171

UNIVERSITY OF LONDON

IMPERIAL COLLEGE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTING AND CONTROL

A COMPARISON OF THE CRAY-1 AND CDC-7600 COMPUTER SYSTEMS

BY

[Ing.] H.C. van der Wijk

A thesis presented in fulfilment of the requirements for the award of the Mphil degree of the University of London, 1981.

CONTENTS

Abstract 7

Acknowledgements 8

Chapter 1 Introduction

1.1 The need for enhanced facilities at U.L.C.C
- 1.1.1 U.L.C.C 9
- 1.1.2 The workload 9
- 1.1.3 The requirements of a new system 11

1.2 The alternative architectures to be considered
- 1.2.1 Front end processor and rear end processor 12
- 1.2.2 High speed computer architectures 12
- 1.2.3 Multi processor 14
- 1.2.4 Array processor 16
- 1.2.5 Pipelined processor 17

1.3 The approach towards selection
- 1.3.1 The selection procedure 18
- 1.3.2 The performance evaluation test 19
  - alternative selection techniques 19
  - instruction mixes 19
  - kernels 20
  - live benchmark 20
  - artificial benchmark 21

1.4 About the project 22

Chapter 2 Comparison at component level

2.1 Introduction 24

2.2 Comparison at storage level
- 2.2.1 Organisational differences 25
- 2.2.2 Data Copy instructions 26

2.3 Comparison at instruction level
- 2.3.1 Instruction sets 28
- 2.3.2 Instruction timings 29

2.4 Pipelining 31

2.5 Vector processing
- 2.5.1 Definition of vector processing 33
- 2.5.2 Advantages and disadvantages of vector processing 36
- 2.5.3 Effect of scalars and short vectors on performance 37
- 2.5.4 Vector processing on the Cray-1 37
- 2.5.5 Vector processing on the C.D.C-7600 39

2.6 Comparison at kernel level, the Los Alamos test
- 2.6.1 Objectives of the test 41
- 2.6.2 The kernels 42
- 2.6.3 The results 43
- 2.6.4 Application of the results to the U.L.C.C workload 44

2.7 Conclusions 45

Chapter 3 Comparison at Benchmark level

3.1 Summary of the benchmarking process 46

3.2 Considerations for the optimal construction of a Benchmark 47

3.3 The forming of the Test Set 48

3.4 The conversion of the Test Set
- 3.4.1 The rules 48
- 3.4.2 The results 48
- 3.4.3 Conclusions 50

3.5 The running of the Test Set
- 3.5.1 The rules 51
- 3.5.2 The software and hardware used 52
- 3.5.3 The results 53
  - throughput ratios 53
  - individual program ratios 53
- 3.5.4 Conclusions 54

3.6 Limitations of Benchmarking
- 3.6.1 Modelling the future workload 55
- 3.6.2 Size of the Test Set 55
- 3.6.3 Representativeness of the Test Set 56
- 3.6.4 Optimising the Test Set 56
- 3.6.5 Hidden problems 57
- 3.6.6 Costs of the Benchmark 57

3.7 Conclusions 58

Chapter 4 Analysis of the Benchmark results

4.1 Introduction 59

4.2 Approach to the Analysis
- 4.2.1 Differences in architecture 59
- 4.2.2 Differences in compilers 60
- 4.2.3 Differences in packages and software tools 62

4.3 Analysis of the C.D.C-7600 results 63
- 4.3.1 M.N.F versus F.T.N(2) 64
- 4.3.2 F.T.N(0,1) versus F.T.N(2) 65
- 4.3.3 Better memory management 66
- 4.3.4 Double precision arithmetic 67

4.4 Analysis of the Cray-1 results 68
- 4.4.1 The effectiveness of automatic vectorisation 69
- 4.4.2 Double precision arithmetic 70
- 4.4.3 Integer arithmetic 70
- 4.4.4 Faster discs 70

4.5 Comparison of the Cray-1 and the C.D.C-7600 71

4.6 Conclusions 72

Chapter 5 Vectorising a subset of the Test Set

5.1 Introduction 74

5.2 Optimisation 75

5.3 Automatic vectorisation

5.3.1 Automatic and non automatic vectorisation 76

5.3.2 The C.F.T compiler 77

5.3.3 Conditions preventing vectorisation 78

5.3.4 Restructuring techniques 80

5.4 Vectorising a subset of the Test Set by restructuring

5.4.1 Hardware and software used 81

5.4.2 The results 82

5.4.3 Problems encountered 84

5.5 Conclusions 86

Chapter 6 Conclusions

6.1 Investigations done 87

6.2 Major conclusions of the project 91

6.3 Suggestions for further investigation 93

Appendix 1 The Cray-1 and C.D.C-7600 architectures 94

Appendix 2 Fortran conversion problems 113

Appendix 3 Storage management for the C.D.C-7600 117

Appendix 4 Restructuring to help vectorisation 122

Appendix 5 The restructured programs 129

Tables 133

Cray-1 Bibliography 141

References 145

ABSTRACT

The purpose of this project is to compare the Cray-1 and the C.D.C-7600 for the processing of a general purpose scientific workload. We compare the two machines at hardware level, at Fortran level and at optimised Fortran level.

The first investigation is to compare the two machines at component level. We conclude that the scalar performance of the Cray-1 is more than twice a C.D.C-7600. Further research is required since this comparison at hardware level does not include the efficiency of the Fortran compilers.

A live benchmark is used to include compiler efficiency in the comparison of the two machines, and to find out how much effort is required to convert programs from the 7600 to the Cray-1. The benchmark test set was formed by 68 programs taken randomly from the U.L.C.C workload. The benchmark suggests a performance (in terms of throughput) for the Cray-1 of 1.65 times a C.D.C-7600. The conversion of programs gave no serious problems. The fluctuations in the performance ratios of the individual programs in the benchmark were very high. Therefore a subset of the benchmark test set is analysed, and the fluctuations are explained.

A subset of the benchmark test set is optimised by restructuring the Fortran source code to help the automatic vectorisation by the Cray Fortran compiler. The performance of the Cray-1 for the optimised subset is 3 times the C.D.C-7600.

ACKNOWLEDGEMENTS

This project was done at the University of London Computer Centre, and partly at the Science Research Council Nuclear Physics Laboratory in Daresbury.

First of all I want to thank Prof. D. Howarth and Dr. R Baker for their comments and advice.

I also want to thank C. Lazou (U.L.C.C), G. Ashley (Cray Research), C. Henfreys (Cray Research) and C. Amorum (Imperial College) for their advice and assistance.

I want to thank U.L.C.C and Cray Research U.K. for the assistance and the computer time they provided.

The project was supervised by Prof. D. Howarth from Imperial College.

CHAPTER 1

INTRODUCTION

1.1 The need for enhanced facilities at U.L.C.C.

1.1.1 U.L.C.C.

The University of London Computer Centre provides a computing service to research staff in Schools, Institutes and Colleges of the University of London and thirteen other Universities in Southern England and Wales. The service is intended for users who want to run medium and large jobs, that are too big or use resources not available at their local installation, as well as providing services to those institutions without their own facilities.

The computer installation at U.L.C.C. consists of a C.D.C-7600 supported by C.D.C-6600, C.D.C.6400 and Cyber72. A communications network provides access to the system.

1.1.2 The workload

The current workload consists of about 3500 Jobs processed on the machines each day. About 1500 jobs per day are processed by the C.D.C-7600 computer. Most of the C.D.C-7600 jobs are written in Fortran. The 7600 jobs are dominated by floating point arithmetic on array elements inside program loops. The workload can be described as a 'general purpose scientific workload'.

The U.L.C.C computing resources currently available are almost fully utilised. Demand is expected to continue to grow and to provide a satisfactory service for at least the next five years equipment which is substantially more powerful than the C.D.C-7600 is needed. In fact a computing power of at least twice a C.D.C-7600 will be necessary. In addition a substantial larger amount of interactive terminals will have to be supported. Several of the large users find themselves severely restricted in their research by limited computing resources. Many of the problems of these users involve manipulating arrays and matrices. At the moment several architectures are on the market that can process arrays and matrices in a very cost effective way. However not the entire workload of U.L.C.C consists of array processing, it also includes scalar processing. So the machine required must be a good scalar processor and be capable of processing a substantial amount of 'vector processing power' in addition to that.

Traditionally the emphasis of the University Computer Centres has been on providing large amounts of computing power. However the data-processing aspect is becoming more important all the time and it will be necessary to extend the facilities with e.g. a data base system in this area.

1.1.3 The requirements of a new system

To provide a satisfactory service for the next five years a new system must meet the following requirements:

The machine must be cost-effective for the processing of the expected workload. The program conversion effort necessary to move programs from the current installation to the new installation must be acceptable. The maintenance costs of the new system must be at an acceptable level; modern architectures tend to have lower maintenance costs than older machines. The new machine must be very reliable. The new machine must have sufficient software to go with it. Definitely there must be a Fortran Compiler, Operating System and a Scheduler. Preferably there should also be other compilers e.g. a Pascal Compiler and most of the packages commonly used at U.L.C.C e.g. N.A.G-library, SHELX, XRAY etc. The new machine must be able to process a workload of about twice the current C.D.C-7600 workload. The of floor space taken up by the new machine must be compatible with that available in the new building; this limitation may rule out some of the multiprocessor candidates. The new system must be able to support a multi access system with up to about 100 interactive terminals. An advantage for the new machine would be to have some vector capability, to provide a more cost effective way for processing vector programs of many large users. The amount of vector processing power needed will be difficult to estimate, since the current workload has been written with a scalar machine in mind.

1.2 The alternative architectures to be considered

1.2.1 Front End Processor and Rear End Processor

In the previous paragraph we outlined the requirements of a new system. However at this moment of time there is no single processor that can meet all these requirements, therefore more than one processor is likely to be needed.

To be able to meet these requirements with existing processors we can split the workload in two sets of jobs. The first set consists of the interactive jobs, the program development jobs and all other short jobs. The second set consists of all the batch jobs that require large amounts of computing. The processor to process the first set of jobs is called the Front End Processor and the processor(s) for the second set is called the Rear End Processor.

From here on in this thesis we will only discuss the Rear End Part of the new system and therefore concentrate on high speed computing.

1.2.2 High speed computer architectures

The C.D.C-7600 is one of the most powerful computing systems available for the processing of a general purpose scientific workload. The machine is built up of fast electronic components, and a significant amount of parallelism is achieved by the pipelining of instructions.

Generally speaking, there are the following major techniques to speed up the computing process: The use of fast logic components, the implementation of often occurring program constructs in hardware, parallelism in processor architecture and memory access, and improvement of system and aplication software. In a typical high speed processor all these techniques will be applied.

The spectacular developments in micro electronics over the last thirty years have enabled a dramatic improvement in the speed of computers. Since the logic speeds are already approaching the speed of light, the ultimate speed, future developments will be of a much smaller order. When speeding up the processing unit, the access to mainstore and registers must also be speeded up.

When certain sequences of instructions occur very often, they can be implemented in hardware, and the instruction set extended, so that less instructions are required to perform this function. This aproach makes such a machine well suited to certain applications, examples are the stack machines and vector processors. In the case of a vector processor, this includes in the instruction set instructions to handle vectors, resulting in more efficient processing of program loops. In the case of a stack machine it includes in the instruction set instructions to make the runtime storage allocation for block structured languages like Pascal and Algol easier. On entering a procedure a group of variables, called a block, is placed on top of a stack of such blocks, and on exit from the procedure the block is removed. Access to the variables is made possible via a set of pointers, that is contained in the 'display registers'. On such 'Algol Machines' often the current block is placed in a cache store to speed up access to the local variables.

The next technique is to make use of parallelism in the computer architecture. To obtain a system with a processing power of e.g. twice a C.D.C-7600, one could simply take two machines of the power of a C.D.C-7600. The processors of such a multi-processor system can be completely separate or communicate with each other via Filestore, Backingstore or Mainstore. A possibly cheaper alternative to having several copies of entire processors, would be to copy only part of the processor. An example is the Array Processor, where one control unit takes care of multiple arithmetic units. Such a machine works on arrays of data. Another approach is pipelining. In a pipelined processor the instruction is segmented into several subprocesses which are executed by dedicated autonomous processing units. Successive instructions can be carried out in an overlapped mode. Vectors can be processed extremely well on a pipelined processor, therefore Vector Processors are pipelined. Parallelism is also relevant for memory access. On a lot of large systems the mainstore consists of a number of independently accessible banks (typically 8, 16 or 32).

The efficiency of system and application software is very important. For every problem to be solved we must try to use the numerically most efficient algorithm. If the programs are written in a high level language, the associated compiler should produce the most efficient object code possible. The overhead of the operating system should be as small as possible, and the jobstream must be scheduled in such a way to ensure a maximum throughput.

The workload at U.L.C.C is a general purpose scientific workload. Most of the programs are written in Fortran, therefore a stack machine seems less suitable. The majority of U.L.C.C processing time is spent performing floating point arithmetic on array elements inside program loops. Therefore the most promising architectures seem to be the pipelined (Vector) Processor, the Array Processor and the Multi Processor systems, which we will discuss in the following paragraphs.

1.2.3 Multi Processor

In this paragraph we look briefly at the multi processors. The processors in a Multi Processor system can be completely seperate or linked via Backingstore, Filestore or Mainstore.

The simplest form of a Multi Processor system is to have completely separate processors. Note that the allocation of a job to the system takes place at input time. An advantage of this method is that it is very easy to implement.

In case of a Twin Processor, each processor uses its own local store. The systems are coupled either at disc store level (e.g. the C.D.C.6600 and the C.D.C-6400 at U.L.C.C) or at backing store level. Job allocation takes place at load time. By backing store is meant slow core, e.g. Large Core Memory on the C.D.C.-7600.

In case of a Dual Processor, both the processors use a common store. Since both processors are competing for a single store the performance of such a Dual Processor is less than twice a single processor. In case of the I.C.L-1905 Dual Processor system the performance is 1.6 - 1.7 of a single processor. The advantage is that one copy of the very expensive mainstore is required. Job allocation takes place at runtime.

Possible disadvantages of such a Multi Processor system compared with a single processor of the same power are: Higher maintenance costs and more floorspace required. However some manufacturers stack processors on top of each other, so floorspace is then no problem, assuming your ceilings are high enough. Smaller processors tend to have a shorter wordlength e.g. 32 bits instead of 60 bits on the C.D.C-7600, therefore to obtain the same accuracy many programs must be run with double precision arithmetic which can degrade the performance of the system. When the number of digits in the exponent of the floating point representation is less in the Multi processor system than in the single processor a problem arises, because a smaller range of numbers can be accomodated during computation.

Possible advantages of a Multi Processor are: Since processors in such a system are often 'well established' conventional type architectures that cater for a much larger market than the often very specialised high speed single processors, better support tends to be given and more software is provided. To be advantageous the processors in a Multi Processor system should preferably be of the same manufacturer, if possible including the Front End processor. Another advantage is that a multi processor system can grow with the workload, this might result in less overcapacity in the beginning of the life of the system, when the system runs out of capacity an extra processor can be added with little or no problems.

1.2.4 Array Processors

A conventional processor operates on one pair of operands at one time. An Array Processor processes arrays of operand pairs at a time. If the processor has N arithmetic units, then N pairs of operands can be processed at one time.

Because one control unit takes care of multiple arithmetic units each arithmetic unit must do the same thing at the same time or else be inhibited or do nothing. Not more than one arithmetic unit must be allowed to modify the same word in memory at the same time. This can be achieved by allocating to each arithmetic unit an exclusive block of memory.

The advantage of Array Processors is that they are extremely good in manipulating arrays. However the scalar performance is generally poor, because scalars are processed in a serial way. Another equally important problem is the fact that a lot of programming effort is required to write programs that utilise the parallelism properly.

An Array Processor is useful when the total workload consists of a small number of codes, that perform array operations most of the time; in such a case these codes can be optimally handcoded in 'vector' form. In the case of the U.L.C.C. workload, which consists of 3000 different jobs a day, it is most unlikely that the majority of users are going to rewrite their programs. However it can be used as an 'add on facility' to a powerful scalar machine.

Some examples of Array Processors are: Illiac IV, I.C.L-DAP. For references see Parkinson77 and Mc Entire70.

1.2.5 Pipelined Processors

The instructions in a computer are processed by functional units. In a pipelined processor a functional unit is often referred to as a 'pipeline'. There are several different functional units in a computer, e.g. for floating point multiply, add, subtract, divide etc. The principle of pipelining is to recognise that execution of an instruction proceeds as a sequence of subprocesses. All subprocesses must complete in a fixed time slot, the time slot is typically one machine cycle for all subprocesses in all pipelines, however for some instructions e.g. divide this is difficult to achieve. Every subprocess is executed by a dedicated autonomous processing unit. Successive instructions can now be carried out in an overlapped mode. In a 'fully segmented pipeline' every subprocess is completed in exactly one clock cycle, thus if a continuous stream of instructions enters such a pipeline, a result is produced every cycle, after a startup period. The startup period is the time it takes to process one instruction, this is often referred to as functional unit time.

Unfortunately instructions cannot always be fully overlapped. Delays can occur because of several reasons, e.g. when the input required for one instruction is the result of a preceding instruction which has not completed yet, before a branch can be taken in a program all parallelism must have ended etc.

A major advantage of pipelining is that a significant amount of overlap in the processing of instructions can be achieved without the programmer having to rewrite his programs. However a significant improvement in the 'pipeline utilisation' can be obtained if the compiler rearranges the sequence of instructions, to avoid resource conflicts. Pipelined processors are very useful for processing a general purpose scientific workload.

The processing of vectors results in the maximum utilisation of the pipeline, so a result is produced every clock cycle. This explains why architectures like the Cray-1 and the Cyber-205 combine vector processing with pipelining; both these machines have a powerful set of vector instructions.

Examples of pipelined architectures are: Atlas,I.B.M-Stretch,Texas Instruments Ti-ASC, Cray-1, (C.D.C-6600), C.D.C-7600, Cyber205, Burroughs Pepe and Amdahl 470 V/6. For references see Ramamoorthy77.

1.3 The approach towards selection

1.3.1 The selection procedure

When it has been concluded that the existing system cannot meet the requirements necessary to provide a satisfactory service in the near future, action must be taken. At this stage there are two options: purchasing a new system or upgrading the existing one, if this is possible.

The first step in selecting a new system is specifying the operational requirements needed to provide a satisfactory service. A Performance Evaluation Test is often part of such a specification. This specification is sent to the manufacturers who are asked to draw up a proposal to meet those requirements.

The next step is the evaluation of the proposals submitted by the manufacturers including the results of the performance evaluation test.

The final step is the selection of the new system. The machine which meets the operational requirements and is most cost effective is likely to be selected, however political motives can also play a part. E.g. 'Buy British' even if there is a more cost effective 'foreign' machine available.

1.3.2 The performance evaluation test

Alternative selection techniques

From paragraph 1.2 we can conclude that most architectures are designed for specific applications. This is reflected by significant differences in the architectural features of the machines available. Therefore a performance evaluation should consist of a comparison of two machines for a model of the expected workload, and not entirely concentrate on differences in architecture, since these differences may be irrelevant for that particular workload.

There are many different techniques to model or represent the expected workload for performance evaluation. The terminology used in the literature on computer evaluation is often confusing and conflicting. The modelling technique definitions used in this thesis are partly based on those from Timmreck73. The most commonly used techniques are: Instruction Mixes, Kernels, Live Benchmark and Artificial Benchmark.

Instruction Mixes

An instruction mix is a set of machine instructions which are taken to be representative of a user's workload. By observing which instructions are used and how often they are used in a large number of programs in our workload during execution, we can model the workload. The advantage of this method is that it requires little effort and therefore it is cheap. A disadvantage is that it does not take into account parallelism that might exist, e.g. in a pipelined processor. Neither does it take into account the efficiency of software like compilers and operating systems. Therefore the method is inaccurate. The method is useful to get a rough impression of the power of a machine.

Kernels

This method involves a set of Kernels which are taken from the users workload, and are optimally handcoded in machine language on both the machines on which they are to be run. By Kernels are meant the parts of the programs which use up most of the processing time during execution.

This performance evaluation method is extremely useful if the workload to be modelled consists of a small number of different programs which use up all the processing time, especially when the Kernels of these programs are already handcoded in machine code. Hence the test set is an exact model of the expected workload.

In a general purpose computing environment the workload consists of thousands of different jobs, each with different kernels and the method is therefore not applicable.

Live Benchmark

In a live benchmark machines are compared by running a representative set of programs drawn directly from the current user's workload. These programs are arranged so that when run they act as a model of the expected workload. The programs in the test set should be drawn in a random way, and the number of programs in the test set must be large enough to make the set representative of the expected workload.

This method assumes that no dramatic changes occur to the profile of the expected workload. This is in many cases a fair assumption. However, different facilities provided by the new system, can effect the profile of the workload significantly. The extent of such a change is difficult to forecast, and therefore modelling of the expected workload becomes difficult.

This performance comparison method takes parallelism into account and also includes the effect of compiler efficiency and operating system efficiency. It is probably the best method available in the difficult case of general purpose computing, but the results should still be taken with caution.

Artificial Benchmark

In an Artificial Benchmark machines are compared by running a small number of programs, selected or specially written, to model different characteristics, that affect the performance in the expected workload. So the first stage in this proces is to determine which workload characteristics are likely to influence the performance on both machines. Examples of such characteristics are: Percentage of time taken up by floating point arithmetic, integer arithmetic, double precision arithmetic, number of disc accesses, tape accesses etc.

An advantage of this method is that the test set can easily be converted to run on many different systems since the size of the set is very small, and therefore much cheaper than a Live Benchmark.

A disadvantage is that the compiler effectiveness is difficult to assess by such a small set of programs. The most serious problem is that it is impossible to prove that all important characteristics of the expected workload have been represented in the test set. The method is in practice far less accurate than a live benchmark.

1.4 About the project

The project described in this thesis is a performance comparison of the Cray-1 and the C.D.C-7600 for the workload expected at U.L.C.C. At the present time the Operational Requirements of the new system have not yet been specified, therefore the comparison in this thesis is not to be interpreted as an official performance test, but simply as a practical investigation.

The reason the Cray-1 has been chosen for this comparison is that the general purpose scientific workload at U.L.C.C which is partly Scalar and partly Vector, seems very well suited to a pipelined architecture like the Cray-1. Since no investigation known to us has taken place into the feasibility of using a machine with extensive vector capabilities for processing a general purpose scientific workload, the project is also an interesting research project.

The first stage in the project is a comparison of the Cray-1 and the C.D.C-7600 at component level. The purpose of this investgation is to establish if the Cray-1 hardware is possibly capable of meeting our requirements and to study the architectural features and differences of the two machines.

The next stage is the construction and running of a Live Benchmark. The modelling of the expected workload at U.L.C.C is very difficult because the profile of the workload is likely to change if a vector machine like the Cray-1 was introduced. It is hoped that many of the large users would restructure or,even better, rewrite their programs to make effective use of the vector capabilities to speed up their programs; unfortunately we do not know how extensive this change would be. Therefore the Live Benchmark which is based on the current workload will give a performance measurement corresponding to the case in which none of the users optimise their programs, this is the most pessimistic estimate that can be achieved and in that sense is very significant.

A second test should be done to investigate what the effect would be if the programs in the Benchmark set were restructured, so only partly rewritten, to suit the Cray-1 and the C.D.C-7600 better. This will give us the performance ratio when all users are prepared to put in a little bit of effort to optimise their programs.

Finally one should test what the effect would be if the programs in the Benchmark set were entirely rewritten, with the two architectures in mind. In practise this would be too time consuming and because extensive knowledge about the physics of the problems represented in the programs is required, which is often not available.

From this we can conclude that it is impossible to give a single figure or possibly even a narrow range of figures for the performance ratio of the Cray-1 and the C.D.C-7600. The performance that will be realised if a Cray-1 or an other vector processor would be installed is going to depend largely on the cooperation of the users. However this study will give us a clear idea of what is possible.

CHAPTER 2

COMPARISON AT COMPONENT LEVEL

2.1 Introduction

In this chapter we compare the Cray-1 and the C.D.C-7600 at component level. We concentrate on those components and features that are likely to influence the throughput times for the processing of a general purpose scientific workload. The features we concentrate on are: the execution times of integer and floating point arithmetic, how fast the operands can be delivered from memory and registers, and finally pipelining and vector processing.

The Cray-1 and the C.D.C-7600 have both been designed by the same architect, Seymour Cray. Both machines have been designed for a general purpose workload consisting of scientific programs requiring large amounts of computing. The Cray-1 and the C.D.C-7600 are both pipelined. The Cray-1 instruction set includes instructions to manipulate vectors. Appendix 1, contains a detailed description of both machines.

At the end of this chapter is a very brief summary of an extensive comparison of the two machines for a scientific workload at Los Alamos. The test was done in 1976. The reason this summary has been included is because it is the most extensive comparison of the two machines that we know of. The full text of the Los Alamos test is found in Keller76. The work at Los Alamos was done on the Cray-1 serial 1, this machine is currently installed at Daresbury Laboratory and was used for all the practical work in this thesis.

2.2 Comparison at storage level

2.2.1 Organisational differences

The Cray-1 has a large main store consisting of up to 1 million words of bipolar memory organised in 16 banks, every word being 64 bits. The 7600 has a Small Core Memory of 32k or 64K words organised in 16 indepently accessible banks, each word being 60 bits. Since the S.C.M is rather small there is also a 8-bank linear access store called Large Core Memory (L.C.M) of 256K or 512K words. The memory sizes given are decimal numbers and not octal.

The Cray-1 has a large amount of register store, about 4K bytes. All arithmetic is done on the V,S and A registers, sometimes referred to as the primary registers. There are 8 V(ector) registers, each V register consists of 64 words of 64 bits. There are 8 S(calar) registers of 64 bits. As an intermediate store between the S registers and Memory there are 64 T registers of 64 bits each. The T registers are used to contain intermediate results and local variables to reduce access time to memory. The S and A registers can also be loaded directly from memory. There are also 64 intermediate B registers of 24 bits for the A registers. The A registers are used to contain addresses and counters.

The C.D.C-7600 has much less register store. There are eight X registers of 60 bits for scalar processing. There are also eight A registers of 18 bits. The loading of A7 and A6 with a memory address results in the contents of the corresponding X6 or X7 register being stored at that address in memory. In the same way A1 till A5 are used for loading data from memory into corresponding X registers, hence since these are only 5 registers no more than 5 memory fetches can be done in parallel. There are eight index registers B0 till B7 of 18 bits long. Since the B registers on the C.D.C-7600 and the A registers on the Cray-1 are used for loopcounters, the maximum loopcounter on the Cray-1 is bigger since there are six extra bits available.

The instruction stack on the C.D.C-7600 is also much smaller than on the Cray-1. The instruction stack on the 7600 is a group of twelve 60 bit registers. The instruction stack on the Cray-1 consists of 4 buffers each of which holds 64 consecutive 16 bit instruction parcels. The stack on the Cray-1 is about five times as big as the C.D.C-7600 stack.

The code density is much higher on the Cray-1 than on the C.D.C-7600. This is because in the case of two parcel instructions the second parcel can be in the next word. On the CDC-7600 it is also not possible to jump to a parcel address inside a word; the Cray-1 allows this. The result on the Cray-1 is much less need for No-Operation instructions, and therefore a more compact code. This is also useful in the instruction stack since more instructions can be contained in the stack.

On the 7600 blocks of data can only be moved between the Large Core Memory and the Small Core Memory. The Cray-1 has instructions to move blocks of data between memory and the T,B or V registers at very high speeds.

Data copy execution times

The following table shows the execution times for the most important data transfer instructions on the Cray-1 and the C.D.C-7600.

For all instructions there are special conditions where different execution times apply, for details see the Cray-1 hardware manual and the C.D.C-7600 hardware manual. Examples of such special conditions are: bank conflicts, instruction in two different instruction buffers or instruction not in instruction stack.

The instruction for moving data between X registers and L.C.M on the 7600 has two different timings. This is because access to L.C.M results in an eight word register being filled called Superword, this takes 17*27.5 nano seconds; if the next reference to L.C.M is in the SWORD register the access time is now only 3*27.5 nano seconds. Therefore sequential access results on average in an access time of 5 cycles. In practise this is not so easy to achieve. Appendix 3 discusses in more detail how to access L.C.M and S.C.M optimally.

The N parameter used in case of block transfers, indicates the number of words to be moved.

The C.D.C-7600

Transfer	Cycles	Nano Seconds
SCM → X	8	220
X ⟶ SCM	1	27.5
X ⟷ X	2	55
X ⟶ LCM	3	82.5
LCM → X	3 or 17	82.5 or 467.5
LCM → SCM	16+N	440+N*27.5
SCM → LCM	12+N	330+N*27.5

The Cray-1

Transfer	Cycles	Nano Seconds
Mem → T	14+N	175+N*12.5
T ⟶ Mem	6+N	75+N*12.5
S ⟶ Mem	1	12.5
Mem → S	11	137.5
Mem → V	9+N	112.5+N*12.5
V ⟶ Mem	N	N*12.5
T ⟷ S	1	12.5
S ⟷ S	1	12.5

2.3 Comparison at instruction level

2.3.1 Instruction sets

On the Cray-1 functional units are provided for 24 bit integer addition and multiplication. There is also a functional unit for adding 64 bit integers. Multiplication and division of 64 bit integers is done in software. The multiplication of 48 bit integers on the 7600 is done in hardware, and 48 bit integer division done in software, but less instructions are required than on the Cray-1 because use can be made of the double precision result of the floating point unit. In the case of integer division operands are packed with exponent zero, then normalised and finally divided in the floating point unit, the result is unpacked and shifted. The Cray-1 has no floating point divide unit but a floating point reciprocal unit instead.

Double precision arithmetic (dp) is also done in software on both machines but fewer instructions are required on the 7600 because of the availability of the double precision results of the floating point multiply and divide units.

The Cray-1 has a special set of instructions to manipulate vectors. Paragraph 2.5 discusses the vector instructions in more detail.

Branch instructions on the Cray-1 and on the 7600 are slightly different. On the 7600 it is only possible to jump to the beginning of a word, on the Cray-1 it is possible to jump to any of the four parcels inside a word. In conditional branches on both machines the content of a register is tested and depending on the content of the register the branch is taken or not. The Cray-1 is restricted by the fact that only register A0 and S0 can be used for testing.

2.3.2 Instruction timings

The instructions in the table on the next page that are implemented in hardware(H/W) have functional units to support them, these timings were taken from the manuals and are accurate. By implemented as 'inline' is meant that the function of the instruction is performed by a sequence of instructions inserted in the actual code, so no jump to a subroutine takes place. The number of instructions in the sequence is indicated between brackets in the table. The timings for 'inline' implemented procedures were obtained by running small Fortran (CFT) programs and checked by analysing the object code produced. These results should be taken with some caution, the CFT compiler might have done some clever optimisation, thereby obscuring the results. By implemented as 'subrt' is meant that the function of the instruction is performed by a subroutine constructed by Cray Research. The name of the subroutine is given between brackets in the table. A disadvantage of using a subroutine is that a subroutine cannot be processed in parallel with the instructions preceding or following that subroutine. By integer(long) is meant 60 bit for adding and 48bit in the case for multiplication and division on the 7600. For the Cray, 64 bit addition, multiply and division.

The branch instruction timings are for conditional branch instructions on the Cray-1 and the C.D.C-7600. Depending on the contents of the A0 register on the Cray-1 or any of the B registers on the 7600 a branch is taken or not. Testing on the contents of B0 is the unconditional branch on the 7600. When the branch is taken, the destination instruction may be in the instruction stack, resulting in a shorter execution time. If the destination is not in the instruction stack a delay occurs. Note that the timings given apply to one type of instruction, but for three different conditions.

The instruction issue time is 1 clock cycle for most instructions, but two parcel instructions have an issue time of 2 clock cycles on both machines.

	C.D.C-7600		CRAY-1		CDC/CRAY
	Time	Imp	Time	Imp	Ratio
* Real(sp)	137.5	H/W	87.5	H/W	1.57
/ Real(sp)	550	H/W	362.5	Inline(4)	1.52
+ Real(sp)	192.5	Inline(2)	75.5	H/W	2.57
- Real(sp)	192.5	Inline(2)	75.5	H/W	2.57
* Real(dp)	615	Inline(8)	2900	Subrt(DMSS)	0.21
/ Real(dp)	2117	Inline(18)	11000	Subrt(DDSS)	0.19
+ Real(dp)	577	Inline(10)	1975	Subrt(DASS)	0.29
- Real(dp)	577	Inline(10)	1975	Subrt(DSSS)	0.29
* Integer(long)	137.5	H/W	462	Inline(30)	0.30
/ Integer(long)	797.5	Inline(5)	987	Subrt(LDSS)	0.81
+ Integer(long)	55.5	H/W	37.5	H/W	1.5
- Integer(long)	55.5	H/W	37.5	H/W	1.5
Branch out stack	302.5	H/W	215.0	H/W	1.40
Branch in stack	82.5	H/W	62.5	H/W	1.32
Branch not taken	55.5	H/W	25.0	H/W	2.22

2.4 Pipelining

In a pipelined processor the instruction is segmented into several subprocesses which are executed by dedicated autonomous processing units. Successive operations can be carried out in an overlapped mode. The functional unit clock cycle timings in the previous paragraph indicate the number of subprocesses required to execute an instruction on that unit. The Cray-1 is fully segmented, so all functional units are capable of producing a result every clock cycle. Vector instructions on the Cray-1 place reservations on the source registers, destination registers and functional units. Scalar instructions on the 7600 and the Cray-1 only place reservations on the destination registers. Vector instructions can be overlapped if they do not conflict with each other in resource needs, i.e. they must use different functional units, and different source and destination registers. When a vector instruction uses the results of a preceding vector instruction as source, but uses a different functional unit and different destination registers, these instructions can be 'chained' together. In case of chaining the second vector instruction can start at 'chain slot time', i.e functional unit time plus two clock cycles. Note that when vector instructions run in parallel, two and even three results can be produced every clock cycle. The C.D.C-7600 is not fully segmented. The divide unit is not pipelined at all, so no two divide instructions can operate in parallel. This is because the algorithm for divide is an iterative process, and cannot be segmented. This explains why there is no divide functional unit on the Cray-1, and a reciprocal functional unit instead, which can be segmented. The multiply floating point unit on the 7600 has two period segmentation. So when a multiply instruction has been issued the next multiplication can only be issued two periods later. Since an 'out of instruction stack' condition prevents any further parallelism at that point, the size of the instruction stack is also significant, the Cray-1 stack is five times the size of the 7600 stack. The higher code density has also some effect on the speed. In the C.D.C-7600 machine code a lot of 'no operation' instructions are required, every cycle that a 'no operation instruction' is issued no other instruction can issue. The following example illustrates the effect of pipelining on the Cray-1.

Fortran source code:

```
X=X*Y+Z
V=W*W
```

Implementation

The implementation language is 'pseudo Cray-1 machine code'. By (X) is meant the contents of the address of variable X in memory. The B column indicates the starting time (in machine beats) of the instruction. The E column indicates the time (in machine beats) in which the instruction ends. Note that instructions that reference memory are two parcel instructions, 32 bits, which take two cycles to issue. Note that (S2+S4) can only start when (S2*S3) has completed. We assume there are no bank conflicts when referencing memory.

	B	E
S1=(W)	0	10
S2=(X)	2	12
S3=(Y)	4	14
S4=(Z)	6	16
S1=S1*S1	11	17
S2=S2*S3	15	21
(V)=S1	18	18
S2=S2+S4	22	27
(X)=S2	28	28

2.5 Vector processing

2.5.1 Definition of vector processing

In the processing of scientific problems most time is spent performing arithmetic operations on array elements inside a Fortran Do-loop. Within a loop, array indices are typically linear functions of the loop control variable. In scalar mode the operand arrays are processed element by element; e.g. first array element is fetched from memory, second element is fetched, operation is done, result is stored, if not end of loop do next iteration. In vector mode for each operation, the array elements that are referenced in the subsequent loop iterations are grouped into sets of data called vectors; e.g. the vector operand is fetched from memory, second vector is fetched, same operation is done on all operand vector elements, result is stored in memory. A vector processor is a computer architecture with an instruction set that includes instructions to manipulate vectors and has the hardware to support them. The following examples illustrate the difference between processing in scalar mode and vector mode. The instructions used and the timings given apply to the Cray-1.

Fortran source code:

```
Dimension AA(15),BB(10,15),CC(15)
Do 20  i=1,15
       CC(I)=AA(I)*BB(J,I)
20 CONTINUE
```

Scalar mode implementation:

In the following code sequence (AA) indicates the base-address of array AA in memory. AA points to the word before the first element of array AA in memory. Register A1 is used as loopcounter and as index register. Register A2 is used as index register. Matrix BB is accessed by row, the distance between the elements is 10. S1,S2,S3 are scalar registers.

		B	E
	A1=1		
	A2=J		
Loop:	S1=(AA)+A1	0	10
	S2=(BB)+A2	2	12
	S3=S1*S2	13	19
	(CC)+A1=S3	20	20
	A1=A1+1	21	22
	A2=A2+10	22	23
	IF(A1<16)GOTO Loop	24	27

=27*15=405 cycles

Vector mode Implementation:

The vector length is indicated by VL, and the distance between consecutive elements in memory is indicated by SKIP.

A0=(AA)	0	1
V1=(A0,VL=15,SKIP=1)	2	20
A0=(BB)+J	3	4
V2=(A0,VL=15,SKIP=10)	21	39
V3=V1*V2	29	52
A0=(CC)	30	31
(A0,VL=15,SKIP=1)=V3	53	53

=53 cycles

2.5.2 The advantages and disadvantages of vector processing

The previous example demonstrates that vector processing can be much faster than scalar processing. In this paragraph we try to determine why it is faster. Vectorising results in the removal of some overheads, e.g. loop control and array element address calculation. In addition the amount of overlap in processing of the instructions can be pushed to its maximum. The penalty paid is the overhead caused by starting up the vector operation.

A loop is a sequence of instructions that is executed until the loop reaches a certain value. Therefore loop control consists of incrementing the loop counter and testing at the end of every loop iteration if the loop counter has reached a certain value, if not, a jump is made to the beginning of the loop sequence.

Address calculation is especially relevant in case of access to arrays with multiple dimensions. e.g. AA(5,6,7,8) The indices of the element to be referenced and the dimensions of that array on declaration are used to calculate the address.

The amount of overlap is increased because the test and jump instruction no longer prevents the overlap of the current loop-instruction-sequence with the instructions in the preceding and following loop iteration. Another factor to be considered is the amount of effort required to restructure or rewrite programs to make them vectorise and the percentage of programs in a general purpose scientific workload that can be vectorised. In our opinion there is no doubt that the majority of scientific problems can be represented in a vectorised form. However most algorithms have been written with a scalar processor in mind. Converting these programs into vector form by restructering is probably not easy and rewriting too expensive. In Chapter 5 we investigate how much effort is required to restructure a subset of the U.L.C.C workload to allow vectorisation on the Cray-1, and how much is gained in performance.

2.5.3 The effect of scalar performance and short vectors on the performance

Before a vector instruction can be initiated a number of registers must be loaded with the required information, this is called the startup time of the vector. For very short vectors the startup time forms a significant part of the execution time. Very short vectors perform worse than their scalar equivalents, since there is no startup overhead in scalar mode. It is essential for a vector processor that is used in a general purpose environment to reduce the startup time to an absolute minimum. A good example of such a machine is the Cray-1, vector processing for elements longer than 3 are already more efficient than the scalar equivalent.

Another important point is the effect of scalar processing power on the effectiveness of a vector processor. A significant part of every workload is scalar. This scalar part includes the time spent in the operating system between jobs,which is essentially sequential, and problems that are scalar by nature and cannot be vectorised, unlike many of the iterative processes. If it is assumed that 50% of the workload run on a particular scalar processor can be vectorised, then the replacement of the scalar processor by a vector processor with the same scalar power but with an infinitely fast vector unit, will give a performance improvement of only a factor 2. If the vector portion is 70% the improvement would be about 3.3. This shows that scalar processing should not be neglected in a vector processor.

2.5.4 Vector processing on the Cray-1

The Cray-1 instruction set includes instructions to manipulate vectors. All arithmetic operations are done on vector registers. A vector register consists of 64 words of 64 bits. There are 8 vector registers. The Cray-1 is a three address machine, arithmetic instructions involve 2 operand registers and a result register. The operands can be two vectors, two scalars, or a scalar and a vector.

If the operand vector is longer than 64, then this vector will not fit into a vector register and has to be broken up in segments of 64 elements and a segment for the remaining elements. Vector registers are used for the same reason as scalar registers are used for scalar processing. Registers have a very short access time. Another point is that no bank conflicts can occur during arithmetic vector operations, therefore instructions finish at predictable times, which probably makes the design of the processor easier.

A vector can be copied from memory to the vector registers, or from vector registers to memory by a single instruction. Spacing between the vector elements to be copied from memory must be constant, therefore a row or diagonal of a matrix can be copied by one instruction.

A vector instruction is executed by entering every clock period one pair of vector elements to the functional unit. After the first pair has been processed a result is produced every clock cycle. The length of a vector, VL, is the number of elements in it. The functional unit time indicates the number of cycles required to process two scalars. The execution time of a vector instruction can now be defined as functional unit time + VL + 2 clock periods. This is in fact the time the result register is reserved.

The Cray-1 Fortran compiler tries to vectorise the 'most inner Do-loops' in standard Fortran programs. The compiler is still being improved to recognise more complex constructs inside the Do-loop. Certain constructs can never be vectorised, e.g. recursion and the program has to be restructured or rewritten to make it vectorise. Another problem is that at compile time it is not known how long the vectors are, if the vector length is shorter than 4 scalar mode processing might be faster.

2.5.5 Vector processing on the C.D.C-7600

The 7600 has no instruction set to manipulate vectors. However by combining the loop instruction sequences of several loop iterations into one instruction sequence, a sort of vector is obtained. The same advantages as for the vector processor apply but the result is far less spectacular. The 'length' of such a vector at which maximum performance is attained is determined by the size of the instruction stack. The size of the instruction stack is twelve 60 bit words, therefore not more than fourty eight 15 bit instructions can be contained. In case of matrix multiplication it was found that the maximum vector length was five. Factors that prohibit producing a result every cycle are the following: There are only five X registers to fetch operands from memory in parallel. Two parcel instructions take two cycles to issue. The multiply unit has 2 period segmentation. The divide unit is not pipelined at all. Chaining of vectors is of course not possible on the 7600.

Vectorisation on the 7600 gives an improvement of a factor 2 for tight Do-loops, and less for Do-loops with a longer instruction sequence, which already tend to have some overlap.

There exists a library of often occurring program constructs called Stacklib, these codes are optimally handcoded in Compass and make use of the 'vector' techniques described in this paragraph. Compass is the machine level language for the C.D.C-7600.

The following example demonstrates how vectorisation in scalar mode can be done. The Fortran source code is the vector multiplication from the previous examples. The assumption made is that the vector lengths used are multiples of three. In practice this is of course not always the case and therefore code has to be added to process the remaining elements. The code used and timings given are for the Cray-1 in scalar mode.

		B	E
	A2=J		
	A1=0		
Loop:	A3=A3+1	0	1
	S0=(AA+1)+A1	1	11
	S1=(BB+1)+A2	3	13
	S2=(AA+2)+A1	5	15
	S3=(BB+2)+A2	7	17
	S4=(AA+3)+A1	9	19
	S5=(BB+3)+A2	11	21
	S6=S0*S1	14	20
	S7=S2*S3	18	24
	S0=S4*S5	22	28
	(CC+1)=S6	21	21
	(CC+2)=S7	25	25
	(CC+3)=S0	29	29
	A2=A2+30	30	31
	A1=A1+3	31	32
	IF (A3<5) GOTO Loop	33	36

=5*36=180 cycles

2.6 Comparison at Kernel level, the Los Alamos test

2.6.1 Objectives of the test

The computational power needed at Los Alamos Scientific Laboratories exceeded the capacity of the fastest computer at that moment installed at L.A.S.L, the C.D.C-7600. Therefore L.A.S.L. in conjunction with the Energy Research and Development Administration (E.R.D.A) undertook to evaluate a new class of scientific computer systems. This class has been identified by E.R.D.A as 'Class VI', one that is capable of executing 20 to 60 million floating point operations per second. In 1976, the Cray-1 was the only machine that could possibly meet the Class VI requirements and that was operational.

The three areas of investigation were: scalar performance, vector performance and reliability. The scalar criterion is that the Cray-1 is at least two times faster than the 7600. The vector operations criteria are defined for three vector lengths. The criteria are that the Cray-1 be capable of executing vector operations at a rate of at least 3, 4 and 5 times at which the C.D.C-7600 can perform the equivalent functions for vectors of length 20, 100 and 500 respectively. Reliability is determined by three factors: System Availability (S.A, Mean Time To Failure (M.T.T.F) and Mean Time To Repair (M.T.T.R). The reliability criteria are that M.T.T.F be at least four hours. M.T.T.R be at most one hour, and S.A be at least 0.8. Furthermore the criteria must be collectively met during any contiguous 20-workday period.

2.6.2 The Kernels

Since no compiler was available on the Cray-1 at that time, any code implemented on the machine was written in assembly code. Optimal code was implemented on both machines; optimal is a yardstick for comparison. Since all coding has to be done in assembly code the total amount of test code had to be limited.

An analysis of the workloads at L.T.S.S and L.A.S.L was done to determine which codes were used most often. The analysis shows that a relatively small number of codes use up most of the C.P.U time. A measure of at least one run per week greater than one hour on the C.D.C-7600 was chosen as the level of significance to merit inclusion in the test.

However it was not possible to rigorously establish representative kernels for testing the vector performance of the Cray-1. This was because none of the existing codes comprising the workload had been converted for vector operations and such conversion efforts were outside the evaluation's time frame. Instead, a set of five 'typical' vector operations was defined to form an artificial benchmark. However the scalar performance was tested by using kernels representative of the workload at Los Alamos.

2.6.3 The results

The kernel timings in Table 6 show that the Cray-1 meets the workload scalar performance criterion (more than twice a 7600).

The vector performance criterion was met by the five vector operations for the three vector lengths. The average speed ratios were 3.39, 4.50 and 5.12 for vector lenths of 20, 100 and 500. The differences are primarily caused by the length of the vector registers on the Cray-1, which is 64. The Cray-1 reaches its best performance for vectors of length 64. Hence a vector of 500 is split into seven vectors of 64 and one of 52 elements. A vector of 100 is split into one vector of 64 and one vector of 36 elements.

Reliability of the Cray-1 was evaluated by running a reliability test code on the machine for long periods of time. Mean time to failure ranged from 0.34 to 0.52 hrs. System availability ranged from 0.85 to 0.95. An analysis of machine failures revealed that memory parity errors were the major cause. The indications are that the Cray-1 with error correcting memory would be an exceptionally reliable machine.

The final conclusions are that the Cray-1 meets every performance criterion established by L.A.S.L and E.R.D.A.

2.6.4 Application of these results to the U.L.C.C. workload

The workloads at U.L.C.C and L.A.S.L are similiar in the sense that they are both scientific workloads, consisting primarily of floating point arithmetic and manipulating arrays inside Do loops. However the workload at L.A.S.L consists of a relatively small number of very long running different codes. U.L.C.C processes about 3000 different jobs every day, a real general purpose workload. A possible consequence of this is that a significant portion of the U.L.C.C workload is less efficient than would be the case in a more specialised computing environment. Double precision arithmetic and 'improper' use of the 7600 L.C.M store did not occur in the L.A.S.L kernels, while they are quite common in the U.L.C.C workload.

A more serious problem is the fact that the L.A.S.L-test is essentially a hardware comparison, the effect of compiler efficiency and automatic vectorisation in particular is not included. In a general purpose environment the programs are generally not optimally handcoded in assembler code but written in Fortran. Therefore the quality of the object code influences the machine performance enormously, especially on pipelined machines.

Since the U.L.C.C workload is primarily written in Fortran it is also essential to find out how much conversion effort is required to transfer programs from the C.D.C-7600 to the Cray-1.

The conclusion is that although the results are to a large extent applicable to the U.L.C.C workload, they are not sufficient. Compiler efficiency, automatic vectorisation and conversion costs will have to be investigated.

2.7 Conclusions

The execution times of the floating point arithmetic instructions on the Cray-1 suggest a scalar performance ratio of 1.5 times a C.D.C-7600. The memory access times sugest a ratio of 1.6 times a 7600. In addition to this the Cray-1 has a larger main store, far more registers, a larger instruction stack, a higher code density and fully segmented pipelining. Therefore it seems reasonable to assume that the processing power of the Cray-1 hardware is at least twice a C.D.C-7600.

The Los Alamos test confirms this conclusion by comparing optimally machine coded kernels that model the L.A.S.L scientific workload on both machines.

However this comparison at component level is not sufficient since it is essentially a hardware comparison of the Cray-1 and the 7600. The main areas still to be investigated are : compiler efficiency, automatic vectorisation, the amount of effort required to convert Fortran programs from the 7600 to the Cray-1, and a more detailed analysis of characteristics of the U.L.C.C workload. The most effective way to investigate these areas is to construct a live benchmark, by selecting a large number of programs randomly from the U.L.C.C workload.

The effect of the vector capability on the performance of the Cray-1 cannot possibly be quantified exactly. This is because the workload at U.L.C.C has essentially been written with a scalar processor in mind. However we can investigate how well the current workload is suited to a vector processor for the following three extreme cases: the first case is to assume that none of the users bothers to adjust their programs to take advantage of the vector capability, so the only vectorisation that takes place is due to the automatic vectorisation by the Cray-1 Fortran compiler (C.F.T). The second case is to assume that all users restructure their most inner Do-loops to make the automatic vectorisation by the C.F.T compiler more effective. The third case is to assume that all users redesign the algorithms of their programs, with the vector capabilities of the Cray-1 in mind.

CHAPTER 3

COMPARISON AT BENCHMARK LEVEL

3.1 Summary of the benchmarking process

In Chapter 2 we concluded that the Cray-1 hardware is capable of processing our type of workload at a rate of more than two times the C.D.C-7600. However further investigation in the performance of the Cray-1 using a 'live benchmark' is necessary. In a live benchmark a representative set of programs, called the Test Set, is chosen directly from the user's workload. These programs are arranged so that when run they act as an model of the workload. The main objectives of the benchmark are to determine the throughput of the Cray-1 when processing the workload compared to the C.D.C-7600, and also to find out how much effort is required to convert the Fortran programs from the 7600 to the Cray-1.

In May 1979 a set of programs was selected from the U.L.C.C workload, by members of the U.L.C.C staff. During the following three months debugging of the selected programs that did not work properly on the 7600 took place. This work was carried out by the Author with the help of M. Langton and W.H.L Williams, and it was supervised by C. Lazou and R. Baker.

In July 1979 the conversion of the C.D.C programs to make them run on the Cray-1 took place. This was done by C. Lazou, G. Ashley and the Author. It took us 6 manweeks of work.

In September 1979 the benchmark was run on the two systems and a detailed internal report was produced by R. Baker and J. Down for U.L.C.C. There was also an internal report discussing the benchmark results produced by G. Ashley for Cray Research U.K. The contents of these reports have contributed to the present chapter.

3.2 Considerations for optimal construction and execution of a Test Set

To make sure that the Test Set of a benchmark is representative of the expected workload we must ideally select our programs randomly from the user's workload, and the number of jobs in the sample must be large enough to make the test 'statistically significant'. If a certain change in the workload profile is expected, we should adjust our Test Set accordingly.

When running the Test Set on different machines, we want the programs ideally to be logically equivalent. By this is meant that when using the same source code and equivalent job control language, exactly identical results must be produced. This implies that only jobs that run to a well defined completion are to be included in the Test Set. When changes are made to programs on one machine, equivalent changes should be made on the other machine. In practice programs are not equivalent. This is due to differences in computer architecture and system software. For example differences in word length will inevitably give differences in the results produced, because of different accuracies. However we should try to limit the differences to an absolute minimum. Therefore we must establish rules to indicate what changes are allowed to be made to the programs, and what differences in results are acceptable.

Checks must be made to ensure that the objectives mentioned are met as closely as possible, that the rules have been followed and that the benchmark was run and timed correctly. To check that the benchmark is representative of the expected workload one could compare certain performance characteristics of the Test Set with those in our workload. Examples of such characteristics are: i.o. activity, percentage of jobs using a particular compiler and the percentage of time performing integer or double precision arithmetic. The problem is that there are so many factors affecting the performance that obtaining a high degree of certainty about the representative nature of the Test Set may be difficult. Some people argue that because so many different factors influence the performance of the workload, there should be such a large number of programs in the benchmark test set that the operation becomes impractical and therefore reject the whole benchmark idea.

3.3 The forming of the Test Set

About sixty programs were taken randomly from the input queue of the C.D.C-7600 over a period of several days. A group of eight programs, that was originally used in a previous test, was added to this. This previous test was a Synthetic Benchmark, the test set used in this benchmark also consisted of user programs.The objective of this Synthetic Benchmark was to investigate particular characteristics of the Cray-1. E.g. an i.o bound job was included to examine its effect on the Cray-1 compared to the 7600.

Jobs in the Test Set of our Live Benchmark that made use of absolute binary files, binary data, user secure programs, jobs using microfiche etc. were removed. Programs which terminated on a time limit on the 7600 were modified to terminate within a specified time limit, for example by changing the number of iterations the main program loops performed. Two programs were found to be character set dependent and were omitted since time did not permit equivalent code to be produced. Programs that produced compile time errors were corrected.

To establish the expected balance of large to small jobs, multiple copies of the small jobs were included in the final set. The size of the final test set was 68 programs, roughly two hours of C.D.C-7600 time. Certain characteristics of the test set were compared with those of a large sample of jobs taken from the workload. We compared the amount of S.C.M usage, amount of L.C.M usage and the amount of input/output to disc. And after taking into account the slightly different distribution of jobs in the benchmark test set, a higher percentage of large jobs, we concluded that the parameters measured were identical.

The cost of producing the benchmark test set was considerable, 10 manweeks of programming effort and several hours of C.D.C-7600 time.

3.4 The conversion of the Test Set

3.4.1 The rules

When the Test Set is converted to run on another machine some of the representativeness may be lost. To ensure that the Test Sets on the two machines are closely logically equivalent we established the following rules:

1) Job control language was to be converted to match the 7600 job control language as closely as possible. In particular, levels of Fortran listings had to be matched (e.g. cross reference listings), and all runs on the C.D.C-7600 produced a source listing.

2) Only the minimum changes necessary to produce 'identical' results to the 7600 were allowed.

3) No program optimisation was allowed.

4) Programs that require very high conversion effort, but run only for a very short time can be rejected.

3.4.2 The results

The U.L.C.C test set was transferred to the Cray-1 on magnetic tape in card image form. The conversion took three people five '15 hour days'. The 68 programs in our benchmark test set contained about 54000 lines of code and 34000 card images of data.

Almost 30 percent of jobs executed completely on the first attempt. About 670 lines of code produced errors at compilation time, half of which were caused by having more than one Fortran statement on a single line of code. This is not allowed in the Cray-1 Fortran (C.F.T).

Both Cray-1 Fortran and C.D.C Fortran support extensions to the A.N.S.I 1966 Fortran standard. However the new Fortran 77 A.N.S.I standard, will tend to force a convergence of facilities in the two Fortran compilers. At the time of conversion, August 1979, these extensions were not entirely compatible and differences were discovered during the compilation phase. Appendix 2 gives a description of known differences between Cray Fortran (C.F.T) and C.D.C Fortran (F.T.N)

The cost of conversion was estimated at 6 manweeks and a few hours of Cray-1 time.

3.4.3 Conclusions

The conversion of Fortran programs from the U.L.C.C workload currently running on the C.D.C-7600 to the Cray-1 gives no serious problems.

3.5 The running of the Test Set

3.5.1 The rules

To ensure the representativeness of our Test Set it must be run on both machines in an identical way and the timings must be done in an identical way. Therefore we established the following rules:

1) Jobs are loaded and executed from the system discs.

2) The measured throughput time is from start of processing to completion of the last job.

3) The Test Set is run three times and the average is taken as the final result. A member of U.L.C.C staff was present

4) The only package used by the Test Set (Shelx) was compiled and stored as a binary image on the Cray-1 discs for use by any programs. (i.e the compilation would not be part of the Test Set)

5) The throughput time is affected by the job mix in the Test Set. If the last job to be processed would be a large i/o bound job, and it was the only job present in the system, every i/o request would result in the C.P.U being idle for the duration of the request. This effect is often called tailing. Therefore the Test Set presented to the system must have the right mix of large and small jobs and the right mix of i/o bound and C.P.U bound jobs. On the 7600 this is done automatically by the scheduler, which makes use of the job classification which indicates the resource usage of that job. Since no job classification was available on the Cray-1, it was allowable to allocate appropriate priorities to the jobs to obtain the same effect. Hence the i/o bound and large jobs were given a high priority.

3.5.2 The software and hardware used

The conversion work and the running of the Cray-1 benchmark test set was carried out at the S.R.C Nuclear Physics Laboratory in Daresbury. The machine installed at Daresbury was the first Cray-1 to be built (serial-1) and is different in several areas from subsequently installed production Cray-1's. In particular, it has only parity bit checking for memory accesses rather than full SECDED corrections on the other Cray-1's. For this reason Serial-1 is a few percent faster than the other Cray-1's. Serial-1 has a main store of half a million words distributed across 16 memory banks. Other Cray-1's either have one million words across 16 memory banks or half a million words across 8 memory banks. The software used on the Cray-1 was based on the 1.05 release. However the Fortran compiler used was C.F.T 1.06.

The construction of the Test Set and the running of the 7600 Test Set was of course done at U.L.C.C London. This machine has a small core memory of 64 K words distributed across 16 independent banks. The large core memory (L.C.M) consists of 256 K words distributed over 4 independent banks. Every bank in L.C.M is eight words wide. The L.C.M is a serial access store. The S.C.M is a random access store. The operating system used was Scope 2.1.4 level 228H. The compilers used were F.T.N and M.N.F. Also a library of scientific routines called N.A.G was used. This library is optimised for the 7600. The Cray-1 version of the N.A.G library was not optimised for the Cray-1, and in particular it is not vectorised.

3.5.3 The results

The Throughput ratios

After running our Test Set three times on both machines we obtained the following results:

	C.D.C-7600	Cray-1
	Time:Hrs:Mins:Secs	Time:Hrs:Mins:Secs
RUN1	2:17:12	1:23:50
RUN2	2:16:00	1:23:50
RUN3	2:17:55	1:23:50
Mean	2:17:02	1:23:50

The throughput ratio was 1.65

Individual Ratios

Table 1 records the measurements made on the two machines for individual jobs. Every job consists of execution time plus compilation time. We see that most jobs execute faster on the Cray-1 than on the C.D.C-7600. However five programs perform better on the 7600. Two of these jobs U2 and U5 were found to perform all their calculations in double precision (128 bit arithmetic). The program U20 spends 60 percent of its time evaluating ALOG and RANF, these 2 routines appear to be slower on the Cray-1 than on the 7600. The program U17 spends most of its time performing 64 bit integer arithmetic. Since this job forms about 25 percent of the Cray-1 throughput time, the throughput ratio for our benchmark test is significantly affected. The program U18 was found to spend 91 percent of its time in a N.A.G routine that uses double precision arithmetic.

The ratio (7600 job time)/(Cray-1 job time) for the individual programs varies from 0.39 to 9.57. Sixteen of the 68 jobs have a performance ratio of below two. Almost fifty percent of jobs have performance ratios between two and four. The very small jobs seem to perform extremely well on the Cray-1.

3.5.4 Conclusions

The overall throughput time is 1.65. The range of ratios varies from 0.39 to 9.57. These results illustrate the enormous variance in the performance ratios of the programs. And in fact more questions arise than are answered. Some of the questions that arise are:

Most jobs that are relatively small have a very high performance ratio (3-5), is this due to differences in compilation times?

What is the effect of vectorisation, do programs with high performance ratios vectorise?

Jobs U17 plus Tim form together 40 percent of the Cray-1 throughput time. Is a throughput that is dominated by a small number of jobs statistically valid?

3.6 Limitations of a live benchmark

3.6.1 Modelling the future workload

It is very difficult to design a test set that models exactly the future workload in a general purpose environment. This is because the workload is always subject to change. It is the authors opinion that there will be no dramatic changes to the U.L.C.C workload, but because of possible changes, the results of a benchmark should always be taken with some caution.

3.6.2 The size of the test set

If a small percentage of jobs in a workload uses up most of the execution time necessary to process that workload, then the throughput time of the test set modelling that workload is also dependent on a small number of jobs and therefore possibly subject to a high degree of error. The best way to solve this problem is to take more of these large jobs and to maintain the right balance between smaller and large jobs by making multiple copies of the smaller jobs. In case of our benchmark we could take 10 extra jobs of the size of U17 and TIM. The disadvantage of such an approach would be that the throughput time would become unacceptably long. The length of the throughput time for our test set would be extended by several hours. This approach of course suggests that jobs with a longer execution time behave differently. In the case of the U.L.C.C workload there is serious doubt whether this is the case.

More useful results can probably be obtained by analysing the individual behaviour of the programs in the test set.

3.6.3 Representativeness of the test set

Earlier on in this chapter we discussed how difficult it is to validate the representativeness of a test set. Because U17 uses up such a high percentage of the throughput time, it is possible that the result is biased by full word integer arithmetic. In March 1981, a hardware monitor will be used to determine the instruction mix in the benchmark test set. The same monitor will be used to determine the instruction mix in the actual U.L.C.C workload over several hours. This will give us more information about the representativeness of the benchmark test set. A serious problem is that it is impossible to differentiate between integer multiply and double precision multiply, since they use the same instruction.

3.6.4 Optimising the workload

In addition to the live benchmark we need an extra test to find out what performance can be obtained when the workload is optimised.

3.6.5 Hidden problems

There are a great many factors that can influence the results of a benchmark. An example is program U20, which computes a random number and then uses it as a count to a loop. The two systems use different random number generators and hence the number of iterations were not equivalent. For the benchmark test the count was set to a fixed number of iterations. Another example was the use of a software tool called Mantrap on the C.D.C-7600. The use of this debugging tool resulted in the F.T.N compiler producing less efficient object code, this factor is illustrated in the analysis of the test set in the next chapter.

3.6.6 The cost of the benchmark

The cost of the benchmark is extremely high: 10 manweeks constructing the test set plus 6 manweeks converting it to the Cray-1, plus several hours of C.D.C-7600 time and several hours of Cray-1 time.

3.7 Conclusions

The enormous variance in individual program results shows that care must be taken in drawing the conclusions from this benchmark. More information is required on individual program performance results.

To find out what the capabilities of the Cray-1 and the 7600 are it will also be a requirement to compare the Cray-1 and the 7600 for an optimised benchmark.

Benchmarking is a very expensive process, but there is no cheaper alternative available that produces results of the same quality.

CHAPTER 4

ANALYSIS OF THE BENCHMARK RESULTS

4.1. Introduction

In the previous chapter we concluded that the variation in the performance ratios for the individual programs was very high. In the current chapter we try to determine the factors that cause these big fluctuations in performance ratios. The results of this study might enable us to obtain a better throughput. This is especially relevant for the C.D.C-7600 at U.L.C.C since the workload will soon reach the maximum capacity of the system and a new system will probably not be introduced within a year. Note that this type of tuning does not effect the actual source code of the user programs.

In the next chapter we try to obtain a better system utilisation by optimising the source code, the programs are restructured to allow object code to be generated by the compiler that makes better use of the parallel capabilities of the two pipelined machines.

4.2 Approach to the analysis

Because the analysis is very time consuming, only a subset containing twenty programs from the original test set was analysed. Programs were selected randomly, but programs using Shelx were not included. One job was not included because it was a duplicate.

The investigation concentrated on the following three areas: Differences in architecture, differences in compilers and differences in software tools and packages.

4.2.1 Differences in architecture

Architectural differences of the Cray-1 and the C.D.C-7600 have been discussed in Chapter 2.

4.2.2 Differences in compilers

The task of a compiler is to translate the high level language (e.g. Fortran) source code program into a logically equivalent machine code program and to check if there are any syntax errors in the source program. If there are any errors, it produces diagnostic messages. The objectives of a compiler should be to produce the most efficient machine code possible and/or to give clear and meaningful messages.

In a student environment the syntax analysis part of the compiler is considered to be extremely important, since it makes it easier for the student to find his syntax errors when clear diagnostic messages are given. Also the execution time of jobs in such a student environment is often very short, often shorter than the compilation time. Therefore compilers were constructed that performed an extremely good syntax analysis and compiled jobs in a very short time, this being done by omitting the optimisation passes of the object code. Such a compiler is called a teaching compiler. The M.N.F compiler on the C.D.C-7600 at U.L.C.C is an example of such a compiler.

In a production environment the efficiency of the object code is important and therefore all possible optimisations are done to produce the best code possible. Such a compiler is called an optimising compiler. The C.F.T compiler on the Cray-1 and the F.T.N compiler on the C.D.C-7600 are examples of optimising compilers.

The F.T.N compiler even tries to rearrange the sequence of instructions to make them go smoother down the pipeline. The F.T.N compiler has been developed over the last fifteen years and is an extremely good optimiser. The F.T.N compiler has several levels of optimisation: OPT=0, OPT=1, OPT=2 and UO (=unsave). The level of optmisation is indicated as an compiler option.

In the OPT=0 compilation mode, compile time evaluations are made of constant subexpressions, redundant instructions and expressions within a statement are eliminated and P.E.R.T critical path scheduling is done to utilise the pipeline as much as possible.

In the OPT=1 compilation mode the following optimisations take place: Redundant instructions and expressions within a sequence of instructions are eliminated (dead code). Subscript calculations are simplified. An attempt is made to keep values of integer variables in machine registers throughout inner Do-loop execution.

In the OPT=2 compilation mode the following additional optimisations take place: Values of simple variables are not retained when they are not referenced by succeeding statements. Invariant (loop independent) subexpressions are evaluated prior to entering the loop containing them. For all loops, the evaluation of subscript expressions containing a recursively defined integer variable (such as i when i=i+1 appears within the loop) is reduced from multiplication to addition. Array addresses, values of simple variables in central memory and subscript expressions are stored in machine registers throughout loop execution for all loops. In all loops and in complicated sections of straight-line code, array references and subscript values are stored in machine registers.

In the UO (unsave) compilation mode the user must also specify OPT=2. The following optimisations are done: In tight Do-loops, indexed array references are prefetched unconditionally without any safety checks. When a basic external function is referenced, the compiler assumes that the contents of certain B registers are preserved for use following the function processing.

The C.F.T Fortran compiler on the Cray-1 performs optimisation similiar to F.T.N but in addition it analyses inner Do-loops to find out whether they can be vectorised. This is called automatic vectorisation. In the next chapter we discuss the vectorisation process in greater detail. The Cray-1 has far more registers available for storing operands and intermediate results than the C.D.C-7600. Therefore a Cray-compiler employing a sophisticated register allocation scheme can reduce the number of main store references dramatically. Especially in the case of big complicated inner Do-loops, that often do not vectorise, the improvement would be significant. Such a register allocation scheme is extra complicated because the intermediate registers must be allocated at compile time, indirect or modified addressing of the intermediate registers is not supported by the architecture. Hence an automatic cache store would make the task easier.

4.2.3 Differences in packages and software tools

Our test set contained several programs referencing two packages very commonly used at U.L.C.C. The packages were SHELX and the NAG library. SHELX is used in chrystallographic x-ray analysis. The NAG library is a mathematic subroutine library.

Both SHELX and the NAG library have been optimised on the C.D.C-7600. The Cray-1 versions of the packages were not optimised. In 1981 the NAG library will probably be vectorised by Cray-research.

The test set also included several programs using a software tool called Mantrap.This facility enables the user to obtain runtime diagnostics to debug his program. Unfortunately the use of Mantrap results in optimisation level 0 being used. This is done because the rearrangement of the instruction sequence would make it impossible to obtain proper diagnostics. The reason that a fairly high proportion of the programs have Mantrap in them, is that a lot of the programs in the test set when selected were in the debugging mode, used extensively by the users.

4.3 Analysis of the C.D.C-7600 results

The results of the analysis are shown in Table 2 at the end of this thesis. The identifier column indicates the program identifier of the jobs in the original test set.

The comment column indicates characteristics of that program, relevant for its performance on the C.D.C-7600 in the original benchmark run. MNF means that the job was compiled using the M.N.F compiler. FTN(2) means that the F.T.N compiler was used with optimisation level 2. MAN indicates that the debugging facility Mantrap was used. DP indicates that the program performs a lot of double precision arithmetic. LEV2 indicates that the program makes use of L.C.M.

The next 2 columns show the compilation and execution times of the programs in the original benchmark run.

The MNF FTN(2) column shows the compilation and execution times of the programs run under FTN(2) that were originally run under M.N.F. The table is continued on the next page.

The FTN(1,0) FTN(2) column shows the compilation and execution times of the programs using FTN(2) that were originally compiled with optimisation level 0 or 1.

The better memory management column shows the compilation and execution times for programs after improving memory management, by making better use of S.C.M. and L.C.M. All of these programs were run with FTN(2).

The DP SP column indicates the compilation and execution times after changing double precision to single precision. These programs were run with FTN(2) and improved memory management.

The ratios shown in the following paragraphs were calculated in the following way: (original benchmark time)/(modified benchmark time) so the timing in the original benchmark is used as base 1.

4.3.1 M.N.F versus F.T.N(2)

In table 2 we see that six of the twenty programs in our subset were run with the M.N.F compiler during the benchmark run. The compilation times of M.N.F. are much shorter than those of F.T.N(2) but the execution times are much longer. We obtain the following ratios for (MNF)/(FTN):

id	compil	exec	comp+exec	seconds saved
U2	0.18	1.1	1.1	6
U3	0.28	2.9	2.7	17
U6	0.37	2.8	2.7	25.5
U9	0.44	1.7	1.7	69
U16	0.48	1.25	0.86	-1.5
U51	0.39	1.5	1.4	23

139 sec
=4% of the subset total

We see that only in the case of U16 the use of M.N.F resulted in a lower job time and that 4% of the total time used by the subset was wasted by M.N.F. The compilation of M.N.F programs is two to three times faster than F.T.N programs. The execution time of F.T.N object programs is 1.5 to three times faster than the M.N.F ones. Therefore we can conclude that , while ignoring other factors, M.N.F should only be used if the compilation time is of the same order as the execution time. Since the main purpose of M.N.F is that of syntax checking it would be worth considering removing the code generating part of the compiler to prevent it being used for execution.

A problem was encountered when jobs were changed from M.N.F to F.T.N. Three of the programs that worked under M.N.F, did not work under F.T.N(2). All of these three jobs had problems with the use of Level-2 statements. So if M.N.F is removed as a compiler some user problems could be expected.

4.3.2 FTN(1,0) versus FTN(2)

On the U.L.C.C installation FTN(opt=2) is used as a default level of optimisation. To make sure that this is the most efficient way of running the workload, we ran our subset with FTN(1) and FTN(2). The results are shown in Table 4. We see that the compilation time of Opt=1 jobs twenty percent shorter is than compiled with Opt=2. The execution time is in most cases about ten percent faster with Opt=2 and in a few cases sixty percent or more faster. Note that if the program TIM had been run with Opt=1 the program would have run 600 seconds slower!!! So there is no doubt that FTN(2) is the correct default option.

In our benchmark run we had three programs that used Mantrap. Mantrap results in optimisation level zero. One program specifies optimisation level one explicitly. After changing these jobs to FTN(2) we obtain the following ratios:

ID	Comp	Exec	Secs saved
U4	0.82	1.25	0.0
U10	0.72	1.72	210
U14	0.90	0.93	-19
U52	0.89	1.27	9

200 sec
=5.7%

We see that 5.7 percent of the execution time of our subset is accounted for by a lower optimisation than FTN(2). In case of U14 the use of FTN(1) gave a lower job time.

4.3.3 Better memory management

In this paragraph we investigate how much can be gained by a better use of the Large Core Memory. L.C.M is an four bank linear select memory. Every bank is of width eight words; a reference to L.C.M results in eight consecutive words from one bank being moved to a register called Super Word (SWORD), and there is one SWORD for every bank. An L.C.M request for a word that resides in the SWORD can be served very fast, since the SWORD does not have to be reloaded. Random access to L.C.M is very inefficient because a SWORD is loaded every time.

The programming language Fortran has defined that multidimensional arrays are stored by column. However 90 percent of the programmers population instinctively refer to their matrices by row. Access by row means random access.

In Appendix 3 we give guidelines for better memory management. By applying these guidelines to our subset we obtained the following results.

ID	Comp	Exec	Secs saved
U4	1.08	1.06	0.1
U8	1.00	1.07	30.5
U9	1.03	1.83	44.7
U10	1.01	1.11	30.8
U14	1.06	1.22	47.2
U15	1.00	2.46	153
U31	1.00	1.00	0.0
U51	1.01	1.80	20.5
WNG	1.00	1.00	0.0
TIM	1.03	1.03	24.9

351.7
=9.78% of the subset total

4.3.4 Double precision arithmetic

Double precision arithmetic on a sixty bit machine like C.D.C-7600 is very rarely required. It is difficult to think of a problem that would require such an amount of precision. Most programs that have double precision in them originate from a machine with shorter wordlength e.g. 16 or 32 bits, in which case using double precision would make sense. Many users are satisfied when they get their programs to work and do not bother about the efficiency of their program.

In the subset we have three programs with double precision. After changing them to single precision they printed identical results, so to our opinion the conversion was justified. The ratios for (double precision)/(single precision) were:

Id	Comp	Exec	Secs saved
U2	1.56	3.22	39.5
U10	1.03	1.10	24
U30	1.06	1.07	0.0

63.5 seconds
=1.8% of the subset total

4.4 Analysis of the Cray-1 results

The results of the analysis are shown in Table 3 near the end of the thesis. The identifier column indicates the program identifier of the program in the original benchmark run.

The comment column indicates characteristics of the programs during the original benchmark run on the Cray-1. DP indicates double precision, INT indicates that integer arithmetic is extensively used. RMS/WMS indicates that the program reads from and writes to disc.

The next two columns indicate the compilation times of the programs during the original benchmark.

The novec column indicates the execution times of the programs with automatic vectorisation switched off. This is done by a special compiler option (CFT(OFF=V)).

The DP SP column shows the execution times of the programs after changing double precision to single precision but with vectorisation switched on.

4.4.1 The effectiveness of automatic vectorisation by CFT

To find out how effective automatic vectorisation was without any help by the user, we run our subset of the test set with automatic vectorisation switched off. The results for (V=OFF)/(V=ON) were as follows:

Id	Ratio	Secs saved
U2	1.0	2.0
U3	1.06	0.5
U4	1.0	0.0
U6	1.0	0.0
U8	1.0	0.0
U9	1.0	1.0
U10	1.06	21.0
U13	1.05	7.0
U14	1.02	2.0
U15	1.07	3.0
U16	1.16	0.0
U30	1.0	0.0
U31	1.87	141.0
U32	1.14	0.0
U51	1.84	11.5
U52	1.11	3.0
U57	1.03	1.0
U58	1.10	1.0
WNG	1.0	0.0
TIM	1.0	0.0

194 seconds
=7.6% of the subset total

From these results we see that only two, U31 (1.87) and U51 (1.84) of the twenty programs in our subset did vectorise without the help of the user.

In the next chapter we try to help the automatic vectorisation by restructuring the source code.

4.4.2 Double precision arithmetic

Three programs of our subset used double precision. After changing these programs to single precision we obtained for (double precision)/(single precision) the following ratios:

Id	Exec	Secs saved
U2	22.46	156.7
U10	1.52	130.4
U32	1.80	0.53

287.6 seconds
=11.4% of the subset total

We see that the effect of double precision is rather dramatic. In program U10 and U32 only part of the arithmetic is double precision. All three programs printed the same results as with double precision, so the conversion was justified. Also note that double precision prevents vectorisation.

4.4.3 Integer arithmetic

Full word integer arithmetic is also implemented in software, so performance of such jobs will be poor. The number of instructions generated would be significantly less if full word integer arithmetic was changed to 48 bit arithmetic. Cray research is currently investigating this. Some of the programs with integer arithmetic are U17, U13 and U32. Because U17 forms such a substantial part of the test set (1229 seconds on the Cray-1) the effect of this job on the throughput time is quite significant.

4.4.4 Faster discs

The program U16 spends a lot of its time reading from and writing to disc. Since the discs connected to the Cray-1 system are faster than the discs connected to the C.D.C-7600 this program performed much better on the Cray-1.

4.5 Comparison of the Cray-1 and the C.D.C-7600

Table 5 shows three histograms. The top histogram shows the original ratios of the execution times of the programs in our subset of the benchmark test set. They were calculated by dividing the execution time for a program on the C.D.C-7600 by the execution time for that program on the Cray-1. The ratio for these two subset totals is (3867=CDC)/(2329=Cray)=1.66. The middle histogram shows the ratios for the execution times after making the following adjustments: MNF changed to FTN(2), double precision changed to single precision, and FTN(1,0) changed to FTN(2). The ratio for these two subset totals is (3464=CDC)/(2077=Cray)=1.67. The third histogram shows ratios for programs with the same adjustments as the middle diagram, but additionally the vectorisation is switched off and C.D.C programs have improved memory management. The ratio for these two subset totals is (3107=CDC)/(2241=Cray)=1.39.

The middle histogram is a better comparison of the Cray and the 7600 because the Cray has no equivalent of Mantrap and no M.N.F compiler, so these should not be included in the comparison of the two machines. Also double precision was changed to single precision for this comparison since it is silly to use double precision on these machines, it could however be argued that this is unfair because double precision will be part of our workload.

The third histogram has been constructed to assess the effectiveness of the Cray-1 compiler in scalar mode. The program U16 performs very well on the Cray-1 because of the faster discs, and U9 performs very well because after improved memory management the program still performs a large amount of random access in L.C.M. The variance of the ratios is small. The ratio for the execution time totals of these two sets is 1.39. This seems rather poor, especially when we compare these results with the Los Alamos results. The Los Alamos test, see Table 6, compared the scalar performance of the two machines for optimally handcoded kernels. All the kernels were in SCM and no vector instructions were used, neither was there double precision arithmetic. Therefore these kernels are comparable to the programs in the third histogram. The conclusion is that the object code produced by CFT is inefficient. A study of the object code produced by the CFT compiler for several of the programs in our test set confirmed that the code was not very efficient.

4.6 Conclusions

In this chapter we concluded that the throughput rate of both the Cray-1 and the C.D.C-7600 could be improved and that the benchmark results would alter if jobs were modified as described below

How to improve the throughput rate of the C.D.C-7600 at U.L.C.C

Do not use the M.N.F compiler for production runs. This can be done by removing the M.N.F compiler altogether or by removing the code generation part of the compiler and so turn it into a syntax checker or by better education of the users.

Do not use Mantrap for production runs. This can be helped by printing a message box on the users listing to remind the user that Mantrap should only be used for debugging purposes

Do not use double precision arithmetic, if possible. This can again be helped by a message box.

Considerable improvement can be obtained by better memory management. This can be helped by providing the user with more information, e.g. a bulletin, about how to use L.C.M and S.C.M. Since this process is fairly simple it is possible to provide this information in a message box on the users listing. So when the compiler finds a LEVEL 2 statement the box is printed.

How to improve the Throughput of the Cray-1

Do not use double precision arithmetic. It is very slow and it prevents vectorisation. The user should be reminded of this by a message box

Since only ten percent of the programs vectorise, it is useful to help the automatic vectorisation by restructuring the source code. This is discussed in the next chapter.

The effect of the results on the Benchmark results

In the current chapter we have been able to explain all of the big fluctuations in the performance ratios.
We conclude that M.N.F and Mantrap should not have been used in the test set for the 7600, since equivalent facilities were not available on the Cray-1. But memory management of the programs in the 7600 test set should not be modified since this problem results from the C.D.C-7600 architecture. We could also remove double precision arithmetic since its use in many cases can not be justfied. However it could be argued that it should be left in the two test sets, since it is part of the U.L.C.C workload and the users are likely to continue to use it in future.

After removing M.N.F and Mantrap by using FTN(2), and changing double precision to single precision we found that the throughput ratio stayed about the same but the variance in ratios was smaller.

When examining the object code produced by the C.F.T compiler we found that this was in several cases very inefficient. In the Los Alamos tests, a number of kernels were optimally handcoded for the Cray-1 and the 7600. The kernels in these tests were similiar to our workload, however the Cray-1 performed much better in that scalar test than in ours. Since the Los Alamos test did not include compiler efficiency, the difference in relative performance must be attributed to the compiler.

CHAPTER 5

VECTORISING A SUBSET OF THE TEST SET

5.1 Introduction

In chapter 2 we have described vectorisation at instruction level. In the current chapter we look at vectorisation at language level. There are a few languages that include vector operations. Examples are Tranquil for ILLIAC IV and DAP Fortran for ICL's Distributed Array Processor. The problem is that such languages are only supported by a few machines. The most commonly used language in a general purpose workload is standard Fortran. Often there is also some work using Pascal and Algol.These languages in their standard form do not include vector operations.

Vectorisation of standard Fortran programs on the Cray-1 is done by the C.F.T compiler. Unfortunately many programs do not vectorise straight away. In the current chapter we discuss why certain program constructs do not vectorise and give some guidelines how they can be restructured to vectorise. Appendix 3 contains a detailed description of different techniques that can be applied to make these programs vectorise. The restructuring techniques have been applied to a subset of the benchmark test set. We discuss the results and our experiences doing these tests.

Extensive support to the practical work in this chapter was given by Cray Research U.K. They provided sufficient Cray-1 time and support to complete the work in time.

5.2 Optimisation

When writing a program to solve a problem, our objective must be to use the numerically most efficient algorithm for that problem and for the machine(s) on which we intend to run the program. Hence knowledge of the physics of the problem and knowledge of the target machine(s) features may be essential to achieve an optimal result.

In practice, a large proportion of programs are inefficient. This is often because a program has been designed with a different type of computer architecture in mind, or the program has been under development for a long period, sometimes tens of years, resulting in unstructured code and a less efficient algorithm. Furthermore significant progress has been made in the field of numerical analysis, so algorithms can be 'outdated' rather quickly.

Since vector processors have been scarce until recently, most scientific programs have been written with a scalar processor in mind. Therefore a lot of problems have been solved with algorithms which are not suitable to a vector processor.

One can split methods of optimisation in two categories: intelligent optimisation and non intelligent optimisation. In case of non intelligent optimisation one tries to optimise the code of the existing programs, thereby accepting the algorithm used, no knowledge of the physics of the problem being required. An example is the optimisation work done by the compiler. In case of intelligent optimisation one looks at the physics of the problem and the features of the target machine, to find the best algorithm for the problem. Intelligent optimisation often means rewriting the entire program. A difficulty is that information about the physics of the problem is not always available.

5.3 Automatic vectorisation

5.3.1 Automatic vectorisation and non automatic vectorisation

In case of non automatic vectorisation it is the user who has to specify explicitly where vectorisation has to take place. If a 'vector language' is available then the vector operations can be neatly specified in the program. An alternative is to have a set of vectorised routines which can be called from standard Fortran programs. Such a set of routines could consist of the basic vector operations and scientific routines which are used most commonly. The Cray-1 has such a set of routines called Scilib. A disadvantage of non-automatic vectorisation is that already existing standard Fortran programs will not vectorise, and that vectorised programs are machine dependent.

In case of automatic vectorisation a program that is written in a high level language is vectorised by a compiler or vectorising program. The Cray-1 Fortan compiler tries to vectorise most innner Do-loops. If the program does still not vectorise the user can restructure the code in the inner Do-loop where most time is spent during execution to make it vectorise. Hence automatic vectorisation is an example of non-intelligent optimisation. An other example is the vectorising package by Levesque. This vectoriser is in fact a precompiler which recognises implicit vector operations in Fortran Do-loops and replaces them with an equivalent vector syntax. Only selected C.P.U-intensive portions should be run by this vectoriser, the vectoriser diagnostics should serve as an aid to restructuring by the user. The package can be used for the C.D.C-7600, the Cray-1 and the Cyber 205. Advantages of automatic vectorisation are that programs when vectorised are still written in standard Fortran, and that many programs originating from a scalar processor will vectorise.

5.3.2 The C.F.T compiler

C.F.T is the optimising Fortran compiler for the Cray-1. The Fortran source programs are broken up into blocks of code by the compiler. A block in which vectorisation takes place is called a vector block. Compiler option (ON=B) can be used to display at the end of every program part which blocks have been created and which of these do vectorise. All optimisation is done on a block basis, and a detailed analysis of several programs in the benchmark test set showed that considerable inefficiencies result from this approach. A significant amount of improvement in efficiency of the code could be obtained if global optimisation in addition to the block optimisation took place. Maybe an extra pass over the object code is possible to optimise the code.

Unfortunately it is not possible to vectorise all the inner Do-loops. In the following paragraphs we discuss some of the conditions that prevent vectorisation and some suggestions for restructuring the code to make some of these loops vectorise. It is unnecessary to vectorise all Do-loops in the program. C.F.T has a special compiler option called Flowtrace (ON=F), which after a program has been executed generates a table containing information on how often every procedure has been called and how much time has been spend in every procedure. Flowtrace enables us to locate the procedure(s) in which most time is spent during execution. All our optimisation efforts should be concentrated on these procedures.

The automatic vectorisation by the compiler can be switched off by a compiler option (OFF=V) or by inserting a compiler directive in the source program (CDIR NOVECTOR). The directive might be useful in case of vectorisation of very short vectors which degrade performance. It is also possible that the compiler concludes that a loop is recursive but that the user knows that the loop is not recursive, in which case he can insert a compiler directive before that loop (CDIR IVDEP), which results in the compiler avoiding vector dependencies. This situation is sometimes referred to as imaginary recursion.The C.F.T compiler is still under development, and it is expected that the compiler will be able to vectorise more complex Do-loop constructs in the future. However it is hoped that further development will not solely concentrate on automatic vectorisation but more on scalar processing. A very interesting development is the change of 64 bit integer multiply and division to 48 bit. Since less instructions are required, a significant number of programs will perform faster.

5.3.3 Conditions preventing vectorisation

There are three main reasons why program loops may not vectorise, these are irregular addressing, recursion and control transfers.

Vectors are moved from memory to the V registers by the vector copy instruction. The vector copy instructions can access consecutive elements and elements with a constant spacing between them. So a row or diagonal of vector elements can be moved by one instruction. The vector arithmetic instruction operates on consecutive elements in V registers only. Hence a problem arises when the spacing between vector elements is not constant. In the following example assume that INDEX(I) contains random numbers, and that (;) indicates a new line.

Example: (J=INDEX(I);V1(I)=V2(J)*C) for I from 1 to N step 1

The pipeline principles discussed in chapter 1 and chapter 2 apply also to the processing of vector instructions. If in the same vector operation a result is used as operand, the operand might not be available when needed. If such a recursive vector operation was allowed on the Cray-1, processing would have to proceed without any overlap at all, and a significant degradation of performance would be the result.

Example: (TEMP=V1(I+L)*V1(I)*1/B;V1(I)=TEMP) for I from 1 to N step 1

If the sign of L is the same as the loop counter, then there is never any recursion. If L larger than N there is never any recursion. If the sign of L is the opposite of the loop counter (I) then there probably is recursion.

The vectorising method used by C.F.T involves a major change in the way the expressions in a particular loop are evaluated. In scalar mode the expressions are evaluated for the individual operand variables in a particular loop iteration. In case of vector mode the expressions are evaluated for groups of variables.

Example: (V1(I-1)=5;V2(I)=V1(I)) for I from 2 to 3 step 1

Scalar mode evaluation: V1(1)=5;V2(2)=V1(2);V1(2)=5;V2(3)=V1(3)

Vector mode evaluation: V1(1)=5;V1(2)=5;V2(2)=V1(2);V2(3)=V1(3)

Hence in vector mode the content of V1(2) is prematurely overwritten and therefore the wrong value, 5, is assigned to V2(2)

Examples of control transfers are: Branches inside loop, branch out of loop, branch into loop, call subroutine, and If statements. These transfers cause problems because they cannot be vectorised and prevent the group-wise processing of variables in a loop. Intrinsic functions like SIN, COS etc. cause no problems. certain types of IF statements can be replaced by specially built in functions like CVMGT, CVMGZ, AMAX1 etc. For more detail see Appendix 4.

5.3.4 Restructuring techniques

The objective of restructuring is to improve the performance of programs by increasing the amount of vectorisation that takes place. Appendix 4 contains a short guide to vectorisation, including examples of several restructuring techniques. For more examples see Higbie79 and Brode78.

The following techniques have been applied to a subset of the benchmark test set: maximising DO-loop lengths, removing IF statements, summation, switching inner and outer DO-loops, replacing temporary scalars with temporary vectors, making functions and subroutines 'inline' and solving irregular addressing.

If the Do-loop length is less than 64 it might be worth investigating if it is possible to extend the vector length.

The removal of IF statements is quite easy in some cases; as mentioned before we can make use of built in routines like CVMGT.

Summation inside a DO-loop can be vectorised by the current CFT compiler. Unfortunately this results in a degradation of performance for short vectors.

Switching inner and outer DO-loops can be very useful in removing recursion or extending the length of the DO-loop.

Replacing temporary scalars with tempoary vectors was necessary in the early versions of the CFT compiler, now it is vectorised automatically.

Making functions and subroutines 'in line' is useful when these routines do not vectorise and are not too big.

Irregular addressing can be solved by first concentrating the scattered array elements into a 'regular addressed array', then the required vector operation is performed and finally the results are scattered back again.

5.4 Vectorising a subset of the benchmark test set by restructuring

5.4.1 Hardware and software used

The vectorisation work was done on the Cray-1 serial-1 at the Science Research council Laboratory in Daresbury, the same machine on which we did the original benchmark test. A copy of the original test set was available on the Cray-1 discs. Modifications to this set were sent from the I.B.M front end system at Daresbury, the updates were kept in the I.B.M filestore. The Update facility on the Cray-1 was used to produce the updated versions of the programs. The Cray Update facility is very similar to the C.D.C Update facility. For references see at the end of this thesis.

Interactive communication with the I.B.M-TSO facility was possible via the S.R.C. network. The S.R.C. network connects many Universities and research establishments in Britain. Routing back of output from the Cray-1 via the S.R.C. network caused a lot of problems because the U.L.C.C. workstation had no printers connected to it. Therefore output had to be routed via Rutherford to Imperial College, and then via the Metronet high speed link to U.L.C.C. The Cray-1 had to be accessed via the I.B.M front-end at Daresbury. This front end is heavily overloaded and therefore it was only possible to access it in the evenings and weekends.

5.4.2 The results

The same subset of the benchmark that was used for the analysis in Chapter 4 was restructured. Before restructuring all double precision arithmetic was changed to single precision, since double precision prevents vectorisation.

At the end of this thesis are two micro fiches. One dated 18/12/80 contains the programs before restructering. The other dated 19/12/80 contains the programs after restructuring. Appendix 5 gives more information about the subset programs, and contains an index to the fiches. The restructuring techniques used are described in Appendix 4.

The execution times of the programs after restructuring are given in the last column of Table 4. Twelve out of twenty programs did vectorise. The total execution time of the subset was halved by restructuring.

The Table on the next page gives the amount of vectorisation that takes place, with and without restructuring. The ratios in the first column are for: (original benchmark subset with vectorisation switched off)/(orginal subset with vectorisation switched on and no restructuring). The ratios in the second column are for: (original subset with vectorisation switched off)/(restructured subset with vectorisation switched on).

If we compare the optimised subset for the CDC-7600 (FTN(2), single prec., better mem. mngment) with the restructured subset for the Cray-1 we get the ratio (3107/1176)=2.64. However if the CDC-7600 code was also restructured this ratio would be lower. Expected improvements from restructuring 7600 code, to achieve a better pipeline utilisation, are on average in the order of ten to twenty percent.

id	original	restructured
U2	1.0	1.0
U3	1.0	1.0
U4	1.0	1.0
U6	1.0	2.57
U8	1.0	3.39
U9	1.0	1.0
U10	1.06	2.17
U13	1.05	2.20
U14	1.02	1.02
U15	1.07	6.25
U16	1.16	1.2
U30	1.0	1.0
U31	1.87	1.87
U32	1.14	1.23
U51	1.84	3.62
U52	1.11	1.11
U57	1.03	3.42
U58	1.10	1.55
XXX	1.0	3.18
TIM	1.0	2.0

5.4.3 Problems encountered

The Scilib routines proved extremely useful, although some of the routines gave little or no improvement in performance. Examples of such routines are: COPYV, MAX and SSUM. It would be useful if the Scilib documentation would give some indication of the improvement in performance that can be expected. This information is very useful when deciding whether restructuring the code to include a Scilib routine is worth the effort. The current Scilib documentation is insufficient, e.g. documentation on matrix inversion is very poor especially in the case when the matrix actually being inverted is smaller than the matrix declared in the DIMENSION statement. Another problem encountered is the fact that a lot of programs manipulate symmetric matices, which are represented as triangular and not as square matrices. Scilib routines only handle square matrices. Before the Scilib operation the triangle has to be converted to a square matrix and after the Scilib operation the square matrix has to be converted back to a triangle again. These overheads are considerable, especially when the vectors are short. It would be useful if the Scilib routines themselves could do the conversion work. An alternative is to convert all triangles and references to triangles in the entire program, which often means rewriting the program entirely.

The analysis of the object code produced by the C.F.T compiler is in many cases far from optimal. The reason is probably the block orientated optimisation by C.F.T, inefficiencies apparently occurring because of lack of communication between the blocks. This is demonstrated by program TIM. When most common subexpressions were removed the program performed 25% faster on the Cray-1 and 4% faster on the 7600. The optimisation in C.F.T can be improved by an extra pass over the object code, the compilation speed of C.F.T is on average about three times faster than F.T.N, therefore the extra compilation time should be no problem.

The latest version of the C.F.T compiler can recognise summations in DO-loops and vectorises them. This results in many cases in a degradation of performance. This facility should only be availble as an option of the compiler, so that it can be switched on when appropriate.

The Flowtrace option in C.F.T was extremely useful. Unfortunately the overheads of Flowtrace are considerable, especially when procedures are being called a great many times. In many cases the information supplied by Flowtrace was insufficient, a procedure can contain several DO-loops and it is not obvious in which DO-loop most time is spent. In such a case counters are put in the loops to find which is executed most often. An alternative to Flowtrace would be simply to give the number of times every procedure is executed. The timings given by Flowtrace should be another compiler option, because they cause the very high overhead. A useful thing to have would be a compiler directive to indicate that the number of iterations in each loop of a particular procedure must be counted and printed at the end of the program.

It is important to realise that the optimised programs in some cases may be 'data set' dependent, since for a different data set more time might be spent in other parts of the program which have not been optimised.

5.5 Conclusions

From the results we conclude that restructuring of Fortran source code to help the automatic vectorisation by the CFT compiler is very effective. The performance of the Cray-1 was improved by a factor two after restructuring the source code of a subset of the benchmark test set. It took us on average one or two days to restructure a program. The time it takes an inexperienced user to restructure his program is half a 'manweek'.

Scilib, the library of scientific subroutines and basic vector operations, is extremely useful. But the Scilib documentation is insufficient. Scilib routines should be able to handle triangular shaped matrices.

The Flowtrace facility is extremely useful, but the time overheads are considerable. An alternative loop-based Flowtrace, without timings, is suggested.

CHAPTER 6

THE CONCLUSIONS

6.1 The results of the investigations

Summary

The first investigation in our project consisted of a comparison of the Cray-1 and the C.D.C-7600 at component level.

The investigate the amount of programming effort to convert Fortran programs from the 7600 to the Cray-1, and to study the efficiency of the Fortran compilers on both machines we run a live benchmark.

A subset of the benchmark was analysed, to explain the enormous variance in performance ratios of the individual programs.

Finally the effect of automatic vectorisation was investigated.

The results of the comparison at component level

In this investigation we compare the Cray-1 and the C.D.C-7600 hardware. We look at memory organisation, memory access times, instruction sets, instruction timings pipelining and vectorisation.

The following differences concerning memory were observed: The Cray-1 has far more register store available, which reduces the number of references to main store. The access times to the bipolar memory of the Cray-1 is shorter, 137 versus 220 nano seconds. The access time to the Large Core Memory of the 7600 is dependent on the way it is being accesssed, random access results in an access time of 467 nano seconds. References to different banks, in S.C.M, L.C.M and the Cray-1 memory can be processed simultaneously. The code density in the Cray-1 memory is higher. The instruction stack of the Cray-1 is five times larger than on the C.D.C-7600.

The following differences at instruction level were observed: Full word integer multiplication and division on the Cray-1 is done in software. Integer multiplication of 48-bit integers on the C.D.C-7600 is done in hardware. Double precision arithmetic is done in software on both machines, but on the 7600 less instructions are required because of the availability of double precision results of the floating point multiply and divide units. The conditional branch instruction on the Cray-1 is less flexible than on the 7600, since only register A0 and S0 can be used for testing. The floating point multiply unit on the Cray-1 is 1.57 times faster than on the 7600. The floating point add is 2.57 times faster than on the 7600. Double precision floating point multiply is five times slower on the Cray-1. Integer multiply is three times slower on the Cray-1, but has more accuracy.

The Cray-1 and the 7600 are both pipelined. The Cray-1 has fully segmented pipelining. The 7600 is not fully segmented, the floating point multiply having two period segmentation and the divide unit not being segmented at all. Vector instructions on the Cray-1 can be processed simultaneously if they do not conflict with each other in resource needs. So on the Cray-1 more than one result per clock cycle can be produced.

From these results we conclude that it is fair to assume that the scalar performance of the Cray-1 is more than twice the 7600.

Results of the comparison at benchmark level

The test set of the benchmark consisted of 68 programs taken from the U.L.C.C workload. The test set contained about 54000 lines of code and 34000 card images of data. It took us 6 manweeks of programming effort to convert these programs from the C.D.C-7600 to the Cray-1. Appendix 2 gives a description of the known differences between Cray Fortran (CFT) and C.D.C Fortran (FTN). The conversion gave no serious problems.

The benchmark test set was run three times on both machines. The average throughput time on the 7600 was 2 hrs, 17 mins, 2 secs. The ratio (7600 throughput time)/(Cray-1 throughput time) was 1.65. The ratio (7600 jobtime)/(Cray-1 jobtime) for individual programs varies from 0.39 to 9.57. Sixteen of 68 jobs have a performance ratio of below two. Almost fifty percent of jobs have performance ratios between 2 and 4. Because of the enormous variance in performance ratios it was very difficult to draw conclusions from these results. Because 40 percent of the throughput time on the Cray-1 is used by only two programs (U17 and TIM) there is a danger that the total throughput time is biassed by these programs. Therefore there is clearly need for an analysis of the individual programs.

The results of the analysis of the test set

A subset of twenty programs was taken from the bencmark test set. The analys showed that the performance of the 7600 was affected by the following factors: the use of the MNF compiler, the use of the FTN compiler with optimisation level 0 or 1, double precision arithmetic and memory management by the programmer. We concluded that 4 percent of the total time used by the subset was wasted by MNF. The use of optimisation level 0 or 1 with the FTN compiler resulted in 5.7 percent of the time being wasted. Poor memory management by the programmer resulted in 9 percent of the time being wasted. the use of double precision arithmetic resulted in 1.8 percent being wasted.

The performance of the Cray-1 was affected by the following factors: automatic vectorisation, double precision arithmetic and integer arithmetic. The automatic vectorisation resulted in a 9 percent shorter processing time for the subset. The use of double precision arithmetic resulted in a 10 percent longer processing time. Further investigation is required to see how much is gained in performance when the user restructures his code to help automatic vectorisation.

The results of vectorising a subset of the test set

The same subset of twenty programs that was used for the analysis was restructured to help the automatic vectorisation by the CFT compiler. The restructuring resulted in an improvement in performance of a factor two.

The performance ratio (7600 optimised)/(Cray-1 restructured) for the subset was 2.65.

The Scilib routines were extremely useful, but the documentation was insufficient.

6.2 Major conclusions

From the comparison of the CDC-7600 and the Cray-1 at hardware level, we conclude that the Cray-1 scalar performance for the processing of a workload dominated by single precision floating point arithmetic (scientific workload) is more than twice a CDC-7600. This conclusion is based on instruction timings, store access times and architectural features of the two machines.

The comparison of the two machines for a general purpose scientific workload modelled by a live benchmark suggests a performance for the Cray-1 of 1.65 times a CDC-7600. The benchmark test set consisted of jobs taken from the U.L.C.C workload. None of the programs had been optimised for the Cray-1, so this is the 'minimum performance' estimate for the Cray-1. The fluctuations in the performance ratios of the individual programs in the benchmark test set were very high, the ratios ranging from 0.2 to 9.

An analysis of twenty programs taken from the U.L.C.C benchmark test set showed that most of these fluctuations are caused by the following factors: automatic vectorisation, double precision floating point arithmetic, integer arithmetic, random access to Large Core Memory use of two different Fortran compilers on the C.D.C-7600, differences in compilation times and the use of different levels of optimisation with the FTN compiler. After these factor had been removed, for as far as possible, 16 out of twenty programs had a performance ratio between 1.5 and 2.

The automatic vectorisation by the C.F.T compiler on the original U.L.C.C benchmark test set was not very effective. In the subset of twenty programs from the benchmark test set only 6.8 percent in performance was gained by automatic vectorisation.

After this subset had been restructured to help the automatic vectorisation process the total execution time for the subset on the Cray-1 was halved. Hence this is the performance that is obtained if all users were to restructure their programs. The time it takes an inexperienced user to restructure his program is estimated at half a 'manweek'. Restructuring can in some cases result in 'unstructured code' which is difficult to modify or to extend.

When programs are completely rewritten with the Cray-1 and the CDC-7600 in mind further improvements in performance are possible. In many cases the performance for the Cray-1 is more than five times a CDC-7600. Many examples of this are given in literature.

6.3 Suggestions for further research

Further research is required to investigate how the performance of the Cray-1 and the CDC-7600 can be improved by modifying the architectures. The major weakness of both machines, but especially the 7600, is that operands cannot be delivered fast enough to the scalar registers. The Cray-1 attempts to solve this by the provision of 64 intermediate T registers, but since these registers have to be allocated at compile time it is difficult for a compiler to make effective use of them. An alternative would be to have an automatic cache to replace them. Research can be done by simulating the architectures.

Further research is required to investigate how the Fortran compiler of the Cray-1 (CFT) and the 7600 (FTN) can be optimised to improve performance. A very interesting problem is the design of a sophisticated algorithm for the optimal use of the T registers on the Cray-1. Another problem is that the CFT compiler breaks programs into blocks, optimisation taking place on a block basis. It should be investigated whether global optimisation is possible by an extra pass over the object code. This should also include a reschedule of the instructions, to make them go more smoothly down the pipeline.

APPENDIX 1

DESCRIPTION OF THE CDC-7600 and CRAY-1 ARCHITECTURES

1 The CDC-7600 computer system

1.1 Basic system description

The design of the CDC-7600 architecture is based on the CDC-6000 series machines. The CDC-6000 series was designed between 1960 and 1964. The design was aimed at two kinds of use, the very large scientific problem and the time sharing of smaller problems. For the large problems, a high speed floating point central processor to a large central storage was obviously chosen. An important feature of the 7600 and its ancestors is the isolation of this central arithmetic from any peripheral activity. The CDC-7600 computer consists of a central processor and several peripheral processors. The central processor can communicate with the central storage and in addition has several fast storage registers within itself.

The system storage consists of the Small Core Memory (SCM) and the Large Core Memory (L.C.M). SCM consists of 16 or 32 banks of memory capable of simultaneous operation, delays occur when successive references are made to the same bank (bank conflict). The size of SCM is 32K or 64K words. LCM is a 4 or 8 bank linear select type memory, every bank is 8 words wide. A reference to an LCM bank results in 8 consecutive words being transferred to an eight word register called the Superword, often referred to as Sword. Every bank has a Sword associated with it. The size of LCM is 256K words.

Computation is performed by a set of functional units. Each unit consists of circuitry capable of performing a specific operation (e.g Add) and certain registers for holding of arguments and results. An arithmetic instruction requires that the functional unit involved reads two arguments from the set of data registers, compute a result and store the result into a data register(three address machine). The functional units are independent and capable of simultaneous operation (pipelining).

The ten peripheral processors (PPU's) have access to the central memory on one side and the peripheral channels on the other. The executive control of the system is always in one of these peripheral processors, with the others operating on assigned peripheral or control tasks.

Unlike the IBM-370 which can read memory in four different sized words, namely bytes, half words, words and double words, the CDC-7600 instructions always fetch exactly one 60 bit word on a read from SCM and always store exactly one 60-bits word on a write.

The clock cycle of the CDC-7600 is 27.5 nanoseconds. Since the central processor is theoretically capable of processing a result every cycle, the power of the processor is 36 MIPS (millions of instructions a second). Ordinary Fortran programs have a much lower performance.

1.2 The central processor description

Twenty four registers are provided to minimize memory references for arithmetic operands and results. They are divided into eight address registers (named A0-A7) 18 bits long, eight index registers (named B0-B7) 18 bits long, eight operand registers (named X0-X7) 60 bits long.

When a memory address is put into A1, the contents of that address is automatically loaded in X1. Similiarly loading A2 to A5 with an address causes X2 to X5 to be loaded with the contents of the memory word in the corresponding A register. When an address is loaded into A6 or A7 the contents of the corresponding X register is stored into memory. A0 and X0 are independent of each other and have no connection with SCM. Loading an A register is the only mechanism for transferring data between SCM and register. B0 is by hardware identical to zero.

To represent negative numbers, one's complement is used. This applies to the 18 bit addresses as well as to the 60 bit arithmetic operands. Note that one's complement implies the existence of different representations for +0 and -0.

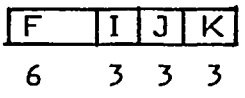
The X registers also serve as addres registers for referencing single words from LCM. X0 is used as the LCM relative starting address in a block copy operation. A0 is used as the relative starting address in a block copy operation and for intermediate results.

1.2.2 Instruction formats

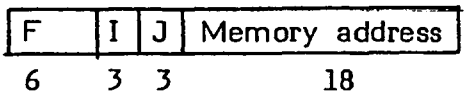
The CDC-7600 is a three address machine. There are 78 instructions which can be divided into 15 bit, 30 bit and 60 bit instructions. The 6000 series used 6 bit opcodes, hence more bits are needed on the 7600. A seven bit opcode would be awkward because most instructions require the registers, each of which requires three bits to specify. A 7 bit-opcode plus three 3-bit register fields yields a 16-bit instruction which does not fit neatly in a 60 bit word the way a 15-bit instruction does. In order to expand the opcode the leftmost three-bit field of Format-2 (see below) became part of the opcode, resulting in format 3. Format 3 is used for branch instructions, in fact this was already done on the 6600 because not enough bits were available to encode all the instructions.

The instructions of the C.D.C-7600 are:

Format 1 :



Format 2 :



Format 3 :

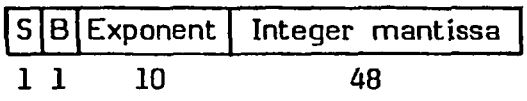


F = Operation code (which also defines the register classes involved.
I = Index of the result register
J = Index of the first operand register
K = Index of the second operand register

In certain (shift and mask) instructions the J and K field together form a 6 bit operand. The processor is able to execute 30 bit instructions only if they are stored in the same instruction word.

Data representation

Negative numbers are represented using 1's complement notation. Integers can be 60 bits, 48 bits and 18 bits long. However when integers longer than 48 bits are multiplied or divided wrong results are produced and no error message is given. Floating point arithmetic expresses a number with the general expression MB^E. M is the mantissa, B is the base number and E is the exponent. B is assumed to be 2 for binary coded quantities. The format of floating point numbers on the C.D.C-7600 is as follows:



S = mantissa sign

B = bias

Characters on the 7600 take up 6 bits, characters can be left justified with 54 unused bits to the right or right justified with 54 unused bits to the left. Note that a string of ten characters fits into one 60 bits word.

Arithmetic instructions

The floating point arithmetic instructions on the 7600 generate double precision results. Two instructions, one single precision and one double precision, are required to retrieve an entire double precision result. Integer multiply is done by the double precision floating multiply instruction, both operands must have an exponent value of +0 or -0 and neither coefficient can be normalised. The result is sign extended to 60 bits and sent to an X register. Integer divide is done in software.

Data copy instructions

The C.D.C-7600 has instructions available for moving 18 bit numbers from any registers (A,B or X) to any registers. There is an instruction to copy a 60-bit X register into another 60-bit X register.

No special instructions for loading registers from memory or storing registers in memory because setting A1 through A5 automatically loads the associated X register from memory and setting A6 or A7 automatically stores X6 or X7 into memory.

It is not possible to move blocks of data from SCM to the registers. However it is possible to move a block of consecutive words in LCM to SCM, at the rate of one word per clock cycle.

1.2.4 Pipelining

The central processor of the C.D.C-7600 is based on a high degree of functional parallelism (pipelining), this is provided by segmented functional units and a number of essential supporting facilities.

The nine functional units are independent of each other and may operate simultaneously. A special case is the floating divide unit, which can not be pipelined because the algorithm to divide is essentially iterative. Therefore only one divide instruction can be processed at a time. The floating point multiply unit has two-clock-period segmentation, this means that the next multiplication can only start two periods after the previous multiply has started. All the other functional units have 1 clock period segmentation. The functional units on the 7600 are: Boolean unit, floating multiply unit, shift unit, normalise unit, floating add unit, long add unit, floating divide unit, population count and fixed add.

The instruction stack is a group of twelve 60-bit registers that hold program instruction words for execution. The instruction stack is filled two words ahead of the program address currently being executed. A small loop occupying up to ten instruction words may be entirely contained within the stack. When this happens the loop may be executed repeatedly without further references to SCM.

A group of twelve 18-bit address registers are associated with the instruction stack. These registers, called the instruction address stack (IAS) hold relative SCM program addresses on a one-for-one basis with program words in the instruction stack. All entries in the instruction stack are compared with the current program address. If coincidence occurs for any entry, the corresponding entry in the instruction stack is sent to the Current Instruction Word register.

The CIW register is divided into four fifteen bit parcels. All four parcels are loaded when an instruction word is read from the IWS. An instruction isssues from the CIW register when conditions in the functional units and operating registers are such that the instruction will be executed without conflicting with previously issued instructions. The other parcels are then shifted left in the CIW register by either 15 or 30 bits, depending on the instruction format. The issue time of 15 bit instructions is 1 clock period, and the issue time of 30 bit instructions is 2 clock periods.

A major advantage of pipelining is that a significant amount of overlap in the processing of instructions is achieved without the programmer having to rewrite his programs. However significant improvements in the utilisation of the pipeline can be obtained if the compiler rearranges the sequence of the instructions.

2 Architecture of the Cray-1 computer system

2.1 Basic system description

The Cray-1 is a large scale computer, particularly intended for large numerical calculations and other applications where an appreciable amount of computing power is needed.

A significant portion of the large numerical calculations involve manipulating arrays of numbers (vectors). Therefore vector operations were included in the instruction set. The advantage of vector instructions is that the overheads of loop control and array index calculation are avoided, instead there is only the overhead of the vector instruction.

By introducing vector registers the startup time of vectors is reduced, even short vectors can be processed faster in vector mode than in scalar mode. This is especially important in case of a general purpose workload where a large proportion of the workload concerns short vectors.

The processor of the Cray-1 is pipelined. The functional units are fully segmented. Note that processing of vectors result in an optimal utilisation of the functional unit (one floating point result every clock cycle). When successive vector instructions do not conflict with each other they can be processed simultaneously, so more than one result per cycle is produced.

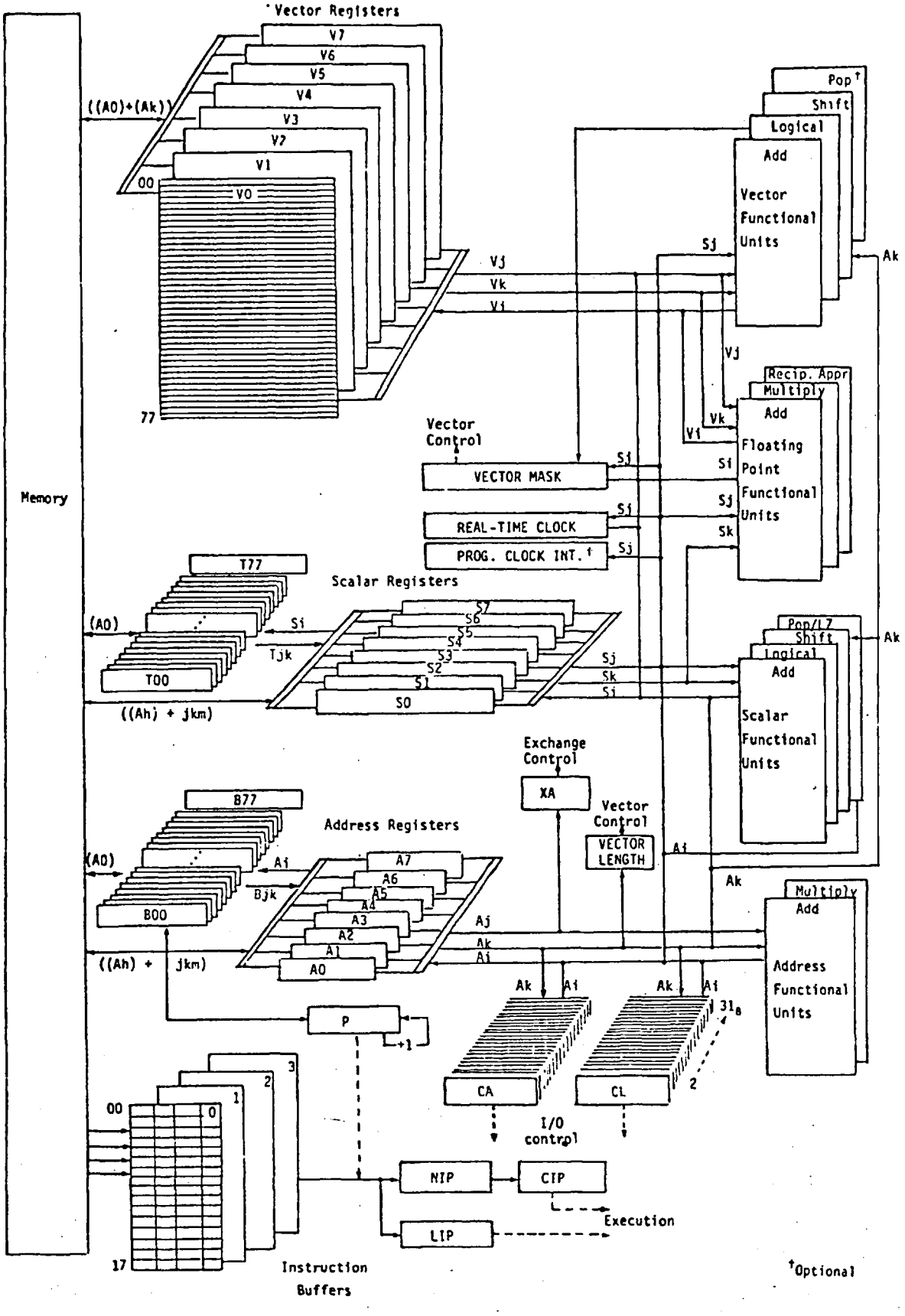
The Cray-1 has a very short clock cycle (12.5 nano seconds) this is due to the use of very fast 'state of the art' components and by keeping the wires in the machine as short as possible, thereby reducing the propagational delays. For the throughput rate of a computer to increase it is not sufficient solely to speedup the arithmetic unit. If the functional units that have been speeded up spent most of their time waiting for operands to be supplied to them, nothing is gained. Therefore the Cray-1 has an extremely fast bipolar main memory, in addition there is 4K bytes of fast register store available.

The standard Cray-1 processor handles all I.O itself. However a recently released processor the Cray-1 S has peripheral processors comparable to the CDC-7600, this processor will perform much better in case of I.O bound jobs than the standard Cray-1 processor.

The small number of distinct hardware components used in the Cray-1 leads to increased reliability through simplified circuit design. Only four different integrated circuits and two different resistors have been used.

The size of the main memory of the Cray-1 is 0.25M, 0.5M or 1M words, wordlength being 64 bits. A 1M word memory is interleaved in 16 independent banks. A 0.5M word memory is interleaved in 8 independent banks. There are 72 modules per bank, each module contributes 1 bit to a 64 bit word. The other 8 bits are used to store an 8 bit check byte required for single bit error correction double bit error detection (SECDED). Data words are stored in 1-bank increments throughout memory. This organisation allows 16-way interleaving of memory accesses and prevents bank conflicts except in the case of memory accesses that step through memory with either 8 or 16 word increment. The cycle time of each bank is 50 nanoseconds. The maximum transfer rate, asssuming full interleaving, is thus 16*64 bits in 50 nano seconds or 320M words/second. In practice, instructions are supplied to the instruction buffers at this rate. Data is supplied to registers at a maximum rate of 80 M words per second (one word every clock period). This rate will be reduced in case of bank conflicts.

The Cray-1 is equipped with up to 12 input and output channels connecting the main memory to external devices, of which any number may be active at one time. Channel widths vary; for a 16 bit channel, transfer rates of 160M bits per second are achievable. The data transfer rate of a single disc storage device used as backing store/ file store, is 36 M bits per second, a number of such devices can be active at one time.


Vector Registers
V7
V6
V5
V4
V3
V2
V1
V0
00
77
((A0)+(Ak))
Memory
Vj
Vk
Vi
Pop†
Shift
Logical
Add
Vector
Functional
Units
Sj
Ak
Recip. Appr
Multiply
Add
Floating
Point
Functional
Units
Si
Sk
Vector
Control
VECTOR MASK
REAL-TIME CLOCK
PROG. CLOCK INT.†
T77
T00
(A0)
Si
Tjk
Scalar Registers
S7
S6
S5
S4
S3
S2
S1
S0
((Ah) + jkm)
Pop/LZ
Scalar
Functional
Units
Exchange
Control
XA
Vector
Control
VECTOR
LENGTH
Ai
B77
B00
Address Registers
Bjk
A7
A6
A5
A4
A3
A2
A1
A0
Aj
Ak
Multiply
Address
Functional
Units
((Ah) + jkm)
P
+1
CA
CL
I/O
control
NIP
CIP
LIP
Execution
Instruction
Buffers
†Optional

2.2 The registers

A special feature of the Cray-1 is the large amount of register storage that is provided, in total more than 4K bytes of registers storage.

The arithmetic functional units take input operands from and store result operands only to the A, S and V primary registers. The two intermediate registers (B and T) can only be accessed by the data copy instructions.

There are eight vector registers. Each vector register consists of 64 words each word is 64 bits. The advantage of vector registers is that the vector operands can be supplied fast enough and the result vector can be stored fast enough. A disadvantage is that the logical vector being manipulated might be longer than the fixed length register. Therefore vector operations that manipulate vectors of length 64 or multiples of 64 perform optimally. A block of data may be transferred between vector register and memory by a single instruction.

The Cray-1 has sixty four 24-bit B registers, which are used as intermediate storage for the A registers. The Bregisters typically contain counters and addresses. Blocks of data may be transferred between memory and the B registers by a single instruction.

There are sixty four 64 bit T registers, they are used as auxiliarly storage for the S registers. A block of data may be transferred between memory and the T registers by a single instruction.

There are eight 24-bit A registers. They are used as address registers for memory references and as index registers. A registers can be loaded directly from memory.

There are eight 24-bit S registers. They are used for data handling and for scalar operations. It is possible to load S registers direcly from memory.Block moves between memory and S or A registers are not possible

In addition to support-registers used in I.O operations (e.g. a channel address and channel limit register for each channel) two scalar registers are provided to support vector operations

VM: Vector mask register, used to select vector elements in merge or sort operations.

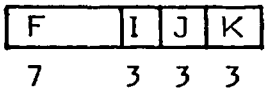
VI: Vector length register, used to define the lengths of vectors used in vector operations.

2.3 Instruction format

The Cray-1 is a three address machine. Each instruction is either a one parcel (16 bit) instruction or a two parcel (32-bit) instruction. Instructions are packed four parcels per word. Parcels in a word are numbered from left to right as 0 through 3 and can be accessed in branch instructions. A two parcel instruction may begin in any parcel of a word and may span a word boundary. A two parcel instruction that begins in the fourth parcel of a word ends in the first parcel of the next word.

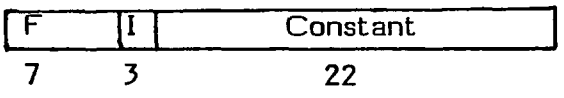
The formats are:

Format 1:



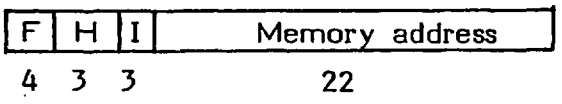
16 bits

Format 2:



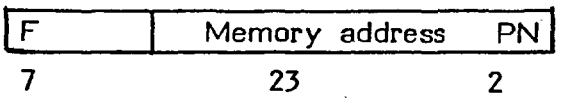
32 bits

Format 3:



32 bits

Format 4:



32 bits

F = operation code
H = index of index register
I = index of result register
J = index of the first operand register
K = index of the second operand register
PN = parcel number

Format 1 is used for arithmetic and logical instructions. The J and K fields designate the two operand registers or are combined to designate a 6 bit B or T register address. In case of shift and mask instructions the 6-bit combined JK field specifies a shift or mask count. Format 1 is also used for instructions that enter immediate constants into A registers. The combined JK filed gives a 6-bit quantity.

Format 2 is used for instructions that enter immediate constants into A registers. The constant field gives a 22-bit quantity.

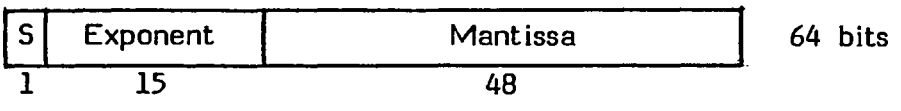
Format 3 is used for instructions that transfer data between S or A registers and memory.

Format 4 is used for branch instructions. Since the parcel number is included in the address, it is allowed to branch to a quarter-word-boundary.

2.4 Data representation

Integers are represented in 2's complement form. Floating point numbers are represented in signed magnitude form. The format is a packed signed binary mantissa and a biased binary integer exponent. The mantissa is a 49-bit signed magnitude value. The exponent is represented as a biased integer in 15 bits. The bias that has been added to the exponent is 40000 .

The unbiased exponent range is : 2^{-20000} to 2^{+17777}



The Fortran compiler on the Cray-1 uses ASCII 8 bit characters. Hence 8 characters fit into one word.

2.5 Instruction types

Data copy instructions

A block of data may be transferred to or from memory. A similiar instruction exists for the T register. A block load from memory to B or T register takes 14 clock cycles plus 1 cycle per word to be transferred. A block load from memory takes 6 clock cycles plus 1 cycle per word to be transferred.

To copy the contents of a T register to a S register takes 1 cycle, from B to A register also takes one cycle. Block moves from T to S registers do not exist.

The A and S registers can be loaded directly from memory, the instructions issue in two cycles and the data is available in 11 cycles.

Vectors may be transferred from the V registers to and from memory.
In case of a transfer from V register to memory, the V register is ready after VL clock periods. In case of a transfer from memory to V register, the V register is ready after VL plus 9 clock periods.

An important point is that memory must be quiet before issue of the B and T register block copy instructions. Subsequent instructions may not issue for 14 clock periods plus one clock period for every word being transferred

Arithmetic instructions

The only integer functional unit provided is for 24 bit integer multiplication. Multiplication and division of 64 bit integers is accomplished by packing the integers into floating point format with a zero-exponent value. The floating point multiply unit recognises the conditions where both operands have zero exponents as a special case and returns the upper 48 bits of the product of the mantissa's as the mantissa of the result and leaves the mantissa field zero. Division of integers would require that they first be converted to floating point format and then divided using the floating point units.

No functional unit is provided for floating point division, this is because the algorithm to divide is essentially iterative which makes the segmentation required for pipelining impossible. To perform floating point division, the Cray-1 is provided with a reciprocal functional unit which can be fully segmented. Floating point division is achieved by a division algorithm based on Newton's method, which makes use of reciprocal approximation. Floating point division is done in the Cray-1 Fortran compiler by an 'in line' procedure of 4 instructions. In vector mode, results are produced at a one clock period rate. These results may be used in other vector operations during chaining because all functional units in the Cray-1 have the same result rate.

The Cray-1 does not provide special hardware for performing double or multiple precision floating point operations. Double precision computations with 95 bit accuracy are available through software routines.

2.6 The processing of instructions

2.6.1 The instruction stack

The instruction stack in the Cray-1 consists of 4 instruction buffers, each of which holds 64 consecutive 16-bit instruction parcels. Instruction parcels are held in the buffers prior to being delivered to the NIP or CIP registers.

The beginning instruction parcel in a buffer always has a parcel address that is an even multiple of 100. This allows the entire range of addresses for instructions in a buffer to be defined by the high-order 16 bits of the beginning parcel address. For each buffer, there is a 16 bit beginning address register that contains this value.

The beginning address registers are scanned each clock period. If the high order 18 bits of the P register match one of the beginning addresses, an in-buffer condition exists and the proper instruction parcel is selected from the instruction buffer. An instruction parcel to be executed is normally sent to the NIP. However, the second half of a two-parcel instruction is blocked from entering the NIP and is sent to the LIP, instead, and is available when the upper half issues from the CIP. Instructions issue from CIP at a maximum rate of one per beat. This rate will decrease if there is delay in fetching an instruction.

On an in-stack condition, if the instruction is in a different buffer than the previous instruction, a change of buffers occurs necessitating a two clock period delay of issue.

In case of out-of-stack condition new instructions must be loaded into one of the instruction buffers for memory. The first group of 16 parcels delivered to the buffer always contains the instruction required for execution. The branch out-of-buffer time is a constant 14 clock periods. The remaining groups arrive at a rate of 16 parcels per clock period and circuilarly fill the buffer.

Both forward and backward branching is possible within the buffers. A branch does not cause reloading of an instruction buffer if the instruction being branched to is within one of the buffers. Multiple copies of instruction parcels cannot occur in the instruction buffer.

Functional unit reservations

Each functional unit is a separate pipeline, accepting scalar operands and generating a scalar result at a rate of one result per clock cycle. A functional unit may be supplied with operations at a rate of one per cycle, either by issue of instructions at this rate (scalars) or by repeated application to an operation on a single pair of vector elements. Hence no reservation need be made of a functional unit applied to a scalar instruction, since the rate of instruction issue can never exceed the rate at which a functional unit can accept instructions. Reservations must be made of a functional unit applied to a vector instruction, however, since a following instruction could issue whilst the unit is unable to accept a further instruction.

Functional units are reserved by vector instructions for VL+4 clock periods. Memory is reserved for VL+5 clock periods on a write operation, VL+4 clock periods on a read operation.

Register reservation

Registers are reserved at instruction issue time. The reservation is removed by the appropriate functional unit. Reservations are used to delay issue of instructions.

For scalar instructions, the result register is reserved for functional unit time + 2 beats after instruction issue.

For vector instructions, the result register is reserved for the functional unit time + (VL)+2 clock periods. The result register is reserved for the functional unit time +7 clock periods if the vector length is less than 5.

For vector instructions, the vector operand registers are reserved for VL clock periods. Vector operand registers are reserved for 5 clock periods, if the vector length is less than 5. The vector register used in a block store to memory is reserved for VL clock periods. Scalar registers are not reserved.

In general, an entire vector register is reserved during the period whilst a functional unit requires access to any element in that register. This reservation may be lifted in respect of one element of the result register at 'chain slot' time, the clock period ocurring at functional unit time plus 2 clock periods. During this clock period, the result is available for use as source operand in a subsequent vector operation; during subsequent clock periods, successive result elements become available as successive source elements. Vector chaining may be applied to any sequence of vector operations with the exception of vector register to memory transfers when main memory bank conflicts apply (when distance between consecutive vector elements in memory is 8 or 16).

Instruction issue conditions

Instructions issue from the CIP register at a maximum rate of one per clock cycle. The following conditions must be met before issue of a scalar instruction:

1) The functional unit must be free. No conflict can arise with other scalar instructions because they do not reserve the functional units. However vector floating point instructions do reserve the floating point units. Memory references may be delayed due to conflicts.

2) The result register must be free. Scalar instructions place reservations only on result registers.

3) The operand register must be free. No reservation is placed on the operand registers by scalar instructions.

4) One input path exists for each group of the four register groups (A,B,S and T). The result register group must be free at the time the results would be stored.

5) No block transfers between main memory and intermediate registers must be going on.

The following conditions must be met before issue of a vector instruction:

1) The functional unit must be free.

2) The result register must be free.

3) The operand registers must be free or at chain slot time.

4) Memory must be quiet if the instruction references memory.

5) No block transfer between main memory and intermediate registers must be going on.

APPENDIX 2

FTN and CFT differences

The actual differences at the time of the conversion of the U.L.C.C benchmark test set are listed below:

1) CFT variables can contain up to eight Hollerith characters: e.g. 8H01234567; and not 10H0123456789 as in FTN.

2) CFT variables contain 64 bits; integer values FLOATed or FIXed must be 46 bits or less. FTN allows 47 bits.

3) CFT DO loop iteration count is calculated as a 24 bit quantity (8,388,607 maximum), FTN uses 18 bits (131,071 maximum).

4) In CFT, subscripts are evaluated as 24 bit quantities as are all intermediate results.

5) CFT uses ASCII 8 bit characters. FTN uses CDC scientific 6 bit characters.

6) CFT does not recognise non-ANSI multiple replacement statements: e.g. A=B=C=D=0. FTN does recognise them.

7) CFT does not recognise more than one statement per line:
e.g. A=B $ C=D. FTN does recognise them.

8) The CFT shift function is left circular with a positive shift count (second argument). The CDC SHIFT function is left circular for positive shift counts; right end off with sign extension for negative counts. CFT does not have a right shift function with sign bit extension.

9) The Cray-1 is a two's complement machine; division by a power of two differs from a right shift when the numerator is negative. The CDC 7600 is a one's complement machine.

10) A CFT ENTRY statement with no dummy argument list may not be called with an actual list specified.

e.g.

```
CALL SAM(A,B,C)
....
....
....
SUBROUTINE GEO(X,Y,Z)
....
....
....
ENTRY SAM
```

is illegal, FTN allows it.

11) CFT supports ANSI standard PROGRAM statement specification and file or dataset replacements/equivalences are handled with operating system job control language statements. FTN handles them in the PROGRAM statement.

12) The Maximum line length for FORMAT conversion is 152 characters in CFT and cannot be changed by a PROGRAM card: e.g.PROGRAM SAM (TAPE2=4000/153) is not allowed. FTN allows it.

13) CFT requires run-time compatibility of type between an I.O. list item and the FORMAT conversion: e.g.

```
      PRINT 10, INTEGER
   10 FORMAT(F10.0)
```

causes a run-time abort, while FTN allows it.

14) In CFT, the first character (not the first non-space character) in a variable FORMAT must be the open parenthesis:

A(1)=8H(F10.0) runs while A(1)=8H(F10.0) does not, FTN allows both constructs.

15) CFT does not allow indexing into an array for a format: e.g.

```
READ(10,INFMT(I))X
```

does not work, it works on FTN.

16) CFT does not allow DATA(X=0.0) whilst FTN does.

17) CFT does not allow using an array name to reference the first element of an array in a expression: e.g.

```
ARRY=0
```

is illegal and does not mean ARRY(1) as in FTN.

18) In CFT, execution of a computed GO TO statement continues at the next statement if the expression is out of range: e.g. the computed GO TO statement in

```
I=3
GO TO(10,20)I
```

does not cause a jump or a run-time error. In FTN it does.

19) CFT DO-loops are processed as per ANSI Fortran 77 standard specification; the iteration count being computed before the loop is executed. The loop is bypassed if the count is zero or negative. FTN DO-loops are processed as per ANSI Fortran 66 standard specifications; the loop is executed once if the count is zero or negative

20) CFT does not support the non-standard list directed I.O: e.g.

```
READ(5,*)A,B,C,D,.....
```

is illegal. FTN does support it and also supports other variants of this which have the implication of requiring re-format of the input data to enable programs to work on the Cray-1.

21) CFT does not recognise a LEVEL 2 statement. FTN uses this non-standard statement to place variables into Large Core Memory.

22) Program path taken as a consequence of testing floating point numbers for equality can differ due to differences (not errors) in the arithmetic unit and different machine word lengths. This technique is dangerous in any circumstance.

23) CFT requires that a STOP statement must normally terminate a main program. FTN allows a RETURN as a substitute.

24) CFT only allows a 'C' or an '*' in column 1, designating comment lines. FTN does allow other characters.

25) CFT does not recognise the EOF test for end of information on a file or dataset. FTN does recognise it.

26) CFT does not support the Double precision Function DCOSH. FTN supports it.

27) CFT does not support numeric name COMMON blocks FTN supports them

28) Compiler options for listing levels, etc. in FTN differs from that of CFT.

29) Facilities for Random access Input/Output exist on both systems but are at a different language level. Equivalent routines exist in non-Cray libraries.

APPENDIX 3

STORAGE MANAGEMENT FOR THE CDC-7600

1. Introduction

The access times to operands in store can vary significantly, depending on where they are stored and how they are accessed. In this chapter we investigate the factors that influence the access times and give some guidelines for optimal storage management.

The Central processing unit of the 7600 contains two levels of main storage: Large Core Memory (LCM) and Small Core Memory (SCM). It is also possible to have a Semiconductor Memory instead of the Small Core Memory. The Semi Conductor Memory is much faster than the SCM.

The size of SCM is 32K or 64K words assigned into 16 or 32 independent banks. This organisation allows 16 or 32 way interleaving.

The size of LCM is 250K or 500K words assigned into 4 or 8 independent banks. Every bank is 8 words wide.

The fact and figures given in the following paragraphs apply specifically to a CDC-7600 configuration like the machine at ULCC.

2. Factors affecting access times to LCM and SCM

The number of accesses to mainstore that can be processed simultaneously is limited by the number of A and associated X registers. There are only 5 registers available for loading elements from store and 2 registers for storing words into store. Hence no more than 5 load instructions can go on in parallel.

Instructions that reference SCM are 2 parcel instructions, 2 parcel instructions take two cycles to issue.

The SCM at ULCC consists of 64K words over 32 independent banks. Every bank is one word wide. If no bank conflict occurs it takes 8 cycles to load a word. Consecutive accesses to the same bank in SCM are possible every 10 clock periods. The change of bank conflicts occurring is rather small since there are 32 banks available. The number of bank conflicts in SCM can be reduced by accessing consecutive words in SCM, which are in different banks.

The LCM at ULCC consists of 256K words over 4 independent memory banks. Every bank is eight words wide. Every bank has an eight word register, called the superword (SWORD), associated with it. If a word referenced in LCM is already in one of the SWORD registers the access time is only 3 cycles. If not,the superword of the relevant bank has to be loaded with 8 consecutive words and the access time is 17 cycles. The delays caused by bank conflicts can be very long since the cycle time of a bank in LCM is 64 cycles. The change of bank conflicts occurring is very high since there are only 4 independent banks. Serial access in LCM can be very fast, since words that are already available in a SWORD can be accessed in 3 cycles.

A block of consecutive words can be moved, at a rate of 1 word per cycle, between LCM and SCM according to the CDC-7600 hardware manual. The LCM at U.L.C.C has only 4 banks of eight words. The cycle time of a bank is 64 cycles. Therefore in 64 cycles only 32 words can be transferred. To our opinion it takes 2 cycles for each word to be transferred during a block transfer.

3. Examples

In the following expression a row or a column in matrix V1 is multiplied by a constant (0.14). The result assigned to a row or a column in matrix V2

```
V1(I,J)=V2(I,J)*0.14
```

We run and timed this DO-loop for the following five cases:

1) V1 and V2 both in SCM and access both matrices by row
2) V1 and V2 both in SCM and access both matrices by column
3) V1 and V2 both in LCM and access both matrices by row
4) V1 and V2 both in LCM and access both matrices by column.
5) We place V1 in SCM and V2 in LCM and access by column. Data is transferred from LCM to SCM via block transfers processed and written to SCM. Matrices are stored in memory by column, as defined by Fortran. Since only consecutive words in store can be moved by a block transfer, access by row is not tested. The distance between two consecutive row elements is equivalent to the number of elements in a column, as defined in the DIMENSION statement for the matrix.

The following results were obtained:

	SCM	LCM	Block transfer
By row	0.63	3.52	X
By column	0.52	0.74,0.46,0.46,3.31	0.69

The access time to memory depends also on where the arrays are stored in memory. If the arrays start in the same bank or neighbouring banks bankflicts can occur. This is very important in LCM since there are only four banks. In the case of access by column to LCM we shifted the vector V1 over a distance of 8, 16, 24 and 32 words. the four timings we obtained are given in the table.The shifting in memory was done by calculating:

V1(I+L)=V2(I)*0.14 for L=8, 16, 24 and 32

The major conclusion from the table above is that access to LCM is up to 6 times faster than access to LCM and that block transfers can be very useful when accessing LCM.

4. Guideline for improved memory management

Step1: Find the part(s) of the program in which most time is spent during execution. We call this the nucleus of the program.

Step2: If LCM is not being referenced in the nucleus there is no need for restructuring the nucleus and go to Step6. Else go to Step3.

Step3: Try to move all arrays referenced in the nucleus to SCM. The arrays referenced only outside the nucleus can be moved to LCM if necessary. This is done by changing the Level 2 statements and the associated Common declarations in the program. This can be done very quickly. If all the arrays referenced in the nucleus do fit in SCM then go to Step6. Else go to Step4.

Step4: If the LCM arrays are referenced sequentially, or if matrices are referenced by column, use can be made of the CDC-Fortran instruction MOVLEV to move blocks of consecutive words from LCM to a working space in SCM and go to Step6. Else go to Step5

Step5: Try to restructure the nucleus to access arrays sequentially or to reference matrices by column and go to Step4. Else you have to rewrite your program or give up.

Step6: Ready!

These guidelines were used very succesfully on the subset of 20 programs taken from the benchmark test set

APPENDIX 4

RESTRUCTURING TO HELP AUTOMATIC VECTORISATION

4.1 Finding the nucleus of the program

When optimising, the first action to take is to find the part of the program in which most time is spent during execution.

The Cray-1 has a special facility called Flowtrace to find the most time intensive parts of the program. Flowtrace is enabled by using the compiler option ON=F on the CFT statement or by using a CDIR$ FLOW compiler directive in the Fortran source deck. There is a matching CDIR$ NOFLOW to disable flowtrace. The flowtrace routine prints a summary on dataset $OUT, listing the following information about each subroutine in a program:

The time spent in the subroutine
The percent of the total time spent in the subroutine
The number of times the subroutine was called
The average time per call spent in the subroutine
A list of the first 14 routines called by the subroutine
A list of the first 14 routines that call the subroutine

For more information on flowtrace see the CRAY-1 Fortran Reference Manual.

A procedure in which a large portion of the execution time is spent, may contain several DO-loops. It is in many cases not possible to determine from the Fortran code which DO-loops the most time is spent in. Therefore counters must be used to count the number of loop iterations in each loop. These counters must be placed in a Common block, because otherwhise they would be lost on procedure exit. It would be nice if Cray-research provided a compiler directive to do this often occurring job.

At the end of every program part on the listing is printed which 'blocks' contain vectorisable DO-loops. Often it is useful to list all 'blocks' that have been generated by the compiler. This is done using the compiler option ON=B.

If the inner DO-loop does vectorise, one should check the vector length. If the vector length is shorter than64, one should try to extend the vector length. Hence a pipeline achieves its maximum performance for vectors of length 64, the size of a vector register.

If the inner DO-loop does not vectorise we have to restructure the loop to make it vectorise, if possible. In the next paragraph we discuss some techniques to restructure the DO-loop.

4.2 Restructuring

In this paragraph we discuss some technique for restructuring DO-loops. For more examples see Brode79 and Higbee79.

1) Switching inner and outer DO-loop to avoid recursion, or to extend the vector length.

```
      DO 100 I=1,N
         DO 100 J=2,N
100         A(I,J)=A(I,J-1)*B(I,J)
```

replace this by:

```
      DO 100 J=2,N
         DO 100 I=1,N
100         A(I,J)=A(I,J-1)*B(I,J)
```

2) Splitting a loop in a scalar loop plus a vector loop

```
      DO 100 I=2,N
         A(I)=B(I)*C(I)
100      D(I)=D(I-1)*A(I)+C(I)
```

replace this by:

```
      DO 100 I=2,N
100      A(I)=B(I)*C(I)          A vector loop

      DO 200 I=2,N
200      D(I)=D(I-1)*A(I)+C(I)   A scalar loop
```

3) Non subscript use of loop counter

```
      DO 100 I=1,N
100         A(I)=SIN(B(I)*I)
```

replace this by:

```
      DATA C/1.0,2.0,3.0,.....N/
      DO 100 I=1,N
100         A(I)=SIN(B(I)*C(I))
```

4) Irregular addressing

```
      DO 100 I=1,N
100         D(E(I))=A(I)*SIN(B(I))
```

replace this by:

```
      DO 100 I=1,N
100         TEMP(I)=A(I)*SIN(B(I))      vector loop

      DO 200 I=1,N
200         D(E(I))=TEMP(I)             scalar loop
```

5) Make subroutines inline

```
      DO 100 I=1,N
100         CALL SUBRT(I)

      SUBROUTINE SUBRT(I)
      COMMON X
      X(I)=Y(I)*B(I)
      RETURN
      END
```

replace this by:

```
      DO 100 I=1,N
100   X(I)=Y(I)*B(I)
```

3 Inline CFT fuctions to remove IF statements

DO-loops containing IF statements will not vectorise. The CFT compiler contains a set of built inline functions to help to remove IF statements, to make DO-loops vectorise.

The routines are: CVMG, MAX and MIN. These routines are ofen slower than the code normally generated for them by the compiler. This is because both branches of the IF statement are evaluated for all cases, the two result vectors are merged to form the final result vector. These routines are normally used to allow the other statements in a DO-loop to be vectorised.

The available MAX and MIN functions are listed in the Cray Fortran Manual in Appendix B-4.

For example the AMAX0 routine returns the maximum value of a list of real variables passed to it as parameters. So AMAX(R1,R2,R3) returns the maximum of R1 and R2 and R3. This can be used as follows:

```
      DO 100 I=1,N
            IF(B(I)).GT.0.0)GO TO 200
            A(I)=0.0
            GO TO 100
200         A(I)=A(I)*B(I)100          CONTINUE
```

replace this by:

```
      DO 100 I=1,N
100         A(I)=A(I)*AMAX0(B(I),0)
```

The routine CVMGZ(X,Y,Z) does the following: Expression X is evaluted and returned by the function CVMGZ if expression Z is zero. Expression Y is evaluated and returned by the function CVMGZ if the expression Z is nonzero.

```
      Y(I)=1.0
      IF(X(I).EQ.0) GO TO 100
      Y(I)=1.0/X(I)
100   CONTINUE
```

replace this by:

```
      Y(I)=1.0/CVMGZ(1.0,X(I),X(I))
```

4. Scilib routines

The Scilib routines are optimally handcoded and vectorised procedures. These routines are extremely useful in optimising programs. Especially, basic vector operations like matrix multiply and dot-product of two vectors are extremely useful. These routines should be used whenever possible. Unfortunately, some of the the routines, like summation of the elements of a vector, are not much faster than their scalar equivalents.

In our benchmark we found that a large portion of the matrix operations involve triangular shaped (symmetric) matrices. The Scilib routines can only handle square matrices. The overheads of converting from square to triangle and back were often so high that little was gained. Maybe a separate set of routines can be provided for triangular shaped matrices.

For a full documentation on Scilib see the Library Reference Manual. The following list of important Scilib routines shows only a part of the Scilib library:

Name	Function
SDOT	Dotproduct of two real arrays
CDOTC,CDOTU	Dot product of two complex arrays.
CCOPY	Copies a complex array into another complex array.
SCOPY	Copies a real array into another array
MXMA	Matrix multiply
MINV	Returns the solution of a general equation set, matrix inversion optional.
SSUM	Sums the elements of a real array.
CSUM	Sums the elements of a complex array.
CFFT,RCFFT,CRFFT	Are used for Fast Fourir Transformations

APPENDIX 5

THE RESTRUCTURURED PROGRAMS

1. Introduction

At the end of this thesis are two microfiches. These microfiches contain the most time intensive parts of the programs in the benchmark subset that was restructured. The first fiche, dated 18-12-80, contains the unmodified 'programs'. The only modifation to these programs was the conversion of double precision arithmetic to single precision arithmetic. Double precision arithmetic prevents vectorisation. The second fiche, dated 19-12-80 contains the restructured programs. Some programs have not been included in this fiche because no restructuring was possible. These programs are: U9, U16, U30, U31.Programs U2 and U3 which could not be restructured were included in this fiche.

In paragraph two of this appendix is a very brief description of the programs in the subset that was restructured.

Paragraph three contains an index to the programs on the two micro fiches.

2. Brief description of the programs in the subset

U2,U3 After the removal of double precision arithmetic in U2, this program was identical to U3. The execution time in the two programs is distributed over several DO-loops in which vectorisation is prohibited by many IF statements. The IF statements can be removed to make the program vectorise, but this would have meant to rewrite the entire program.

U4 This program cannot be vectorised. Most time was spent in routines FSHAPE and FMATRIX. In FMATRIX small matrices, of triangular shape, are manipulated. In both procedures were the vectors too short.

U6 In this program we removed recursion by splitting a DO-loop in a vector loop and a scalar loop. See fiche 2 at lines 791 and 840.

U8 This program was written very inefficiently. After making several routines at different levels inline, we managed to vectorise the program. Extensive use was made of the SCILIB routine SDOT.

U9 This program spends 42% of its time in a routine called LISOLV. Vectorisation is not possible because of recursion and irregular addressing. A small amount of vectorisation can be obtained by splitting the DO-loop.

U10 This program is easy to vectorise. It contains basic vector operations like matrix multiply which can be replaced by Scilib routines.

U13 This program can be vectorised quite easily, 98 percent of execution time being spend in one procedure. The program vectorised after removal of an IF statement. This program also performs some integer artithmetic.

U14 This program performs most of its time complex arithmetic. Most of the time is spent in a little routine called RESPOT. This routine is used to find the maximum in a grid of complex numbers. Since RESPOT was called tens of millions of times we made it inline with the main program, this gave no improvement. Further restructuring work on the main program DO-loops in which most time was spent now, gave no improvements to our surprise. Further improvements on this programs should be possible.

U15 This program spends 95 percent of its time in a routine called MAT. It did vectorise after removal of recursion.

U16 Fourty one percent of the execution time of this program is spent in one routine. There are 5 DO-loops in this routine which are accessed equally often. These DO-loops are difficult or impossible to vectorise.

U30 Fourty nine percent of the execution time of this program is spent in a routine called SKEX5. This routine could not be vectorised.

U31 The program did already vectorise quite well (automatic vectorisation). Fourty one percent of its time is spent in one routine called FORIT. This procedure performs a sort of Fast Fourir Transformation. Scilib does provide a vectorised version of FFT, but we were not sure how to use it in this particular case.

U32 Sixty two percent of the Execution time of this program is spent in a routine called SELECT. The IF statements and irregular addressing prevent vectorisation.

U51 Seventy eight percent of the time is spent in SOLV. Some DO-loops are cases of imaginary recursion. They can be forced to vectorise by the compiler directive CDIR$ IVDEP.

U52 Fourty three percent of the time is spent in a routine called MASUB. By loopsplitting and removing imaginary recursion we can obtain some vectorisation.

U57 More than 90 percent of the time is spent in a routine called SINNER. We can make use of the Scilib routine SDOT to speed up the program.

U58 Thirty five percent of the time is spent in FRATYS. The DO-loop in this procedure is a case of 'non subscript use of a loop counter'. This was solved by loop-splitting. Thirty nine percent of the time is spent in C06AAX in which some sort of Fourir transformation takes place. It was again not clear how to apply the Scilib FFT routine.

XXX This program was easy to optimise. First some IF statements had to be removed using CVMGT and then CDIR$ IVDEP was used to remove 'imaginary recursion'.

TIM This program spends all of its time in an unusually large DO-loop. This DO-loop contains many IF statements. Some vectorisable code at the beginning of the loop can be split off. An interesting aspect of the program are the 'common subexpressions'. The FTN compiler seems to pick up most of them while the CFT compiler does not. Restructuring work by G. Ashley from Cray Research resulted in the execution time being halved. The number of modifications was high. Further restructuring work is still going on by the Author of the program.

3. Index to the programs on the micro fiches

The programs on each fiche are linenumbered continuously. Fiche 1, the unmodified programs, is numbered from 00000011 to 00005586. Fiche 2, the restructured programs, is numbered from 00000000 to 00003672.

At the end of every 'program' there is a Flowtrace. E.g. the flowtrace for program U2 is found on line 190 of fiche 1. The ROUTINE column indicates the name of the routine. TIME indicates the number of seconds spend in the routine. The (:) column specifies the percentage of the total execution time of the program that is spent in that procedure. The CALLED column specifies the number of seconds spent per call in the procedure. TOTAL specifies the execution time of the program without the Flowtrace overheads.

Modified lines are marked on the right hand side with GNAGNA.

The following table is an index to the microfiches. The numbers indicate the line numbers at which the programs are found. So program U8 starts at linenumber 00000973.

	Fiche-1	Fiche-2
U2	11	2
U3	209	200
U4	411	403
U6	685	678
U8	973	993
U9	1562	-
U10	1916	3475
U13	2078	1692
U14	2323	3578
U15	3017	1939
U16	3071	-
U30	3380	-
U31	3665	-
U32	3783	1997
U51	4133	2360
U52	4448	2617
U57	4650	2830
U58	4699	2876
XXX	5282	3053
TIM	4877	-

TABLE 1

The results of the Live Benchmark

Id	CDC-7600 Job Time (secs)	CRAY-1 Job Time (secs)	7600 Ratio CRAY
U1	29.32	12.76	2.29
U2	63.23	162.16	0.39
U3	26.34	6.26	4.21
U4	3.09	1.09	2.83
U5	56.71	142.75	0.40
U6	39.57	7.66	5.17
U8	473.07	268.64	1.76
U9	169.48	17.70	9.57
U10	513.25	388.10	1.32
U11	246.93	116.33	2.12
U12	570.17	226.01	2.52
U13	191.36	115.96	1.65
U14	246.49	112.11	2.20
U15	258.73	57.22	4.52
U16	10.21	2.85	3.58
U17	905.47	1229.25	0.74
U18	137.83	338.14	0.41
U19	0.29	0.05	5.80
U20	17.58	41.01	0.43
U21	2.34	1.41	1.66
U22	1.21	0.32	3.78
U23	4.53	3.01	1.50
U24	1.86	0.47	3.96
U25	0.72	0.24	3.00
U26	0.56	0.09	6.22
U27	5.27	1.11	4.75
U28	0.39	0.06	6.50
U29	1.31	0.42	3.12
U30	3.77	1.66	2.27
U31	383.58	160.70	2.39
U32	8.30	3.86	2.15
U33	1.31	0.60	2.10
U34	15.86	5.65	2.81
U35	2.71	11.01	2.68
U36	28.83	15.42	1.87
U37	1.13	0.31	3.64
U38	1.96	0.47	4.17
U39	1.61	0.47	3.42
U40	86.55	26.14	3.31
U41	5.23	0.72	7.26
U42	0.53	0.11	4.82

TABLE 1 continued

Id	CDC-7600 Job Time (secs)	CRAY-1 Job Time (secs)	7600 Ratio CRAY
U44	2.15	0.44	4.89
U45	0.57	0.11	5.18
U46	1.89	0.35	5.40
U49	1.45	0.27	5.37
U50	21.56	5.57	3.87
U51	71.22	12.13	5.87
U52	49.45	20.90	2.37
U54	71.28	34.73	2.05
U57	49.38	38.47	1.28
U58	32.07	14.45	2.22
U59	1.86	0.45	4.13
U60	1.99	0.83	2.40
U61	6.48	1.18	5.49
U62	97.76	33.19	2.94
U63	15.49	12.40	1.25
U64	280.10	122.78	2.28
XXX	355.26	154.09	2.31
U66	30.94	13.64	2.27
U67	46.53	40.40	1.15
TIM	979.81	802.84	1.22
U70	1.41	0.46	3.06
U71	1.65	0.31	5.32
U72	32.28	11.59	2.79
U73	20.73	6.85	3.03
U74	15.16	9.95	1.52
U75	20.47	8.86	2.31
U76	80.49	20.52	3.92

TABLE 2

Analysis of indiviual benchmark programs on the CDC-7600

		Original Run		MNF FTN(2)	
Id	Comment	Compil	Exec	Compil	Exec
U2	MNF,dp	0.134	62.969	0.739	57.181
U3	MNF	0.128	26.063	0.495	9.114
U4	FTN(2),MAN,LEV-2	1.354	1.511		
U6	MNF	0.282	39.126	0.768	13.621
U8	FTN(2),LEV2	2.397	470.516		
U9	MNF,LEV2	1.687	167.418	3.816	98.668
U10	FTN(2),dp,MAN,LEV2	6.573	506.503		
U13	FTN(2),INT	0.704	191.039		
U14	FTN(1),MAN,LEV2,CMPLX	3.594	242.860		
U15	FTN(2),LEV2	1.185	256.801		
U16	MNF,RMS/WMS	2.809	6.982	5.786	5.582
U30	FTN(2),dp	2.946	0.620		
U31	FTN(2),LEV2	12.125	370.978		
U32	FTN(2)	3.583	4.496		
U51	MNF,LEV2	1.393	69.512	4.510	46.099
U52	FTN(1)	5.908	43.217		
U57	FTN(2)	0.828	48.352		
U58	FTN(2)	2.334	29.535		
XXX	FTN(2),LEV2	3.724	351.351		
TIM	FTN(2),LEV2	3.053	976.851		

TABLE 2 continued

	FTN(1,0) FTN(2)		Better Mem Mangmnt		Dp Sp	
Id	Compil	Exec	Compil	Exec	Compil	Exec
U2					0.473	17.760
U3						
U4	1.622	1.193	1.502	1.186		
U6						
U8			2.340	439.0		
U9			3.700	53.915		
U10	7.625	295.766	7.567	264.981	7.401	241.045
U13						
U14	4.011	261.329	3.780	214.170		
U15			1.1821	103.792		
U16						
U30					2.792	0.581
U31			12.125	370.978		
U32						
U51			4.457	25.6		
U52	6.573	34.017				
U57						
U58						
XXX			3.724	351.351		
TIM			2.945	951.976		

TABLE 3

Analysis of individual benchmark programs on the CRAY-1

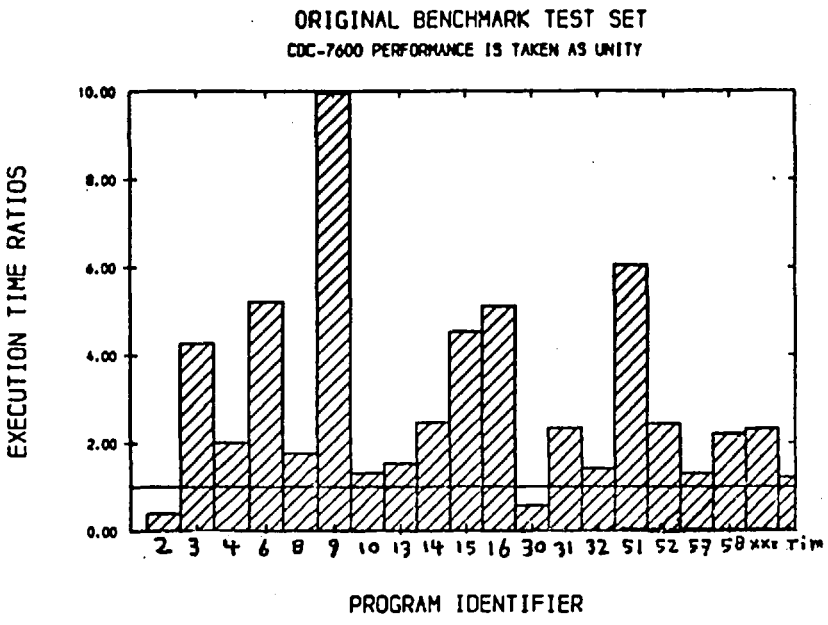
		Original Run		No-Vectoris	SP DP	Restructured
Id	Comment	Compil	Exec	Exec	Exec	Exec
U2	dp	0.103	162.0	164.0	7.30	7.3
U3		0.09	6.1	6.6		6.1
U4		0.300	0.76	0.757		0.76
U6		O.158	7.5	7.67		2.99
U8		0.422	267.6	268.4		79.
U9		0.825	16.8	18.0		16.8
U10	dp	1.539	387.0	408.7	269.6	188.
U13	INT	0.141	125.0	132.7		56.
U14		0.673	98.0	100.0		98.
U15		0.348	56.6	60.0		9.6
U16	WMS,RMS	1.405	1.37	1.6		1.37
U30	dp	0.500	1.08	1.08	0.547	0.55
U31		1.951	159.0	298.3		159.
U32	INT	0.643	3.20	3.67		2.98
U51		0.843	11.5	21.0		5.8
U52		1.661	17.7	20.7		17.7
U57		0.126	37.0	38.0		11.1
U58		0.365	13.4	14.71		9.5
XXX		0.7259	154.0	153.8		49.
TIM		0.497	803.0	820.0		454.

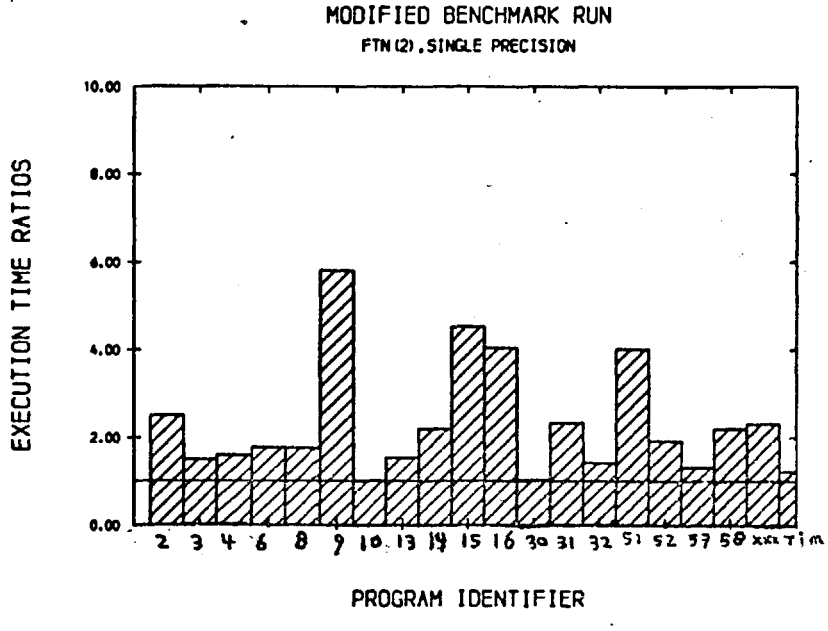
TABLE 4

Comparison of FTN(1) and FTN(2)

	OPT=2		OPT=1		RATIO(OPT1/OPT2)	
Id	Compil	Exec	Compil	Exec	Compil	Exec
U2	0.739	57.18	0.571	60.334	0.77	1.06
U3	0.495	9.11	0.373	9.688	0.75	1.07
U4	1.622	1.19	1.340	1.517	0.82	1.25
U6	0.768	13.62	0.611	15.571	0.80	1.15
U8	2.397	470.516	1.986	529.656	0.83	1.13
U9	3.816	98.668	3.588	101.045	0.94	1.03
U10	7.625	295.77	6.573	506.503	0.86	1.7
U13	0.704	191.04	0.533	311.618	0.76	1.63
U14	4.011	261.33	3.594	242.860	0.90	0.93
U15	1.185	256.80	1.443	256.667	0.8	1.00
U16	5.790	5.582	4.864	5.934	0.84	1.05
U30	2.946	0.620	2.303	0.628	0.77	1.11
U31	12.125	370.970	-.-	478.883	-.-	1.29
U32	3.583	4.496	2.847	4.720	0.80	1.04
U51	4.510	46.890	3.885	51.109	0.87	1.11
U52	6.573	34.017	5.908	43.217	0.89	1.27
U57	0.828	48.352	0.687	117.159	0.83	2.44
U58	2.334	29.535	1.938	39.808	0.83	1.34
XXX	3.724	351.351	3.185	402.032	0.86	1.15
TIM	3.053	971.851	2.225	1612.576	0.74	1.65

TABLE 5


ORIGINAL BENCHMARK TEST SET
CDC-7600 PERFORMANCE IS TAKEN AS UNITY
EXECUTION TIME RATIOS
10.00
8.00
6.00
4.00
2.00
0.00
2 3 4 6 8 9 10 13 14 15 16 30 31 32 51 52 57 58 xxx Tim
PROGRAM IDENTIFIER


MODIFIED BENCHMARK RUN
FTN(2), SINGLE PRECISION
EXECUTION TIME RATIOS
10.00
8.00
6.00
4.00
2.00
0.00
2 3 4 6 8 9 10 13 14 15 16 30 31 32 51 52 57 58 xxx Tim
PROGRAM IDENTIFIER

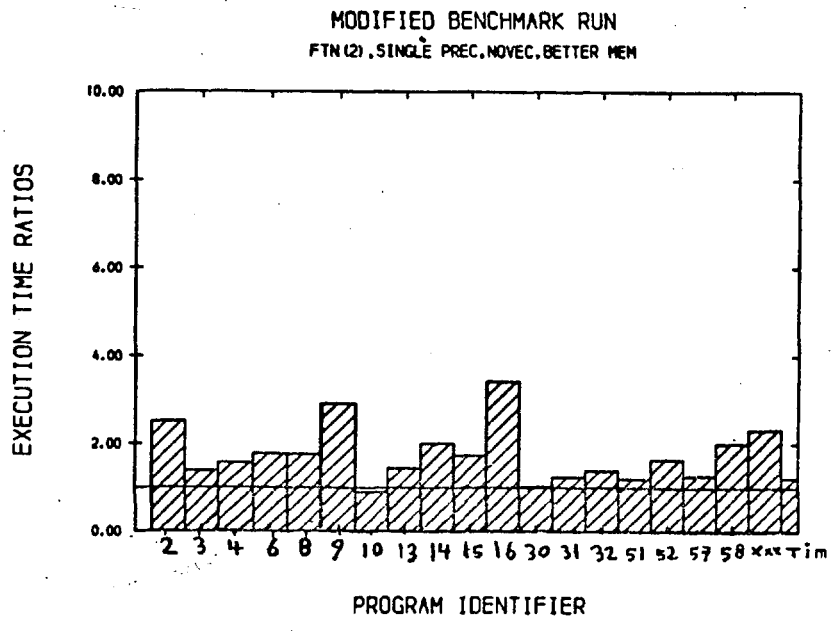
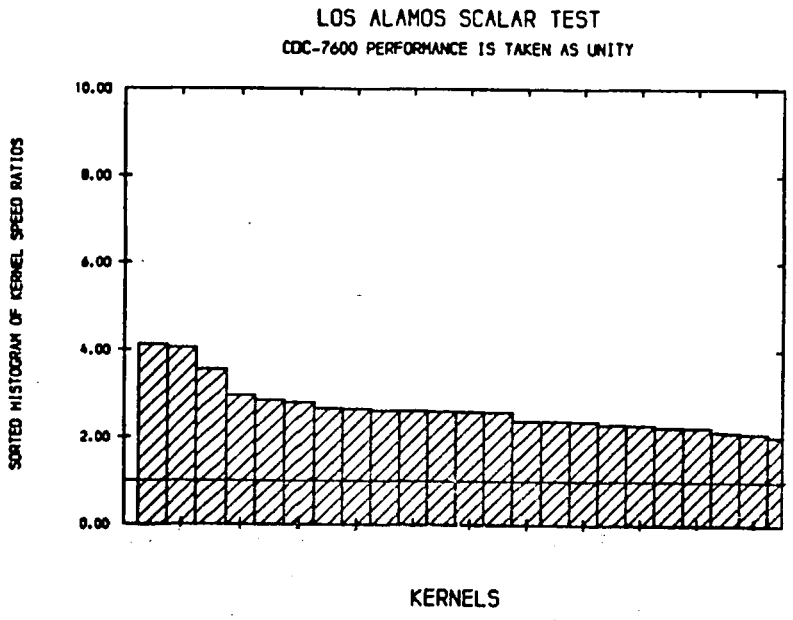

MODIFIED BENCHMARK RUN
FTN(2), SINGLE PREC, NOVEC, BETTER MEM
EXECUTION TIME RATIOS
10.00
8.00
6.00
4.00
2.00
0.00
2 3 4 6 8 9 10 13 14 15 16 30 31 32 51 52 57 58 xxx Tim
PROGRAM IDENTIFIER

TABLE 6


LOS ALAMOS SCALAR TEST
CDC-7600 PERFORMANCE IS TAKEN AS UNITY
SORTED HISTOGRAM OF KERNEL SPEED RATIOS
10.00
8.00
6.00
4.00
2.00
0.00
KERNELS

CRAY-1 BIBLIOGRAPHY

Auerbach
Cray-1 report
Auerbach Computer Technology Report

Baskett, Keller
An evaluation of the Cray-1
In High Speed Computer and Algorithm organisation 71-84
Academic Press London 1977

Brode
How to get more out of your vector processor
LASL workshop on vector and parallel processors
September 1978

Brode
Vectoriser system user's guide
Massachusetts computer associates 1979

Buzby
Vectorisation for the Cray-1 of some methods for solving elliptic difference equations.
In High Speed Computer and Algorithm organisation

Calahan
Performance of linear algebra codes on the Cray-1
Fifth SPE Symposium on reservoir simulation, Denver, Colorado
Jan 1979

Cray Research, Update Reference Manual
publication 2240013
Minnesota,1980

Cray Research, Library Reference Manual (Scilib)
publication 2240014
Minnesota,1980

Dorr
The Cray-1 at Los Alamos
Datamation october 1978

Ewald
A high performance graphics system for the Cray-1
Fifth annual conference on computer graphics and interactive techniques
SIGGRAPH 78, october 1978

Gary
Analysis of applications programs ans software requirements for high speed computers.
In High Speed Computer algorithm organisation 329-354
Academic Press London 1972

Gardner
Cray Crystallography and polytypism of naturally occurring tridymite
American Crystallographic Association Abstracts, 24 August 1974

Higbie
Vectorisation and conversion of Fortran programs for the Cray-1 (CFT) compiler
Cray research publication 2240207, 1979

Higbie
Application of vector processing
Computer design, April 1978

Hockney
The large parallel computer and university research
Contemp. Phys. (GB) Vol20,nr2 149-85, March-April 1979

Johnson
An Introduction to vector processing
Computer design February 1978

Jordan
Some linear algebraic algorithms and their performance on the Cray-1
In High Speed Computer Algorithm organisation
Academic Press 1977

Jordan
Vector reduction operations on the Cray-1 and their performance
International conference on parallel processing
Publication IEEE, August 1977 Detroit

Keller
Cray-1 Evaluation final report
Los Alamos Scientific Laboratory
Los Alamos,NM,LA-6456-MS 1976

Russel
The Cray-1 computer system
Communications ACM Vol 21 nr 1, 1978

Sites
Programming tools' statement counts and procedure timings
Sigplan Not. (USA) Vil 13, nr 12, 98-101, December 1978

Sites
An Analysis of the Cray-1 Computer
Fifth annual symposium on computer architecture
April 1978

Swartztrauber
A parallel algorithm for solving general tridiagonal equations
Math. Comp. (USA) Vol 33 nr 145 185-99 Jan 1979

REFERENCES

Amorum
An evaluation of vectorisation techniques
M.S.C thesis, dept. of Computing and Control, Imperial College 1980

Baker
CDC-6000 Series register save/restore
Software Pratice and Experience, Vol 2, 377-387 1972

Graham
The parallel and the pipeline computers
Datamation April 1970

Hendrickson
When you wish upon a Star
University of California, Lawrence Livermore Laboratory, 1976

Mc Intyre
An introduction to the Illiac IV computer
Datamation April 1970

Parkinson
An Introduction to Array Processors
Systems International, November 1977

Ramamoorthy
Pipeline architecture
Computing Surveys Vol 9 nr1, March 1977

Thornton
Parallel operation in the Control Data 6600
AFIPS Proc part 2 Vol 26 pp 33-40 1964

Timmreck
Computer selection methodology
Computing surveys, Vol 5 Nr 4 December 1973