i, a set of elementary operations that use the output value of e_i directly (immediate or direct successors).

  Direct successor: An elementary operation e_j is a direct successor of e_i if and only if at least one of the data inputs of e_j is the output of e_i. The following notation is used to denote direct successor relationships: $e_i \rightarrow e_j$. The inverse relationship is direct predecessor or immediate predecessor:

  Direct predecessor: An elementary operation e_i is a direct predecessor of e_j if and only if e_j is a direct successor of e_i.

  Using the above notation, the definition of the successor set of elementary operation e_i is $S_i = \{e_j : e_i \rightarrow e_j\}$. The successor set is empty if and only if the elementary operation supplies a system output (assuming there are no loops in the system). Similarly, a predecessor set is defined:

- **Predecessor set**, P_i, a set of elementary operations that are direct predecessors of elementary operation e_i: $P_i = \{e_j : e_j \rightarrow e_i\}$

- **Execution time**, τ_i, the total time required for the elementary operation to produce its output.

Property	Notation	Assigned during
Start time	t_i	scheduling
Predecessor set	P_i	graph generation
Successor set	S_i	graph generation
Execution time	τ_i	graph generation
Resource of operation type	ρ_i	graph generation
ASAP execution time	s_i	scheduling
ALAP execution time	l_i	scheduling

Table 1: **Extended Elementary Operation attributes in HLS**

Even if a considerable set of scheduling and allocation heuristics exists for constant-execution time systems, allowing different execution times for EOs increases the usefulness of HLS systems. The freedom to use different execution times for EOs allow to extend the HLS to hardware-software co–design/co–synthesis. Practically all models of real hardware-software co–design systems require different execution times for efficient designs.

EOs may start processing their inputs when all of their immediate predecessors have produced their output. In other words,

$$t_j \geq \max_{e_i : e_i \in P_j} (t_i + \tau_i)$$

The following additional conditions are applicable to timing calculations (Fig. 2):

- e_i requires all its input data to hold a stable value during the whole duration time τ_i.
- e_i may change its output during the whole duration time τ_i.
- e_i holds its actual output stable until its next start.

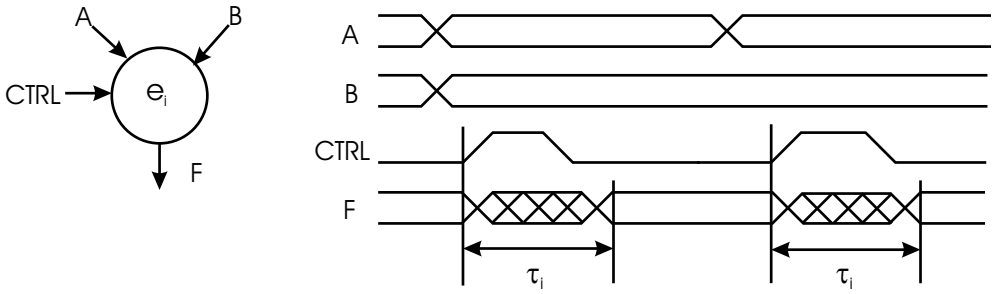


Figure 2: **Timing diagram of Extendend Elementary Operations.** Inputs of the i^{th} EO, e_i, are stable during its working (execution time = τ_i cycles in this example). Output is stable after the working cycle.

Execution time is assigned after partitioning, and may depend on data (operation) size as well as on operation type (ρ_i).

- **Resource of operation type**, ρ_i, identifies the resource requirement of elementary operations. Operations of the same type may be executed in the same kind of processor [41]. Note that the choice of operation types depends on the environment. In some systems, for example, multiplication and addition are performed in general-purpose ALUs, thus $\rho_+ = \rho_*$ for an addition and a multiplication. In most practical systems, multiplication and addition are separated to reduce cost; in these systems the operation types obviously differ for addition and multiplication vertices. The operation type attribute is used during scheduling and allocation, and is a property of the initial problem graph (i.e., it is fixed at the time of graph generation, and does not change later).

- **ASAP execution time**, s_i , denotes the first ('as soon as') possible time cycle elementary operation e_i may be started. Similarly,

- **ALAP execution time**, l_i , denotes the last ('as late as') possible

time cycle elementary operation e_i may be started.

ASAP and ALAP are the starting time cycles for the trivial greedy scheduling strategies. ASAP attempts to start execution of elementary operations as soon all their predecessors become available. ALAP delays starting an operation as late as possible, triggering the elementary operation at the last time cycle when system latency is not increased (3. Figure).

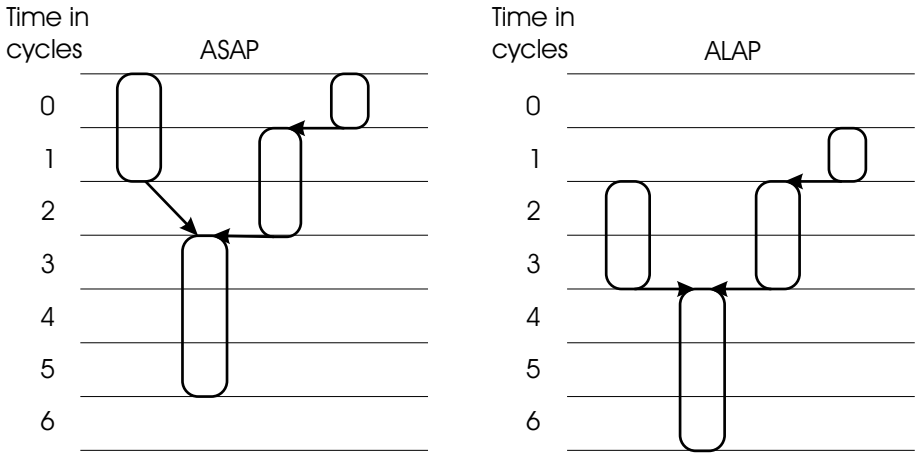


Figure 3: **ASAP and ALAP concepts**

Illustration of As Soon As Possible and As Late As Possible greedy timing algorithms. Latency of pipeline computation is set to 6 in this example

Using the above notations, the following timing constraints can be formulated in the system:

1. Every elementary operation must start execution not sooner than its ASAP time cycle, and not later than its ALAP cycle: $s_i \leq t_i \leq l_i$

2. No elementary operation may be triggered before every one of its predecessors has finished executing:

$$s_i = \max_{e_j : e_j \in P_i} (t_j + \tau_j)$$

3. ASAP times of elementary operations may be found by calculating the maximum of the sum of ASAP cycles and execution times of their direct predecessors. The resulting equation is the special case of the previous equation:

$$s_j = \left\{ \begin{array}{rl} \max_{e_i:e_i \in P_j}(s_i + \tau_i) & if\ P_j \neq 0 \\ 0 & if\ P_j = 0 \end{array} \right\}$$

Note that direct input operations have no predecessors, and their ASAP cycle is cycle 0 by definition. ASAP time cycles for other elementary operations may be found inductively based on this equation.

4. A similar equation holds for ALAP time cycles. In this case, the ALAP time of every elementary operation must enable the direct successor vertices to finish execution before their ALAP times. The ALAP times of elementary operations producing system outputs is fixed so that the last system output is stable by time cycle L, where L is a preset constant.

$$l_j = \left\{ \begin{array}{rl} \min_{e_i:e_i \in S_j}(l_i - \tau_j) & if\ S_j \neq 0 \\ L - \tau_j & if\ S_j = 0 \end{array} \right\}$$

Elementary operations supplying system output values are assumed to have an ALAP value set by system requirements if there is a latency (L) constraint. In systems without latency limits, all resource requirements may be fulfilled, since inserting sufficient delay would resolve all resource conflicts. Such a solution is not practical in most systems.

2.1.2 Pipeline data processing

Latency time provided by the CDF graph constrains data processing in traditional systems. However, the new input does not have to wait until the last EO belonging to the previous input is finished. Pipelining aims to start

the computation of the new input before the processing of the previous input is finished. This property is expressed by the restart time r. Restart time r tells us the minimum time span (i.e., the number of time cycles) between two inputs (Fig. 4). By decreasing r, the throughput of the system can be increased. One talks about pipeline operation when the restart time is smaller than the time of latency. That is, during pipeline operation, more than one inputs are processed parallel. HLS with EOs is restricted to components that do not feature pipeline operation and can process a single input at a time (Fig. 4). Later in Subsection 2.1.5, I resolve this restriction by extending the concept of EO to *pipelined operations* (POs). The generalization allows one to consider novel problems, including PC farms, code–morphing processors, etc.

2.1.3 Satisfying restart time constraints

It is assumed that during the execution of an EO of the CDF graph the input of the EO is not modified and it is the output of the EO that may change. It is also assumed that the output of the EO will not change until the start of all successor operations of the CDF graph. It then follows that unit i is occupied for time duration q_i and that q_i satisfies the constraint $q_i > \tau_i$. Time duration q_i is called resource occupancy time (ROT), whereas the time interval of ROT is called resource occupancy period (ROT period). ROT is equal to execution time τ_i plus the longest execution time of the successor EOs of the CDF graph (Fig. 5):

$$q_i = \tau_i + \max_{e_j : e_j \in S_i} \tau_j$$

Restart time r is to be ensured by the CDF graph. If execution time τ_i of EO e_i is smaller than r, but the belonging ROT is longer than r (i.e.,

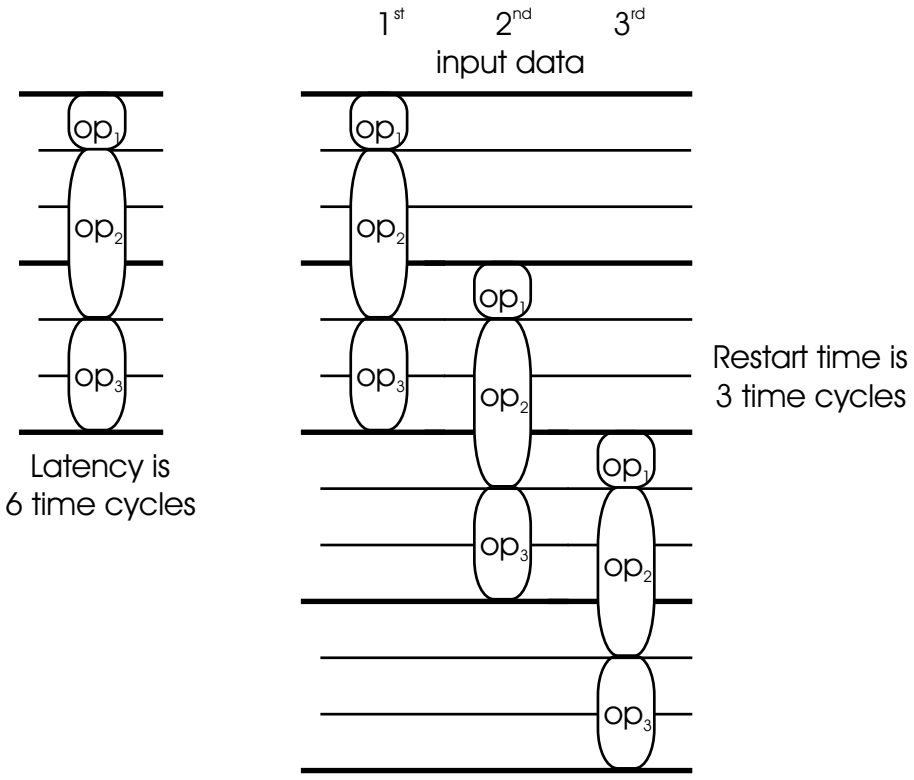


Figure 4: **Concepts of restart time and latency time in pipeline operation.** Operations can be started more frequently than their computation takes. (l=6) of the Control Data Flow Graph

$q_i + 1 > r$) then the constraint on the restart time is not satisfied. (Here, the extra +1 time cycle is included to exclude hazards and possible fast transients.) The solution is to insert an inexpensive buffer between the EO and its successor(s). Buffers have execution time 1 and can keep the data for arbitrary duration. In turn, the successor of EO e_i is the buffer and the ROT satisfies the following equation: $q_i = \tau_i + 1$. If $\tau_i + 2 > r$ then multiple and parallel working copies of the EO can solve the problem. If the number of such copies of the EO of the CDF graph, c_i is large enough then the following inequality holds: $q_i + 1 < r * c_i$. In turn, the constraint on the ROT becomes

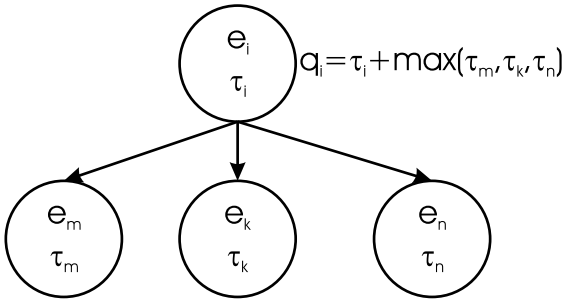


Figure 5: **Resource occupancy time (ROT)**
The resource occupancy time q_i of EO e_i is equal to the execution time plus the maximum of the execution times of the successor EOs.

satisfied. Each copy of the EO requires a front buffer. If there are more than one data for an EO then synchronization of the inputs is to be warranted.

Loops need special treatments. Minimal restart time gives rise to the following constraint for ROT $r \leq \tau(loop)+1$ where $\tau(loop)$ is the loop latency time. Conditional branching can be viewed as a special EO, a multiplexer, and does not require special treatment.

2.1.4 Allocation with IPs

The resource allocation is correct if (i) every EO belongs to a dedicated resource, (ii) if every resource is capable of performing the EO dedicated to it, and (iii) if there is no temporal overlap between the ROT periods of the EOs dedicated to an individual resource.

In order to consider IPs I proceed as follows. I introduce *compatibility classes*. A compatibility class m_l encapsulates EOs with non-overlapping ROT periods taken modulo restart time r (Fig. 6). A maximum compatibility class encapsulates *all* EOs that are all compatible with each other. Maximum compatibility classes will be denoted by capital letter M. I say that the EOs belonging to the same compatibility class are non-concurrent.

Expressed in mathematical terms: time intervals $[t_i, t_i + q_i] \mod r$ are pairwise disjunct for each EO e_i belonging to the same compatibility class and the class can not be extended by new EOs.

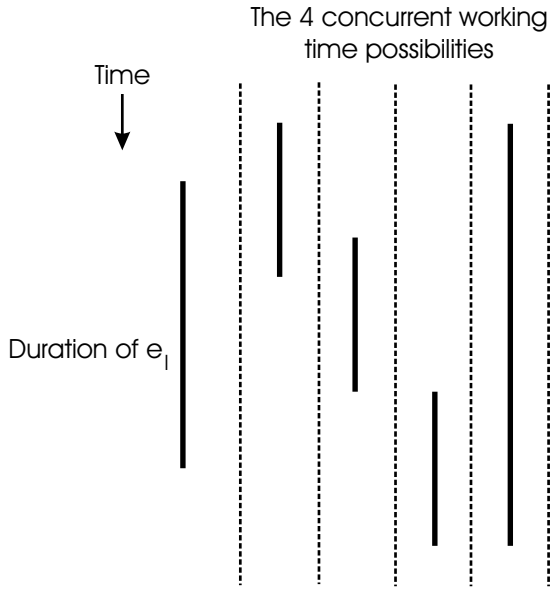


Figure 6: **Four types of concurrencies**

Two EOs can be concurrent in four different ways. Compatibility classes are defined as the **maximal sets** of non-concurrent EOs.

The i^{th} IP is denoted by Π_i. If EO e_k can be executed by IP Π_i then $\Pi_i(e_k) = 1$, otherwise $\Pi_i(e_k) = 0$. From my point of view, IP Π_i is defined by its cost c_i and by the set of EOs that this IP can execute $R_i = \{e_k : \Pi_i(e_k) = 1\}$ (Figure 7). Correct allocation is possible if sets R_i cover the set of the EOs of the CDF graph. Allocation can proceed by choosing pairs of sets, M_l and R_i and I render the i^{th} IP, Π_i to all the EOs belonging to the disjunction. Such steps are to be continued until no EO is left without an IP rendered. The objective of the optimization is the minimization of cost. Such minimization is possible because there are many ways to cover the set of EOs. A novel heuristic solution has been developed at the Technical

University of Budapest [3, 4].

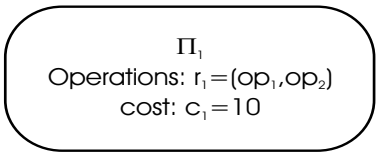


Figure 7: **Definition of Intellectual Property (IP)**
An IP is defined by the EOs that it can execute and by its cost. The illustrative example shows the first IP (Π_1) with two EOs (op_1, op_2) and cost c_1.

2.1.5 Pipelined Operations

Here, I generalize the concept of extended elementary operations to *pipelined operations* (POs): A PO, according to my definition, may consists of more operations. These embedded operations can be EOs, or POs or both. The description of a PO includes the description of its first EOs. For example, if the PO starts with a pipelined addition, then the description of this PO contains the description of the first EOs of the pipelined addition. These first EOs will be called the *front EOs*. One can define the *back EOs* in a similar fashion. PO p_i is defined by its algorithmic capabilities and by the following parameters:

τ_i **execution time (or latency time)** the number of time cycles that output is computed from the input.

r_i **restart time** the number of time cycles required between data

θ_i **input hold time** the number of time cycles that the input data should be kept stable after execution of this input has been started. The hold time is the maximum execution time of the front EOs of the PO. This

condition is sufficient provided that the PO is properly designed and provides stable input for all or its non-front EOs.

λ_i **last EO time** this is the maximum execution time of the back EOs of the PO.

Properties of non-front and non-back EOs can be left out from the PO description. Every data of every buffer is 1. The restart time of the full system is r_f.

2.1.6 PO scheduling

The concepts of Timing, Start time, Successor set, Direct successor, Direct predecessor, Predecessor set are left unchanged, except the notation of subsection 2.1.1 is to be modified: all e_i should be replaced by p_i for every i.

POs may start processing their inputs when all of their immediate predecessors have produced their outputs. In other words,

$$t_j \geq \max_{p_i : p_i \in P_j} (t_i + \tau_i)$$

The constraints to be applied to timing calculations for POs differ somewhat from the corresponding conditions of EOs:

1. p_i requires all its input data to hold a stable value during its hold time θ_i.

2. p_i may change its output during the time interval $[t_i + \tau_i - \lambda_i, t_i + \tau_i)$.

3. p_i holds its actual output stable until its last non pipeline operation start.

- **ASAP execution time**, s_i, denotes the first ('as soon as') possible time cycle when the first elementary operation of p_i may be started. Similarly,

- **ALAP execution time**, l_i, denotes the last ('as late as') possible time cycle when the last elementary operation of p_i may be started.

2.1.7 PO restart time considerations

In this subsection I consider constraints on the restart time.

Insertion of a buffer: Consider data processing in PO p_i. Consider the first and the last EOs of the PO. The execution time of these EOs are θ_i and λ_i, respectively. Assume that the output of PO p_i is used by PO p_j. PO p_j requires stable input for time cycles θ_j. That is the last EO of the predecessor PO works for time λ_i and should hold its output for the extra time θ_j. The emerging constraint is $\lambda_i + \theta_j + 1 \leq R$ for every $p_i \rightarrow p_j$. (+1 is included to avoid possible transients.). If this condition is not satisfied then a buffer should be inserted between POs p_i and p_j.

Applying multiple copies of POs: Multiple copies of a PO are to be applied if $r_i + 1 > R$. The number of the copies, c_i should satisfy the constraint $r_i + 1 \leq c_i * R$. Multiplied POs need buffers in front of their inputs. This way stable inputs to the POs are ensured.

Loops need similar treatments for EOs and POs.

2.1.8 PO allocation using IPs

Let us define ROT and ROT period for every PO. A ROT will be denoted by q_i for PO p_i: q_i is the time interval that the resource, assigned to p_i, can not start the execution of another PO. If the successor of PO p_i is PO p_j then q_i includes execution time τ_i *and* θ_j as we have seen before. In turn,

$q_i = \tau_i + \max_{j:p_j \in S_i} \theta_j$ for every p_i. ROT period is the time interval that PO q_i occupies the resource, i.e., for PO q_i it is equal to the interval $[t_i, t_i + q_i)$.

In order to consider IPs I proceed as follows. I introduce compatibility classes in a similar way as before. A maximum compatibility class M_k encapsulates all POs with non-overlapping ROT periods taken modulo restart time r. I say that the POs belonging to the same compatibility class are non-concurrent. Expressed in mathematical terms: time intervals $[t_i, t_i + q_i)$ mod r are pair-wise disjunct for each PO p_i belonging to the same maximum compatibility class and the class can not be extended by new POs.

The i^{th} IP is denoted by Π_i. If PO p_k can be executed by IP Π_i then $\Pi_i(p_k) = 1$, otherwise $\Pi_(p_k) = 0$. From my point of view, IP Π_i is defined by its cost c_i and by the set of POs that this IP can execute $R_i = \{p_k : \Pi_i(p_k) = 1\}$ (Figure 7). Communication time (in time cycle units) is to be given too. Communication time $\delta_{i,j}$ is the time duration of inputting data from the Π_i to the Π_j. $i = j$ is allowed. Correct allocation is possible if sets R_i cover the set of the POs of the CDF graph. Allocation can proceed by choosing pairs of sets, M_k and R_i and I render the i^{th} IP, Π_i to all the POs belonging to $M_k \bigcap R_i$. Such steps are to be continued until no PO is left without an IP rendered to it. The objective of the optimization is the minimization of cost. Such minimization is possible because there are many ways to cover the set of POs. Rendering should satisfy the constraints raised by communication times. Successor of PO p_i can not start before data computed by p_i does not become available to it upon propagating the data through the communication layer.

2.2 Conclusions

I have extended the well-tried concepts and operations of high level synthesis to the networks. I've generalized the elementary operations of HLS as pipeline operations. Doing so, one is able to plan a hierarchical system, that is, to use the previously prepared pipeline systems as one single operation in more complex systems. Moreover, in many scales a task can be described with operations of various difficulties that suit scale-free networks well.

3 Weblog Algorithm in Scale Free Small Worlds

In this chapter I introduce a selection based algorithm, with the help of which the valuable resources can be easily searched for in certain network structures. I demonstrate the algorithm in a crawler system, but it can be naturally generalized for other problems.

According to the 'No Free Lunch Theorem' (NFL Theorem) [55] there is no performance difference between optimization or search algorithms if we test the algorithms on every possible problem. Therefore, it seems that improved performances of specific algorithms, such as selection, are consequences of specific problem properties. Finding these properties will aid the development of optimized solutions. Recent research shows that evolving structures, both natural and artificial (like the Web), exhibit scale-free or scale-free small world properties [6, 1]. In [2] the authors show that a particular evolutionary algorithm has better performance in artificial scale-free environment than in lattice, small world, or random environments. That is, the structure of the environment has an impact on the efficiency of the algorithms.

I have designed a controllable experiment and compared the performances of different algorithms on different structures. I aimed to keep the complexity of experiments and the number of designer specifiable parameters minimal. I considered the structure of the environment, the algorithms, and the fitness values. The choice for the structure was the World Wide Web (WWW). WWW is considered the largest source of rapidly changing data. The WWW has a scale-free small world (SFSW) structure [6, 23].

The everyday usage of the Web is the search for novel information. Therefore, I have a natural reward function; the number of novel documents that the algorithms can find. I consider this property as one of the most impor-

tant components of this work: *I did not specify the temporal and structural details of the reward system.*

I ran controlled experiments on a time stamped and downloaded portion of a large WWW news site. This allowed us to preserve the temporal structure of the rewards, but also supported the modification of the underlying connectivity structure, i.e., how the novel information can be found.

Agents were Web crawlers, or foragers. Crawlers travel from link to link foraging new, not-yet-seen information. The agents used the simplest possible algorithms. Selective learning concerned the selection and memorization of good links. I have also used a simple version of reinforcement learning (RL). For a review on RL see, e.g., [48]. RL was used alone, and was also combined with selective learning. My choice of RL is motivated by its structural and algorithmic simplicity and that RL is concerned with the optimization of the expected value of long-term cumulated profit.

For a recent review on evolutionary computing, see [14]. For reviews on related evolutionary theories and the dynamics of self-modifying systems see [17, 9] and [22, 11], respectively. Similar concepts have been studied in other evolutionary systems where organisms compete for space and resources and cooperate through direct interaction (see, e.g., [34] and references therein.)

Hybrid algorithms have a long history. A well-known example is the TD-Gammon program of Tesauro [50]. The author applied MLP function approximators for value estimation in RL. Reinforcement learning has also been used in concurrent learning problems like this: robots had to learn to forage together via direct interaction [27]. Another combinations of the algorithms concerns evolutionary learning embedded into the framework of RL to improve decision making [45, 32, 51, 25].

It is important to note, that communication and competition among the

foragers are indirect. Only the first submitter of a document may receive positive reinforcement and this is the only interaction among the crawlers. This work is different from other studies using combinations of genetic, evolutionary, function approximation, and reinforcement learning algorithms, in that i) it does not require explicit fitness function, ii) we do not have control over the original environment, iii) we can change the environment in a reproducible fashion, iv) collaborating individuals use value estimation under 'evolutionary pressure', and v) individuals work without direct interaction with each other. The crawler system is a self-assembling system, which is made of adaptive components, and the communication between components is kept as little as possible.

The chapter is organized as follows. I review the related web crawler tools, including those [24, 26, 37] that this work is based upon, in Section 3.1. I describe the algorithms and the forager architecture in Section 3.2. This section contains the necessary algorithmic details imposed by the task, the search of the Web. I present my experiments on the Web and the controlled simulations in Section 3.3. Discussions can be found in Section 3.4. Conclusions are drawn in Section 3.5.

3.1 Related work

There are important problems that have been studied in the context of crawlers. Angkawattanawit, Rungsawang [43], and Menczer [28] study topic specific crawlers. Risvik et al. [42] address research issues related to the exponential growth of the Web. Cho and Gracia-Molina [8], Menczer [28] and Edwards et. al [13] study the problem of different refresh rates of URLs (possibly as high as hourly or as low as yearly).

An introduction to and a broad overview of topic specific crawlers are

provided in [43]. They propose to learn starting URLs, topic keywords and URL ordering through consecutive crawling attempts. They show that the learning of starting URLs and the use of consecutive crawling attempts can increase the efficiency of the crawlers. The used heuristic is similar to the weblog algorithm [37], which also finds good starting URLs and periodically restarts the crawling from the newly learned ones. The main limitation of this work is that it is incapable of addressing the freshness (i.e., modification) of already visited Web pages.

Menczer [28] describes some disadvantages of current Web search engines on the dynamic Web, e.g., the low ratio of fresh or relevant documents. He proposes to complement the search engines with intelligent crawlers, or web mining agents to overcome those disadvantages. He introduces the InfoSpider architecture that uses genetic algorithm and reinforcement learning, also describes the MySpider implementation of it, which starts from the 100 top pages of AltaVista. The weblog algorithm uses local selection for finding good starting URLs for searches, thus not depending on any search engines. Dependence on a search engine can be a suffer limitation of most existing search agents, like MySpiders. Note however, that it is an easy matter to combine the present algorithm with URLs offered by search engines.

Risvik and Michelsen [42] overview different dimensions of web dynamics and show the arising problems in a search engine model. The main part of the paper focuses on the problems that crawlers need to overcome on the dynamic Web. As a possible solution the authors propose a heterogenous crawling architecture. The main limitation of their crawling architecture is that they must divide the web to be crawled into distinct portions manually before the crawling starts. A weblog like distributed algorithm – as suggested here – my be used in that architecture to overcome this limitation.

Cho and Garcia-Molina [8] define mathematically the freshness and age of documents of search engines. They propose the Poisson process as a model for page refreshment. The authors also propose various refresh policies and study their effectiveness both theoretically and on real data. They present the optimal refresh policies for their freshness and age metrics under the Poisson page refresh model. The authors show that these policies are superior to others on real data, too. Although they show that in their database more than 20 percent of the documents are changed each day, they disclosed these documents from their studies. Their crawler visited the documents once each day for 5 months, thus can not measure the exact change rate of those documents. While in this work I definitely concentrate on these frequently changing documents.

3.2 Forager architecture

There are two different kinds of agents: the foragers and the reinforcing agent (RA). The fleet of foragers crawls the web and sends the URLs of the selected documents to the reinforcing agent. The RA determines which forager should work for the RA and how long a forager should work. The RA sends reinforcements to the foragers based on the received URLs.

I employ a fleet of foragers to study the competition among individual foragers. A forager has simple and limited capabilities, like a stack for a limited number of starting URLs and a simple, content based URL ordering. The foragers compete with each other for finding the most relevant documents. In this way they efficiently and quickly collect new relevant documents without direct interaction.

At first I present the basic algorithms, followed by the algorithms for the reinforcing agent and the foragers.

3.2.1 Algorithms

My constraints on finding the minimal set of algorithms were as follows: The algorithms should (i) allow the identification of unimportant parameters, (ii) support the specialization of the individuals (the foragers), (iii) allow the joining of evolutionary learning and individual learning, (iv) minimize communication as much as possible. I shall return to these points in Section 3.4.

Weblog algorithm and starting URL selection

A forager periodically restarts from a URL randomly selected from the list of starting URLs. The sequence of visited URLs between two restarts forms a path. The starting URL list is formed from the 10 first URLs of the weblog. In the weblog there are 100 URLs with their associated weblog values in descending order. The weblog value of a URL estimates the expected sum of rewards during a path after visiting that URL. The weblog update algorithm modifies the weblog before a new path is started. The weblog value of a URL already in the weblog is modified toward the sum of rewards ($sumR$) in the remaining part of the path after that URL:

$$newValue = (1 - \beta)\, oldValue + \beta\, sumR,$$

where β was set to 0.3. A new URL has the value of actual sum of rewards in the remaining part of the path. If a URL has a high weblog value it means that around that URL there are many relevant documents. Therefore it may worth it to start a search from that URL.

Without the weblog update algorithm the weblog and thus the starting URL list remains the same throughout the searches. The weblog algorithm is a very simple version of evolutionary algorithms. Here, evolution may occur at two different levels: the list of URLs of the forager is evolving by the

reordering of the weblog. Also, a forager may multiply, and its weblog, or part of it may spread through inheritance. This way, the weblog algorithm incorporates the basic features of evolutionary algorithms. This simple form shall be satisfactory for my purposes.

Reinforcement Learning based URL ordering

A forager can modify its URL ordering based on the received reinforcements of the sent URLs. The (immediate) profit is the difference of received rewards and penalties at any given step. Immediate profit is a myopic characterization of a step to a URL. Foragers have an adaptive continuous value estimator and follow the *policy* that maximizes the expected long term cumulated profit (LTP) instead of the immediate profit. Such estimators can be easily realized in neural systems [48, 49, 44]. Policy and profit estimation are interlinked concepts: profit estimation determines the policy, whereas policy influences choices and, in turn, the expected LTP. (For a review, see [48].) Here, choices are based on the greedy LTP policy: The forager visits the URL, which belongs to the *frontier* (the list of linked but not yet visited URLs, see later) and has the highest estimated LTP.

In the particular simulation each forager has a $k(=50)$ dimensional probabilistic term-frequency inverse document-frequency (PrTFIDF) text classifier [21], generated on a previously downloaded portion of the Geocities database. Fifty clusters were created by Boley's clustering algorithm [7] from the downloaded documents. The PrTFIDF classifiers were trained on these clusters plus an additional one, the $(k+1)^{th}$, representing general texts from the internet. The PrTFIDF outputs were non-linearly mapped to the interval [-1,+1] by a hyperbolic-tangent function. The classifier was applied to reduce the texts to a small dimensional representation. The output vector of the classifier for the page of URL A is $\mathbf{state}(\mathbf{A}) = (state(A)_1, \ldots, state(A)_k)$.

(The $(k+1)^{th}$ output was dismissed.) This output vector is stored for each URL.

A linear function approximator is used for LTP estimation. It encompasses k parameters, the *weight vector* **weight** $= (weight_1, \ldots, weight_k)$. The LTP of document of URL A is estimated as the scalar product of **state**(**A**) and **weight**: $value(A) = \sum_{i=1}^{k} weight_i \, state(A)_i$. During URL ordering the URL with highest LTP estimation is selected. (For more details, see, [37].)

The weight vector of each forager is tuned by temporal difference learning (TD-learning) [47, 49, 44]. Let us denote the current URL by URL_n, the next URL to be visited by URL_{n+1}, the output of the classifier for URL_j by **state**($\mathbf{URL_j}$) and the estimated LTP of a URL URL_j by $value(URL_j) = \sum_{i=1}^{k} weight_i \, state(URL_j)_i$. Assume that leaving URL_n to URL_{n+1} the immediate profit is r_{n+1}. The estimation is perfect if $value(URL_n) = value(URL_{n+1}) + r_{n+1}$. Future profits are typically discounted in such estimations as $value(URL_n) = \gamma value(URL_{n+1}) + r_{n+1}$, where $0 < \gamma < 1$. The error of value estimation is

$$\delta(n, n+1) = r_{n+1} + \gamma value(URL_{n+1}) - value(URL_n).$$

I used throughout the simulations $\gamma = 0.9$. For each step $URL_n \to URL_{n+1}$ the weights of the value function were tuned to decrease the error of value estimation based on the received immediate profit r_{n+1}. The $\delta(n, n+1)$ estimation error was used to correct the parameters. The i^{th} component of the weight vector, $weight_i$, was corrected by

$$\Delta weight_i = \alpha \; \delta(n, n+1) \; state(URL_n)_i$$

with $\alpha = 0.1$ and $i = 1, \ldots, k$. These modified weights would improve value estimation in stationary and observable environments (see, e.g, [48] and

references therein), but were also found efficient in large Web environments [37].

Without the reinforcement learning based URL ordering update algorithm the weight vector remains the same throughout the search.

Document relevancy

A document or page is possibly relevant for a forager if it is not older than 24 hours and the forager has not marked it previously. The selected documents are sent to the RA for further evaluation.

Multiplication of a forager

During multiplication the weblog is randomly divided into two equal sized parts (one for the original and one for the new forager). The parameters of the URL ordering algorithm (the weight vector of the value estimation) are either copied or new random parameters are generated. If the forager has a URL ordering update algorithm then the parameters are copied. If the forager does not have any URL ordering update algorithm then new random parameters are generated.

3.2.2 Reinforcing agent

A reinforcing agent controls the 'life' of foragers. It can start, stop, multiply or delete foragers. RA receives the URLs of documents selected by the foragers, and responds with reinforcements for the received URLs. The response is 100 (a.u.) for a relevant document and -1 (a.u.) for a not relevant document. A document is relevant if it is not yet seen by the reinforcing agent and it is not older than 24 hours. The reinforcing agent maintains the score of each forager working for it. Initially each forager has 100 (a.u.) score. When a forager sends a URL to the RA, the forager's score is decreased by 0.05. After each relevant page sent by the forager, the forager's score is increased

by 1.

When the forager's score reaches 200 and the number of foragers is smaller than 16 then the forager is multiplied. That is a new forager is created with the same algorithms as the original one has, but with slightly different parameters. When the forager's score goes below 0 and the number of foragers is larger than 2 then the forager is deleted. Note that a forager can be multiplied or deleted immediately after it has been stopped by the RA and before the next forager is activated.

Foragers on the same computer are working in time slices one after each other. Each forager works for some amount of time determined by the RA. Then the RA stops that forager and starts the next one selected by the RA.

3.2.3 Foragers

A forager is initialized with parameters defining the URL ordering, and either with a weblog or with a seed of URLs. After its initialization a forager crawls in search paths, that is after a given number of steps the search restarts and the steps between two restarts form a path. During each path the forager takes 100 steps, i.e., selects the next URL to be visited with a URL ordering algorithm. At the beginning of a path a URL is selected randomly from the starting URL list. This list is formed from the 10 first URLs of the weblog. The weblog contains the possibly good starting URLs with their associated weblog values in descending order. The weblog algorithm modifies the weblog and so thus the starting URL list before a new path is started. When a forager is restarted by the RA, after the RA has stopped it, the forager continues from the internal state in which it was stopped.

The URL ordering algorithm selects a URL to be the next step from the frontier URL set. The selected URL is removed from the frontier and added

to the visited URL set to avoid loops. After downloading the pages, only those URLs (linked from the visited URL) are added to the frontier which are not in the visited set.

In each step the forager downloads the page of the selected URL and all of the pages linked from the page of selected URL. It sends the URLs of the possibly relevant pages to the reinforcing agent. The forager receives reinforcements on any previously sent but not yet reinforced URLs and calls the URL ordering update algorithm with the received reinforcements.

3.3 Experiments

I conducted an 18 day long experiment on the Web to gather realistic data. I used the gathered data in simulations to compare the weblog update (Section 3.2.1) and reinforcement learning algorithms (Section 3.2.1). In the Web experiment I used a fleet of foragers using combination of reinforcement learning and weblog update algorithms to eliminate possible biases on the gathered data. First I describe the experiment on the Web then the simulations. I analyze the results in the next section.

3.3.1 Data collection on the Web

I ran the experiment on the Web on a single personal computer with Celeron 1000 MHz processor and 512 MB RAM. I implemented the forager architecture (described in Section 3.2) in Java programming language.

In this experiment a fixed number of foragers were competing with each other to collect news at the CNN web site. The foragers were running in equal time intervals in a predefined order. Each forager had a 3 minute time interval and after that interval the forager was allowed to finish the step started before the end of the time interval. I deployed 8 foragers using the

weblog update and the reinforcement learning based URL ordering update algorithms (8 WR foragers). I also deployed 8 other foragers using the weblog update algorithm but without reinforcement learning (8 WL foragers). The predefined order of foragers was the following: 8 WR foragers were followed by the 8 WL foragers.

I investigated the link structure of the gathered Web pages. As it is shown in Fig. 8 the links have a power-law distribution ($P(k) = k^{\gamma}$) with $\gamma = -1.3$ for outgoing links and $\gamma = -2.57$ for incoming links. That is the link structure has the scale-free property. The clustering coefficient [54] of the link structure is 0.02 and the diameter of the graph is 7.2893. I applied two different random permutations to the origin and to the endpoint of the links, keeping the edge distribution unchanged but randomly rewiring the links. The new graph had 0.003 clustering coefficient and 8.2163 diameter. That is the clustering coefficient was smaller than the original value by an order of magnitude, but the diameter is almost the same. Therefore we can conclude that the links of gathered pages form *scale-free small world* structure.

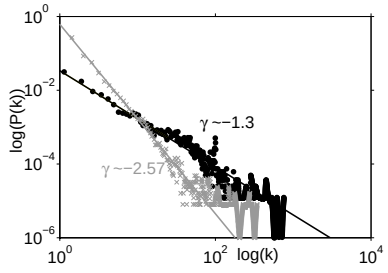


Figure 8: **Scale-free property of the Internet domain**. Log-log scale distribution of the number of (incoming and outgoing) links of all URLs found during the time course of investigation. Horizontal axis: number of edges ($\log k$). Vertical axis: relative frequency of number of edges at different URLs ($\log P(k)$). Dots and dark line correspond to outgoing links, crosses and gray line correspond to incoming links.

The data storage for simulation is a central issue in the experiments. Pages are stored with 2 indices (and time stamps). One index is the URL index, the other is the page index. Multiple pages can have the same URL index if they were downloaded from the same URL. The page index uniquely identifies a page content and the URL from where the page was downloaded. For any foragers, at each page download I stored the followings (with a time stamp containing the time of page download):

1. if the page is relevant according to the RA then store 'relevant'
2. if the page is from a new URL then store the new URL with a new URL index and the page's state vector with a new page index
3. if the content of the page is changed since the last download then store the page's state vector with a new page index but keep the URL index
4. in both previous cases store the links of the page as links to page indices of the linked pages
   (a) if a linked page is from a new URL then store the new URL with a new URL index and the linked page's state vector with a new page index
   (b) if the content of the linked page is changed since the last check then store the page's state vector with a new page index but same URL index

3.3.2 Simulations

For the simulations I implemented the forager architecture in Matlab. The foragers were simulated as if they were running on one computer as described in the previous section.

Simulation specification

During simulations I used the Web pages that I gathered previously to generate different environments (note that the links of pages point to local pages (not to pages on the Web) since a link was stored as a link to a local page index):

- Simulated documents had the same state vector representation for URL ordering as the real pages had
- Simulated relevant documents were the same as the relevant documents on the Web
- Pages and links appeared at the same (relative) time when they were found in the Web experiment - using the new URL indices and their time stamps
- Pages and links are refreshed or changed at the same relative time as the changes were detected in the Web experiment – using the new page indices for existing URL indices and their time stamps
- Simulated time of a page download was the average download time of a real page during the Web experiment.

I generated 4 different environments for the simulations:

1. **SFSW**: each simulated page had exactly the same links as the original page had on the Web (a simulated page linked those simulated pages, page indices of those pages, which were linked by the original Web page).
2. **SF1**: in each second the new simulated pages had the same number of links as the original pages on the Web. A new simulated page linked

to simulated pages selected by the preferential attachment algorithm from the existing simulated pages.

3. **SF2**: the previous algorithm applied for the **SF1** environment.

4. **SFRandom**: similar to the **SF1**, but the linked simulated pages are selected from a uniform distribution of the pages.

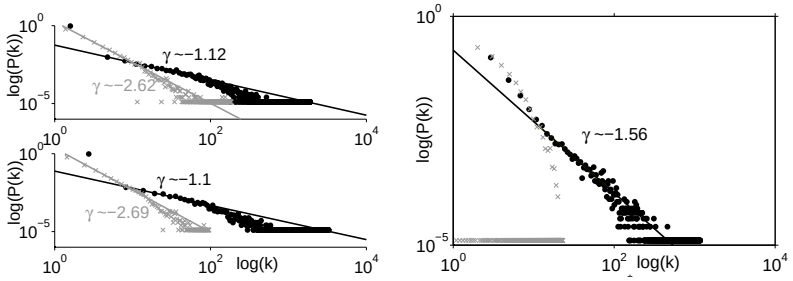
The SFSW environment has exactly the same scale-free and small world properties as the web environment downloaded by the web foragers. The SF1 and SF2 environments have 10 times smaller clustering coefficient than the SFSW environment has. These environments have scale-free degree distributions, although those are slightly different from the web environment (see Fig. 9(a)).

The SFRandom environment, also, has 10 times smaller clustering coefficient then SFSW. The SFRandom environment has scale-free outgoing link degree distribution. But because of the uniform selection of linked documents the incoming link degree distribution is exponential (see Fig. 9(b)). With the above given constraints this environment is the most random in the sense that all of the free parameters (linked documents) were selected from the uniform random distribution.

I conducted simulations with four different kinds of foragers in each environment:

1. **WR foragers** used both the weblog update and the reinforcement learning based URL ordering update algorithms.

2. **WL foragers** used only the weblog update algorithm without URL ordering update. Each WL forager had a different weight vector for URL value estimation – during multiplication the new forager got a new random weight vector.



(a) SF1,SF2 (b) SFRandom

Figure 9: **Degree distribution of the environments.**
Dots and dark line correspond to outgoing link distribution. X-s and gray line correspond to incoming link distribution. (a) Upper: degree distributions of SF1 environment. Lower: degree distributions of SF2 environment. (b) : degree distributions of SFRandom environment.

3. **RL foragers** used only the reinforcement learning based URL ordering update algorithm without the weblog update algorithm. RL foragers had the same weblog with the first 10 URLs of the gathered pages – that is the starting URL of the Web experiment and the first 9 visited URLs during that experiment.

4. **Fix foragers** did not use the weblog update and the reinforcement learning based URL ordering update algorithms. These foragers had fixed starting URLs and fixed weight vectors, but the latter was different for each Fixed forager.

In each case, initially there were 2 foragers and they were allowed to multiply until reaching the population of 16 foragers. The simulation for each type of foragers were repeated 3 times with different initial weight vectors for each forager. The variance of the results show that there is only a small difference between simulations using the same kind of foragers, even if the

foragers were started with different random weight vectors in each simulation.

Simulation measurements

The first thing that I should note concerns the efficiency as a function of the number of crawlers. On a single computer, and under the time sharing method I applied, and without direct competition between the different crawlers, I found that bipartition gives rise to a transient decrease of the efficiency of the new crawlers, but it quickly recovers. Within the limits of the number of crawlers that I studied (between 2 and 24), performance of the fleet is a slowly increasing function of the number of crawlers. I fixed the number of the crawlers and this slow dependence did not enter my considerations. Table 2 shows the investigated parameters during simulations.

Table 2: **Investigated parameters**

downloaded	number of downloaded documents
sent	number of documents sent to the RA
relevant	number of found relevant documents
found URLs	number of found URLs
download efficiency	ratio of relevant to downloaded documents in 3 hour time window throughout the simulation.
sent efficiency	ratio of relevant to sent documents in 3 hour time window throughout the simulation.
exploration	ratio of found URLs to downloaded at the end of the simulation
freshness	ratio of the number of current found relevant documents and the number of all found relevant documents [8]. A stored document is current, up-to-date, if its content is exactly the same as the content of the corresponding URL in the environment.
age	A stored current document has 0 age, the age of an obsolete page is the time since the last refresh of the page on the Web [8].

Parameter 'download efficiency' is relevant for the site where the foragers should be deployed to gather the new information. Parameter 'sent efficiency' is relevant for the RA. Note that during simulations I am able to immediately

and precisely calculate freshness and age values. In a real Web experiment this is impossible, because of the time needed to download and compare the contents of all of the real Web pages to the stored ones.

3.4 Discussion

I observed, that the efficiency of the algorithms depends strongly on the weight vectors. As I have mentioned above, the number of foragers had slight effects on the efficiency. This observation is supported by the fact that upon bipartition the weight vectors of the descendant foragers are similar, causing the descendant foragers to follow similar paths and to spoil the performance of the other. This is the reason that efficiency shows a transient decrease, but as a result of adaptation it quickly disappears.

The dependence of the efficiency on the number of starting points is not too much different. Two foragers could use 20 different starting points, whereas 16 foragers could use 160 different starting points, but the efficiency was only slightly influenced by this order of magnitude difference. That is, the number of starting points and the number of foragers are less important than the weight vectors in my simulations indicating that 20 starting points or so, which can adapt, can efficiently shatter the structure. Note that other parameters, e.g., larger number of crawlers, more than one computer can change this picture. The weak dependence, however, is most advantageous for my purpose namely to be confident that I can study structural dependencies.

The measured parameter values are presented in Fig. 10. The figure contains the values for each measured parameter for each type of forager in each type of environment. From the subfigures I can conclude the followings:

It can be seen in the top left and top middle subfigures that freshness and age values of different foragers are changing in the SFSW environment

(marker x) while in the other 3 environments the values are almost the same for the different type of foragers. RL and Fix foragers can get trapped in clustered environments, can not easily escape (find outgoing links leaving the cluster) from clusters containing less relevant documents. The weblog algorithm provides a way to go to the found best clusters and in this way to escape from the worse clusters. This can be the reason why the RL and Fix foragers in the SFSW environment performs worse than WL and WR foragers.

In the top right subfigure it can be seen that finding new documents is the hardest in the SFRandom environment for all foragers. In this environment the pages has exponential incoming link degree distribution which means that there are relatively more small degree pages than in the other 3 scale-free environments. It is harder to get to pages which links to many pages. Therefore it is harder for the foragers to find not yet seen documents.

It can be seen in the middle left and middle subfigures that finding relevant or possibly relevant documents is the easiest in the SFSW environment for all foragers. This environment is more clustered than the other three environments. When a forager finds a relevant document in a cluster then it finds other relevant documents in that cluster while it can not escape from the cluster. The other 3 environments are less clustered. Foragers can get out from clusters easier by following the links and therefore have to forage the entire environment for new relevant documents.

In the bottom left subfigure it can be seen that the download efficiency is the best in the SFSW environment for all foragers. This is because of the high number of found relevant documents compared to the other three environments.

In the bottom middle subfigure it can be seen that the WR forager's sent

efficiency is the same in all environments. Although these foragers found less relevant documents in not SFSW environments but also sent back less documents to the Reinforcing Agent. The other three foragers sent more less documents to Reinforcing Agent in the not SFSW environments, therefore their sent efficiencies are the worst in the SFSW environment. Although these foragers also found the most relevant documents in the SFSW environment, compared to the other 3 environments.

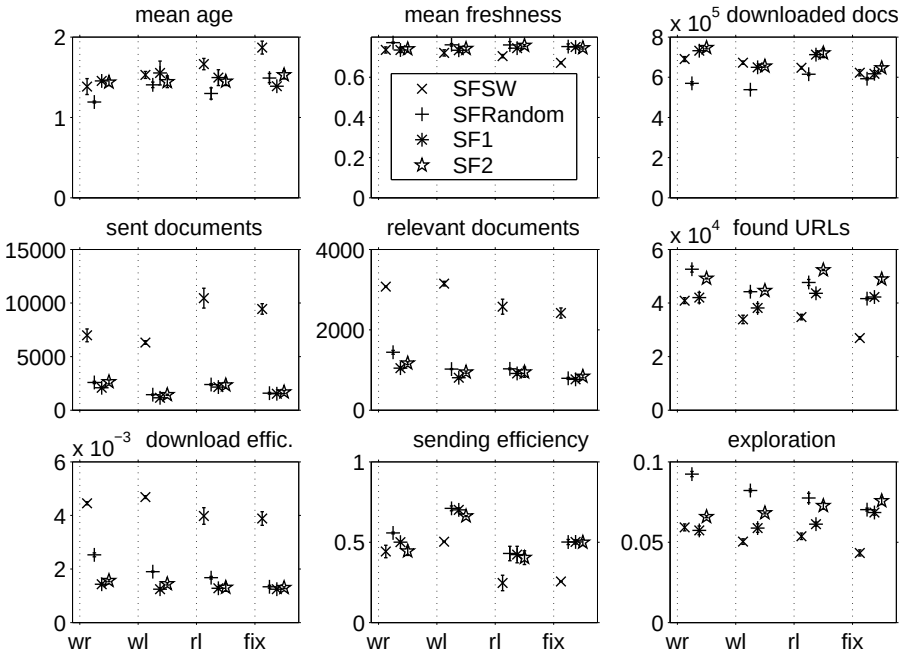


Figure 10: **Measured parameter values.**
The subfigures show the investigated parameters. Each subfigure contains the parameter values for the four type of foragers in four columns as shown below the bottom subfigures. The 4 different markers correspond to the measured parameter values in the 4 environments as shown in the top middle figure legend. On each marker an error bar shows the standard deviation of the corresponding parameter values for the 3 simulations. Mean age is in hours in the upper left subfigure.

It can be seen in the middle right subfigure that finding new URLs is the

hardest in the SFSW environment because of its clustered nature. The foragers checks the same URLs for changed documents in the clusters therefore they can collect many new relevant documents. In the other three environments foragers do not get trapped as much in the clusters and they search the whole environment continuously.

Now, consider Table 3, which contains some of the data of Fig. 10. The number of sent relevant documents is somewhat larger for the WL foragers than for WR foragers in SFSW environments (3149 and 3075, respectively, i.e., the difference is about 2.3%). This slight difference changes sign and becomes much more pronounced in all other environments. For example, in the SFRandom environment the numbers are 1029 and 1441, i.e., the difference is about 28%, but in the opposite direction. It is important to note that, if these foragers compete with each other, then the slight 2.3% difference or the larger 28% difference both enter the argument of an exponential because of bipartitions. Thus, even slight differences can become large, and may give rise to overwhelming population differences. Such competitive runs are under way. The issue becomes more pronounced for real situations, where time is not shared on a single computer and all foragers may search for food at all times.

My results and the results of Annunziato et al. [2] can not be compared directly. The most obvious reason is that their investigations were restricted to SF, SW and random structures, but they did not study the SFSW structure, which plays a central role in this work. Artificial studies are, however, desirable, because in such examples one can finely gauge the different components and may find the necessary and sufficient conditions of my findings.

I consider the following findings important: Bipartition gives rise to certain transient disadvantages if the descendants are similar. They will have

Table 3: **Quantitative results for algorithm–structure pairs**

	Weblog		Weblog and RL		RL	
	SFSW	SFR	SFSW	SFR	SFSW	SFR
No of sent docs	6313	1443	6985	2585	10455	2396
No of relevant docs	3149	1023	3075	1441	2575	1029
Sending efficiency	0.5035	0.7106	0.4425	0.5574	0.2463	0.4299
No of found URLs	33882	44217	40888	52636	34759	47668

to share the food until they start to learn. Still, selective learning can be more effective than selective learning combined with other methods, or than other methods alone if the environment is SFSW. It seems that selection fits the SFSW structure and vice versa. This could be a good reason for the abundance of emergent SFSW structures in nature. This can be understood through the NFL Theorem if it is read backwards: If we find that a particular algorithm is more efficient than random search or than any other algorithm, then this winning algorithm 'knows' (fits) the most the underlying problem among the investigated algorithms. Based on the computer simulations it seems that highly clustered SFSW structures and simple selective learning algorithms match each other.

3.4.1 General weblog algorithm

In this chapter I demonstrated the algorithm to find valuable resources in a web crawler problem. Here the valuable resources were the URLs from which much new that day information can be found. This algorithm can be applied for the task execution problem easily. The processing units have to find the reliable partner processing units for executing tasks together. This

algorithm enables the processing units to find the reliable partner processing units through the edges of the communication network of the processing units.

3.5 Conclusions

I investigated algorithms using evolutionary, reinforcement learning, and combined evolutionary and reinforcement learning strategies. I experimented with environments having different degree distributions and clustering coefficients. I generated different topological environments, using data collected during real Web search. My study focused on the task of searching for new relevant documents. I found that in the scale-free small world environment the evolutionary weblog update algorithm performs the best. It outperformed a reinforcement learning based algorithm and the combinations of these two algorithms. I conjecture that the highly clustered nature and the small diameter of the environment match simple selection over other more sophisticated learning schemes. However, when the scale-free nature of the environment was kept but the small diameter of the environment was increased simply by restructuring the environment, then other algorithms performed better than the simple selection. I found in the 3 not small world environments that the combination of the weblog and reinforcement learning algorithms are the best. That is, when the diameter of the world becomes larger, then estimation of the long-term cumulated reward becomes important. Moreover, the combined algorithm showed the smallest performance variation both on the scale-free small world and on scale-free environments.

In the next chapter I show how to apply the weblog algorithm to find the reliable partner processing units in the task execution problem.

4 Survivable Pipeline Protocol

In this chapter I introduce a novel peer-to-peer procedure that is called Survivable Pipeline Protocol (SPP). The goals of SPP are as follows: i) create scheduled and allocated pipelines alike to hardware pipeline systems; ii) maintain, check and forecast the quality of the pipeline during operation; iii) fix the pipeline if necessary; iv) work efficiently in networks which have scale-free small-world (SFSW) connection structure (for a review on SFSW see e.g. [6, 1]. The idea of SPP is to extend the concept of hardware pipelines to network pipelines. The scheduling and allocation of network resources (like communication time, bandwidth, processing power) creates network pipelines both for communication and for computation. The assumed SFSW property and the related algorithmic structure of the sensor and computing network are based on i) the mobile and evolutionary nature of the functioning of the system in hostile environments, ii) that all known evolved structures exhibit SFSW communication and iii) that I extensively studied algorithms in different topologies and found efficient and reasonable adaptive methods [35, 36]. That is I expect that SPP will find resources fast and will be competitive with respect to other algorithms.

I have extended the concept of elementary operations of high level synthesis of pipelined datapaths [5] to pipeline operations (POs) [38]. A synthesized pipeline system can be fully described by one PO enabling to design even larger pipeline systems using the already synthesized ones. The PO concept allows the scalable extension to networks, provided that the appropriate protocol, e.g., SPP, is developed. The prototype application of SPP is the position monitoring system for moving wheelchairs of handicapped children at the Center of the Hungarian Bliss Foundation.

First, I review peer-to-peer basics (Section 4.1). It is followed by the

description of the protocol (Section 4.2). Experimental results are provided in Section 4.3. Outlook about the prototype application can be found in Section 4.4.

4.1 P2P basics

In peer-to-peer networks usually there is no central decision making peer that decides the task of the other peers, that is each peer can decide and choose its own tasks. Also, the resources within the network can be found by their unique identifiers (IDs) in a distributed way. Such resources are, e.g., data, code, peers, and communication channels. I will call the communication channels as pipes. Each pipe has its unique ID which is independent from the peers communicating through the pipe. The protocol requires two basic pipe types: i) unicast unidirectional pipe and ii) multicast unidirectional pipe. The pipes should be able to accept data from more peers and send the data to the receiver peer(s). This feature fits into the distributed nature of these networks as a peer does not have to know if other peers are sending data on the given pipe or not. If the network provides the ability to create and maintain peer groups with the ability that communication is restricted to the groups, then the protocol will have great advantages in wireless (but also in wireline) applications.

I have implemented SPP over JXTA P2P network in Java [33]. JXTA has the required communication channels (Point-to-point Pipe, Propagate Pipe) and supports peer groups. The so called Rendezvous Peers can maintain peer groups with some basic peer group services, and peer groups can be hierarchically organized. Any peer can be a Rendezvous Peer, so can start to form a peer group or even more peer groups. A peer can be the member of multiple peer groups at the same time.

4.2 Survivable Pipeline Protocol

To be able to organize a task (which may contain computation, sensing, and multi-hop communication parts) SPP requires the following task description: i) a Data-flow Graph with Pipeline Operation vertices, where an operation can be a computation, sensing, or sending data from a source to a destination; ii) the maximal allowed latency and restart time of the task; iii) reliability parameters, like the minimal ratio of allocation of resources.

In SPP there are 3 different roles of peers: i) processor, ii) distributor and iii) manager. A processor peer is responsible for executing the accepted subtask on the received data with the accepted latency and restart time and for maintaining enough distributors for its subtask. The distributors are responsible for listening for incoming data and sending it to a processor. If a data consists of multiple parts then these parts do not have to arrive to the same distributor, but the different parts have to be sent to the very same processor. Distributors are responsible for maintaining one well working manager for the subtask. The manager is responsible for providing a data allocation for the distributors which tells which data should be sent to which processor. If a processor provides more resources (e.g., more CPU time) then it should receive more data for processing compared to a processor which provides less resources. Also, the manager is responsible for finding a manager for an arbitrary first part of the successor task of its subtask if any. A manager can be a distributor and a distributor can be a processor, too.

SPP needs 6 pipe collections for these functions:

SPP Pipe It is a multicast unidirectional pipe. Requests for organizing a task and requests for processor peers for a subtask are sent on this pipe.

Data Pipe It is a unicast unidirectional pipe. At each data item the pipe is connected to some receiving peer and the data is sent through the

pipe.

Distributor Pipe It is a multicast unidirectional pipe. The manager sends the data allocation to the distributors on this pipe.

Task Pipe It is a multicast unidirectional pipe. The manager sends information to all processor peers on this channel, e.g., stops the processing of this subtask. Also, distributors of the subtask must send their peer ID regularly to prove their liveliness to the processors.

Manager Pipe It is a unicast bidirectional pipe. If peer P wants to organize a task then asks the submitter of the request on this pipe if the submitter allows for P to organize the task. Upon acceptance, it becomes the manager of the task and can advertise subtasks. The manager of a (sub)task can check the liveliness of the manager of the successor task(s), can stop processing of the successor task(s), and receive the accepted subtask(s)from the successor manager(s). Also manager of the predecessor subtask can be notified about the intent to stop processing this subtask.

TaskManager Pipe It is a unicast bidirectional pipe (one pipe for sending information to the manager, and a pipe for each peer which waits for answer from the manager). The manager of a subtask can be checked for liveliness, can be stopped, can be notified about the quitting of a processor, and can be asked to permit to be a processor for this subtask.

Each maintained subtask of a task corresponds to one manager, and if supported to a distinct peer group. This is required to have only one data allocation for a subtask so the parts of a data will be sent to the same processor by the distributors. The permission requests (for organization and becoming a processor) are required to have only one manager (and peer group) for a subtask and to have only the necessary amount of processing

power for a subtask according to the task specifications.

The manager is responsible to find an other manager for an arbitrary first part of its subtask's successor task. That is the manager is not responsible for the whole successor task. It may decide to find another peer which will organize some part of the successor task and that peer will find another peer to organize some part of its own successor task until the process ends.

When a manager has to find some other peers then it uses a special peer list to select the possibly best peers at first. This list is called weblog [36, 35, 39]. The weblog is the ordered list of peers in descending order according to the weblog value of the peers. The higher the weblog value of a peer the higher the reliability of the peer for the execution of the given task. The selected peers to which the request will be sent are the first few peers from the list. The weblog value of a peer is raised when it accepts a subtask processing or organization and after successfully processed data. The weblog value of a peer is lowered if the peer stops processing the subtask, or the organization of the successor task fails, and after not successfully processed data.

If a peer accepts to process a subtask with a given latency and restart time, then it can decide to brake that subtask into smaller subsubtasks and find other peers to organize and process these subsubtasks alike to the organization of the task. That is, the peer will organize some first part of the subtask with the given timing constraints, finds processors for the first part, and finds a manager for some part of the successor subsubtask. In this way a hierarchical task processing structure can be formed that can break the task into parts according to previous experiences and may better exploit the available resources then prewired protocols.

The following list contains the general steps of a successful task organi-

zation and maintenance:

1. **Start Condition:**
   (a) **SPP Peer Group** exists.
   (b) **SPP Pipe** exists in SPP Peer Group.
2. **Create:** Create a peer group for the task.
   (a) Peer submits organization request on **SPP Pipe**.
   (b) At least one peer receives the request.
   (c) At least one peer accepts an executable part of the task and the continuation of the organization chain.
   (d) Peer asks for permission from the submitter peer.
   (e) Upon permission peer creates a peer group for the permitted subtask. This peer will be the first processor, distributor and manager of the accepted subtask.
3. **Monitoring activities**
   (a) Subtask manager liveliness monitoring
   (b) Monitoring of number of subtask distributors
   (c) Processor (self-)monitoring
   (d) Monitoring of liveliness of successor subtask's manager
4. **Actions Related to Subtask Survival:**
   (a) Decide if accepted subtask is processed reliably.
   (b) If not, for example the number of peers is below threshold to provide the required restart time of the task, then select partner peers from weblog and send processor request to selected partner peers on **SPP Pipe**.
   (c) Wait for some time
   (d) If answers do not satisfy constraints then send processor request to all peers on **SPP Pipe**.

(e) Wait for some time

(f) If answers do not satisfy constraints then notify the manager of the predecessor task that the maintenance of subtask has failed and subtask will be stopped at this peer.

5. **Actions Related to Successor Task Survival:**

   (a) Decide if parts of successor task of the accepted subtask are organized reliably.

   (b) If not, for example the successor manager does not respond in time, then select successor peers from weblog and send an organize request to selected successor peers on **SPP Pipe**.

   (c) Wait for some time

   (d) If answers do not satisfy constraints then send organize request to all peers on **SPP Pipe**.

   (e) Wait for some time

   (f) If answers do not satisfy constraints then notify the manager of the predecessor task that the maintenance of this subtask has failed and subtask will be stopped at this peer.

6. **End of task:**

   - If the pipeline breaks or if the task is finished then peers are released from the pipeline through the **Manager Pipe** and **Task Pipe**.

An example task is shown in Fig. 11. It contains 5 Pipeline Operations, PO1-PO5. PO2 gives input to both PO3 and PO4 while PO5 requires input from both PO3 and PO4. The initial task organize request for the whole task was submitted by Peer P0. P1-P5 peers are the managers for the processing of PO1-PO5 respectively. But P2 is responsible for the whole subtask PO2-PO5, too. The other peers P6-P13 are processors of the operations. Larger

rectangles denote peer groups. For each task and subtask a separate peer group is to be created to localize the communication of the peers of a subtask. Arrows are data pipes. An arrow going into a PO is a pipe on which the distributors are listening for data. Arrows going into peers are peer group local pipes on which the distributors send the data to the processors. When a processor is processed a data it sends the data to the successor task on the pipe that the distributors of the successor task are listening on.

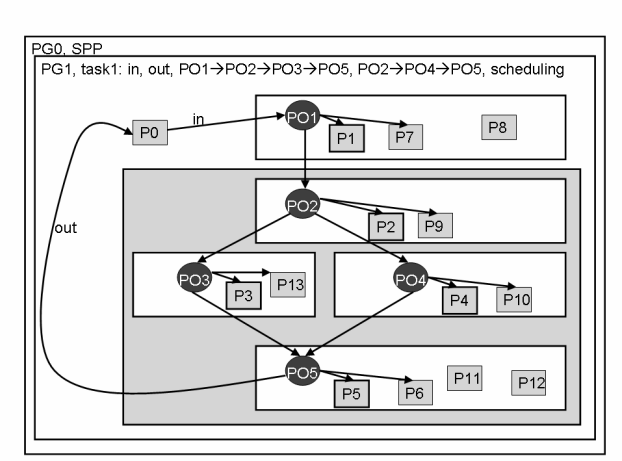


Figure 11: **Example SPP task**

In this particular example, the 'story' is as follows. At first, peer P0 sends the organize request of the whole task. Peer P1 accepts the first PO and P0 allows it to organize the processing of PO1. P1 sends organize request for the successor task containing PO2-PO5. P2 accepts the organization of the whole successor task and P1 allows to organize it, therefore the darker rectangle peer group is created. P2 decides to brake into parts the accepted task. P2 organizes the processing of PO2 and sends organize request for the remaining task PO3-PO5. PO3 is accepted by P3, then PO4 is accepted by P4 and allowed by P3 and finally PO5 is accepted by P5 and allowed by P4.

The manager peers are already started to find processor peers to fulfill the reliability and timing requirements of the task. Once enough peers are found, data processing can be started according to experience based confidence. Peer P0 starts to send data and reinforcement about previously processed data. If a processed data is received within time constraints then a reward, otherwise a penalty is given to the system.

4.2.1 Downscaled version for RF-MEMS devices

I have downscaled the above protocol for small processors which communicate through radio signals. The subtasks to be executed by one processor belongs to one group of processors, as in the above protocol. To simplify the protocol the manager and distributor roles are distributed among the processors of a subtask. The actual manager of a subtask is the processor which is sending its processed data to the successor subtask. The distributor role is realized by a free peer list which contains the not working peers in order. When the processors of a subtask receives a data they check which processor is the first in their own free peer lists. Ideally at each processor the same one is the first, so that processor starts the processing. The IDs of the first few processors are sent in the data and acknowledgement messages to overcome inconsistencies among the lists. In this version only the SPP and Data Pipes are used. The SPP Pipe is used for task announcements and Data Pipes are used for all other communication. The Data Pipes to the successor operation are used for communication between the processors of a PO.

4.3 Experimental results

4.3.1 JXTA SPP

I have tested the JXTA implementation of SPP on PCs with the following scenarios: i) a single PC is enough to execute a task with a single PO (latency=1000ms, restart time=2000ms), ii) 10 PCs are required to execute the previous task, iii) there are PCs which execute some POs faster then other PCs (latency=1000 vs. 2000 ms, restart time=2000 vs. 4000 ms), 4 faster PCs are required to execute a task of two POs of the same type. The restart time, the average latency of data processing and the weblog values of PCs are shown in Table 4.

Table 4: **JXTA SPP results**

	restart time	avg. latency	weblog values
i)	2000ms	1127ms	1000
ii)	200ms	1138ms	1000
iii)	1500ms	2149vs3155ms	4000vs3000

I have found that the latency of the data processing slightly increased in the second experiment compared to the first experiment but data was 10 times faster processed by 10 PCs than by 1 PC. That is, in this example, the computational capacity of the SPP network scales linearly with the amount of available resources. In the third experiment the faster PCs have higher weblog values than the slower PCs, therefore the faster PCs will be selected first for processing during organization.

In summary the JXTA implementation of SPP can be used over the internet to create and maintain a scalable computation power where the tasks are optimized to use the best resources. Moreover this implementation can be used as a grid system with an easy to use programming interface.

4.3.2 TinyOS SPP

I have implemented the SPP for the Berkeley Mica2 mote based Cricket in NesC language over TinyOS operating system [19]. This implementation supports only one specific task because of the limited capabilities of such small devices (ATMega 128L, 7.37MHz, 8 bit processor, 128 KB program memory, 4 KB data memory, 512 KB serial flash memory). The particular application I have developed for the motes is to calculate their relative positions to each other. Each Cricket mote has 2 ultrasound transducers to measure distances from each other. The task of the system is to maintain a coordinate system in the plane of the sensors, that is to select 3 base motes, and to calculate the positions of the motes in this coordinate system.

The task contains 4 POs, the 3 bases and the relative position calculation PO. The POs are connected one after each other, and after the fourth PO, the first PO (the first base) starts the processing again. The first base PO's coordinates are always (0,0), its only task is to emit an RF and UH beacon. The second base determines the direction of the x axes of the coordinate system. So its coordinates are x,0, where x is the distance from the first base. The second base emits an RF and UH beacon after the first base. The third base determines the direction of the y axes. The y axes is directed so that the third base has a positive y coordinate. The third base calculates its coordinates based on the beacons from the first two bases and emits a third RF and UH beacon. The beacons contain the coordinates of the emitting base. Finally the other motes calculate their coordinates based on the beacons from the 3 bases. But the first base does not know yet when to start the next beacon. Therefore there is an order among the 4^{th} PO's motes. In each turn one mote will signal the first base to continue the beacons. In this way the ordinary motes have a chance to send their locations to others,

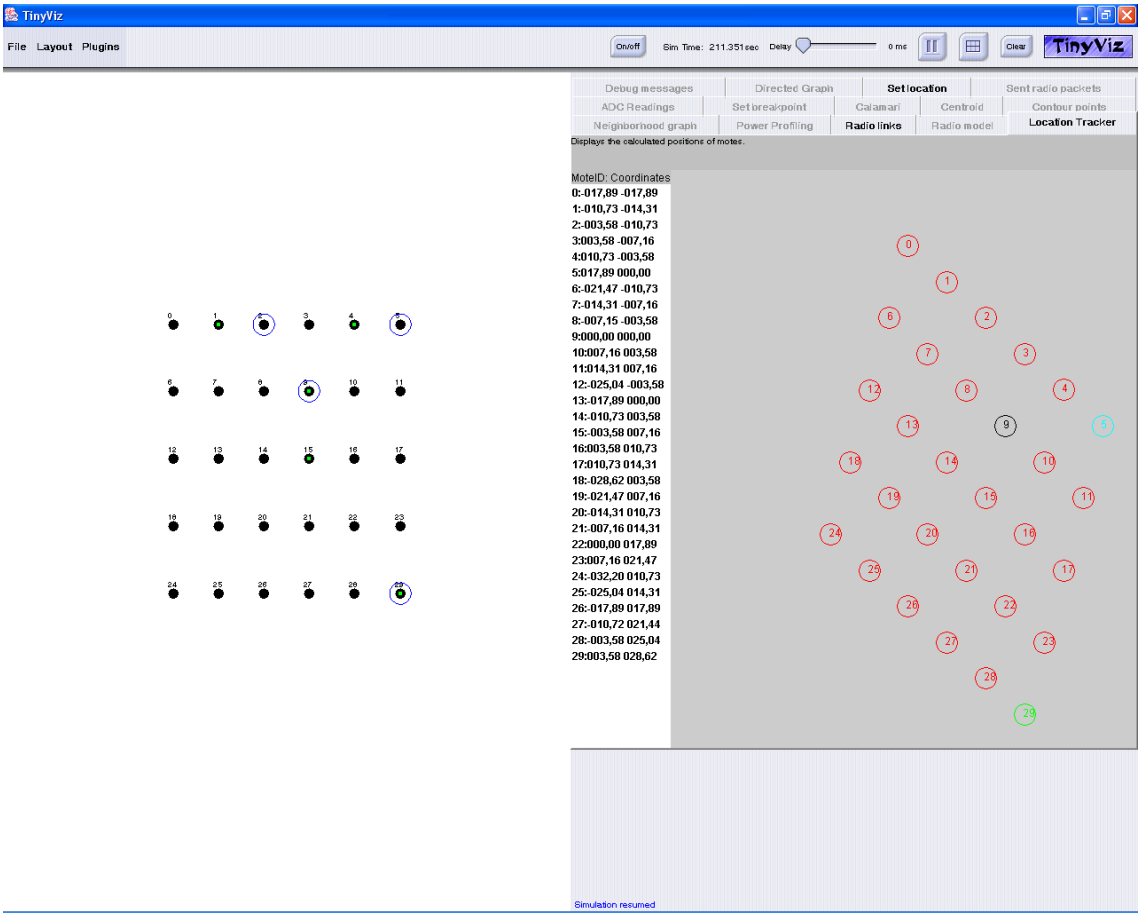


Figure 12: **Simulation of 30 motes**

for example, to warn them that they are too close to each other.

Figure 12 shows the simulation when 30 motes are working together. Three motes are selected as the 3 bases and the other motes calculate their relative positions. The motes were started within 10 seconds. This makes hard to create an order among the motes of the 4^{th} PO.

Figures 14, 13 show the real experiment with 15 motes organized in a circle. It can be seen on the computer screen that the motes successfully transmitted their relative position to the PC.

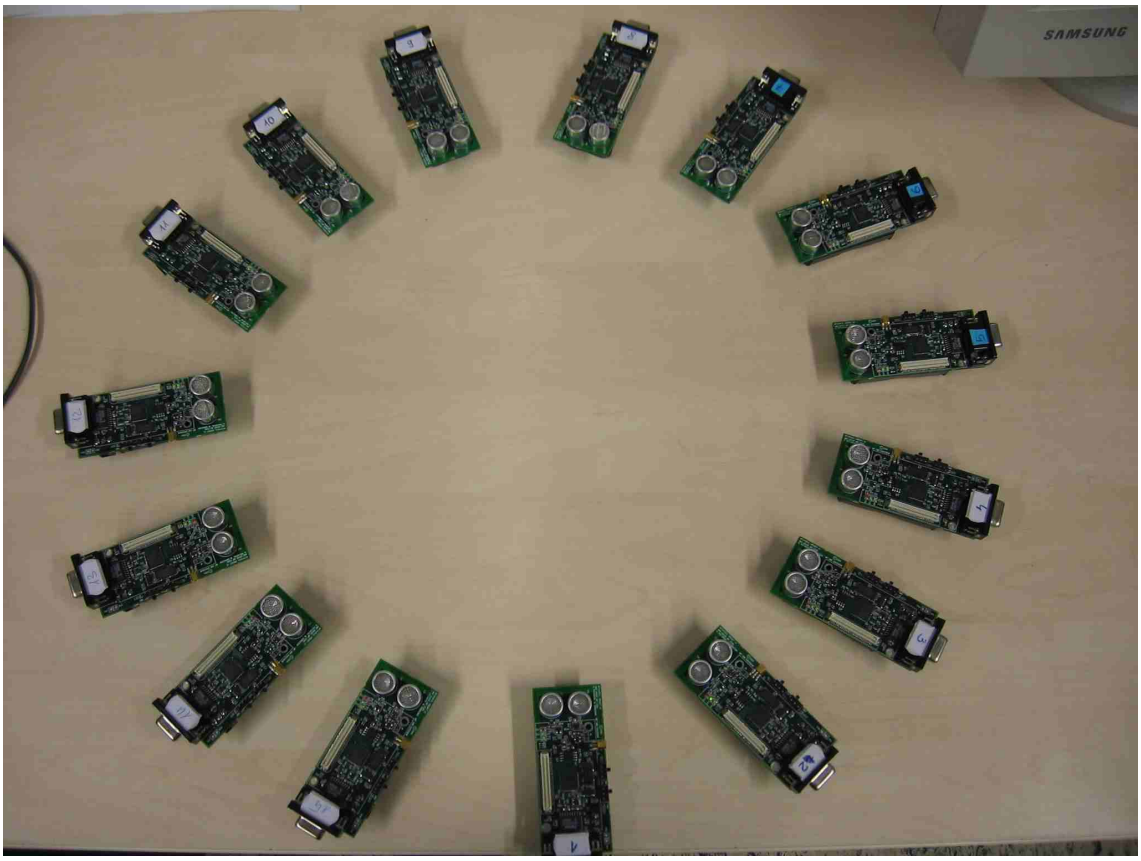


Figure 13: **15 motes of real experiment in a circle**

Figure 14: **Calculated positions of real experiment with 15 motes in a circle**

Table 5: **Startup and working times**

motes	startup time		working time	
	mean	std	mean	std
simulation				
5	11	5.19	0.475	0.01
10	61	33.63	0.51	0.01
15	70	32.37	0.5	0.007
30	259	94.94	0.52	0.01
real				
5	34	28	0.52	0.03
10	51	13	0.52	0.01
15	93	25	0.53	0.01

Table 5 shows mean startup times for the tinyos experiments. It can be seen that the startup time varies a lot depending on the time when the motes find a good starting order among each other. The working time is nearly constant, that is 6 beacons are emitted in each second regardless of the number of motes.

This system can be used, for example, on wheelchairs to be able to locate them, to predict or prevent collisions.

4.4 Outlook and prototype application

For the prototype application assume that there are handicapped people using wheelchairs and are moving within a house and in its neighborhood. On each wheelchair there are rf-mems units, which have limited radio bandwidth, computational power, and memory. The task of these units is to monitor the positions of the wheelchairs. Monitoring can serve different purposes, for

example, caretakers may wish to find one of the patients, help may be asked by the user of the wheelchair, or accidents need to be prevented. There are rf-mems units also on the walls of the house and a few may be placed to the garden around the house. The wheelchair units could establish their approximate locations,e.g., ultrasound and rf signals can be used to measure distances. Units have to measure the distance from neighboring wheelchair units and from the wall units continuously to determine the relative and the absolute locations. Because of the limited memory, limited power, and the limited communication bandwidth, rf-mems units of the wheelchairs, alone, may not be able to determine their position relative to each other and relative to the house. On the one hand, the information to be established is small. On the other hand, if not regulated properly, units will not be able to collect the necessary information. Information transfer should follow a particular order (i.e., a particular schedule) and that order may change from time to time. The Survivable Pipeline Protocol can be used to solve the computational and communication needs of determining the positions of the wheelchairs.

5 Summary

In my dissertation I have presented a distributed self-organizing protocol along with its main components. The protocol is able to set up and maintain executing of pipeline tasks in heterogeneous dynamic networks.

To achieve this, on the one part, I have extended the well-tried concepts and operations of high level synthesis to the networks. I've generalized the elementary operations of HLS as pipeline operations [38]. Doing so, one is able to plan a hierarchical system, that is, to use the previously prepared pipeline systems as one single operation in more complex systems. Moreover, in many scales a task can be described with operations of various difficulties that suit scale-free networks well.

On the other hand, using the newest achievements of machine learning, I have developed an algorithm, with the help of which the valuable resources can be searched for efficiently through the edges of the network. This algorithm enables the processing units to find the reliable partner processing units for task execution. I have developed and tested the algorithm in a real problem, in which I could freely change the network structure without affecting other parameters of the problem. My study focused on the task of searching for new that day information on the web [36, 35, 39]. I investigated algorithms using evolutionary, reinforcement learning, and combined evolutionary and reinforcement learning strategies. I experimented with environments having different degree distributions and clustering coefficients. I generated different topological environments, using data collected during real Web search. I found that in the scale-free small world environment the evolutionary weblog update algorithm performs better than the other algorithms. It outperformed a reinforcement learning based algorithm and the combinations of these two algorithms. I conjecture that the highly clustered

nature and the small diameter of the environment suites simple selection over other more sophisticated learning schemes. However, when the scale-free nature of the environment was kept but the small diameter of the environment was increased simply by restructuring the environment then other algorithms performed better than the simple selection. I found in these environments that the combination of the weblog and reinforcement learning algorithms are better than the individual algorithms. That is, when the diameter of the world becomes larger, then estimation of the long-term cumulated reward becomes important. Moreover, the combined algorithm showed the smallest performance variation both on the scale-free small world and on scale-free environments. In the web crawler experiments the valuable resources are the URLs from which the crawlers can find much new that day information. In the task execution problem the resources are the processing units. A processing unit is a good resource for an other processing unit if those can efficiently work together on a task. This algorithm enables the units to find the suitable partners to complete the tasks, just like the crawlers found the right start-off URLs.

Finally I have developed a scalable protocol for networks of computing, communicating and sensing units based on my studies of high level synthesis, reinforcement learning, selective learning, and the efficiencies of these algorithms in networks with different structures. The protocol is efficient in scale-free small worlds and can be extended with long-term cumulated profit estimation to be efficient in environments which do not meet the small world criteria. The protocol is scalable, it can be applied both for network of PCs and for network of motes, furthermore the computational capacity of the network scales linearly with the available computational capacity. The protocol is survivable, that is it adapts to the changing units in the network

and maintains the functions of the network while there are enough resources. Based on this protocol, reliable distributed medical sensing and therapeutic systems can be created [31, 29, 30].

To summarize, I have extended the elementary operations to pipeline operations and I have developed an efficient adaptive algorithm in scale-free small worlds and a combined algorithm (with reinforcement learning) in not small world environments. The protocol has four special features which originate from the fact that I've implemented the concepts and procedures over the hardware-software co-design on networks, combining them with the newest achievements of machine learning: i) scalability, applicable also in heterogeneous dynamic networks; ii) natural ability to execute hierarchical tasks; iii) the units can process and forward data in the pipeline way; iv) survivability, that is it adapts to the constantly changing units of the network and maintains the functions while there are enough resources in the network.

References

[1] R. Albert and A.L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47–91, 2002.

[2] M. Annunziato, R. Huerta, M. Lucchetti, and L. S. Tsimrig. Artificial life optimization over complex networks. In *Fourth International ICSC Symposium on ENGINEERING OF INTELLIGENT SYSTEMS*, March 2004.

[3] P. Arató, I. Jankovits, S. Szigeti, and T. Visegrády. *High–level Synthesis of Pipelined Datapaths.* PANEM, Budapest, 1999.

[4] P. Arató, T. Kandár, Z. Mohr, and T. Visegrády. An algorithm for decomposition into predefined ips. Technical report, Technical University of Budapest, 2000.

[5] P. Arató, T. Visegrády, and I. Jankovits. *High Level Synthesis of Pipelined Datapaths.* Wiley and Sons, Ltd., Chichester, England, 2001.

[6] A.L. Barabási, R. Albert, and H. Jeong. Scale-free characteristics of random networks: The topology of the world wide web. *Physica A*, 281:69–77, 2000.

[7] D.L. Boley. Principal direction division partitioning. *Data Mining and Knowledge Discovery*, 2:325–244, 1998.

[8] J. Cho and H. Garcia-Molina. Effective page refresh policies for web crawlers. *ACM Transactions on Database Systems*, 28(4):390–426, 2003.

[9] C.W. Clark and M. Mangel. *Dynamic State Variable Models in Ecology: Methods and Applications.* Oxford University Press, Oxford UK, 2000.

[10] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems concepts and design.* Addison Wesley, 4 edition, 2005.

[11] V. Csányi. *Evolutionary Systems and Society: A General Theory of Life, Mind, and Culture.* Duke University Press, Durham, NC, 1989.

[12] G. DeMicheli. *Synthesis and Optimization os Digital Circuits.* McGraw Hill, 1994.

[13] J. Edwards, K. McCurley, and J. Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *Proceedings of the tenth international conference on World Wide Web*, pages 106–113, 2001.

[14] A. E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing.* Springer, 2003.

[15] M. Eisenring and J. Teich. Domain-specific interface generation from dataflow specifications. In *Codes CASHE'98, the 6th Int. Workshop on Hardware/Software Codesign*, pages 43–47, 1998.

[16] R. Ernst. *Embedded System Architectures, System Level Synthesis*, volume 357 of *Nato Science Series.* Kluwer Academic Publisher, 1999. Edited by A.A. Jerraya and J. Mermet.

[17] J.M. Fryxell and P. Lundberg. *Individual Behavior and Community Dynamics.* Chapman and Hall, London, 1998.

[18] D. Gajski. *Silicon Compilation.* AddisonWesley, Reading, MA, Editor, 1988.

[19] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems.

In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2003.

[20] A. A. Jerraya and J. Mermet. *System Level Synthesis*. NATO Science Series. Kluwer Academic Publisher, 1998.

[21] Thorsten Joachims. A probabilistic analysis of the Rocchio algorithm with TFIDF for text categorization. In Douglas H. Fisher, editor, *Proceedings of ICML-97, 14th International Conference on Machine Learning*, pages 143–151, Nashville, US, 1997. Morgan Kaufmann Publishers, San Francisco, US.

[22] G. Kampis. *Self-modifying Systems in Biology and Cognitive Science: A New Framework for Dynamics, Information and Complexity*. Pergamon, Oxford UK, 1991.

[23] J. Kleinberg and S. Lawrence. The structure of the web. *Science*, 294:1849–1850, 2001.

[24] I. Kókai and A. Lőrincz. Fast adapting value estimation based hybrid architecture for searching the world-wide web. *Applied Soft Computing*, 2:11–23, 2002.

[25] T. Kondo and K. Ito. A reinforcement learning with evolutionary state recruitment strategy for autonomous mobile robots control. *Robotics and Autonomous Systems*, 46:11–124, 2004.

[26] A. Lőrincz, I. Kókai, and A. Meretei. Intelligent high-performance crawlers used to reveal topic-specific structure of the WWW. *Int. J. Founds. Comp. Sci.*, 13:477–495, 2002.

[27] Maja J. Mataric. Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4(1):73–83, 1997.

[28] F. Menczer. Complementing search engines with online web mining agents. *Decision Support Systems*, 35:195–212, 2003.

[29] A. Meretei, Zs. Palotai, and A. Lőrincz. Embedded neural prosthesis. US Patent and Trademark Office, NDC-5039USNP, 2006. pending.

[30] A. Meretei, Zs. Palotai, and A. Lőrincz. Posture and movement monitor. US Patent and Trademark Office, NDC-5040USNP, 2006. pending.

[31] A. Meretei, Zs. Palotai, and A. Lőrincz. Systems and methods for sensing physiologic parameters of the human body and achieving a therapeutic effect. United States Patent 20070043591, U.S. Patent and Trademark Office, 2007.

[32] D.E. Moriarty, A.C. Schultz, and J.J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:199–229, 1999.

[33] S. Oaks, B. Traversat, and L. Gong. *JXTA in a Nuthsell.* O'Reilley, 2002. http://www.jxta.org.

[34] E. Pachepsky, T. Taylor, and S. Jones. Mutualism promotes diversity and stability in a simple artificial ecosystem. *Artificial Life*, 8(1):5–24, 2002.

[35] Zs. Palotai, Cs. Farkas, and A. Lőrincz. Selection in scale free small worlds. In M. Pechoucek, P. Petta, and L. Zs. Varga, editors, *Multi-Agent Systems and Applications IV, 4th International Central and East-*

ern European Conference on Multi-Agent Systems, CEEMAS 2005, LNAI 3690, pages 579–582. Springer-Verlag, 2005.

[36] Zs. Palotai, Cs. Farkas, and A. Lőrincz. Is selection optimal for scale-free small worlds? In *European Conference on Complex Systems, Paris*, 2005. accepted.

[37] Zs. Palotai, B. Gábor, and A. Lőrincz. Adaptive highlighting of links to assist surfing on the internet. *Int. J. of Information Technology and Decision Making*, 2005. (to appear).

[38] Zs. Palotai, T. Kandár, Z. Mohr, T. Visegrády, G. Ziegler, P. Arató, and A. őrincz. Value prediction in hls allocation problems using intellectual properties. *Applied Artificial Intelligence*, 16:151–192, 2002.

[39] Zs. Palotai, S. Mandusitz, and A. Lőrincz. Distributed novel news mining from the internet with an evolutionary news forager community. In *Int. Joint Conf. on Neural Networks*, Budapest, Hungary, 26 - 29 July 2004. IEEE Operations Center, Piscataway, NJ 08855-1331. Paper No. 1095. IJCNN2004 CD-ROM Conference Proceedings, IEEE Catalog Number: 04CH37541C, ISBN: 0-7803-8360-5.

[40] N. Park and A. Parker. Shewa: A program for synthesis of pipelines. In *Proceedings of the 23. Des. Automation Conference*, pages 454–460, 1986.

[41] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioural synthesis of asics. *IEEE Transactions on Computer Aided Design*, 1989.

[42] K. M. Risvik and R. Michelsen. Search engines and web dynamics. *Computer Networks*, 32:289–302, 2002.

[43] A. Rungsawang and N. Angkawattanawit. Learnable topic-specific web crawler. *Computer Applications*, xx:xxx–xxx, 2004.

[44] W. Schultz. Multiple reward systems in the brain. *Nature Review of Neuroscience*, 1:199–207, 200.

[45] A. Stafylopatis and K. Blekas. Autonomous vehicle navigation using evolutionary reinforcement learning. *European Journal of Operational Research*, 108:306–318, 19998.

[46] J. Staunstrup and W. Wolf. *Hardware–Software Codesign: Principles and Practice*. Kluwer Academic Publisher, Editors, 1997.

[47] R. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 1988.

[48] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT Pess, 1998.

[49] I. Szita and A. Lőrincz. Kalman filter control embedded into the reinforcement learning framework. *Neural Computation*, 16:491–499, 2004.

[50] G. J. Tesauro. Temporal difference learning and td-gammon. *Communication of the ACM*, 38:58–68, 1995.

[51] K. Tuyls, D. Heytens, A. Nowe, and B. Manderick. Extended replicator dynamics as a key to reinforcement learning in multi-agent systems. In N. Lavrac et al., editor, *ECML 2003, LNAI 2837*, pages 421–431. Springer-Verlag, Berlin, 2003.

[52] F. Vahid and L. Tauro. An object–oriented communication library for hardware - software codesign. pages 81–87, 1997.

[53] T. Visegrády. *Hardware-Software Codesign in a High Level Synthesis Environment*. PhD thesis, University of New Hampshire, May 1999.

[54] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[55] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[56] J.S. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, and A.R. Newton. The javatime approach to mixed hardware-software system design. *NATO ASI SystemLevel Synthesis*, August 1998.