Scheduling in Multi-X
a performance evaluation

Dinand Roeland

RuG

Technische Informatica

Afstudeerverslag

Scheduling in Multi-X
a performance evaluation

Dinand Roeland

Rijksuniversiteit Groningen
Technische Informatica
Postbus 800
9700 AV Groningen

27 juli 2000

Abstract

This master's thesis project is carried out at Ericsson Utvecklings AB, the research and development centre for Ericsson's Network Core Products. One of the major products of Ericsson Utvecklings AB is the AXE telephone switching system. The need for capacity in AXE switches is increasing at an unexpected rate due to new services like ISDN, GSM and the Internet. To keep up with competition, AXE's central processor's capacity needs to be doubled every third year. A number of projects have been started in order to cope with this problem. One of them is the Gemini project, which will propose an architecture based on commercially available components. One of the main prerequisites for Gemini's new architecture is backward compatibility with existing AXE software. On the long term, the capacity of Gemini's architecture will not be sufficient. The Multi-X project investigates methods of utilising execution on parallel processors, with Gemini's architecture as a premise.

The focus of Multi-X is a new technique called task-level speculative execution. This technique is a form of implicit parallelism; parallelism is extracted from conventional imperative and unmodified code. With speculative execution, assumptions are made on the data and control flow in a program. By looking ahead in the instruction stream, instructions can be run in parallel. Instructions have to be rolled back, i.e. undone, on incorrect assumptions. Instruction-level parallelism is a commonly known technique used in most of today's modern processors. By the beginning of the 1990s, several studies proved the limitations of performance-increase using instruction-level parallelism. These limitations are mainly caused by imperfect predictions of future data and control flow. Since mid 1990s a trend can be observed to not only exploit parallelism at a fine-grained instruction level, but simultaneously also at a coarse-grained task level.

The Multi-X project has developed a prototype to prove the concept of task-level speculative execution on a multiprocessor system based on Gemini's architecture. This prototype is still being improved, tested and modified. No results on performance increase are available yet. As a final phase in Multi-X, the prototype can be optimised in a number of ways. One optimisation is task scheduling, where scheduling is defined as 'the process of deciding what task to execute where and when'. This thesis will focus on the consequences in performance of different task distribution principles.

In this thesis project, the Multi-X prototype is modelled and an optimal scheduling algorithm is defined in order to calculate an upper bound on performance. Furthermore, three practical scheduling algorithms are proposed and a simulator is developed to investigate their performance. Live-recorded telephone switch traffic is used as input data to calculations and simulations. The three proposed practical ways of scheduling are first-come-first-serve scheduling, function-based scheduling and source-based scheduling. With first-come-first-serve scheduling the next task is simply sent to an arbitrary idle processor. This algorithm is currently used in the Multi-X prototype. Function-based scheduling is an attempt to avoid unnecessary rollbacks by examining what functions a task will execute. Source-based scheduling is an attempt to map individual subscribers to individual processors in order to exploit cache-affinity and diminish inter-task dependencies.

The simulations performed in this thesis project are the first on the Multi-X concept where all relevant parameters are taken into account. The concept is proved to hold theoretically, whereas a critical note is made concerning the feasibility of a real implementation. Scheduling should really be seen as an optimisation. Only a moderate performance increase can be accomplished with scheduling. First-come-first-serve scheduling performs very well, even compared to optimal scheduling. Ideas from function-based scheduling can be used additionally to improve performance even more. Source-based scheduling results in poor performance, mainly caused by load imbalance.

A lot of issues remain to be tested, simulated and analysed. This thesis is only one step forward towards a better understanding of a multiprocessor architecture running telephone switch traffic.

Acknowledgements

I would like to thank my colleagues at Ericsson Utveckling AB, especially: Lennart Petterson for giving me the opportunity to work on this exciting project; Yusuf Sulaiman for offering a great deal of background knowledge on traffic data; Lars-Åke Johansson for helping me to verify traffic data; and Ruben Rizzi for the interesting and often hilarious conversations during numerous coffee breaks. Also thanks to Marcus Dahlström for providing detailed information on Multi-X and to Nikhil Tikekar for reflections on ideas and concepts.

I would like to thank Professor Ben Spaanenburg at the department of Mathematics and Computing Science, Rijksuniversiteit Groningen, for support and guidance during this project. Special thanks to Nico Kruithof for reviewing this report and for the fruitful co-operation during our studies.

Finally, I am very grateful for the support of my beloved wife Marika and my dear parents. Thank you for helping me to start master studies.

'*The fear of the Lord is the beginning of wisdom, and knowledge of the Holy One is understanding.*'
(Proverbs 9:10)

Dinand Roeland
Stockholm, June 2000

Table of contents

List of figures

List of tables

1 Introduction

This thesis is on speeding up sequential programs by parallelising them. Probably ever since the introduction of the Von Neumann architecture, people have been trying to overcome the strictly sequential execution imposed by this architecture. In 1967, Amdahl [2] argued not to have too high expectations on parallelising sequential programs by introducing a very straightforward paradigm. If N is the number of processors, s is the amount of time spent (by a serial processor) on serial parts of a program and p is the amount of time spent (by a serial processor) on parts of the program that can be done in parallel, then the potential speedup is given by Equation 1.

$$Speedup = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{p}{N}}$$

Equation 1 Amdahl's law

Even when using an infinite amount of processors, the maximal speedup is bound to s^{-1}. Although criticised later [21], Amdahl's law does point out that truly big gains in parallel programming can only be achieved by reducing serial bottlenecks.

The question remains how to exploit parallelism from a sequential program. The easiest way, from a computational point of view, is to push the burden on the programmer and force him to (re-)state the problem in a parallel program. This is called *explicit parallelism*. Another way is to use paradigms such as functional or logic programming where a compiler can readily generate parallel code [40]. This thesis considers *inherent parallelism* or *implicit parallelism*, that is, parallelism from conventional imperative programs, which have not been explicitly modified to extract parallelism. The drive for inherent parallelism comes from a number of directions. Firstly, the great majority of the world's code is written in imperative languages and it seems almost inconceivable that it could all be rewritten to explicitly take advantage of parallelism. Secondly, it is now difficult to take advantage of and keep busy the number of transistors available on modern CPU chips without making some use of parallelism. Thirdly, at some point in the next decade or two Moore's law will fail and the steady exponential increase in performance of silicon based computers will cease. Parallelism will then be needed to maintain performance increases [8].

Extracting inherent parallelism can be done at compiler-time, at run-time or both. Run-time techniques include out-of-order execution, register renaming, alias renaming, branch prediction and multiple-path execution. The processor can look ahead during execution and execute instruction *out-of-order* and in parallel, provided no logical inconsistencies arise as a result of doing so. With *register renaming*, the processor removes storage conflicts by providing additional registers to re-establish the correspondence between registers and values. The additional registers are allocated dynamically by hardware, and the registers are associated with the values needed by the program. *Alias analysis* is like register renaming, but for memory locations. Branches impede the ability of the processor to fetch instructions because they make instruction-fetching dependent on the result of instruction execution. When the outcome of a branch is not known, the instruction fetcher is stalled, reducing the chances that the processor can find instructions to execute in parallel. To overcome this problem, the outcome of a branch can be predicted. *Branch prediction* [58] can be done dynamically by hardware or statically by annotating the program with prediction information during compile-time. Until the true outcome of the branch is known, instructions on the predicted path can be executed speculatively. When using *speculative execution*, these instructions have to be undone (by performing a *rollback*) if the assumption was incorrect. Another way to avoid the impediments of branches is to have the processor pursue both paths at a branch. This technique is called *multiple-path execution*; the processor can simply discard the results of the incorrect path. Since branches happen quite often in normal code, we may encounter another branch before we have resolved the previous one. In order not to use up all machine resources, multiple-path execution and branch prediction can co-operate to achieve good results. Extracting inherent parallelism at compiler-time is based on a static analysis of the code to execute. Using this analysis, the instructions can be *(re-) scheduled*. Here, scheduling can be defined as the process of arranging the order of instructions in object code so that they are executed by hardware in an optimum order. A common scheduling technique is *loop unrolling*. When a loop is unrolled, the instructions for two or more loop iterations are written explicitly. This gives the processor a larger scope to look ahead and execute instructions in parallel. Techniques to extract inherent parallelism at a fine-grained instruction-level are also known under the common name of *instruction-level parallelism* [31, 70, 71]. Most of the techniques mentioned here were discussed in the classic article "Look-Ahead Processors" by Keller in 1975 [34].

In the beginning of the 1990s, several studies [37, 70] proved the limitations of instruction-level parallelism. Using only fine-grained techniques, the speedup tends to be limited to a factor of approximately ten. The reason for this is that instruction-level parallelism relies heavily on good branch prediction algorithms. Even if prediction algorithms have a 90% accuracy [31, 49], this will only leave 60% after five speculative branches. In typical programs one in seven instructions is a branch [25], which does not leave much scope for extracting parallelism. Because of this limitation and the ever-growing amount of transistors on processor chips, a trend can now be observed to not only exploit parallelism at a fine-grained level, but also at a coarse-grained level [10, 61, 63]. This is called *task-level parallelism* or *thread-level parallelism*.

Several architectures for extracting task-level parallelism have been proposed, including *Multiscalar Processors* of the University of Wisconsin-Madison [59], the *Superthreaded Processor Architecture* [67], the *WarpEngine* of the University of Waikato [9], Stanford's *Hydra* [48], MIT's *M-Machine* [20] and the *Simultaneous Multithreading* architecture [69]. It is here where Ericsson's *Multi-X* prototype comes in. This thesis is a part of the Multi-X project. The next chapter describes the background of the Multi-X project and specifies the focus of this thesis project.

2 Background

Ericsson Utvecklings AB is the research and development centre for Ericsson's Network Core Products. One of the products of Ericsson Utvecklings is the *AXE telephone switching system*. AXE is the world's most deployed communication architecture with 130 million lines in service in some 125 countries. This thesis project has been done at the *System Processors* department, that is responsible for the research and development of AXE's central processor system called *APZ CP*. The latest version available on the market is the APZ 212 30 [28].

This chapter describes the context of this thesis project. The first section outlines the capacity problem Ericsson faces today. The next section introduces Multi-X as one of the solutions to this problem. Section three gives a general overview of the Multi-X prototype and the succeeding section describes related prototypes. The last section of this chapter specifies the goal for this thesis project.

2.1 Problem context

Due to new services like GSM, ISDN and Internet, the need for capacity in the APZ CP is increasing at a previously unexpected rate. To keep up with competition, APZ's capacity needs to doubled every three years. In order to cope with this problem, a number of projects have been started. One of these is the *Gemini* project. Until now, Ericsson itself designs all hardware (ASICs). The Gemini project proposes an architecture based on commercially available components. This way, a number of design problems can be pushed onto external suppliers like Compaq, SUN or Intel. It will also give Ericsson designers more flexibility when designing new versions of the APZ CP. An enormous amount of software has been written for earlier versions of the APZ CP. Therefore, Gemini's new architecture is backward compatible. No modifications in existing code are allowed. A virtual machine is used to fit existing code into the new architecture. Both the latest version of the APZ CP and the proposed architecture by Gemini are single processor architectures executing code sequentially, but exploiting instruction-level parallelism internally.

2.2 Multi-X as a proposed solution

It has been proved that there is a significant amount of task-level parallelism to be exploited in APZ CP code, which might lead to an increase in performance (conform Amdahl's law) [62, 72]. The proposed method to exploit this parallelism is *task-level speculative execution*. Within the *Multi-X* (Multiple Executor) project, methods of utilising execution on parallel processors are investigated. More specifically, Multi-X has been started in order to evaluate and demonstrate techniques based on *task-level speculative execution* and *resource locking* [51, 52]. Multi-X will be based on the (expected) results of Gemini. This way, the performance of unmodified APZ CP code will increase while using commercially available hardware components. Like speculative execution on an instruction level, speculative execution on a task-level will be limited by the intrinsic features of the existing code that has been written with a single sequential processor in mind [72]. Therefore, the Multi-X project has to be seen as a first step in the architectural evolution from a single processor and sequential programs towards multiple processors and explicitly parallelised programs [65]. The Multi-X project is carried out in co-operation with Chalmers University of Gothenburg [6].

Two projects within Ericsson that have involved parallel processing are CMC and Tor. The former resulted in an architecture for which existing software had to be modified. The costs for implementation would have been too high. Trying to overcome these problems, a number of projects have been started. One of them is Multi-X. The latter has even investigated speculative execution but was cancelled because of a strategic change to build a future architecture on commercially available components.

2.3 Overview of the Multi-X prototype

The APZ CP can be seen as an *online transaction processing* system. The structure of an APZ CP transaction is visualised in Figure 1. Transactions can be divided into one or more uninterrupted sequences of program functions. Such a sequence will be called *transaction task* or simply *task*. A *function* is an uninterrupted sequence of instructions, as in imperative programming languages. Transactions are started by an *external message*. These messages originate from one of the *remote processors (RP)*, which handle events (e.g. the starting of a phone call). A task is initiated by a message and the executions of a task might give rise to one or more *internal messages*. We define the tasks started by these messages to be the *followers* of the initiating task. A message includes the address of the first function to be executed and data parameters [16, 17].

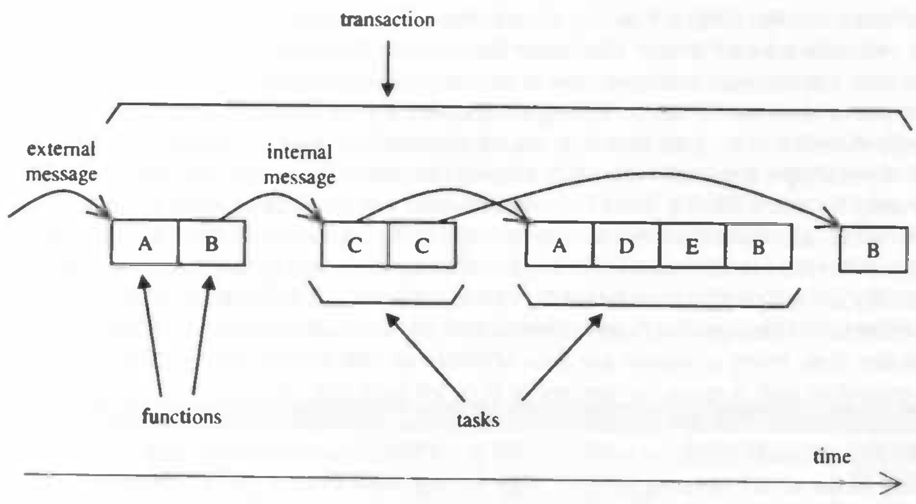

Figure 1 Relationship between transactions, tasks, functions and messages

Figure 2 gives an overview of the Multi-X architecture. A traffic generator simulates external events. The *signalling processor unit (SPU)* administers all messages, both external and internal, in a *first-in-first-out (FIFO) message queue*. Tasks are executed by assigning a message to an *instruction processor unit (IPU)*. IPUs exploit instruction-level parallelism. Main memory is shared, while every IPU has its own cache. Except for the number of IPUs, the Multi-X prototype resembles the architecture of the existing APZ CP. This way, the restriction not to change existing code can be met more easily. The prototype is implemented on a Sun Enterprise Fileserver with four processors; three of these are used as an IPU.

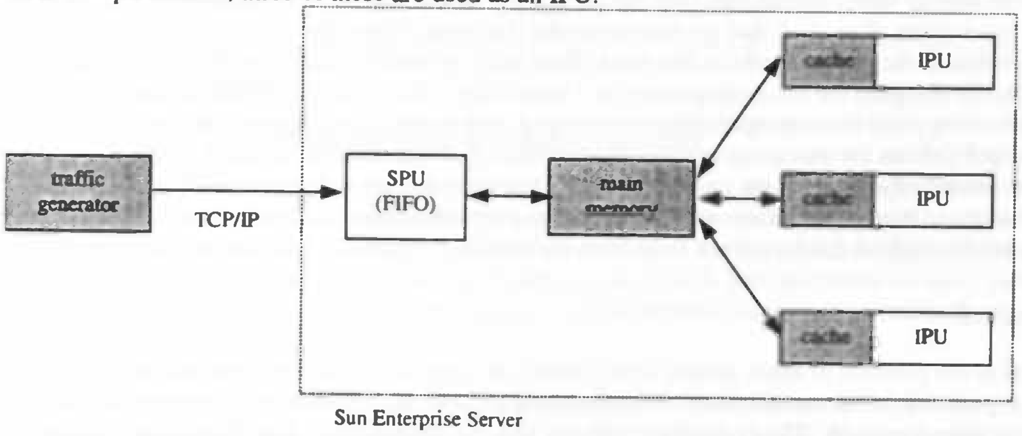

Figure 2 Architecture of the Multi-X prototype

In the proposed Gemini architecture (and therefore also in the Multi-X architecture), functions are written in the PLEX-C language, which is specifically designed for telecommunication exchanges. PLEX-C code is compiled into ASA assembly code for the APZ CP. ASA code is interpreted[1] by a virtual machine (AXE VM) in order to run existing code on a new architecture based on commercially available components [18, 28, 51].

In the initial phase of the Multi-X prototype, *resource locking* without speculative execution was investigated. This proved not to increase performance [72, 73], and speculative execution was included in the Multi-X prototype. The next task is simply assigned to an idle IPU, assuming no data collisions will occur. This is called *data speculation* and implies the execution of an instruction before the execution of a preceding instruction on which it may be or is data dependent. *Control speculation*, e.g. branch prediction, implies the execution of an instruction before the execution of a preceding instruction on which it is control dependent [41]. To date, the Multi-X prototype only supports data speculation. Speculation is done blindly; no predictions are made whether

[1] For optimisation reasons, most code will eventually be (JIT-) compiled. However, conceptually, ASA code is still interpreted.

speculations will be successful or not. When inserted in the FIFO, every task receives a task number[2]. All common data areas have a marker field, containing the numbers of the tasks that have lastly read or written the data area. Common variables are global only within one function, conform static variables in C. To access variables in another function, internal messages have to be used (conform methods in C++). In ASA code, there are basically three types of common variables: 1) single variables, e.g. an integer; 2) single arrays, a collection of elements of one single variable; 3) arrayed structure, where a structure is a set of single variables and single arrays. In the Multi-X prototype, every element of an arrayed structure has one marker field and every single variable and single array have one marker field. By comparing the marker field with the number of the executing task on every read or write, data collisions can be detected. On collision, one or more tasks are rolled back. When completed, a task is ready to be *committed*, i.e. all modifications made by this task to common variables are made permanent. By allowing only the oldest task to be committed, the original sequential execution order is preserved (*in-order commit*). Messages originating from a task are not inserted in the FIFO until the task has been committed. At any time, every processor executes at most one task and no other task is started on a processor until the executing task is ready for commit or is rolled back [26, 56].

Performing commits and rollbacks might be implemented in different ways [27]. One way is to carry out data modifications directly in the actual memory location after having copied the original contents to a temporary area. Another way is to copy data from the actual memory into a separate memory to read from and write to. The first way is called the *original method* and will result in relatively fast commits but slow rollbacks. The second way is called the *temporary method* and will result in slow commits but fast rollbacks. At this moment, Multi-X uses the original method.

Two *collision detection algorithms* have been proposed. The first was proposed by *Hjalmarson* [26] and allows only one speculatively executed read or write per common data area at any time. If more are requested, the youngest task waits. If data collision is detected, a rollback on the colliding task(s) and all younger tasks is performed. The marker fields contain one field for the number of the task that has lastly written the data area and one field for the number of the task that has lastly read the data area. These fields are reset at initialisation and at rollback or commit of the task assigned to that field. The total number of uncommitted tasks is virtually unlimited. Tikekar designed the second algorithm at a later stage. This algorithm is more aggressive (i.e. avoids waiting) by allowing more than one speculatively executed read or write per common data area at any time. The *speculation depth* defines the maximum number of speculatively executed data accesses. At any time, the number of uncommitted tasks does not exceed the speculation depth. This way, the size of the marker fields is linear with the speculation depth. Moreover, the algorithm prevents unnecessary rollbacks. Both algorithm are often referred to throughout this report and have been included in "Appendix A Collision detection algorithms".

2.4 Related work

As mentioned in the previous chapter, several other projects attempt to exploit both instruction-level parallelism and task-level parallelism. All architectures mentioned here are, like the Multi-X prototype, multiprocessor systems with a shared memory. The *M-Machine* [20, 33] and the *Simultaneous Multithreading* architecture [69] exploit instruction-level parallelism by statically scheduling independent instructions to be executed simultaneously on independent functional units of a processor. Task-level parallelism is exploited by dynamically interleaving the instruction streams of several processors in order to utilise unused functional units. These architectures rely heavily on hardware synchronisation mechanisms and compile-time program transformations. Neither architecture uses some form of speculative execution.

In the *Multiscalar Processors* approach [29, 41, 59], a sequential program is statically represented as a control flow graph. The nodes are basic blocks (tasks) and arcs from one node to another represent the flow of control. A sequencer speculates which task will be executed next and assigns this task to the next free processing unit. On mis-speculation, a rollback will be performed on that task and all its successors. To facilitate sequential semantics, tasks commit in the same order as they are assigned to processing units. Within a task, data values might be speculated and a unidirectional ring between the processing units is used to forward computed data values from the oldest to the youngest task. The *Superthreaded Processor Architecture* [67] is similar to the Multiscalar architecture, but does not include data speculation. As a result, less memory resources are needed for administration. The *WarpEngine* [8, 9] organises up to 16 instructions into one task. Speculative execution on

[2] In the current implementation of the Multi-X prototype, a task is not assigned a number until its message is extracted from the FIFO. Since only the oldest message is extracted at any time, this is conceptually the same as assigning a number when inserting a message.

control or data selects complete tasks for execution. Each task is dynamically allocated a time stamp when it is scheduled for execution. This time stamp is used for all read and write requests issued from within the task. The time stamps impose a single linear temporal order relating all the reads and writes. The way this is done is to associate with each block an interval of time and to allow it to schedule a number of child tasks. The intervals of the children are disjoint and contained within the parent interval. All the time intervals form a tree and a sequential execution of a program will consist of a pre-order traversal of this tree. Rollback of instructions is initiated when the memory returns a revised value from a read. Only instructions (and children) dependent on the revised value have to be rolled back and re-executed. *Hydra* [24, 47, 48] is targeted for multiple threads, originated from multiple programs or a single compiler-annotated program. Hydra is influenced by the Multiscalar approach, but focuses on the parallelisation of loop iterations. Speculation is mainly on data, using data forwarding techniques like Multiscalar. Control speculation is minimised by placing a larger demand on the compiler to divide programs in tasks and by introducing some run-time software control. Table 1 summarises a number of features of the architectures mentioned.

Architecture	Target	Speculation	Support	Commit	Rollback	Scheduling
Multiscalar	single sequential program	control and data	compile-time transformations, run-time hardware	in-order	task and all its successors	next task to next free processing unit
Superthreaded	single sequential program	control	compile-time transformations, run-time hardware	in-order	task and all its successors	next task to next free processing unit
WarpEngine	single sequential program	control and data	run-time hardware, compile-time transformations suggested as optimisation	in-order, out-of-order suggested as optimisation	partial rollback only instructions directly affected	heuristics using time stamp
Hydra	multiple threads	data	compile-time transformations, run-time hardware, run-time software	in-order	per task	round-robin in loop iteration
Multi-X	online transactions	data	run-time software	in-order	task and all newer tasks	next task to next free processing unit

Table 1 Features of architectures using speculative execution

Compared to architectures using speculation, Multi-X is unique in a number of ways. Firstly, the Multi-X prototype is targeted specifically for online transaction processing, not for sequential or multi-threaded programs. The dynamic behaviour of online transactions makes static analysis difficult. Secondly, the Multi-X prototype as proposed so far is a run-time software-only approach. The restriction on using only commercially available component implies no hardware support for speculative execution. The restriction on unmodified code implies no compile-time support.

The Hydra approach resembles the Multi-X prototype best. This architecture is able to speedup online transaction processing by a factor three compared to a single processor [23]. Unfortunately, Hydra research focuses on loop constructs, which are rare in APZ CP code. However, the techniques used are also applicable to exploit method-level parallelism. In method speculation, sequential method invocations in object-oriented programs are mapped onto speculative tasks that are executed in parallel with the in-order thread. This does resemble Multi-X task invocation by messages more accurately. An empirical study proves method speculation to be able to speedup Java programs by a factor of approximately two [7].

A number of products have emerged from the research architectures mentioned above. One is a new compile-time loop parallelisation technique for Java programs [32]. Two processors exploiting task-level parallelism by speculation are Sun's MAJC architecture [64] and NEC's MP98 [43]. A feasibility study on the use of these processors is conducted in the Multi-X project.

2.5 Thesis project specification

The Multi-X prototype might be optimised in a number of ways. One optimisation proposed to analyse the functions' code statically and extract parts that certainly can be executed in parallel. This approach alone, however, proved no substantial performance gains [12].

Another way to optimise the performance of the Multi-X prototype is to modify existing *parameters*; e.g. the collision detection algorithm, the granularity of tasks, the granularity of the marker area, the cache coherence protocol or the task scheduling. It is unknown to what extent these parameters influence the overall performance and how they are co-related. The task numbering needed for the proposed collision detection algorithms impose a too strict order on the tasks. Allowing tasks to be committed out-of-order might also give some performance gains. Rollback and commit is done per task, but the *task granularity* might also be defined as a single function or a complete transaction (see Figure 1). Collision detection is based on *marker areas*, which are now defined as single variables, single array or an element of an arrayed structure. Other granularities are possible; e.g. a number of related variables. Caches are known to influence performance substantially, and different *cache coherence protocols* might influence the overall performance of the Multi-X prototype. This is investigated in a master thesis project [53] carried out in parallel with this master thesis project.

This thesis will focus on the consequences of different task distribution principles, also called *task scheduling*. A preliminary survey has been done, resulting in six straightforward distribution principles [30]. The goal of this thesis is to gain a more profound understanding of how scheduling influences the overall performance of the Multi-X prototype; i.e.: how does scheduling relate to other parameters in the Multi-X prototype and what is its importance?; how can tasks be scheduled?; what performance can we expect from different ways of scheduling? In the next chapters, concepts will be defined more clearly and the questions mentioned above will, at least partly, be answered.

The next chapter explores scheduling, starting with definitions and a summary of scheduling in other projects. The last two sections focus on scheduling in Multi-X and define a number of scheduling algorithms to investigate. Chapter 4, "Measuring scheduling performance" defines how the performance of the proposed scheduling algorithms will be measured. The succeeding chapter presents measurement results and this report ends with conclusions in chapter 6.

3 Exploring scheduling

In this chapter, scheduling as a whole is explored, starting with general definitions and ending with Multi-X specific proposals. The first section gives a theoretical framework on scheduling. The second section explores scheduling solutions proposed by related projects. Section three defines the position of scheduling in Multi-X, in order to propose a number of relevant scheduling methods in the last section.

3.1 Definition of scheduling

If a task is generally defined as an uninterrupted sequence of program functions or instructions, then *scheduling* can be defined as *'the process of deciding what task to execute where and when'*[3]. Scheduling decisions are made on (dynamic) properties of the objects to be executed or on (dynamic) properties of the environment in which the objects are executing. The goal of scheduling is to improve performance (e.g. the total execution time) by minimising one or more costs (e.g. the number of cache misses in cache-affinity scheduling [68] or the number of missed deadlines in real-time environments [1]).

A large number of scientific papers have been written on scheduling, a lot of them introducing a new way of scheduling to minimise some specific cost for some specific environment. Casavant [5] proposes a taxonomy of scheduling in distributed computing systems. Figure 3 is an abstract of this taxonomy.

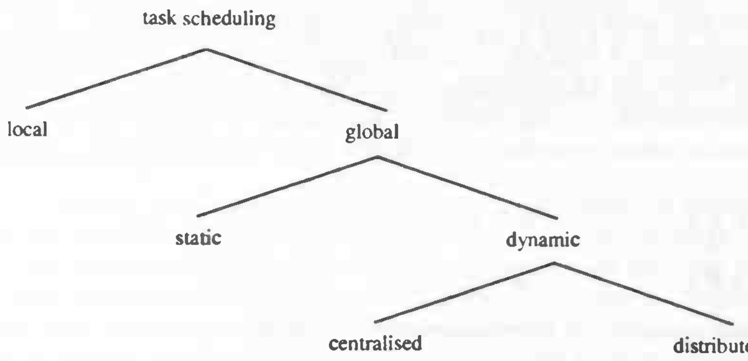

Figure 3 Casavant's taxonomy of task scheduling in distributed computing systems

Local scheduling is involved with the assignment of processes to the time-slices of a single-processor system. *Global scheduling* is the problem of deciding where to execute a process, and the job of local scheduling is left to the operating system. Given perfect information at compile-time about the execution time and the memory referencing behaviour of the individual tasks, a *static scheduling* strategy can pre-compute an optimal schedule. Since all scheduling decisions are made at compile-time, there is no run-time overhead associated with static scheduling. In many applications, however, the necessary information is not available at compile time, which may lead to sub-optimal performance. To compensate for this lack of a priori information, *dynamic scheduling* postpones the assignment of tasks to processors until the program is executing. Scheduling decisions are then adjusted to match the dynamically changing conditions encountered at run-time. Centralised and distributed dynamic scheduling strategies have been proposed. Typically, the *distributed strategies* spread the scheduling operation over the processors so that idle processors assign work to themselves from a central or distributed queue of available tasks. The *centralised strategies* consider storing global information at a centralised location and use this information to make more comprehensive scheduling decisions. To do this, they use one or more dedicated processor's computing and storage resources. A major issue to take into account in designing distributed or centralised dynamic scheduling algorithms is the overhead associated with executing such algorithms. In many of the existing techniques, the time required to perform this scheduling adds directly to the total program execution time [22].

[3] Note that scheduling is not restricted to one level. Within a task, instructions might be (re-) scheduled. See also the definition of scheduling in chapter 1, "Introduction".

3.2 Learning from other projects

Before going deeper into scheduling targeted specifically for Multi-X, it is interesting to see what other projects have achieved on this topic. In order to relate as much as possible to the Multi-X prototype, four areas have been investigated: 1) scheduling in similar projects at Ericsson; 2) scheduling in instruction-level parallelism; 3) scheduling in task-level parallelism; 4) scheduling in online transaction processing.

3.2.1 Similar projects at Ericsson

The only project conducted at Ericsson similar to Multi-X is the Tor project [50]. This project had the same premises as Multi-X, but was a hardware-only approach. The Tor project did not provide any new ideas on scheduling. The scheduling algorithm used was the same as the initial scheduling algorithm in the Multi-X prototype; the next task in the queue is simply sent to an idle IPU. No other ways of scheduling have been investigated.

3.2.2 Instruction-level parallelism

Scheduling for instruction-level parallelism [25, 31, 42, 70, 71] deals with distributing a single instruction stream among the processor's functional units and reordering instructions. Scheduling is constrained by data and control dependencies. Data dependencies can be divided into four groups. For *true data dependencies*, the result of the first instruction is an operand of the second. These are also called *read after write (RAW)* dependencies. The second group is *anti-dependencies*, where the first instruction uses the old value in some location and the second sets that location to a new value. These dependencies are also *called write after read (WAR)*. Instructions have an *output dependency* or a *write after write (WAW)* dependency, if both instructions assign a value to the same location. The last group of dependencies is *read after read (RAR)* dependencies, where two instructions read from the same location. WAR, WAW and RAR dependencies are false data dependencies and can be resolved using register renaming and alias analysis (see also chapter 1, "Introduction"). There is a *control dependency* between a branch and an instruction whose execution is conditional on it. Branch predictors can partially resolve control dependencies.

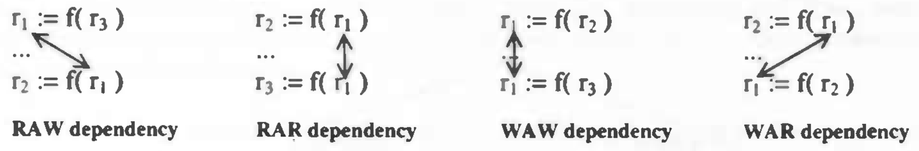

Figure 4 Classes of data dependencies

At compile-time, a sequential program can be divided into basic blocks. A basic block is a code sequence that does not contain a branch or a branch target, except at the beginning or at the end. Scheduling such code sequences is easy, since we know that every instruction in the block is executed if the first one is. We can simply make a graph of the dependencies among the instructions and order the instructions so as to minimise the stalls. The *critical path* of such a graph is that path with the minimum execution time of the entire sequence – no amount of instruction concurrency can make the execution time shorter. The critical path places a lower bound on the execution time and identifies the operations that determine this execution time. The optimal scheduling can be calculated, but this is often time-consuming. Mostly, a near optimum is calculated using heuristics. Examples of such scheduling algorithms are *list scheduling* and *branch-and-bound scheduling*. List scheduling encompasses a class of algorithms that schedule operations one at a time from a list of operations to be scheduled, using prioritisation to resolve contention for execution units. Branch-and-bound scheduling [45] considers all possible instruction orderings and selects the optimum ordering. Heuristics are used to limit the size of the search tree.

Though scheduling basic blocks causes the execution time of each basic block to be nearly optimal, this does not necessarily cause the execution time of the overall program to be nearly optimal, because the processor hesitates at each branch. The ability to look across basic-block boundaries gives the scheduler more flexibility to create a good schedule. Also, the scheduler knows about critical paths that span several basic blocks. Examples of these scheduling algorithms are *trace scheduling*, *loop unrolling* and *software pipelining*. A trace is a possible path through a section of code, spanning more than one basic block. The sequence of instructions in the trace is determined by assuming a particular outcome for every branch in the sequence. The effectiveness on trace scheduling depends on software knowing the likely execution trace. With trace scheduling, the execution time of likely traces is reduced at the expense of unlikely ones. Compensation code is inserted to recover from incorrect predictions. With loop unrolling, the instructions for two or more loop iterations are written explicitly. The result is a larger loop that executes half as many times. Unrolling exposes more instructions per loop iteration to the scheduler. Software pipelining is also targeted for loop iterations and might be used in combination with loop unrolling. Basically, if the processor has *n* execution units, iteration *i* is executed sequentially on execution unit *i* modulo *n*. So the first iteration is execution on the first execution unit. After a certain time interval, the second iteration is executed simultaneously at the second execution unit. The length of the time interval is determined by the dependencies of the variables in the iteration's basic block.

Just the main scheduling algorithms for instruction-level parallelism have been mentioned here. All algorithms can be implemented for both static scheduling (during compilation, or at least before execution) and dynamic scheduling (at run-time), although some are more appropriate for static scheduling. We have made a difference between scheduling only within one basic block and scheduling beyond basic block boundaries. In reality, this distinction is not that clear; e.g. *branch scheduling* where a scheduler simply tries to fill stall cycles after a branch with useful instructions. The techniques mentioned might also be combined, e.g. [38]. When applying scheduling techniques for instruction-level parallelism to Multi-X, a number of observations can be made. Firstly, the APZ CP is a transaction processing system. This means little static information is available, unlike static scheduling techniques for instruction-level parallelism. A solution for this is to statically calculate all possible traces for every external signal and to base scheduling decisions on this information. As mentioned in the previous chapter, this approach has already been investigated and proved no substantial performance gains [12]. Secondly, within every function (see "Figure 1 Relationship between transactions, tasks, functions and messages"), basic blocks can be identified and a near-optimal scheduling can be calculated statically. This approach is seen as an optimisation that is beyond the scope of this thesis project. This thesis will focus on higher-level (task-level) scheduling. This is, for the time being, assumed independent of lower-level (basic block-level or instruction level) scheduling. Lastly, some scheduling techniques for instruction-level parallelism are loop transformations. As mentioned, in APZ CP code, loops are rare. This excludes techniques like loop unrolling and software pipelining. In section 3.4, "Different ways of scheduling for Multi-X", some of the ideas from scheduling for instruction-level parallelism will be used for scheduling within Multi-X.

3.2.3 Task-level parallelism

Section 2.4, "Related work", shows the results of a literature survey on scheduling in task-level parallelism. Table 1 shows the different ways of scheduling in the different architectures. Since loop constructions are rare in APZ CP code, round robin scheduling as in Hydra cannot be used here. The WarpEngine proposed scheduling based on time-stamp heuristics, but this is not elaborated. The other prototypes implement scheduling as is done now in the Multi-X prototype. From this literature survey, it becomes clear that this thesis project is the first in its kind to investigate the consequences of scheduling in task-level parallelism!

3.2.4 Online transaction processing

Until now, the focus of this section has been on scheduling seen from a speculative execution viewpoint. We might, however, see the APZ CP as an online transaction processing (OLTP) database system and see how close this resembles the Multi-X prototype. From this, we might find some ideas for scheduling in Multi-X.

In traditional transaction processing systems [4], transactions have four critical properties known under the acronym ACID (Atomic, Consistent, Isolated, Durable). A database state transition is *atomic* if it executes completely or not at all (commit versus rollback). A transaction is *consistent* if it preserves the internal consistency of the database. *Isolation* means that a program running transactions in a multi-user environment must behave exactly as it would in a single-user mode. This topic is also called concurrency control (the problem), serialisability (the underlying theory) or locking (the technique). *Durability* requires that the results of transactions having completed successfully must not be forgotten by the system; from its perspective, they have become part of reality. The transaction processing system guarantees atomicity, isolation and durability. Consistency is a responsibility shared between transaction programs (the user) and the transaction processing system.

Two-phase locking mostly attains the isolation property. It says that a transaction must get all of its locks before releasing any of them. The two-phase locking theorem states that if all transactions are two-phased locked, then the execution is serialisable. This implies a basic difference with the Multi-X prototype. Let us take Figure 5 as an example where two transactions share a variable *x*. Let us assume (for this example) that every transaction in Multi-X consists of only one task.

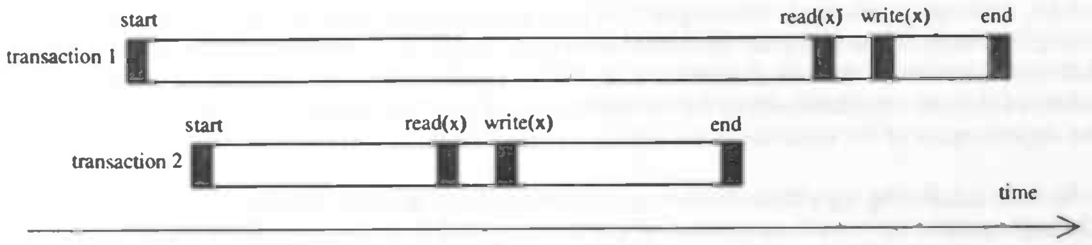

Figure 5 Two transactions running in parallel

In traditional transaction processing system with two-phase locking, no collisiön will occur and transaction 2 will commit before transaction 1. In Multi-X, the transaction number (i.e. the transaction start time) dictates the commit order. Since the prototype prescribes in-order commit, transaction 2 will not commit until transaction 1 has committed. Locked data areas are not released until commit, which causes a data collision at the reading of x in transaction 1. Running transactions on a transaction processing system with two-phase locking results in *some* serial execution order of those transactions. It is this basic property that contradicts the Multi-X prototype, where transactions run in *the* serial execution order, dictated by the transaction number. The reason for in-order commit is the requirement for an exact similar behaviour to the serial execution on today's single-processor system. With in-order commit, this can be proved and implemented fairly easy. The Multi-X prototype might later be extended as to relax the strict commit ordering, but for now this is beyond the scope of this thesis project.

Because of this difference, scheduling techniques for online transaction processing have not been investigated more thoroughly. Two related fields of research where the ACID properties are relaxed or the transaction structure is more advanced are *workflow management* [13, 39] and *real-time databases* [1, 3, 35].

A workflow consists of many steps, where each step executes as a (traditional) transaction. Workflow management proved to focus mainly on long-lived computations resembling business processes. Often, workflow transaction models even allow human intervention. This does not correspond with the high-speed, near real-time transaction processing of the Multi-X prototype. Scheduling with time constraints in workflow management is still a relatively unexplored field of research [14]. No relevant scheduling algorithms have been found in workflow literature.

Like a traditional database system, a real-time database system must process transactions and guarantee that the database consistency is not violated. However, conventional database systems do not emphasise the notion of time constraints or deadlines for transactions. The performance goal of a system is usually expressed in terms of desired average response times rather than constraints for individual transactions. Thus, when the system makes scheduling decisions (e.g., which transaction gets a lock, which transaction is aborted), individual real-time constraints are ignored. A real-time database strives to minimise the time constraints that are violated, since it is very difficult to guarantee all time constraints. Whereas traditional database systems mostly use pessimistic concurrency (e.g. two-phase locking), real-time databases sometimes use optimistic concurrency control. Here, the execution of each transaction consists of three phases: a read phase, a validation phase and possibly a write phase, in that order. During the read phase, all writes take place on local copies. Then, if it can be established during the validation phase that the changes the transaction made will not violate serialisability with respect to all committed transactions, the local copies are made global. Only then, in the write phase, these copies become accessible to other transactions. When comparing this to the prototype of Multi-X in section 2.3, "Overview of the Multi-X prototype", it becomes clear that we might view Multi-X as an implementation of a real-time transaction processing system. However, scheduling algorithms for real-time databases base decisions mostly on transaction properties like release time (earliest time a transaction can start), deadline or estimated duration of the transaction. Only the first property is available in the APZ CP transactions. Deadlines are avoided by re-routing traffic at a higher management level.

3.3 Position of scheduling in the Multi-X prototype

In order to define a number of scheduling methods for Multi-X, we first need to have a better understanding of the position of scheduling in the Multi-X prototype. Taking Casavant's classification (Figure 3) as a guideline, only global scheduling will be considered in this thesis. Clearly, global scheduling in Multi-X will have to be dynamic because of the dynamic behaviour of online transaction processing. The architecture outlined in "Figure 2 Architecture of the Multi-X prototype" tends to centralised scheduling, but does not exclude distributed scheduling.

The overall goal for scheduling in Multi-X is to improve speedup (= performance) by minimising execution time (= the cost). The question is how to minimise execution time. To be able to answer this question, we need to know what parameters affect scheduling. Figure 6 gives an overview of those parameters. This set of parameters has been obtained from a number of sources, including the definition of the Multi-X prototype (see section 2.3, "Overview of the Multi-X prototype") and [10]. These are the most important parameters, but more maybe yet unknown or underestimated.

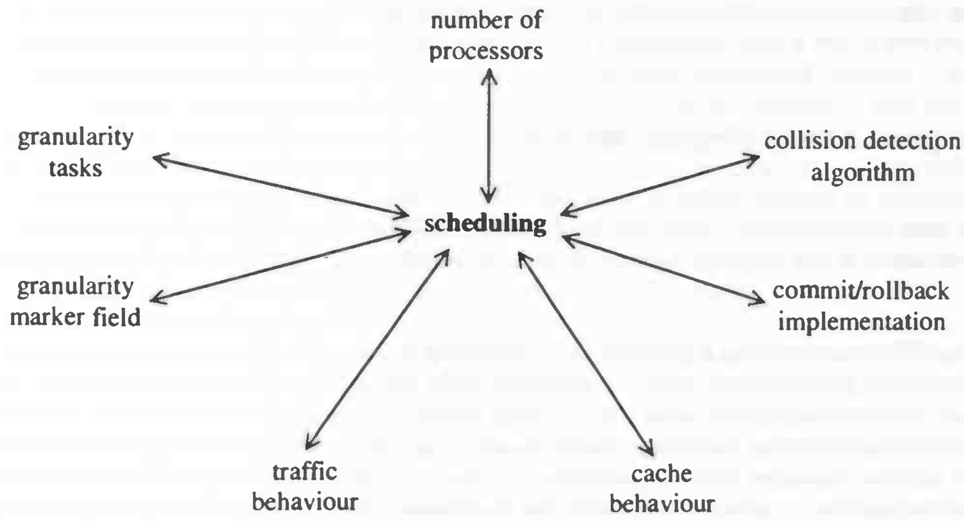

Figure 6 Relation of scheduling to other parameters in the Multi-X prototype

- The *number of processors* will influence scheduling performance dramatically. When measuring scheduling performance, the number of processors will have to be variable. More processors give more freedom to the scheduler and more processing capacity, but also more data collisions between the scheduled tasks. The graph speedup = f(number of processors) will therefore be a curve with a top at the optimum number of processors.

- Two *collision detection algorithms* have been proposed so far, as listed on "Appendix A Collision detection algorithms". More might be proposed in the future. When measuring scheduling performance, both algorithms have to be tested and compared. Since every task will eventually rollback or commit, the number of rollbacks and commits will have to be an output parameter of the scheduler. This gives also rise to four other output parameters indicating the use of a processor: *idle time* are those cycles spent when no task is running on a processor; *wasted time* are those cycles spent on running a task that is rolled back (later), including the execution of the rollback itself; *executed time* are those cycles spent on running a task that is committed (later), including the execution of the commit itself. In Hjalmarson's algorithm, a task might even be put on wait. The cycles spent on waiting are measured by the *waiting time*.

- As explained in section 2.3, "Overview of the Multi-X prototype", performing *commits and rollbacks* might be implemented in different ways. In this thesis project, this parameter will be static and equal to the Multi-X prototype (i.e. the original method).

- *Cache behaviour* and scheduling are expected to relate in two ways. First of all, scheduling itself will influence cache performance. E.g., if a specific function is often executed and consistently scheduled at the same processor, the function's code and data will remain in the cache memory and cache-hit ratio will increase. Besides this, cache dimensioning like size, number of levels and coherence protocol will influence scheduling algorithms differently. Parallel to this thesis project, another thesis project will investigate cache behaviour [53]. A cache simulator will be implemented, which might be combined to scheduling performance measurements.

- The *traffic behaviour* will also influence scheduling performance. E.g., short GSM calls will give other traffic characteristics than long ISDN connections. Also, traffic intensity changes in time, e.g. peaks during daytime and low intensity at night-time. Different standards exist for e.g. mobile communications. All these properties result in different traffic characteristics, which will probably need different ways of scheduling. How to define the traffic behaviour parameter is discussed more detailed in section 4.2, "Finding the right traffic data".

- The *granularity of the marker field* has been explained and defined in section 2.3. A more fine-grained marker field will result in less data collisions but more overhead. A more coarse-grained marker field will result in less overhead but more data collisions. In this thesis project, this parameter will be static and equal to the Multi-X prototype.

- *Task granularity* is defined in "Figure 1 Relationship between transactions, tasks, functions and messages". But we might also define a task as a function or an entire transaction. We will not change the definition of a task or make it variable. The current definition resembles reality best and will be easiest to implement.

3.4 Different ways of scheduling for Multi-X

After having examined the relevant parameters, we can conclude that we can minimise execution time by: a) minimising processor idle cycles; b) minimising the number of rollbacks, and c) minimising cache-miss ratio. In this section, a number of scheduling algorithms will be proposed aiming to minimise these costs in different ways.

When comparing different scheduling algorithms, it is interesting to know the upper and lower bound of scheduling performance, given the parameters as explained in the previous section. A lower bound will be difficult to define, but every scheduling algorithm resulting in a speedup equal to or smaller than 1.0 can be discarded. In that case, we have not gained any performance compared to an ordinary sequential execution on a single-processor system. The upper bound is equal to an *optimal scheduling*. As explained in section 3.2.2, "Instruction-level parallelism", optimal performance can be obtained by calculating the critical path of an execution trace. In order to do this, we need to know all instructions to execute in advance. How this is done, is explained in the following chapter.

The literature survey summarised in section 3.2, "Learning from other projects" merely gave some ideas, but did not result in any algorithms specifically suitable for Multi-X. Consequently, we have to go back to the basic definition of scheduling. In the first section of this chapter, we stated that scheduling decisions are made on (dynamic) properties of the objects to be executed or on (dynamic) properties of the environment in which the objects are executing. Based on this, we define three different ways of scheduling: first-come-first-serve scheduling, function-based scheduling and source-based scheduling.

1. *First-come-first-serve scheduling* means simply to send the next task to an arbitrary idle processor. This method is used at the moment in the Multi-X prototype. Scheduling decisions are neither based on any property of the tasks nor on any property of the environment. An advantage of first-come-first-serve scheduling is that it is easy to implement with little overhead. It will be easy to maximise processor load but no actions are taken to avoid rollbacks. Cache-hit ratio is not taken into account either.

2. From several sources in speculative execution literature, it becomes clear that avoiding data collisions is often better than curing data collisions [7, 36, 41]. *Function-based scheduling* is an attempt to avoid unnecessary rollbacks by examining what functions a task will execute. If two tasks use the same function, it is likely that they share a variable. This, in its turn, will cause a data collision and at least one rollback. It might be better to postpone one of the tasks. In fact, function-based scheduling focuses on the properties of the objects to be scheduled, but ignores properties of the environment. Like trace scheduling in section "Instruction-level parallelism", we will have to be able to look ahead and predict the future execution trace. At task start only the first function to be executed is known from the initiating message. This means we will have to extend the Multi-X prototype with control speculation (e.g. branch predictors), since currently only data speculation is supported. The prediction mechanism will have to be implemented in software, since we assume no hardware support for task-level parallelism. This will increase scheduling overhead. Maximising processor load will be more difficult for function-based scheduling than for first-come-first-serve scheduling. However, the number of rollbacks will decrease and the assignment of tasks to processors might be done carefully in order to increase cache-hit ratio. The function-based scheduling as defined here is a division of function in *time*. Another solution is to divide functions in *space*; i.e. by assigning every function to one specific processor. Space-division is hard in our case, since we defined the task granularity as a number of functions.

3. With *source-based scheduling*, we focus on the properties of the environment but ignore the properties of the objects to be scheduled. The idea is to base scheduling decisions on the origin of the external messages

(see "Figure 1 Relationship between transactions, tasks, functions and messages"). All external messages originate from some remote processor. Every remote processor serves a number of devices, and every device serves a number of subscribers. Consequently, source-based scheduling is an attempt to map individual subscribers to individual processors. Since every subscriber has its own data, cache-hit ratio will improve. Since different subscribers will use different services in time, i.e. functions, source-based scheduling is also expected to decrease the number of rollbacks. Balancing and maximising processor load is assumed not to raise any difficulties. If it does, higher-level traffic management might solve this by dividing load more evenly from the different remote processors.

The three different ways of scheduling proposed here are by no means exhaustive; they just indicate three basic solutions. In fact, these are three extremes that give a broad understanding of scheduling behaviour in Multi-X. The best solution for Multi-X will probably be some combination of the scheduling algorithms mentioned here.

Even if the best scheduling algorithm can be defined for a particular set of parameters (see Figure 6), this scheduling algorithm will probably not be best if some parameter changes. This leads to the notion of an *adaptive scheduling algorithm* [66]. We define a scheduling algorithm to be adaptive if '*the rule base for making scheduling decisions is not fixed, but changes with the dynamic behaviour of the environment*'. The environment does not only include the set of parameters as in Figure 6, but also output parameters of the current scheduling (e.g. number of rollbacks or waiting time). Note that this definition differs from the definition of dynamic scheduling in Casavant's classification (Figure 3). A dynamic scheduling algorithm might very well have a fixed rule base. One way to implement an adaptive scheduling algorithm is a two-level approach. A number of non-adaptive scheduling algorithms are defined, all optimised for a specific environment. At a higher level, another algorithm classifies the environment (at run-time) and selects the best suiting non-adaptive algorithm. The classification algorithm might be implemented using techniques from the field of computational intelligence, e.g. fuzzy logic as in [11] or neural networks. One step further is to let the high-level algorithm not only classify the environment, but also produce a scheduling algorithm. This might be solved using neural networks [46, 60] or even genetic algorithms [74]. For now, an adaptive scheduling algorithm is remote future. This thesis will focus on the non-adaptive approaches mentioned above. Based on that knowledge, an adaptive algorithm might be defined in a later stage. The next chapter will summarise the different ways of scheduling and will describe how to investigate these.

4 Measuring scheduling performance

The previous chapter defined four ways of scheduling to investigate: three practical algorithms (first-come-first-serve scheduling, function-based scheduling and source-based scheduling) and one theoretical algorithm (optimal scheduling). The latter will give us an upper bound on scheduling performance. As output parameters for the scheduling algorithm we defined the number of commits, the number of rollbacks and for every processor: executed time, wasted time, waiting time and idle time. The cache-hit ratio will indirectly be an output parameter if the scheduling algorithm can, in some way, be linked to the cache simulator. Performance, expressed in speedup, can be deduced from these output parameters.

In the first section of this chapter, three ways of measuring performance parameters are compared. One way, implementing a simulator, is chosen for measuring performance in this thesis project. The succeeding chapter defines the data to be used as input for the simulator and describes how this data is obtained. Some statistics on the input data are presented. Section 4.3 describes the model of the prototype used as a basis for the simulator. Most issues have already been highlighted in previous chapters. Here, we merely present some important implementation details. Performance of optimum scheduling, as a theoretical approach, cannot be measured using the simulator. The last section of this chapter explains how optimal scheduling will be calculated.

4.1 How to measure performance parameters

Now we know what to investigate, we can decide how to investigate scheduling performance. Basically, we have three possibilities: implementing the scheduling algorithms in the Multi-X prototype, defining a mathematical or statistical model or simulating the Multi-X prototype.

1. *Implementing* the scheduling algorithms in the Multi-X prototype is probably easiest for the three practical algorithms. It will give quick results and will take into account the entire environment, even those parameters that are currently unknown or underestimated. Unfortunately, there are a number of disadvantages. The main disadvantage is that this approach excludes optimal scheduling entirely, since we need to be able to foresee a trace of execution. For the other algorithms, it gives us little freedom to test new ideas, e.g. the control speculation needed for function-based scheduling. Another disadvantage is the dependency that would arise between this thesis project and the development of the Multi-X prototype. At the beginning of this thesis project, the initial Multi-X prototype implementation was not finished yet and was expected not to be stable for the next coming months. Furthermore, measurements are difficult to perform on the prototype's implementation. Since the prototype is processing transactions at a high speed, measurements influence performance substantially. At the beginning of this thesis project, the project group was still investigating how to do correct measurements and to what extent measurements influence performance. Even if measurements can be done correctly, they will still include overhead, e.g. caused by the data collision algorithm or by the implementation of rollback and commit. Eventually, these overheads have to be taken into account. For now, the prototype's implementation can be optimised in several ways, which might lead to a lower overhead. In this thesis project, we want to focus on the performance of scheduling primarily, irrespective of the way particular parts of the system will be implemented. For these reasons, the approach of implementing scheduling algorithms in the prototype is not preferred.

2. By defining a *mathematical model*, all disadvantages mentioned above are avoided. But two other problems arise. The first problem is traffic modelling. We have to model not only the prototype but also the environment as depicted in "Figure 6 Relation of scheduling to other parameters in the Multi-X prototype". Especially traffic behaviour will be hard to model. In fact, Multi-X is a test case to find out if it is possible to run traffic on a multiprocessor system. Statistical information is only available for single-processor systems; e.g. the number of variables accessed per function, the number of functions per task, etc. It is yet unknown what traffic characteristics are important for Multi-X or for a multiprocessor system in general. The second problem is the behaviour of the functions to be executed. Here, it is also totally unknown how these functions will behave in a multiprocessor system, e.g. what variables will cause data collisions and what not?; what functions are parallelisable and what function have to be run in a serial fashion? The Multi-X project has just started to answer these questions, e.g. [12, 73]. For these reasons, a mathematical model will lack important details. Even if we can find out all relevant details, the model will be very complex. For these reasons, the approach of a mathematical model is not preferred either.

3. The best solution would be to combine to advantages of the two approaches mentioned above, while trying to avoid the disadvantages. We need to model the prototype to get clear results independent of some implementation. At the same time, we need real traffic in order to be able to avoid modelling unknown details on traffic behaviour. A *simulator* will be able to combine these requirements and will therefore be developed in this thesis project. Although developing a simulator makes it easier to take all relevant details into account (compared to a mathematical model), we still might miss some. This is the main disadvantage of this approach.

The properties of the Multi-X prototype as explained in section 2.3, "Overview of the Multi-X prototype", are the basis for the simulator model. We also include the static parameters of "Figure 6 Relation of scheduling to other parameters in the Multi-X prototype" (commit/rollback implementation, marker field granularity, task granularity) in this model. Cache behaviour can be modelled by linking the scheduling simulator to the cache simulator. The number of processors and the collision detection algorithm will be variable input parameters for the simulator. The next section describes what traffic data will be used and how it is obtained. The subsequent section describes the details of the simulator model.

4.2 Finding the right traffic data

One of the most time-consuming parts of this thesis project was to find the right input data for the scheduling simulator. In this section, we will start by defining more precisely what data is needed. After having defined the required data, section 4.2.2 describes where this data was found and how it is pre-processed to the desired format. A great deal of time has been spent on verifying the acquired data. This process is described in section 4.2.3. The last section presents some important statistics on the acquired data.

4.2.1 What traffic data is needed

In order to model traffic behaviour correctly, we will have to use a trace of live data. To be able to simulate all proposed scheduling algorithms, this trace should include the following data: a list of messages; for every message (i.e. every task), a list of functions; for every function, a list of instructions; for every instruction, the memory references of code segment and data segment. At the highest level, the trace should be a list of messages. For every message, we need to know if it was generated externally or internally. If it was generated externally, we need to know the initiating remote processor. If it was generated internally, we need to know the initiating task. This way, we can reconstruct the sequence of tasks for every transaction. We also know the source of every task in a transaction. To be able to simulate data collisions, we need to know the references to the data segment (i.e. the data areas guarded by a marker field) and the function, since every variable is global only within its own function.

In order to reconstruct a live traffic flow, we also need timing information. We need to know at what time externally generated messages are inserted in the FIFO queue and we need to know the execution time of every instruction. Both issues are problematic. Even if we know the execution time of every instruction, it will be an execution time obtained from *a specific* environment; i.e. a specific cache-hit ratio, a specific memory access time, etc. When simulating different ways of scheduling, we have to abstract from that specific environment and make an estimation of the execution time for every instruction in the environment of our simulator. Related problems arise with the arrival time of externally generated messages. Even if we know these times, they will relate to some architecture, i.e. a single-processor system. When simulating architectures with a different number of processors, that architecture will likely process a different number of transactions per second. When using the same timing information in a simulation, this will lead to either an overload or an underload of the simulating architecture. In some way, we have to abstract from this timing information too.

The solution to these problems was found by reconstructing timing information from the instruction name and memory references. This information is needed anyway in order to simulate data collisions. If the instruction name is known, we can obtain minimum, maximum and average instruction execution time from statistical information [44]. This statistical information is based on application measurements done in another project. Execution time is expressed in cycles, not in seconds. From this information and the memory references, we can compute an estimation of the instruction's execution time by feeding every reference to the cache simulator and count the number of cache hits (see Equation 2, conform [53]). If we know the execution time of every instruction, we also know the total execution time of every task and every transaction. The problem of external message insertion timing was solved by simply inserting the next externally generated message as soon as the total number of messages in the FIFO queue gets below a certain limit. The average number of messages in the FIFO queue for current APZ CP systems is 2-4. We will set the limit to 10, for two reasons: 1) the current APZ

CP system is a single-processor system, a multiprocessor system will give a higher average of processed tasks per second; 2) a high limit means a heavy-loaded system. In fact, we abstract from traffic peaks and bursts by modelling a constant heavy load. We assume that if our simulating architecture performs well in this situation, it will also perform well with an ordinary traffic load or with bursty traffic.

$$t_{execution} = t_{execution,\min} + \frac{\#misses}{\#references} \cdot (t_{execution,\max} - t_{execution,\min})$$

Equation 2 Calculating instruction execution time

4.2.2 Where to find traffic data

Now we know what traffic data is needed, where can we find it? The initial attempt was to record a trace from the running Multi-X prototype. Unfortunately, this turned out to be hard to implement and time consuming. Other problems with the Multi-X prototype were described in the previous section. Eventually, the Tor project mentioned in section 3.2.1 proved to have most data available. Here, live recordings were done from a number of real (single-processor) APZ CP systems [55, 56, 57]. Three different types of traffic are available: CMS, a US standard for wireless communication; GSM, the European standard for mobile communication; and Transgate, a standard for traffic between telecommunication exchanges. For every type of traffic, three to five recordings were available, all recorded in the same format. Every recording contains approximately 200ms of live traffic. Figure 7 gives a simplified overview of the pre-processing performed on the recordings. The pre-processing results in four data files that can be used as input to the simulator. The 'live recording' file is a list of instructions executed; for every instruction the contents of the address bus, the contents of the data bus, function number and priority are recorded. Priority indicates if this instruction was ordinary traffic or a background process. Background processes are only started if the FIFO message queue is empty and are interrupted if an external message arrives. These instructions are filtered away, since we will simulate a heavily loaded system running real traffic. Simulating background processes will also lead to a much more complex simulator, e.g. different priority levels would demand a more complex data collision detection algorithm. The 'live recording' and 'memory dump' file are fed to an instruction disassembler. The output of the disassembler is combined with the results of instruction timing calculations and results in the 'trace info' file. This file contains all information listed at the beginning of this section. The disassembler was taken from the Tor project and modified and extended to the needs of this thesis investigation. Four extraction routines produce the simulator's input files. Data in these files is internally connected by pointers; i.e. every message is a part of a task sequence (a transaction), every task is a sequence of functions, every function is a sequence of instructions.

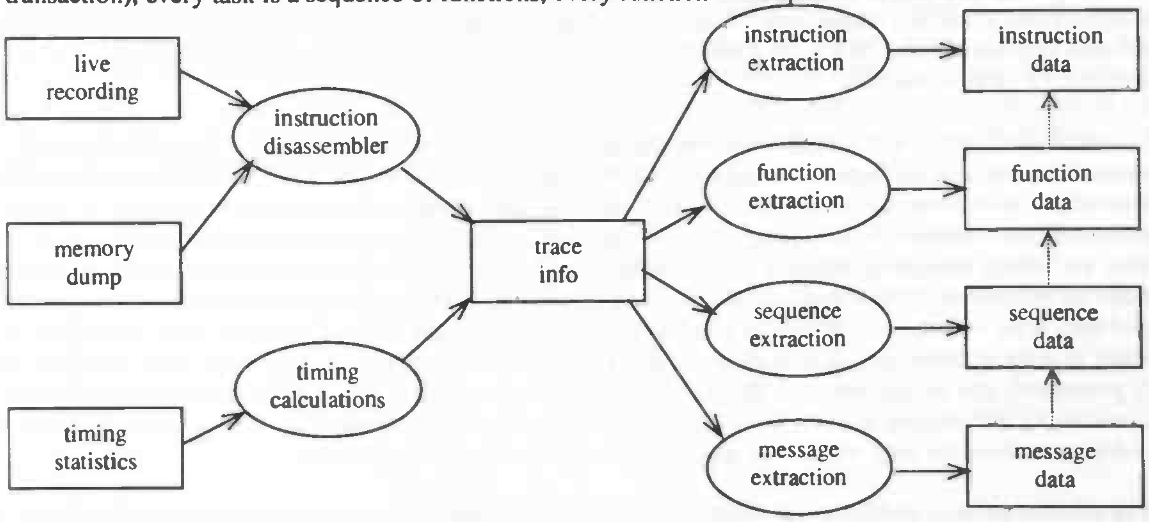

Figure 7 Data flow in the pre-processing of simulator input data

4.2.3 Verifying traffic data

A number of problems arose when pre-processing the recordings. Intelligent filtering solved most problems, but left some errors in the resulting data files. Only those problems are mentioned here and an upper bound of the resulting errors is defined:

1. The 'live recording' file could not be recorded perfectly, the recording equipment left 1-2% errors. For these errors, instruction address and/or bus data contents are wrong. Most of the addresses could be restored by

predicting the address of every instruction from addresses of preceding instructions. The first instructions of a function and instructions following a branch could not be restored this way. It turned out that most errors occurred on the last instruction(s) of a task. These errors, in its turn, cause the pre-processor's extraction functions to miss task boundaries. Eventually, the overall error on task boundaries was proved not to exceed a 3% limit. This figure is however 'worst-case'; in reality the error is expected not to exceed 1%.

2. In order to obtain the memory references of an instruction, we not only need the instruction name but also its parameters. Some parameters are available on the recorded data bus, while other parameters reside in the processor's internal registers, which have not been recorded. Mostly, the contents of these registers could be traced by finding the last instruction (re-) setting the particular register. In some cases, the contents of a register originates from a message, whose data is not recorded. If message data is not copied to a register, this data could not be traced. Those memory references were simply discarded. It has been proved that less than 0.4% of all memory references is discarded for this reason.

3. If task boundaries, function boundaries and memory references can be extracted, then all necessary information within every task is known. We also need to reconstruct the sequence of tasks in every transaction, since this information is not recorded. This was solved by tracing every message sent in the extracted tasks (messages are sent by special instructions). From this information and by simulating a FIFO queue, the sequence of tasks can be deduced. For reasons similar to and related to the two problems described above, not all send message instructions could be extracted correctly. These messages were simply discarded. Compared to the total number of send message instructions, not more than 2% of the send message instructions was discarded. This results in a number of transactions to be shorter than reality.

4. Like arguments of some instructions, information on remote processors needed for source-based scheduling is sent with a message. This information is stored in an internal register on task entry. In 25% of all externally generated messages, this information could not be recovered. See section 5.4, "Source-based scheduling", for a more detailed discussion on this problem.

Except for the remote processor information, the errors mentioned above are within tolerable boundaries. Lowering these error bounds means a great deal of extra work and has been renounced in favour of more simulations. The resulting data has been verified extensively, both at a low level (by comparing instruction data execution code with the original code) and at a high level (by comparing statistics on the traces with known statistics). All tests showed the resulting data to resemble reality to a good extent.

4.2.4 Statistics on used traffic data

Although the recordings were done in the same format and the pre-processing code is general to all recordings, pre-processing, analysing, verifying and simulating one recording turned out to be very time-consuming. In the remainder of this report, only one CMS recording is used. For the other recordings, verification samples were taken at random in the pre-processing stages of the investigation. All tests gave results similar to the one recording we use throughout this report. The CMS traffic type was chosen because the Multi-X prototype also runs this type of traffic. By focusing on CMS traffic, results can be compared more easily with the prototype.

percentage real traffic in recording	84%
percentage background processes (filtered away)	16%
total number of transactions	2,672
total number of tasks	14,045
total number of functions	28,051
total number of instructions	1,588,536
total number of cycles	8,079,225
total number of variables accessed	556,922
percentage single variables plus single arrays	42%
percentage arrayed structure variables	58%
percentage externally generated traffic messages	10%
percentage internally generated traffic messages	81%
percentage operating system messages	9%
average number of tasks per traffic transaction	9.22
average number of functions per task	2.00
average number of instructions per task	113.10
average number of variable references per task	39.65
average number of cycles per task	575.24
average number of cycles per function	288.02
average number of cycles per instruction	5.09
clock frequency of recorded system in MHz	40
total execution time in ms	201.98
average execution time per task in us	14.38

Table 2 Some statistics on the pre-processed data

Table 2 shows some statistics on the pre-processed recording used throughout the remainder of this report. All information is on the 84% of real traffic. The other 16% are background processes that have been filtered away. On analysis of the recorded data, it turns out that not all transactions start with an external message initiated from a remote processor. Approximately half of all transactions are initiated by the operation system. These transactions perform various administrative tasks. For source-based scheduling, a distinction is made between *real traffic transactions* and *operating system transactions* (see section 5.4, "Source-based scheduling", for a more detailed discussion on this topic). The timing information mentioned in Table 2 is based on the average execution time of every instruction. This information was extracted from timing statistics, as depicted in Figure 7. Figure 8, Figure 9 and Figure 10 show some extra information on the pre-processed data. From this information, it becomes clear that most transactions are small.

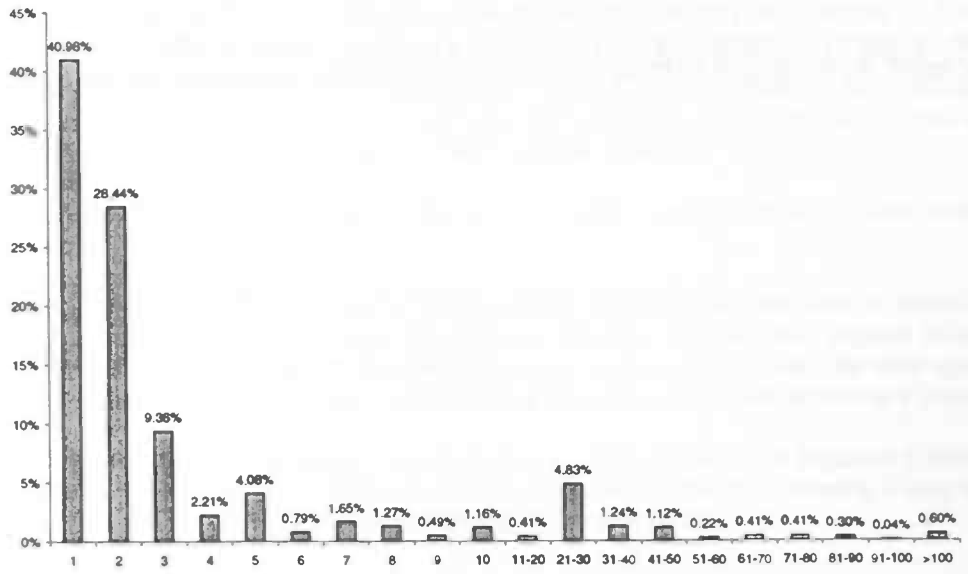

Figure 8 Distribution transaction size (in number of tasks)

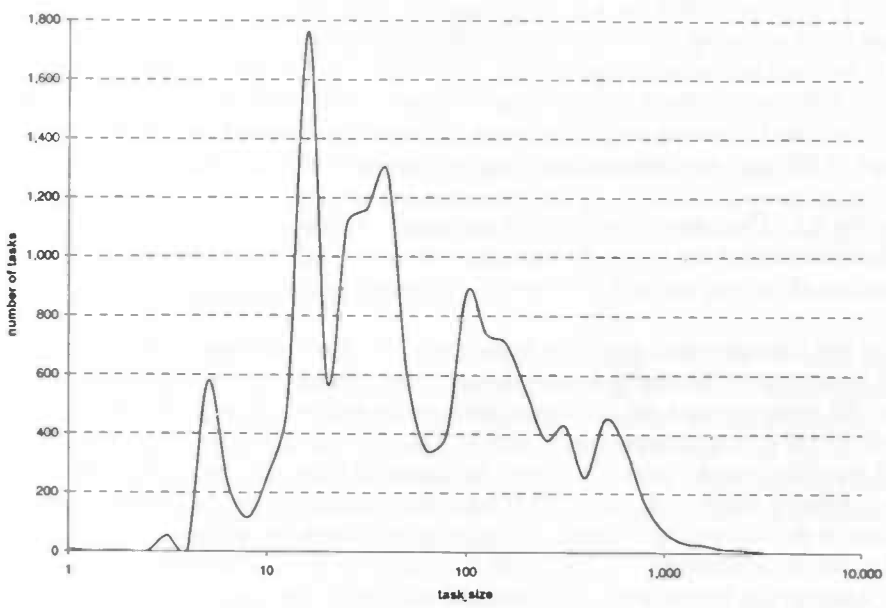

Figure 9 Distribution of task size (in number of instructions)

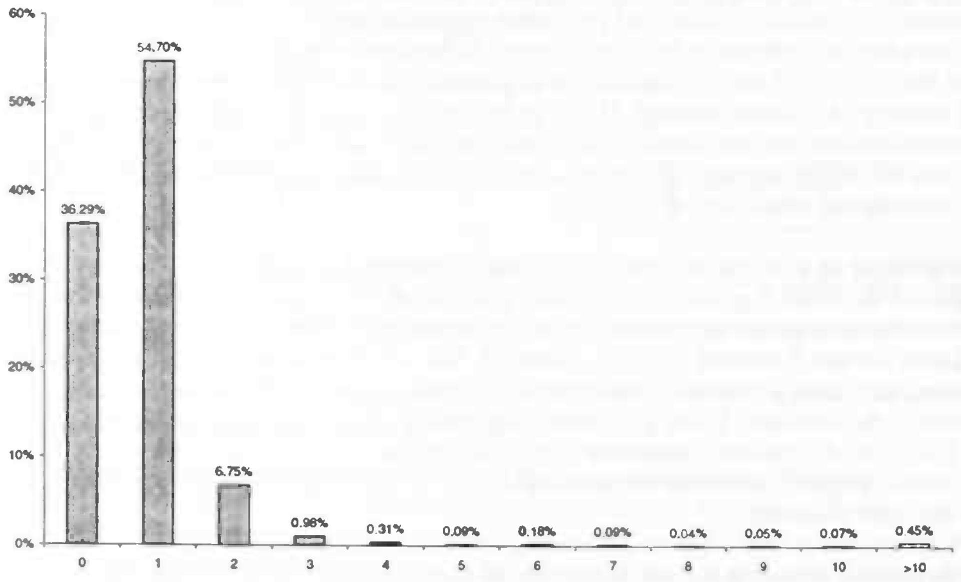

Figure 10 Distribution followers per task (in number of tasks)

The entire set of input data does not constitute more than 200ms of live traffic. In the next chapter, it is checked if this is enough to get stable results.

4.3 Modelling the prototype

The basic properties of the model used for the implementation of the simulator have been described throughout this report, see: section 2.3, "Overview of the Multi-X prototype"; section 3.3, "Position of scheduling in the Multi-X prototype"; section 4.1, "How to measure performance parameters"; section 4.2.1, "What traffic data is needed"; and "Appendix A Collision detection algorithms". A number of important details on the implementation of the model:

1. Instructions are not really executed by the simulator. For every instruction, the simulator simply waits for a number of cycles. All timing is measured in cycles, since this information is available in the simulator's instruction data input file. The core of the simulator is an event queue, where every instruction executed on some processor results in an event.

2. The simulator does not model any overhead, e.g. administrative processing on starting, ending, committing or rolling back a task or on accessing data areas. Some of the disadvantages of overhead are mentioned in section 4.1, "How to measure performance parameters". Two main reasons not to model overhead are: 1) It is hard to quantify the different overheads. The current implementation might be optimised in several ways. E.g., in the future, a dedicated processor might hide some of the administrative tasks. 2) We would like to measure the potential of different scheduling algorithms, independent of some specific implementation.

3. As explained in section 2.3, "Overview of the Multi-X prototype", rollback and commit can be implemented in several ways. Since instructions are not really executed and since we do not model any overhead, the simulator is independent of the way rollback and commit is implemented.

4. It was decided not to link the scheduling simulator to the cache simulator. The main reasons for this decision are the time needed to implement this and to avoid too many dependencies on the thesis project resulting in the cache simulator. All measurements are done using the average number of cycles for every instruction, as obtained from the statistical timing information (Figure 7). This average is obtained from a specific environment with a specific cache-hit ratio. Of course, the cache-hit ratio will change for different ways of scheduling and for a different number of processors. This topic is discussed for the simulation of various scheduling algorithms in the succeeding chapter. A number of problems that will arise when the scheduling simulator is linked to the cache simulator: 1) The cache simulator works on an instruction basis and the cache simulator on a memory reference basis. This means that, if an instruction has several memory references, the interleaving with instruction executed on other processors is not as in reality. 2) Cache memory is large nowadays. It has not been investigated if the number of instructions available in the recordings is enough to initialise caches and give stable simulation results. 3) The simulator's input data files include the addresses referred in the code segment. References to the data segment can be added by extending the functionality of the extractors in pre-processing. The AXE CP system, however, has even a third type of memory: the pointer segment. This segment includes pointers to various administrative tables (e.g. signal specifications) and was added to ease system administration. Since a large part of the instructions uses the pointer segment, this information should be added when simulating cache behaviour. It has not been investigated what effort this will take.

5. All rollbacks are done on a certain task causing the data collision and *all* older tasks. This resembles the implementation of the Multi-X prototype. In Tikekar's algorithm, this might be optimised as listed on "Appendix A Collision detection algorithms". In the implementation of the prototype, rolled back tasks are not restarted until the task it collided with has completed. This avoids unnecessary clustered rollbacks. In the simulator, rolled back tasks are restarted immediately. The main reason for this is a simplified implementation of the simulator. It also gives more clear results of the potential of the scheduling algorithms. The order of rollbacks is important in the real implementation. Since we do not really execute instructions, rollback order is irrelevant for the simulator. As in the Multi-X prototype, a rolled back task is restarted on the same processor.

6. A processor is released as soon as the last instruction has been executed. An ended task does not necessarily commit immediately, since we model in-order commit. Since rollback is done from a task up to the youngest task and rolled back tasks are restarted at the same processor, this might cause several tasks to compete for the same processor. In this case, the oldest task takes priority.

7. In Hjalmarson's algorithm, tasks might be put to wait. Processors are not released during wait, as in the Multi-X prototype.

8. In Tikekar's algorithm, the marker field's size is limited to the allowed level of speculation. In the current implementation of the prototype, the number of uncommitted tasks never exceeds the level of speculation. This is achieved by simply stalling task extraction from the FIFO queue. The same approach is used in the scheduling simulator. Another implementation is to put tasks on wait as soon as a marker field overflows.

Based on the simplifications in the simulator, *speedup* can be defined as in Equation 3. Note that this definition is based on in-order commit. After having processed the entire input data, speedup is simply the quotient of the number of cycles in the input data and the global clock cycle count.

$$speedup(n) = \frac{\text{sum of cycles in input data task 1..n}}{\text{global clock cycle count at commit of task n}}$$

Equation 3 Definition of speedup

The implementation of the simulator has not been proved correct. A large amount of pre- and post-conditions were defined and inserted in the code in order to enforce a correct execution. Only small sub-sets of input data have been tested extensively. For the entire set of input data, tests were mainly done by analysing results.

4.4 Calculating optimal scheduling

The simulator described in the previous section can be used to simulate the three practical scheduling algorithms: first-come-first-serve scheduling, function-based scheduling and source-based scheduling. But is it able to simulate optimal scheduling? To answer this, we have to define optimal scheduling more clearly. We define optimal scheduling to be '*the way of scheduling resulting in the maximum speedup for a given set of input data when assuming infinite resources and assuming all transactions to execute are known in advance*'. Infinite resources includes an infinite amount of processors and an infinite amount of uncommitted task (the latter is finite in reality because of limitations in memory size).

How can optimal scheduling be obtained? Intuitively, rollbacks cause sub-optimal scheduling. But waiting in Hjalmarson's algorithm sometimes leads to a better speedup. E.g. the situation Figure 11. With Hjalmarson's algorithms task 2 waits for task 1 to commit before reading the shared variable. If task 2 would not wait, it had to be started later causing task 3 to commit later.

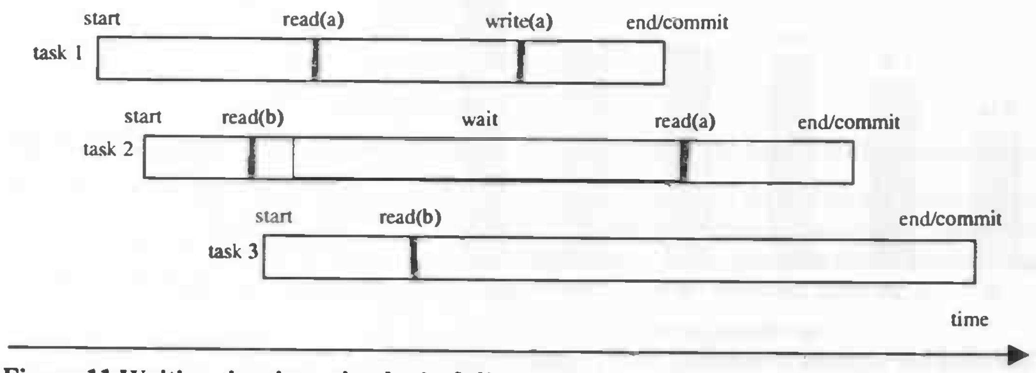

Figure 11 Waiting time in optimal scheduling

Clearly, optimal scheduling cannot be simulated but has to be calculated. For both data collision detection algorithms, an algorithm has been developed to calculate optimal scheduling. These algorithms are listed on "Appendix B Algorithms and proof for optimal scheduling" and a proof of correctness is given. In general, proving correctness for optimal scheduling with limited resources is very hard. For this reason, the algorithms have not been extended for a limited number of processors. For Tikekar's data collision algorithm, performance will be influenced to a large extent by the allowed level of speculation. The straightforward implementation of limiting the level of speculation in the Multi-X prototype made it easy to extend the optimal scheduling algorithm with the level of speculation as a parameter. In the succeeding chapter results of optimal scheduling and of scheduling using the simulator are shown and analysed.

5 Simulation results

The previous chapter defined how to measure performance for the proposed scheduling solutions. A simulator was designed to measure the three practical algorithms. Two algorithms were defined to calculate optimal scheduling. This chapter presents the results from these calculations and measurements. As explained in the previous section, all measurements and calculations are done on a static set of input data with average execution times for every instruction.

Results of optimal scheduling are described in the first section of this chapter. The three succeeding section describe first-come-first-serve, function-based and source-based scheduling respectively. Conclusions are presented in chapter 6. All source data of the figures presented in this chapter are listed on "Appendix C Data on simulation results".

5.1 Optimal scheduling

Optimal scheduling for Hjalmarson's data collision detection algorithm gives a speedup of 2.47. The results for Tikekar's algorithm are shown in Figure 12. Tikekar's algorithm was tested for a speculation depth of 12, 28, 44, 60, 76 and for an unlimited speculation depth. This resembles a marker field size of 64, 96, 128, 160, 192 and an unlimited number of bits respectively.

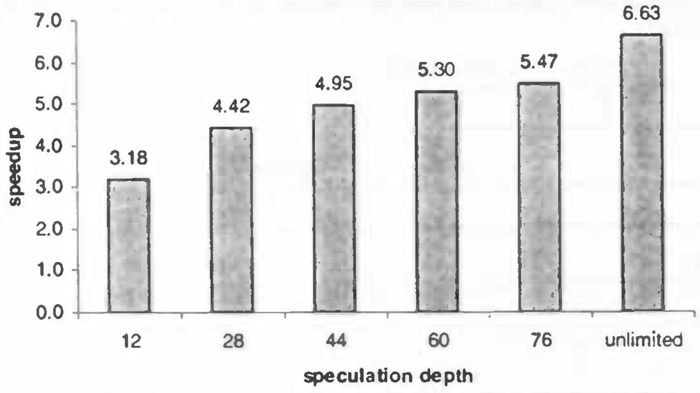

Figure 12 Speedup for optimal scheduling with Tikekar's algorithm

The result of Hjalmarson's algorithm is somewhat disappointing. For a speedup of 2.47 we need unlimited resources and have not included any overhead. The main reason for this limited speedup is the stall time inherent to this implementation of collision detection. In fact, even read after read dependencies (see "Figure 4 Classes of data dependencies") lead to stall times. Tikekar's data collision detection algorithm was proposed at a later stage to avoid unnecessary dependencies and rollbacks. A drawback of this algorithm is an increase in required memory. The implementation of the prototype uses a speculation depth of 12, resulting in a memory requirement four times as high as for Hjalmarson's algorithm. Figure 12 shows the potential of Tikekar's algorithm to be much higher, even for a speculation depth of 12 tasks. Speedup increases substantially when raising speculation depth to 28.

We still have to check if the results of the calculations are stable. For this, we plot speedup as a function of (commit) time; Figure 40 and Figure 41 on "Appendix C Data on simulation results". In both graphs, speedup stabilises after having processed half of the input data. This indicates the input data to be large enough. Figure 41 shows the results of optimal scheduling with a speculation depth of 12. For a higher speculation depth, speedup stabilises even sooner. Generally, speedup is higher at the beginning of processing, since tasks have fewer older tasks to be dependent on at that stage.

The speedup calculated for optimal scheduling will never be reached in reality. In the succeeding three sections, we will simulate more practical approaches, which can be compared to the upper bound of optimal scheduling.

5.2 First-come-first-serve scheduling

With first-come-first-serve scheduling we simply send the next task to an arbitrary idle processor. Note that this way of scheduling is *deadlock free*; at least the oldest task is always executable since it does not have any dependencies. The oldest task will therefore never be rolled back either. In Figure 13 the results are shown for both data collision detection algorithm. For Tikekar's algorithm, this is for a speculation depth of 12. Up to six processors have been measured since not more will be used in practice anyway. Optimal scheduling is added as an asymptote, since this is only calculated for an unlimited number of processors.

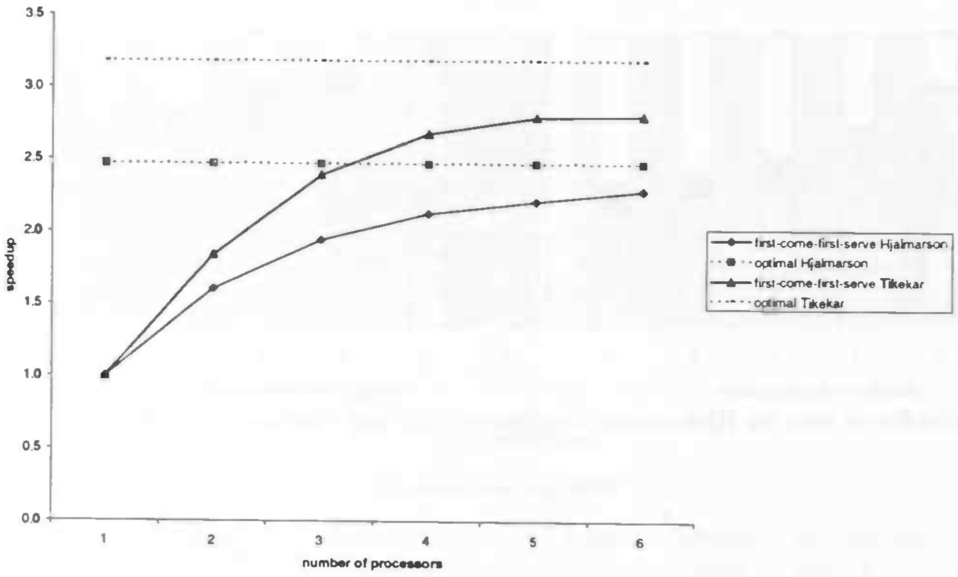

Figure 13 Comparison first-come-first-serve scheduling for both data collision detection algorithms

As expected, Tikekar's algorithm performs better than Hjalmarson's algorithm. It is interesting to notice, though, that Hjalmarson's algorithm does exploit its potential better than Tikekar's algorithm. However, for Tikekar's algorithm rollback is done for all tasks from a certain task up to the youngest task. This can be optimised later, see also the definition of the algorithm on "Appendix A Collision detection algorithms".

Before going on, we first have to check if the length of the input data is sufficient for the simulator to reach stable results. As for optimal scheduling, this is checked by plotting speedup in time. Results are shown in Figure 42 and Figure 43 on "Appendix C Data on simulation results". Only results for four processors, for both data collision detection algorithms are shown. For Tikekar's algorithm, only the results of a speculation depth of 12 are shown. Some other speculation depths and number of processors have been tested at random and gave similar results. Surprisingly, speedup is very low during the first 25% of simulation. This can be explained from Figure 44, which shows that the first 2,700 tasks are relatively small with one exception, the largest task in the input data. Very large tasks cause many depending younger tasks, resulting in a nearly sequential behaviour. Very small tasks cause large sequences of cascaded rollbacks, resulting in a decrease of performance. Speedup increases as soon as these 2,700 tasks are processed; for Hjalmarson's algorithm after 800,000 cycles. For all graphs, speedup stabilises after 75% of the input data has been processed. This proves the input data to be just large enough. Since it has been proved that the input data is large enough for both optimal scheduling and first-come-first-serve scheduling, this is not checked again for function-based scheduling and source-based scheduling.

Let's take a closer look at the results of the simulations. Figure 14 shows the distribution of time for a different number of processors. The definition of idle time, waited time, wasted time and executed time is given in 3.3, "Position of scheduling in the Multi-X prototype". Time is expressed as a percentage of the total number of cycles; e.g., the total sum of executed cycles on four processors is divided by the total sum of all cycles on these four processors. Waiting time increases exponentially for Hjalmarson's algorithm, while idle time remains virtually zero. For six processors, idle time is just 0.80%; this is only due to the fact that processors remain idle at the very last part of the simulation when there are no more tasks to be scheduled. Wasted time increases with the number of processors, but remains low, especially for a small number of processors (5.27% for four processors). Figure 15 shows more detailed information on rollbacks. *Rollback frequency* is defined as '*the quotient of the number of rolled back tasks and the total number of committed tasks*'. For both algorithms, rollback frequency increases exponentially. This is mainly because of repeating rollbacks. This includes all tasks that are rolled back once more after a previous rollback and before they have been re-started. Wasted time gives a better indication of time lost on rollbacks. For Tikekar's algorithm, wasted time does not increase exponentially. However, we do not include any overhead. If we would, these curves will deteriorate. Tikekar's algorithm does not put any tasks on wait, but a price is paid in wasted time; more than double the amount of Hjalmarson's algorithm. Idle time is surprisingly high in Tikekar's algorithm. This is due to the limits in speculation depth. For four processors, idle time is nearly 17%. Execution time is higher for Tikekar's algorithm, for every amount of processors. This results in a higher speedup, as plotted in Figure 13.

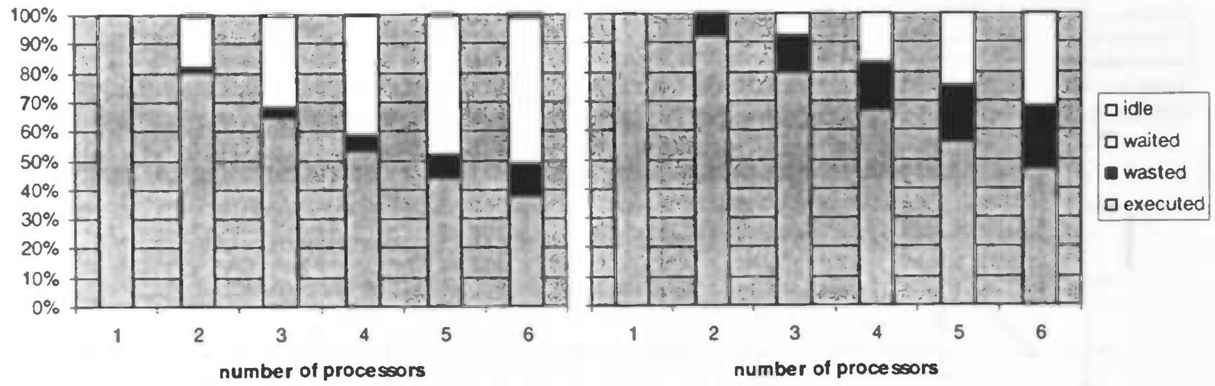

Figure 14 Distribution of time for Hjalmarson's algorithm (left) and Tikekar's algorithm (right)

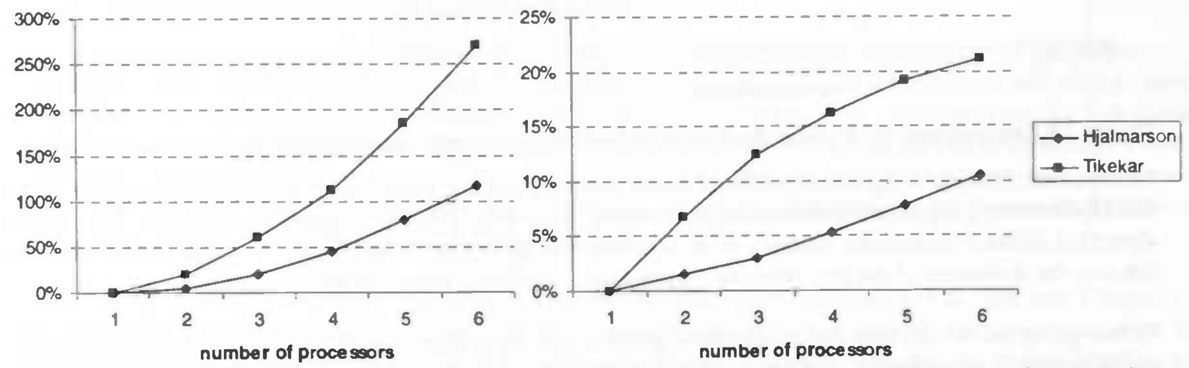

Figure 15 Rollback frequency (left) and wasted time (right) for both data collision detection algorithms

Both speedup graphs saturate at a certain level. If we had taken overhead into account, the graphs would have been curves with a top at the optimum number of processors. E.g., for Tikekar's algorithm, speedup is 2.79 for five processors and 2.80 for six processors. Wasted time is 19.15% and 20.09% respectively, but rollback frequency increases with nearly 100% from 184% to 270%. In reality, wasted time will be higher due to the increase in rollback frequency, resulting in a lower speedup for six processors.

What is the effect of a larger speculation depth for Tikekar's algorithm? Figure 16 shows speedup as a function of speculation depth. Surprisingly, speedup hardly increases with a higher speculation depth for two or three processors. Only with more than three processors, we see a substantial increase in speedup when increasing speculation depth from 12 to 28. Increasing speculation depth beyond 28 only pays off for a large number of processors.

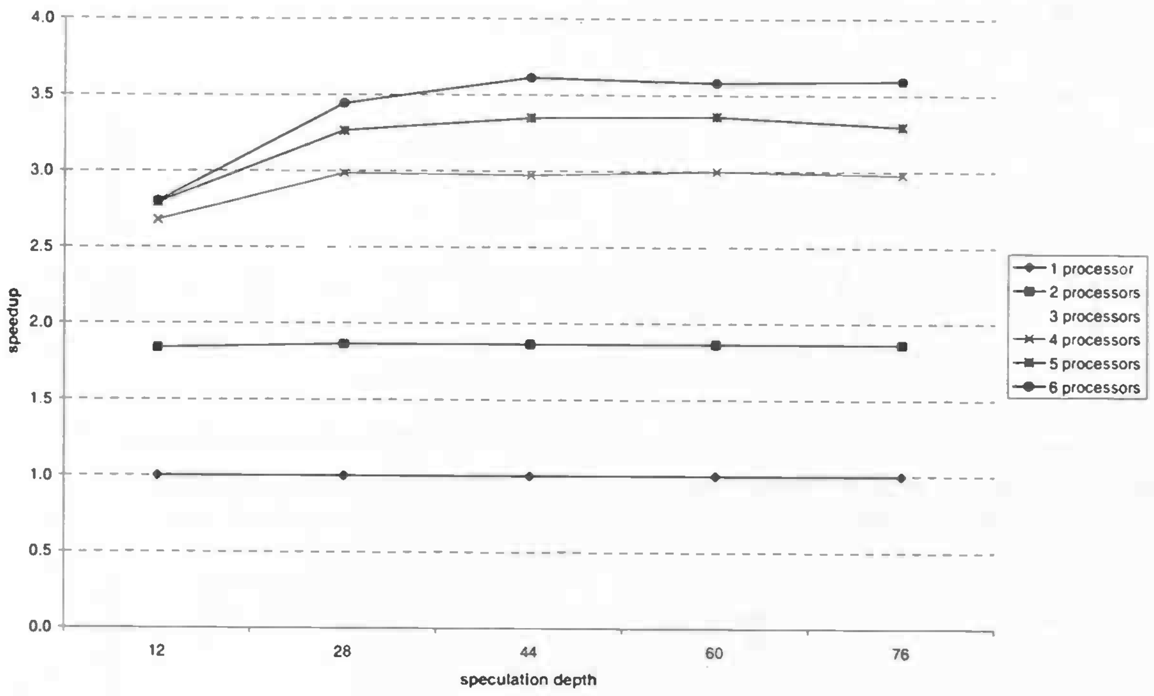

Figure 16 Speedup as a function of speculation depth

This behaviour can partly be explained from Figure 14. From this figure, it becomes clear that idle time is virtually zero for two processors, but increases rapidly when increasing the number of processors. Figure 17 plots all idle time information. With first-come-first-serve scheduling, processors are idle only if the total number of uncommitted tasks has reached the speculation depth limit. E.g., if speculation depth is 12, four processors are each running a task and eight tasks are ended but not yet committed, then a 13th task cannot be started on the fifth processor. This processor is idle until a task commits. Obviously, such situations rarely occur for a small number of processors. Increasing speculation depth does not increase performance since there are no cycles left to exploit the increased speculation depth. For a large number of processors, limited speculation depth does result in a considerable amount of idle cycles. Increasing speculation depth decreases the number of idle cycles, resulting in a higher speedup. At the same time, a price is paid by an increased number of rollbacks and an increased wasted time. This explains the relatively smaller differences in increase of speedup when increasing the number of processors. Increasing speculation depth too much for a high number of processors (5 or 6 processors, speculation depth 76 tasks) even leads to a decrease in speedup. This can be explained from the larger sequences of cascaded rollbacks.

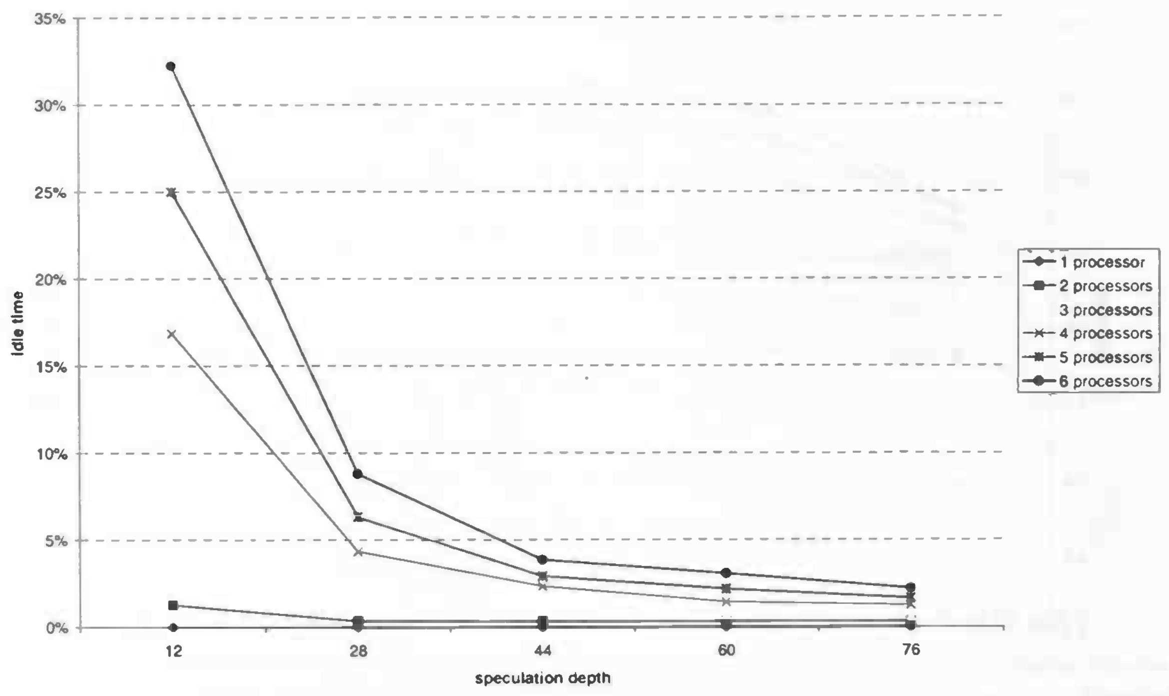

Figure 17 Idle time as a function of speculation depth

All measurements on first-come-first-serve scheduling were taken with an average number of cycles per instruction. With a different cache-hit ratio, this number of cycles per individual instruction will change (see "Equation 2 Calculating instruction execution time" in section 4.2.1). However, cache-hit ratio for first-come-first-serve scheduling is not expected to change very much compared to single-processor architecture. There are no reasons to assume cache-hit ratio will deteriorate. At the same time, we have not taken any precautions to try to increase cache-hit ratio. For this reason, first-come-first-serve scheduling is expected to give similar results, even if the scheduling simulator is linked to the cache simulator.

After having investigated optimal scheduling and first-come-first-serve scheduling, it is clear that Tikekar's algorithm has more performance potential than Hjalmarson's algorithm. The same conclusion can be draw from results of the prototype's implementation. Therefore, only Tikekar's algorithm is used for testing function-based scheduling and source-based scheduling.

5.3 Function-based scheduling

The goal of function-based scheduling is to avoid unnecessary rollbacks by examining what functions a task will execute. If two tasks share a function, it is likely that they share variables and cause a data collision. Let us assume we can foresee all functions that every task will execute. In fact, this assumption means the scheduler can predict functions to execute perfectly. Perfect prediction is not feasible, but the assumption is fair since more than 75% of all tasks only executes one function; see Figure 18. The first function to execute is always known from the initiating signal.

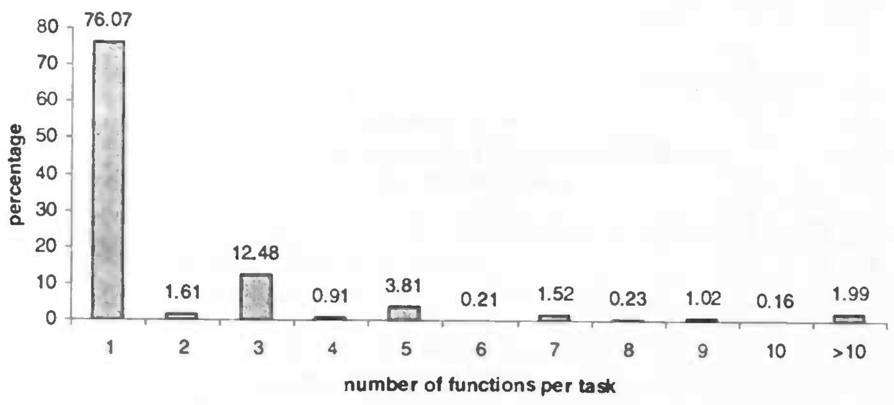

Figure 18 Distribution of number of functions per task

Less than 25% of all tasks executes more than one function and some functions might be executed several times. The number of *unique* functions executed per task support the assumption even better; see Figure 19.

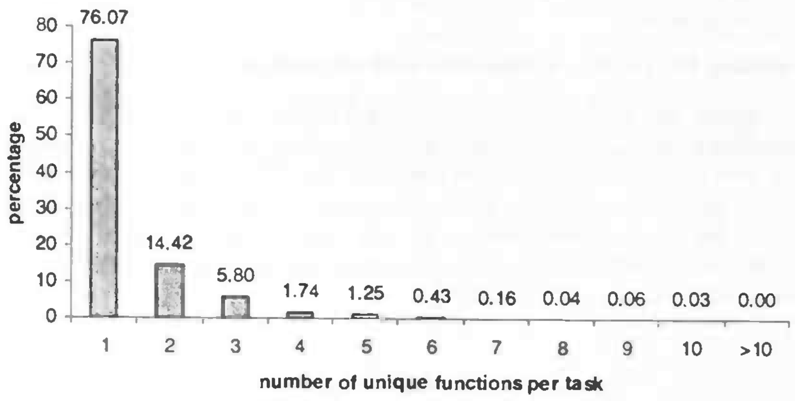

Figure 19 Distribution of number of unique functions per task

If all functions to execute are known in advance, a simple scheduling rule can be formulated: "*a task can start execution if it does not share a function with any older uncommitted task*". With perfect prediction, this way of scheduling will not cause a single rollback. Note that this way of scheduling is deadlock free, since the oldest task is always executable and will not be rolled back. Results of this simulation are shown in Figure 20, where function-based scheduling is compared to the other ways of scheduling. As mentioned, we only test Tikekar's algorithm. Only a speculation depth of 12 tasks was tested; this resembles the level of speculation used in the prototype's implementation.

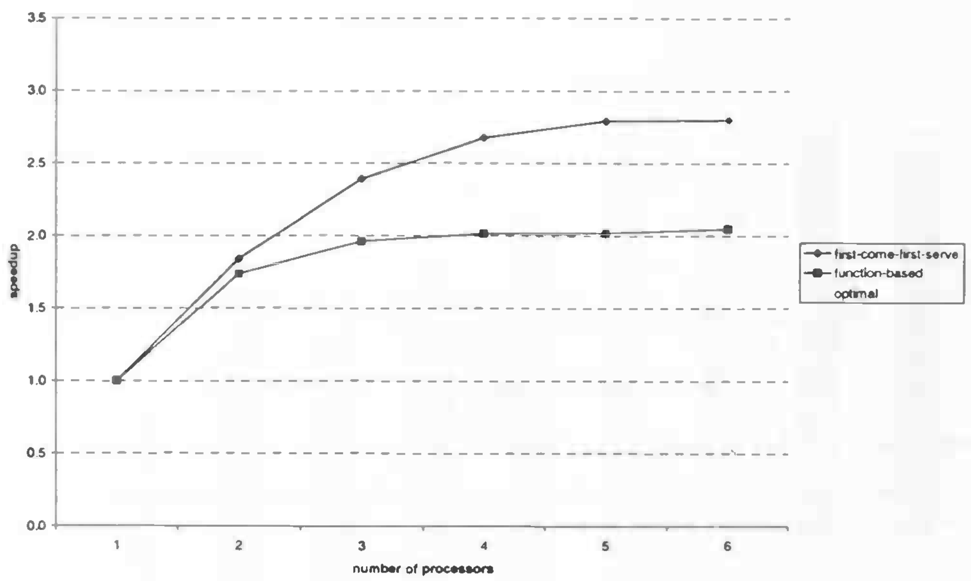

Figure 20 Comparison of scheduling for Tikekar's algorithm with a speculation depth of 12 tasks

As expected, function-based scheduling with perfect function prediction does not lead to any rollbacks. For two processors, function-based scheduling gives a good result, similar to first-come-first-serve scheduling. Note that first-come-first-serve scheduling with Hjalmarson's algorithm does not outperform function-based scheduling with Tikekar's algorithm up to three processors. Speedup for function-based scheduling is limited if more than two processors are used. The reason for this is shown in Figure 21; idle time increases drastically since too many tasks are not executable. In fact, this way of scheduling proves once more that speedup for speculative execution is limited to a significant degree when resources are locked [72].

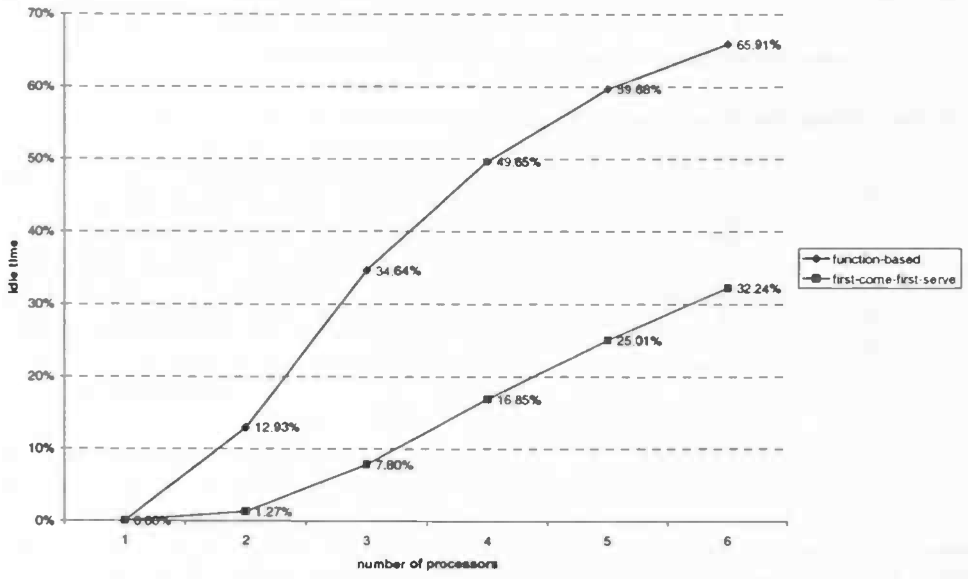

Figure 21 Comparison idle time Tikekar's algorithm with a speculation depth of 12 tasks

Somehow, we have to relax locking restrictions. An obvious way to do this is to lock functions that often cause data collisions but to allow execution for functions that rarely cause data collisions. As a first step, data collisions in first-come-first-serve scheduling were analysed. This was done for Tikekar's scheduling algorithm, a speculation depth of 12 tasks and two processors. The two-processor case was chosen to ease analysis of colliding functions; for three processors it is not always clear what functions caused a data collision in cascaded and repeating rollbacks. Table 3 shows some statistics on the data collisions in this simulation.

total number of collisions	2,557	
number of collisions, without repetitions	715	(28%)
total number of functions installed	899	
total number of functions used in recording	299	
number of functions that cause at least one collision in this recording	51	(17%)
number of functions that never collide in this recording	248	(83%)
total number of function calls	28,047	
number of calls to functions that never collide	8,172	(29%)
number of calls to functions that collide at least once	19,836	(71%)
number of collisions on single variables and single arrays	578	(81%)
number of collisions on arrayed structure variables	137	(19%)

Table 3 Some statistics on data collisions

Since rolled back tasks are re-started as soon as possible, these tasks might collide again with the same task it collided with earlier. The first data collisions only constitute 28% of all collisions. It is these first data collisions we focus on in Figure 22. By trying to avoid the first data collisions, repeating collision will not occur. Figure 22 shows the *collision frequency* for the 25 functions that collide most often. Collision frequency is defined as '*the quotient between the number of times a function causes a data collision and the total number of data collisions*'. Note that these 25 functions constitute 91% of all data collision, the first 10 functions cause 68% of all data collisions. Single variables and single arrays cause 81% of all data collisions, while these constitute only 42% of all variable references (see "Table 2 Some statistics on the pre-processed data" in section 4.2.4). Arrayed structure variables are referred most often (58%), but only cause 19% of all data collisions.

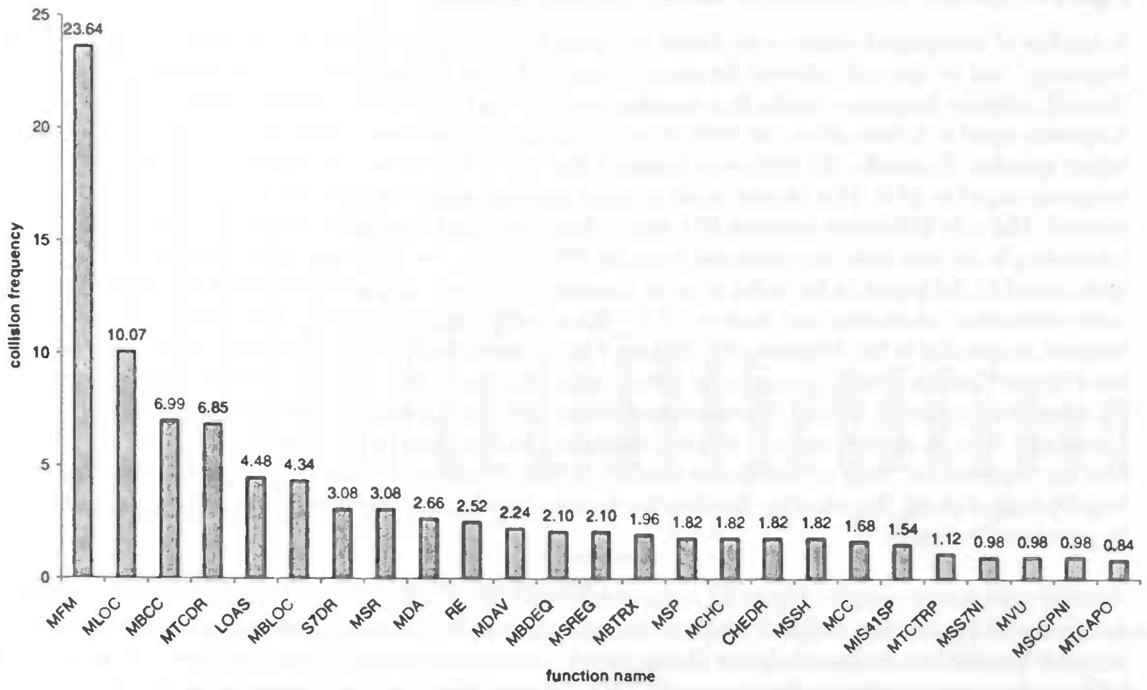

Figure 22 Collision frequency for functions in recording

The information in Figure 22 can be used for making scheduling decisions. For now, this list is static but can later be build up and maintained during run-time, resulting in a dynamic scheduling algorithm. The scheduling rule is extended by allowing all functions that have a collision frequency below a certain limit to execute. Results on speedup on shown in Figure 23. The dashed line is added for comparison and indicates speedup for first-come-first-serve scheduling.

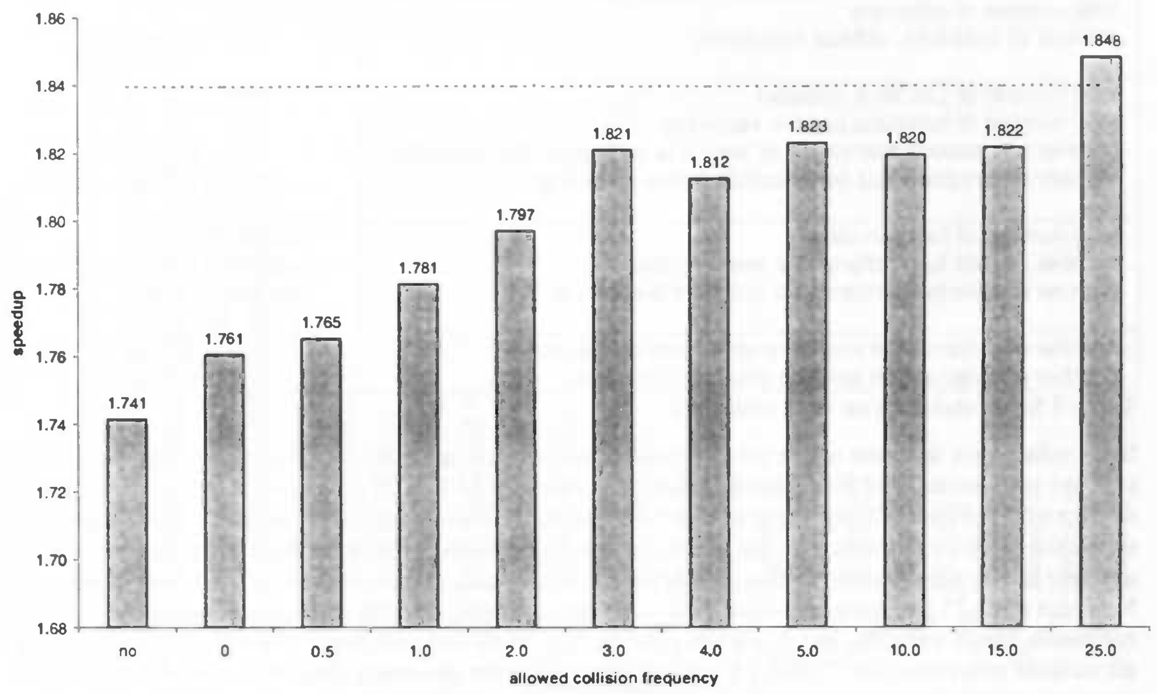

Figure 23 Speedup as a function of allowed collision frequency

A number of unexpected results were found in Figure 23. Firstly, the difference between 'no allowed collision frequency' and an allowed collision frequency equal to 0. This is explained from the statistics in Table 3. No allowed collision frequency results in a speedup exactly equal to plain functional scheduling. A collision frequency equal to 0 does allow the 29% of calls to functions that never collide to execute. This results in a higher speedup. Secondly, the difference between first-come-first-serve scheduling and an allowed collision frequency equal to 25.0. This should result in equal speedup, since the collision frequency allows all tasks to execute. The only difference between this way of function-based scheduling and first-come-first-serve scheduling is the way tasks are extracted from the FIFO queue. For function-based scheduling, the scheduler looks ahead in the queue as far as the level of speculation allows. This is beyond the limit of 10 tasks in first-come-first-serve scheduling (see section 4.2.1, "What traffic data is needed"). A possible explanation for the increase in speedup is the following hypothesis. The increased look-ahead capability of the scheduler causes more distinct transactions to reside in the actual (enlarged) queue than first-come-first-serve scheduling. This increases the number of distinct functions used, since different transactions most likely use different services (i.e. functions). This, in its turn, reduces rollback frequency and increases execution time. This is supported by the data on "Appendix C Data on simulation results". If this hypothesis is correct, it implicitly also means that a heavily loaded Multi-X system performs better than an under-loaded system (in speedup, assuming all tasks can be processed in time)[4].

Another unexpected result in Figure 23 is the overall path of the speedup curve. The curve was expected to increase with the allowed collision frequency up to a certain maximum, the optimum collision frequency, and to decrease beyond this maximum. From Figure 24 it becomes clear that idle time does decrease when the allowed collision frequency increases. However, rollback frequency does not increase monotonously. By increasing the allowed collision frequency from 4.0 to 5.0, rollback frequency decreases from 11% to 9%.

[4] In fact, Wall [70] discovered the same, but reasoned the other way around: performance is restricted because of the limited number of instructions in the scheduler's pool.

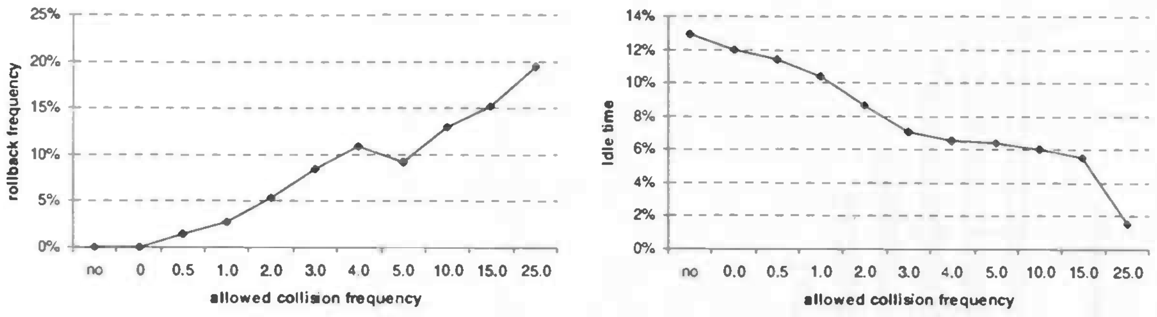

Figure 24 Rollback frequency (left) and idle time (right) as a function of allowed collision frequency

This behaviour can be explained by comparing Figure 25 to Figure 22. Usage in Figure 25 is defined as the quotient of the number of tasks using that function and the sum of unique functions per task. By increasing the allowed collision frequency from 4.0 to 5.0, the functions LOAS and MBLOC are allowed to execute without waiting. The former function is not used that often (21st in Figure 25), the latter is one of the most frequently used functions (2nd in Figure 25). By allowing MBLOC to execute without waiting, we release a frequently used function that leads to relatively few rollbacks. Since MBLOC is allowed to execute without waiting, the relative chance for other tasks to collide decreases which shows in Figure 24.

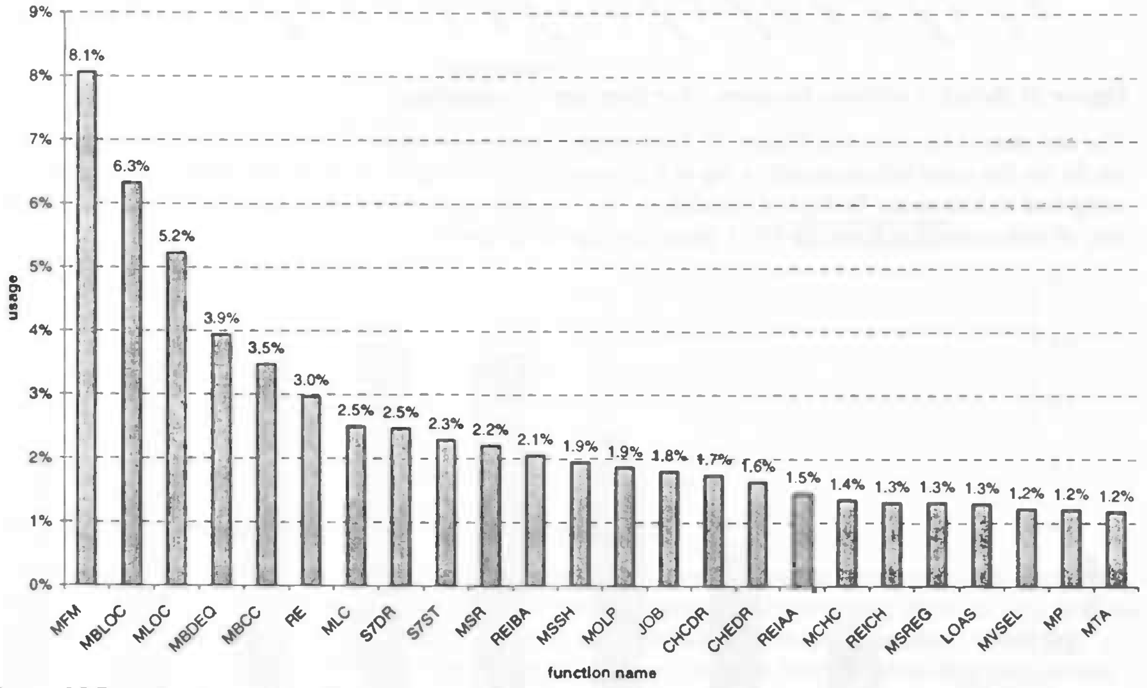

Figure 25 Distribution of the 25 mostly used functions

In fact, Figure 22 lists an *absolute collision frequency*. Instead, collision frequency should be expressed relative to the usage of a function. *Relative collision frequency* is defined as '*the quotient of the number of times a function leads to a data collision and the number of times a function is used by tasks*'. Every function is only counted once per task, even if a function is used several times in a task or leads to several data collisions for that same task. This results in Figure 26. The list is similar to Figure 22 but sorted differently. As the absolute collision frequency, this list is static but can later be built up and maintained at run-time. There is an initialisation problem, which also shows in Figure 26. The first function CCD is used only three times and causes one rollback; this means function CCD causes a rollback in 33% of all cases (as a comparison: the next function, MTCDR, is used 150 times and leads to 49 rollbacks). We should first run the simulator for a while to build up a list of relative collision frequencies before using it. For now, there is no more input data available and we ignore this initialisation problem.

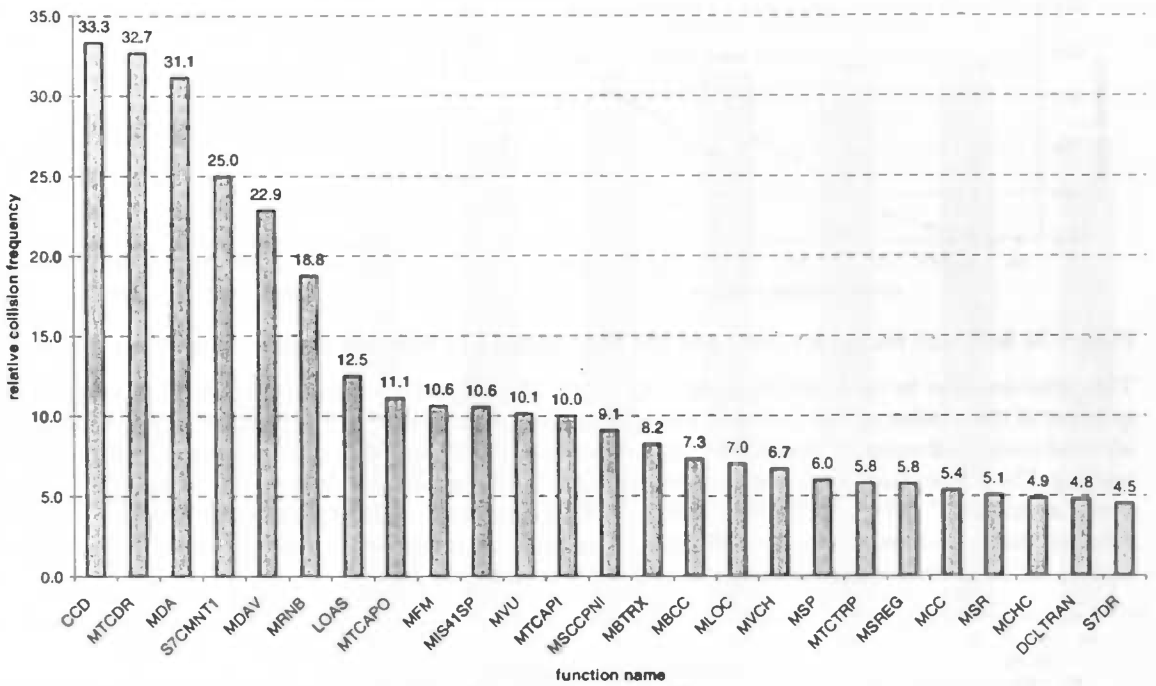

Figure 26 Relative collision frequency for functions in recording

The new results are shown in Figure 27. First-come-first-serve is added as a dashed line for comparison. Here, we do see the expected curve with a top at a relative collision frequency of 20. Speedup is increased with 1% compared to first-come-first-serve scheduling. But this also includes the extra speedup gained from a different way of task extraction from the FIFO queue (as explained above).

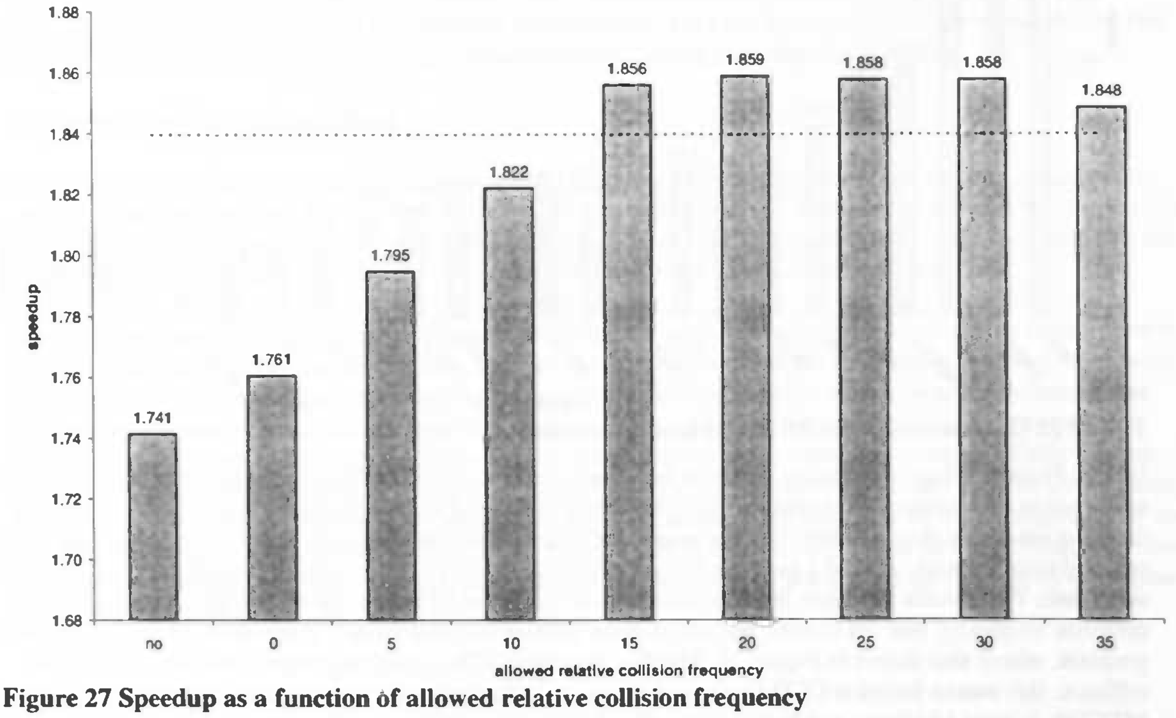

Figure 27 Speedup as a function of allowed relative collision frequency

Although speedup gain is very moderate, rollback frequency might be of more importance. The 1.840 speedup for first-come-first-serve scheduling was achieved at a rollback frequency of 20.4%; the 1.859 speedup for functional scheduling was achieved with only 15.0% rollback. If we had taken overhead into account in the model, this lower rollback frequency would have caused a bigger difference in speedup. Rollback frequency is shown in Figure 28. As expected, rollback frequency does increase monotonously with the allowed relative collision frequency.

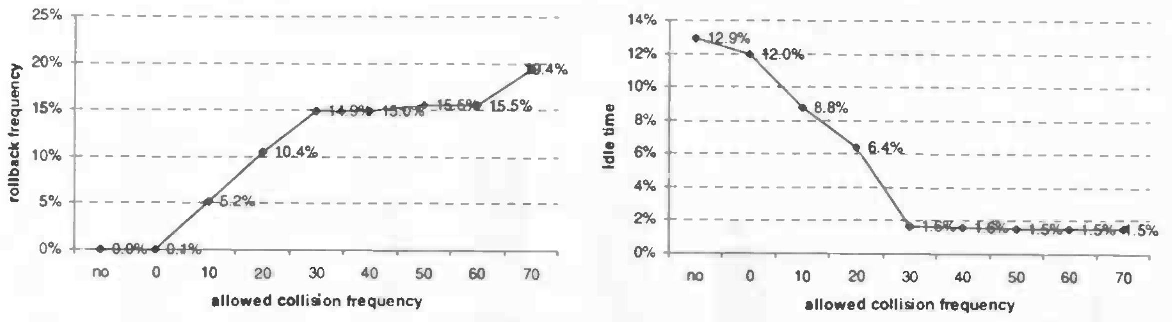

Figure 28 Rollback frequency (left) and idle time (right) as a function of relative collision frequency

So far, we assumed perfect prediction. How will performance change when relaxing this assumption? Figure 29 shows results for a limited prediction of 1, 2 and 3 tasks. Every prediction itself is still perfect. E.g. a perfect prediction, two ahead. In this case, up to two functions in a task can be foreseen without any error, but a third function can not be foreseen at all.

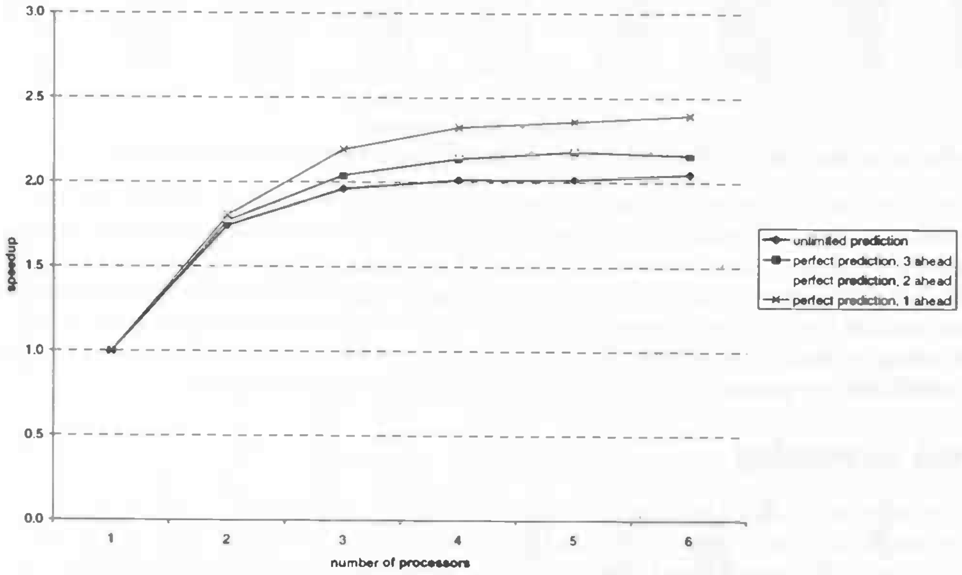

Figure 29 Function-based scheduling with limited prediction

Limited prediction was expected to decrease performance. However, it increases performance since locking is relaxed. No prediction mechanism is needed to implement function-based scheduling since the first function in a task is always known from the initiating message. As a last measurement on function-based scheduling, a simulation was done for the relative collision frequency approach without function prediction (i.e. perfect prediction, 1 function ahead). Results are shown in Figure 30. The results are similar to Figure 27; the optimum collision frequency is still at 20. Optimal speedup is exactly equal to Figure 27 and no price is paid for the limited prediction. Both unlimited perfect prediction and limited prediction result in a 15.0% rollback frequency. This is 5.4% below first-come-first-serve scheduling.

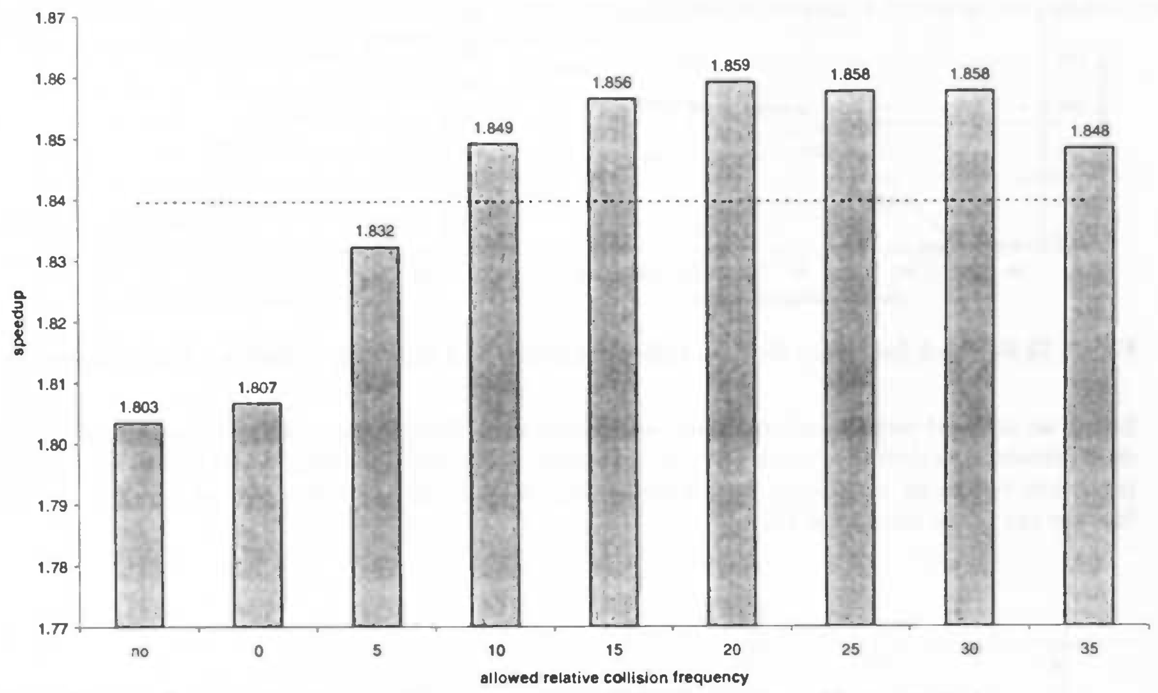

Figure 30 Speedup as a function of allowed relative collision frequency with no prediction

Because of limited time, measurements for function-based scheduling have only been done for the two-processor case and a speculation depth of 12 tasks. When increasing the number of processors, the collision frequency list should be optimised for that case. Also, according to Figure 14 and Figure 17, speculation depth will have to increase to avoid idle cycles caused by a limited level of speculation. Cache-hit ratio has not been investigated, but might increase slightly if enough processors are available and the same task is assigned to the same processor. When taking overhead into account, the difference between first-come-first-serve scheduling and function-based scheduling is expected to increase in favour of function-based scheduling.

5.4 Source-based scheduling

The goal of source-based scheduling is to map individual subscribers to individual processors. Since every subscriber has its own data, cache-hit ratio is expected to improve. Since different subscribers will use different services in time, i.e. functions, source-based scheduling is also expected to decrease the number of rollbacks. Balancing and maximising processor load is assumed not to raise too many difficulties.

To investigate the feasibility of source-based scheduling, the distribution of external messages from different remote processors has been examined first. In this examination, it turned out that almost half of all transactions are not initiated by a message from a remote processor but by a message from the operating system. These transactions are used for various administrative purposes. We call these transactions operating system transaction, as distinct from real traffic transactions. Table 4 gives some statistics (compare "Table 2 Some statistics on the pre-processed data"). Although operating system transactions constitute 48% of all transactions, all tasks directly or indirectly started by an operating system message are only 13% of the total number of tasks. Furthermore, these tasks are small; the sum of all cycles of these tasks is only 6% of all traffic. Based on these observations it can be concluded that operating system transactions will not disturb the concept of source-based scheduling very much.

total number of transactions	2,672	
total number of transactions initiated from a remote processor	1,383	(52%)
total number of OS transactions	1,289	(48%)
average number of tasks per OS transaction	1.42	
total number of tasks in recording	14,045	
number of OS tasks	1,827	(13%)
total number of cycles in recording	8,078,187	
number of OS cycles	474,151	(6%)

Table 4 Some statistics on operating system transactions

Remote processors serve several devices and every device serves several subscribers. Internally, remote processors are organised in several software modules. Mostly, module-device mapping is one-to-one or one-to-many, but in some situations several modules can serve one device [19]. From the recordings used here, both initiating remote processors and its module could be derived for 75% of all externally generated (traffic) messages. An even finer-grained mapping from external message to device or subscriber could not be traced back. The module-device mapping is specific for every switching system configuration and could not be recovered for the recordings used. Figure 31 shows the results for the remote processor ids. Apart from some exceptions, load seems to be distributed fairly well. Even load in time was analysed for a number of individual remote processors. This distribution proved to be quite fair too.

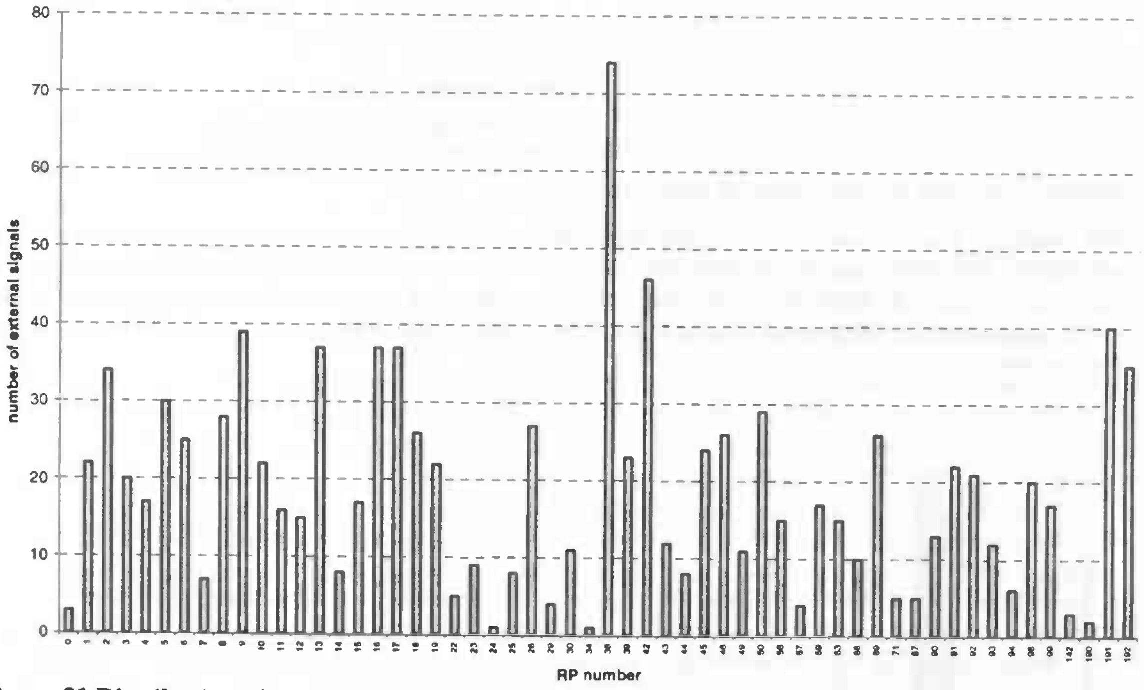

Figure 31 Distribution of external signals from remote processors

For source-based scheduling we will base scheduling decisions on the combination of remote processor id and module id. Once the first task in a transaction is assigned to a processor, all following tasks in that transaction will be assigned to the same processor. This will exploit cache affinity for that processor. The key combination of processor id and module id was fairly unique for the given input data. Only 11 times, two consecutive transactions had the same key. Based on this knowledge, the following scheduling rule was formulated and implemented: "*for every external message, start its task on the least loaded processor*". Note that specific remote processor or module information is not included in this rule and that this rule does not make a difference between operating system transactions and real traffic transactions. At the later stage, this might be added, e.g. to exploit cache affinity even better. Processor load is defined as '*the number of uncommitted task that have run or are running on that processor*'. At any time, there is at most one task running on every processor. Tasks that have run on a processor are either ended successfully and wait for commit or have been rolled back and wait for the processor to become available. We count tasks waiting for commit to the load since these tasks might be rolled back later. Note that this way of scheduling cannot lead to deadlocks since every task is assigned to a processor and the oldest task will never be rolled back. Results are shown in series "source-based, speculation depth 12" of Figure 32. As a first step, a maximum speculation depth of 12 was used as in the Multi-X prototype.

In the simulations for source-based scheduling, cache-hit ratio cannot be measured. However, we can measure rollbacks and test the assumption that load balancing is fair.

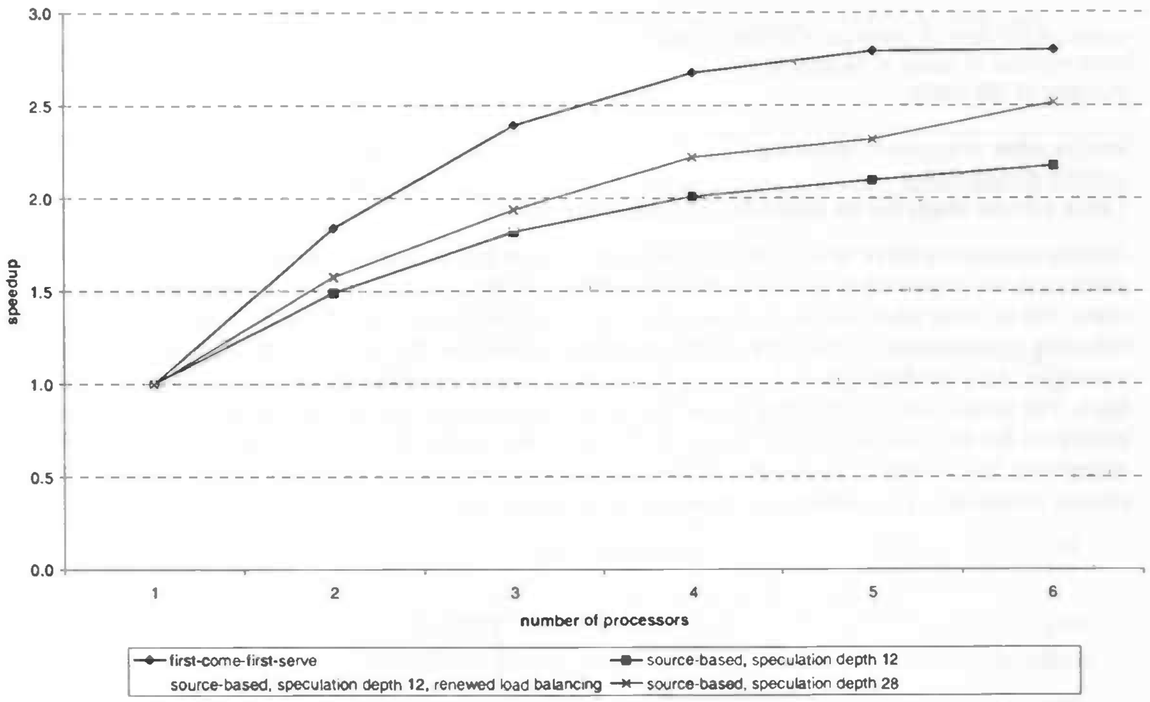

Figure 32 Speedup for source-based scheduling

The results for source-based scheduling are disappointing. Speedup is much lower than first-come-first-serve scheduling. The main cause for this behaviour is load imbalance, as shown in series "source-based, speculation depth 12" of Figure 33. Here, the four-processor case is depicted. For every processor, the number of non-idle cycles is compared to the average number of non-idle cycles. The last column shows the absolute average for all four processors.

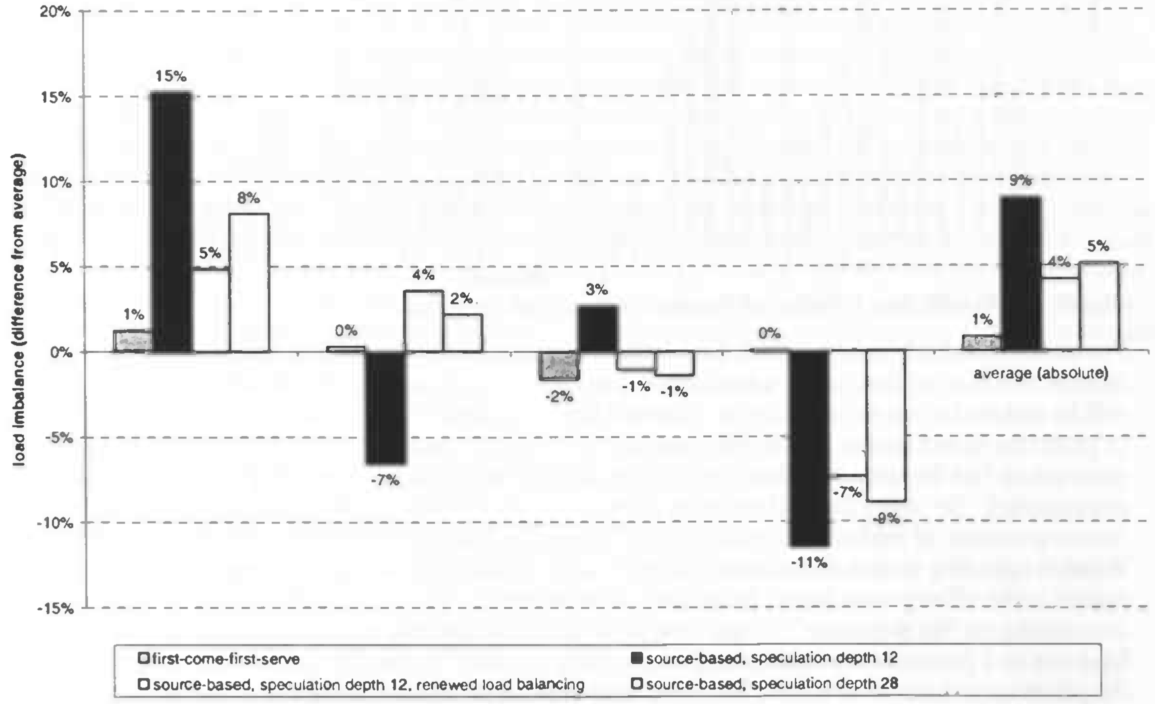

Figure 33 Load balancing for source-based scheduling compared to first-come-first-serve for 4 processors

If several processors are idle, the next task is scheduled at the lowest processor number. This explains the high figures for processor one and the low figures for processor four. Even first-come-first-serve scheduling has 1%

load imbalance; this is only due to the fact that processors remain idle at the very last part of the simulation when there are no more tasks to be scheduled. Why does source-based scheduling lead to an unbalanced load? After detailed analysis, it turned out that this is mainly caused by the limit on speculation. Whereas a limit in speculation depth caused many idle cycles for first-come-first-serve scheduling, the penalty for source-based scheduling is even worse since follower tasks are automatically assigned to the parent task's processor. The total idle time is increased compared to first-come-first-serve scheduling; see series "source-based, speculation depth 12" of Figure 34.

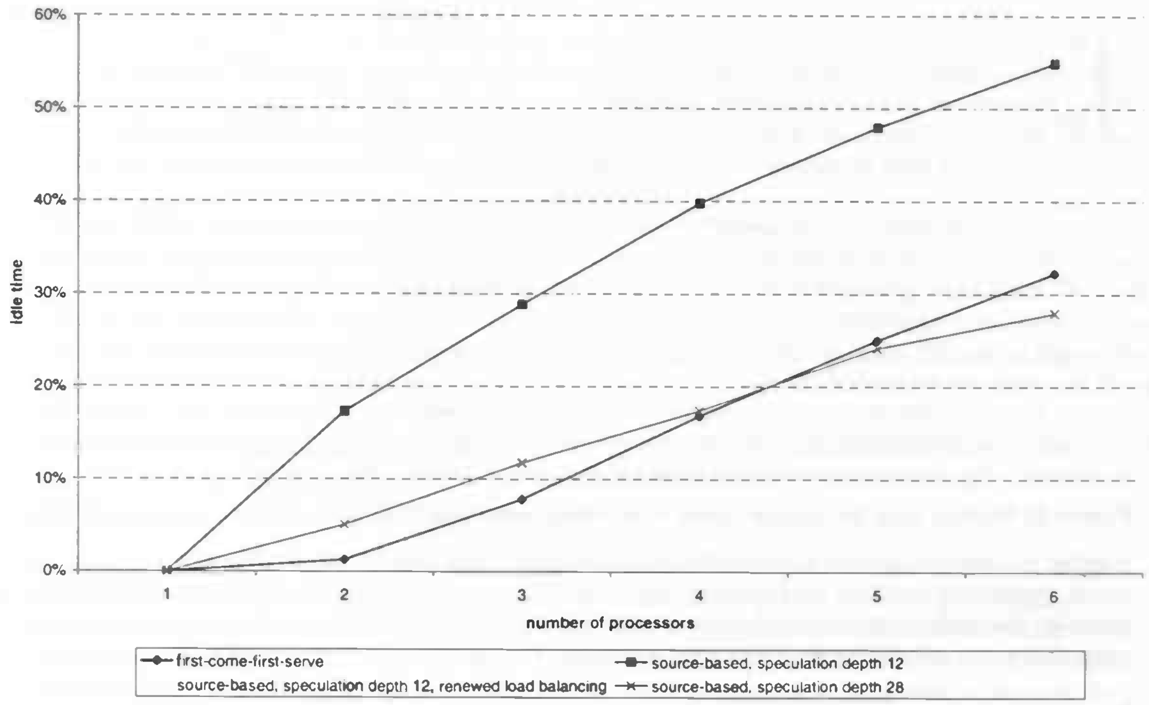

Figure 34 Idle time for source-based scheduling compared to first-come-first-serve scheduling

Is it possible to refine the load-balancing rule? In the current rule, the contents of the FIFO queue is not taken into account. Here, a lot of follower tasks might reside which have already been assigned to a specific processor. Therefore, the definition of processor load was restated to '*the number of uncommitted task assigned to that processor*'. This includes all tasks; those that have been running, are running and will run on a processor. Results are shown in series "source-based, speculation depth 12, renewed load balancing" in Figure 32, Figure 33 and Figure 34. Load balancing is improved and idle time decreases, but the increase in speedup is not substantial. The reason is shown in Figure 35; the number of rollbacks and wasted time increases with the new load balancing rule. With four processors, wasted time is much less than for first-come-first-serve scheduling but speedup is restricted too much by the number of idle cycles. Wasted time seems to decrease for a larger number of processors. This is only caused by the fact that wasted time is expressed relative to the total number of cycles. In absolute terms, the number of wasted cycles did increase monotonously with the number of processors.

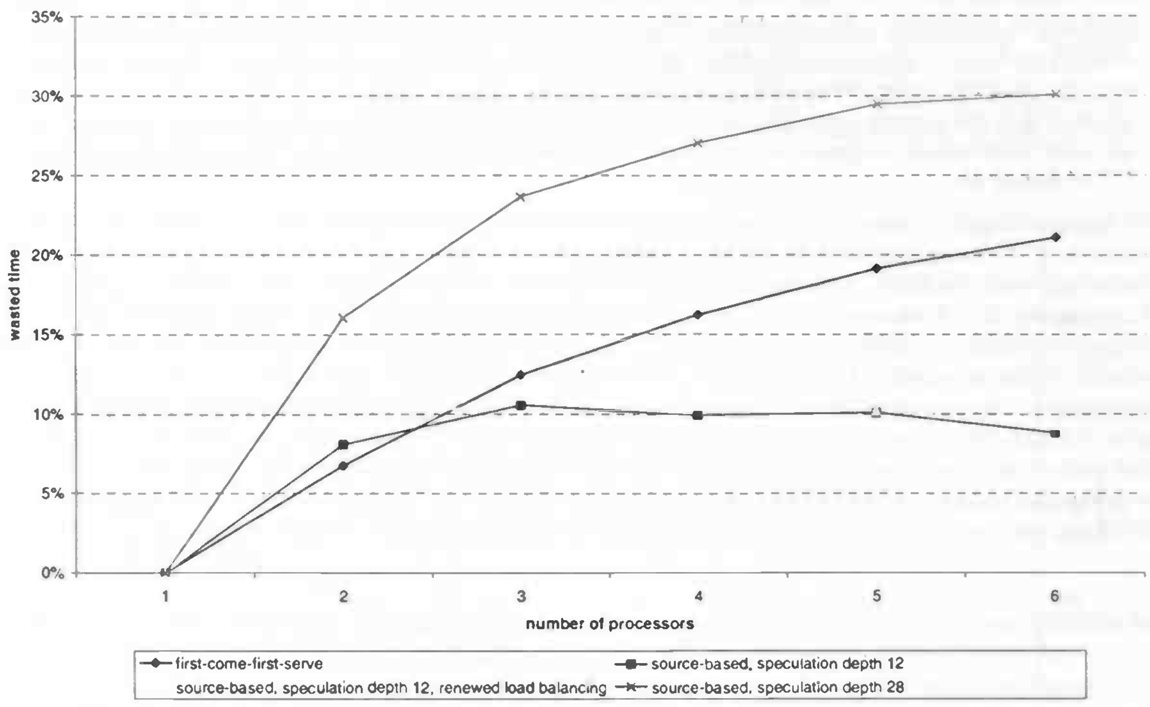

Figure 35 Wasted time for source-based scheduling compared to first-come-first-serve scheduling

Another solution to decrease load imbalance is to enlarge speculation depth. Results are shown in series "source-based, speculation depth 28" in Figure 32, Figure 33, Figure 34 and Figure 35. The increase in speculation depth improves load balancing, but a high price is paid in wasted time. Resulting speedup does not come up to first-come-first-serve scheduling for a speculation depth of 12 tasks.

From these measurements, it can be concluded that load-imbalance *does* lead to a deterioration in performance. Even if cache affinity can be exploited and measured, a 30% increase in performance is needed to come to the same speedup as first-come-first-serve scheduling. It is very doubtful if this can be accomplished. The underlying cause for load imbalance is not an uneven distribution of tasks from remote processors. This distribution proved to be balanced both in time and in space (Figure 31). Limits on speculation depth proved not to be the real problem either. The real reason seems to be the strict task ordering combined with the fact that every follower is sent to the same processors as its initiator. The latter demands a higher speculation depth. The former causes larger sequences of cascaded rollbacks, resulting in an increased wasted time. Source-based scheduling was also expected to lead to a decrease in rollbacks. This assumption seems to be supported by results in Figure 35. However, when comparing similar speedup, the assumption does not hold. E.g., speedup is 1.84 for first-come-first-serve scheduling with two processors and a speculation depth of 12 tasks. This leads to a rollback frequency of 20.4%. For source-based scheduling with 3 processors and the same level of speculation, a speedup of 1.82 and 1.85 is achieved by the two ways of load balancing. For these figures, rollback frequency is 21.6% and 20.7% respectively; similar to first-come-first-serve scheduling (see "Appendix C Data on simulation results" for detailed data).

6 Conclusions

In this chapter, some conclusions on this thesis work are formulated. Section two summarises results from measurements. Section three present some general conclusions on scheduling and Multi-X. In the first section, the main design decisions are evaluated.

6.1 Evaluation of design decisions

After having explored scheduling in chapter 3, a number of design decisions have been made on measuring the proposed ways of scheduling. The most important design decisions were: the choice to implement a simulator using live traffic data as input; the simplifications not to link this scheduling simulator to the cache simulator and not to take into account any overhead; the way timing information is reconstructed from the input data.

1. The decision to implement a simulator was based on a number of arguments, which proved to hold throughout this thesis project. Implementing scheduling algorithms in the Multi-X prototype would not have been possible. The prototype has only recently been proved to process a low traffic load correctly. Platform stability and measurability are still problematic. Furthermore, it would not have been possible to test new ideas like control speculation for function-based scheduling. The traffic data used as input to the simulator proved to be much more complex than expected. A great deal of time has been spent on extracting the right information and verifying the resulting data. The resulting data let to a better understanding of multiprocessor issues. E.g., single variables and single arrays turned out to cause by far most data collisions. These insights would have been missed if scheduling algorithms were evaluated using a mathematical model.

2. It was decided not to link the scheduling simulator to the cache simulator. The reasons for this were to avoid dependencies on the thesis project resulting in the cache simulator and the time needed to implement and (re-) measure performance. It would have been interesting to see performance results with cache simulation. However, for the reasons mentioned the simplification was mainly forced and hard to avoid. Overhead was not taken into account for two reasons: it was hard to quantify overhead; and we want to measure the plain performance of scheduling without any disturbances. The simplification in overhead made it easy to implement the simulator and analyse its results. Taking into account overhead would have led to fewer results. At this moment, it is fairly well decided in what way the prototype will be implemented; it should be possible by now to quantify overhead. The plain performance without overhead is described in this report. These results are very useful, but are only a first step towards the understanding of scheduling in Multi-X.

3. From the live recordings, no timing information could be extracted. To solve this problem, instruction execution times were taken from statistical information. Internal messages are inserted in the FIFO queue as soon as the initiating task commits; as in the Multi-X prototype. External messages are inserted in the FIFO queue as soon as the number of messages in the queue drops below a certain limit. The limit was set high, so a high traffic load is simulated. In fact, *throughput* is measured this way because the queue is never empty. This might result in too pessimistic figures for speedup. In reality, there will be intervals in time where no messages or very few messages reside in the FIFO queue. This means less data dependencies, which in its turn leads to a higher speedup. Actually, speedup is hard to define in this case but should somehow be related to system load. For the recordings used throughout this report, the real system load was only 35%. Obviously, compressing time the way done here has led to data dependencies that did not exist in reality; e.g., two tasks residing at the same time in the FIFO queue. Then again, if 30% processor load is representative, a multiprocessor system is superfluous. A multiprocessor system *will* in reality be loaded with a higher number of transactions per second. One of the prerequisites for Multi-X is unmodified code; i.e. unmodified traffic. The approach in this thesis is the best we can do for now. At a later stage, performance should be measured with various limits in the number of FIFO-tasks and with various time intervals for the insertion of externally generated messages.

6.2 Results from measurements

Four ways of scheduling were investigated in this thesis project; optimal scheduling in order to define a theoretical upper bound on performance and three practical algorithms. Not all combinations of parameters could be measured because of limited time. A lot remains to be measured; e.g. overhead, cache performance, other types of traffic, other speculation depths for functional scheduling, variable queue length.

1. Optimal scheduling was measured for both data collision detection algorithms. Hjalmarson's algorithm resulted in a speedup of 2.47. Tikekar's algorithm resulted in a maximum speedup of 6.63. With a speculation depth limited to 12 tasks, maximum speedup dropped to 3.18. These results are somewhat disappointing since no overhead is taken into account and optimal scheduling assumes an infinite look-ahead capability of the scheduler.

2. First-come-first-serve scheduling is used at this moment in the prototype. It is easy to implement and does not lead to much extra scheduling overhead. Configurations from one up to six processors were measured. Speedup for Hjalmarson's algorithm comes up to 92% of optimal scheduling. Tikekar's algorithm comes up to 88% of optimal scheduling, for a speculation depth of 12 tasks. These figures are for six processors. In reality the optimal number of processors will be lower, probably four. Since we do not take overhead into account, speedup saturates with a high number of processors, where it would decrease in reality. Tikekar's algorithm proved to perform better, both in the simulations done here and in the prototype's implementation. Therefore, all other measurements have been done on this algorithm only. The level of speculation in the Multi-X prototype is currently set to 12 tasks. Simulations show that an increase in speculation does not lead to a large increase in performance. Only for more than three processors, we see an increase in speedup when increasing speculation depth to 28 tasks. Increasing speculation depth beyond 28 tasks will not pay off in real implementations.

3. Function-based scheduling is an attempt to avoid unnecessary rollbacks by examining what functions will be executed in a task. Simulations proved that knowing the first function to execute is sufficient for implementing function-based scheduling. This first function can easily be traced from the initiating message. No (software) control predicting mechanisms are needed. Function-based scheduling led to a very moderate increase in speedup of 1% compared to first-come-first-serve scheduling. More importantly, rollback frequency was decreased substantially. This will increase function-based scheduling performance when overhead is taken into account. Even cache-affinity might lead to some extra performance gains. The proposed method for implementing function-based scheduling is a dynamic collision frequency list. This can be implemented with a fairly low overhead.

4. The idea behind source-based scheduling is to map individual subscribers to individual processors. This was expected to increase cache-hit rates and decrease the number of rollbacks. Balancing load was assumed not to raise many difficulties. Simulations show these assumptions to be incorrect. The number of rollbacks did not decrease when comparing speedup similar to first-come-first-serve scheduling. Load balancing does lead to problems, expressed either in a high number of idle cycles or a high number of wasted cycles. Source-based scheduling is dissuaded unless traffic dimension changes or cache affinity can increase performance by 30%. The former was discussed in the last part of section 6.1. The latter is not likely to be realised.

When comparing performance of the scheduling algorithms, first-come-first-serve scheduling performs very well. Ideas from function-based scheduling can be additive to first-come-first-serve. The resulting performance is quite near optimal scheduling. It will be hard to come closer to optimal scheduling, since this algorithm assumed an infinite look-ahead impossible to implement.

6.3 General conclusions, discussion and future topics

A number of general conclusions can be formulated:

1. This simulation was the first simulation of the Multi-X model with all relevant parameters included. Theoretically, the concept has been proved to hold. But is it possible to use the concept in a real implementation? This will mostly depend on the overhead induced by speculative execution. Let us make some rough estimation. From "Table 2 Some statistics on the pre-processed data", it becomes clear that 35% of all instructions refer to a variable. All these instructions have to use the data collision detection algorithm. This means that every single read or write instruction expands to at least six instructions: one to get a lock; one to read the marker field; one to classify the operation from the collision detection algorithm; one to write the marker field; one for the actual read or write; and one to release the marker field lock. An APZ CP instruction takes on average 5 cycles. Let us assume the extra five instructions can be implemented efficiently and optimised by cache affinity, resulting in 1 cycle execution time for each instruction. Then every 100 instructions, i.e. 500 cycles, expand to 675 cycles. This is still without any overhead on rollback, commit and scheduling. Let us assume we use four processors and a level of speculation of 12 tasks. This

leads to 16% wasted time in our measurement. This can be improved by function-based scheduling, let us assume down to 10%. Then the original 500 cycles expand to approximately 750 cycles. A 50% overhead with optimistic assumptions! The original speedup will therefore halve from 2.68 to 1.34. Of course, a lot of assumptions have been made here and figures have to be checked. But it does point out that expectations should not be set too high. Although 35% increase in real performance is not much, it is gained without changing any other application code and using commercially available components. If the goal is doubling APZ CP performance every third year, Multi-X is one step towards that goal. A step relatively cost effective and fast to implement. Furthermore, it is one step towards a new multiprocessor architecture [65]. New application code can be implemented with this architecture in mind, whereas existing application code is able to run without changes.

2. From measurements done here, it becomes clear that scheduling should really be seen as an *optimisation*. Only a moderate performance increase can be accomplished by scheduling. Other parameters (see "Figure 6 Relation of scheduling to other parameters in the Multi-X prototype") are much more important than scheduling. E.g., the difference between Hjalmarson's algorithm and Tikekar's algorithm for the 'collision detection algorithm' parameter. Even an efficient implementation will probably lead to more performance gains than an advanced scheduling algorithm.

3. Source-based scheduling resulted in poor performance caused by load imbalance. Clearly, scheduling is too restricted by the task numbering used in the Multi-X model. This numbering virtually dictates some form of first-come-first-serve scheduling, since it does not allow changing execution order. In (real-time) database literature, similar problems are known and analysed [15, 54]. Here, many constrains are artefacts. This results in an over-constrained and infeasible scheduling problem. In the Multi-X model, task numbering allows for a relatively easy implementation. At the same time, the induced task ordering is clearly more restricted than strictly necessary. Can task ordering be relaxed in the Multi-X model?

4. With function-based scheduling some performance was gained. This was achieved by analysing traffic behaviour and translating that knowledge to scheduling rules. There is still a lot to investigate on traffic behaviour. E.g., can variables that cause data collisions be classified more precisely? Where in the tasks are references done? Can this be exploited when scheduling new tasks?

References

1. Robert K. Abbot, Hector Garcia-Molina. *Scheduling Real-Time Transactions: A Performance Evaluation*. ACM Transactions on Database Systems, Vol. 17, No. 3, September 1992.

2. Gene M. Amdahl. *The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. AFIPS Conference Proceedings, pp. 483-485, 1967.

3. Naser S. Barghouti, Gail E. Kaiser. *Concurrency Control in Advanced Database Applications*. ACM Computing Surveys, Vol. 23, No. 3, September 1991.

4. Philip A. Bernstein, Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, 1997.

5. Thomas L. Casavant, Jon G. Kuhl. *A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems*. IEEE Transactions on Software Engineering, Volume 14, Number 1, January 1988.

6. Chalmers University of Technology. *High-Performance Computer Architecture Group*. URL: http://www.ce.chalmers.se/~case/.

7. Michael K. Chen, Kunle Olukotun. *Exploiting Method-Level Parallelism in Single-Threaded Java Programs*. Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Paris, France, October 12-18, 1998.

8. John G. Cleary, Richard H. Littin, J.A. David McWha, Murray W. Pearson. *Constraints on Parallelism Beyond 10 Instructions Per Cycle*. University of Waikato, New Zealand, Report No: 97/27, 1997.

9. John G. Cleary, Murray W. Pearson, H. Kinawi. *The Architecture of an Optimistic CPU: The WarpEngine*. Proceedings of the Hawaian International Conference of System Science, Hawaii, USA, Vol. 1, pp. 163-172, January 1995.

10. David E. Culler, Jaswinder Pal Singh, Anoop Gupta. *Parallel Computer Architecture - a Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

11. Sascha Dierkes, Ludger Frese. *Load Balancing with a Fuzzy Decision Algorithm: Description of the Approach and first Simulations*. University of Dortmund, Chair for Computer Science I, June 20, 1995.

12. Niklas Dykström. *Undersökning av parallelt exekverbara trådar i AXE 10-tillämpningar*. Kungliga Tekniska Högskolan, NADA, 1999.

13. Johann Eder, Walter Liebhart. *Workflow Transactions*. In: P. Lawrence (ed.) - Workflow Handbook 1997. Handbook of the Workflow Management Coalition WfMC., Wiley & Sons, pp 195 – 202, 1997.

14. Johann Eder, Euthimios Panagos, Michael Rabinovich. *Time Constraints in Workflow Systems*. In: M. Jarke, A. Oberweis (eds.): Advanced Information Systems Engineering, 11th International Conference, CAiSE'99, Springer Verlag, LNCS 1626, Heidelberg, Germany, pp. 286-300, June 1999.

15. Cecilia Ekelin, Jan Jonsson. *Real-Time System Constraints: Where do They Come From and Where do They Go?* Proceedings of the International Workshop on Real-Time Constraints, Alexandria, Virginia, USA, October 1999.

16. Ericsson Telecom AB. *AXE 10 Architecture Description*. Ericsson Internal, No.: EN/LZT 101 1845-R2, 1995.

17. Ericsson Telecom AB. *AXE Survey. The platform and the applications*. Ericsson Internal, 1998.

18. Ericsson Telecom AB. *Plex-C 1*. Ericsson Internal, No.: EN/LZT 101 1279 R5A, 1997.

19. Ericsson Telecom AB. *Plex-C 2*. Ericsson Internal, No.: EN/LZT 101 1280 R4A, 1997.

20. Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, Whay S. Lee. *The M-Machine Multicomputer*. Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, Michican, USA, 1995.

21. John L. Gustafson. *Reevaluating Amdahl's Law*. Communications of the ACM (CACM), Volume 31, Number 5, pp. 532-533, May 1988.

22. Babak Hamidzadeh, David J. Lilja. *Dynamic Scheduling Strategies for Shared-Memory Multiprocessors*. Proceedings of International Conference on Distributed Computing Systems, May 1996.

23. Lance Hammond, Ben Hubbert, Michael Siu, Manohar Prabhu, Mike Chen, Kunle Olukotun. *The Stanford Hydra CMP*. IEEE MICRO Magazine, March-April 2000.

24. Lance Hammond, Mark Willey, Kunle Olukotun. *Data Speculation Support for a Chip Multiprocessor*. Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 1998.

25. John L. Hennesy, David A. Patterson. *Computer Architecture - A Quantitive Approach*. Morgan Kaufmann Publishers, 1996.

26. Tomas Hjalmarson. *Multi-X Data Structures and Algorithms*. Ericsson Internal Document, No.: UAB/M/U-99:005, October 1999.

27. Tomas Hjalmarson, Nikhil Tikekar, Lars-Åke Johansson. *Multi-X August 1999 Report*. Ericsson Internal Document, June 1999.

28. Per Holmberg, Nils Isaksson. *APZ 212 30 – Ericsson's new high-capacity AXE central processor*. Ericsson Review, No. 3, 1999.

29. Quinn Jacobson, Steve Bennett, Nikhil Sharma, James E. Smith. *Control Flow Speculation in Multiscalar Processors*. 3rd International Conference on High Performance Computer Architecture, February 1997.

30. Lars-Åke Johansson. *Load Distribution in Multiprocessor System*. Ericsson Internal Report, No.: UAB/M/L-99:043, November 1999.

31. M. Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, 1991.

32. Iffat H. Kazi, David J. Lilja. *JavaSpMT: A Speculative Thread Pipelining Parallelization Model for Java Programs*. Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), May 2000.

33. Stephen W. Keckler, William J. Dally. *Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism*. 19th Annual International Symposium in Computer Architecture, Queensland, Australia, 1992.

34. Robert M. Keller. *Look-Ahead Processors*. Computing Surveys, Vol. 7, No. 4, December 1975.

35. C.M. Krishna, Kang G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.

36. Ganesh Lakshminarayana, Kamal S. Khouri and Niraj K. Jha. *Wavesched: a novel scheduling technique for control-flow intensive behavioral descriptions*. Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design, San Jose, California, USA, November 9-13, 1997.

37. Monica S. Lam, Robert P. Wilson. *Limits of Control Flow on Parallelism*. Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, May 19-21, 1992.

38. Jack L. Lo, Susan J. Eggers. *Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level Parallelism*. Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995.

39. C. Mohan. *Recent Trends in Workflow Management Products, Standards and Research.* Proceedings of NATO Advanced Study Institute (ASI) on Workflow Management Systems and Interoperability, Istanbul, August 1997.

40. Johan Montelius. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages.* PhD Thesis, Computing Science Department, Uppsala University, May 1997.

41. Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, Gurindar S. Sohi. *Dynamic Speculation and Synchronization of Data Dependences.* Proceedings of the 24th International Symposium on Computer Architecture, Denver, Colorado, USA, June 2-4, 1997.

42. Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

43. NEC Laboratories. *MP98 – A Mobile Processor.* URL: http://www.labs.nec.co.jp/MP98.

44. Linh Nguyen. *The Execution Times in APZ 212 20.* Ericsson Internal, No.: 1/1551-CNZ 211 383 Uen, April 1996.

45. Nils J. Nilsson. *Principles of Artificial Intelligence,* Morgan Kaufmann Publishers, 1980.

46. Mehrdad Nourari, Christos Papachristou, Yoshiyasu Takefuji. *A Neural Network Based Algorithm for the Scheduling Problem in High-Level Synthesis.* Proceedings of the IEEE European Conference on Design Automation, Hamburg, Germany, September 7-10, 1992.

47. Kunle Olukotun, Jules Bergmann, Kun-Yung Chang, Basem A. Nayfeh. *Rationale and Design of the Hydra Multiprocessor.* Stanford University Computer Systems Lab Technical Report CSL-TR-94-645, 1994.

48. Jeffery Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica Lam, Kunle Olukotun. *Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor.* Stanford University Computer Systems Lab Technical Report CSL-TR-97-715, February 1997.

49. Chris H. Perleberg, Alan J. Smith. *Branch Target Buffer Design and Optimization.* IEEE Transactions on Computers, Volume 42, Number 4, April 1993.

50. Johan Petersson, et al. *APZ-TOR: Dimensioning.* Ericsson Internal Report, No.: 10/0062-1/FCP 105 306, January 1999.

51. Lennart Petterson. *Overview of the Multi-X Prototype.* Ericsson Internal Report, No.: UAB/B/T-99:189, November 1999.

52. Lennart Petterson. *Project Specification for Multi-X Stage 2.* Ericsson Internal, No.: UAB/B/X-99:397 Uen, September 1999.

53. Mirco Porcari. *Cache-Simulator of a Multiprocessor Architecture for AXE CP.* Master's Thesis, Chalmers University of Technology, May 2000.

54. Krithi Ramamritham. *Where do Time Constraints Come From and Where Do They Go?* International Journal of Database Management, Vol. 7, No. 2, Spring 1996.

55. Thomas Siljeströmer. *ASA210C, General Assembler Instructions.* Ericsson Internal, No.: 13/1551-ANZ 211 51 Uen, January 1995.

56. Thomas Siljeströmer. *CPS Principles.* Ericsson Internal, No.: 2/1551-ANZ 211 60 Uen, January 1995.

57. Thomas Siljeströmer. *Summary of Reference Information.* Ericsson Internal, No.: 1/1551-ANZ 211 60 Uen, September 1997.

58. James E. Smith. *A study of branch prediction strategies.* Proceedings of the 8th Annual Symposium on Computer Architecture, pages 135-148, May 1981.

59. Gurindar S. Sohi, Scott E. Breach, T. N. Vijaykumar. *Multiscalar Processors*. 22th International Symposium on Computer Architecture (ISCA-22), 1995.

60. Tan H. Soon, Robert de Souze. *Intelligent simulation-based scheduling of workcells: an approach*. Integrated Manufacturing Systems, Document No: 8/1 [1997] 6-23, 1997.

61. Per Stenström. *Architectural Trends for Shared-Memory Multiprocessors*. 30th Hawaii International Conference on System Sciences, 1997.

62. Per Stenström. *Metoder för effektivt utnyttjande av multiprocessorteknologi i transaktionsorienterade system*. Projektplan 2000-2001, Chalmers Tekniska Högskolan, 1999.

63. Per Stenström, Erik Hagersten, David J. Lilja, Margaret Martonosi, Madan Venugopal. *Trends in Shared-Memory Multiprocessing*. IEEE Computer, Vol. 30, No. 12, December 1997.

64. Sun Microsystems, Inc. *MAJC™ Architecture Tutorial*. URL: http://www.sun.com/microelectronics/MAJC/documentation/docs/majctutorial.pdf, 1999.

65. Nikhil Tikekar. *A Step by Step Migration Stategy*. Ericsson Internal Report, December 1999.

66. Nikhil Tikekar. *Flexible Multiscalar Architecture*. Ericsson Internal Note, December 1999.

67. Jenn-Yuan Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, Pen-Chung Yew. *The Superthreaded Processor Architecture*. IEEE Transactions on Computers, vol. 48, no. 9, September 1999.

68. Andrew Tucker. *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors*. PhD Thesis, Department of Computer Science, Stanford University, Technical Report CSL-TR-94-601, November 1993.

69. Dean M. Tullsen, Susan J. Eggers, Henry M. Levy. *Simultaneous Multithreading: Maximizing On-Chip Parallelism*. Proceedings of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June 1995.

70. David W. Wall. *Limits of Instruction-Level Parallelism*. Western Research Laboratory, Research Report 93/6, November 1993.

71. David W. Wall. *Speculative Execution and Instruction-Level Parallelism*. Western Research Laboratory, Technical Note TN-42, March 1994.

72. Fredrik Warg. *A Platform for Evaluation of Multiprocessors in Throughput-Oriented Systems*. Master's Thesis, Luleå Tekniska Universitet, 1999.

73. Philip Yao. *A study of evaluating parallel execution of existing non-parallel telecom applications*. Master's Thesis, Kungliga Tekniska Högskolan, NADA, 1999.

74. Kelvin K. Yue, David J. Lilja. *Designing Multiprocessor Scheduling Algorithms Using a Distributed Genetic Algorithm System*. University of Minnesota, Technical Report No: HPPC-96-03, May 1996.

Appendix A Collision detection algorithms

At this moment, two data collision detection algorithms have been proposed. The first section of this appendix describes Hjalmarson's approach, which is mainly based on locking. The second section describes Tikekar's approach. This algorithm was proposed at a later stage and is more aggressive by allowing a larger speculation depth per variable. It is important to notice that all instruction in the APZ CP that access a variable are either read instructions from one variable or write instructions to one variable. No instruction performs both a read and a write and no instruction accesses more than one variable. Both algorithms mentioned below have been simplified and only show the code relevant for this thesis.

Hjalmarson's algorithm

In Hjalmarson's algorithm, the marker field is defined as in Figure 36. The variable 'wcon' is set to the number of the first task writing the data area of this marker field. The variable 'rcon' is set to the number of the first task reading the data area of this marker field. Tasks are assigned a number circularly from 1 to 255. Value 0 is reserved indicating 'no task'. The algorithm itself is described in Table 5 and Table 6.

```
struct marker {
 unsigned char wcon;
 unsigned char rcon;
};
```

Figure 36 Marker field for Hjalmarson's algorithm

on write		wcon			
		0 (unused)	< own (older)	= own (equal)	> own (younger)
rcon	0 (unused)	write wcon; carry on	wait	carry on	rollback wcon; wait
	< own (older)				
	= own (equal)				
	> own (younger)	rollback rcon; write wcon; carry on	rollback rcon; wait	rollback rcon; carry on	rollback rcon and wcon; wait

Table 5 Data collision detection algorithm at writing

on read		wcon			
		0 (unused)	< own (older)	= own (equal)	> own (younger)
	0 (unused)	write rcon; carry on			rollback wcon; wait
rcon	< own (older)	wait			
	= own (equal)	carry on			
	> own (younger)	rollback rcon; wait (optimisation to 'carry on' possible)			rollback rcon and wcon; wait

Table 6 Data collision detection algorithm at reading

Tikekar's algorithm

In Tikekar's algorithm, marker fields are defined differently (Figure 37). The first variable is used to gain atomic access when modifying the marker field or its data area. The variable 'oldestTaskNr' carries the number of the oldest task accessing this data area. Task numbering is done as in Hjalmarson's algorithm. The displacement bits indicate the other (younger) tasks that have accessed the field. E.g. if 'oldestTaskNr' is 34 and 'displacementBits' contains the pattern 00 01 00 00 10 01, then this indicates that task 34 had read, 35 has written and 38 has read this data area. The size of 'displacementBits' is linear to the level of speculation; 6 in our example. The algorithm is listed in Figure 38.

```
struct marker {
 int atomicAccess;
 unsigned char oldestTaskNr;
 unsigned char displacementBits[f(LEVEL_OF_SPECULATION)];
}
```
Figure 37 Marker field for Tikekar's algorithm

```
DATACOLLISIONDETECTIONTIKEKAR
1  get atomic access
2  get action from Table 7 or Table 8
3  if action is 'proceed normal' then
4      perform required read or write operation
5      if the existing oldestTaskNr is older than own task number then
6          set the relevant displacement bits
7      else
8          left shift displacement bits with the difference between oldestTaskNr and own task number
9          overwrite oldestTaskNr with own task number
10         set relevant displacement bits
11     release atomic access
12 else
13     if writing then
14         rollback all younger tasks that have read or written
15     else { reading }
16         rollback all younger tasks that have written
17     start from beginning of algorithm
```
Figure 38 Tikekar's data collision detection algorithm

If only one of the actions in the tables is met then the corresponding action should be taken. If more than one condition is met then rollback actions take priority.

on write	older than self	younger than self	empty
if others read bit set	proceed normal	initiate rollback of other	proceed normal
if others write bit set	proceed normal	initiate rollback of other	proceed normal

Table 7 Action table for writing

on read	older than self	younger than self	empty
if others read bit set	proceed normal	proceed normal	proceed normal
if others write bit set	proceed normal	initiate rollback of other	proceed normal

Table 8 Action table for reading

Appendix B Algorithms and proof for optimal scheduling

In this appendix, the algorithms and their proof of correctness is given for calculating optimal scheduling. The first section is on Hjalmarson's algorithm, the second on Tikekar's. See "Appendix A Collision detection algorithms" for a listing of these data collision detection algorithms.

Optimal scheduling for Hjalmarson's algorithm

```
OPTIMALHJALMARSON
input:   A trace $T$ of tasks, sorted in time of original sequential execution. See section 4.2.1, "What traffic data
         is needed" for the definition of the contents of a trace.
output:  For every task $v$ in $T$, $v_{commit\ time}$, the earliest commit time for $v$.
{ init }
for each task $v$ in T do
    if the corresponding message is external then
        $v_{\text{end time}} = 0$
        insert $v$ in FIFO queue $q$
    $p_{commit\ time} = 0$
{ main }
until $q$ empty do
    get task $v$ from $q$
    for each instructions $i$ in task $v$ do
        $v_{\text{end time}}$ += $i_{\text{execution time}}$
        if $i$ reads a variable $x$ then
            $w_{commit\ time}$ = latest commit time of all older tasks reading $x$
            if $v_{end\ time} < w_{\text{commit time}}$ then
                $v_{end\ time} = w_{commit\ time}$
            $w_{write\ time}$ = latest write time of all older tasks writing $x$
            if $v_{end\ time} < w_{\text{write time}}$ then
                $v_{end\ time} = w_{write\ time}$
        if $i$ writes a variable $x$ then
            $w_{commit\ time}$ = latest commit time of all older tasks writing $x$
            if $v_{end\ time} < w_{\text{commit time}}$ then
                $v_{end\ time} = w_{oommit\ time}$
            $w_{read\ time}$ = latest read time of all older tasks reading $x$
            if $v_{end\ time} < w_{\text{read time}}$ then
                $v_{end\ time} = w_{read\ time}$
    if $v_{\text{end time}} < p_{\text{commit time}}$ then
        $v_{\text{end time}} = p_{\text{commit time}}$
    $v_{commit\ time} = v_{end\ time}$
    $p_{commit\ time} = v_{commit\ time}$
    for each follower task $f$ of $v$, from youngest to oldest, do
        $f_{\text{end time}} = v_{\text{commit time}}$
        insert $f$ in $q$
```

Figure 39 Hjalmarson's data collision detection algorithm

theorem 1
For every task v in T, $v_{commit\ time}$ is the earliest commit time for v.

proof
To prove this theorem, we will first define and prove a number of lemmas.

lemma 1
For every task v extracted from q, v is 'as young as possible' and the calculated $v_{commit\ time}$ is the earliest commit time for v.

proof
Using induction to $|q|$, the number of tasks extracted from q:
1. basis ($|q|=0$). Trivial.
2. hypothesis ($|q|=n$). For every task v in the n tasks extracted from q, v is 'as young as possible' and the calculated $v_{commit\ time}$ is the earliest commit time for v.
3. step ($|q|=n+1$). Follows immediately from lemma 3 and lemma 6.

lemma 2
Task $n+1$ is inserted in q as early as possible.

proof
Task $n+1$ is either externally generated or internally generated:
- If task $n+1$ is externally generated, then it is inserted on initialisation in lines 1-5. Task $n+1$ cannot be inserted earlier, since this would violate the sequential order of T.
- If task $n+1$ is internally generated, then it is inserted in lines 30-32. According to the Multi-X model, an internally generated task is inserted on commit of the initiating task. Inserting task $n+1$ earlier would violate this rule.

lemma 3
Task $n+1$ is 'as young as possible'.

proof
The age of a task is the task number received on insertion in the FIFO queue (Multi-X model). According to lemma 2, task $n+1$ is inserted as early as possible.

lemma 4
On extraction of task $n+1$, the execution start time of $n+1$ is as early as possible.

proof
The execution start time on extraction is defined by $n+1_{end\ time}$. Task $n+1$ is either externally generated or internally generated:
- If task $n+1$ is externally generated, then $n+1_{end\ time}$ is set to zero at initialisation.
- If task $n+1$ is internally generated, then this task cannot start execution before the initiating task has committed. In line 31, $n+1_{end\ time}$ is set to the commit time of the initiating task. According to the induction hypothesis, this time is as early as possible.

lemma 5
Every instruction in $n+1$ ends as early as possible.

proof
Using induction to $|n+1|$, the number of instructions in $n+1$:
1. basis ($|n+1|=0$). Follows immediately from lemma 4.
2. hypothesis ($|n+1|=m$). All instructions up and until m end as early as possible.
3. step ($|n+1|=m+1$):

Instruction m cannot end before it has executed its number of cycles. Adding less cycles in line 11 would violate this rule.

From the algorithm listed on "Appendix A Collision detection algorithms", the fact that $n+1$ is 'as young as possible' (lemma 3) and the fact that all older tasks are 'as young as possible' (induction hypothesis lemma 1), six rules can be defined:
1. Task $n+1$ cannot write a variable that is written by older tasks before all these older tasks have committed.
2. Task $n+1$ cannot write a variable that is read by older tasks before all these older tasks have read the variable for the last time.
3. Task $n+1$ can immediately write a variable that is neither read nor written by older tasks.

4. Task *n+1* cannot read a variable that is read by older tasks before all these older tasks have committed.
5. Task *n+1* cannot read a variable that is written by older tasks before all these older tasks have written the variable for the last time.
6. Task *n+1* can immediately read a variable that is neither read nor written by older tasks.

Proof for the six rules listed above:

1. Writing the variable earlier will lead to either a rollback (if an older task writes the variable later) or a wait (if an older tasks has written the variable). In the latter case, the release time for that variable is the commit time of the older task (Multi-X model).
2. Writing the variable earlier will lead to a rollback (if an older task writes the variable later).
3. Trivial.
4. Reading the variable earlier will lead to either a rollback (if an older task reads the variable later) or a wait (if an older tasks has read the variable). In the latter case, the release time for that variable is the commit time of the older task (Multi-X model).
5. Reading the variable earlier will lead to a rollback (if an older task reads the variable later).
6. Trivial.

From these six rules follow two rules for optimal scheduling:

1. The earliest time for task *n+1* to write a variable is the latest time in the combined set of the commit times of all older tasks writing the variable and the latest read times of all tasks reading the variable. If the set is empty, the variable can be written immediately.
2. The earliest time for task *n+1* to read a variable is the latest time in the combined set of the commit times of all older tasks reading the variable and the latest write times of all tasks writing the variable. If the set is empty, the variable can be read immediately.

From these rules, lines 12-25 follow immediately. Similar to lemma 3-lemma 5 and using lemma 2, it can be proved that every instruction in all older ends as early as possible. Adding less cycles in lines 12-25 would violate the scheduling rules mentioned above, so every instruction reading or writing a variable ends as early as possible.

lemma 6
The calculated $n+1_{commit\ time}$ is the earliest commit time for *n+1*.

proof
Follows immediately from lemma 5 and the in-order commit rule in the Multi-X model (lines 26-29 and line 6).

lemma 7
Every task in T is extracted once from q.

proof
All external tasks are inserted in q exactly once, at initialisation. All internal tasks are inserted in q exactly once, in lines 30-32. All inserted tasks are extracted exactly once, in line 8-9.

proof theorem 1
The proof for theorem 1 follows immediately from lemma 1 and lemma 7.

Optimal scheduling for Tikakar's algorithm

OPTIMALTIKEKAR
input: A trace T of tasks, sorted in time of original sequential execution. See section 4.2.1, "What traffic data is needed" for the definition of the contents of a trace.
output: For every task v in T, $v_{commit\ time}$, the earliest commit time for v.

```
{ init }
for each task $v$ in T do
if the corresponding message is external then
    $v_{end\ time} = 0$
    insert $v$ in FIFO queue $q$
$p_{commit\ time} = 0$
{ main }
until $q$ empty do
    get task $v$ from $q$
    w = task 'level of speculation' older than v
    if $v_{end\ time} < w_{commit\ time}$ then
        $v_{end\ time} = w_{commit\ time}$
    for each instructions $i$ in task $v$ do
        $v_{end\ time}$ += $i_{execution\ time}$
        if $i$ reads a variable $x$ then
            $t_{last\ write}$ = FINDLASTWRITE($v$, $x$)
            if $t_{last\ write} > v_{end\ time}$ then
                $v_{end\ time} = t_{last\ write}$
        if $i$ writes a variable $x$ then
            $t_{last\ reference}$ = FINDLASTREADORWRITE($v$, $x$)
            if $t_{last\ reference} > v_{end\ time}$ then
                $v_{end\ time} = t_{last\ reference}$
    if $v_{end\ time} < p_{commit\ time}$ then
        $v_{end\ time} = p_{commit\ time}$
    $v_{commit\ time} = v_{end\ time}$
    $p_{commit\ time} = v_{commit\ time}$
    for each follower task $f$ of $v$, from youngest to oldest, do
        $f_{end\ time} = v_{commit\ time}$
        insert $f$ in $q$
```

FINDLASTWRITE
input: A task v, a variable x.
output: Latest time a task older than v wrote variable x.

```
$t_{last\ write\ found} = 0$
for each task $w$ older than $v$, from youngest to oldest do
    if task $w$ writes variable $x$ then
        find last instruction $i$ writing variable $x$
        $t_{following}$ = sum of execution times for all instructions in $w$ succeeding $i$
        $t_{last\ write} = w_{end\ time} - t_{following}$
            if $t_{last\ write} > t_{last\ write\ found}$ then
                $t_{last\ write\ found} = t_{last\ write}$
return $t_{last\ write\ found}$
```

FINDLASTREADORWRITE
input: A task v, a variable x.
output: Latest time a task older than v read or wrote variable x.
Implementation similar to FINDLASTWRITE.

theorem 2
For every task v in T, $v_{commit\ time}$ is the earliest commit time for v.

proof without lines 10-12
To prove this theorem, we will first define and prove a number of lemmas. This proof excludes lines 10-12, i.e. it proofs optimal scheduling without any restrictions on the level of speculation.

lemma 8
For every task *v* extracted from *q*, *v* is 'as young as possible' and the calculated $v_{commit\ time}$ is the earliest commit time for *v*.

proof
Using induction to $|q|$, the number of tasks extracted from *q*:
1. basis ($|q|$=0). Trivial.
2. hypothesis ($|q|$=n). For every task v in the n tasks extracted from *q*, *v* is 'as young as possible' and the calculated $v_{commit\ time}$ is the earliest commit time for *v*.
3. step ($|q|$=n+1). Follows immediately from lemma 8and lemma 13.

lemma 9
Task *n+1* is inserted in *q* as early as possible.

proof
Task *n+1* is either externally generated or internally generated:
- If task *n+1* is externally generated, then it is inserted on initialisation in lines 1-5. Task *n+1* cannot be inserted earlier, since this would violate the sequential order of T.
- If task *n+1* is internally generated, then it is inserted in lines 27-29. According to the Multi-X model, an internally generated task is inserted on commit of the initiating task. Inserting task *n+1* earlier would violate this rule.

lemma 10
Task *n+1* is 'as young as possible'.

proof
The age of a task is the task number received on insertion in the FIFO queue (Multi-X model). According to lemma 9, task *n+1* is inserted as early as possible.

lemma 11
On extraction of task *n+1*, task *n+1* cannot start before $n{+}1_{end\ time}$.

proof
The execution start time on extraction is defined by $n{+}1_{end\ time}$. Task *n+1* is either externally generated or internally generated:
- If task *n+1* is externally generated, then $n{+}1_{end\ time}$ is set to zero at initialisation.
- If task *n+1* is internally generated, then this task cannot start execution before the initiating task has committed. In line 28, $n{+}1_{end\ time}$ is set to the commit time of the initiating task. According to the induction hypothesis, this time is as early as possible.

lemma 12
Every instruction in *n+1* cannot end earlier than calculated.

proof
Using induction to $|n+1|$, the number of instructions in *n+1*:
1. basis ($|n+1|$=0). Follows immediately from lemma 11.
2. hypothesis ($|n+1|$=m). All instructions up and until *m* cannot end earlier than calculated.
3. step ($|n+1|$=m+1):

Instruction *m* cannot end earlier than before it has executed its number of cycles. Adding less cycles in line 14 would violate this rule.

From the algorithm listed on "Appendix A Collision detection algorithms", the fact that *n+1* is 'as young as possible' (lemma 10) and the fact that all older tasks are 'as young as possible' (induction hypothesis lemma 8), four rules can be defined.

1. Task *n+1* cannot write a variable that is read or written by older tasks before all these older tasks have read or written the variable for the last time.
2. Task *n+1* can immediately write a variable otherwise.
3. Task *n+1* cannot read a variable that is written by older tasks before all these older tasks have written the variable for the last time.
4. Task *n+1* can immediately read a variable otherwise.

Proof for the four rules listed above:

1. Writing the variable earlier will lead to a rollback.
2. Trivial.
3. Writing the variable earlier will lead to a rollback.
4. Trivial.

From these rules follow two rules for optimal scheduling:

1. The earliest time for task *n+1* to write a variable is the latest time an older task reads or writes the variable. If no such older task exists, the variable can be written immediately.
2. The earliest time for task *n+1* to read a variable is the latest time an older writes the variable. If no such older task exists, the variable can be read immediately.

From the induction hypothesis of lemma 8 and the two scheduling rules above, lines 15-22 follow immediately. Adding less cycles in lines 15-22 would violate the scheduling rules mentioned above, so every instruction reading or writing a variable cannot end earlier.

lemma 13

The calculated $n+1_{commit\ time}$ is the earliest commit time for *n+1*.

proof

Follows immediately from lemma 12 and the in-order commit rule in the Multi-X model (lines 23-26 and line 6).

lemma 14

Every task in T is extracted once from q.

proof

All external tasks are inserted in q exactly once, at initialisation. All internal tasks are inserted in q exactly once, in lines 27-29. All inserted tasks are extracted exactly once, in line 8-9.

proof theorem 2 without lines 10-12

The proof for theorem 1 follows immediately from lemma 8 and lemma 14

proof theorem 2

Follows immediately from the proof of theorem 2 without lines 10-12 and the Multi-X prototype implementation on level of speculation (i.e. the number of uncommitted tasks never exceeds the level of speculation).

Appendix C Data on simulation results

On this appendix, all data of figures presented in chapter 5 is listed. Section two to four present data on first-come-first-serve, function-based and source-based scheduling respectively. In the first section some graphs are included supporting the check if simulator input data is long enough in time, see also section 5.1 and section 5.2.

Speedup in time

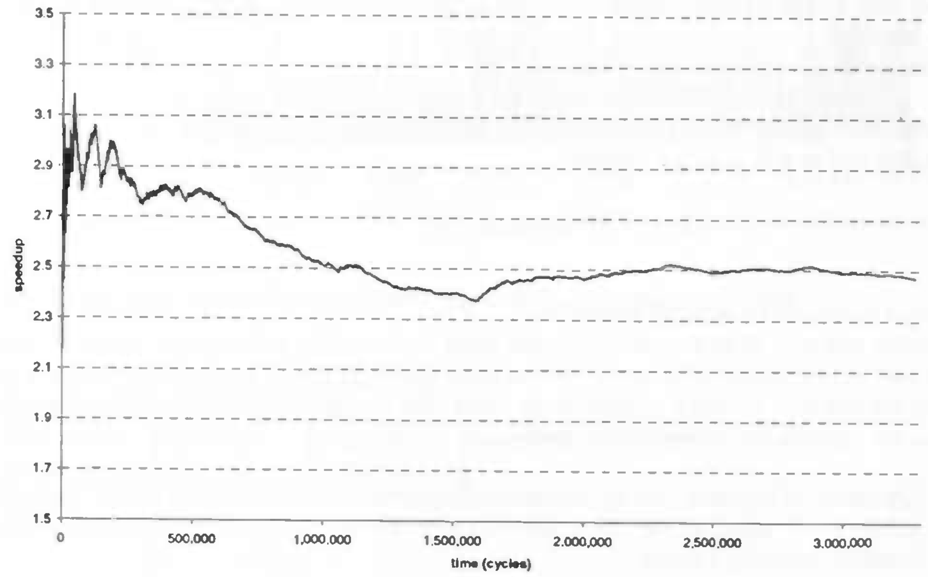

Figure 40 Speedup in time for optimal scheduling with Hjalmarson's algorithm

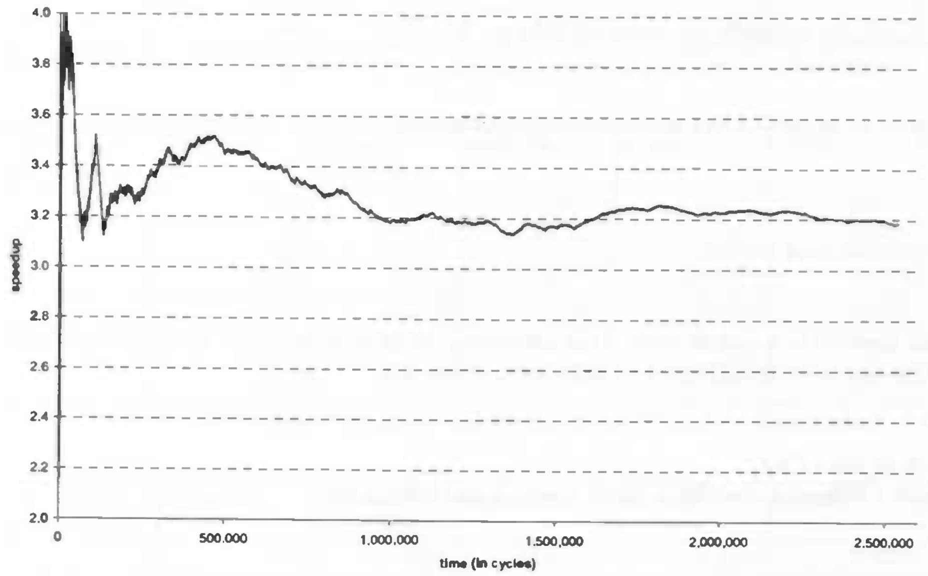

Figure 41 Speedup in time for optimal scheduling with Tikekar's algorithm, speculation depth 12

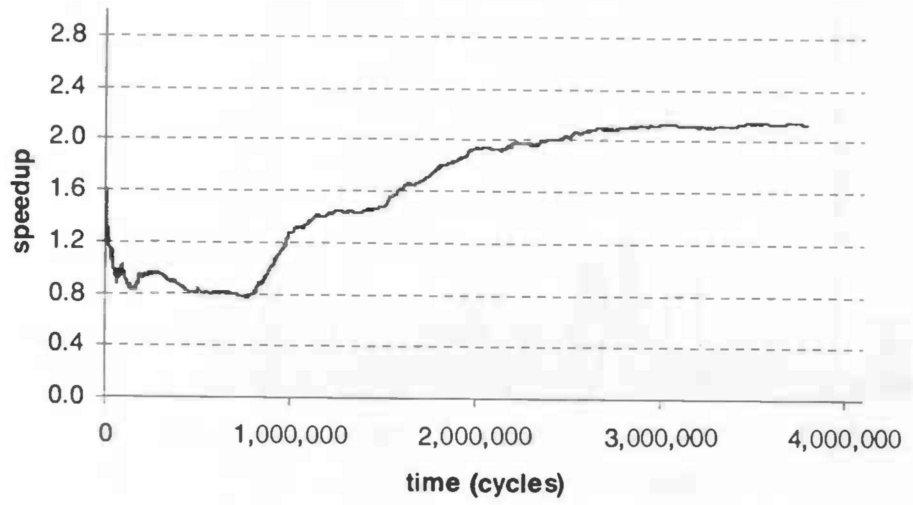

Figure 42 Speedup in time for first-come-first-serve scheduling, Hjalmarson's algorithm, 4 processors

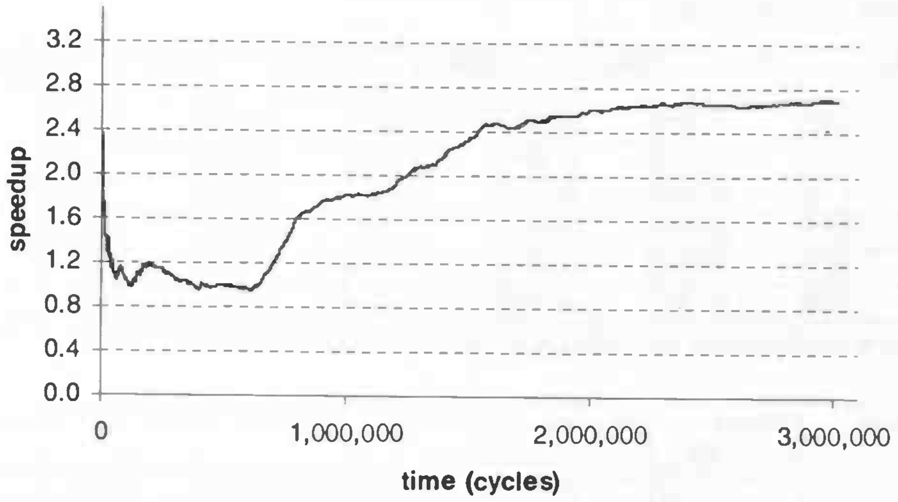

Figure 43 Speedup in time for first-come-first-serve scheduling, Tikekar's algorithm, speculation depth 12, 4 processors

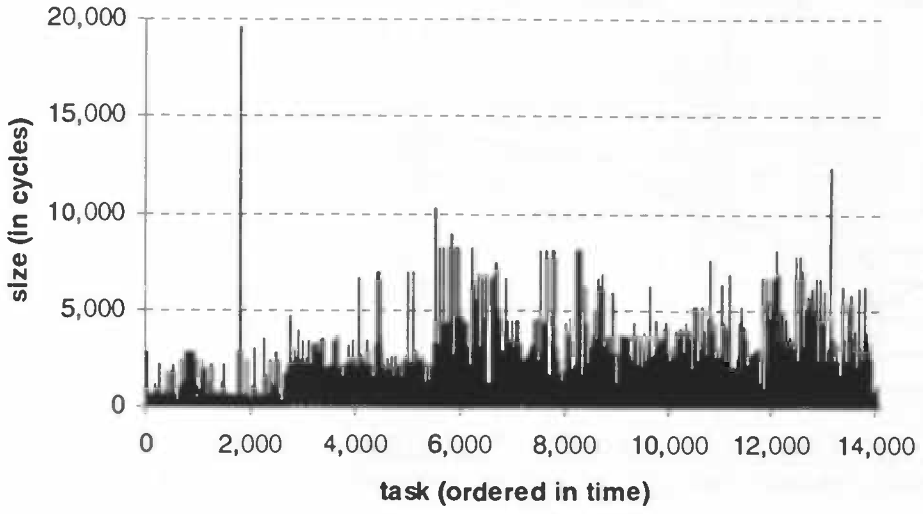

Figure 44 Task, ordered at position in input data, in number of cycles

First-come-first-serve scheduling

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.00%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.60	4.15%	80.23%	1.50%	17.97%	0.29%	100.00%
3	1.94	20.78%	64.73%	3.04%	31.77%	0.46%	100.00%
4	2.13	45.44%	53.14%	5.27%	41.02%	0.57%	100.00%
5	2.21	78.51%	44.19%	7.81%	47.31%	0.69%	100.00%
6	2.28	116.75%	38.06%	10.50%	50.65%	0.80%	100.00%

Table 9 Results first-come-first-serve scheduling with Hjalmarson's algorithm

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.0%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.84	20.4%	91.98%	6.74%	0.00%	1.27%	100.00%
3	2.39	61.3%	79.74%	12.46%	0.00%	7.80%	100.00%
4	2.68	111.8%	66.92%	16.23%	0.00%	16.85%	100.00%
5	2.79	184.4%	55.84%	19.15%	0.00%	25.01%	100.00%
6	2.80	270.9%	46.67%	21.09%	0.00%	32.24%	100.00%

Table 10 Results first-come-first-serve scheduling with Tikekar's algorithm at a speculation depth of 12

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.0%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.86	19.8%	93.20%	6.45%	0.00%	0.36%	100.00%
3	2.52	61.8%	84.00%	14.32%	0.00%	1.68%	100.00%
4	2.99	113.6%	74.63%	21.04%	0.00%	4.33%	100.00%
5	3.26	202.8%	65.29%	28.36%	0.00%	6.35%	100.00%
6	3.44	316.9%	57.39%	33.80%	0.00%	8.81%	100.00%

Table 11 Results first-come-first-serve scheduling with Tikekar's algorithm at a speculation depth of 28

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.0%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.86	19.5%	93.24%	6.42%	0.00%	0.34%	100.00%
3	2.54	62.8%	84.52%	14.37%	0.00%	1.11%	100.00%
4	2.97	129.4%	74.27%	23.38%	0.00%	2.35%	100.00%
5	3.35	220.5%	66.97%	30.11%	0.00%	2.92%	100.00%
6	3.61	310.9%	60.19%	35.95%	0.00%	3.86%	100.00%

Table 12 Results first-come-first-serve scheduling with Tikekar's algorithm at a speculation depth of 44

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.0%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.86	19.5%	93.24%	6.42%	0.00%	0.34%	100.00%
3	2.52	63.5%	84.15%	15.16%	0.00%	0.69%	100.00%
4	3.00	130.6%	74.91%	23.65%	0.00%	1.44%	100.00%
5	3.36	214.1%	67.16%	30.65%	0.00%	2.18%	100.00%
6	3.58	331.6%	59.68%	37.25%	0.00%	3.07%	100.00%

Table 13 Results first-come-first-serve scheduling with Tikekar's algorithm at a speculation depth of 60

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.0%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.86	19.5%	93.24%	6.42%	0.00%	0.34%	100.00%
3	2.53	64.9%	84.45%	14.97%	0.00%	0.58%	100.00%
4	2.98	135.6%	74.38%	24.38%	0.00%	1.24%	100.00%
5	3.29	232.5%	65.86%	32.50%	0.00%	1.64%	100.00%
6	3.59	347.2%	59.88%	37.91%	0.00%	2.21%	100.00%

Table 14 Results first-come-first-serve scheduling with Tikekar's algorithm at a speculation depth of 76

Function-based scheduling

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.00%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.74	0.00%	87.07%	0.00%	0.00%	12.93%	100.00%
3	1.96	0.00%	65.36%	0.00%	0.00%	34.64%	100.00%
4	2.01	0.00%	50.35%	0.00%	0.00%	49.65%	100.00%
5	2.02	0.00%	40.32%	0.00%	0.00%	59.68%	100.00%
6	2.05	0.00%	34.09%	0.00%	0.00%	65.91%	100.00%

Table 15 Results function-based scheduling with Tikekar's algorithm, speculation depth 12, perfect prediction

allowed collision frequency	speedup	rollback frequency	executed	wasted	waited	idle	total
no	1.741	0.0%	87.07%	0.00%	0.00%	12.93%	100.00%
0	1.761	0.1%	88.03%	0.01%	0.00%	11.96%	100.00%
0.5	1.765	1.5%	88.26%	0.34%	0.00%	11.39%	100.00%
1.0	1.781	2.7%	89.07%	0.57%	0.00%	10.36%	100.00%
2.0	1.797	5.3%	89.85%	1.54%	0.00%	8.60%	100.00%
3.0	1.821	8.5%	91.05%	1.89%	0.00%	7.06%	100.00%
4.0	1.812	10.9%	90.62%	2.83%	0.00%	6.54%	100.00%
5.0	1.823	9.2%	91.15%	2.46%	0.00%	6.39%	100.00%
10.0	1.820	13.0%	90.98%	3.01%	0.00%	6.01%	100.00%
15.0	1.822	15.2%	91.08%	3.40%	0.00%	5.52%	100.00%
25.0	1.848	19.4%	92.41%	6.04%	0.00%	1.55%	100.00%

Table 16 Results function-based scheduling with collision frequency, Tikekar's algorithm, speculation depth 12, perfect prediction

allowed relative collision frequency	speedup	rollback frequency	executed	wasted	waited	idle	total
no	1.741	0.0%	87.07%	0.00%	0.00%	12.93%	100.00%
0	1.761	0.1%	88.03%	0.01%	0.00%	11.96%	100.00%
5	1.795	5.2%	89.73%	1.49%	0.00%	8.78%	100.00%
10	1.822	10.4%	91.10%	2.53%	0.00%	6.37%	100.00%
15	1.856	14.9%	92.80%	5.56%	0.00%	1.64%	100.00%
20	1.859	15.0%	92.94%	5.44%	0.00%	1.63%	100.00%
25	1.858	15.5%	92.88%	5.59%	0.00%	1.52%	100.00%
30	1.858	15.5%	92.88%	5.59%	0.00%	1.52%	100.00%
35	1.848	19.4%	92.41%	6.04%	0.00%	1.55%	100.00%

Table 17 Results function-based scheduling with relative collision frequency, Tikekar's algorithm, speculation depth 12, perfect prediction

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.00%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.77	3.28%	88.57%	2.89%	0.00%	8.54%	100.00%
3	2.04	7.21%	68.09%	4.32%	0.00%	27.59%	100.00%
4	2.14	9.56%	53.54%	3.89%	0.00%	42.57%	100.00%
5	2.18	10.86%	43.66%	3.58%	0.00%	52.76%	100.00%
6	2.16	11.94%	36.05%	3.27%	0.00%	60.69%	100.00%

Table 18 Results function-based scheduling with Tikekar's algorithm, speculation depth 12, perfect prediction, 3 ahead

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.00%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.79	3.65%	89.30%	3.18%	0.00%	7.52%	100.00%
3	2.08	9.36%	69.40%	4.87%	0.00%	25.73%	100.00%
4	2.17	11.72%	54.15%	4.92%	0.00%	40.93%	100.00%
5	2.20	13.41%	44.08%	4.54%	0.00%	51.38%	100.00%
6	2.20	14.72%	36.63%	4.01%	0.00%	59.36%	100.00%

Table 19 Results function-based scheduling with Tikekar's algorithm, speculation depth 12, perfect prediction, 2 ahead

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.00%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.80	4.89%	90.17%	4.29%	0.00%	5.54%	100.00%
3	2.19	13.20%	73.12%	7.35%	0.00%	19.53%	100.00%
4	2.32	22.37%	58.11%	8.79%	0.00%	33.10%	100.00%
5	2.36	30.19%	47.19%	8.93%	0.00%	43.88%	100.00%
6	2.40	33.02%	39.92%	8.13%	0.00%	51.95%	100.00%

Table 20 Results function-based scheduling with Tikekar's algorithm, speculation depth 12, perfect prediction, 1 ahead

allowed relative collision frequency	speedup	rollback frequency	executed	wasted	waited	idle	total
no	1.803	4.9%	90.17%	4.29%	0.00%	5.54%	100.00%
0	1.807	5.0%	90.33%	4.27%	0.00%	5.40%	100.00%
5	1.832	8.8%	91.61%	4.86%	0.00%	3.53%	100.00%
10	1.849	14.2%	92.45%	5.43%	0.00%	2.12%	100.00%
15	1.856	14.9%	92.82%	5.55%	0.00%	1.63%	100.00%
20	1.859	15.0%	92.96%	5.42%	0.00%	1.62%	100.00%
25	1.858	15.5%	92.88%	5.59%	0.00%	1.52%	100.00%
30	1.858	15.5%	92.88%	5.59%	0.00%	1.52%	100.00%
35	1.848	19.4%	92.41%	6.04%	0.00%	1.55%	100.00%

Table 21 Results function-based scheduling with relative collision frequency, Tikekar's algorithm, speculation depth 12, perfect prediction 1 function ahead

Source-based scheduling

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.00%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.49	21.59%	74.58%	8.08%	0.00%	17.34%	100.00%
3	1.82	42.56%	60.58%	10.57%	0.00%	28.85%	100.00%
4	2.01	56.97%	50.25%	9.95%	0.00%	39.80%	100.00%
5	2.10	77.55%	41.94%	10.10%	0.00%	47.96%	100.00%
6	2.18	90.36%	36.26%	8.82%	0.00%	54.91%	100.00%

Table 22 Results source-based scheduling with Tikekar's algorithm, speculation depth 12

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.00%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.54	20.69%	77.13%	8.77%	0.00%	14.09%	100.00%
3	1.85	43.30%	61.60%	11.26%	0.00%	27.14%	100.00%
4	2.07	62.57%	51.65%	11.50%	0.00%	36.84%	100.00%
5	2.16	79.77%	43.25%	10.23%	0.00%	46.52%	100.00%
6	2.28	84.36%	38.00%	9.30%	0.00%	52.70%	100.00%

Table 23 Results source-based scheduling with Tikekar's algorithm, speculation depth 12, improved load balancing

number of processors	speedup	rollback frequency	executed	wasted	waited	idle	total
1	1.00	0.00%	100.00%	0.00%	0.00%	0.00%	100.00%
2	1.58	36.52%	78.89%	16.04%	0.00%	5.07%	100.00%
3	1.94	77.53%	64.58%	23.66%	0.00%	11.76%	100.00%
4	2.22	119.44%	55.51%	27.07%	0.00%	17.42%	100.00%
5	2.32	179.05%	46.34%	29.49%	0.00%	24.17%	100.00%
6	2.52	228.00%	41.94%	30.10%	0.00%	27.96%	100.00%

Table 24 Results source-based scheduling with Tikekar's algorithm, speculation depth 28