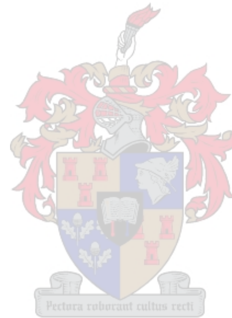
The Transputer Virtual Memory System.

by
Sias Mostert
June 7, 1990



Thesis presented in partial fulfillment of the requirements for
the
Master of Engineering at the University of Stellenbosch.

STUDY LEADER: Mr P.J. Bakkes

Stellenbosch University http://scholar.sun.ac.za

DECLARATION

I hereby declare that the work for this thesis was done and written by myself and that it has not been submitted to any other university for the purpose of obtaining a degree.

June 7, 1990

Stellenbosch University http://scholar.sun.ac.za

ACKNOWLEDGEMENTS

I would like to thank the following for their support and encouragement during the my work on the thesis:

- Mr P.J. Bakkes, my study leader, for his guidance, advice and patience.
- Prof J.J. du Plessis for his guidance and advice.
- My wife Belinda for her support.
- The heavenly Father without whom this would not be possible.

Stellenbosch University http://scholar.sun.ac.za

Abstract

The transputer virtual memory system provide, for the transputer without memory management primitives, a viable virtual memory system. This report evaluates the architecture and its parameters. The basic software is also implemented and described. The disk subsystem with software and hardware is also evaluated in a single disk environment.

It is shown that the unique features of the TVM system has advantages and disadvantages when compared to conventional virtual memory systems. One of the advantages is that a conventional operating system with memory protection can now also be implemented on the transputer. The main conclusion is that this is a performance effective implementation of a virtual memory system with unique features that should be exploited further.

OPSOMMING

Die transputer virtuele geheue verskaf, vir 'n verwerker sonder virtuele geheue ondersteuning, 'n doeltreffende virtuele geheue stelsel. Die verslag evalueer die argitektuur en sy parameters. Die skyfsubstelsel met programmatuur en apparatuur word ook geevalueer in 'n enkel skyfkoppelvlak omgewing.

Daar word bewys dat die unieke eienskappe van die TVG (transputer virtuele geheue) voor- en nadele besit wanneer dit vergelyk word met konvensionele virtuele geheue stelsels. Een van die voordele is dat 'n konvensionele bedryfstelsel met geheue beskerming nou op 'n transputer ge-implementeer kan word. Die hoofnadeel agv die spesifieke argitektuur gee slegs 'n 15% degradering in werkverrigting. Dit word egter slegs oor 'n sekere datagrootte ervaar en kom tipies nie ter sprake wanneer daar massiewe programme geloop word nie.

Stellenbosch University http://scholar.sun.ac.za

Contents

1 Introduction **1**

I Relevant literature **3**

2 Introduction to Literature Study **4**

2.1 Traditional workloads 4

3 Virtual memory hardware **5**

3.1 Basic hardware 5

3.2 Hardware support 6

3.2.1 Distributed and Slave memory 6

3.2.2 Hardware support for measurements 7

3.2.3 Addressing mechanisms 7

3.3 Determining page size 8

4 Memory management **10**

4.1 Basic principles 10

4.2 Measures for evaluation 11

4.3 Page replacement strategies 11

4.3.1 Terminology 12

4.3.2 The optimal page replacement strategy 12

4.3.3 Algorithm classification according to amount of data used 12

4.3.4 Algorithm classification according to inclusion property 13

Stellenbosch University http://scholar.sun.ac.za

4.3.5 Known page replacement algorithms 13
4.4 Page prediction strategies 17
4.4.1 Demand prepaging 17
4.4.2 Sequential prepaging 20
4.4.3 Determining optimal buffer sizes. 20
4.5 Other methods of improving performance 21

II Transputer virtual memory **23**

5 TVM Hardware **24**
5.1 Basic architecture mechanisms 25
5.1.1 Two processor system 25
5.1.2 Memory hierarchy 26
5.1.3 Hardware in support of TVM 27
5.2 TVM system architecture 28
5.3 Optimal parameters for TVM 28
5.3.1 The benchmarks 30
5.3.2 The measure for comparison 35
5.3.3 The active cache size 35
5.3.4 The non-active cache size 39
5.3.5 The window size 42
5.3.6 Page size 42
5.4 Performance implications of TVM architecture 45
5.5 Detail HW design 47

6 TVM Software **48**
6.1 Program specification 48
6.2 Program design 49
6.2.1 Process harness 49

Stellenbosch University http://scholar.sun.ac.za

6.2.2 Modular construction 49

6.2.3 Data structures 49

6.2.4 Program flow 53

6.3 Program evaluation 54

6.3.1 Execution times 55

6.3.2 Replacement algorithms 55

6.4 Future development 57

6.4.1 Stack algorithms 57

6.4.2 Prediction 57

6.4.3 Disk organization 58

6.5 Other ways to improve performance 59

III Secondary memory system 61

7 Hardware 62

7.1 Overview of solutions 62

7.1.1 XC to diskinterfaces 62

7.1.2 Disk subsystem architecture 63

7.1.3 Diskinterface architecture 64

7.2 Diskinterface design 66

7.3 Performance evaluation 66

8 Software 68

8.1 Program specification 68

8.2 Program design 68

8.3 Performance evaluation 70

IV In conclusion 71

9 Effect of VM on program execution 72

Stellenbosch University http://scholar.sun.ac.za

10 Conclusions 74

V Appendices 78

A Transputer virtual memory hardware 79

B TVM registers 80

C TVM PAL equations 81

D TVM users manual 82

E M212 disk interface 83

F SCSI disk interface 84

G Interfacing transputers 85

Stellenbosch University http://scholar.sun.ac.za

List of Figures

3.1 Lower bound on access times 8

4.1 A typical lifetime function. 15
4.2 Life time knee nd space time minimum. 16
4.3 The effect of prepaging on matrix multiplication. 19
4.4 Obtaining access frequencies from a success function 21

5.1 Simplified memory hierarchy diagram. 26
5.2 Block diagram of TVM system. 29
5.3 The memory map for mat100 with VAL parameters and VAR parameters. . . . 31
5.4 Memory maps for increasing number of simultaneous accessed data structures. . 33
5.5 Memory map for the NORM benchmark. 34
5.6 Norm program : execution time against increasing active cache size. 36
5.7 Matrix program : various parameters against active cache size. 37
5.8 Increasing the number of simultaneous accessed data structures. 38
5.9 Execution times for matrices of different dimensions against active cache size. . . 39
5.10 The improvement over one NAC in execution time for bigger NAC's. 40
5.11 Execution time versus NAC size for optimum amount of ac pages. 41
5.12 The execution times for the various matrix dimensions against window size. . . . 43
5.13 Page fault handling time vs page size. 44
5.14 The effect on execution time when the page size is variable. 45
5.15 The % of time wasted vs the dimensional size for matrix. 46

6.1 Process diagram for TVM. 50

Stellenbosch University http://scholar.sun.ac.za

6.2 Module hierarchy for TVM. 51

6.3 The inter relationship between the tables. 53

6.4 Main algorithm on MMU. 54

6.5 The execution time for matrix 150 under FIFO and RANDOM replacement algorithms. 56

6.6 The execution time for matrix 200 under FIFO and RANDOM replacement algorithms. 56

6.7 Disk access times for different page sizes. 59

6.8 Execution times for matrix algorithm and its transpose algorithm. 60

7.1 Evaluation of disk channel architecture. 65

7.2 TVM scsi disk interface block diagram. 67

9.1 Percentage performance of virtual memory system when compared to execution in real memory. 72

9.2 Percentage performance of virtual memory system with very small window when compared to execution in real memory. 73

Stellenbosch University http://scholar.sun.ac.za

Chapter 1

Introduction

The transputer is a very fast microprocessor (10 MIPS) with an onboard scheduler and communication processors. A basic design aim of the manufacturer was one processor per user. Thus no multiuser support in the form of memory management and protection have been included in the transputer. This includes a lack of virtual memory supporting hardware.

Many applications need the fast processor in addition to more memory than can be provided in the form of fast read/write memory. The transputer virtual memory system provide in this need.

Fundamental differences between the TVM system and conventional virtual memory systems are:

1. The virtual memory provided must be totally transparent to the user. Specifically no operating primitives must be necessary to use the virtual memory.
2. A dedicated disk storage subsystem will be available for the paged system.
3. The workload designed for is large scientific programs and NOT a multiprogramming environment.

This report will investigate the performance of the designed architecture and will show that this is a performance efficient virtual memory system when king size jobs are run on it.

The TVM system will be investigated with different size jobs. For small jobs with a data requirement less than 8 mega byte this system provides directly supported read/write memory, thus no performance degradation will result. The medium size jobs from 8 Mbyte to 13 Mbyte expose the systems' weak spots. King size jobs, that is with memory requirements greater than 13 Mbyte will run as efficiently as on any other virtual memory system with the same memory size parameters.

The unique features of the TVM system can be exploited to further enhance the performance efficiency of the TVM system. One performance influence of particular interest is the multiple disk channels which is connected to the memory management unit. The other unique feature

Stellenbosch University http://scholar.sun.ac.za

open for exploitation is the memory management unit when it is not busy servicing a page fault.

The report begins with a literature survey of virtual memory systems. The first reported virtual memory system was reported in 1961! The report continues with the architecture description and evaluation. This is followed by a description of the current software implemented and an evaluation thereof. The disk subsystem hardware and software follows with a basic evaluation of its performance. The report is concluded with the final conclusions and recommendations.

Stellenbosch University http://scholar.sun.ac.za

Part I

Relevant literature

Stellenbosch University http://scholar.sun.ac.za

Chapter 2

Introduction to Literature Study

The first machine to use virtual memory was the ATLAS computer from Manchester. Since then virtual memory has been investigated and many results have been published. Thus more than twenty nine years have past since the introduction of virtual memory. It is then expected that many advances would have been made and that the theory of virtual memory would be relatively well known. In the rest of this chapter existing virtual memory systems will be considered to extract from them the lessons learnt so far in the design of virtual memory systems.

Any virtual memory system can be decomposed into the hardware architecture and the management software running on it. Both these subjects will now be considered independently.

2.1 Traditional workloads

Virtual memory was invented with the purpose of giving programmers the much needed unlimited supply of memory. The principles of virtual memory were soon utilized in multiprogrammed and time-sharing computers. It provided a mechanism for holding in main memory many user processes much larger than the available memory space. Thus many of the early studies considered evaluation of a virtual memory system within a multiprogramming environment of utmost importance [Denning 70].

The TVM system was designed to provide a powerful workstation for a single user. The main purpose is to provide one programmer with a powerful processor with 'unlimited' memory. Thus under most circumstances the evaluation of virtual memory under multiprogrammed workloads is of little use. However the transputer supports parallel execution threads which again is a multiprogrammed workload. This document does not look into the performance evaluation of the TVM under multiprogrammed workloads, but for the right user base this evaluation could be very applicable.

Stellenbosch University http://scholar.sun.ac.za

Chapter 3

Virtual memory hardware

3.1 Basic hardware

The basis of virtual memory is to disassociate the address referenced by a process from the address space available in primary storage. It follows that some kind of mapping mechanism must exist to facilitate this transformation. This transformation mechanism must not slow down the memory references of a process.

Due to efficiency considerations the main memory is divided into equally sized sections called pages. These pages are then the smallest unit managed by the virtual memory system. The mapping mechanism then consists of a address transformation unit taking an address which can be considered as a composite address consisting of a pair (p,d). Where the first n bits form p the page address and the last m bits form d the distance into the currently addressed page.

Due to the obvious limitation on the size of main memory, a mechanism must exist to stop the executing process when an address referenced does not exist in the main storage. This condition should generate a page fault event which interrupts the executing processor which will then execute the page management software. On completion of making the addressed page available in the main store, the processor then restarts the interrupted process by re-executing the interrupted instruction.

The above mechanism implies a few assumptions:

1. Only one processor is used for both process execution and page fault handling.
2. The processor must have restartable instructions referring to memory.

It will be shown that the transputer virtual memory system functions while not satisfying any of the above assumptions.

The main store not containing all the address space of a process, must be backed up by a second level of storage. In all cases known this second level of storage is a moving arm disk.

Stellenbosch University http://scholar.sun.ac.za

The main memory of the processor then is just a 'managed buffer' for the processors' address space which is mapped onto the disk. This leads one to belief that virtual memory systems are very slow because the memory speed is the speed of the disk. Fortunately programs exhibit certain behaviour patterns which makes virtual memory in many cases not much slower than real memory.

3.2 Hardware support

Described in the previous section is the basic hardware required to support virtual memory. There are however a few hardware implementable options to be considered. Three specific areas are considered in [Denning 70]:

- Slave memory vs distributed memory.
- Hardware support for measurements to improve the management software.
- Addressing mechanisms.

3.2.1 Distributed and Slave memory

Both slave memory and distributed memory consists of memory hierarchies. The difference is in accessing the different levels of the hierarchy. In a slave memory system access to the memory level closest to the processor does not result in any delay. But any access to a data item in a level further away from the closest level results in a page fault event and the data item must be loaded into the closest level before program execution can continue. One example of slave memory is a cache.

Distributed memory although also consisting of different levels of memory does no generate a page fault event for accessing a data item in any level. Thus the processor can access any data item in any level. The cost of accessing data items further away from the processor lays in the longer time taken by the address translation mechanisms for levels further away from the closest level.

None of the virtual memory systems found by the author in literature employs the distributed memory hierarchy. This can be attributed to the difficulty in implementing such a mechanism. Nearly all modern cache systems does however employ the distributed memory hierarchy. This could be ascribed to the fundamental difference between cache and virtual memory systems. Cache systems provide faster access to addressed items than main memory allows and the cache is normally a small subset of the main memory. The probability of finding the item in main memory when it is not in the cache is 100times in the order of 2 to 5 times as slow as the cache. which means that there is a small time penalty paid for accessing the main memory.

Virtual memory systems though provide the user with a much larger address space that can be provided by fast random access memory. Access times to disk are orders of magnitude longer

Stellenbosch University http://scholar.sun.ac.za

than to main memory. Thus any hardware translation mechanism addressing the disk is a gross under utilization of the speed achievable in hardware.

However if a virtual memory system consisted of more than the two levels of memory associated with main memory and disk memory then, accesses to the intermediate levels might warrant a hardware address translation unit as found in a distributed memory environment.

3.2.2 Hardware support for measurements

For efficient management of the virtual memory space information is needed. Many of the management policies known today require information with regard to page accesses which cannot be measured with software. The following measures can easily be measured with the minimal of hardware support.

1. Setting a modify bit.
2. Setting a referenced bit.
3. Setting an unused bit.
4. Incrementing counters for each access to a page.

The significance of these measures can be deduced from the section on virtual memory management.

3.2.3 Addressing mechanisms

These mechanisms refer to the basic address translation mechanisms. The basic criteria for any such mechanism is the minimal extra delay introduced due to the mapping process. The first level of memory is accessed with the normal memory address cycle. If a multilevel memory hierarchy exist then the access to lower levels must introduce a minimal delay.

The only hardware mechanism satisfying the no delay requirement of the memory level closest to the processor is associative mapping. This is however a costly mechanism in terms of the amount of hardware required. The largest mechanism known to the author is a 512 page unit used with a modern cache controller.

There is however a result reported in [Deitel 83] which indicates that a small associative mapping mechanism of 16 pages combined with a slower mechanism for all the other cases, results in a performance of 90 % of a full associative mapping mechanism for all the pages in main memory.

The lower levels in a multi level memory hierarchy virtual memory use a slower mapping mechanism, some of which is described in [Deitel 83]. These slower mechanisms could be implemented in two ways. One mechanism would be a real time virtual address translator

Stellenbosch University http://scholar.sun.ac.za

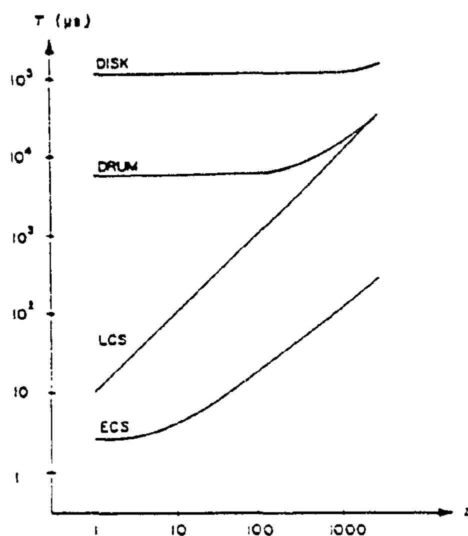


Figure 3.1: Lower bound on access times

with a delay time of two memory accesses. On subsequent accesses to the page a page fault is generated and the page number moved into the associative mapping. The alternative is that on the first reference to a page a page fault is generated and the page number moved into the associative mapping mechanism.

It will be shown later that because of the locality property of programs executing, there is little need for the first slower mapping mechanism described because the probability that another location in the same page will be referenced is very high.

3.3 Determining page size

The optimal page size for a virtual memory system depends on hardware and software considerations. In this section only the hardware aspects which influence the page size are considered.

The lookup and transfer time for a page from the secondary storage to primary storage and memory fragmentation are the two hardware parameters influenced by the page size. Both lookup and transfer times will further be referred to as the access time of a device.

In the article by [Denning 70] a relation between the access time for different memory technologies is given. The relations of importance to us is the moving arm disk and an intermediate memory level. From fig 3.1 it can be seen that for page sizes up to 1000 words the access time to a disk is constant due to the dominance of the seek and rotational delay. The technologies indicated on the graph are completely outdated, but the principle of a slower memory than main memory in a hierarchy is very relevant. If such a level existed corresponding to the ECS graph, then smaller page sizes would perform better.

Memory fragmentation consists of two types in a paged memory system [Denning 70] viz. internal fragmentation and table fragmentation. The previous of the two refers to a page not

Stellenbosch University http://scholar.sun.ac.za

completely filled with items to be referenced by the processor on requesting that page to be loaded. This phenomena indicates smaller pages are to be preferred for efficient memory use.

Table fragmentation refers to the amount of table space needed to manage the pages in a virtual memory system. The more pages the bigger the tables and the less memory available for buffer'ng pages. This phenomena indicates that larger pages will be better because the tables will then be smaller.

From the above discussion it is clear that the use of a disk for secondary storage implies that for page sizes from one word to around 1000 words the access time is the same. Thus there is an advantage to use a page size of 1000 words. Fragmentation though has another influence. Internal fragmentation indicates the smaller the page the better. While table fragmentation advocates bigger page sizes. The page size decision from a hardware point of view is thus a compromise which must take into account the current technology.

Stellenbosch University http://scholar.sun.ac.za

Chapter 4

Memory management

The memory management system for a virtual memory system also have a first order effect on the performance of a virtual memory system. The main function of a management system is to make sure that the next location the main process wants to access will be available.

Some basic strategies investigated and reported on in literature will be discussed. This is followed by some measures of performance used by researchers. From the basic strategy chosen some page replacement algorithms follow. Another way for the management software to increase performance is by prediction. This concept have also been investigated and will be reported on.

4.1 Basic principles

According to [Deitel 83] there are two main strategies ie. fetch strategies and replacement strategies. The first has to do with when to bring a page into real memory and the second with when to remove a page from virtual memory.

Fetch strategies can be divided into demand fetching and prediction fetching. Demand fetching only fetches a page when addressed by the executing program. While prediction fetching tries to predict the next page which will be requested and then loads that page. Demand paging is the dominant method employed today. Prefetching can however improve the execution time of a program by 10 to 20 % according to [Smith 78]. While [Trivedi 76] only states that there is an improvement but he does not quantify it.

Replacement strategies are numerous and even an optimal algorithm has been suggested. These various algorithms are first classified according to existing criteria and then there relative performance compared.

Other methods to increase the execution speed of virtual memory programs is to restructure the program to fit the underlying hardware better. A few of these methods have also been investigated and will be reported on.

Stellenbosch University http://scholar.sun.ac.za

Hence will be considered the most important measures for comparing the different strategies. Then different replacement strategies will be discussed. Lastly the question of when to fetch a page will be addressed.

4.2 Measures for evaluation

[Trivedi 76], in a paper on the effects of prepaging for an array workload, gives three measures for comparing virtual memory systems. Each of these measures are the most appropriate in optimizing some parameter of a virtual memory system. The parameter to optimize will be discussed in the context of each of the measures.

1. Number of 'page pulls'. This is the number of transfers from secondary storage to main memory. This parameter is of importance when the channel traffic is to be minimized.
2. Number of page faults. This measure is of importance when the CPU utilization is to be maximized.
3. The space time product. This parameter is defined as

$$c(t_1, t_2) = \int_{t_1}^{t_2} m(t)dt$$

   where $m(t)$ indicates the occupation of $m(t)$ pages in memory at time t. This measure is of importance when maximizing memory utilization.

The author is of opinion that not any one measure should dominate, but that at least a combination if the first two measures should be used. The most important measure should be the program execution speed. This is however very restricted in that it only accounts for one type of program. Most computer systems are however used for specific purposes and the system architecture should be optimized for these.

Another important measure is described in [Mattson 70] with regard to the evaluation of storage hierarchies. The *success function* is defined as

> The relative frequency of successes as a function of capacity is given by the success function. Where a success is defined as an access into a level of a multilevel c and the item searched for was there.

This success function will be shown to be useful in determining the various buffer sizes of the various levels of memory given the trace of a program.

4.3 Page replacement strategies

The page replacement strategy of a virtual memory system can truly be called the crux of any virtual memory system. Its behaviour determines to a great extent the performance of a virtual

Stellenbosch University http://scholar.sun.ac.za

memory system where no king size jobs are run [Yoshizawa 88]. Page replacement comes into play when the main processor has requested a page and the question is which page must be replaced. There is an optimal algorithm [Denning 70] which is discussed first followed by many approximations realizable in computer systems.

4.3.1 Terminology

The abbreviations used further have the following meaning:

LRU Least recently used page replacement algorithm.

FIFO First in First out page replacement algorithm.

NUR Not used recently page replacement algorithm.

LFU Least frequently used page replacement algorithm.

WS Working set memory management strategy.

4.3.2 The optimal page replacement strategy

The optimal page replacement algorithm will replace the page not to be used for the furthest time into the future [Denning 70]. This algorithm is not realizable since it requires advance information about the behaviour of the program to run. Any practical algorithm then approximates the optimal algorithm.

4.3.3 Algorithm classification according to amount of data used

Belady [Belady 66] carried out a study on page replacement algorithms and classified them according to the amount of information used to make a decision. This classification is:

- The replacement algorithm is not based on any information about memory usage. The algorithms falling under this category are random replacement and FIFO replacement.
- Pages are classified according to the history of their most recent use in memory. Algorithms falling in this category are LRU and NUR.
- Pages are classified according to their presence and absence from main memory. All pages are considered. These types of algorithms never developed very far.

From this classification it can be deduced how well certain types of algorithms will fare. Also can be deduced what type of information is necessary for the efficient management of virtual memory.

Stellenbosch University http://scholar.sun.ac.za

4.3.4 Algorithm classification according to inclusion property

Replacement algorithms whose traces obey the inclusion property [Mattson 70] are called stack algorithms because for any program trace the stack can be efficiently computed and from the stack the success function can be deduced. Refer to [Mattson 70] for more details.

A stack algorithm will always lead to less page faults in a larger buffer space, while a non stack algorithm would not. What is of importance however is that the FIFO replacement strategy is not a stack algorithm while LRU, LFU, NUR and even the random replacement policies are.

4.3.5 Known page replacement algorithms

The optimal page replacement algorithm has been described in a previous section. Some of the algorithms which follow tries to approximate the optimal algorithm while others go out from certain assumptions about program behaviour.

There is one parameter which influences the basic outlook on replacement algorithms. The real memory window mapping onto virtual memory could either be of a fixed size or a variable size. In the former case it is easy to manage the free pages, ie. it is wise to fill as many pages as possible with pages referenced by the program to increase the likelihood of another 'hit' or 'success'. In the latter case the decision of exactly how many pages must be allocated to each job at a specific time is a hard choice. It is in the latter case where not only the replacement but also the freing mechanism are important.

The TVM system has only a fixed size window onto virtual memory. This leads one to believe that these more sophisticated techniques are not relevant, but in the case of freing pages to make place for a sudden surge of page faults or for prepaged pages these techniques could lead to improved performance. The two techniques which falls under the last category are the working set strategy and the page fault frequency algorithms.

Random page replacement

This strategy assumes that the pages referenced in a program follows a random pattern normally uniformly distributed. With the assumption made, there is little reason to replace pages other than random. One reassuring fact about a random replacement algorithm is that [Mattson 70] has shown that a random replacement algorithm for a specific buffer size c can be represented by an equivalent stack algorithm. This implies that for bigger buffer sizes the random replacement algorithm will indeed perform better.

[Belady 66] has shown that for king size jobs (program memory requirement far greater than real memory) the random replacement algorithm did not fare much worse than the other more optimal algorithms tested by him. This can be ascribed to the fact that king size jobs flush the real memory every so often that there is little value in keeping extra information about the used pages.

Stellenbosch University http://scholar.sun.ac.za

FIFO page replacement

Two schools of thought could arrive at this algorithm. On the one hand, one could argue that to approximate the optimal page replacement algorithm a possible solution would be to replace the page the longest in memory. This is just the tail of a FIFO queue. The other way to arrive at the FIFO replacement algorithm is to argue that it is just a special case of the random algorithm involving much less computation.

While both arguments is true, it has been proved that the FIFO replacement algorithm is not a stack algorithm. This implies that for bigger buffer sizes c there is not necessarily an improvement of the virtual memory system with a FIFO replacement algorithm.

LRU-Least recently used

This algorithm is most likely the best performer of the demand replacement algorithms. The page to be replaced is the one who has been referenced the longest time back. This approximates the page that will not be used for the longest time in the future very well because of the working set behaviour of program execution.

LFU-Least Frequently Used

An approximation of LRU. Measure how intensively a page has been referenced. Those pages with the least number of references within the last time frame are replaced. This algorithm has a grave possibility of removing a page just moved into memory.

NUR-Not Used Recently

An approximation of LRU with little overhead [Deitel 83]. Pages are divided into four groups according to how they were referenced. Pages not referenced at all forms the group to be replaced first. Pages modified but not referenced form the second group to be replaced. Pages referenced but not modified forms the third group to be replaced. And then if none of the previous type of pages are in the virtual memory to be replaced then pages modified and referenced are selected for replacement.

This scheme ensures that the last group does not contain all the pages by periodically resetting the referenced bits of all the pages. This ensures that those pages actively referenced remains in the last two groups and these pages are then selected last for replacement.

Unfortunately the writer [Deitel 83] does not compare the performance of the techniques described by him. This could be a subject of further investigation.

Stellenbosch University http://scholar.sun.ac.za

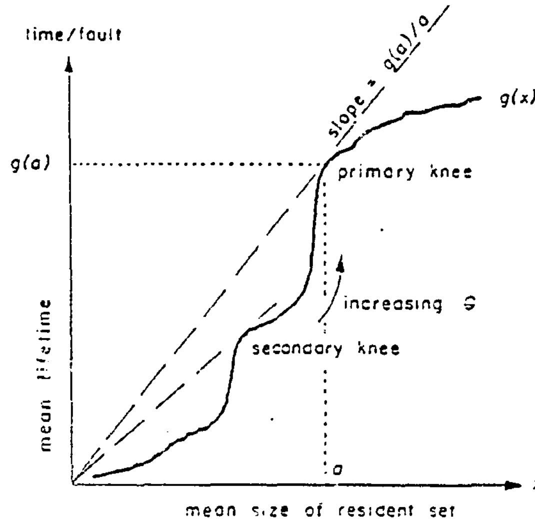


Figure 4.1: A typical lifetime function.

The working set principle

It has been shown by many authors of whom [Denning 72], [Denning 80] is the most notable that it models the memory demand of a running program very well. The principle states that the working set is the set of pages used during the most recent sample interval. Algorithms for exercising working set control is given in [Denning 80]. The same author has shown that it is a policy which performs very well because all the stack algorithms are just special cases of this policy [Denning 80].

This policy also provide the mechanism to remove from memory all those pages not to be referenced in the next time interval, thereby creating open page slots for either prediction or a sudden surge of new pages requested.

The key parameter in the working set algorithm is the inter-reference interval [Gupta 78]. This is the time between successive references to the same page. The idea is that pages not referenced for a time T will not be referenced soon and could therefor be removed from the working set. In the case where memory is bounded and filled with pages all referenced within time shorter than T, this algorithm is exactly the same as the LRU algorithm. It increases performance of the system in those cases where there are more than one page not referenced within time T. These pages can then be removed from memory to make space for either the surge or predicted pages as mentioned before. It is in this last context that it might be of relevance to the TVM system.

The previous paragraph describes the principle of working sets, but it is in general not easily implemented. Another way to determine the optimum working set size is through use of the lifetime function. This function $g(x)$ gives the mean number of references between page faults when the resident set size is x. These functions have been shown to exhibit a knee, fig 4.1. This knee corresponds to the 'optimum' working set size of the program whose lifetime function

Stellenbosch University http://scholar.sun.ac.za

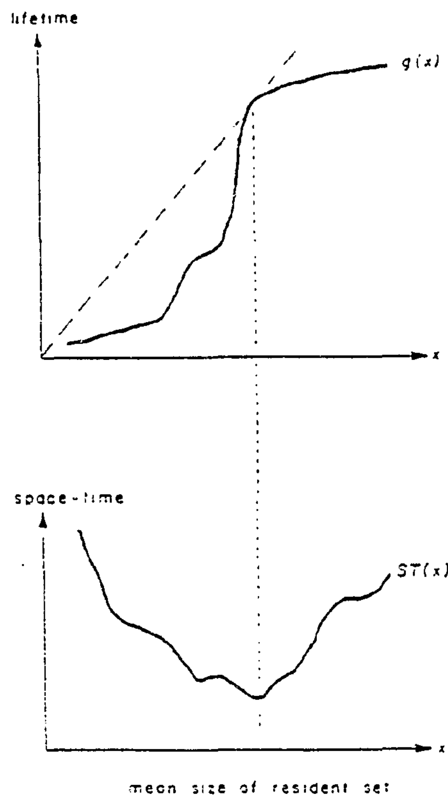


Figure 4.2: Life time knee nd space time minimum.

has been measured. In [Denning 80] he goes on to show that of all the criteria to maintain the smallest working set size, holding the working set size near to the corresponding knee size is the most robust.

Recall that the parameter which is a measure of the efficient use of memory is the space-time product. This measure has also been shown to be within 2 % of its minimum on the knee of the life-time function. See fig 4.2 for an example.

The working set policy as well as the next policy called page fault frequency are both so called *local* policies. This applies in a multiprogramming environment where the choice is between managing all the pages at the same time or managing each processes pages as a unit. The last mentioned option is referred to as local. It has been shown that for a high level of multiprogramming with small jobs the local policies perform better [Denning 80]. But for king size jobs the two policies exhibit performance of equal magnitude [Oliver 74].

Page fault frequency algorithm

The page fault frequency algorithm was introduced by Chu and Opderbeck which was to be an easily implemented alternative to WS [Denning 80]. It relies only on hardware usage bits and

Stellenbosch University http://scholar.sun.ac.za

an interval timer and is invoked only on page faults. This makes this policy easy to implement on most hardware bases.

For the page fault frequency algorithm the locality set of pages is estimated by observing the page fault rate. If the fault rate is greater than P, the allocation of pages is increased. If the fault rate is lower than P the allocation of pages is decreased. The fault rate is indirectly measured by considering the interfault interval. If this interval is less than $1/P$ then at the time of a page fault an extra page is allocated. If this interval is more than $1/P$ then the allocation is decreased by paging out all those pages not referenced within this interval.

The above algorithm described in [Gupta 78] also reverts to a LRU in a bounded memory buffer where all the pages have been allocated. There is still performance to be gained even in a fixed size buffer with this mechanism as mentioned in the previous section.

The writer [Gupta 78] goes on to investigate the sensitivity of the working set algorithm and the page fault frequency algorithm and concludes that the working set algorithm maintains a better representation of the working set over a much wider class of processes executing in virtual memory.

4.4 Page prediction strategies

4.4.1 Demand prepaging

Prepaging has been the subject of research from the very start of virtual memory systems. It has however not been widespread implemented. This can be attributed to the following [Trivedi 76]:

1. Difficult to implement.
2. If probability of wrong prediction is high, page faults may increase.
3. It may increase 'page pulls' significantly.

The same author then goes on to define an optimal demand prepage algorithm which is not realizable, but provides an upper bound on performance attainable with demand prepaging algorithms. This algorithm can shortly be described by:

```
event ? page fault
  scan future reference string
  fetch the first c pages that will be referenced in future
  goto event
```

Stellenbosch University http://scholar.sun.ac.za

It can be clearly seen that for this algorithm to work a complete future reference string must be available to the page fault handler. This could be obtained by running the program once and recording its references, but in general this would not be possible.

The same algorithm is proven not to be a stack algorithm. Stack algorithms has the property that the page fault rate decreases with increasing buffer size. But for this algorithm alone it can be proved that with increasing c the page fault rate is a non-increasing function.

[Trivedi 76] goes on to investigate two approximations to the optimal algorithm. Two bits of information are associated with each page. One called the dead bit which indicates that the page involved can be removed. The other being a prepage bit which indicates that the page should be made available in the main store as soon as possible. One principle adhered to is that no predicted pages can be pulled in if there aren't any dead pages around. This prevents the prepage mechanism to negatively influence the demand mechanism when the prediction mechanism would make the wrong decision. The algorithm in pseudo code then is:

```
event ? page fault
  remove all dead pages from memory (Free dead pages)
  get page demanded
  from list of pages marked as prepaged pull in as many as
                                              there is space
  goto event
```

Where this freing mechanism is utilized the prefix F is added to the replacement policy eg. LRU becomes FLRU.

The same author then compares the performance of this algorithm called FDPLRU, with the optimal algorithm DPMIN and with conventional demand page only replacement algorithms on the specific problem of matrix operations. There are some remarkable improvements over all the matrix operations. The graph for matrix multiplication is given in fig 4.3. The notation is as follow:

n_1 dimension of full matrix

m dimension of sub matrices

$c = m^2$

$\pi(c)$ the number of page faults with c pages available

The page replacement algorithms compared are:

- LRU. Least Recently Used.
- FLRU. Free policy combined with LRU.

Stellenbosch University http://scholar.sun.ac.za

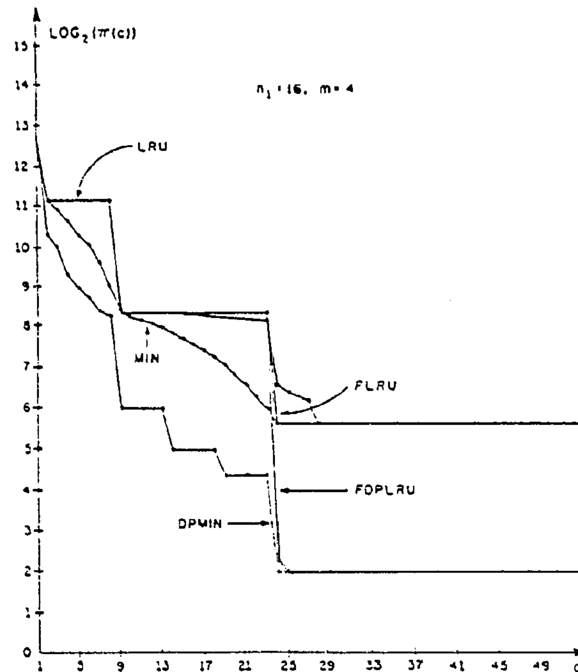


Figure 4.3: The effect of prepaging on matrix multiplication.

- MIN. Optimal page replacement algorithm as defined by [Denning 70] and [Belady 66].
- FDPLRU. Free policy, Demand Prepaging, LRU policy.
- DPMIN. Optimal prepaging algorithm as suggested by [Trivedi 76].

It has been shown that given the following assumptions, there is a significant decline in the number of page faults if a prepaging algorithm is used in a matrix environment. The assumptions are:

- The programmer must know the memory reference pattern of his program with regard to dead pages and prepagable pages.
- There must exist a mechanism to represent this information in the program and to transfer this information timely to the memory manager process.
- The rise in the number of prepage pulls must either overlap other disk operations or not be significantly higher than the case where no prepaging is done. It will be shown later for the TVM system that the number of page pulls has a much more profound effect on the execution time as the number of page faults.
- From the previous point it is clear that the measure for comparison by [Trivedi 76] was taken as page faults alone. It has been discussed under performance measures why this cannot be taken as the only measure.

Stellenbosch University http://scholar.sun.ac.za

The above two assumptions can easily be realized in a library environment where the effort to calculate the parameters are only done once. Further it will be shown that this should be the method of preference for implementing prepaging on the TVM system.

Another important observation from fig 4.3 is that freing dead pages does not lead to a significant decline in the number of page faults. This is contrary to the authors earlier remarks that having empty pages to cope with emergencies might improve performance.

4.4.2 Sequential prepaging

Another author investigated the improvements realizable with sequential prefetching. That is whenever a page is referenced its successor is also pulled in from secondary storage. The conclusions reached by [Smith 78] are:

1. Sequential prefetching is most effective for small page sizes ie. 32 to 64 bytes as performance degrade for bigger page sizes.

2. For such a strategy to work it must be efficiently implemented.

3. A 10% to 25% improvement in the execution speed has been measured.

The above conclusions make one big assumption ie. that the transfer time increases with some linear function for increasing page size. This is only true of a cache system where the source of the pages is the main memory. In a virtual memory system the average cost of transferring a page from disk to main memory is almost constant up to 1000 words. Refer to hardware paragraph on page size influences.

The fact that can be deduced from the above is that in general it would not pay to implement sequential prefetching unless the prefetching disk access time can be overlapped with another demand paged access time. It will be shown that because of the secondary storage organization in the TVM this it is indeed possible to overlap disk operations and sequential prefetching thus becomes a viable alternative to investigate.

4.4.3 Determining optimal buffer sizes.

In designing a virtual memory system with a multilevel memory system it is important to be able to determine the amount of memory that must be available in each level. [Mattson 70] showed that under certain conditions the size of the various levels could be played off against each other with reasonable precision.

The technique he developed takes an address trace and efficiently determines the exact number of references to each level of a c as a function of page size, replacement algorithm, number of levels and the capacity of each level. The conditions under which this analysis can be done are:

1. The replacement algorithm must induce a single priority list for all the levels.

Stellenbosch University http://scholar.sun.ac.za

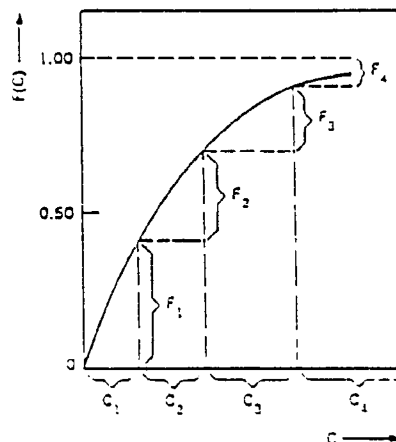


Figure 4.4: Obtaining access frequencies from a success function

2. The replacement algorithms must belong to the class of stack algorithms.

As a graphic example of this technique consider fig 4.4.3 where a success function for a given program is given. From this success function the various buffer capacities can be read off for certain access frequencies to the various levels.

The notation is:

- $F(C)$ is the success function for the program running in unrestricted memory.
- $C_1 \ldots C_4$ are the buffer capacities at the various levels.
- $F_1, ..F_4$ are the relative access frequencies to the corresponding levels of memory.

The implications of this theory are immense. For a given algorithm class which is to run on a virtual memory machine it can be exactly determined what is the optimum configuration to minimize the execution time.

4.5 Other methods of improving performance

Of all the improvements that can be implemented to a virtual memory system there is one the user can make. The user can restructure the program to fit the underlying architecture better. This would in general require a lot of effort from a programmer who is actually using a virtual memory machine as a huge linear addressable memory space. [Hatfield 71] however has shown that improvements in the order of 2 to 1 up to 10 to 1 has been achieved by restructuring a program.

[Hatfield 71] has suggested three ways to improve the performance of a program running on a virtual memory machine. These are:

Stellenbosch University http://scholar.sun.ac.za

1. Minimize the number of page faults be constructing a nearness matrix to determine reordering of program parts that will reduce page faults.

2. Reordering and duplication of code usage.

3. Optimizing compilers.

From applying the first two principles the following conclusions were reached by him.

1. The method applied favoured bigger size pages because the effect of reordering code and data means that items referenced together are grouped in the same or adjacent pages.

2. Improvements an order of magnitude has been found.

The measures suggested by the author are in general difficult to apply. Again they could be implemented in a library where the effort is done once and utilized many times. There are however other guidelines which should be followed which will lead to a significant improvement with very little effort. These will be discussed under running programs efficiently in the TVM system.

Stellenbosch University http://scholar.sun.ac.za

Part II

Transputer virtual memory

Stellenbosch University http://scholar.sun.ac.za

Chapter 5

TVM Hardware

The transputer is a very fast microprocessor (10 MIPS) with an onboard scheduler and communication processors. A basic design aim was one processor per user. Thus no multiuser support in the form of memory management and protection have been included in the transputer. This includes a lack of virtual memory supporting hardware.

The TVM project started out in July 1988 to provide the transputer with viable virtual memory. The design was reported in September 1988 on by one of its inceptors [Bakkes 89]. The first design was completed in December 1988 by [Pina 89] and debugged in January 1989 by [Dorgeloh 89] and the author. The improvements in the first prototype was included in design of prototype 2 and this was again debugged by the author and [Dorgeloh 89]. Design revision three including more memory was completed and debugged. Through use of revision three a major design error was discovered; only one active window on virtual memory was available at any time. By now revision 4 was designed including even more real memory and parity checking. This design was never realized.

Revision 5 was designed by the author and completely debugged by March 1990. Revision 5 included up to 16 simultaneous active windows on the virtual memory, overcoming the main problem of earlier versions.

In all the designs the system parameters were determined by the available technology. Never were there any study to determine the optimum size for any of the system parameters. For example the main memory is now selectable between 4 Megabytes and 8 Megabytes. This technology is currently very cheap. In the earlier designs only 2 Megabytes were available. Regardless of the fact that no in depth study was made to determine optimum parameters, a viable system was realized. The question were how ever, how 'good' is the system?

This chapter goes on to describe the basic architecture. The optimal parameters for TVM will then be determined. The performance implications of the TVM architecture will then be compared to other systems. The final details of the hardware can be found in appendix A.

Stellenbosch University http://scholar.sun.ac.za

5.1 Basic architecture mechanisms

The TVM system consists of three distinct subsystems viz.

1. Two transputers each with its private memory and a method to stop the one transputer in the address phase of an instruction.
2. A system memory hierarchy as seen from the one transputer.
3. An address translation unit for the one transputer to address more than its physical memory.

The first two subsystems contain some unique features not found in other virtual memory machines. The last subsystem is just an implementation of a mechanism inherently found in all virtual memory systems. Each of these will now be described in more detail.

The following notation will be used in the rest of the report.

main transputer Also called the user transputer is the processor the virtual memory is supplied for. Abbreviated : XU.

memory manage unit The second transputer on the TVM system. Also called the controller. Abbreviated : XC.

main memory The physical memory associated with the XU.

cache The window in the XU memory space available to point to virtual memory. It does not include the main memory.

active cache The section of the cache available to point simultaneously into virtual memory.

non-active cache The part of the cache not allocated to the active cache.

window The memory managed by the XC not visible to XU, but faster than the disk.

secondary memory Also referred to as disk memory.

5.1.1 Two processor system

The transputer not having any virtual memory support in hardware such as a restartable instruction, must be immediately stopped and isolated from its memory on detection of a page fault. The only way to achieve this with a transputer is to put it in wait. This leaves no processing power available to process the page fault. Another processor is thus needed.

This second processor chosen was also a transputer. This facilitated easier common access to the same memory and provided a fast processor to handle the page faults in the shortest time. Using a second processor also meant that memory management could carry on while the main

Stellenbosch University http://scholar.sun.ac.za

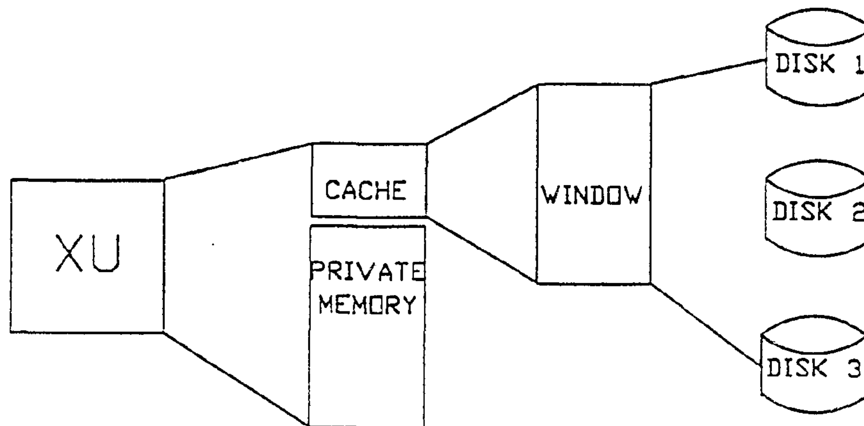


Figure 5.1: Simplified memory hierarchy diagram.

processor was not generating a page fault! This provides for the first level of parallelism in the system which is not found in other systems.

The other significant advantage of the second processor is that a mechanism have been provided to implement a multiprogramming environment on the transputer supporting memory protection! This can be done because the operating system would then reside over the two processors with the second transputer implementing amongst other operations the memory protection function.

5.1.2 Memory hierarchy

The two processor model also brought with it its share of problems. Allowing common access to dynamic memory can be done, but the circuitry becomes complex. This last problem is manifested in the hand back cycle where a valid RAS cycle must be reconstructed by the XC. One solution for th problem was to use static ram for the shared memory. This again implied for the same PCB space less memory could be accommodated. This last decision lead to another deviation from conventional architectures.

To provide the main transputer with 1 Megabyte or more of static RAM was at the time of the project definition too expensive, component wise and PCB space wise. So it was decided to create a much smaller window onto the virtual address space viz. 256kbyte. This smaller window will henceforth be called the cache. The rest of the memory which would normally be found on a processor ie. 1 Megabyte to 8 Megabyte, would still be provided to the main transputer but without any ability to swap pages into and out from this memory. The performance implications of this design decision will be dealt with in section 5.4.

The XC processor doing the memory management also needed some memory of its own to run

Stellenbosch University http://scholar.sun.ac.za

the memory management software. But primarily it needed memory to keep the page tables in. The size of the controller memory is primarily determined by the page size, because this in turn determines how many pages will fit into the virtual memory space provided. The first versions had 2 Megabyte of memory which would not provide for all the space needed when 1 kbyte pages were used in a 2 Gigabyte virtual memory, but if the need arose the page table itself could be kept on disk and only the current section in use could be kept in the XC's main memory. Fig 5.1 illustrates the full memory hierarchy.

The page size is also determined by the transfer speed of the different page sizes from disk. The page size also depend on management parameters. For instance is the program restructured to localize execution? If so, larger pages will give better performance. The optimal page size is thus not a cut and dry case.

The main determiner of page size came from quite a different source. Hardware considerations have played the major role from the start in determining the page size. In the early hardware versions the page size went from 1kbyte to 256kbyte. Where the upper limit is just due to the size of the cache. This smallest page size fitted just into the width of the comparitors and registers used which made it a handy size. These page sizes were to be more or less compatible with existing virtual memory systems where a page size from 512 bytes to 16kbytes have been reported.

During the redesign of the address translation mechanism the minimum page size was changed to 16kbyte. This again was due to hardware considerations as it saved enough PCB space so that four of these mechanism could fit on the same PCB piggyback. The effect of this page size change will be discussed in the section on optimal parameters for the TVM system.

5.1.3 Hardware in support of TVM

The hardware needed to support virtual memory on the main transputer consists of a stopping mechanism which has to cut of the XU control signals to main memory before they take effect and it must put the XU in wait. The first action is the most important and implies that a decision on page fault must be taken in the first section of the address phase.

A hand back mechanism must also exist which allow the XC to hand control of the main memory and the cache back to the XU. This mechanism in the case of dynamic memory must reconstruct a valid RAS cycle before handing back. In general this is difficult and therefor a static ram sharable cache was designed. The address from the XU processor have to be redirected in the active cache to the correct corresponding address. This mechanism is done with bitforcing the corresponding cache page address.

The address translation mechanism to force a XU address onto a page in the cache is the most important parameter in any virtual memory system. In the first three versions of the hardware there were only one such mechanism or also called an active cache page. This meant that a program running in virtual memory with its code in one location and just one data structure at another location would incur a page fault cost for every instruction fetched and every data item referenced. This page fault cost was to be small if the page to be referenced would be in

Stellenbosch University http://scholar.sun.ac.za

the rest of the cache. It was possible to bring this page fault handling time down to $10\mu s$. But this just constituted a $1/10\mu s = 100kHz$ computer! This is clearly not a step forward.

The author then redesigned the mapping mechanism to incorporate up to 16 active cache pages. This meant that 16 disjoint areas in the virtual memory space could be addressed at the same time. This is also in line with existing architectures [Hyde 88] which describes Motorola devices supporting from 16 to 64 active cache pages simultaneously. The question arises is 16 enough? This will be answered in the next section when the optimal amount of active cache pages will be predicted for the TVM system.

The mapping mechanism described above is exactly equivalent to the mechanism described by [Deitel 83]. Thus it will be of interest to determine whether the 90 % performance marked could be reached.

Various other subsystems exist, but the above are the only relevant to a performance analysis of the TVM system.

5.2 TVM system architecture

The complete TVM system architecture for one active register set is given in fig 5.2. The parts in green are duplicated for each active register set. The detailed schematic diagrams, PAL device listings and register legends can be found in appendix A.

The system parameters for the various buffers are:

active cache up to 16 pages in 4 page increments.

window up to 8 Megabyte in 4 Megabyte increments.

page size from 16 kbyte to 256 kbyte.

5.3 Optimal parameters for TVM

This section will evaluate the effectiveness of the various memory hierarchy levels and the page size. An optimum will be determined in every case.

To evaluate a virtual memory system the various workloads to be run on it must be investigated. In the case of the TVM the fact that virtual memory starts only after the main memory of the XU transputer implies that a very small subset of applications need to be considered in evaluating the performance of the TVM. For instance all compilations, editing and programs with small memory requirements will run in the main memory without ever using the virtual memory. This means that *NO* performance penalty is paid using the virtual memory system for smaller problems.

Stellenbosch University http://scholar.sun.ac.za

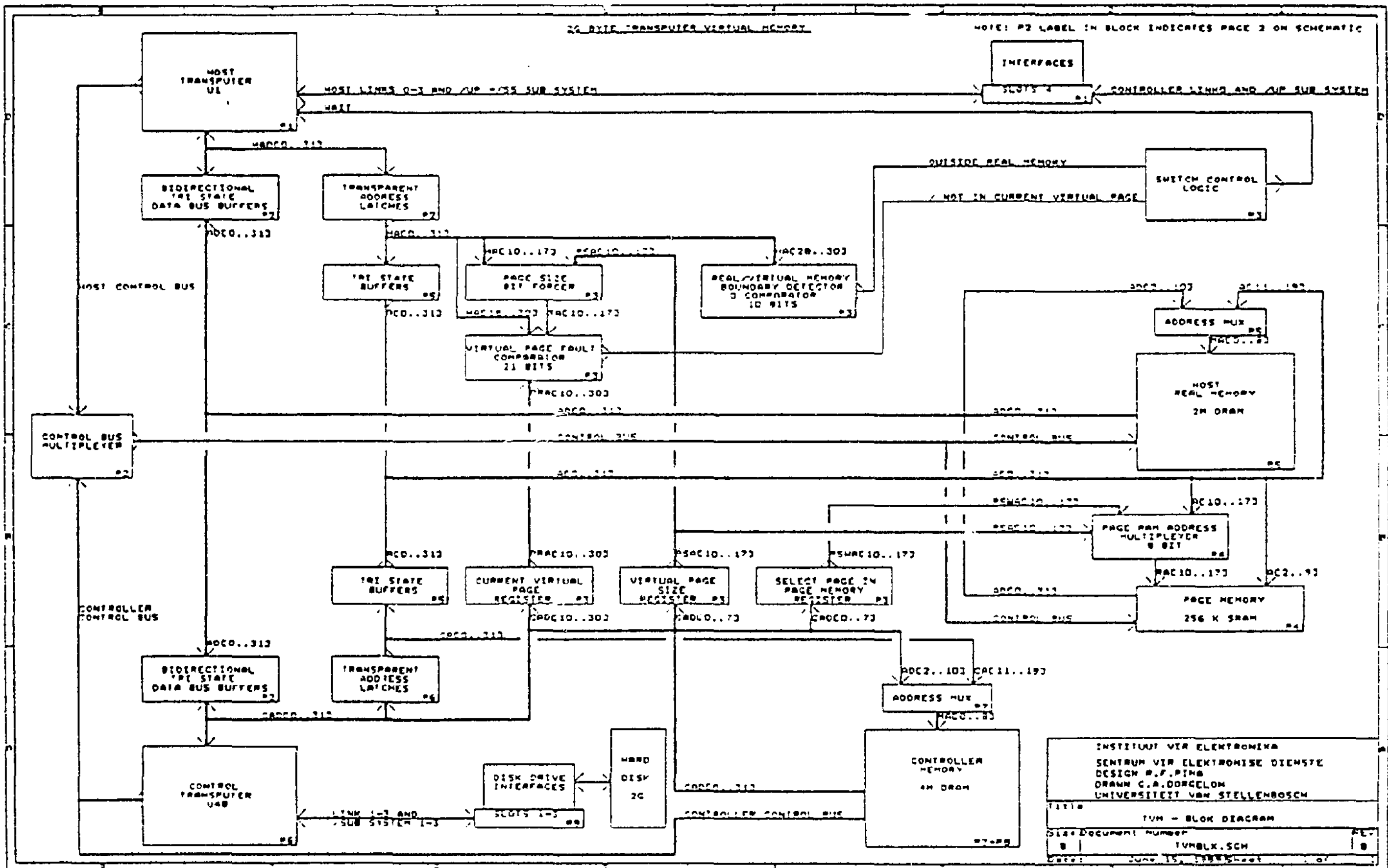


Figure 5.2: Block diagram of TVM system.

Stellenbosch University http://scholar.sun.ac.za

From the previous paragraph it is clear that the only application programs of interest are those with memory requirements greater than the available main memory. In this category are only a few applications. One being the manipulation of large matrices and the other a large data base. So it is necessary to evaluate the TVM system only for these problem classes.

The benchmarks chosen will be run in virtual memory alone. This means that no section of the program will run in the real memory provided with the XU. This imply that the virtual memory alone is evaluated and no second order effects due to the real memory need to be considered.

Notice that nowhere was there any reference to multiuser applications as this does not make sense in a transputer environment with one processor per user. Even though the conventional virtual memory systems were implemented for better CPU and system utilization in a multi-programming environment, the TVM implementation of virtual memory again tries to provide a large address space as intended by virtual memory in the first place.

5.3.1 The benchmarks

The two benchmarks ran to evaluate the TVM system are the following programs for which the memory requirements can easily be adjusted to evaluate certain parameters.

$$A = A * A^{-1}$$

$$B = 1/max(B) * B$$

The matrix benchmark.

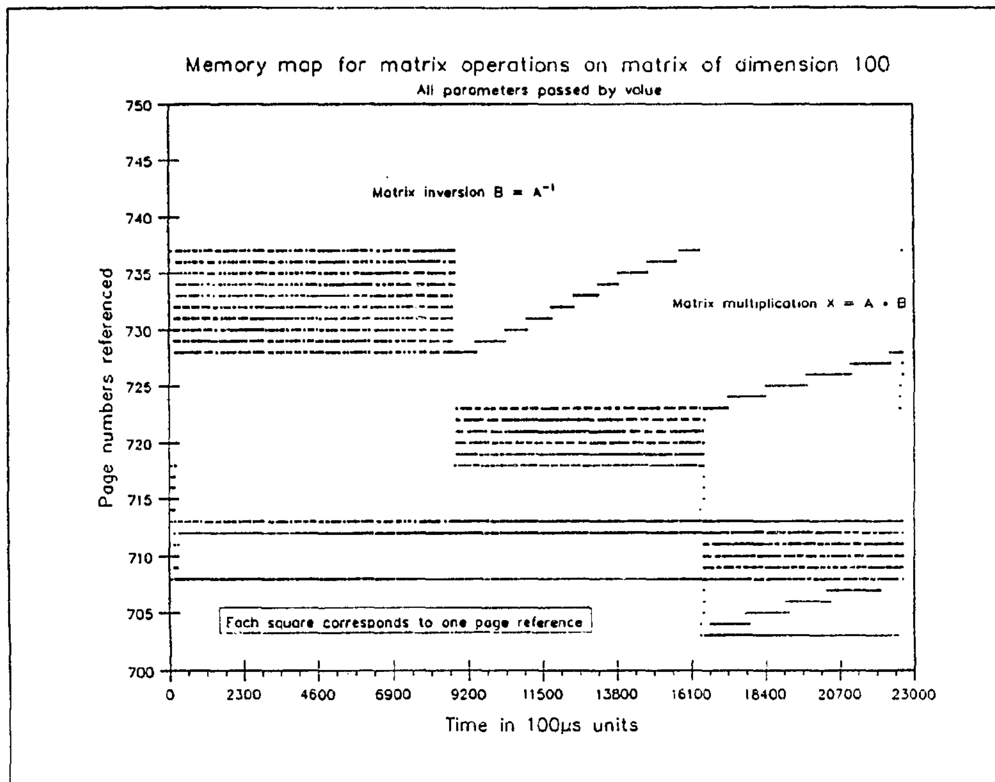
The first benchmark generates the matrix A with random numbers, inverts the matrix and then multiplies it with itself. This results in the identity matrix as answer. This program will be referred to as *MATRIX*. For the dimension of A given as N the memory requirement is the relation $56 * N^2$ for the algorithm with parameters passed by value and $40 * N^2$ for parameters passed by reference.

Of particular interest is the memory map produced when running this benchmark. The memory reference map gives an indication of the locality of the program, which in turn provides the user with feedback on whether to restructure his program and on the optimum number of active cache pages to use.

The memory map for the basic matrix algorithm with pass by value parameters is given in fig 5.3. When comparing this with the memory map for the pass by reference, it is clear that the pass by value implies a copying of the data structures before it is used inside a procedure.

From the memory map one could deduce that the working set size of the matrix inversion is eleven pages. This however corresponds to the complete data structure accessed during the inversion operation. The time interval resolution is to course to make a clear distinction of

Stellenbosch University http://scholar.sun.ac.za



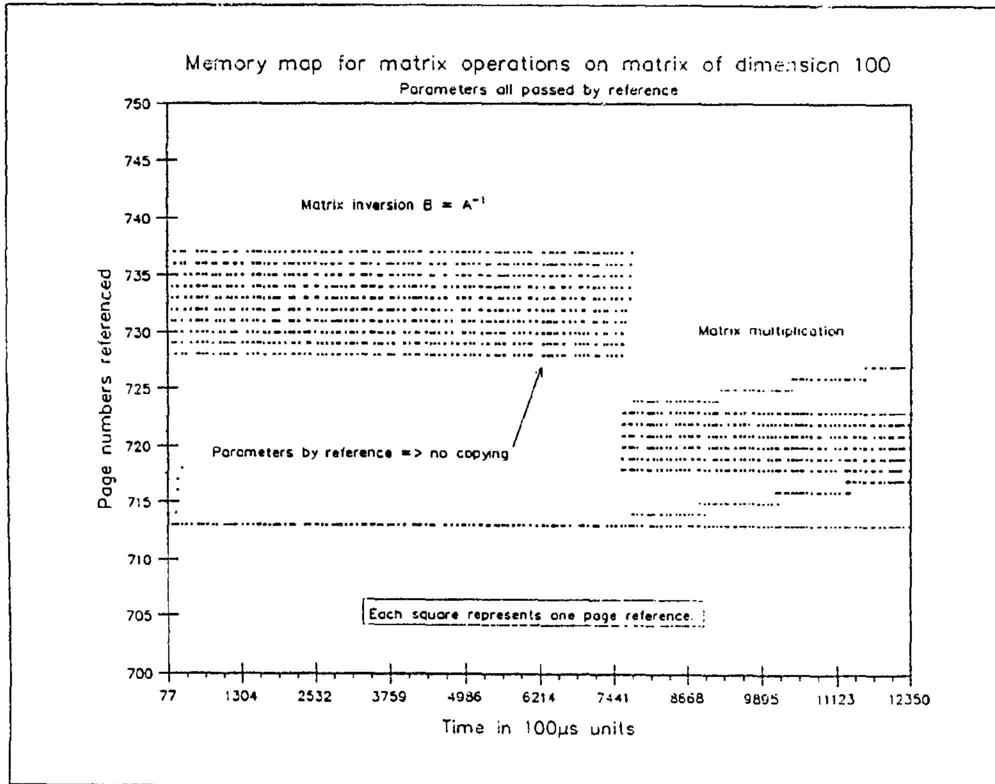


Figure 5.3: The memory map for mat100 with VAL parameters and VAR parameters.

Stellenbosch University http://scholar.sun.ac.za

exactly how many pages are required. It will be shown that only a fraction of the data structure size need be accessed simultaneously for efficient execution in the TVM system.

The courseness of the memory reference map stems from the sampled nature of the memory map. There are from eight upwards pages sampled at the same instant depending on the sampling interval. These eight pages correspond to the eight active cache pages. Because the implemented algorithm does not yet detect all the pages' status on each page fault, the information as to which pages were not accessed during a specific interval are not yet available. Under close examination it is clear that there are eight or more samples in any one column. This gives rise to not being able to deduce the efficient working set size from the memory map. The information contained in the memory map is of interest though as it shows clearly the locality of the program execution versus time.

The question now arise how would the memory map develope if more data structures were to be accessed simultaneously. The matrix benchmark was expanded so that during the multiplication phase more and more data structures were accessed simultaneously. The memory reference maps for increasing number of data structures can be found in fig 5.4. From the results given it is clear that for each additional data structure and additional active cache page would result in a performance improvement.

The second memory reference map of the two uses enough simultaneous active cache pages to make the pattern clearly visible despite the resolution courseness. The memory reference pattern of the B matrix is now also clearly visible.

Normalising benchmark.

The second benchmark generates a vector of random numbers, scans the whole vector to determine the maximum value and then divide the vector with the maximum. This program will be referred to as *NORM*. The memory requirements for the program is $8 * M$ where M is the dimension of the vector operated on.

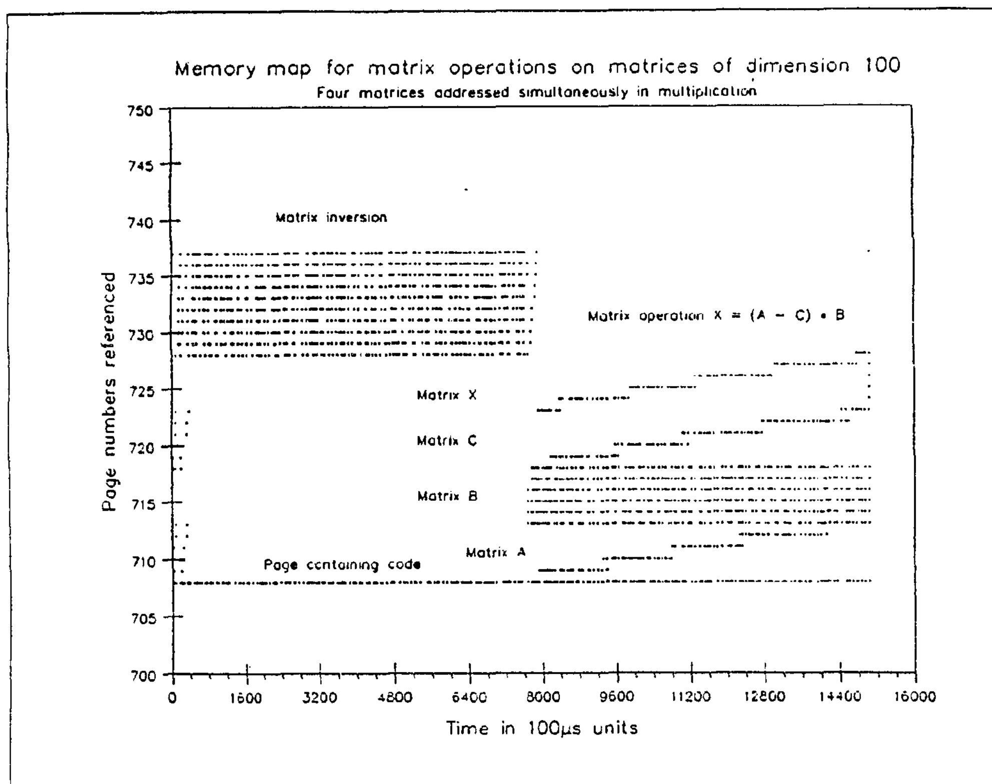
The memory reference map for the NORM benchmark is given in fig 5.5. It is quite like one would expect given the resolution problem. Clearly the data structure is scanned three times during the execution of the program.

Default TVM system parameters.

The memory management software running in all the benchmarks under the hardware chapter uses a FIFO page replacement algorithm on all three memory levels. Further demand paging is utilized. If nothing else about the size of any of the variables is said assume the following.

1. The active cache size is 8.
2. The non active cache size is 8.
3. The window size is 128.

Stellenbosch University http://scholar.sun.ac.za



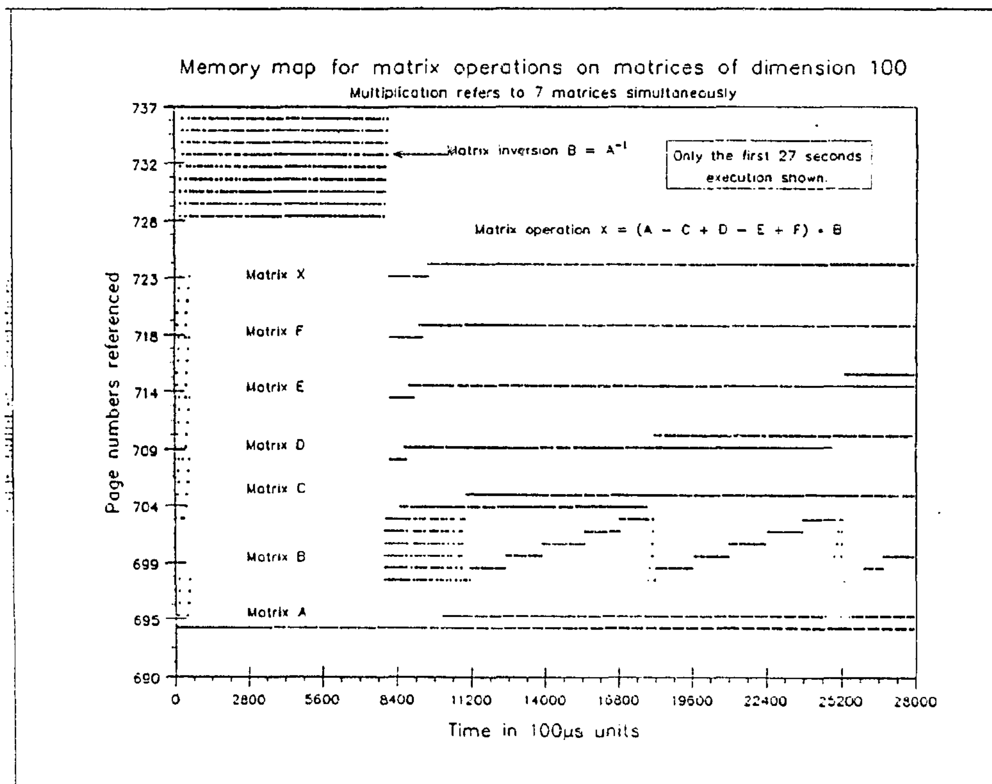


Figure 5.4: Memory maps for increasing number of simultaneous accessed data structures.

Stellenbosch University http://scholar.sun.ac.za

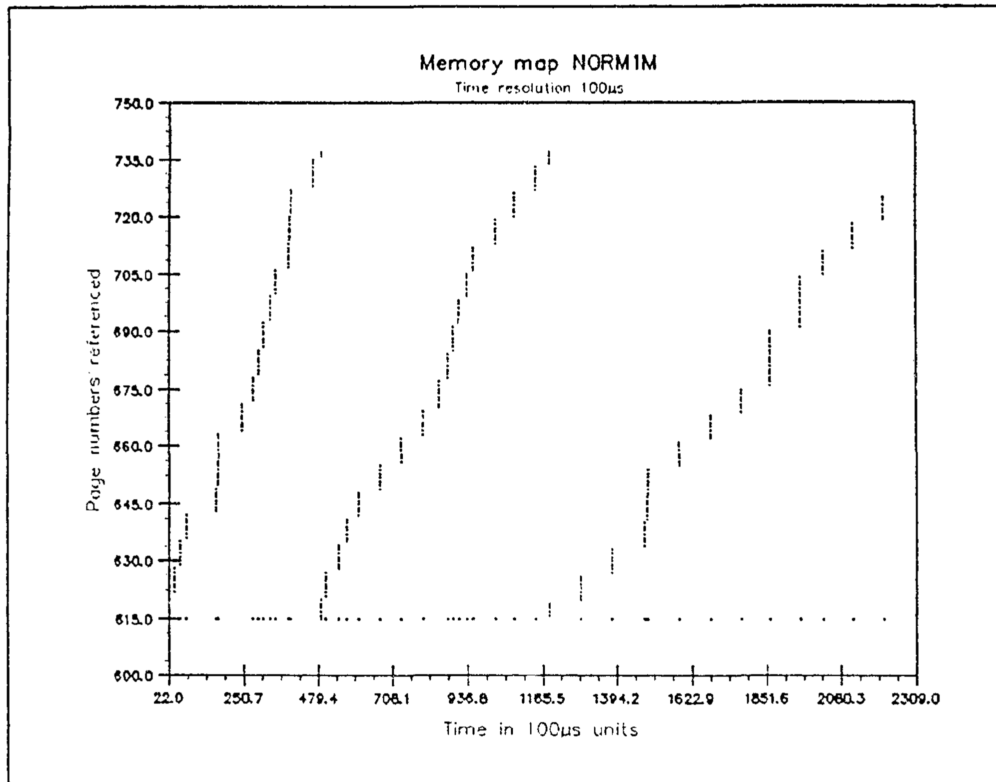


Figure 5.5: Memory map for the NORM benchmark.

Stellenbosch University http://scholar.sun.ac.za

4. The benchmark program is run in virtual memory mapped at 20 Megabyte.

5.3.2 The measure for comparison

The measure for comparison used in all the hardware evaluations is primarily the execution time. This deviates from the specific measures such as number of page faults, number of page pulls and the success function described in section 4.2. This is because the execution time incorporates all the individual measures and with the correct weight function. The other measures would then just be used where execution time does not provided any information w.r.t. an improvement or deterioration of performance.

5.3.3 The active cache size

This is one of the two single most important parameters in any virtual memory system. In section 3.2.3 it was already reported the lessons learnt from using only one active cache page. The question now is what is the optimum size for the active cache?

The answer is, it depends on the workload being run in the following way. Take for instance the database search program. If run completely in virtual memory ie. code and data, the number of simultaneous accessed sections in the virtual address space depends on which phase the program is in. During the initialization phase there is the code (which may be fragmented over two pages) and one data structure. This implies at most three active cache pages will be enough at any given time to address any place in the initial matrix. During the scanning phase it is only the code and one data structure again only two pages are needed. The same applies for the normalising phase. So it would be reasonable to expect that two or at most three active cache pages would give the same performance benefit as four or more pages would give.

The above argument rests on the assumption that the application program does not reference one data item per page before jumping to another page. In the latter case the minimum number of active cache pages giving acceptable performance would be far greater. It will be shown in section 6.5 that for the application classes the TVM needs to be evaluated, the data can be restructured in most cases to localize accesses. In the latter case the argument in the previous paragraph is valid.

Executing benchmark normalise

The normalise benchmark was discussed in section 5.3.1 and from the memory reference map there and the knowledge about the problem behaviour it is expected that the NORM benchmark will not run significantly faster in three or more active cache pages than in two active cache pages. Considering fig 5.6 it is clearly seen that increasing the number of active cache pages above two has no significant effect on the execution time. Also shown on the graph are the number of references to the non-active cache, window and disk for each active cache size.

Stellenbosch University http://scholar.sun.ac.za

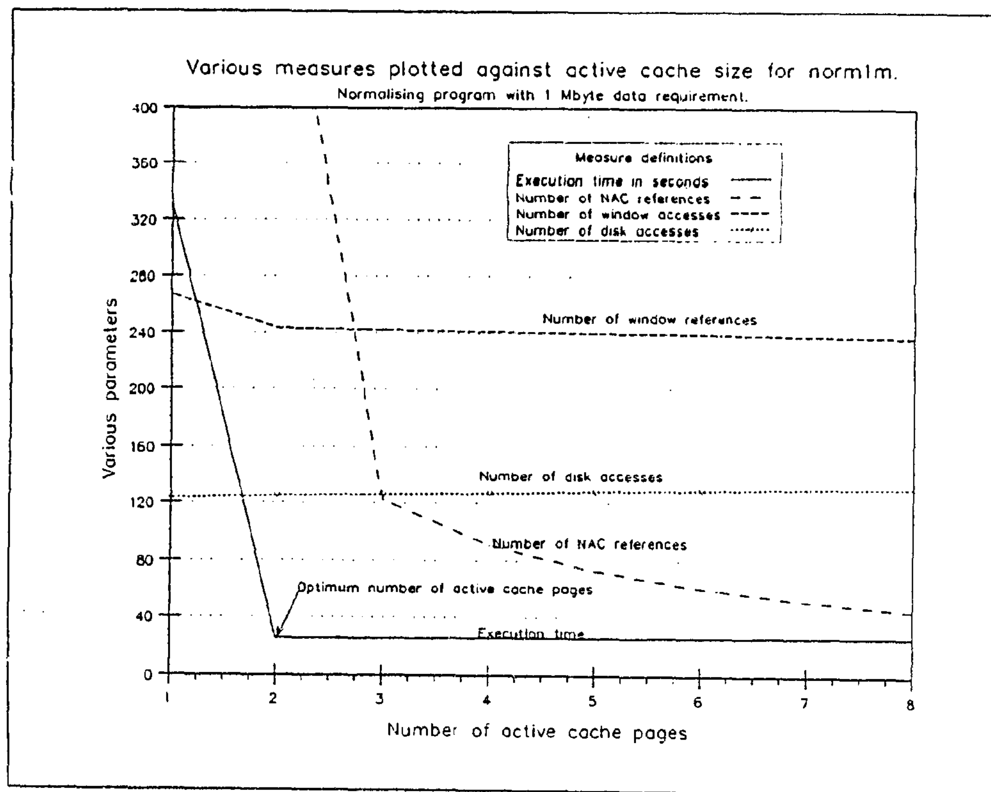


Figure 5.6: Norm program : execution time against increasing active cache size.

From the figure it can be seen that as the active cache was made smaller, more and more references to the non active cache were made. At two active cache pages the number of active cache references increased substantially, but the execution time stayed constant. The number of disk accesses or page pulls decreased by one though. This supports the authors' argument that the number of page pulls alone is not a significant measure on its own.

Another interesting observation is that the number of page pulls decrease with decreasing active cache size. This is a by product of the FIFO replacement algorithm used which fares better with a smaller active cache.

The matrix benchmark.

Considering the matrix application, there is also an optimal number of active cache pages, but it is more difficult to predict. The first phase consists of generating the matrix. This is similar to the database search problem so two pages should ensure optimum performance. The second phase consists of gaussian elimination to determine the inverse of a matrix. In this phase two rows of the matrix are accessed simultaneously in addition to the code. This leads to at least three pages being referenced simultaneous.

The last phase consists of a matrix multiplication where three data structures and the code is referenced at any one time. This implies at least four active cache pages for optimum results.

Stellenbosch University http://scholar.sun.ac.za

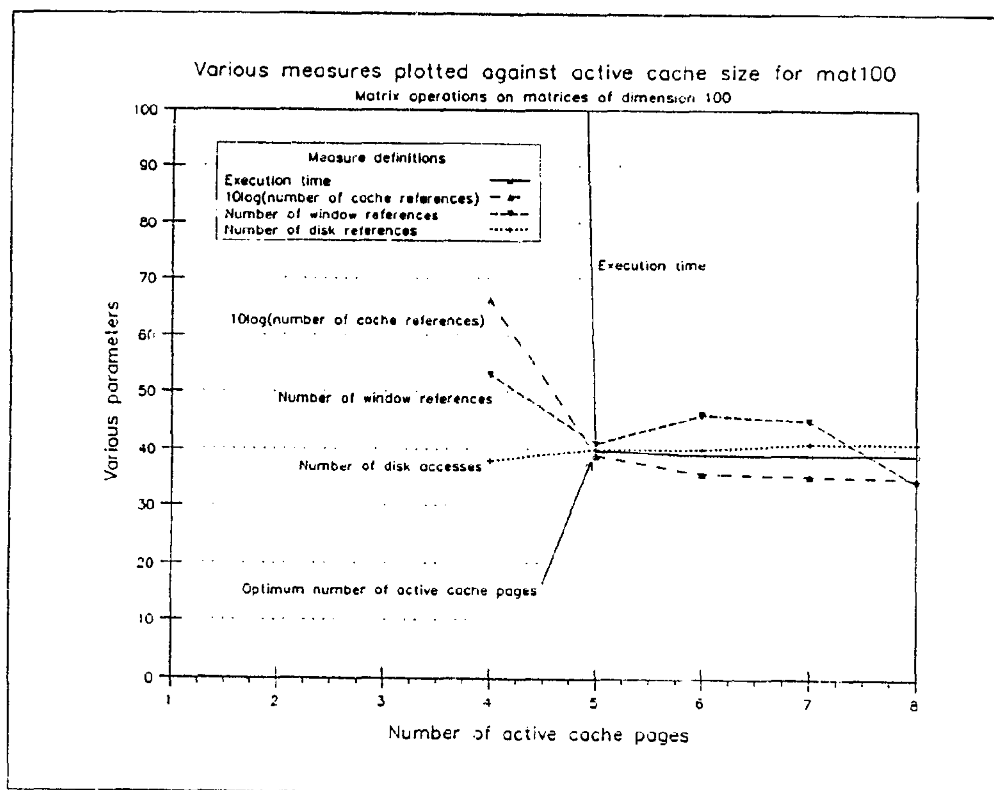


Figure 5.7: Matrix program : various parameters against active cache size.

The measured results are given in fig 5.7. From the graph it is clear that the minimum number of active cache pages required for optimal performance is five. Considering the memory map the it cannot clearly be seen why five is the optimum number. Instead looking at the first section of the memory map it seems that 11 pages would be optimum. This phenomena has been explained in section 5.3.1 as due to the sampled nature of the memory map.

From 5.7 it is observed that the number of page pulls is again a decreasing function and that the number of non active cache references again dominate. The number of window references reaches a local minimum at five active cache pages. This could be attributed to the FIFO replacement algorithm running on the active cache which would reach its optimum operating point at five active cache pages. Thus less wrong decisions would force pages out to the window.

The TVM system as available at the time of the measurements had eight active cache pages to use. Another relevant question is how many simultaneous active data structures can be efficiently run with eight active cache pages? The number of data structures in the matrix multiplication phase were increased and the execution time and other parameters measured for this increase. The results are shown in fig 5.8. From the matrix multiplication phase it can clearly be seen the number of pages required to execute the operation efficiently.

From 5.8 it can be observed that the increase from three to four simultaneously accessed data structures in the multiplication phase has no detrimental effect on the execution time. This gives the clue that the inverse operation accesses four structures simultaneously. From four

Stellenbosch University http://scholar.sun.ac.za

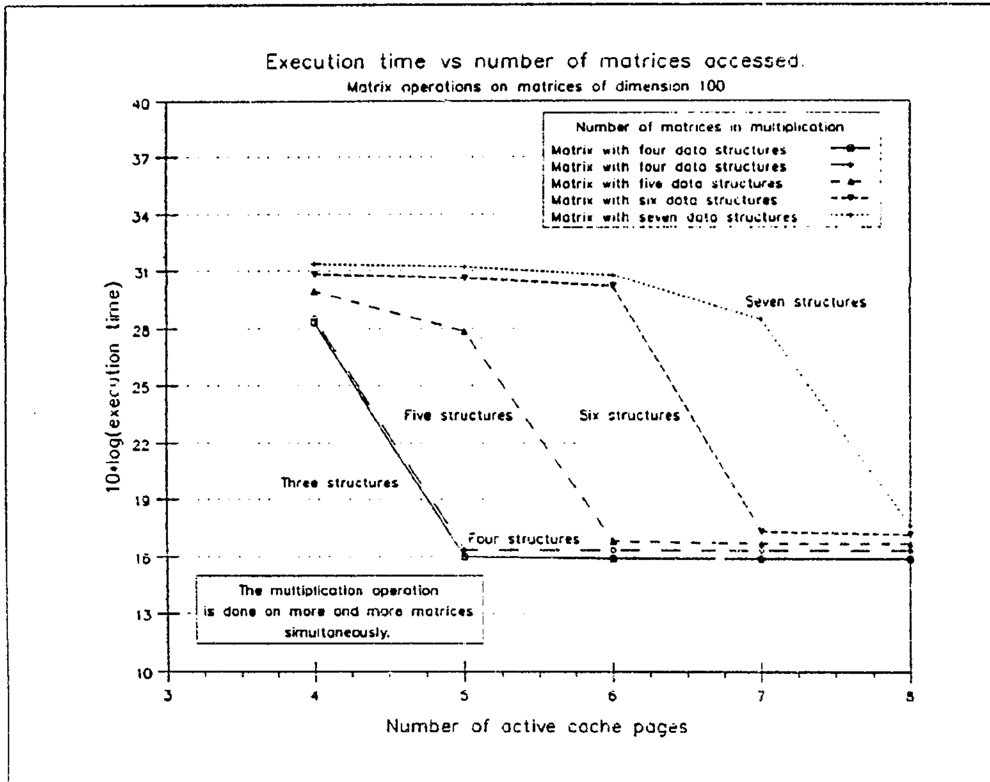


Figure 5.8: Increasing the number of simultaneous accessed data structures.

Stellenbosch University http://scholar.sun.ac.za

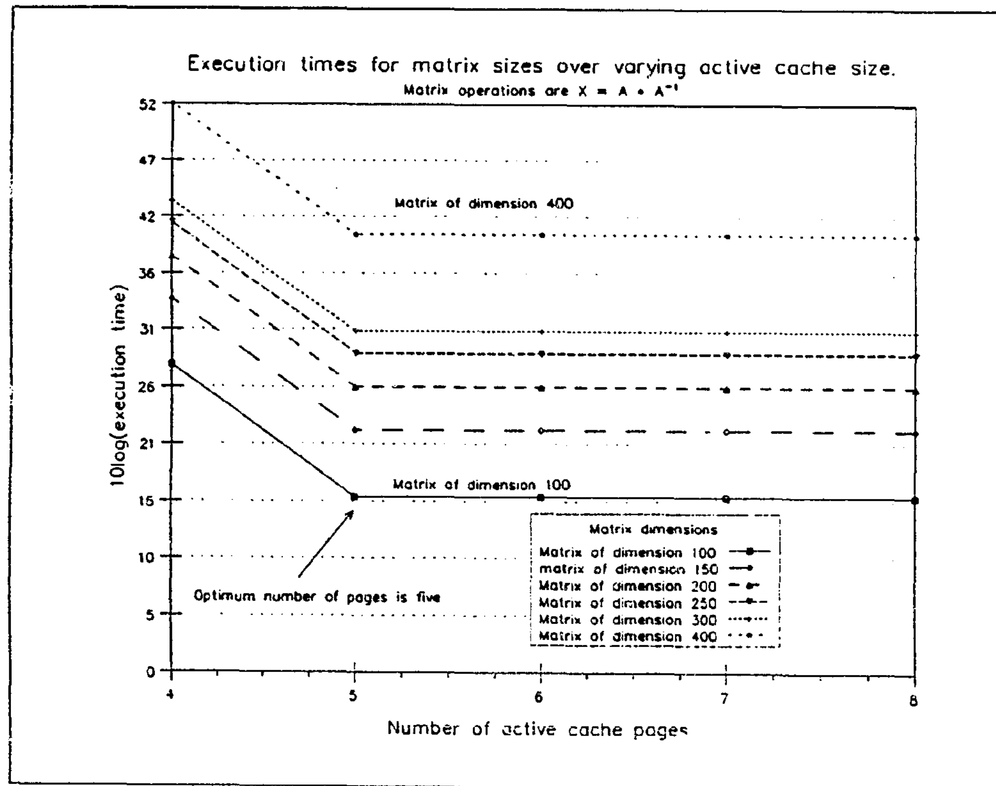


Figure 5.9: Executio.. times for matrices of different dimensions against active cache size.

structures onwards it is clear that the multiplication phase dominates the execution time.

One result of particular importance is that it does not matter what the dimension of the matrix or vector is, it is the number of simultaneously accessed data structures which determine the optimum active cache size. See fig 5.9 for a comparison between the execution time for matrix dimensions N = 100, ... 400.

5.3.4 The non-active cache size

The non-active cache refers to the cache not directly pointed to by the address translation mechanism, but the access time to it is still faster than the window because it only takes the overhead of updating the tables and moving the address translation pointers to address into this memory level. The total cost can be as low as $20\mu s$ but is normally in the region of $60\mu s$ to $80\mu s$. This high value depends on how much data is collected for each page fault.

It was decided to manage this as a separate memory level because in general the number of simultaneous active cache pages could be less than the available amount of so called cache pages.

The non-active cache could be or not be a first order influence on the performance of the virtual memory system. No where in literature has a similar structure been reported on. Although it

Stellenbosch University http://scholar.sun.ac.za

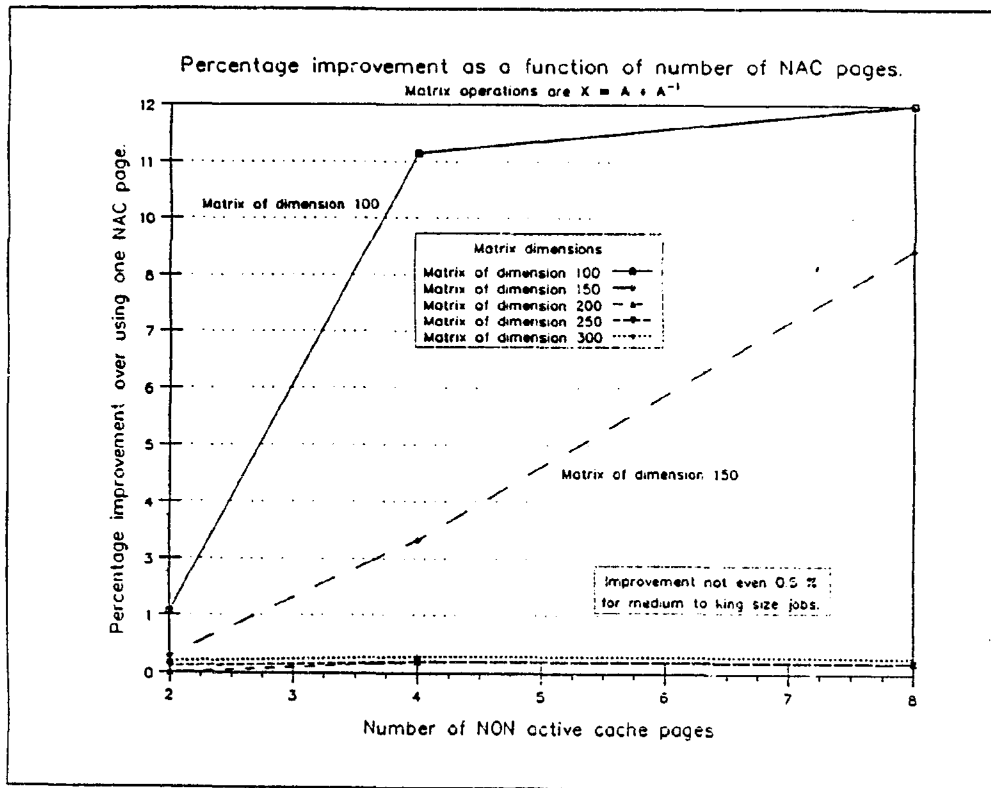


Figure 5.10: The improvement over one NAC in execution time for bigger NAC's.

corresponds exactly to the not directly addressable memory in conventional systems, its size is so small the results partaining to the conventional architectures does not apply.

The non-active cache will further be referred to as the NAC. Two experiments were done to determine what the influence on performance is of the NAC. In the current architecture with a maximum of eight active cache pages a maximum of 8 NAC pages are available. The first experiment took the active cache size as eight which is more than ample for the efficient execution of the two benchmark programs. In 5.10 it can clearly be seen that when the active cache size is big enough for efficient execution of the application the effect of the NAC is disappointing. The improvement for bigger data structures gets lower and lower!

From the previous graph it is clear that the active cache plays a much more dominant role than the NAC. This can also be explained as the NAC is to small to have a significant impact on the NAC hit ratio. The NAC should however play an important role in the case where the active cache is too small to hold the current working set. This means that for every page fault all the table updating as well as a page move must be made to establish addressability on the requested page.

The experiment conducted did include a set of five ac pages and the matrix benchmark. Recall that this is the optimum set before performance degradation sets in. The number of NAC pages were v ried from 0 to 11. The results are shown in fig 5.11.

Stellenbosch University http://scholar.sun.ac.za

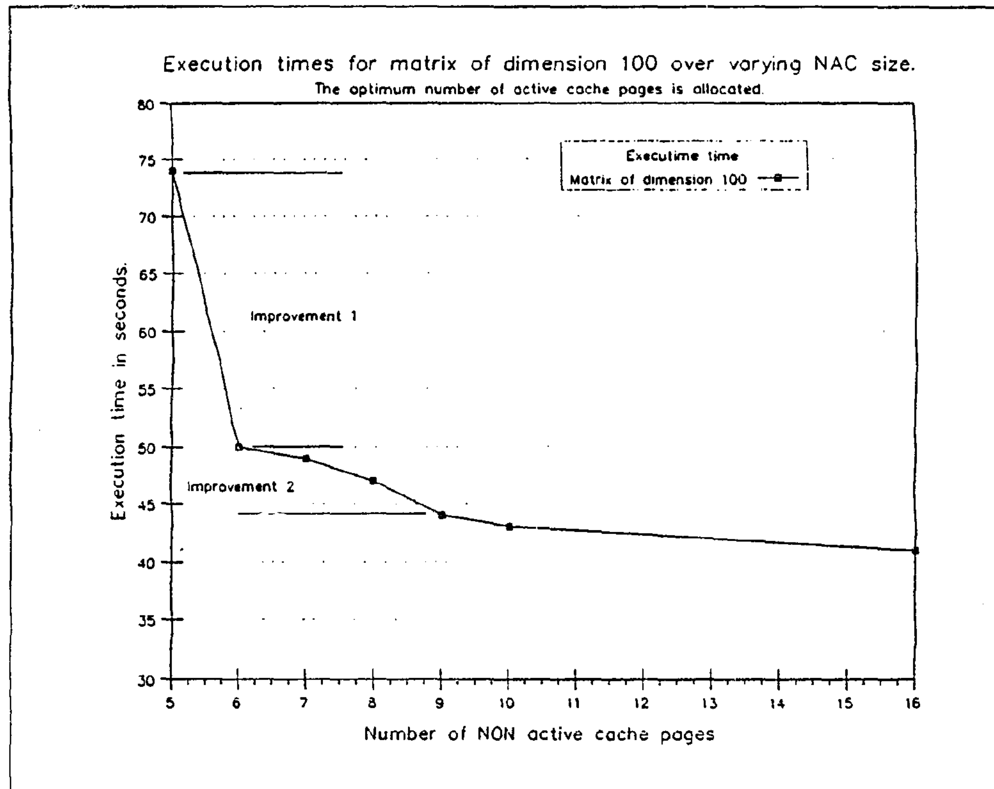


Figure 5.11: Execution time versus NAC size for optimum amount of ac pages.

Stellenbosch University http://scholar.sun.ac.za

The results are of some significance. The first improvement from 74 to 50 seconds stem from FIFO replacement algorithm used. The code pages are swapped out regardless when it has its turn at the front of the queue. This leads to the degradation in performance when this page is swapped into the window instead of the NAC.

The next ten seconds which represent 12 % of the possible execution time represents the degradation due to the window memory not being addressable by the XU. This number could be inflated due to the FIFO algorithm. This implies for each page access to the window the page must be moved to the active cache before execution can continue. This issue will be further investigated in section 5.4.

5.3.5 The window size

The window in the TVM system corresponds to the main memory in conventional virtual memory architectures. In those architectures the whole of the main memory can be occupied by pages. The active set of these is then again between 16 to 64 of the window pages. It is however deduced from some literature that all the pages in main memory could also be active in some architectures. This would normally only be found in mainframe computers where a multiuser environment exist in which each users' working set would be in main memory at the same time.

In principle the window is just a disk buffer and a keeper of the working set defined over a longer period of time. This working set corresponds to the measured working set in section 5.3.1. For smaller problems this window would be large enough to hold all the data items saving on disk access times except for the initial loading of the program into the window area. For programs with data structures much larger than the window the net effect of the window won't be significant because the window wouldn't be large enough to hold any working set ie. none of the pages in the window would reside in the active cache more than once except the code pages. This effect can clearly be seen in fig 5.12 where the mat150 program does not execute faster when more than 32 pages are allocated to it. The data requirements for the mat150 program is 1.26 Mbyte which corresponds to 77 pages.

The previous analysis is for medium sized jobs. For king size jobs completely flushing the window with every data structure, the effect of the window should not be significant. Rather the memory management strategy exploiting the parallelism in the disk interfaces will provide performance improvements. The last mentioned aspect as one which should be investigated.

5.3.6 Page size

In the introduction of this chapter it was explained that the latest page sizes were due to implementation [illegible]. The question arises is the page size the optimal page size? If not what is the optimal page size? It was clear from section 3.3 that there are many factors determining the efficiency of the page size selected. Two sets of measurements on the TVM system will give an indication of the relevance of the arguments in 3.3.

Stellenbosch University http://scholar.sun.ac.za

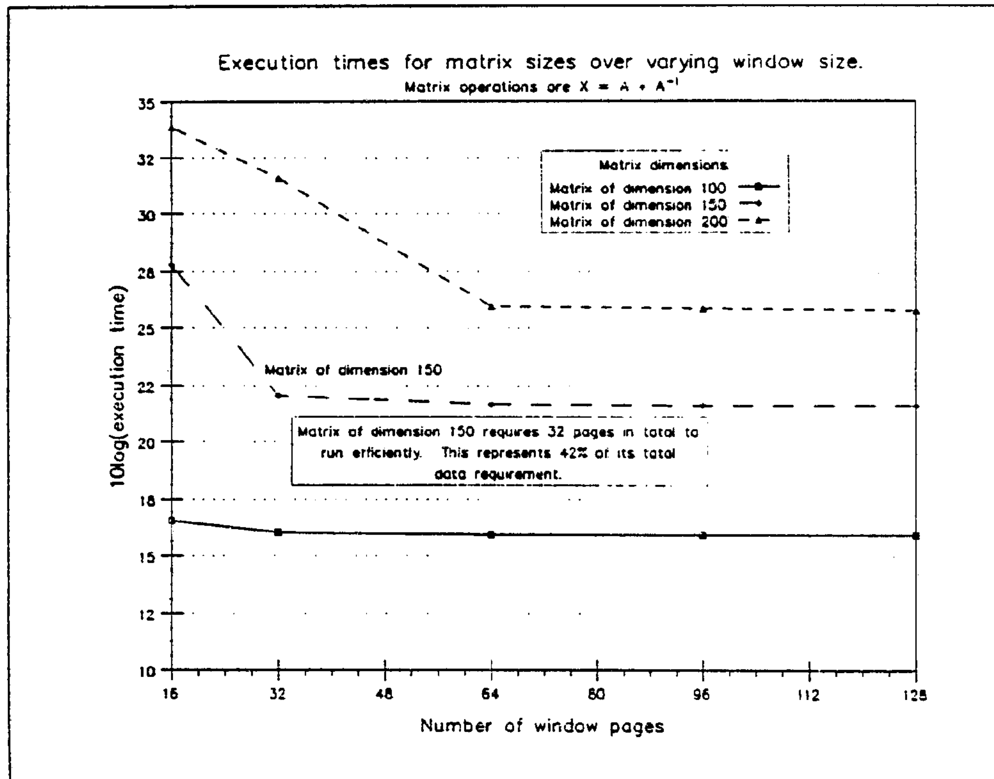


Figure 5.12: The execution times for the various matrix dimensions against window size.

Stellenbosch University http://scholar.sun.ac.za

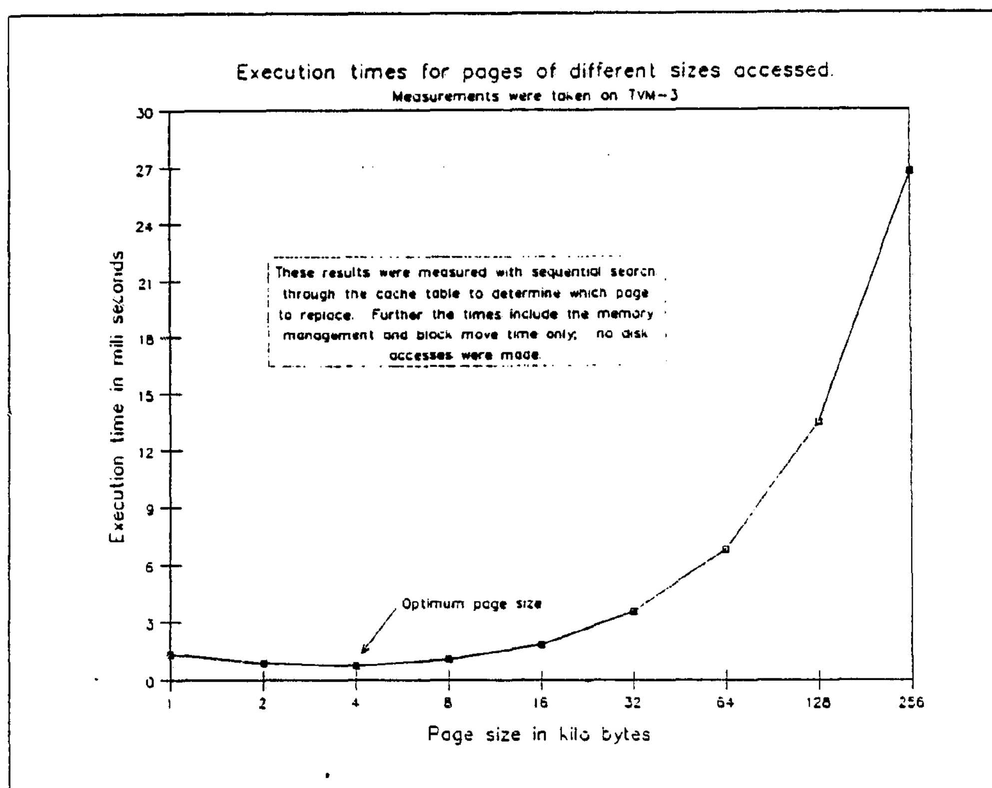


Figure 5.13: Page fault handling time vs page size.

The first set of measurements were taken on hardware revision 3. On this revision only one set of active cache page registers was implemented, but the page size was variable from 1kbyte to 256kbyte. Comprehensive replacement algorithms were implemented on the cache and the window. The data structures were simple linear structures which had to be searched for an entry. On this hardware and software combination measurements w.r.t. the time it takes to handle a page fault were made. The results are given in 5.13.

From this figure it is clear that the taking the overhead into consideration the optimum page size is 4 kbyte. It is also clear that any page size from 1k to 16k would not result in any significant performance degradation due to window access time. The times measured did not include disk access time. But from 3.3 it was clear that if disk access times were taken into account, better performance would also be obtained from larger page sizes.

Looking at the problem from an application point of view, the matrix benchmark was run on a 16 kbyte and 32 kbyte page size. From table 5.14 it is clear that the larger page size perform better when a smaller than optimum number of active cache pages are allocated. This can only be ascribed to the highly localized and linear nature of the matrix data structures which means that less overhead is spent on doing operations on the same amount of data. When more than the minimum number of pages required for efficient performance are allocated there is still a decrease in the execution time. This also points to the use of larger pages for the matrix type algorithm class.

Stellenbosch University http://scholar.sun.ac.za

	16k	32k
mat100 (ac.p = 4)	690	501
mat100 (ac.p = 8)	39	38.2

Figure 5.14: The effect on execution time when the page size is variable.

These results should be even more applicable for king size jobs, where disk accesses has to be made for every page fault. The seek time and rotation latency constitutes the overhead. This overhead will only have to be done once for a particular disk access.

5.4 Performance implications of TVM architecture

The active cache and window approximates structure in [Deitel 83] very well which should give up to 90 % of the performance attainable. The only significant disadvantage of the TVM architecture is the extra block move time incurred for each page fault where the page is in the window.

For the two application programs it can be calculated what the percentage of time is which is wasted due to this architectural feature.

$$T_s = ((W_r * 1.5 * M)/Ex) * 100$$

Where

T_s is the percentage of wasted time.

W_r is the number of references to the window.

1.5 represents every page moved in, but only half of those moved out as a result of a write reference.

M is the time for a page to moved in one direction.

Ex is the total execution time for the program.

Fig 5.15 gives the relation of % time wasted vs dimension size for the matrix benchmark. Recall that all these measurements were made with the FIFO page replacement strategy. This means that the performance degradation might be exaggerated as a result of this simple memory management policy.

The results are of great importance. It indicates that the loss in execution time is never larger than 15incurred for programs with data structures in the range as indicated. This implies that efforts to increase the TVMs' performance by providing more cache space would not have a significant effect on the performance.

Stellenbosch University http://scholar.sun.ac.za

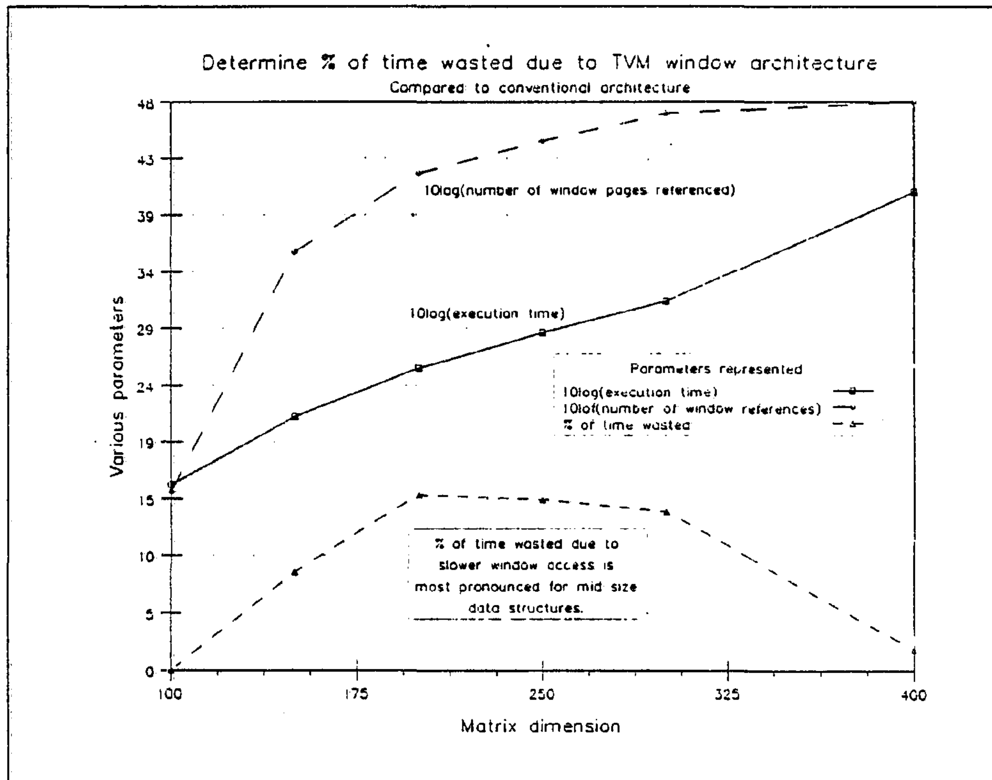


Figure 5.15: The % of time wasted vs the dimensional size for matrix.

Stellenbosch University http://scholar.sun.ac.za

Another fact which is reinforced by this figure is that the window play a very insignificant role in the execution of king size jobs. This is clear from the small performance loss incurred for the matrix of dimension 400.

5.5 Detail HW design

The detailed schematics can be found in appendix A. In appendix B all the relevant registers and their meanings can be found. Appendix C provide all the pal equations and a typical user hardware configuration with user documentation is provided in appendix D.

Stellenbosch University http://scholar.sun.ac.za

Chapter 6

TVM Software

The memory management software is just as important as the hardware. This chapter describes the memory management from the program specification right through to the evaluation of its performance. Many of the possible management strategies have been described in chapter 4 and will not be discussed again.

6.1 Program specification

The program specification could be done in two ways. The one is functional description in words and the minimal of exact mathematical relations. The other way would be to describe the whole system in as precise form as possible through the rigorous use of some form of mathematical theory of specification. The author choose the first method as his knowledge of the second method is not well enough developed to apply it on such a large program.

On the highest abstraction level the memory management software must:

1. Run on the XC such that its existence is transparent to the user of the XU transputer.
2. Every page addressed by the XU not currently visible to the XU, the XC must make this page available in the XU's address space.
3. The data integrity of every page to be replaced in the XU's address space must at all times be maintained by the XC.
4. All the memory management actions must be executed by the XC in the shortest time possible as to impose the minimum of overhead on the XU.

Stellenbosch University http://scholar.sun.ac.za

6.2 Program design

The program was designed in a top-down fashion and implemented in a bottom up way. The design process sets off with a process interaction diagram. This diagram indicates clearly the processes and the communication channels between them. This stage is followed by a modular decomposition of all the data structures and the operations defined on them. One or more of these modules may be identified in each process.

6.2.1 Process harness

The process harness describes the parallel processes and the communication channels between them. The fig 6.1 is a representation of the process harness as it exists at the time of writing. It is considered to add two more processes, one doing just roll back of dirty pages and the other rolling in of predicted pages. This will immediate complicate the mutual exclusion needed on the data structures.

6.2.2 Modular construction

The module is a data structure hidden from the users with access to it through operations provided on it. The concept of modular program construction stems from the old principle of divide and conquer. Programs constructed without modular decomposition would become so complex that writing it any other way would make the maintenance of it impossible.

Modules also tend to build on each other ie. the one module provide the lowest form of access to a data structure while the following module the builds on the first by defining its operations only in terms of the lower modules' operations and not directly on the data structure. The module hierarchy is given in fig 6.2.

The individual operations and parameters for each operation can be found in the source code of the program. Program available on request.

6.2.3 Data structures

The efficient management of an entity relies on information. The information w.r.t. the status of the pages are stored in various tables to enable efficient management of the TVM's virtual memory.

There are four basic tables in TVM, each corresponding to a memory level.

hit table A table containing an entry for each page in the 2 Gigabyte memory space.

window table A table containing an entry for each page in the window.

Stellenbosch University http://scholar.sun.ac.za

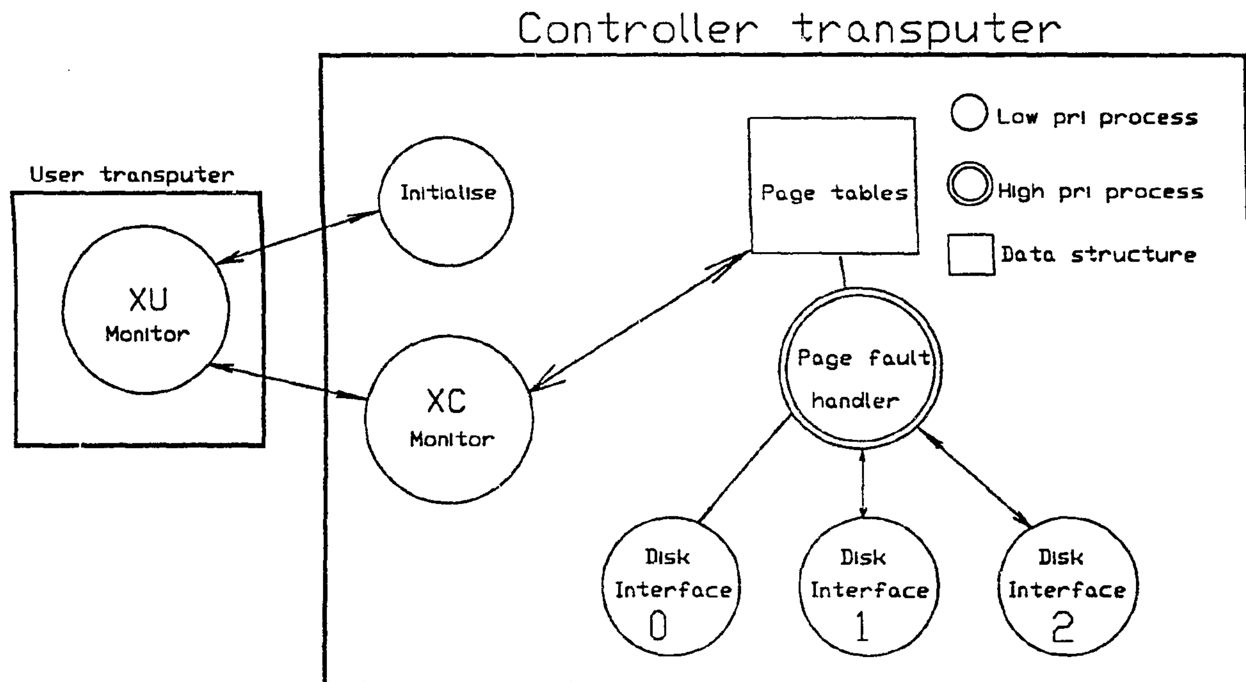


Figure 6.1: Process diagram for TVM.

Stellenbosch University http://scholar.sun.ac.za

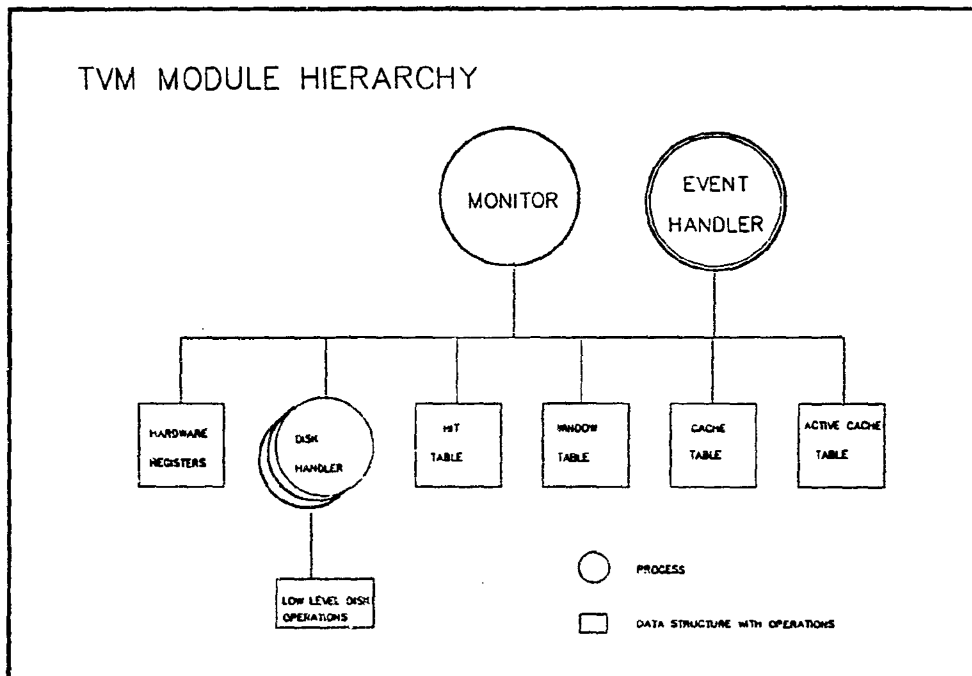


Figure 6.2: Module hierarchy for TVM.

Stellenbosch University http://scholar.sun.ac.za

cache table Contains an entry for each entry in the cache.

active cache table A subset of the cache table.

Two additional tables are for monitoring purposes only, ie. the trace table and the memory map table. These tables contain information entered during the running of a program and can be extracted by the post mortem monitor function.

Returning to the tables necessary for virtual memory management. The hit table has an entry for every page in the virtual memory of 2 Gigabyte. With a smallest page size of 16 kbyte this implies a table size of 131072 multiplied by the number of bytes per entry. This table size is small compared to the table size of 2 Megabyte which would result from 1 kbyte page size. The advantage of larger page sizes are immediately evident. The information kept for each entry in the hit table is.

```
| block no. | dirty bit | reference bit | in cache bit | in window bit |
```

Where the block number corresponds to the index in the cache of window tables where the page can be found. The four bits are used in the management of the virtual memory.

The hit table is the basic information unit. It would be possible to achieve memory management only with the hit table. But because it contains information on three different substructures, it would be very inefficient to use the hit table for searching in the three mentioned sub structures. The hit table does have one main advantage; the basic index into the hit table is the page number which is a direct lookup method available in any section of the event handler.

Because of the inefficiency which would result if the hit table was used for all the management functions of the substructures like for instance the searching for a page to replace, a separate table for each substructure was created. Because these tables are much smaller they are much more efficiently accessed. Each of the structure tables must however have a pointer back to the hit table because it is the only structure with all the relevant detail for those procedures who need it. Each of the three structure tables will now be discussed.

The window table are kept up to date to allow the efficient determination of a page to replace. Thus the only information kept in each window table entry is the page number found at the corresponding window index.

The cache table again are kept up to date for the efficient determination of the next cache page to replace. Because pages are moved between the window and the cache, the cache table must contain besides the page number also a pointer to the corresponding window page where the page resides in the window. For each cache table entry then two data items are stored.

```
|   page number   |   window block number   |
```

Stellenbosch University http://scholar.sun.ac.za

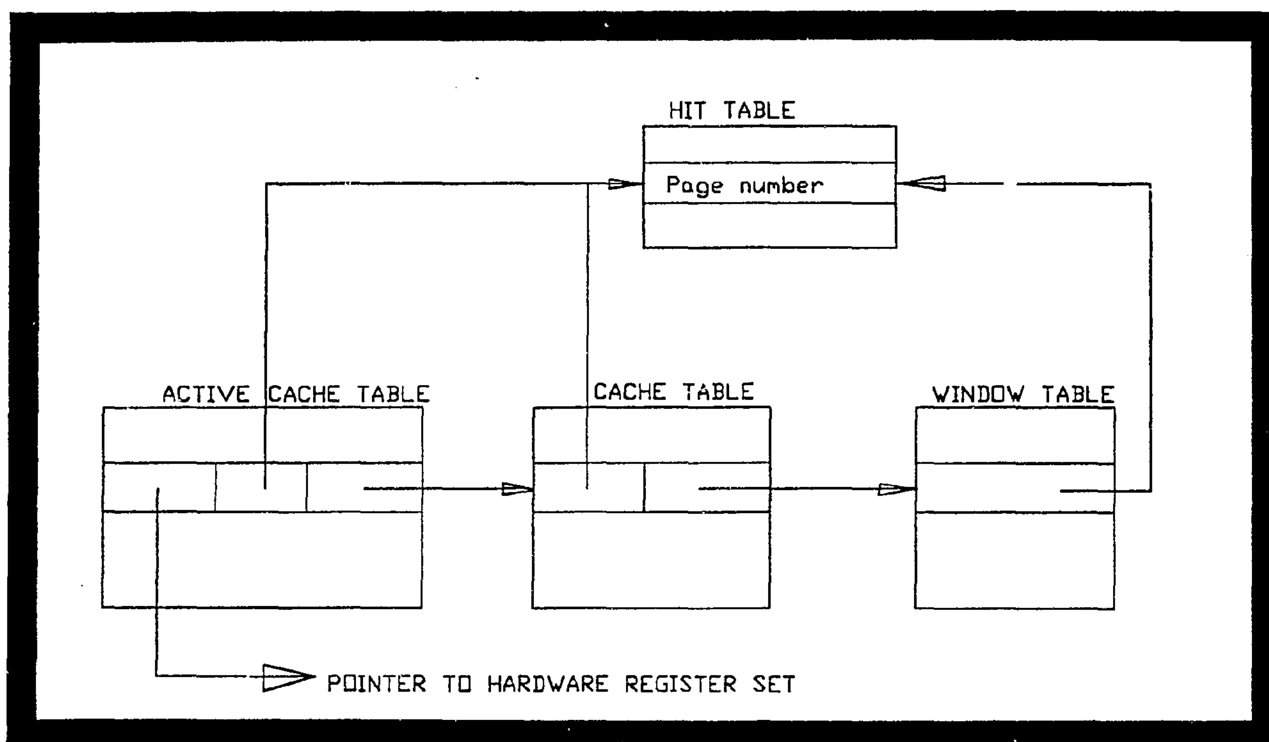


Figure 6.3: The inter relationship between the tables.

The active cache table is a subset of the cache table. According to the same principles as explained in the previous paragraph, each active cache table entry has the corresponding page number found at the active cache index location and a pointer back to the location in cache where the page can be found.

```
| page number  | cache block number | hardware register set pointer |
```

If one considers all the tables the inter relationship between the tables is depicted in fig6.3.

6.2.4 Program flow

The program flow consists of three main phases:

```
Initialization of operating parameters.
Memory management.
Post mortem investigation of TVM status.
```

Stellenbosch University http://scholar.sun.ac.za


```
event ? dummy
page.number := f( XU.address.register )

get active cache page to replace
if dirty active cache page
   update hit table

if page number in cache
   get.cache.index
else
   get cache page to replace
   if dirty cache page
      roll out to window
   if page in window
      move page from window to cache
   if page on disk
      get window page to replace
      if dirty window page
         roll out to disk
      move page disk to window to cache

set active cache pointers
hand back to XU
```

Figure 6.4: Main algorithm on MMU.

The first and last phases are user controlled from the XU transputer. Under normal operating circumstances the user would not need to alter the default operating parameters or do an investigation of how the TVM behaved, but it is useful tools in optimizing the TVM for a specific program.

The memory management operations are initiated by the event handler, and the event handler is triggered by a page fault. This event handler is the heart of the XC and the main algorithm is given in fig 6.4.

6.3 Program evaluation

The program was evaluated in two ways. The first is the determine the execution time of the time critical sections. The second evaluation compares two replacement strategies with each other to determine the better strategy.

Stellenbosch University http://scholar.sun.ac.za

6.3.1 Execution times

The execution times for the event handler were of utmost importance up to revision 3 of the hardware because of only one active page. This active page had to be swapped between code and data for each single reference and the page fault handling time was of the essence. In version 4 and 5 of the hardware up to 16 active cache pages are allowed simultaneously. Thus the need for an ultra fast page fault handler does not exist any more. Regardless of the last fact the page fault handler must still be executed efficiently to ensure acceptable levels of performance. The time for transferring pages between the different memory levels are given in the following table.

	Time in micro seconds	Overhead
Cache to cache	65	65
Cache to window to cache	4523	427
Window to cache	2335	287
Disk to window to cache	78584	-
Window to disk to window to cache	137268	-

From the second and third entry the overhead in handling a page fault can be determined. The block move instruction used to move the data has an execution time of $8*50ns+2*w$ where w is the number of words to transfer. The overhead is measured in micro seconds and constitutes 10 % of the time to handle a page fault. This figure is inflated by all the data collection and processing which included in the time measured.

The overhead for the cache to cache move could be as low as 10 micro sec. The overhead is the only action taking place in a cache to cache move. The figure again includes most of the data collection time as well.

6.3.2 Replacement algorithms

Only two replacement algorithms were implemented for evaluation. It is expected that the more sophisticated algorithms won't perform much better because the TVM system was designed with king size jobs in mind.

The FIFO and the Random replacement algorithm will be compared. The FIFO algorithm is the worst case, since it is not a stack algorithm with the useful property that the number of page faults decrease with increasing the buffer size. While the random replacement algorithm, assuming nothing about the executing process, should not fare much better than the FIFO, but at least performance should improve with larger buffers because for a particular run the random replacement algorithm can be modeled as a stack algorithm. The results from running the same program on both replacement algorithms are shown in table 6.5 and table 6.6.

Stellenbosch University http://scholar.sun.ac.za

Number of window pages	Execution time	
	random	fifo
16	582	601
32	158	160
64	147	147
96	149	145
128	146	145

Figure 6.5: The execution time for matrix 150 under FIFO and RANDOM replacement algorithms.

Number of window pages	Execution time	
	random	fifo
16	2412	2455
32	1037	1436
64	399	392
96	396	383
128	387	376

Figure 6.6: The execution time for matrix 200 under FIFO and RANDOM replacement algorithms.

Stellenbosch University http://scholar.sun.ac.za

The small window size simulates the effect of a king size job. The random replacement algorithm performs better for the smaller window sizes. This could be due to the random replacement algorithm making less bad judgements by not replacing the code often.

The larger window size should lower the probability of both algorithms to make a bad judgement. It is interesting to note that with bigger window size it is the random replacement algorithm performing worse. This could be due to the fact that with the bigger window size the fifo algorithm takes longer to replace a page on average than the random algorithm. This is because of the more page space the FIFO algorithm takes longer to get to the same page to replace.

6.4 Future development

The TVM software is not at its optimum. The existing XC program will perform adequately in all circumstances, but the question is could the TVM system perform better? There are three aspects not yet implemented on the TVM system, two of which is described in chapter 4. These are the near optimal stack replacement algorithms and prediction algorithms. The third aspect concurs the organization of the pages on the disk subsystem. All three of these improvements will investigated in future fourth year projects.

6.4.1 Stack algorithms

The near optimal stack replacement algorithms is expected to perform better on the smaller programs, but not significantly better on the larger programs. This fact has been introduced earlier in connection with the % of time wasted because of the window architecture (see section 5.4). There it was shown that for king size jobs the number of window references were so low that there were no more than 2% degradation due to window references. This implies that it does not matter how well the window will be managed, no significant performance increase will be seen because very few pages accessed will be in the window.

6.4.2 Prediction

The prediction algorithms is another way to improve the performance of any virtual memory system. For the TVM system this option is of particular relevance as the memory management is done by a different processor from the one running the main program. The prediction operation can therefor be done in parallel with the execution of the user program. Further there are three separate channels to disk which can also run in parallel.

Two paradigms were discussed in chapter 4.4 of which both could be applied in the TVM environment. The first is the sequential prefetch algorithm. This stated that on every page fault also fetch the following page. In the virtual memory with its disks organist as three way memory this could be accomplished by fetching the next two pages from the other disks at the

Stellenbosch University http://scholar.sun.ac.za

same time. Such a prefetching action would consume on average no extra time in handling a page fault. This assumes of course that there is space in the window for three new pages.

A more realistic algorithm would search the window for a page to replace on the disk furthest from the disk with the requested page. Thus the page to be swapped back can be done concurrently with the page requested being swapped in and its successor also being swapped in. This would be applicable if another page does not have to be swapped out to accommodate the successor page.

The second paradigm would hold even a bigger promise of performance increase as it uses advance information given by the user program itself. Without considering the user program lets assume the XC program would receive from time to time information on blocks of dead pages and blocks of pages which would be referenced some time in the future. On each page fault as many dead pages as possible could be removed and as much space available could be filled with pages predicted to be referenced in the future. To ensure that this prediction process does not interfere with the XC process when it would not give a performance improvement, it would only pull in predicted pages if there is space to do so.

When considering the user application class of matrix algorithms, it is clear that the user can indicate to the XC when certain sections of the data structures will not be used. The user could also indicate which sections of the data will be referenced in the future. The trade off with this method would be to determine on what granularity this information is optimally useable by the XC. For example in the matrix multiplication problem when a row of the first matrix has been multiplied with a column of the second matrix both of these structures are redundant. Would it be better to signal on a row for row basis that these data items are dead, or on a matrix for matrix basis?

The action of signalling to the XC that a page is dead or will be referenced in future is a simple task of sending a link message containing the pointer to the structure in question to the XC. The XC can then decide whether it could use it or not.

6.4.3 Disk organization

The disk subsystem as will be explained in the next chapter consists of three autonomous interfaces connected to the XC. Data to these interfaces can be transferred in parallel without major performance degradation on the XC. This is a situation which should be exploited by reordering the pages to maximize performance.

Storing every third byte per disk would be one way of trying to exploit the parallelism available. This would mean that the data had to be reconstructed back at the XC which would consume uneccessary time. Still striving to exploit the parallelism available one could dived each page into three sections and store a third on each disk. This would solve the problem of reconstructing the data at the XC. But this would not necessarily lead to better performance. When considering the 16 kbyte pages used divided by three leads to 5 kbyte pages. When considering the average access time for a disk transfer the time is

Stellenbosch University http://scholar.sun.ac.za

Data size	Total access time in ms
512	38.6
1k	39.2
2k	40
4k	42
5k	44
8k	47
16k	57
32k	76
64k	115
128k	192
256k	346

Figure 6.7: Disk access times for different page sizes.

$$average.access.time = average.seek.time + average.latency + data.transfer.time$$

The average seek time is 28 ms; the average latency = 10 ms and the transfer time = 850 kbyte/s * data size. When considering the following table it is clear that a third reduction in transfer amount only results in a 30% saving in time. Thus there might be other organizations providing a better performance.

It would be better to consider organizations based on the knowledge of the application program running on the XU transputer. One other scheme which would result in a significant performance benefit is to place every third page on the same channel. This means that when the one channel is occupied the other two can also be occupied with either pages being rolled out or rolled in in anticipation. These options will only be comparable when measured results are available.

There is one other measure which could be taken to improve performance. The disks could be inherently organist in the page size instead of some smaller fraction of a page size. For instance the disks could be formatted with a block size of 16 kbyte. This should increase the average data transfer time to 1.25 Mbyte/s because the intersector gap would disappear for 16 kbyte blocks. For a 16 kbyte block transfer the access time would be 38 milli sec. This is an improvement of 33 % ! This last option must also be investigated.

6.5 Other ways to improve performance

All the ways to improve performance so far takes great effort from both the system programmer and the user. There is however a few simple rules which if used by the user substantial performance improvements are possible.

Stellenbosch University http://scholar.sun.ac.za

Dimension	Data size	Normal algorithm	Transpose algorithm
40	89600	2.4	2.4
160	1433600	609	183
228	2911104	6608	544
320	5734400	25419	1633
420	9878400	141215	16275

Figure 6.8: Execution times for matrix algorithm and its transpose algorithm.

These methods has to do with the way a program and its data are organist. In section4.5 a very involved method was discussed to localize a programs' data structures and code. For the application to matrix algorithms this process can be done be the user through knowledge of the referencing patterns of the algorithms which he codes.

Take for example the matrix multiplication algorithm.

$$C = A * B$$

Each row of A is multiplied by the corresponding column in B. If this algorithm is coded in Pascal then the underlying organization of matrices are row major. This has no effect on the program executing in real memory. In virtual memory this however means that each row of A may be found in one or more adjacent pages, but each element of the columns of B will be found on a different page. This implies that for every multiplication operation a page fault will result. Contrast this to a small modification in the algorithm where B's transpose is stored before the operation.

$$C = A * B^T$$

In the last instance the rows of A are found in one or more adjacent pages and the columns of B are also found in one or more adjacent pages. This have a remarkable performance increase as shown in table 6.8.

This same principle could be applied to many matrix algorithms. Many other has the inherent property already for instance the Guassian elimination used to calculate the inverse of A, has the property of scanning through rows to do its operations. This fits in perfectly with the organization of the underlying virtual memory architecture. These performance enhancements are possible for all virtual memory architectures.

Stellenbosch University http://scholar.sun.ac.za

Part III

Secondary memory system

Stellenbosch University http://scholar.sun.ac.za

Chapter 7

Hardware

The secondary memory system consists of one or more moving head disk drives. One or more of these disk drives can be connected to a disk interface or channel. The interfaces are connected to the XC through the high speed INMOS links. At most three of these disk interfaces can be connected to the XC due to the other link on the XC being used for downloading its process from the XU and for monitor actions from the XU.

This chapter describes the design of the secondary memory subsystem of the TVM system, both the hardware and the software.

7.1 Overview of solutions

There are three distinct issues which need to be considered.

- The interconnection of the XC to the disk interfaces.
- The disk subsystem architecture.
- The disk interface standard to the disk itself.

Each of these will now be considered in more detail.

7.1.1 XC to diskinterfaces

There are two viable options for connecting the XC to the disk interfaces. The one is connecting the disk interfaces via the remaining 3 INMOS links. The second is mapping the disk interfaces in the memory map of the XC. The first option is the implemented on the TVM system and it does have the advantage of a decoupled system, but it would be of interest to determine which one of the two gives the highest performance.

Stellenbosch University http://scholar.sun.ac.za

The link connected option have a bandwidth capability that of the INMOS link. This at 20 Mbit/s is 1.8 Mbyte/s. The advantage of the INMOS link is that it is a DMA engine closely tuned with the transputer. Thus once initiating the transfer the transputer processor is not ever required to give any assistance. The three links can run simultaneously each at 1.8 Mbyte/s without affecting the performance of the main processor significantly if the processor is currently busy with a CPU bounded process.

Under the circumstances described above the only memory mapped solution competitive in time is a DMA solution see [Mostert 89]. Again considering three interfaces, each can be mapped as a buffered DMA port with the buffer associated with each port some multiple which leads to efficient disk transfers. The speed from this buffer to the XC's main memory is 8 Mbyte/s. This is four times as high as the link interface and gives the same performance advantages as the hardware links. The main drawback is a tightly coupled transputer and disk interfaces.

The first measure was chosen to provide the TVM system with any disk interface standard which would be deemed viable. The only prerequisite would be the INMOS link to the XC. This design option proved very valuable as it will turn out in the next section.

7.1.2 Disk subsystem architecture

The disk subsystem is defined as the architecture from the point on the disk interface which is compatible with the disk communications bus and includes all the disks on one interface. The interface providing a basic communications path to the processor is called a channel with the understanding that only one disk can communicate over a channel at any one time, while many disks may be doing a seek operation.

The first disk subsystem made provision for ST401 interfaces. This interface is supported by the INMOS M212. A disk interface incorporating these components were designed (refer E) and evaluated. The average transfer rate were found to be 100 kbyte/s which is far slower that can be achieved with other inter- faces. The standard just mentioned can only support two disks on each interface. This means that the architecture were very much restricted to three channels and two disks per channel.

During the development of the TVM the SCSI bus standard became in more widespread used. This interface can provide data throughputs of up to 5 Mbyte/s for the 8 bit wide SCSI bus. Further this bus supports up to seven disks on each interface card. This provided the option to connect even more disk memory on each channel. The three channels stayed as this was a basic property of the interconnection between the XC and the disk interfaces and that was decided on previously. The simple interface between the XC and the disk interfaces proved very valuable as a different interface could easily be attached to the XC.

The design question is how many disks must be connected to each channel? One obvious parameter is the amount of disk space needed when small fixed sized disks are used. But is there a performance advantage to be gained when many smaller disks are connected in stead of on[illegible]g disk per channel?

Stellenbosch University http://scholar.sun.ac.za

One intuitively would say that more than one disk per channel could lead to improved performance if all the disks seek at the same time. The multiple disks per channel could be modelled as a pipe line. The basic problem with the seeking disks are then clearly visible as the problem to keep a pipeline filled. To keep the pipeline on the TVM system filled would require a very sophisticated memory management algorithm.

With a sophisticated software algorithm which does multiple writeback operations and multiple prediction operations a performance increase could be achieved. This is an option which could be best exploited when the exact page referencing structure is known in advance. In data search applications this advance information would be known. This would then increase the performance for these applications.

7.1.3 Diskinterface architecture

This section will describe the disk interface architecture which was designed based on comparative evaluation studies. It will be shown that the fastest architecture would be of no performance benefit than the second fastest architecture because of a communications bottle neck elsewhere.

From [Mostert 89] it is clear that the only interface mechanism which would not degrade performance is a 'custom' DMA mechanism. Assuming that the basic mechanism from the SCSI bus to the disk interface's memory is DMA, there is still a multitude of configurations each with its own performance characteristics. Three of these DMA supporting architectures will now be explored to determine the best architecture for the disk interface.

The first option disables the transputer during the SCSI bus DMA data transfer phase. The transputer is then enabled and sends the data received from the SCSI bus to the disk host. This creates a butt joint data transfer pattern as indicated in fig 7.1 under option A.

The second option implements a two port memory with both the SCSI device and the transputer with access to it. The DMA data transfer from the SCSI bus to the two port memory can then be done simultaneously with the transputer sending a previous block of data via its link to the disk host. This overlaps two of the operations in option A. What is added however is the time to transfer the data from the two port memory to the transputers' own memory. See fig 7.1 under option B for a graphical representation.

The third option implements dual buffers both accessible from the transputer and the SCSI DMA device. While the one is being filled by the SCSI DMA device the transputer can empty the other to the link. This results in saving the transferring time in option B by the transputer from the two port memory to its own memory. See fig 7.1 under option C for a graphical representation.

From fig 7.1 it is clear that option B is 30 % faster than option A for more than one block. Option B implements a pipe line which implies that the performance increase can only be achieved if the pipe line is full. During normal disk operations files consisting of many blocks are transferred which satisfies this basic requirement.

Stellenbosch University http://scholar.sun.ac.za

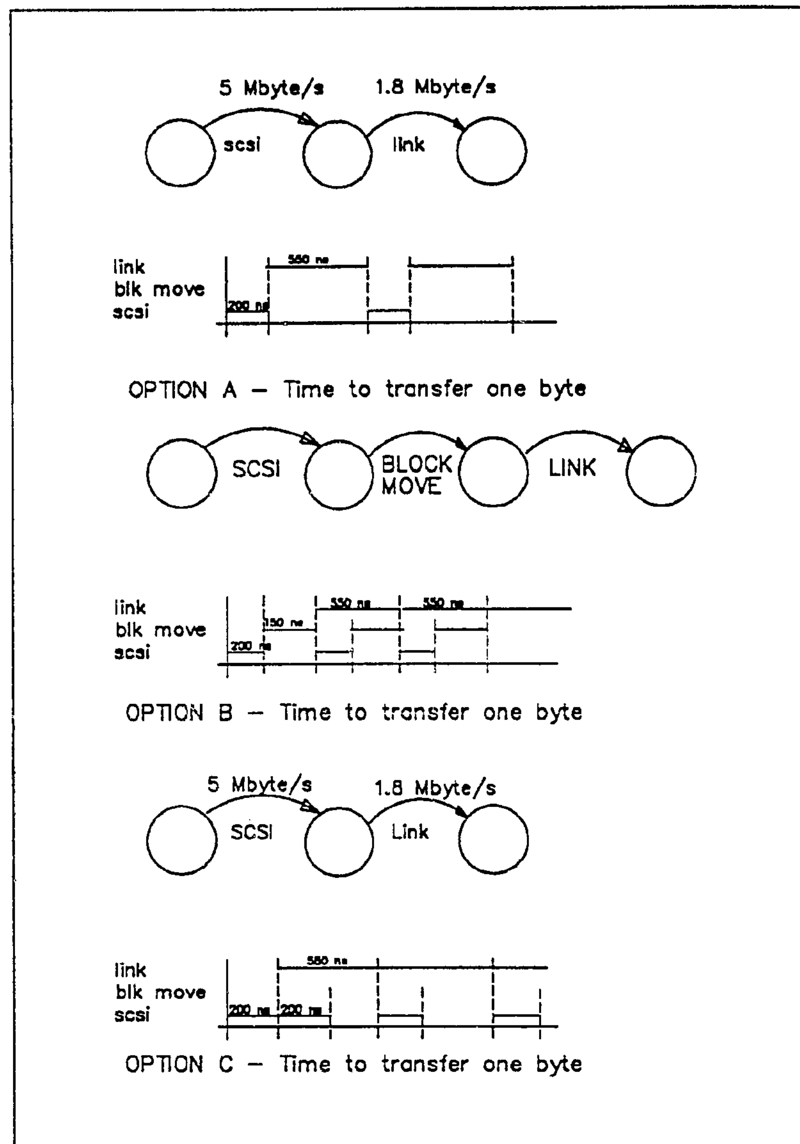


Figure 7.1: Evaluation of disk channel architecture.

Stellenbosch University http://scholar.sun.ac.za

Also clearly visible from the time diagram for option C is that option C is not faster than option B with the exception of the startup time. This is because the basic bottle neck is the serial link communication.

From the above discussion it is clear why the author opted for option B.

7.2 Diskinterface design

A block diagram of the disk interface is given in fig 7.2. This interface is built around the 16 bit T212 processor to provide INMOS link compatible communication with the XC and a SCSI bus controller to provide compatibility with the SCSI bus to the disk drives. As explained in the previous section the single buffered DMA solution will give the best results and this is implemented by a shared buffer between the T212 and the SCSI bus controller.

The DMA mechanism is a very simple address generator always starting at the same address. No stopping comparators were provided as the SCSI-bus supporting chip included an end of block counter and signal. This simplified the DMA mechanism to the simplest implementation.

The complete schematic diagram, pal equations and register explanation can be found in appendix F.

7.3 Performance evaluation

The performance of the hardware was measured with a logic state analyzer. The results showed that the SCSI interface to memory bandwidth is as high as 10 Mbyte/s. The maximum the SCSI bus controller can handle is 5 Mbyte/s. Thus the disk interface is 'fast' enough for the fastest SCSI disk device that supports the SCSI interface data rate of 5 Mbyte/s.

The performance of the specific disks used in the evaluation showed that over one sector the transfer rate is 1.25 Mbyte/s. Because of intersector gaps this transfer rate lowers to 850 kbyte/s over the whole sector with 512 byte sectors. If data transfers of greater than 1Mbyte are made the data transfer rate lowers even further to 450 kbyte/s. This last decrease is due to inter track switching of heads and inter silinder head movement.

Stellenbosch University http://scholar.sun.ac.za

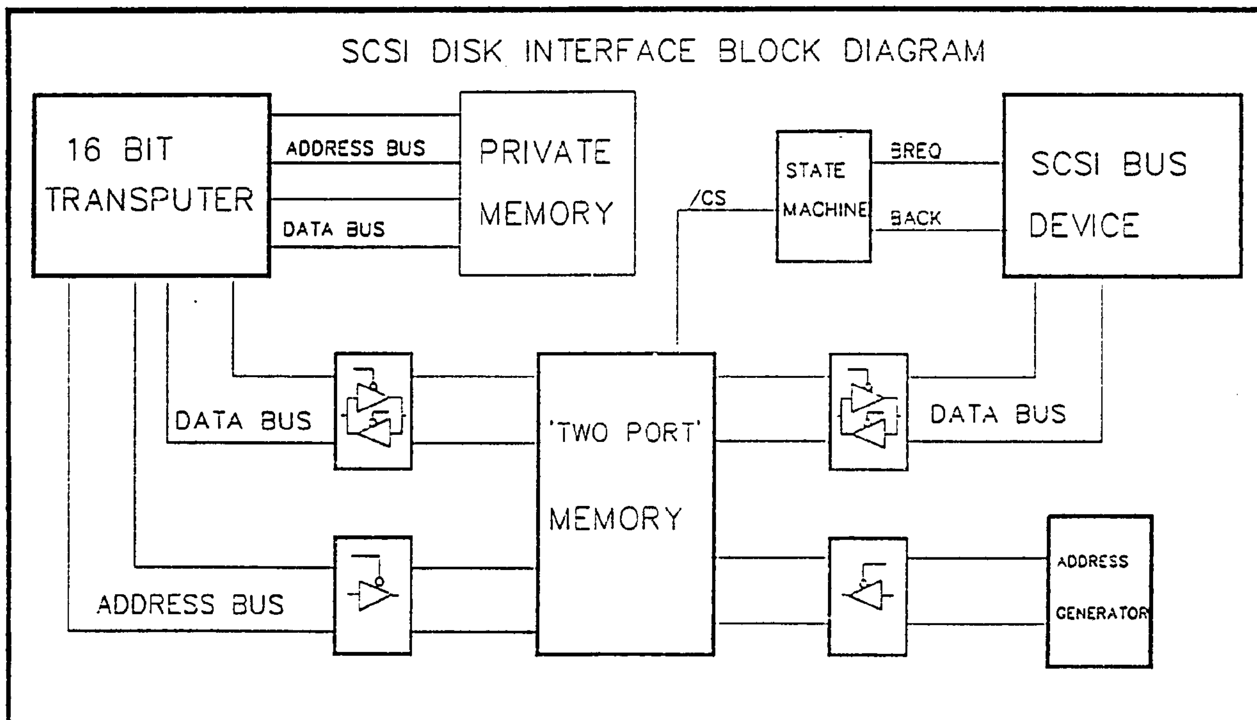


Figure 7.2: TVM scsi disk interface block diagram.

Stellenbosch University http://scholar.sun.ac.za

Chapter 8

Software

8.1 Program specification

The program on the disk interface must provide a simple interface to the XC program. The operations and their parameters are:

```
to.scsi ! format; interleave
from.scsi ? report

to.scsi ! read.data; begin.block; no.of.blocks
from.scsi ? data
from.scsi ? report

to.scsi ! write.data; begin.block; no.of.blocks
to.scsi ! data
from.scsi ? report
```

8.2 Program design

The main algorithm is a selection loop running for ever servicing requests from who ever is using the disk. The three operations as described under the specification is supported. The reading and writing operations does incorporate a parallel process mechanism to achieve maximum efficiency.

To utilize the full overlapping potential of the single DMA buffer and the INMOS links, there is only one solution of significance. That is the two DMA engines must be kept going in parallel.

Stellenbosch University http://scholar.sun.ac.za

Thus the algorithm for writing out data to the disk is:

```
PAR
  move.data.to.shmem (in.channel, dma.to.shm, shm.to.dma)
  hl.dma.data.out (shm.to.dma, dma.to.shm)
```

The two processes synchronize before the data is moved from the T212 main memory to the buffer block also visible to the SCSI bus DMA engine. The synchronization scheleton is:

```
PROC  mov.data.to.shmem (CHAN OF ANY inc,
                         CHAN OF INT dma.to.shm, shm.to.dma)
  SEQ
    buf.no := 0
    SEQ i = 0 FOR no.of.buffers-1
      SEQ
        inc ? data.buf[buf.no]
        dma.to.shm ? ready
        shmem := data.buf[buf.no]
        shm.to.dma ! ready
    inc ? [data.buf[buf.no] FROM 0 FOR no.of.last.blocks*block.size.int]
    dma.to.shm ? ready
    shmem := data.buf[buf.no]
    shm.to.dma ! ready
:
PROC  hl.dma.data.out (CHAN OF INT shm.to.dma, dma.to.shm)
  SEQ
    SEQ no.of.accesses.to.disk
      SEQ
        command.out(cmd.buf, 6, success, status0, status1, phase)

        dma.to.shm ! ready
        shm.to.dma ? ready

        enable.host.dma.transfer (dma.host.write)
        dma.data.out(no.of.blocks.in.buf,success,status0,status1)
        disable.host.dma.transfer ()
        status.in( status, success, status0, status1)
:
```

Stellenbosch University http://scholar.sun.ac.za

8.3 Performance evaluation

The average throughput over big data sizes is 450 kbyte/s. This is less than half the peak transfer rate when measured on one block of 1.25 Mbyte/s. When considering the highest transfer rate over a full track of 850 kbyte/s, the figure of 450 kbyte/s is acceptable if the seek time and rotational latency is taken into consideration.

Stellenbosch University http://scholar.sun.ac.za

Part IV

In conclusion

Stellenbosch University http://scholar.sun.ac.za

Chapter 9

Effect of VM on program execution

The effect of virtual memory on the execution time of a program has not yet been investigated. Table 9.1 contains the measured results of programs run on the TVM system.

The results in 9.1 refer to the program being contained to a large extent in the window memory. At least the current data structure is kept in the window memory.

When the current data structure can not be kept in the window memory, the performance should degrade due to the additional page pulls that have to be carried out. Table 9.2 gives the results for a forced small window size of 16 pages. The matrix program of dimensions 150 and 200 are run on this artificially small window to simulate the situation as stated.

The results in table 9.2 are indicative of a non-optimized disk subsystem. The performance loss would be the same for programs with very large data structures. This would indicate that another solution other than virtual memory is sought for these programs. Perhaps a parallel solution?

Matrix dimension	Total data size	Real memory	Virtual memory	% performance of virtual memory execution
100	0.5 M	36	39	92%
150	1.25M	122	145	84%
200	2.25M	288	380	75%
250	3.5M	560	739	75%
300	5M	998	1355	74%

Figure 9.1: Percentage performance of virtual memory system when compared to execution in real memory.

Stellenbosch University http://scholar.sun.ac.za

Matrix dimension	Total data size	Real memory	Virtual memory	% performance of virtual memory execution
150	1.25M	122	600	20%
200	2.25M	288	2420	12%

Figure 9.2: Percentage performance of virtual memory system with very small window when compared to execution in real memory.

Stellenbosch University http://scholar.sun.ac.za

Chapter 10

Conclusions

The requirement for a virtual memory based transputer have been met with a viable and performance effective implementation. This applies to a single user environment where small, medium size and king size jobs are run.

The various architectural features found on the TVM system have been shown to have advantages and disadvantages.

The unique two processor system have two advantages. The first is a multiuser operating system can now safely be implemented on a transputer. Secondly the memory management actions, like prediction can continue while the main transputer is busy with non-page fault generating actions. If the first mentioned is to be implemented, the non active cache should be expanded to create more page space so that context switching can be done faster.

For the memory hierarchy with the window memory separated from the main processor it has been shown that up to 15 % performance is lost because of this architectural feature when compared to a conventional virtual memory architecture.

With regard to the optimum sizes for the different levels of the memory hierarchy. The optimum active cache size have been shown to equal the number of data structures accessed simultaneously plus one page for the code. This size could be more depending on the fragmentation of the code.

The non active cache have been shown to have an insignificant performance impact. This is due to its small size. It did however compensate for the non-optimal replacement algorithm by making sure that the page containing the code never went any further than the fast non active cache.

The window show the same behavioral characteristics as the active cache, but on another level of memory requirement. It was shown that as long as the data structures accessed could be completely held in the window, the performance decrease due to virtual memory was 8% to 16%. But as soon as the window size is far smaller than the basic data structures addressed, the virtual memory system performance degrade to the level where the effect of the window is non-existent. This last phenomena takes place when king size jobs are run on the system. The

Stellenbosch University http://scholar.sun.ac.za

performance loss in the latter case can be as high as 90%. This poor performance is a result of the disk subsystem performance and should be addressed by optimizing the disk subsystem.

The disk subsystem was designed and evaluated for one channel and one disk. As a result no performance measures were available for a possible optimum disk subsystem. The various measures to increase performance were however discussed of which the most notable is the inherent parallel architecture of the three channels which should be exploited.

The disk interface itself was evaluated, designed and its performance measured. It was shown that the second fastest architecture with only one two port memory performed just as well as a double two port memory architecture would.

The basic memory management software was developed. This included all the data structures necessary for efficient management. The basic memory management policy of demand paging was used throughout. Two page replacement algorithms were evaluated ie. FIFO and random replacement. From literature is was shown that the FIFO page replacement algorithm is far from optimal and even worse than the random page replacement algorithm. It was shown in this report that both algorithms have nearly the same performance and that FIFO performs better with more memory and random performs better with less memory.

In conclusion the TVM implementation is a cost effective and performance efficient virtual memory implementation for the transputer lacking any virtual memory support primitives.

Stellenbosch University http://scholar.sun.ac.za

Bibliography

[Bakkes 89] P.J. Bakkes. Conversation with P.J. Bakkes on the TVM system. *Stellenbosch University, Department of Electronic Engineering* (1989,1990).

[Belady 66] L.A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal.* Vol. 5. No. 2. (1966), pp.78-101.

[Deitel 83] Harvey M. Deitel. An Introduction to Operating Systems. *Addison-Wesley.* pp. 181-243.

[Denning 70] Peter J. Denning. Virtual Memory. *Computing Surveys.* Vol. 2. No. 3. (1970), pp.153-189.

[Denning 72] Peter J. Denning. Properties of the Working Set Model. *Communications of the ACM.* Vol. 15 (1972), pp.191-198.

[Denning 80] Peter J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering.* Vol. SE-6(1980), pp. 64-84.

[Du Plessis 89] J.J. du Plessis. Conversation with J.J. du Plessis on the TVM system. *Stellenbosch University, Department of Electronic Engineering* (1989).

[Dorgeloh 89] G.A. Dorgeloh. Tegnoloog helped debugging first, second and third prototypes. *Stellenbosch University, Centre for Electronic Services (SED).* (1989).

[Gupta 78] Ram K. Gupta. Working Set and Page Fault Frequency Paging Algortihms: A Performance Comparison. *IEEE Transactions on Computers.* Vol. C-27 (1978), pp. 706-712.

[Hatfield 71] D.J. Hatfield. Program restructuring for virtual memory. *IBM Systems Journal.* Vol. 3 (1971), pp.168-192.

[Hyde 88] Randall L. Hyde. Overview of memory management. *Byte.* Vol. 13 No. 4 (1988), pp. 219-225.

[Mattson 70] R.L. Mattson et. all. Evaluation techniques for storage hierarchies. *IBM Systems Journal.* Vol. 2 (1970), pp.78-117.

[Mostert 89] Sias Mostert. Interfacing transputers to the real world. *Paper presented at Parallel Simposium.* CSIR, (1989).

Stellenbosch University http://scholar.sun.ac.za

[Oliver 74] N.A. Oliver. Experimental data on page replacement algorithm. *National Computer Conference.* (1974), pp.179-184.

[Pina 89] R.F. Pina. Designer of the first prototype of the TVM system. *Stellenbosch University, Centre for Electronic Services (SED).* (1988).

[Pohm 81] A. V. Pohm and T. A. Smay. Computer Memory Systems. *COMPUTER.* October 1981, pp. 93-110.

[Smith 78] Alan Jay Smith. Sequential Program Prefetching in Memory Hierarchies. *COMPUTER.* December 1978, pp. 7-21.

[Trivedi 76] Kishor S. Trivedi. Prepaging and Applications to Array Algorithms. *IEEE Transations on COmputers.* Vol. C-25 (1976), pp.915-921.

[Yoshizawa 88] Yasufumi Yoshizawa. Adaptive Storage Control for Page Frame Supply in Large Scale Computer Systems. *1988 ACM SIGMETRICS Conference.* May 24-27, 1988. (1988), pp.235-243.

Stellenbosch University http://scholar.sun.ac.za

Part V

Appendices

Stellenbosch University http://scholar.sun.ac.za

Appendix A

Transputer virtual memory hardware

PROPRIETARY INFORMATION. AVAILABLE ON REQUEST.

Stellenbosch University http://scholar.sun.ac.za

Appendix B

TVM registers

PROPRIETARY INFORMATION. AVAILABLE ON REQUEST.

Stellenbosch University http://scholar.sun.ac.za

Appendix C

TVM PAL equations

PROPRIETARY INFORMATION. AVAILABLE ON REQUEST.

Stellenbosch University http://scholar.sun.ac.za

Appendix D

TVM users manual

Stellenbosch University http://scholar.sun.ac.za

Appendix E

M212 disk interface

PROPRIETARY INFORMATION. AVAILABLE ON REQUEST.

Stellenbosch University http://scholar.sun.ac.za

Appendix F

SCSI disk interface

PROPRIETARY INFORMATION. AVAILABLE ON REQUEST.

Stellenbosch University http://scholar.sun.ac.za

Appendix G

Interfacing transputers

Stellenbosch University http://scholar.sun.ac.za

SM0908 8 September 1989

REAL WORLD INTERFACING WITH TRANSPUTERS

S MOSTERT

1. Introduction

The transputer is a high performance processor with an instruction troughput of 10 MIPS. The question arise as to how to get data into and from a transputer or transputer network.

This paper focus on the mechanisms available on the transputer and evaluate them according to S/W complexity, H/W complexity and maximum attainable data bandwidth. Where possible examples are included of implementations at Stellenbosch University.

Contents

1. Introduction

2. Transputer architecture

3. Services included in application spesific interface

4. Peripheral interfacing

    4.1 Memory mapped

    4.2 Link adapters

    4.3 Memory swapping

    4.4 DMA

        Implementation 1

        Implementation 2

        Implementation 3

    4.5 DDMA

        Peak transfer then post processing

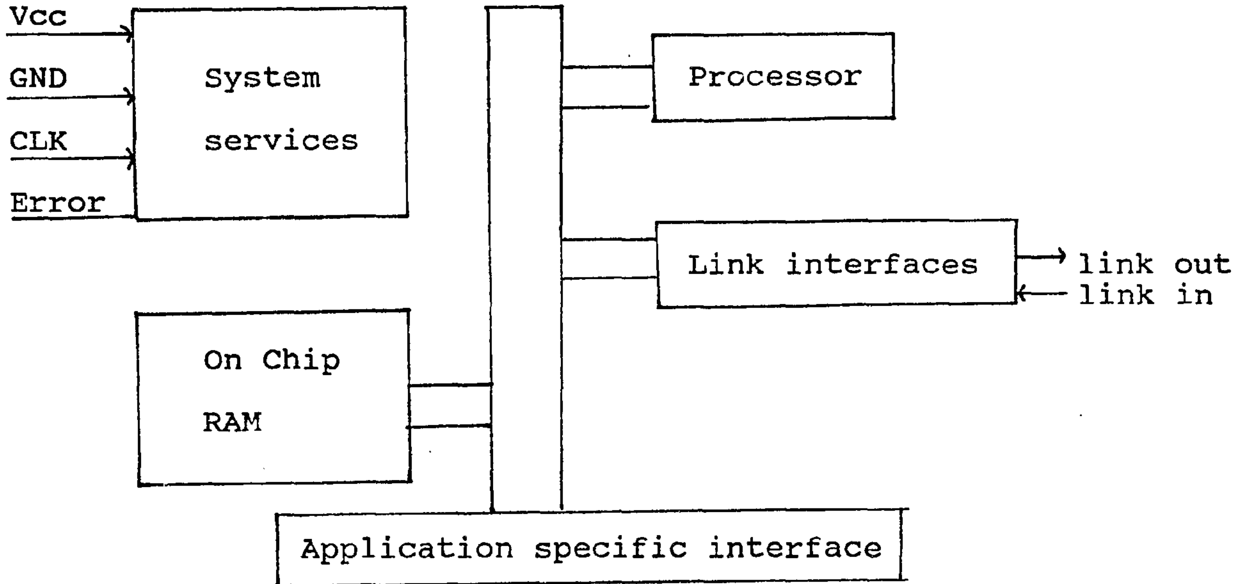
        Peak transfer while processing

5. Concluding remarks

Stellenbosch University http://scholar.sun.ac.za

2. Transputer architecture

The processor architecture can be summed up as follow:



- memory interface
- disk interface

There are thus two mediums of getting data to and from the transputer i.e.

	Memory Bus		Links
	MUX AD Bus	DEMUX AD Bus	
Components	T414, T800	T212, T222, T801	All transputer compatible components.
Basic transfer measure:	150ns	100ns	20 MBits/sec
Attainable BW	13MB/s	10MB/s, 20MB/s	1,8 MB/s

for blockmove instruction

Stellenbosch University http://scholar.sun.ac.za

3. SERVICES INCLUDED IN APPLICATION SPECIFIC INTERFACE

It is useful to consider these services as they do have an upper data BW.

3.1 Interrupts

The signals provided are:

event-request
event-acknowledge

The dominating time delay for using events, is the interrupt latency. For the following processors it is given as:

T212	2,65μs
T414	2,9μs
T800 (without FPU)	2,9μs
T800 (with FPU)	3,9μs

For the T800 with FPU operations this translats to an upper bound of 1MByte/s data bandwidth. The above analysis assumes an interrupt per entity to transfer. These figures becomes part of the startup time for transfers longer than one entity.

3.2 Memconfig and memwait

Memconfig determines the primary period for external memory references. (Upperbounds given in paragraph 2.)

While memwait is an input to the transputer allowing slower devices than basic memory to communicate with transputer.

3.3 Memreq and Memgranted

These signals are provided to facilitate a DMA controller taking over the bus.

Assuming a DMA transfer takes place in 150 ns on a T800. Assume further a 32 bit peripheral bus is available (highly unlikely).

Then for cycle stealing DMA we have:

Time to transfer one word = 750ns

This translates to a data transfer BW of 5,3 MB/s.

Back to the real world with a 16 bit databus is means a data transfer BW of 2,6 MB/s, with an 8 bit databus 1,3 MB/s.

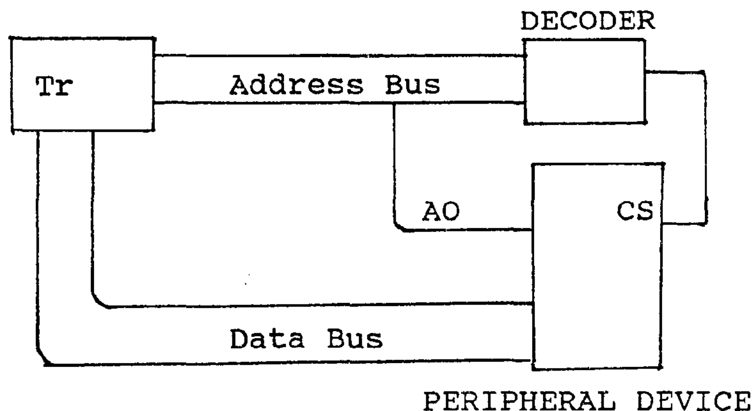
Stellenbosch University http://scholar.sun.ac.za

4. PERIPHERAL INTERFACING

4.1 Memory mapped interfacing

The most common method of interfacing to any microprocessor.

Architecture



Constraints on data bandwidth

The main constraint is the time a basic loop iteration takes to service the external device. The OCCAM code for this basic loop is:

```
i := 0
WHILE i < length                                    ]A
SEQ
  WHILE (status /\ 1) <> 1                          ] 750ns
    SKIP
  buffer [i] := port1                               ] +A = 1,1µs
```

i.e. max expected BW = 540 kB/s
-> assuming 8 bit peripheral interface.

For a 16 bit peripheral interface the expected reslults are as follow:

Stellenbosch University http://scholar.sun.ac.za

<u>Algorithm</u>

```
i := 0
WHILE i < length
SEQ
  WHILE (status /\ 1) <>1                                  ] P
    SKIP
  buftmp := port2 /\ #FFFF                                 ] R1 +250ns + 200ns
  WHILE (status /\ 1) <>1                                  ] P
    SKIP
  buf[i] := buftmp \/ (port2 << 16)                        ] R2 + S + 200ns
  i := i + 1
```

Total execution time $P = 750ns \times 2$ $(600ns \times 2)$

$R_2 = 900ns$

$R_1 = 500ns$

$S = 800ns$

$250ns$

$4,35\mu s$

i.e. expected BW 229 kW/s (W = 4 bytes)

or 460 kE/s (E = 2 bytes)

or 919 kB/s (B = 1 byte)

For a 32 bit peripheral interface the results for one byte peripheral interface can be multiplied by a factor of 4.

i.e. expected BW 540 kB/s x 4 = 2,1 MB/s

But this BW is lower than can be achieved. For word transfers the program time delays are:

polling : 600ns

inputting: 850ns

loop : -

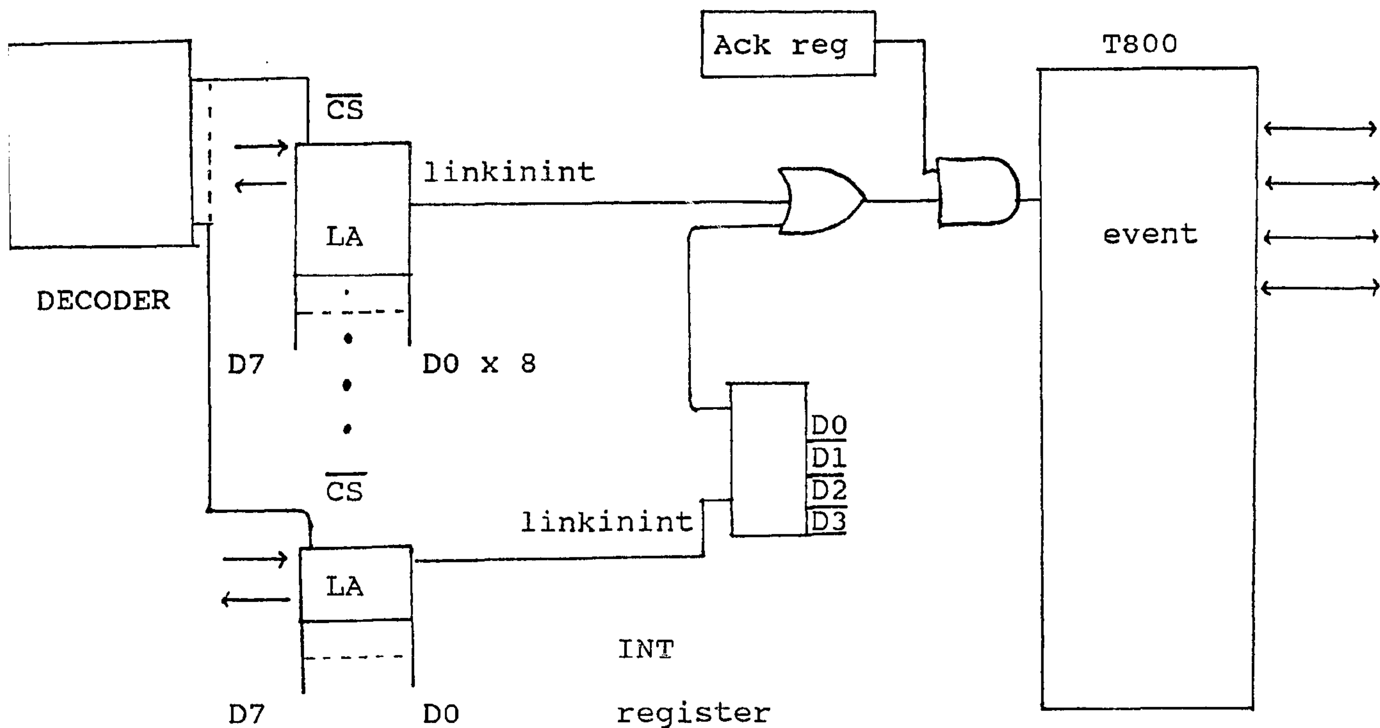
which gives a BW of 689 KW/s = 2,7 MB/s

Stellenbosch University http://scholar.sun.ac.za

Implementations

The implementation considered under memory mapped I/O is the so called TR8 and AILTR8 card.

Architecture



-byte interface to transputer

-expected throughput ≤ 540 kB/s

-measured throughput (@ 10 mBit/sec link speed)

= 384 KB/s

30% slower than expected

Discussion of memory mapped interfacing

1. From expected results; less than 2x gain in going from 8bit to 16bit interface. (540 kB/s -> 919 kB/s)

2. Practical implementation of 8 bit system 30% slower than expected. One possible explanation is status not ready when polled.

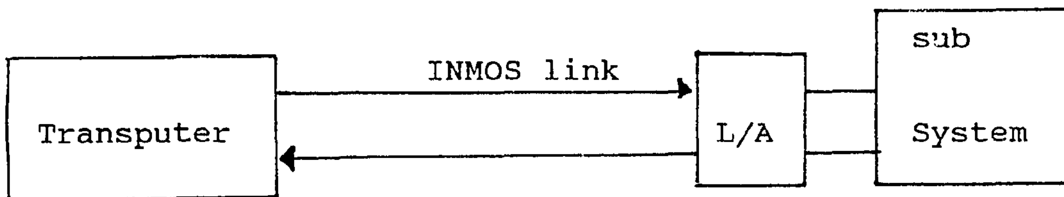
3. From expected results; significant gain from 8 bit -> 32 bit bus.

Stellenbosch University http://scholar.sun.ac.za

4.2 Interfacing with linkadapters

Use "high" speed links to communicate with external interface. Gain advantage that subsystem completely isolated from Tr.

Architecture



The subsystem can either be intelligent (include a μP) or dum.

Constraints on data bandwidth

The high speed link places a upper bound of 1,8 MByte/s (at a link speed of 20 MBit/s) on the data bandwidth.

In most applications the sub system (if intelligent) will determine the data bandwidth.

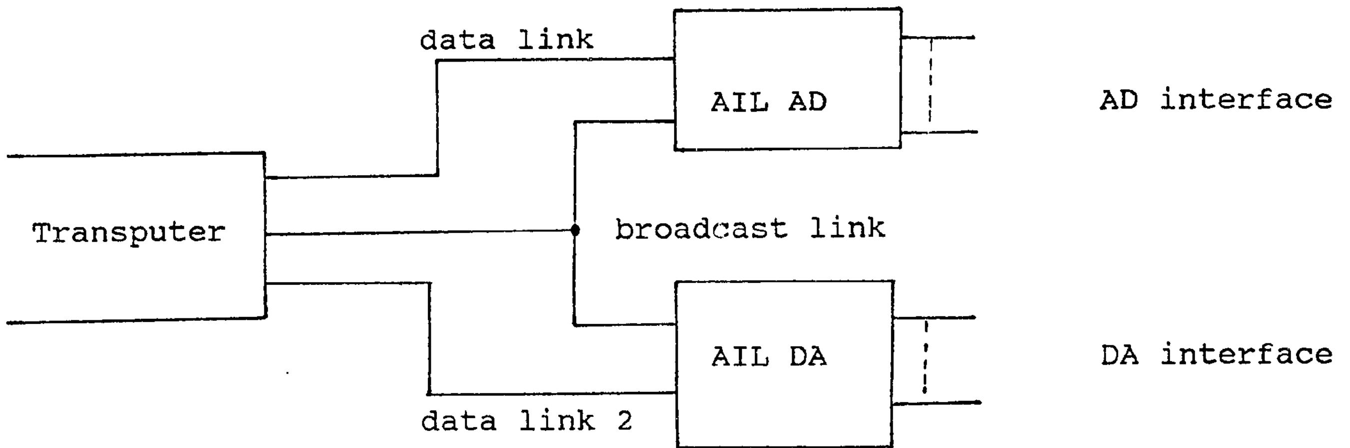
The application considered is one of the last mentioned. Thus the application will be considered first before expected results will be predicted.

Stellenbosch University http://scholar.sun.ac.za

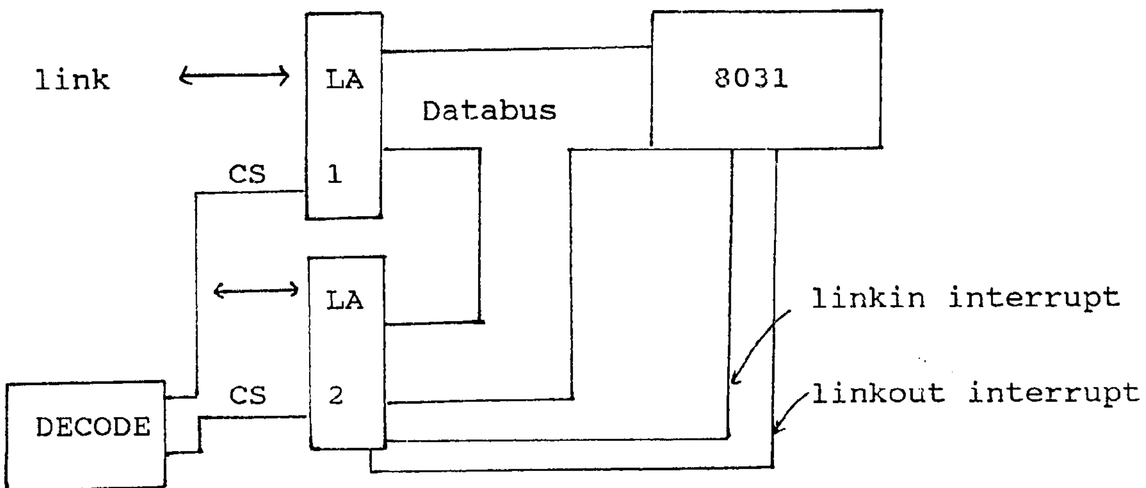
Implementation

The HIL (Hardware in Loop also called AIL) system will be considered as an example.

Architecture



More detailed architecture of AILcard



Implementation constraints on data bandwidth

For this specific implementation, the 8031 is definitly the determiner of the upper data bandwidth bound.

Examining the basic program will reveal the time constraints.

Stellenbosch University http://scholar.sun.ac.za

Algorithm for transmitting message of length L

```
         mov R0,msg-ptr            1 )
         mov R1, L                 1 }  4µs  message initialisation
         mov dptr,#la1-out         2 )
repeat:  mov a,@R0                 1 )
w1:      jnb la1outint,w1          2 |
         mov @dptr,a               2 }  8µs  iteration executed once
         inc R0                    1 |       for each byte to be
         djnz R1, repeat           2 )       transmitted
```

Provided that transputer is synchronised to recieve this message, the data throughput can be calculated as:

to send 1 byte: $t = 12\mu s$ BW = 83 KBytes/sec

255 bytes: $t = \frac{4+8x256}{256}$

$= 8\mu s$ BW = 125 KBytes/sec

Measured throughput

The throughput is measured not only in terms of communication BW, but also includes the peripheral servicing by the 8031.

The throughput of the AIL cards are 8kE/s = 16KB/s.

Discussion on link interfaced peripherals

1. Advantage gained in decoupled system with subsystem having a true transputer face.

2. Not "very high" data bandwidth capability, but more than adequate for 8KHz (16 bit) transfer rate.

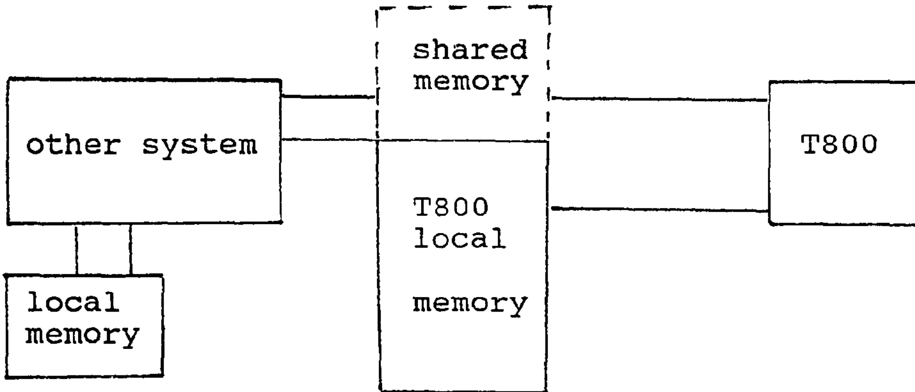
3. AILcards, because of link interface, can be connected easily to <u>any</u> transputer system - flexibility.

Stellenbosch University http://scholar.sun.ac.za

4.3 Memory swapping

Essence is external interface sharing a common block of memory with the transputer.

Architecture



Constraints on data bandwidth

The only constraint the tranputer is subject to is the time the other system takes to fill the shared memory. The transputer (T800) can then blockmove the data from shared memory to own memory or use the memory still in the shared memory block.

Blockmove application

Time to move block 8+2 * external memory cycle x N

= 8+300 ns x N

where N is no. of 32 bit words to move

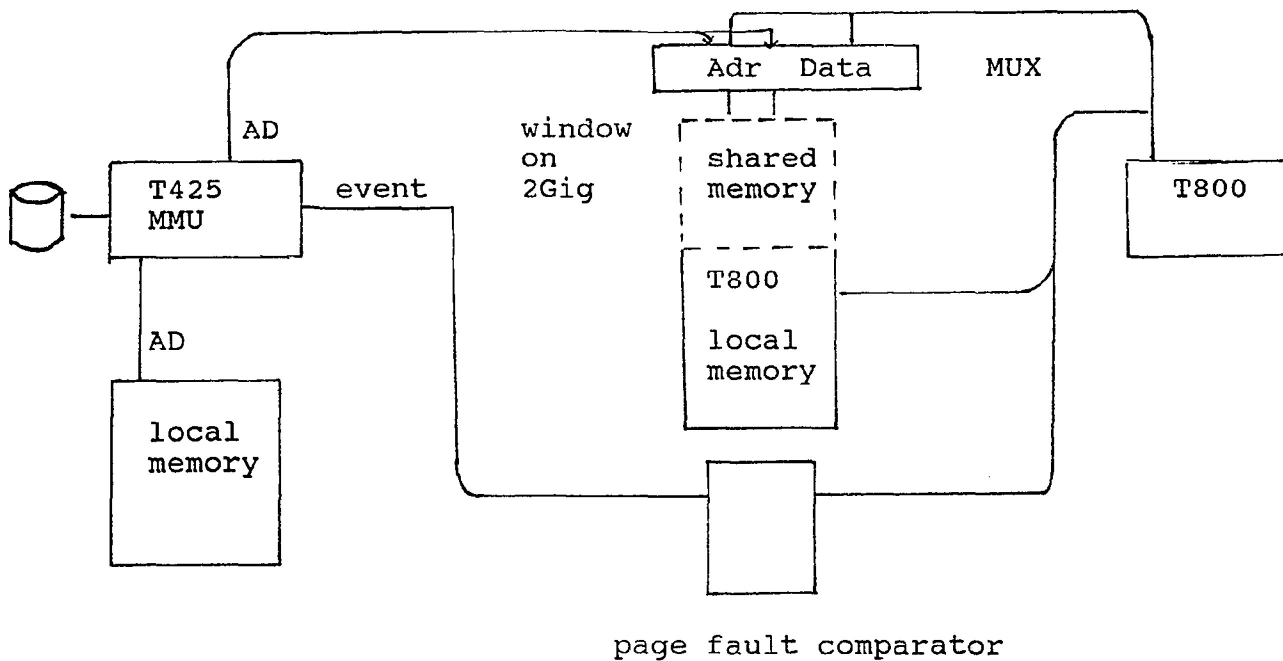
i.e. data bandwidth = 3,3 MW/s = 13 MB/s

Stellenbosch University http://scholar.sun.ac.za

Implementation

The implementation considered under shared memory interfacing is the Transputer virtual memory system.

Architecture



page fault comparator

- 32 bit interface to T800
- data used from shared memory - no time delay
- only time delay associated with transfer of data from disk to shared memory.

Time constraints associated with virtual memory system

- page fault with page on disk - few ms.
- page fault with page in local memory of MMU - few µs
- "page fault" with page in window.

  version 1 - one active page - 10 µs (measured)

  version 2 - 16 active pages - 0s !!!

Stellenbosch University http://scholar.sun.ac.za

4.4 DMA interfacing

- "pure" DMA interfacing to T800;

  BW dictated by MEMREQ, MEMgranted timing.

  BW = 1,3 MHz

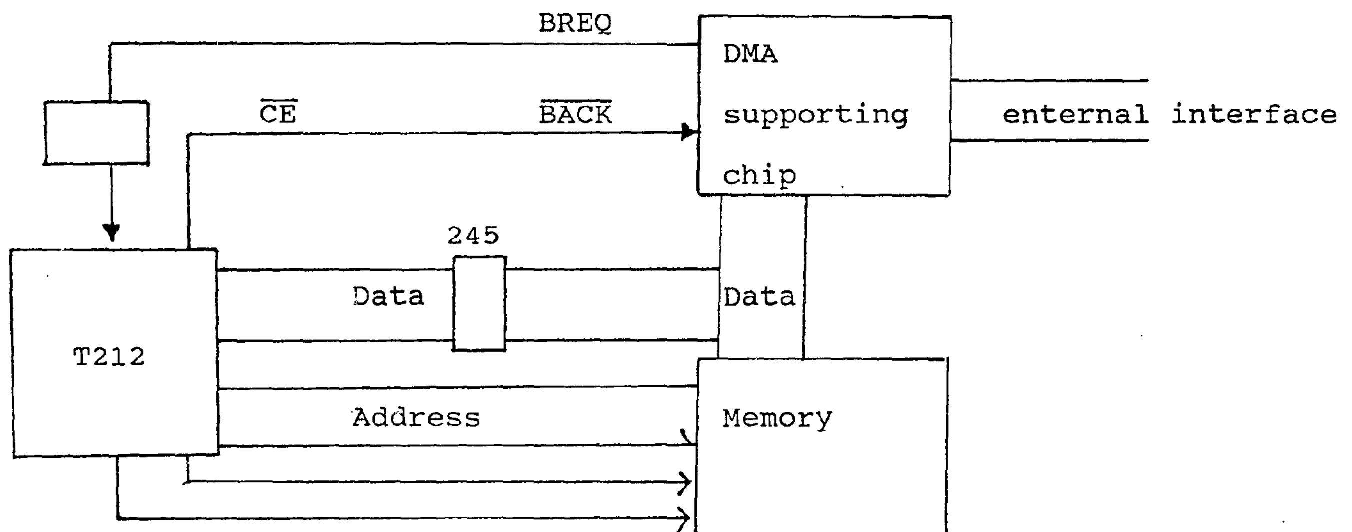
There are also other methods for interfacing DMA supporting chips to the transputer. These other methods are only viable now since a fast processor is available.

Three methods will be evaluated on prediction only.

4.4.1 Implémentation 1

transputer takes over all functions of DMA controller

Architecture



Algorithm

```
i: = 0
WHILE i < length
SEQ
  WHILE (breq /\1)<>1
    SKIP
  buf[i]: = constant
  i: = i+1
```

Assumptions

- i resides in internal memory

Stellenbosch University http://scholar.sun.ac.za

Constraints on data bandwidth

Assuming DMA supporting chip has higher BW than transputer system. Then the upperbound on data BW is the transputer. Further on 8 bit databus is used.

The algorithm exactly the same as for polled interfacing in memory space ! One major difference.

Data is transported directly from and to memory space and I/O device! Saving one external reference per entity.

Thus expected data BW = 2x polled memory mapped system

= 2x 540 kB/s

= 1,08 MB/s

On closer inspection algorithm reveals that loop control stays the same.

Thus more realistic expected BW = 1/1,65µs

= 0,6MB/s

Snags (on a T800)

- 8 bit and 16 bit interface will jump 75% and 50% of the memory space which is wasted.

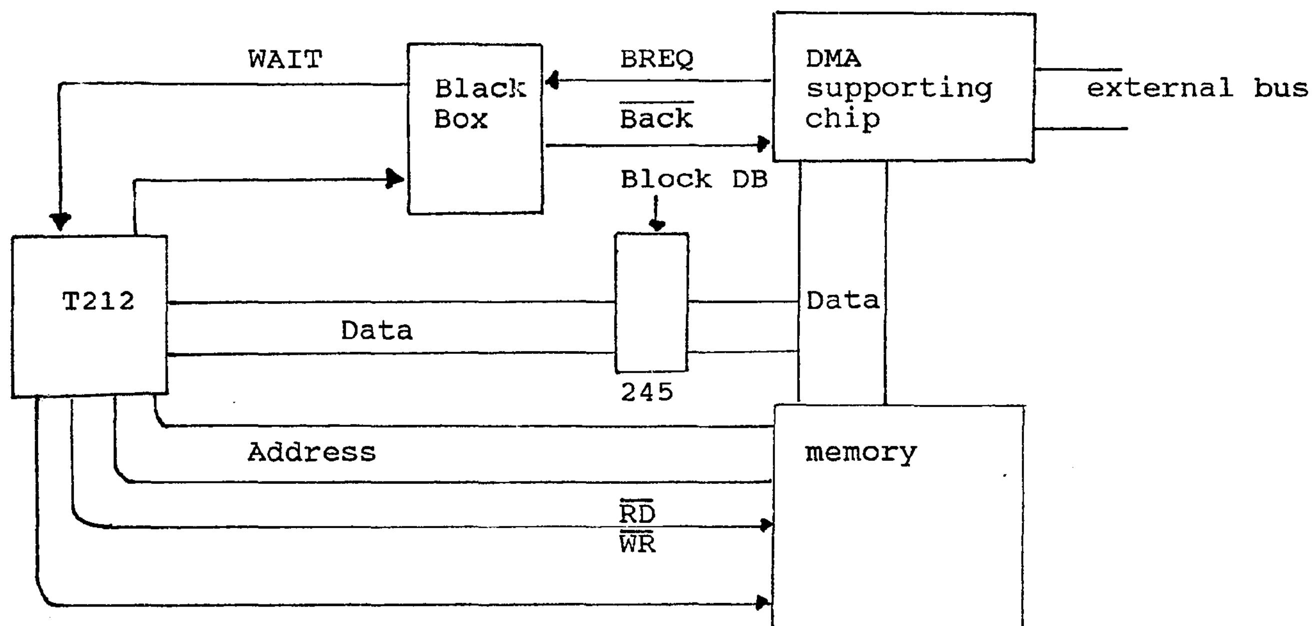
- The above problem can only be solved with complicated H/W.

Stellenbosch University http://scholar.sun.ac.za

4.4.2 Implementation 2

Transputer only does address generation. Signal sensing and generation is done in H/W.

Architecture



Algorithm

```
block.DB  := disable
SEQ i = 0 for N                      --  from external bus to memory
  buf [i] := dummy = const.
block.DB  :=enable


--   block.DB  := disable
SEQ i = 0 for N                      --  from memory to external bus
  dummy   := buf[i]
--   block.DB  := enable
```

Constraints on data BW

- Transputer still determines upperbound.
- However the polling of BREQ is no longer necessary.

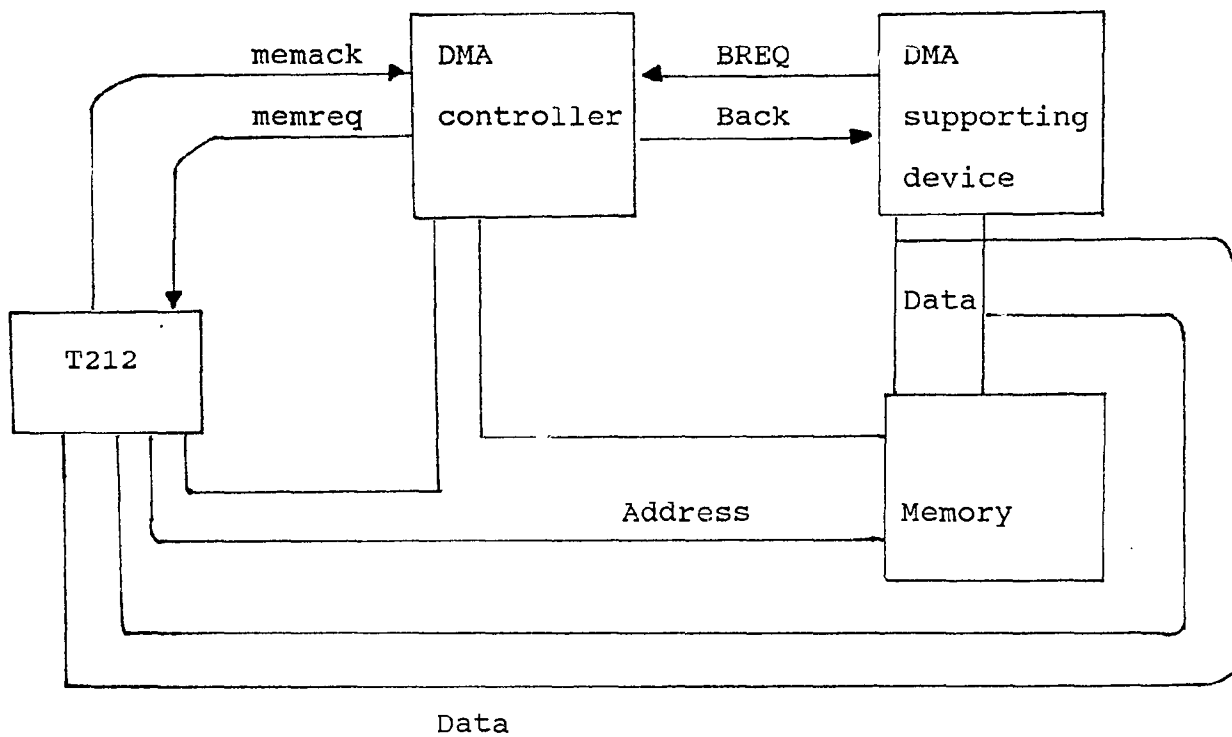
Stellenbosch University http://scholar.sun.ac.za

	T212	T800 (waste mem approach)
Thus expected data transfer BW =	900ns pw	900ns
	1,1 MW/s	1,1 Mentity/s
	2,2 MB/s	2,2 MB/s (16bit entities

4.4.3 Implementation 3

conventional DMA coprocessor

Architecture



Memreq puts T212 bus in tristate; DMA controller takes control of Adr bus; DMA supporting device takes control of Data bus.

Algorithm

```
-- pseudo language
setup DMA controller
wait for it to finish (possibly doing something else)
```

Stellenbosch University http://scholar.sun.ac.za

Constraints on data Bandwidth

- hardware constraints

- already mentioned as 5,3 MB/s
  (assuming 32 bit interface)
  with normal byte wide devices. 1,3 MB/s

4.5 Discrete DMA interface

Highest possible bandwidth. DMA controller kept SIMPLE and built up from "discrete components."

Main assumption for DDMA systems:
Data can be transferred faster if the transputer is completely left out of the transfer!

Thus it can be expected that a data transfer BW can be achieved striving to the memory BW in the limit.

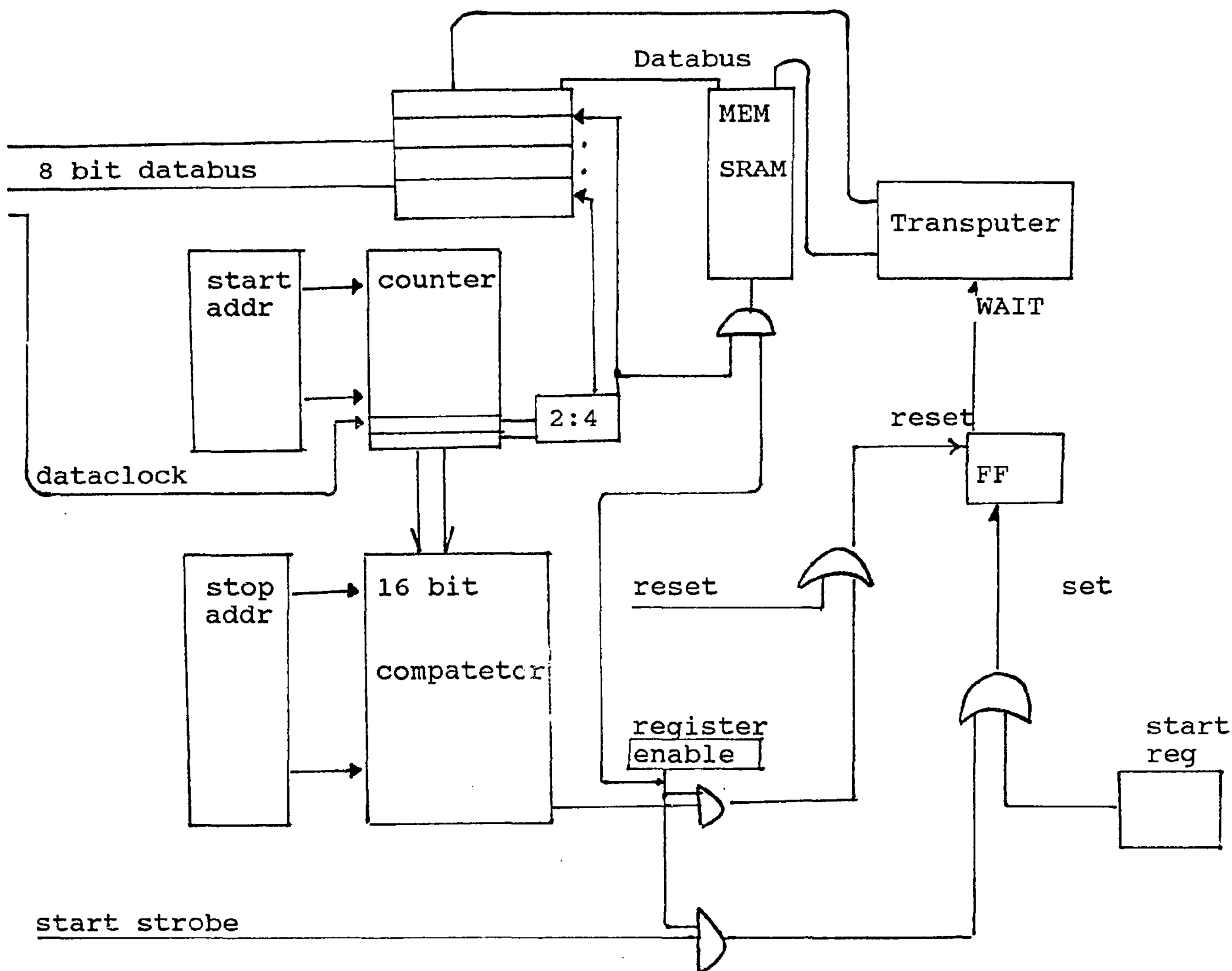
4.5.1 Peak transfer then post processing

Implementation

High speed data capture from a radar source for post mortum evaluation with MATLAB.

Stellenbosch University http://scholar.sun.ac.za

Architecture



Constraints on data BW

MEMORY BW ONLY CONSTRAINT !

Studying the example further we can extract a few numbers to compare with previous implementations.

Assume "slow" SRAM with 100ns access time.
This memory can be accessed within 125 ns.

Thus the BW into memory is 8 MW/s
= 32 MB/s

During this time the transputer is switched off with the WAIT input.

Stellenbosch University http://scholar.sun.ac.za

Algorithm

-- pseudo code

setup registers

enable data transfer

(wait until finished)

process data

4.5.2 Peak transfer while processing

DDMA as discussed so far will function only at top speed for short periods of time.

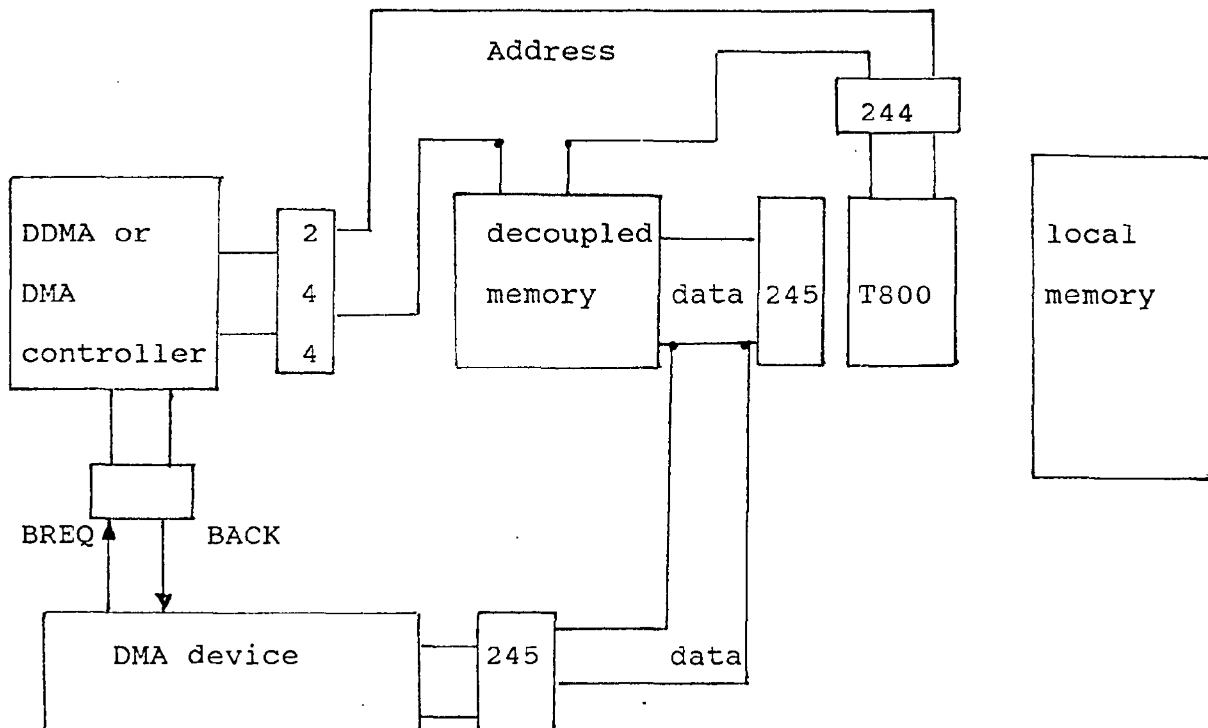
What about real time applications where sustained data transfer rates of 5MB/s or more must be achieved?

Solution lies in divide and conquer.

Decoupled buffer technique

The basic idea is to decouple the processor from the high speed input/output, but not to switch the transputer off.

Architecture



Stellenbosch University http://scholar.sun.ac.za

This has the advantage that while data is being transfered to and from the buffer the transputer can continue with other tasks.

The above system will work in the case where the next data location cannot be predicted either in time or place.

For a continious data stream at least a double buffer is needed.

5. Concluding remarks

We have considered four basic methods of interfacing I/O systems to transputer networks.

Memory mapping

- conventional, closely coupled systems

- program control

- BW: 540 kB/s (8 bit) - 919 kB/s (16 bit) - 2,7 MB/s (32 bit)

Link interfacing

- transputer compatible decoupled systems

- completely OCCAM compatible

- BW: 83 kBytes/s - 125 kByte/s practical 16 kB/s -> 1,8 MBytes/s

Mem to MEM interfacing

- shared memory, allowing processing while transferring data

-BW: (transputer subsystem determines)

DMA interfacing

- direct access to memory databus

- considered 4 implementations with varying amouts of processor interference

- BW: 0,6 MB/s ; 1,1 MB/s ; 1,3 MB/s ; 32 MB/s (assuming 8 bit subsystem)