McGill University
School of Computer Science

DLX Simulator Directed Profiling

ACAPS Technical Memo 48
July 7, 1992

Sumithra Jagannath

**Advanced Computer Architecture and
Program Structures Group**

3480 University St • Montreal • Canada • H3A 2A7

Contents

Abstract

Performance analysis tools are assuming more and more importance with the evolution of new architectures, and new compiler technologies for these architectures. Improving the run-time behaviour of programs and making them more efficient is one of the main goals of computer science research today, and performance analysis tools aid in pursuing this goal. Hence they are at the centre of ongoing research activities in computer science. *Trace driven analysis tools* and *profilers* are two kinds of performance analysis tools that have been developed to aid in evaluating programs and the architectures of machines.

This paper describes a performance analysis tool **DLXProf**, that uses both the principles of trace-driven simulation and profiling. Some important features of this tool are discussed. The difference of **DLXProf** from other profiling tools is that this tool profiles memory references that come from pointer or array types, and stack, heap and global memory references. It was found from literature study conducted about the related work done in this area that **DLXProf** pioneers this kind of profiling. Other profiling tools like **GPROF** and **TPROF**, and some trace driven analysis tools like **TRAPEDS** are studied in relation with **DLXProf**. In addition, the design and implementation of the profiler on the **DLX** Simulator are described in detail.

The profiler was tested on the assembly codes of some application programs and interesting results were obtained. For instance, it was found that storing arrays in caches in programs with array intensive computations was very useful in reducing cache misses. Recursive programs were studied and pictures of the stack were obtained. They showed the intensive usage of the stack space. Applications that used the heap were tested to obtain pictures of the heap space. In conclusion, some improvements and future directions to the profiler are suggested.

1 Profilers and Profiling – an introduction

Performance analysis tools are assuming more and more importance with the evolution of new architectures and new compiler technologies for these architectures. Performance debugging on uniprocessors and multiprocessors is an important means of analyzing high-level performance problems of programs run on these machines. Improving the run-time behavior of programs and making them more efficient is one of the main goals of computer science research today, and performance analysis tools aid in pursuing this goal. Hence they are at the center of ongoing research activities in computer science. *Trace-driven analysis tools* and *profilers* are two kinds of performance analysis tools that have been developed to aid in evaluating programs and the architectures of machines, eg., trace-driven simulation is very useful in studying cache behavior. Caches are considerably influencing the performance of programs and will continue to do so in future. Hence a careful study of cache design choice is crucial to the design of future computer systems [21]. Trace-driven simulation is not an easy task. It requires a large amount of disk space to store the program traces and simulations that would produce results.

Performance analysis using profilers is another technique that is very important in debugging applications and also in identifying bottlenecks that affect performance in applications. A *profiler* is a tool that is used for performance debugging. Profilers are usually implemented as observer routines that monitor the executing program for different characteristics and behaviors. A profiler can also be a tool that assesses the cost of routines at various levels of a program. One drawback of profilers is that they interfere with the target applications, thereby influencing the results obtained.

Hence these performance analysis tools play an important role in the research of program behavior. There are some differences between a profiler and a trace-based performance analysis tool. The first is that a profiler intrudes and influences the target program's behavior by its presence while collecting information from it, while a trace-based performance analysis tool does not intrude. The second is that when a program that is profiled has finished executing, the data about the run-time behavior of the program has already been collected by the profiler, and is readily available. But in the case of a trace-based analyzer, the program trace is produced during execution, and this large trace cannot be useful right away. It needs to be post processed in order to produce the required information.

The traditional uses of profiling have been to measure the execution counts or execution times of routines. These profilers usually measure the run-time characteristics of programs and are of one of the following kinds :

- Profiling to count the number of times a particular routine is called during the execution of the program, known as **execution count profiling**. This execution count profiling is used to determine if an algorithm is performing as expected. Improvements to the algorithm may be suggested by careful examination of the profiling results. Subtle errors in an algorithm can also be discovered with the execution count profile results. Hence at this level, the profiler code can be thought of as debugging statements which show the number of times a piece of code is executed. This kind of profiling code is usually added only to routines in the program that are called often enough so that the extra code that a profiler adds does not significantly alter the execution behavior of the program. Another use of this profiling is to test if a particular routine is executed at all, for exhaustive testing.

- The second kind of profiling is to calculate the execution time of routines ie., profiling to display the timing information about statements or routines. Execution time profiling tells us the amount of time spent in each routine. This will tell us which of the routines take up maximum time and help us optimize them. For instance, a *flat profile* has been built with the **GPROF**, a **UNIX** profiler, which will give us an idea of the routines that are responsible for large fractions of the execution time. It shows us if a single function is overwhelmingly responsible for the total execution time of the program.

- A profiler may also be a tool that gathers information about the arcs of the dynamic call graph traversed by the execution of the program. Post processing of this data will enable us to build a dynamic call graph for the execution of the program and to propagate execution times along the edges of this graph to be able to attribute the times for routines to the routines that invoke them.

Due to the intrusion of traditional profilers with the behavior of the programs being profiled, there are limits placed on the size of the profiler code that can be added to a program.

This report describes a performance analysis tool **DLXProf**, that uses both the principles of trace-driven simulation and profiling. One important feature of this tool is that it overcomes the drawback of other profiling tools by being totally non-intrusive with the target applications that are being profiled, and the drawback of other trace-based performance analysis tools by not generating enormous quantities of traces and simulations of the results. The difference of **DLXProf** from other profiling tools is that it profiles memory references. It obtains data about memory references that come from pointer type variables, array types, local and global variables at the compilation level and obtains the run time results. It also profiles stack, heap and global memory references during execution time. A study conducted in the area of profiling did not reveal an existing profiler with the same ideas as **DLXProf**. It uses the techniques of trace-based analysis by annotating the assembly code with some useful information, and it uses this information to do the profiling. Each memory reference that occurs during the execution of the program is analyzed along with the information from the annotations to classify the references.

There exist other trace-based analysis tools that use some of the techniques used with **DLXProf**, with respect to gathering information at the compilation level using the **GNU C** compiler and annotating the assembly code with the information, but these tools do not use this information for profiling. Further discussion about this is provided in the related work section.

The rest of this report is organized as follows : Section 2 discusses the main features of **DLXProf**, and brings out the advantages it has over other conventional profilers that have been built so far. Section 3 discusses some other trace-driven analysis and profiling tools that exist and other related work done in this area. Section 4 discusses the implementation details of **DLXProf** both at the **GNU C** Compiler level and at the **DLX** simulator level. Section 5 deals with the test programs that were run through the profiler and explains the graphs and results obtained. Section 6 gives some concluding comments on the tool and summarizes the report on the whole. The final section gives suggestions for future improvements and modifications to **DLXProf**.

2 Compiler and Simulator Directed Profiler "DLXProf"

DLXProf is a profiler that has been built on the **DLX** simulator to give a profile of memory references of C programs executing on the simulator. The profiling done by **DLXProf** is a little different from the conventional profiling. Instead of measuring execution counts of routines, or their execution times, the idea is to use some of the information available at compile time that is usually lost at run-time in order to collect some higher level information that is not otherwise possible at run-time. Hence, a part of this project has been an attempt to pass down extra information about the high level source code through the compiler to the execution level and study the run-time behavior. Such information could be for instance, looking at the value in a register at a particular time to be able to say

- if it is holding a local or a global variable to that function,
- if it has some pointer information, or an array element,

so that when a memory reference to this location occurs, it can be categorized as a pointer reference to memory or an array reference to memory, etc. The compiler has a better perspective on a program than at the assembly level and so this extra information has to be gathered during compilation. It is possible to gather some information during execution, which is also useful for performance analysis, and this has also been done.

Extra information could prove useful in understanding how typical benchmarks perform with respect to memory references for eg., how often they access global memory, or the heap space. This could give us an idea of how this information relates to the behavior of these programs in caches. We could study cache behavior for different types of data structures, which could give an idea of which ones lead to more misses. More specifically, We could relate cache misses with these references to memory and see what kind of accesses (local, global, or heap) leads to more misses in the cache. By finding out how often a memory location was accessed, bottlenecks can be identified and optimized. Hence **DLXProf** is a performance analysis tool. Although it might not be possible to generalize the findings and say that for most programs, the maximum number of misses come from for eg., heap accesses, we could try to understand the particular programs under study.

DLXProf has a special feature that it is interactive with the program being simulated, but non-intrusive at the same time, unlike other conventional profilers. This has been achieved by using the **DLX** simulator, a **RISC** architecture simulator developed at Berkeley [13]. The profiler code has been embedded in the code for the simulator, and so the profile data can be obtained by the simulator during execution, without interfering with the natural execution of the program. The phases of compilation and execution are modified with the profiling added to look as in Fig. 1. The C program to be compiled and executed is given to the compiler. The **GNU C** compiler has been chosen for this. Attempts have been made to collect information at the **AST** and **RTL** levels of the compiler and pass it down to the assembler code, to be used by the **DLX** simulator during execution. This information includes identifying the operands of an instruction as containing local or global variable information or pointer or array values, along with the names of the variables. This is passed down to the assembly code in the form of special comments added to the code. This forms the *augmented assembler code* identified in Fig.1.

Assembly code is normally given to the **DLX** simulator to be executed. But now the simulator is modified to read this *augmented assembler code*. This is the *Modified DLX simulator* identified in Fig.1. The modified simulator also has profiling code added to it to collect the run-time information and to do the performance analysis. This forms the *performance measurements phase* shown in the same figure. The results are then analyzed and graphs are plotted.

3 Related work

It would be interesting to see if other such trace-based performance analysis tools and profiling tools have been developed and how they compare or relate to **DLXProf**. Hence in this section, we give an overview, first of trace-driven simulation and then of profiling.

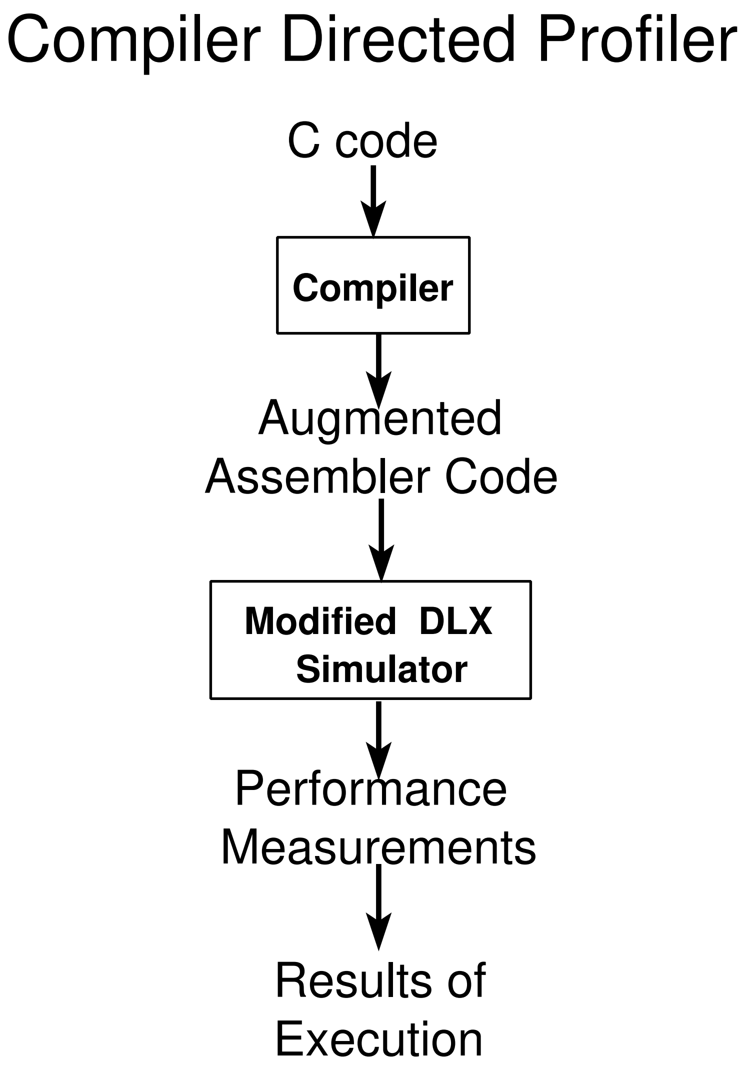

Figure 1: Conceptual model of "DLXProf"

3.1 Trace-Driven Simulation

Trace-driven simulation is an important aid in performance analysis of computer systems. Many areas of computer performance analyses require detailed traces of events that occur during a program's execution[15]. The main aim of this project is performance analysis, as is the aim of trace-driven simulation. Hence there is some similarity between some of the techniques used for the latter and those used in this project. The techniques discussed here are useful in address tracing to help in analyzing cache performance and other design alternatives.

There are several tracing techniques that exist and can be enumerated as follows :

1. **Hardware Monitoring Based Traces** can directly record memory bus activity and the actual addresses can be sent to off-chip caches or main memory modules [7, 3]. This technique can capture both user and operating system references. The drawbacks of this technique are its complexity, cost, and lack of flexibility.

2. **Instruction Interrupt Based Traces** interrupt the execution of the program after each instruction. Then the virtual address references for each type of instruction is calculated. Operating system routines disable these interrupts. So operating system execution cannot be traced with this method. Also, the frequent interruption slows down the program execution considerably.

3. **Software simulation Based Traces** can be used to provide user traces and model the execution time of processors. It can also emulate the operating system activities [19, 11]. This is the technique that has been used for **DLXProf**, using the **DLX** simulator as the software simulator to simulate the execution of the programs and to obtain the dynamic profile results.

4. **Microprogramming Based Traces** alter the machine microcode to capture address traces. The main drawback of this approach is that commercial multicomputers contain their microcode in ROM and are not easy to alter. **ATUM** [1] is a technique that belongs to this category.

5. **TRAPEDS** is a technique that produces accurate and efficient multicomputer traces [5]. This technique performs *execution based simulation* whereby it gathers accurate program statistics for a program, as it is executing[16]. The aim of the **TRAPEDS** technique is to produce tracing data, from where performance analysis is done, whereas in **DLXProf**, profile data gathered during execution is used to do performance analysis. But the technique used for tracing in the **TRAPEDS** and that for the profiling in **DLXProf** are quite similar. In both the techniques, dynamic results are collected, and this collection is done using some extra information from a modified assembly code. In the **TRAPEDS** technique, information about basic blocks, memory accesses, the type of access and addressing modes are recorded in the assembly file. The number of instruction fetches required to load the instructions executed since the previous memory reference is also recorded in the file. But, with **DLXProf**, the assembly file carries information about registers saying what kind of value the register carries ie., whether it has pointer or array information, or local or global variable information. In the **TRAPEDS** technique, this modified assembly file is reassembled and

linked with other files to create a modified executable file, whereas in this project, the modified assembly file is used as such and read in by the simulator. The simulator **DLXSim** has been modified to use this information for profiling. The modified executable file of **TRAPEDS** is used to produce address trace information. This technique in **TRAPEDS** interferes a little with the program being traced, and so special care has been taken to reduce this interference, whereas in the **DLXProf**, there is no interference and the results of execution are the same as those obtained with the simulator, without the profiler added.

There exists another trace-based performance analysis technique besides **TRAPEDS** that works somewhat like the technique being followed with **DLXProf.** It is called the *Abstract Execution Technique* which has been developed for tracing programs [15]. The main idea of this technique is to record a small set of events during a traced program's execution, which would serve as input to an abstract version of the program. This program would then generate a full trace by re-executing selected portions of the original program. Although the aim of **DLXProf** is not to produce program traces but to collect profile information, there is some similarity in the technique used in [15] and in this project. The trace in the former is produced by collecting some information from the **GNU C** compiler. The compiler has been modified to record some important events during compilation and a *scheme file* is produced with a description of how to interpret the trace data to recreate a full address trace. For this, the **GCC** has been modified at the **Register Transfer Language** (**RTL**) level. The **RTL** instructions are used to collect data necessary for tracing such as recording the address of a load instruction etc. Later, this tracing code will have to be executed before the instruction is executed. In the attempt to build **DLXProf**, one of the stages was to gather information from the **GCC** as mentioned earlier. This collection of information is done at the **Abstract Syntax Tree** (**AST**) and **RTL** levels instead of just at the **RTL** level as in [15], and instead of creating a separate *schema file* with this added information, the assembly file is modified to include it. This is the difference between the two techniques. Also, **DLXProf** uses this information to do profiling and not tracing.

Pixie [9] is another trace-based performance analyzer and it is a tool used to annotate the object code to help in tracing. This annotation is comparable to the annotation of the assembler code that has been done with **DLXProf**.

3.2 Profiling

Almost all the related work in profiling has been done on the conventional profiling explained earlier. The literature search that has been conducted has not revealed an existing implementation of the kind of profiling that has been done in this project. Some examples of profilers that were built for conventional profiling can be described as below.

1. **gprof** [12] : **gprof** is a **UNIX**-based call graph profiler that accounts for the execution times of routines that are being called in the execution times of the calling routines. Since it is often

desirable to increase the speed of execution of large programs, especially if a small portion of it dominates the execution time, **gprof** is a useful tool to evaluate alternative implementations that would reduce the execution time. **gprof** takes advantage of the structural and hierarchical nature of programs and provides a profile which can be used to compare and assess the costs of various implementations of abstractions. The overhead in the usage of the profiler is low.

gprof does the first two kinds of profiling mentioned in section 2. The approach used to gather execution count information is as follows. Monitoring routines are called at various points in the program to gather the *execution count profile*. This approach has the advantage that the counters used for profiling are managed separately by these routines, so that they do not interfere with the program being executed. For the same reason of non-interference, the monitoring routine should not produce trace output during execution, since this extra information would be unmanageable. Hence in **gprof**, the solution to this problem is to gather the profile information into memory at run-time and write it into a file when the program exits. A post-processor then processes this file to give a listing of the profile data. This solution also allows the combination of many executions of the program by this post-processor to give a profile of all of them. The monitoring routine counts the number of times a profiled routine is invoked. A call graph is drawn with the arc in the graphs associated with the count. By summing the counts on the arcs, the call counts for routines can be determined.

The second kind of profiling ie., gathering *execution time* information is done in one of two ways. In the first, the execution time of a routine is measured by finding out the time elapsed from routine entry to routine exit. In the second method, the value of the program counter is sampled at some interval and the execution time is determined from the distribution of such samples. This method will only provide a statistical approximation unlike the former which provides exact timings.

Once the arcs of the call graph have been gathered, the time for each routine should be attributed to the routines that call it. For this, a dynamic call graph is built with arcs from the calling routine to the called routine and times are propagated from the descendants to the ancestors after sorting the call graph topographically. Data is presented to the user in two formats. The first format called the *flat profile*, just lists the routines without regard to the amount of time used by descendants and the second called the *call graph profile*, incorporates the call graph of the program.

gprof is useful for improving routines. The programmer can identify poorly coded routines using the profile information and new algorithms can be evaluated. Another important use is in analyzing control flow of unfamiliar code. **gprof** helps such codes to get a clearer view of the structure of the program. Some drawbacks of **gprof** are that certain types of programs cannot be analyzed easily with it. These are programs that exhibit a large degree of recursion such as recursive descent compilers. But the profiler is easy to use and can be incorporated into the program being tested without prior planning by the programmer. The execution overhead added to the program being profiled is not much and the program can be measured in its natural environment. It does not need special operating systems or compilers.

2. **Parasight** [22] : It is very crucial to debug the performance of a parallel environment in order to make full use of the multiprocessor resources. Hence a profiler called **Parasight** has been built

for this purpose. It is a profiler tool which does non-intrusive performance analysis and high-level debugging. It is implemented as an interactive profiler over **gprof** which can be dynamically controlled by the programmer, while running concurrently with the target program. **Parasight** also allows the user to define custom profiling routines which will measure user-defined abstractions.

Existing **UNIX** profilers such as **gprof** have certain drawbacks such as their interference, and the large volume of output generated by them. Hence **Parasight** was an attempt to gain insight into these problems and implement the core of a profiler which is similar to **gprof** in its functionality, but with some improvements and differences. It allows very fine control of the extent and intrusiveness of the profiler with the target program's execution. It collects data about the elapsed time and also statistical data on program behavior.

Parasight runs under **MACH**, a **UNIX 4.3BSD** kernel implementation. It is a parallel programming environment for symmetric shared-memory multiprocessors. **Parasight** is a general purpose platform from which users can apply their abstractions to independently develop applications. Its key feature is that it can create and link "parasite" programs into a parallel application, which can then interact with the program memory to control program instrumentation in a transparent manner. These parasites are programs that monitor or observe the target applications.

When invoked, a multitasking environment is initialized and a large flat memory space is created. A new thread is spawned in the multiprocessing environment and a dynamic loader loads and links the target program, building a symbol table resident in memory to be used later to link many such parasites. Thus the parasite environment has been created and the target program is effectively enveloped in this environment. But this does not intrude with the target program's execution and no performance penalty is imposed because the profiler is also an independent thread that runs as a concurrent process with the executing program. These parasites will work on separate threads of control on separate processors than the ones used by the program. Since the threads also run in the same shared memory space as the programs, they have access to the program's global symbols and to the symbols declared in other parasites. All routines in the parasites or the programs are cross-callable ie., they can be accessed from other processes or threads.

Parasites can monitor a specific user program, or they can be used as a more general purpose low-level debugger or profiler. Instrumentation primitives called "scan-points" are used to further reduce interference, and these can be inserted and deleted at runtime. Scan-points are like break-points which can call arbitrary user supplied subroutines. These subroutines can be independently loaded and linked like parasites. But they differ from conventional break-points in that they do not use traps or system services. These scan-points inserted at runtime at the beginning of each program function, basically substitute for the compiler inserted calls to the profiler subroutines. Provision has been made not only to support experimentation, but also rapid prototyping of general purpose tools, by a statement level C interpreter implemented as a parasite.

Parasight has been very useful in collecting preliminary measurements to identify bottlenecks in a program. It provides a broad overview of the program's behavior and allows the user to zoom-in on problem areas. It also allows the programmer to formulate different abstractions and apply them to the same program to extract different kinds of information. Another unexpected result of the work on **Parasight** was that it was found useful in profiling sequential programs also. Using parallel processing with a shared memory architecture helped get an interactive profiler that would not have been possible otherwise. The sequential program formed one thread and the profiler formed another independent thread that could interact with the program.

3. **SEPDS** [4] : In distributed computer environments, many factors determine system performance. The main problem affecting performance here is the impact of synchronization on processor utilization. Hence profiling can be useful to obtain the parallelism profile, synchronization profile, and speed-up profile of programs running in distributed environments. A system called **SEPDS** (Support Environment for Prototyping Distributed Systems) has been developed at Delft University of Technology using Verdix Ada on Sun workstations [2]. This system assists the designer to solve the problems of partitioning systems into sets of processes, to determine the appropriate level of parallelism and the allocation strategy. The system provides tools for prototyping, simulating and profiling a distributed system to support these features. Profiling has been done with the **SEPDS** based on the client-server model.

   The paper [4] actually discusses how the simulation approach was used to find the synchronization overhead and parallelism, using the profiler as one of the tools with the simulator. It then compares the simulation approach with the **SEPDS** prototyping approach and compares the results of the two approaches. The profiling is only a portion of this simulation and we will concentrate only on it. A discussion of the results of this simulation or the prototype approach in detail is outside the scope of this report.

   The design and architecture of **SEPDS** is based on the Extended Data Flow Graph model (**EDFG**). The model consists of actors that describe operations, links that receive data from one actor and distribute it to one or more actors using connecting arcs called edges. A node is enabled for execution when the appropriate input arcs contain tokens. An enabled node takes tokens from its input arc and produces tokens on its output arcs. The **SEPDS** profiler is a simple profiling tool based on the same model. It consists of a kernel and some profiling utilities. The kernel is similar to an operating system kernel with a two level scheduler and a set of primitives. The two levels of schedulers are the object schedulers and the actor schedulers. In the profiling process, the object scheduler picks up one object at a time from the prototype and sends it to the actor scheduler. For each object, a set of actors ready for execution are identified. The actor scheduler executes these actors according to the **EDFG** and calls the appropriate primitives which move or send the tokens from the input to the output channels relevant to the **EDFG**. Once an actor is fired, profiling information for the actor is collected.

   The various kinds of profiling information that can be collected are parallelism profile which shows the available parallelism in the current profile, actor profiling which shows the execution and frequency of the execution of the actors, synchronization profile which shows the time spent in synchronization between objects and the communication profile which shows the overhead in communication through the communication medium. Since the main problem in increasing the performance of distributed computer environments and parallel applications is the impact of synchronization overhead caused by the intrinsic properties of the application and the architectural and organizational features of the system, the **SEPDS** system has been used to profile this information. The particular focus is on such issues as how many processes are waiting for resources and how many servers are involved in servicing the requests.

4. **TPROF** [10] : In this paper , the authors discuss the use of profiling and some other principles in measuring the run-time performance of 3D graphics systems. Some of the tools they have developed for this purpose are the **TGEN**, a tracing program that intercepts graphics library calls from an application program and records them in a file, and the **TPROF** which is a profiler that interprets the trace file and computes workload statistics. Thus, here, we see another use of profiling.

   TPROF profiler was developed to better understand the workload presented to a graphics system by real application programs. **GR** is a graphics interface. **TPROF** feeds a trace to the **GR** profiler module. This module then tracks state changes and counts primitives. There is an interpreter that operates a graphics pipeline in a feedback mode, returning primitives after the various graphics operations like transformation, clipping and culling. The profiler then uses this information to compute statistics on what is actually sent to the rasterizer. The profiler generates three kinds of data : global, geometry related, and pixel related, which would give a better idea as to how the performance can be improved in all the three aspects. The developers of this system are also implementing a profiler called **SPGEN** that generates a system profile for a real system. **SPGEN** is a portable and application-independent benchmark that measures the performance of a system's graphics-pipeline stages by executing and timing small sequences of GR calls.

5. Finally, in the context of related work done in this area, a discussion of a framework for determining average program execution times and their variance with profiling [20] is necessary. It is important for a compiler to obtain estimates of the execution times of subcomputations of programs in order to determine the overhead values in the target architecture. For this, a counter-based execution profile of a program gives the frequency information and this is used to compute the average execution time and variance values. The program's interval structure and control dependence graph are used as the basis for this purpose.

   Frequency values are obtained from a counter-based execution profile of the program. This is done in a slightly different way from the naive approach of incrementing a counter once per basic block, by using the control dependence relation to increment the counter once for each control dependence branch condition. Such an approach based on the control dependence and counter variables would be useful for any type of code optimization that needs frequency information. The **PTRAN** or Parallel TRANslation system developed at IBM Research [8] forms part of the framework used for estimating execution times. It contains a program database that can be used to store the frequency, execution time and variance information. The execution time information in **PTRAN** is being used to partition the input program into jobs for parallel execution.

   The execution frequencies of conditional branch probabilities and loop frequencies can be estimated by program analysis methods also, but it is important to use the execution profile of the program in order to obtain these values because the former is too restrictive, and the latter comes in handy when compile-time analysis is unsuccessful. Most execution profilers that are found are connected with the operating system and do run-time sampling of the program counter. This gives a realistic measure of the relative execution time spent in each procedure for a particular execution run. But this approach fails when we want to determine the execution frequencies of individual statements or small procedures due to the coarse granularity of the sampling interval. This makes it necessary to use a counter-based profiler. Apart from being very flexible in that the frequency information can be generated on any machine, it can be used to estimate execution times for different optimizations of the program on different target architectures. It eliminates any need to make calls to the operating system for the profiling because the profiling code is directly compiled into the program. But the overhead of introducing extra code in a program is still present, which is eliminated in the strategy that has been followed with **DLXProf**.

Now, focusing on the approach used to determine the average execution times for all nodes in the forward control dependence graph, for a simple architecture, it would be enough to count the number of instructions required to implement a primitive operation. A more sophisticated approach has to be used for pipelined architectures, vector instructions or the effects of cache usage. These approaches are not discussed in depth since they are beyond the scope of discussion in this report.

4 Implementation of DLXProf

There are two basic software components that are necessary for the implementation of **DLXProf**. The first is the **GCC** compiler and the second is the **DLX** simulator. **GCC** forms the basis for the profiler to collect information at the compilation level from its abstract syntax tree and Register Transfer Language intermediate representation. This information would be passed on to the assembly code which the compiler generates for the **DLX** architecture. Hence the implementation at this level involved studying the structures of **AST** and the **RTL** and getting the required information out of the **AST** into a structure appended to the **RTL**, and finally including it in the assembly file.

At the execution level, the **DLX** simulator had to be modified to read in this modified assembly code. This has been included in the "load" phase of the simulator. During the "simulate" phase, the information has to made use of to do the profiling. Another aspect of the profiling during execution is to gather information about memory references that come from stack, heap and global references. This has also been implemented. All the implementation is in the C programming language.

Based on the above discussion, the implementation can be divided into two phases.

4.1 GCC Phase

The **GCC** phase involves the gathering of information at the **Abstract Syntax Tree** (**AST**) level and storing it into the **RTL** structure, from where it is given to the assembly file as special comments. This has been implemented as follows : The **GCC** creates the **RTL** as the intermediate representation for a program. The frontend of the **GCC** generates the **RTL** code for each function and hands it to the backend of the compiler. The **RTL** represents intermediate code in a high-level, algebraic form that describes what the three address instruction does [6]. All the phases of the backend then work on this **RTL** code and assembly code is finally generated. This form of intermediate representation makes the code easily retargetable and hence the choice of the compiler. Another reason for choosing this compiler is that it produces assembly code for the **DLX** simulator. This simulator is an essential part of this project due to the following reasons :

- One of the main aims of this project is to build a totally non-intrusive profiler. If the profiler code is built in along with the code for the simulator, this objective can be attained.
- Secondly, It is necessary to have a simulator that can be modified by researchers without special permission and the **DLX** simulator is popular among researchers for this purpose.

Also, it is necessary to have a C compiler that can be modified easily without special permission and the **GNU C** compiler is free software that can be used for research. Hence, the **GCC** and **DLX** simulator are essential components for this project.

Information about local and global variables and arrays and pointers has to be gathered at the very early stages of **RTL** generation from the **Abstract Syntax Tree**. At this stage, the **AST** is traversed and **RTL** is generated for each expression in the tree. This is where we have the information about the variables like, their names, type etc., that we need to pass down to the **RTL**. So as each expression is converted into **RTL**, the information has to be pulled out of the tree and tagged onto the **RTL** instruction(s) for that expression. An **RTL** instruction is represented with a recursive data structure. All such instructions are chained together in the form of a doubly linked list. For more details on the structure of the **RTL** instructions, see elsewhere [6].

The data structure used to store extra information in the **RTL** for profiling is shown in Fig. 2. The entry "pointer_to_info" is a pointer from each **RTL** instruction to the structure shown. The entry "operand_array [3]" is an array whose three elements are going to hold information about the three operands of any instruction. Each element is a pointer to a structure called "measure_array [MAX_INFO_NUM]" that would hold all the information about that operand. This structure that would hold the information could be anything, since the arrays are void pointers. For instance, to mark operand 1 as containing a local variable, the pointer operand_array[0].array_ptr[0] would be cast to an "enum" (in C programming language) element "L" representing a local variable. If operand 2 held an array element, the pointer would be cast to an enum "A" and so on. If it is required to store the name and scope of the variable that is being referenced, the structure can be more complex to store this information also.

The Abstract Syntax Tree representation of the program in the **GCC** has been studied in detail. For more details on its structure, refer elsewhere [14]. It is possible to get the desired information from the tree about variables and this has been done as a part of this project. The necessary parts of the tree at which the information can be gathered have been identified and the information is ready to be passed on

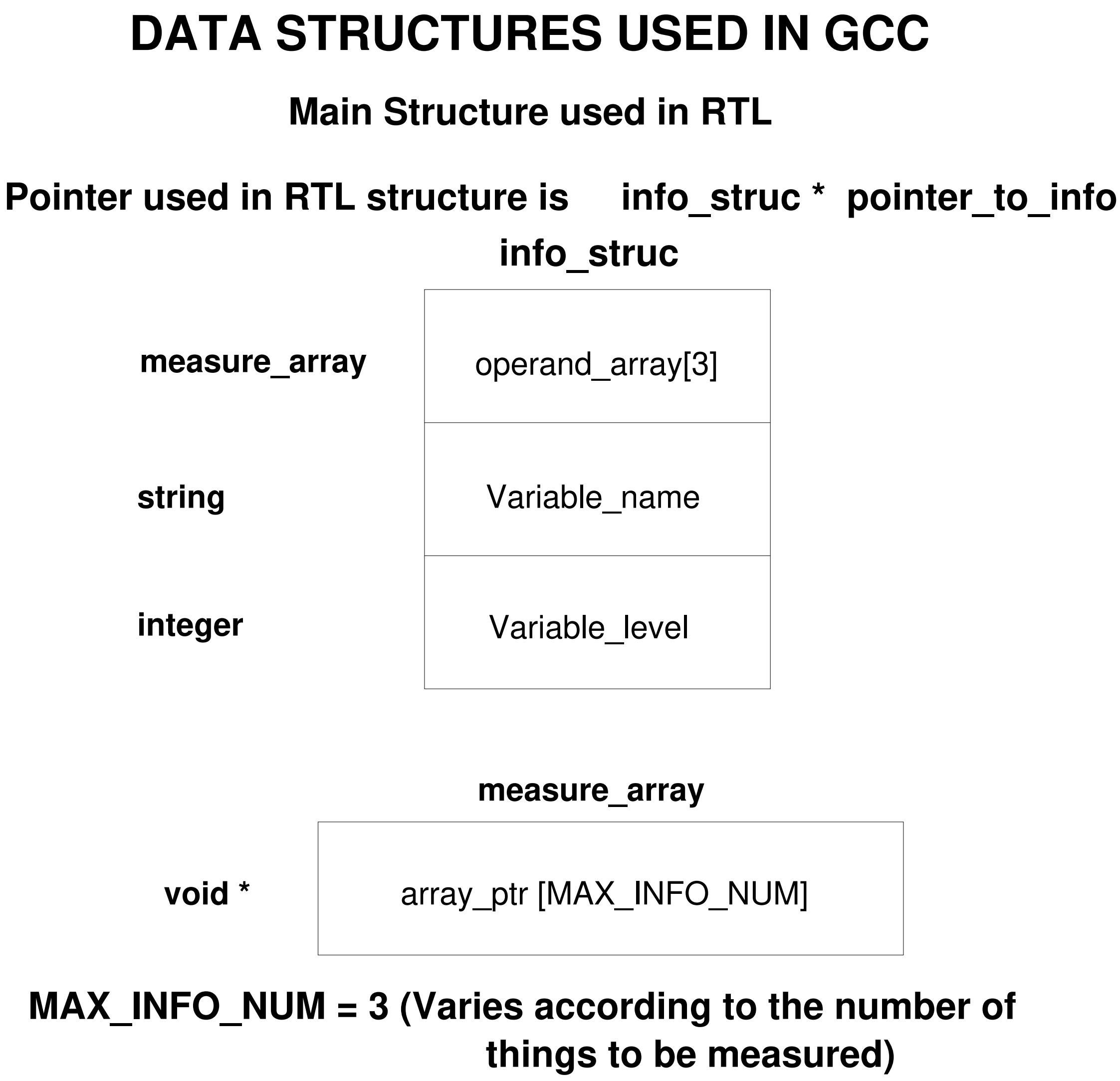

Figure 2: Data structures in RTL

to the backend of the compiler to be included in the assembly code when it is generated. The files of the **GCC** that have been modified to accomplish this are "expr.c" and "statement.c".

Later when assembly instructions are generated from these **RTL** instructions, this extra information from each **RTL** instruction is included in the assembly file as special comments enclosed between semicolons. This implementation is in the file "var-info.c". This file has all the routines needed to read in the information from each **RTL** instruction, and extract it. This extracted information is passed to the routines that generate the assembly code in file "final.c", to be included in the assembly file. The routines in "var-info.c" are called from routines in "toplev.c", which is the top level file that calls all the stages of compilation after the parsing stage. The routine "rest_of_compilation" is responsible for using the **RTL** instructions and invoking all the optimization stages with it before generating assembly code. The routines in "var-info.c" are called by this function. A sample assembly file with special comments added would appear as shown in appendix A. From the file we see that there are three words on each line of comment giving some information about the each of the three operands of the instruction.

The approach used here for gathering information from the **GNU C** compiler is a novel idea. But it was not successful due to the following reasons :

- **RTL** generation from the **AST** was a very complex process and was not easily understandable. It was a one-to-many mapping between an expression in the tree and an **RTL** instruction, and the C code for generating these multiple instructions was too complex. Hence the information gathering was not a very easy task. It would require a simpler intermediate representation than the **RTL** to accomplish it.
- The next few stages after the generation of the **RTL** are the back-end optimization stages. For more details on the various optimizations performed in these stages see elsewhere [6]. When we see how these optimizations are implemented, it is clear that many of the **RTL** instructions are moved around in the doubly linked list structure and also some deletion and insertion of **RTL** instructions goes on. Hence it was found that the information that was gathered while the **RTL** was generated was not left undisturbed until assembly code was generated. This resulted in the loss of some of the vital information gathered for profiling.

Though this approach is unsuccessful at present, the data structures and routines have been left as such to use in case a simpler intermediate code generator is available to use with it. The idea has not been abandoned.

4.2 DLX simulator phase

The second phase is at the **DLX** simulator level. This involves the actual implementation of **DLXProf.** There are two parts to this profiler which will be discussed shortly. The simulator operates in two phases to simulate the execution of programs : the *load* phase and the *simulate* phase. During the load phase,

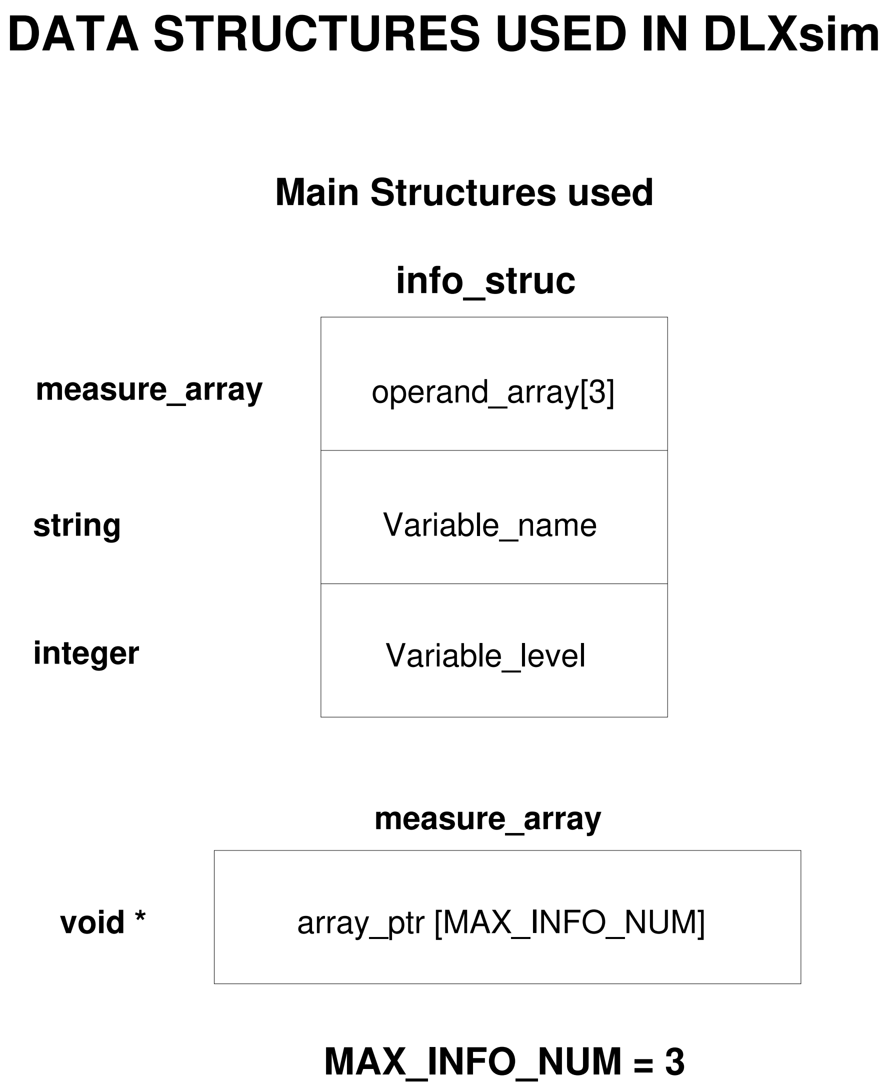

Figure 3: Data structures in DLXsim

opcodes and operands are determined for the instructions, and the program is loaded into a memory array. During the simulate phase, one instruction at a time is taken off from this array and executed. The load phase, hence, reduces the time necessary for execution during the simulate phase.

Since it has been assumed that some day a C compiler with a simpler intermediate representation will be available to generate assembly instructions with special information in the form of comments, the simulator has been prepared for this eventuality. For an assembly file with comments added, a modified **DLX** simulator which can read in and interpret these comments along with the normal reading-in of the assembly instructions into the memory array is necessary. For this, the structure shown in Fig. 2 has been recreated in the simulator as shown in Fig. 3. The assembly file would contain some information about the three operands of an instruction such as, the type of the operand (either local, global, array or pointer type) and the name of the variable and its scope. These would be read from the file and put into the appropriate operand indices in operand_array, variable_name and variable_level respectively. There is a pointer from every assembly instruction in the memory array to such a structure. When each instruction is executed from the memory array during the simulate phase, a *profiler* is invoked if the instruction happens to be a memory reference instruction. Let us call this profiler 1 and it works as follows.

It reads in the extra information tagged onto that instruction and increments counters for local variable references if the information happens to say that the memory reference is to a local variable, or a global variable counter if it is a global memory reference and so on for pointers, arrays etc.

At present, the profiler has the capability to do this if the assembly instructions are augmented with comments. But it can also do a different kind of profiling which is independent of whether comments are added to the assembly file or not. This profiling determines the number of memory references that come from stack, heap and global memory accesses. Let us call this profiler 2. It has been implemented as follows.

Whenever there is a memory reference, profiler 2 is invoked. Memory references can be of three types, references to the stack space, references to the heap and references to the global address space. The simulator supplies the memory needed for the execution of any program as follows : It asks the system for a large block of memory which it numbers from 0 to 65535 as shown in Fig. 4 (which is the default size. If larger programs are to be run, the user has to specify how much is required and the simulator will get it from the system). Then, during the load phase of the program, it loads the assembly code starting at address 0x0000. At the end of the assembly code, it allocates some space for global variables (it determines how much is needed for global variables, global arrays, etc. during the load phase from the code that is being loaded in). This forms the global address space. At the end of this, the heap begins and grows upwards. The stack however, begins at 65535 and grows downwards. There is no limit as to where the stack ends or where the heap ends. They both work their way towards each other. At different times in the program, a location that was previously a stack location, could now be a heap location because of a memory allocation that was done by the program. The three spaces are as shown in Fig. 4.

DLXProf maintains three tables, one for stack references, one for heap space references, and the third for global memory references. The structure of the tables is shown in Fig. 5. The table is maintained

Division of DLX Memory

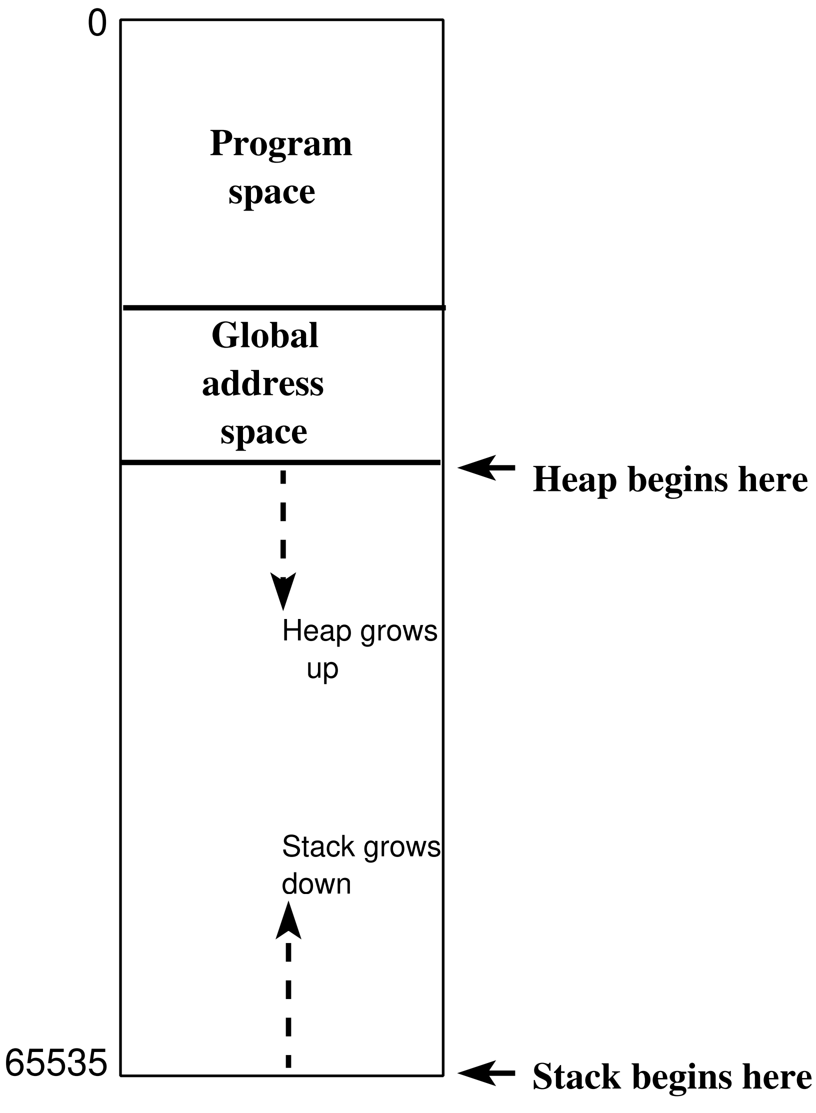

Figure 4: Heap, stack and global address spaces

in the form of a balanced-binary tree so that the search and insertion do not take up considerable time. This structure was chosen because it is one of the most efficient for searching and inserting elements into a list. These operations can be completed in O(log N) steps, where N is the number of items in the list. Hence for larger N, we can see that this structure will be very efficient. From the structure, we can see that the address that is being referenced is stored, along with the information collected about the address during profiling, which includes the number of times the address location was referenced during the entire execution, and a total of the cycles during which the access was made in order to determine the frequency of access to the location at the end.

The profiler then takes advantage of the division of memory into global space, heap space and stack space to determine in which of these three spaces the address of the memory location being referenced lies. Once it has determined this, it searches the appropriate table it to see if the address being referenced already exists. If it does, it increments a counter that counts the number of times that particular address was referenced. It calculates the number of cycles before which this address was last referenced and keeps a total of this cycle difference to be able to calculate the frequency of access to that memory address at the end of execution. If the address was not already referenced, there is no entry in the table and a new entry is made.

From the information in this table, two kinds of graphs are plotted to show the results of the profiling, one that shows the percentage of times each address location was referenced during the execution, and another that shows the frequency of access to each memory location in the stack, heap and global address spaces. It is also possible to plot the number of times an address location was accessed once every cycle, the number of times an address was accessed once every two cycles and so on. This could show the frequency of access to address locations in a different light. There is another data structure that is maintained to store this information. This is an array of integers which is indexed using the cycle of access. For instance, index 1 of this array would be incremented each time an address was accessed every one cycle, index 2 would be incremented for every access in 2 cycles and so on. This is also plotted after execution.

All of this profiling is done without interfering with the program being executed. This helps to overcome one of the main difficulties of conventional profiling, which is the intrusiveness of the profiler. Conventional profiling places a limit on the profiler code that can be added for this same reason, which has been eliminated in this project.

5 Results obtained with the profiler

DLXProf can prove useful in studying performance of programs with regard to the memory references that are made, since a lot of the execution time is spent in doing memory accesses. One of the long-term goals of this profiler is to collect information about pointer and array accesses using **GCC**, but at present, the profiler gathers stack, heap and global memory access information using only the **DLXsim** for reasons discussed earlier. Heap and remote access information can prove particularly useful in the analysis of cache performance.

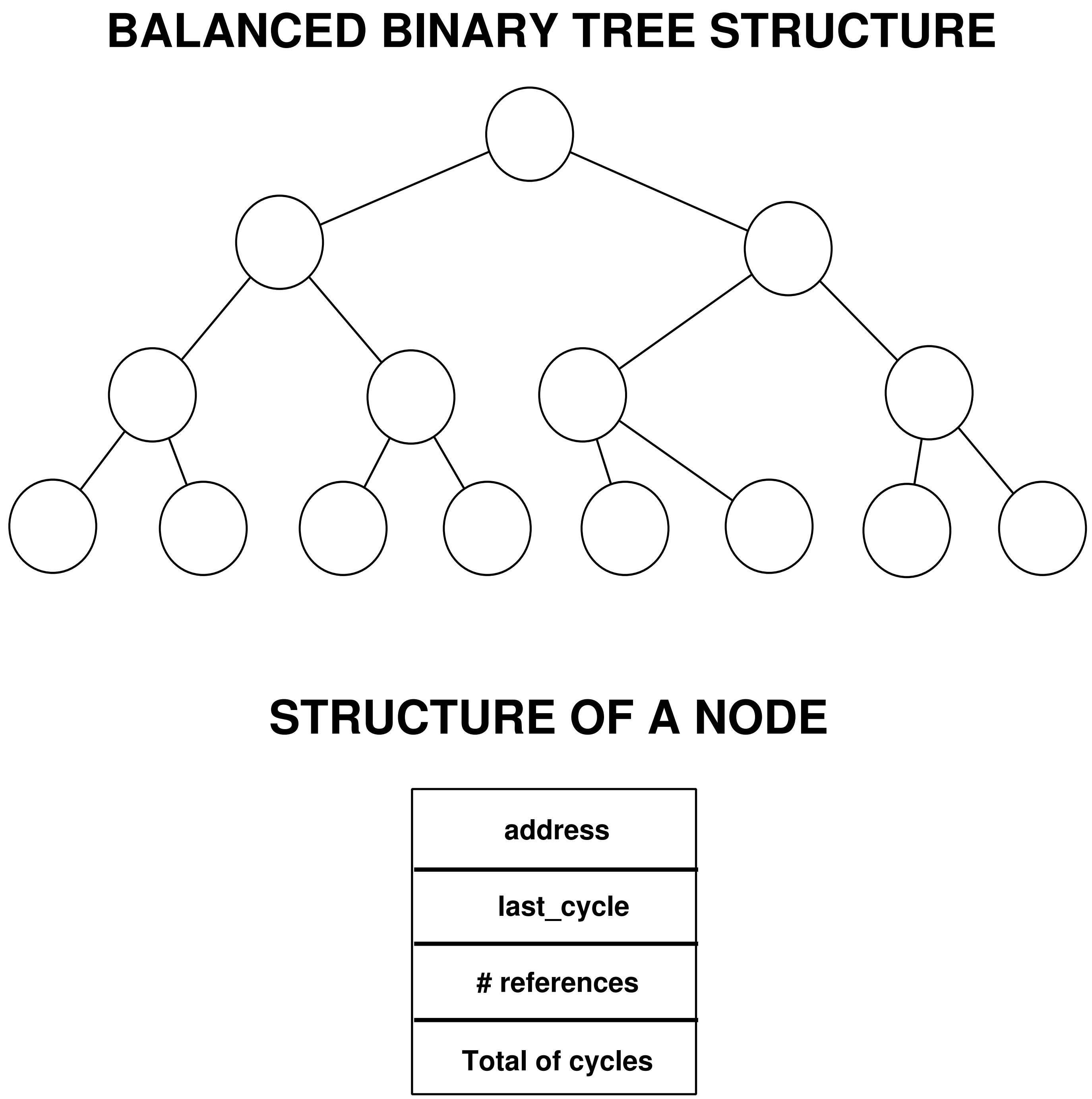

Figure 5: Data structure for profiling global, stack and heap references

5.1 Measurements

DLXProf collects the following information about stack, heap and global memory accesses:

1. The *Cycle Behavior* of programs in the stack, heap and global memory. This keeps track of the machine cycles during which access was made to a particular memory location in the heap, stack or global memory and gives statistics as to what percentage of references were made once every 1 cycle, what percentage was made once every two cycles, and so on, until 100000 cycles. Above this number, the references are counted in a separate counter. It was found from all the C programs that were tested on the profiler that no references were made beyond 75000 cycles.

2. The *Average Access Time* of memory addresses given in cycles. This keeps track of the total of all the machine cycles at which a particular memory location was accessed and the total number of accesses that were made to the location, and from these, gives the average cycles elapsed between accesses to that location. This is done for all the three kinds of memory accesses.

3. The *Number of Accesses* to a particular memory location, either in the stack, heap or global address space. Considering stack accesses, this is calculated by keeping track of the total number of references to a particular stack location and, finding the percentage of this over the total number of stack references made during the entire execution of the program. thus giving the percentage of times this location was accessed during the entire execution. The same is done for heaps and global accesses. It is a trivial change to the implementation of the profiler to obtain the actual number of times each location was accessed, rather than get this data as a percentage.

The profiler writes out these results into files which can be directly fed into graph plotting programs for plots.

Several programs and benchmarks were profiled with **DLXProf** to get the above profiles and graphs were plotted with these results. A discussion of the programs that were profiled and the observations made follows.

1. **Tomcatv:** This program is a vectorized version of a mesh generation algorithm with Thompsons solver written by W. Gentzsh and F.H Regensburg of Germany. This program manipulates two dimensional arrays and was used as a test to study the profile results of programs that use arrays. See appendix A for a C source listing of Tomcatv. This program was originally written in **FORTRAN**, but has been translated into C using the utility "f2c". Figs. 6 to 14 show the results obtained by profiling **Tomcatv**.

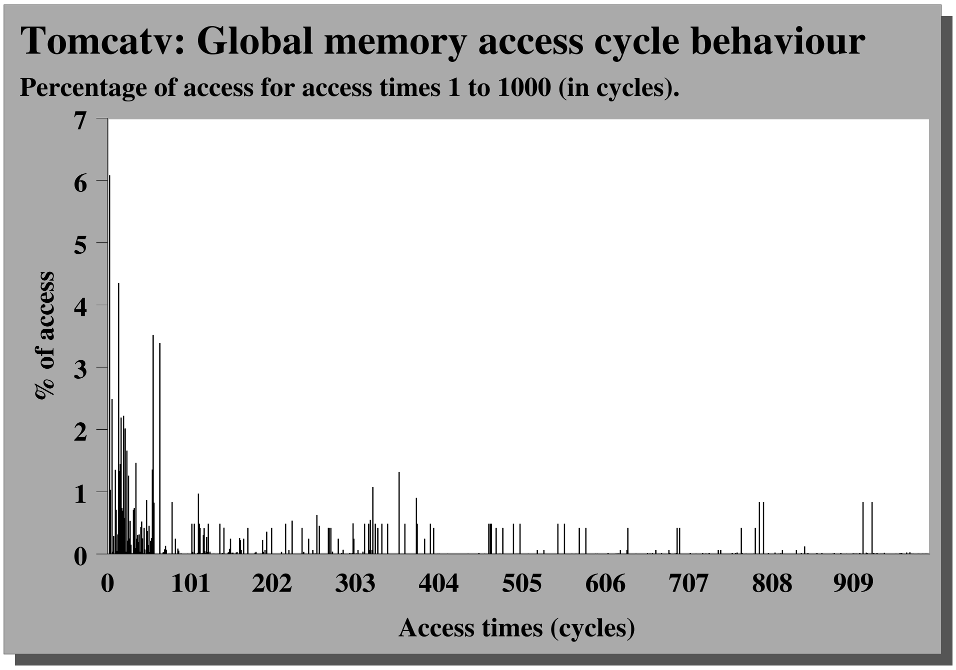

Figure 6:

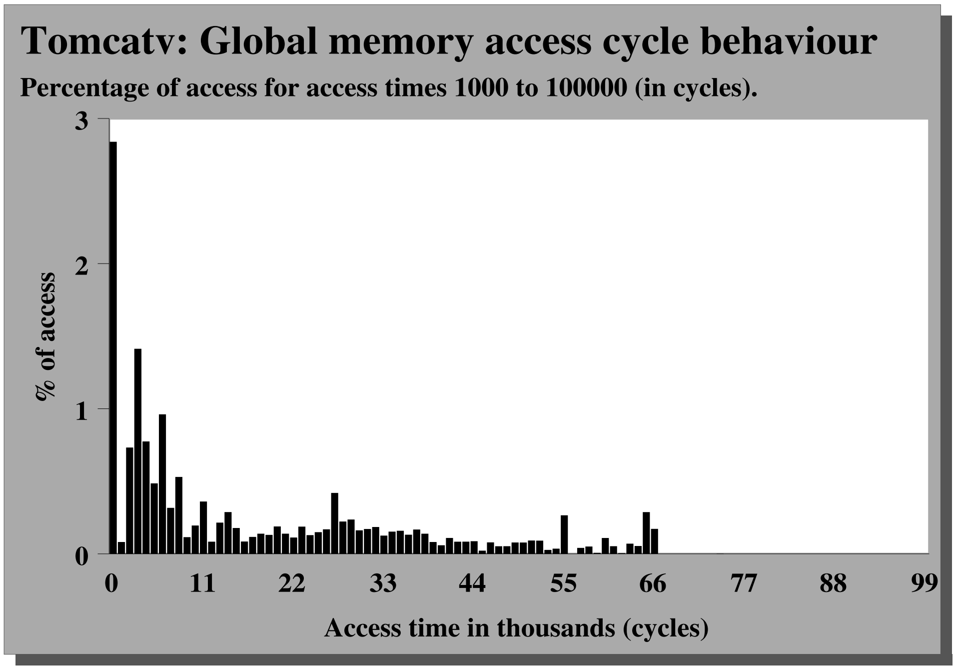

Figure 7:

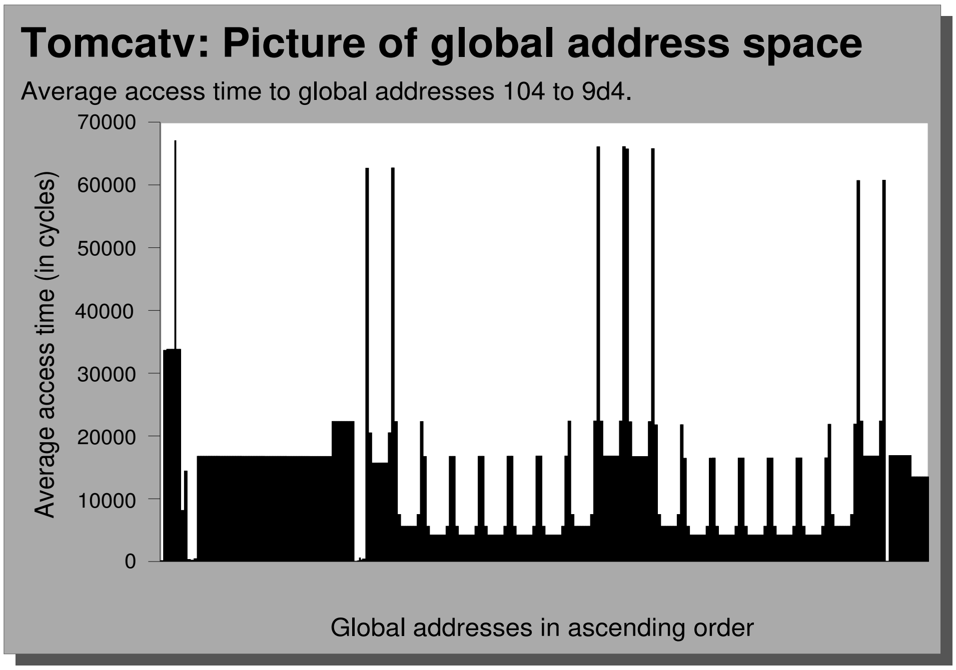

Figure 8:

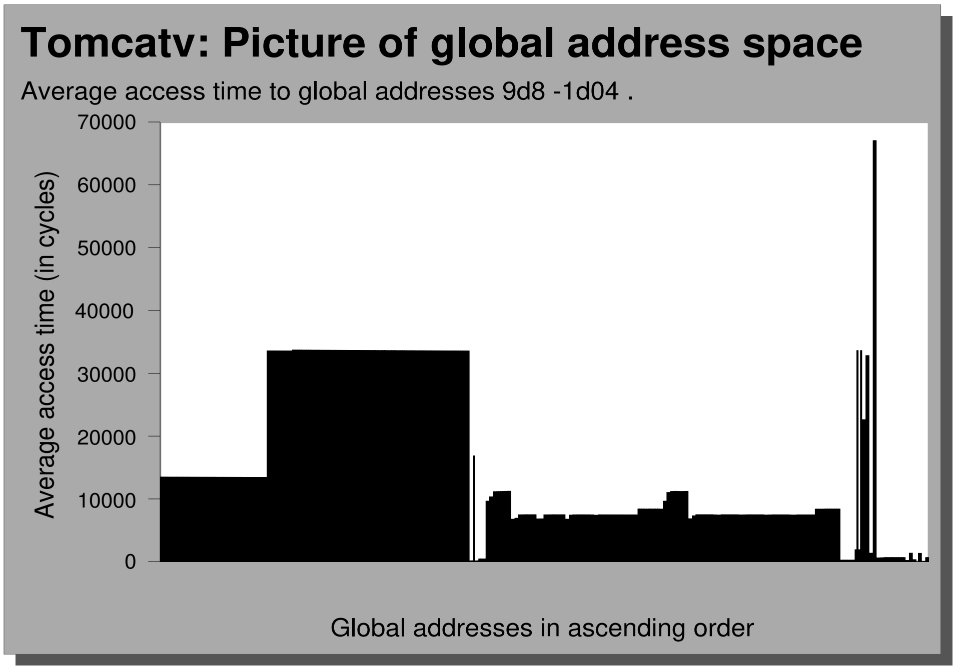

Figure 9:

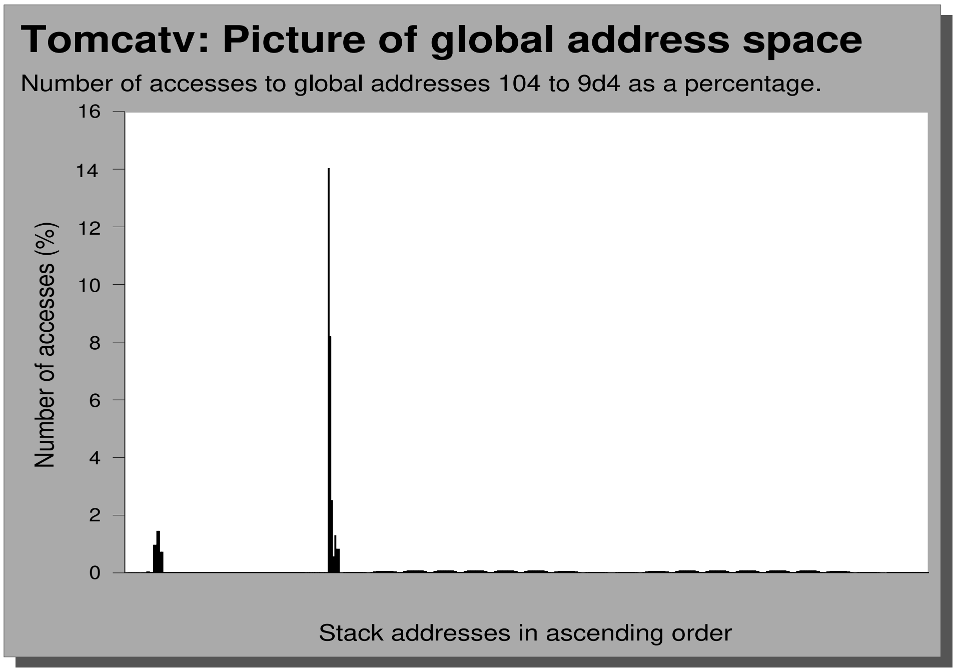

Figure 10:

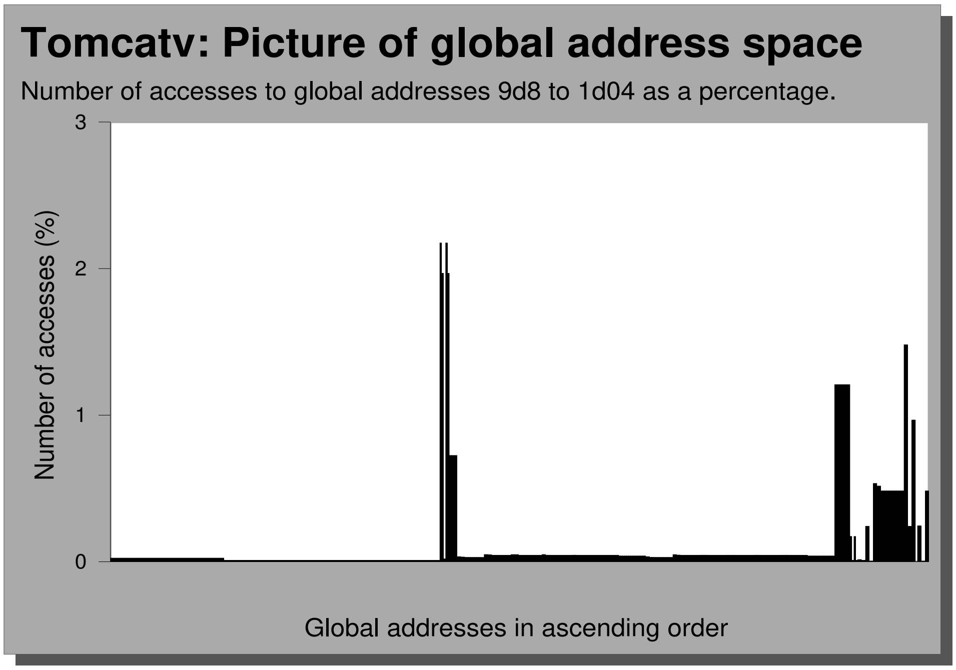

Figure 11:

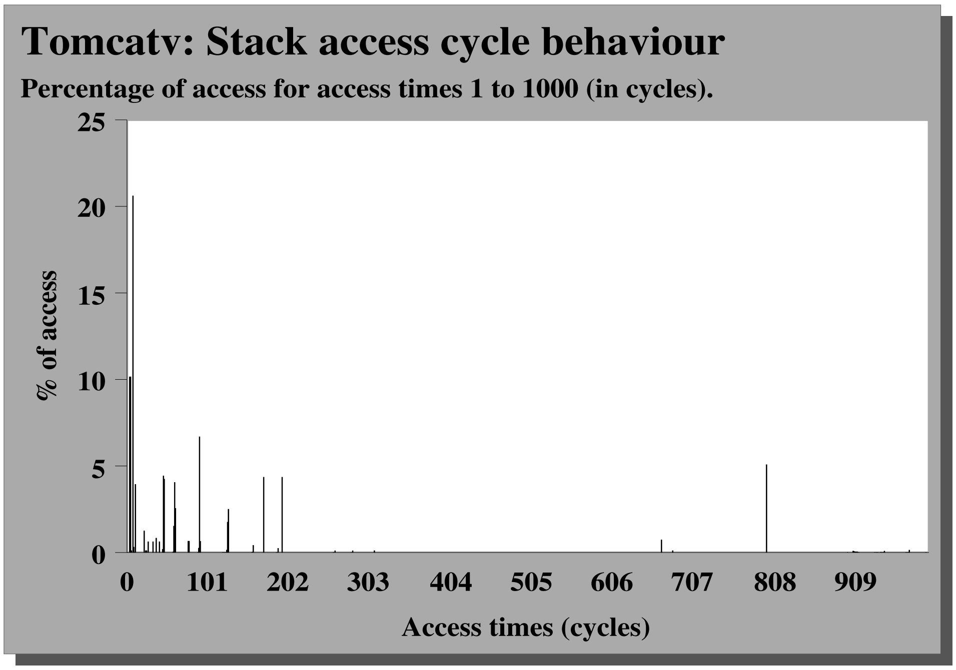

Figure 12:

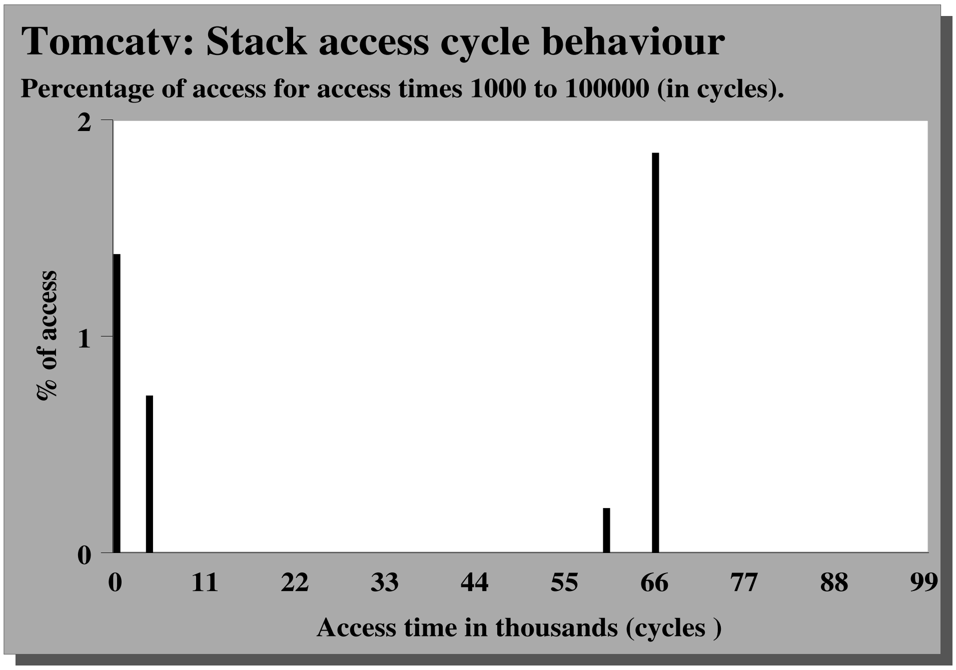

Figure 13:

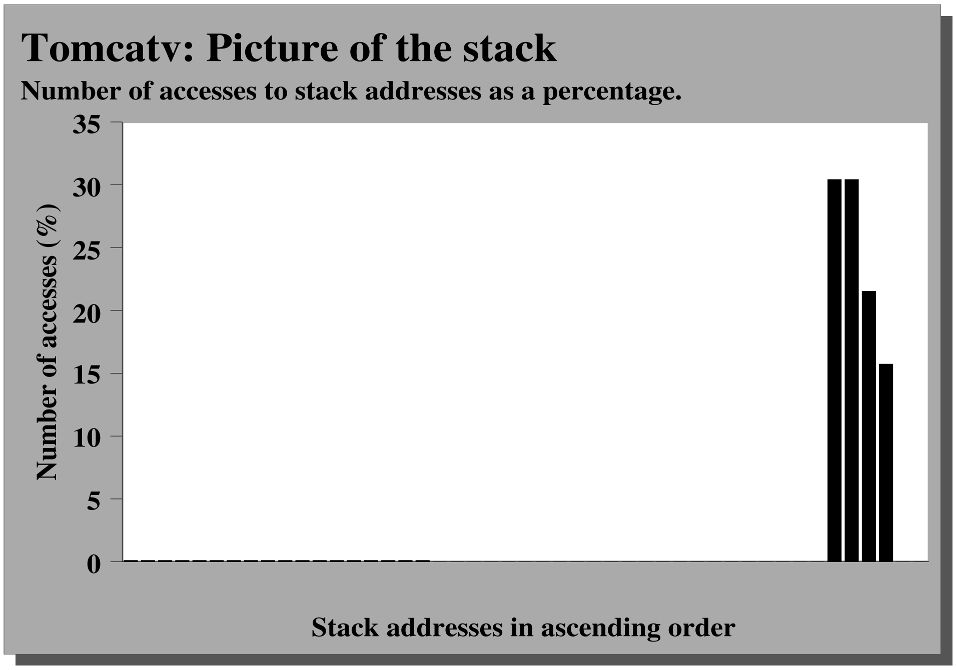

Figure 14:

2. **Bubble sort using arrays:** This program (see appendix A) sorts an array of numbers using the bubble sort algorithm. The numbers to be sorted are generated in a random fashion. The size of the array is 500, and is single dimensional. The profile results can be compared with those of **Tomcatv** to see if there are similarities in the results obtained from programs that use arrays. Figs. 15 to 19 show the results obtained by profiling the **Bubble Sort** test.

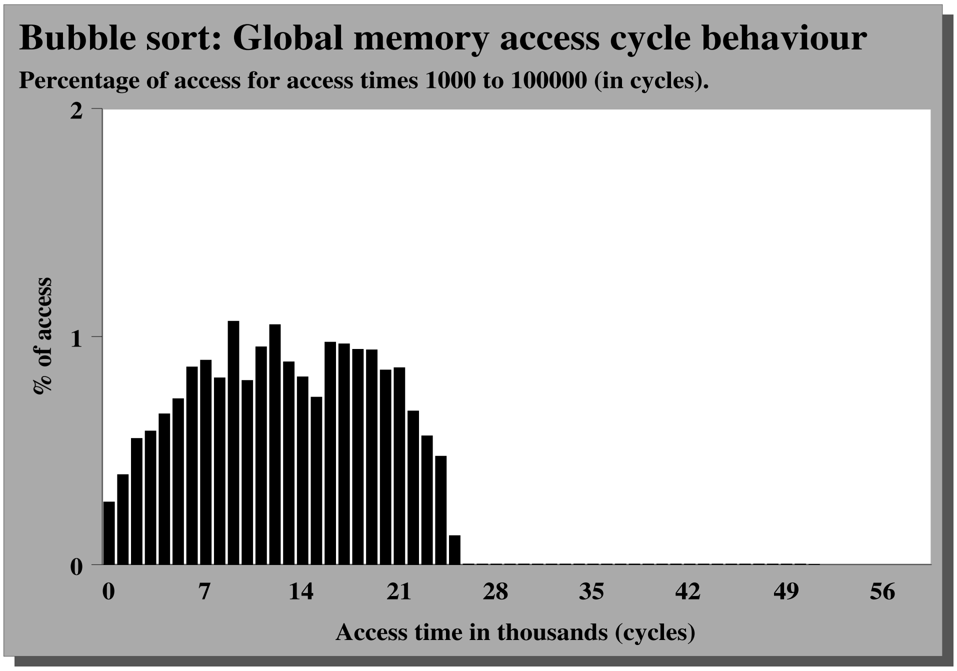

Figure 15:

3. **Towers of Hanoi:** The Towers of Hanoi is an interesting problem to study recursion. The problem is concerned with a popular children's game consisting of three poles and a number of different sized disks. Each disk has a hole in the center, allowing the disks to be stacked around the poles. Initially, the disks are stacked on the leftmost pole in order of their size, with the largest on the bottom, and the smallest at the top. The object of the game is to transfer the disks from the leftmost pole to the rightmost pole, without ever placing a larger disk on a smaller disk. Only one disk may be moved at a time, and each disk must always be placed over one of the poles. The program that was tested solves this problem using recursion. See appendix B for a listing of this program. Figs. 20 and 21 show the results of profling this test.

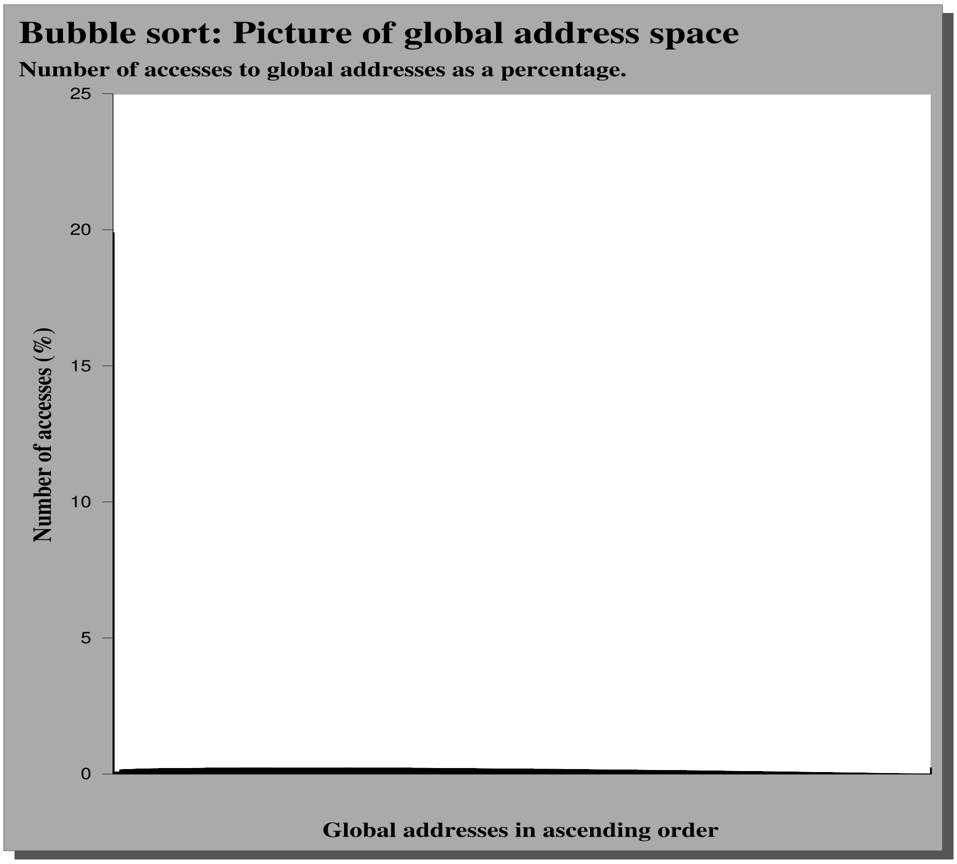
Bubble sort: Picture of global address space
Number of accesses to global addresses as a percentage.
25
20
15
10
5
0
Number of accesses (%)
Global addresses in ascending order

Figure 16:

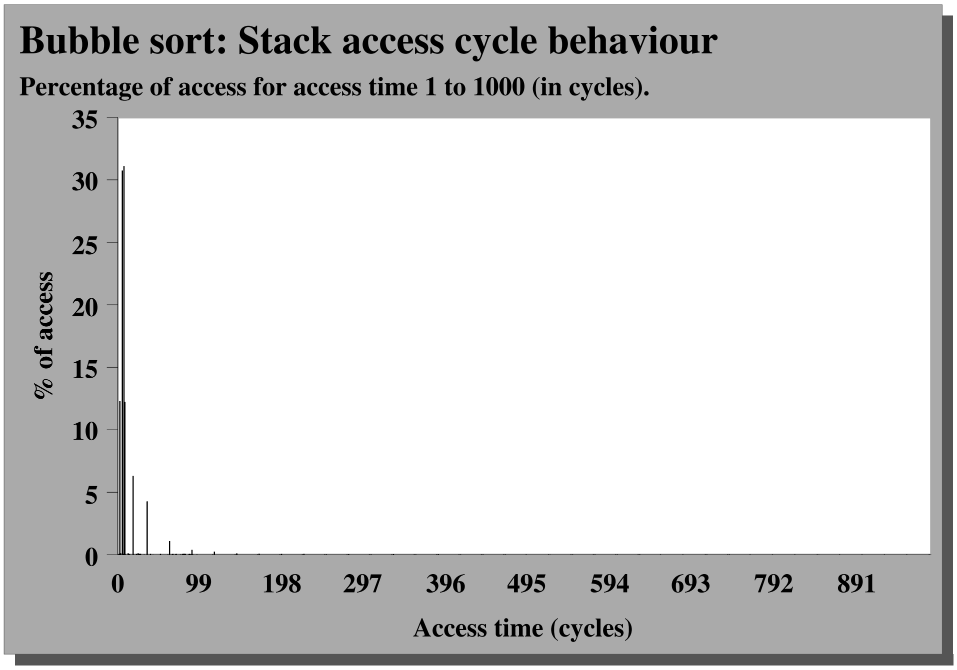

Figure 17:

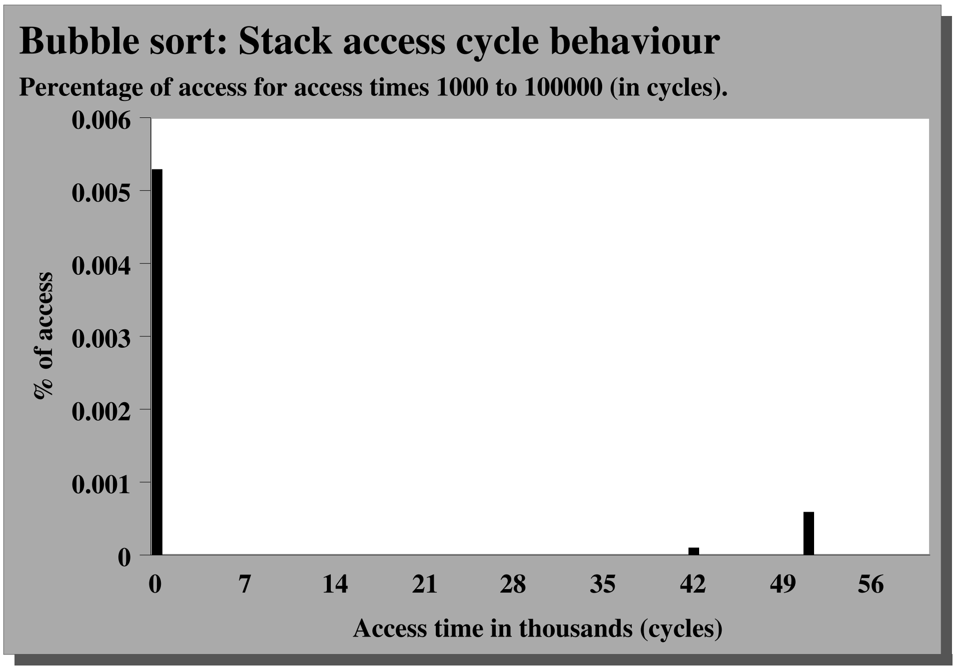

Figure 18:

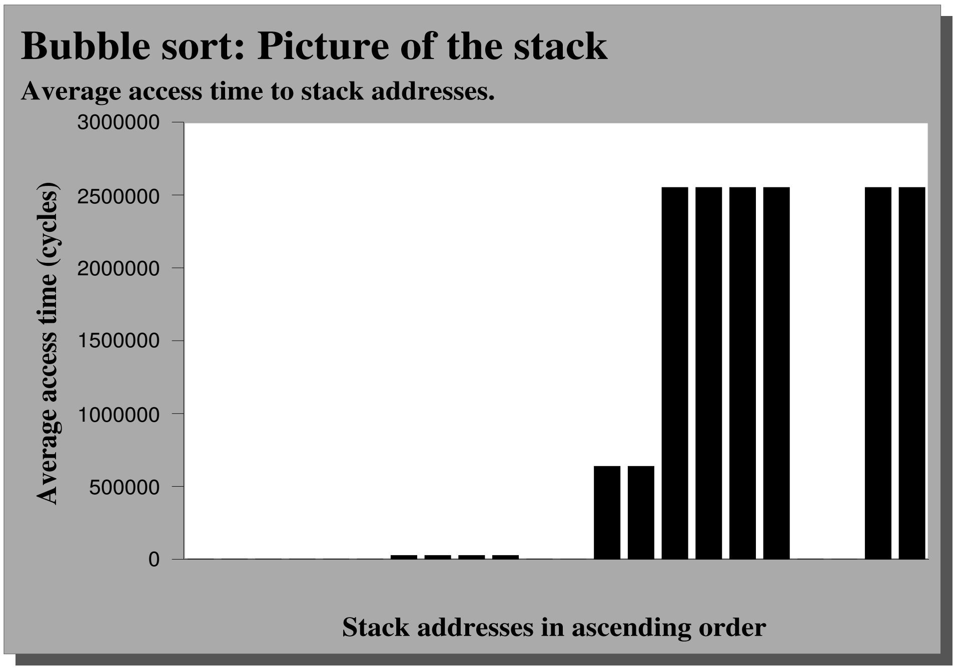

Figure 19:

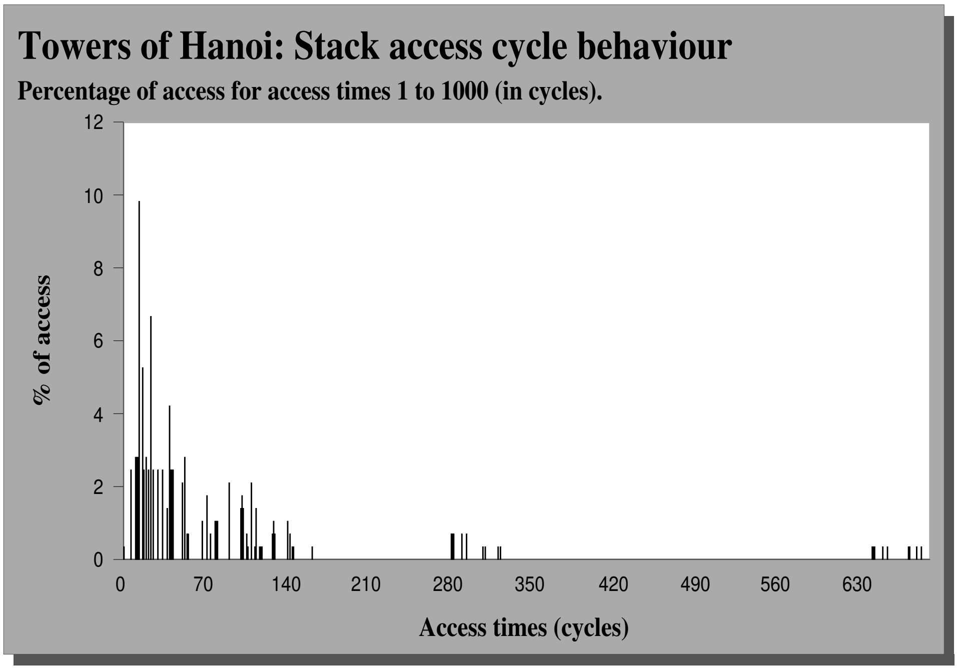

Figure 20:

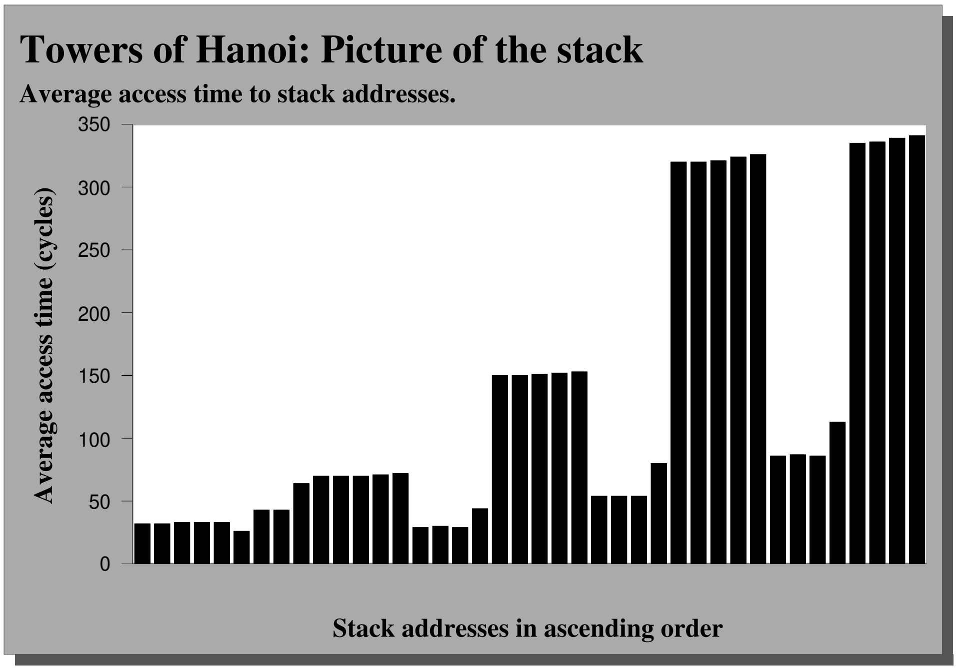

Figure 21:

4. **Fibonacci Numbers:** Fibonacci numbers form an interesting sequence in which each number is equal to the sum of the previous two numbers. The program that has been tested here determines the first 20 Fibonacci numbers. This program also uses recursion to calculate the sequence and so the results can be compared with those of the Towers of Hanoi problem to see if there are similarities in the profile results for recursive programs. See appendix B for a source listing. Figs. 22 to 25 show the results of profiling this test.

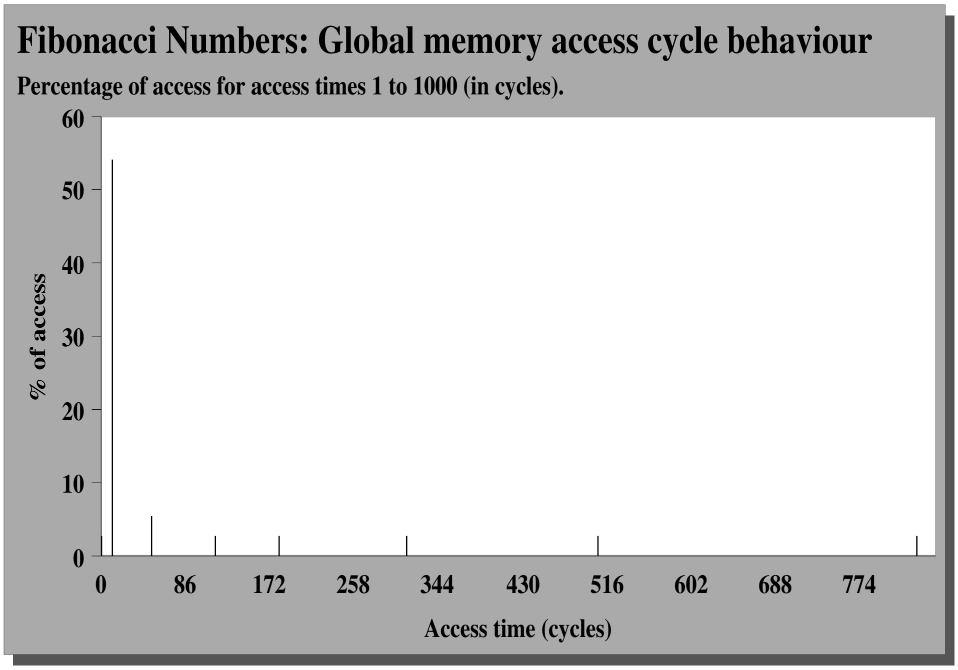

Figure 22:

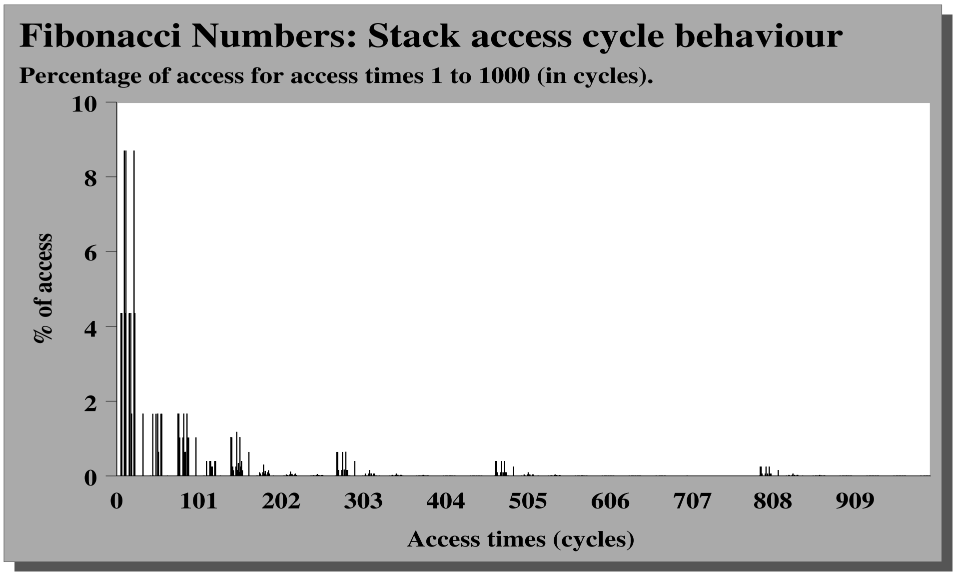

Figure 23:

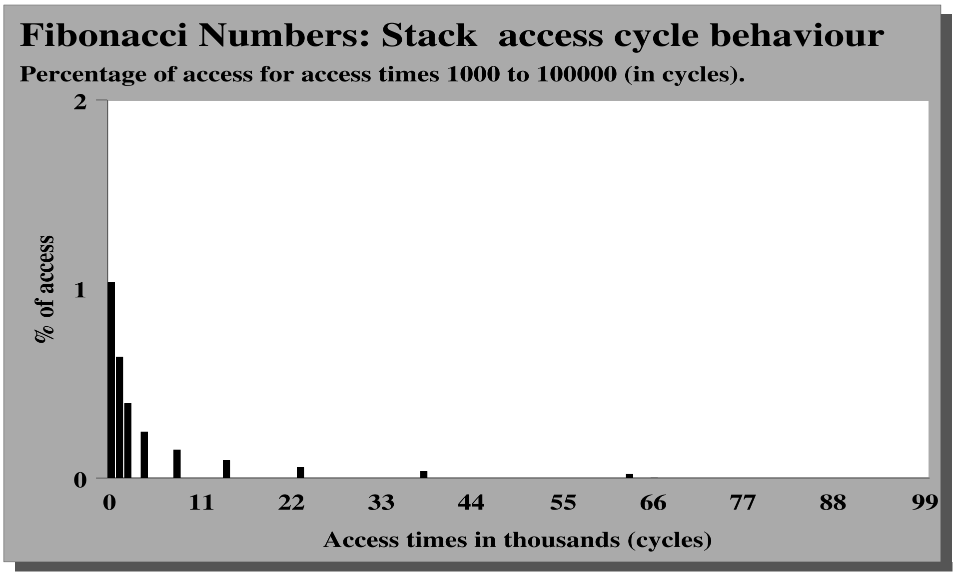

Figure 24:

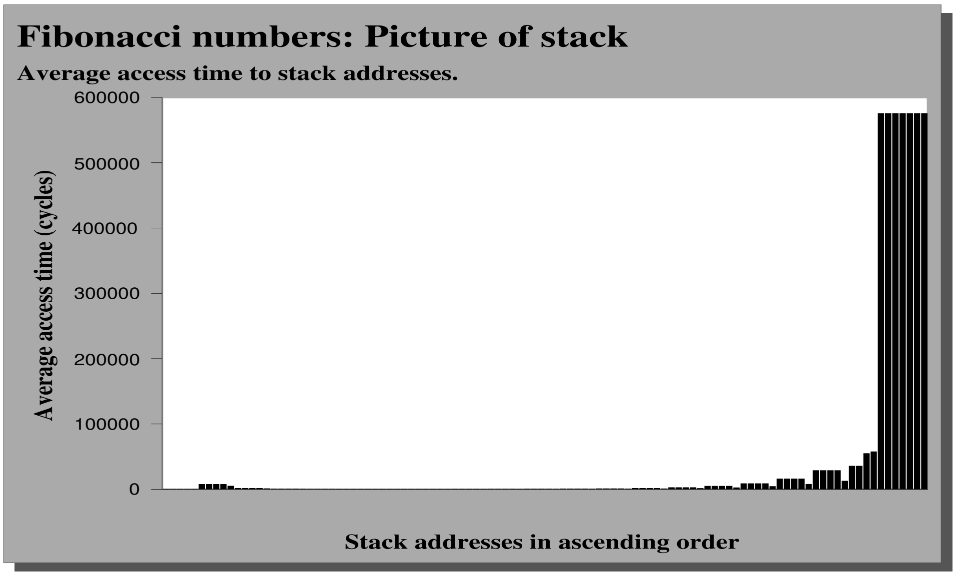

Figure 25:

The above programs do not involve any heap references. The following tests were used to study the profile results of programs using the heap space.

5. **Binary-search with Heap sort:** This program searches an array which is in the heap space, for numbers generated using a random number generator. The array is sorted first using heap sort, since binary search only works on a sorted array. The array is of size 100. Refer to appendix C for a source listing. Heap sort is an iterative implementation of the algorithm, and the principle used is as follows : The list to be sorted is rearranged to form a heap and then the top of the heap is removed repeatedly and transferred to its final proper position during the sort phase. The binary search assumes that the elements are sorted in ascending order. Figs. 26 to 32 show the results obtained from profling this test.

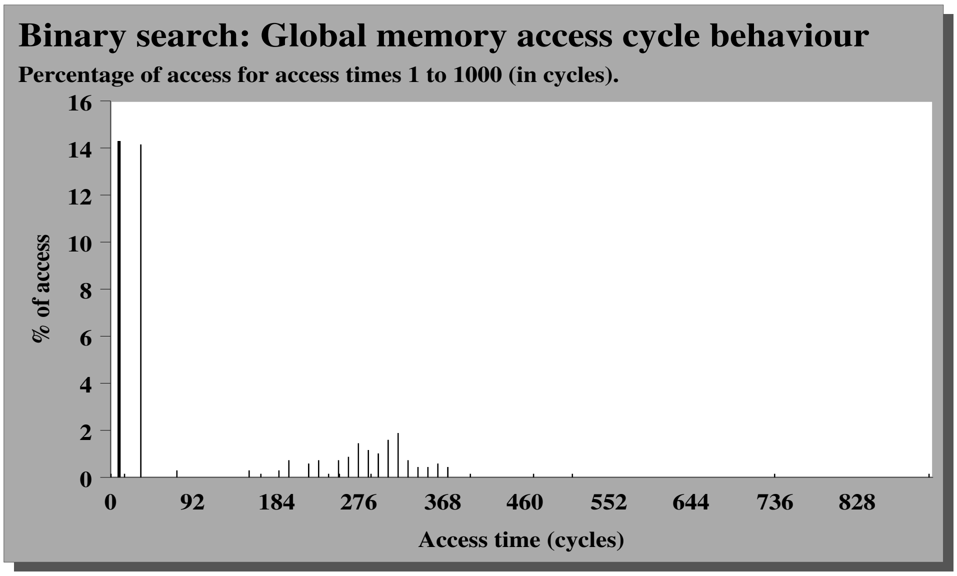

Figure 26:

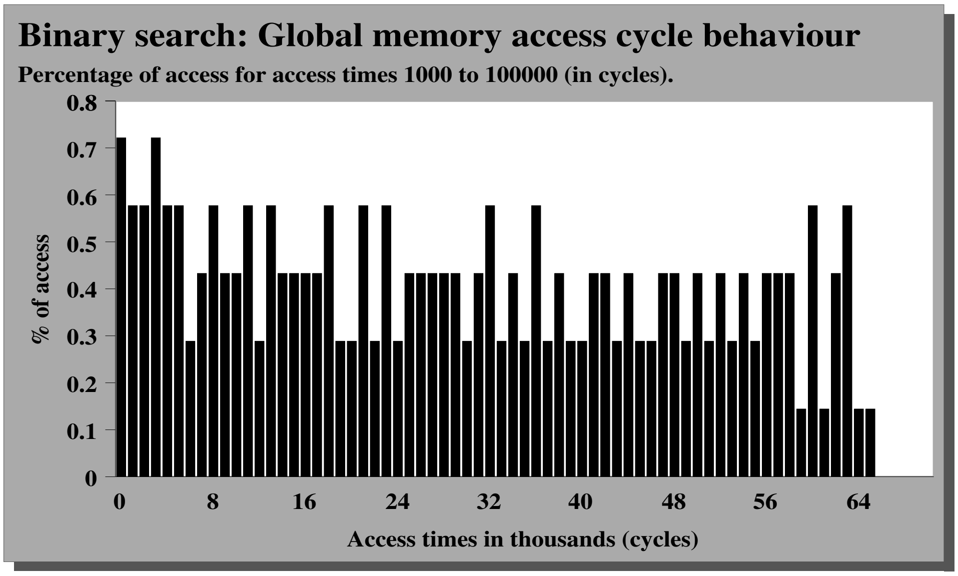

Figure 27:

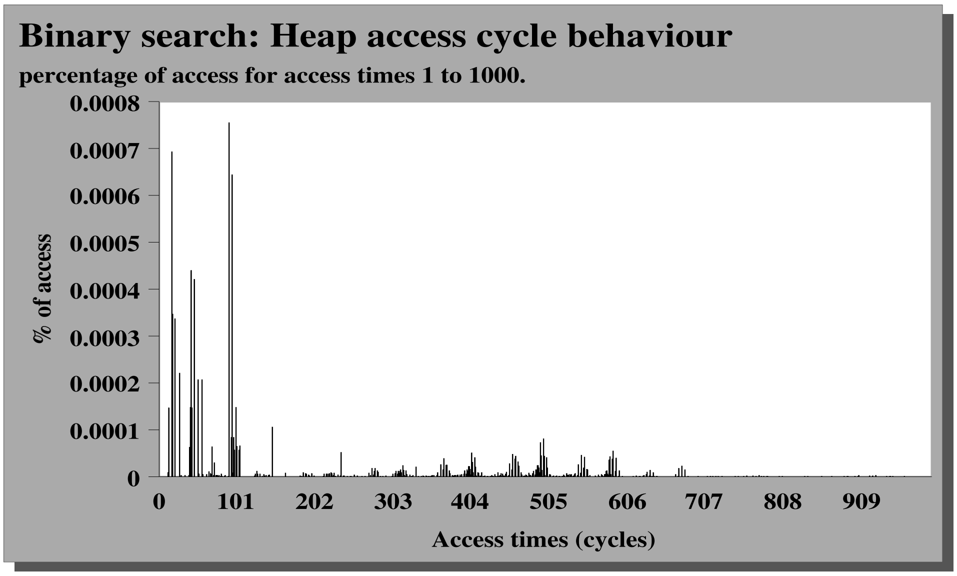

Figure 28:

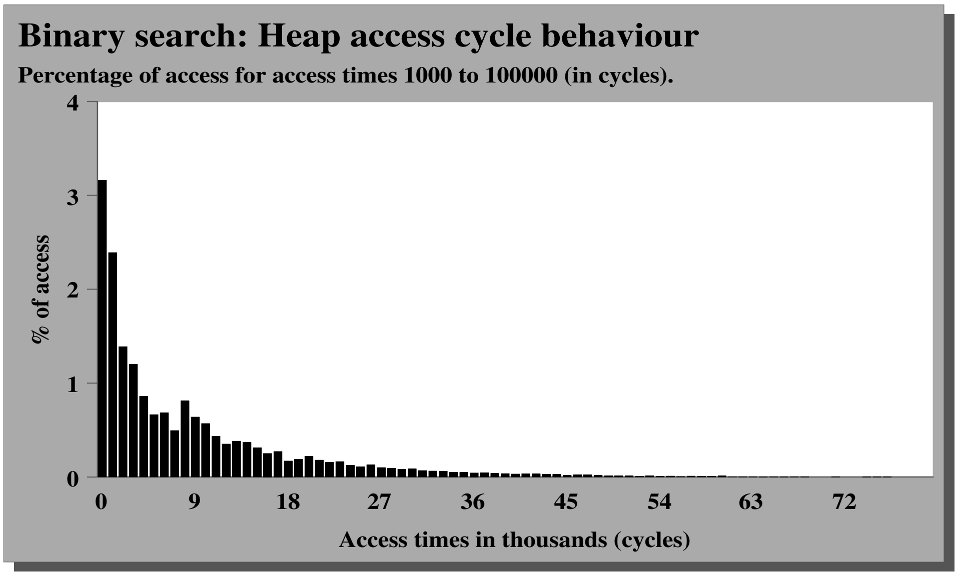

Figure 29:

6. **Balanced-binary tree creation and traversal :** This algorithm builds a balanced binary tree using a non-recursive implementation. Given a table of records that form a balanced binary tree, the algorithm searches for a given key. If it is not found, the new element is added at the appropriate place in the tree and the tree is rebalanced if necessary. Thus, the records can be retrieved in sorted order by traversing the tree using in-order traversal. The completed tree has 200 elements. The source listing for the algorithm is given in appendix C. Figs. 33 to 39 show the results of profiling this test.

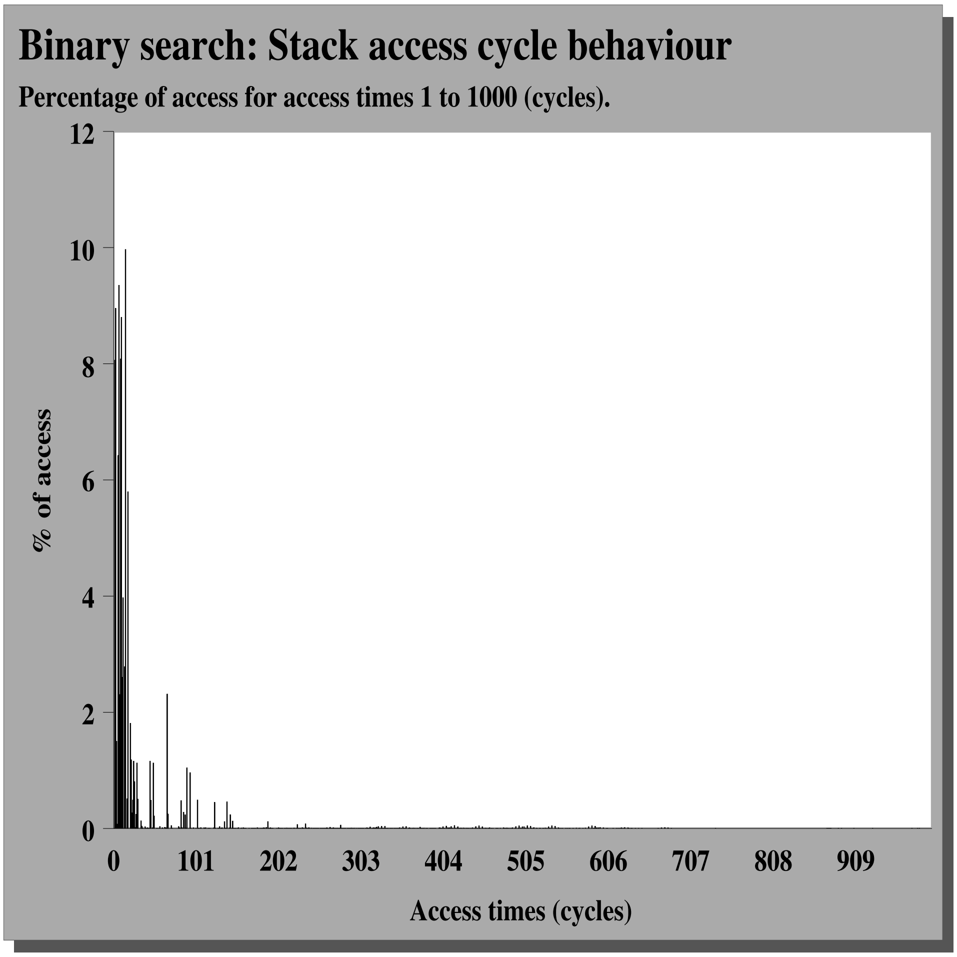
Binary search: Stack access cycle behaviour
Percentage of access for access times 1 to 1000 (cycles).
12
10
8
6
4
2
0
% of access
0
101
202
303
404
505
606
707
808
909
Access times (cycles)

Figure 30:

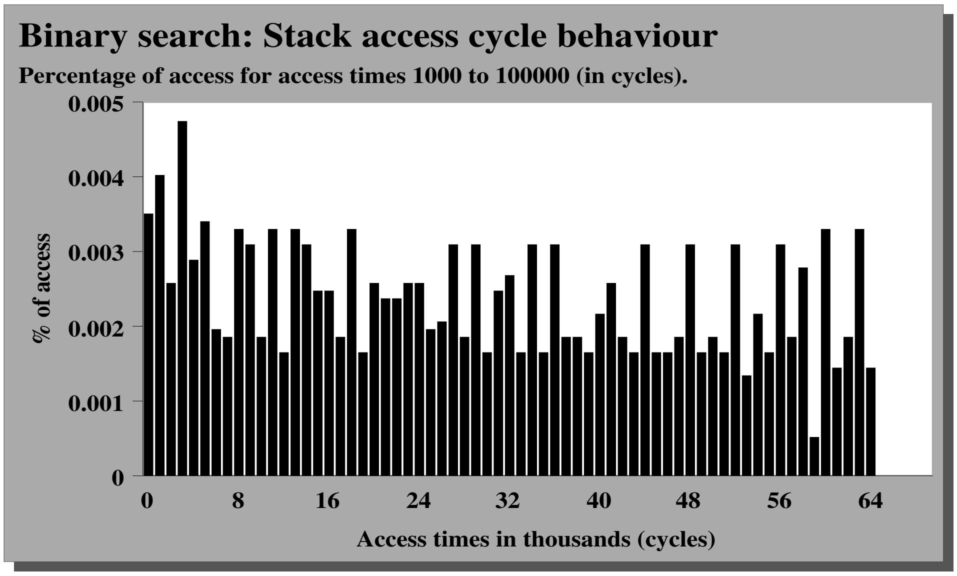

Figure 31:

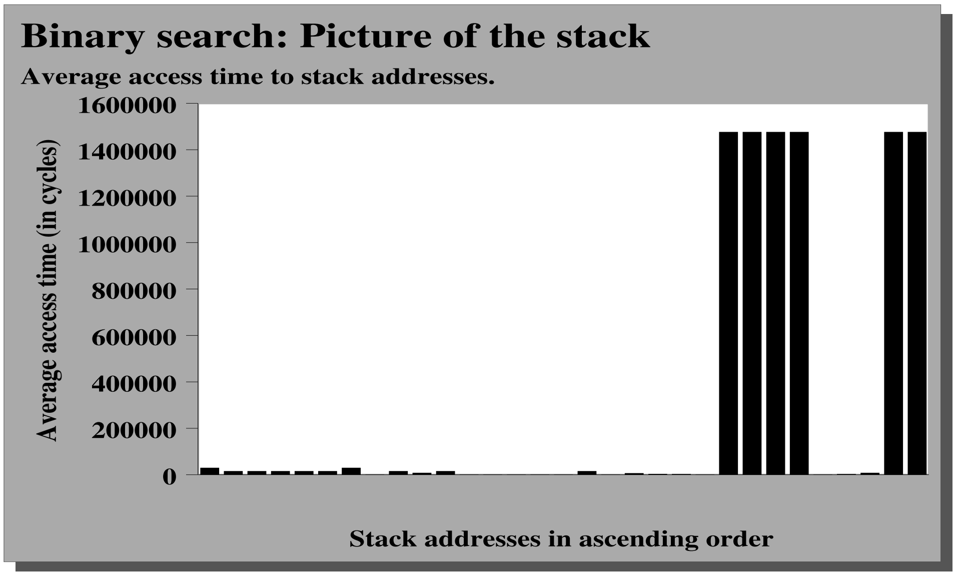

Figure 32:

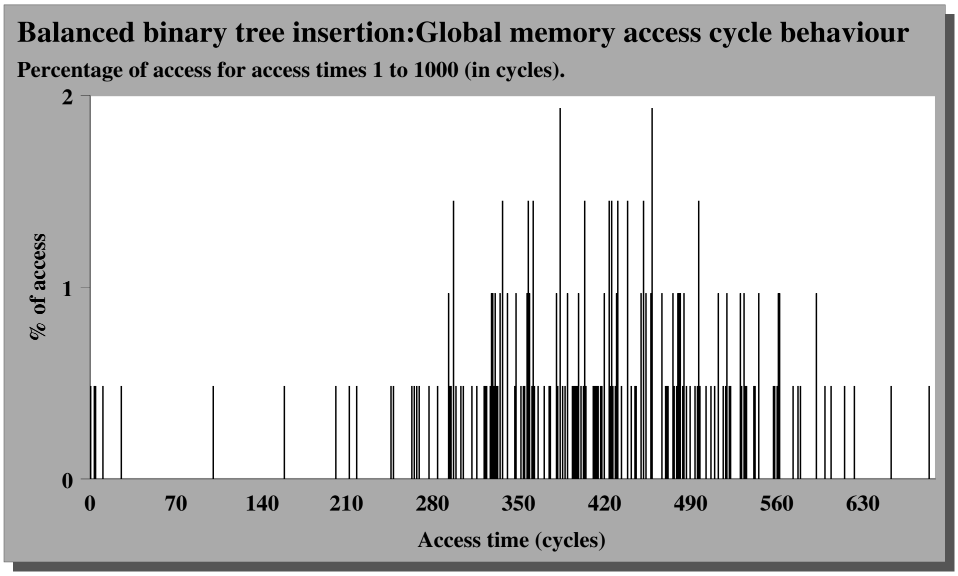

Figure 33:

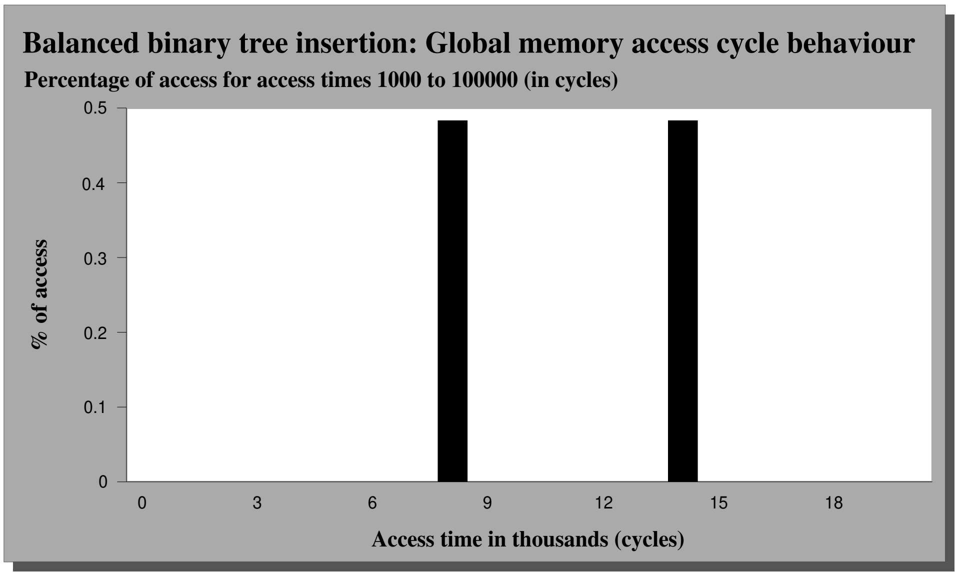

Figure 34:

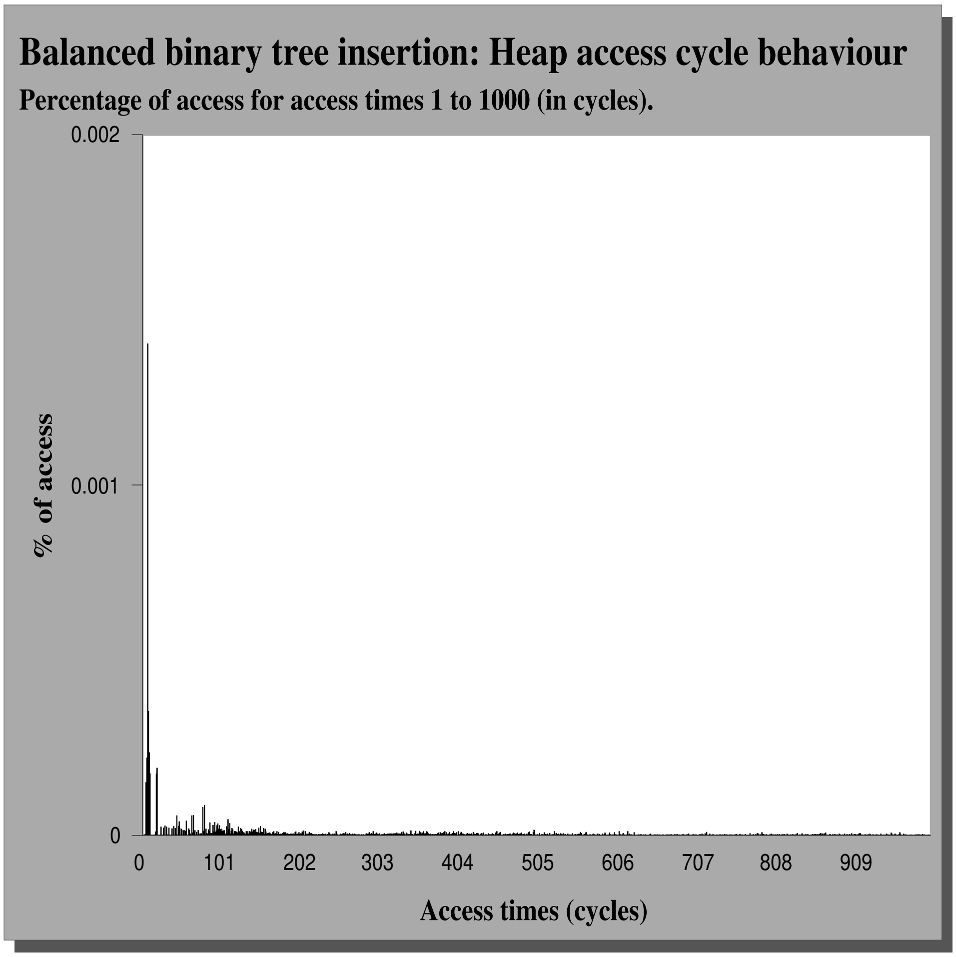
Balanced binary tree insertion: Heap access cycle behaviour
Percentage of access for access times 1 to 1000 (in cycles).
0.002
0.001
0
% of access
0
101
202
303
404
505
606
707
808
909
Access times (cycles)

Figure 35:

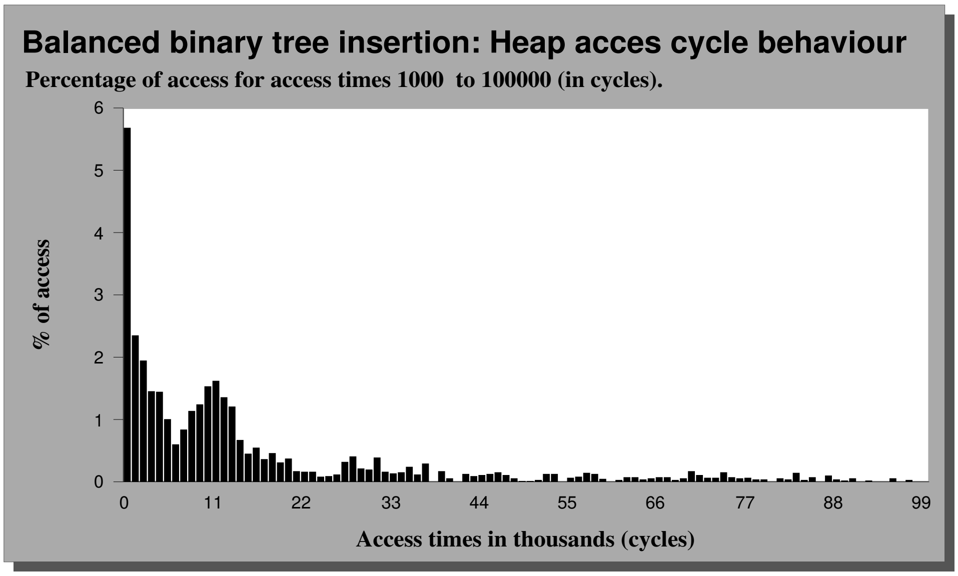

Figure 36:

7. **A simple word processor:** The main purpose of this program is to get the profile results of a linked list creation and traversal. This program reads in a text from a file and separates the words and lines. It creates a linked list for each line with each element of the list holding one word of the line. All such lines of the text are linked together, forming several linked lists. The algorithm then traverses the list line by line to print the text. See appendix C for the listing. Figs. 40 to 44 show the results of profiling this test with **DLXProf**.

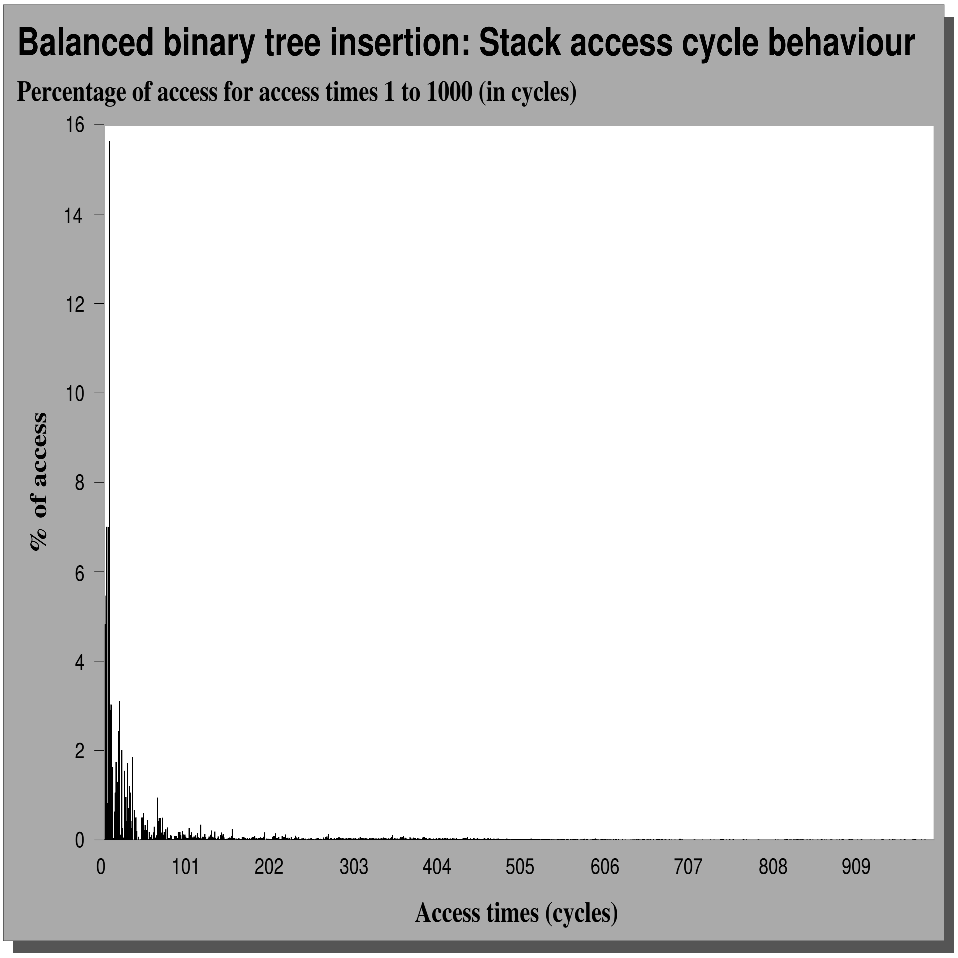
Balanced binary tree insertion: Stack access cycle behaviour
Percentage of access for access times 1 to 1000 (in cycles)
% of access
16
14
12
10
8
6
4
2
0
0
101
202
303
404
505
606
707
808
909
Access times (cycles)

Figure 37:

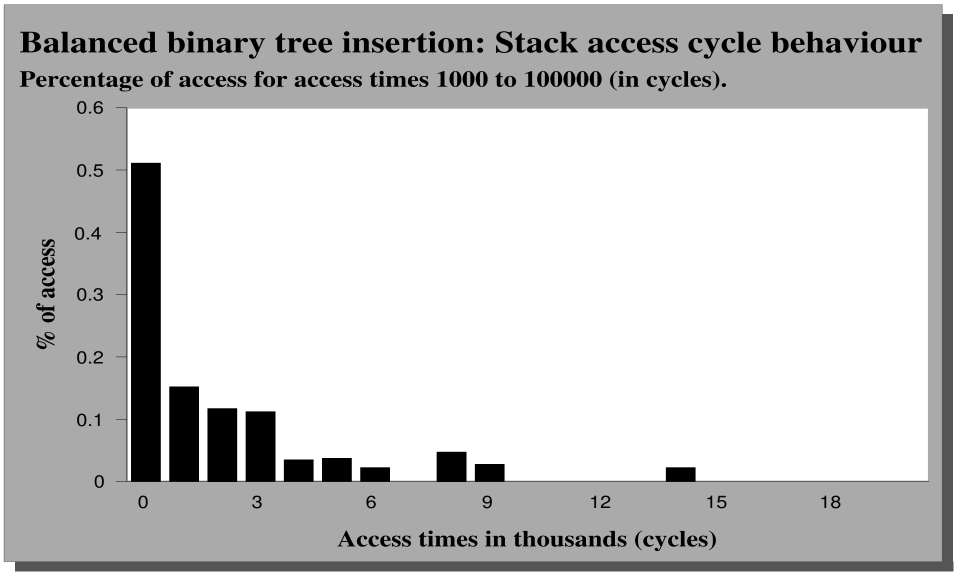

Figure 38:

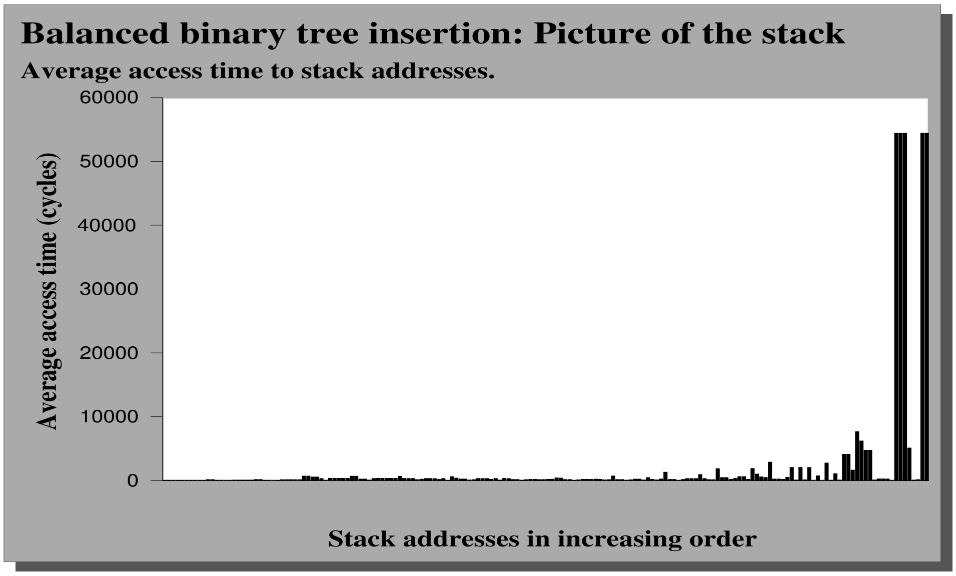

Figure 39:

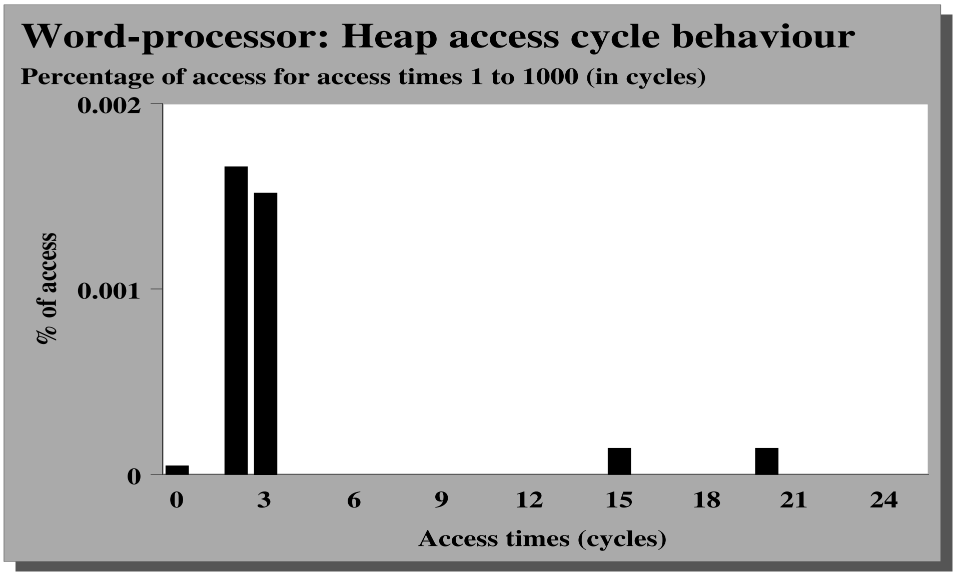

Figure 40:

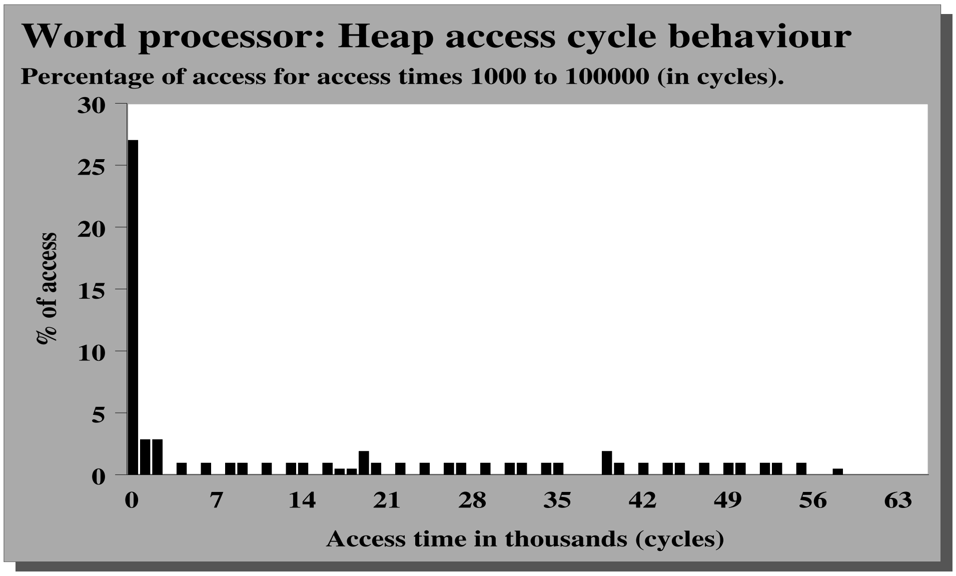

Figure 41:

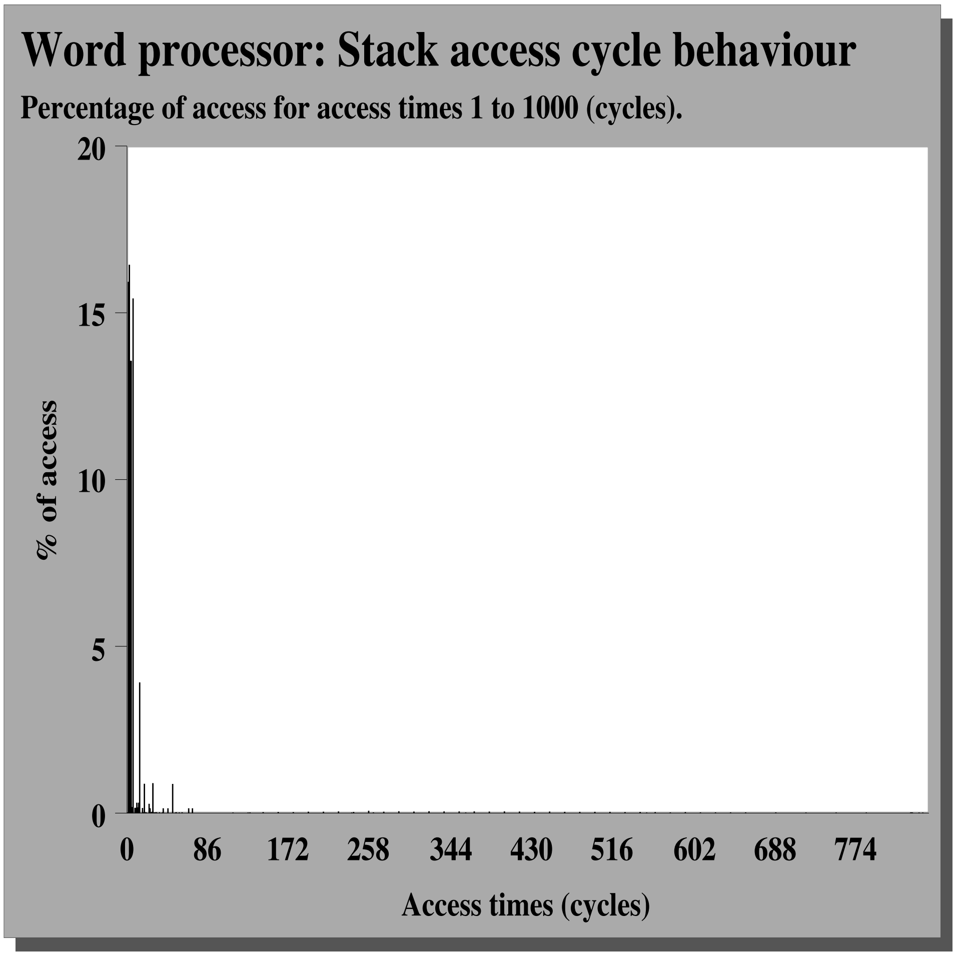
Word processor: Stack access cycle behaviour
Percentage of access for access times 1 to 1000 (cycles).
20
15
10
5
0
% of access
0
86
172
258
344
430
516
602
688
774
Access times (cycles)

Figure 42:

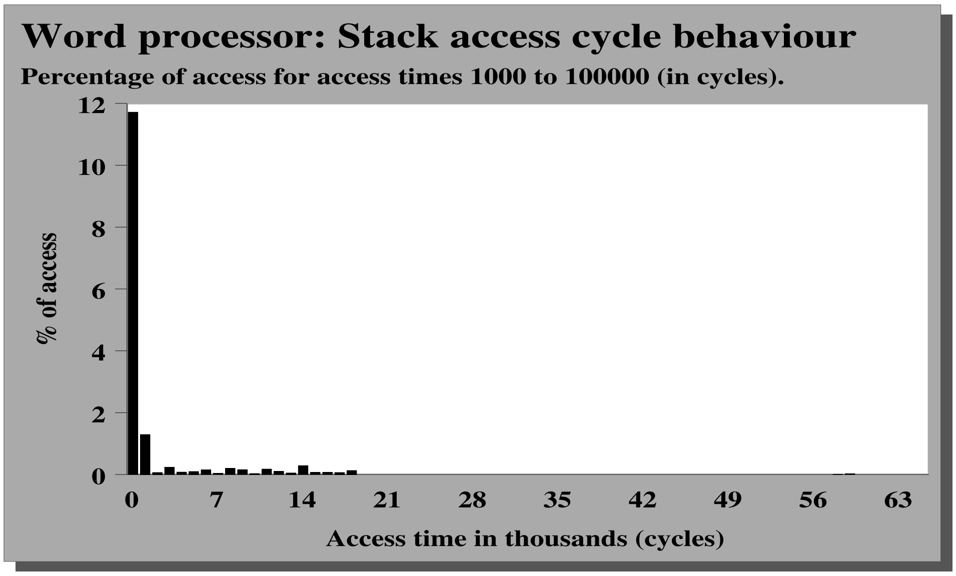

Figure 43:

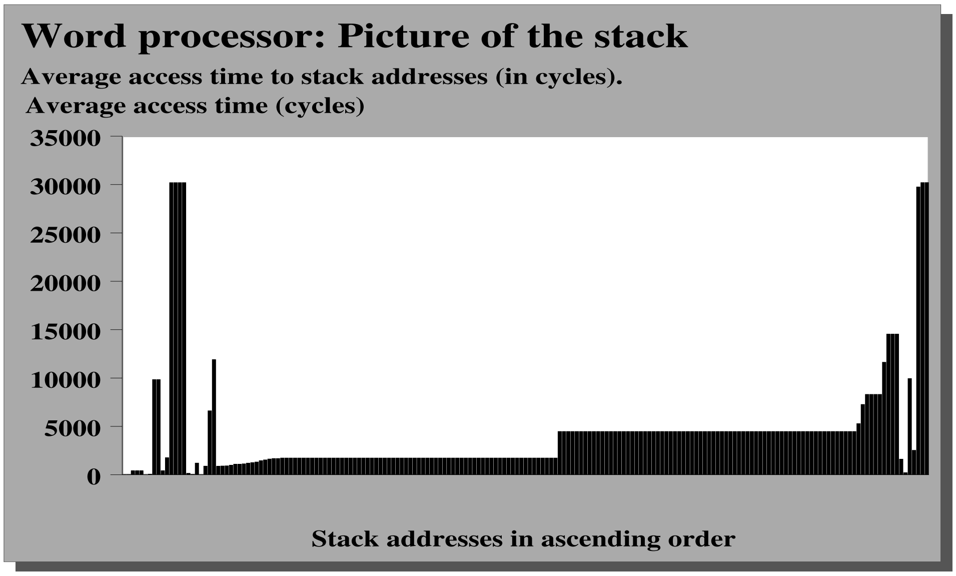

Figure 44:

5.2 Observations

1. First, let us discuss the profile results obtained using **Tomcatv** and the **Bubble Sort** tests. These tests primarily show the behavior of programs with array manipulations. The array in both these tests resides in the global address space described in Fig. 4.

   From Figs. 6 to 19, the following observations can be made. There is a lot of stack usage due to the local variables and function calls. Array elements sometimes reside on the stack when they are passed as parameters to functions. Since the array space is fixed during the load phase of the simulator, the stack need not keep track of where the array is resident at run time. From Fig. 45 we can see that the top of the stack is accessed more than the other parts because the variables of the function that does the actual bubble sort reside on the top. The last bar in Fig. 45 represents the function **main**, which calls other functions. Hence this gives a clear picture of the stack.

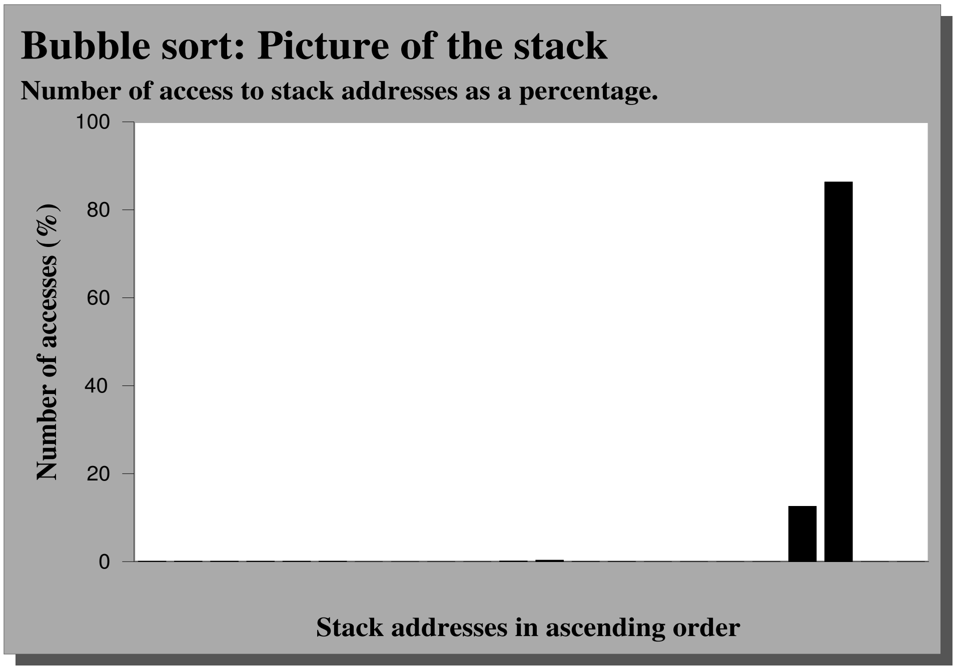

Figure 45:

   From the graphs for the cycle behavior for **Tomcatv** we can see that for more than 95% of the accesses, whether they are to stack or to global addresses, very few accesses are more than a 1000 cycles apart. On an average most accesses are less than 200 cycles apart, and very few greater than that. This is an important observation that could be useful in studying cache behavior. If addresses that are accessed most frequently are missed in the cache, the penalty would be more. Hence the performance can be improved by keeping these addresses in the cache at all times. In the case of **Tomcatv** and **Bubble Sort**, most of the accesses are to the array. Hence the following paragraph discusses how caching can be useful in programs with a lot of array references.

   From the graphs for **Tomcatv** and **Bubble Sort** we find that most global memory accesses are

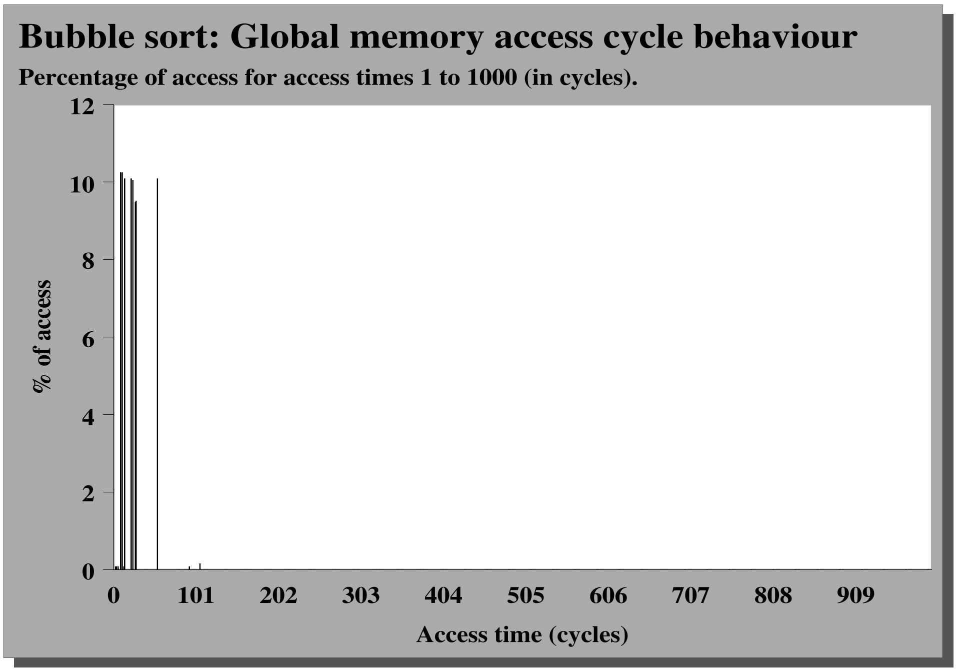

Figure 46:

to the array, and they are very frequent. For instance, from Fig. 46 we find that most accesses to global memory are less than 200 cycles apart. Also, from the Fig. 47 we find that the access time to all global addresses is the same, since there are loops that go through one element of the array at a time. So, each element gets accessed the same number of times as the others. Hence in programs where array manipulations form a major part of the computation such as these, a lot of time will be spent in accessing global memory. Accessing remote memory in this fashion is expensive. So it would be useful to keep the array in the cache, in order to reduce accesses to the main memory, cache misses, and the execution time of the program.

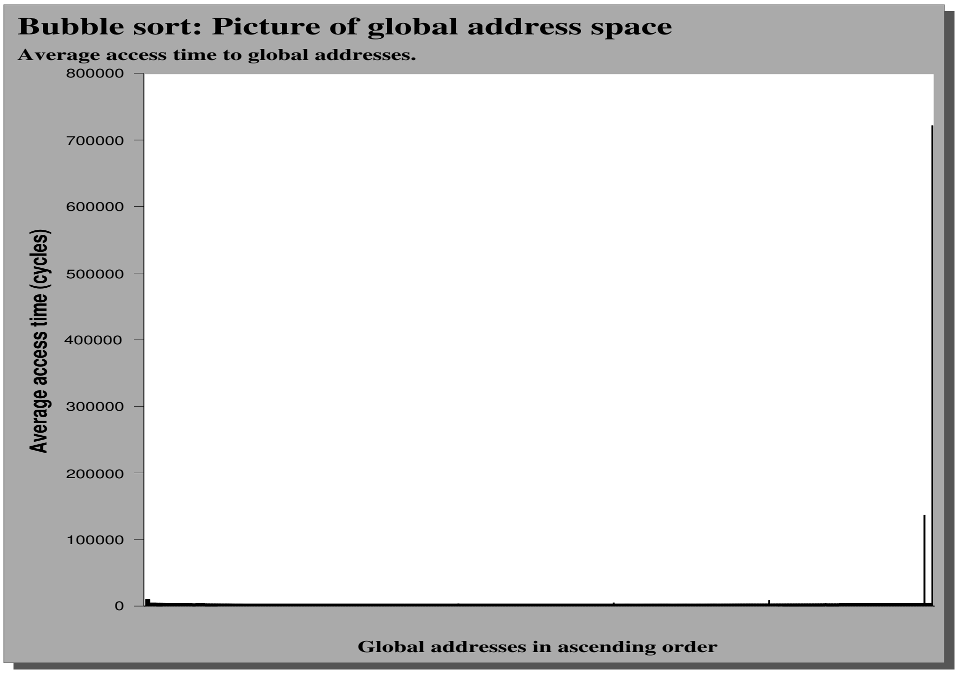

Figure 47:

2. Next we can take a look at the programs that were used to study what happens when there are recursive routines in a program. Both **Towers of Hanoi** and **Fibonacci Numbers** programs have a main routine that calls a recursive routine, but the difference between the two is that in the former, there are no global or heap memory accesses. Every thing works on the stack, and in the latter, there is some global memory access. From the graphs for both these tests, we can see that the picture of the stack shows a very systematic behavior. This is because of the recursive calls. The stack grows during each call of the procedure to itself and shrinks when it rewinds the recursion. This can be seen from the Fig. 48 where stack accesses increase for each call of the procedure and decrease for each return.

   From Fig. 49 we can see that the maximum number of references are to the top of the stack, and the number decreases systematically down to the bottom. These two graphs have different shapes even though they both depict recursion. This is because in the former, the recursive call to the function is in a loop, and in the latter it is a single recursive call. Hence in the latter, the stack continually grows till the function returns, whereas in the former the stack grows and shrinks each time the recursive function is called from the loop. Since all computations take place in the stack for the **Towers of hanoi** problem, we can see that the average access time to the stack never exceeds 1000 cycles. But the **Fibonacci numbers** problem also has some global memory activity once in a while, and so we see some of the addresses being accessed more than 1000 cycles apart which constitute less than 2 percent of the accesses.

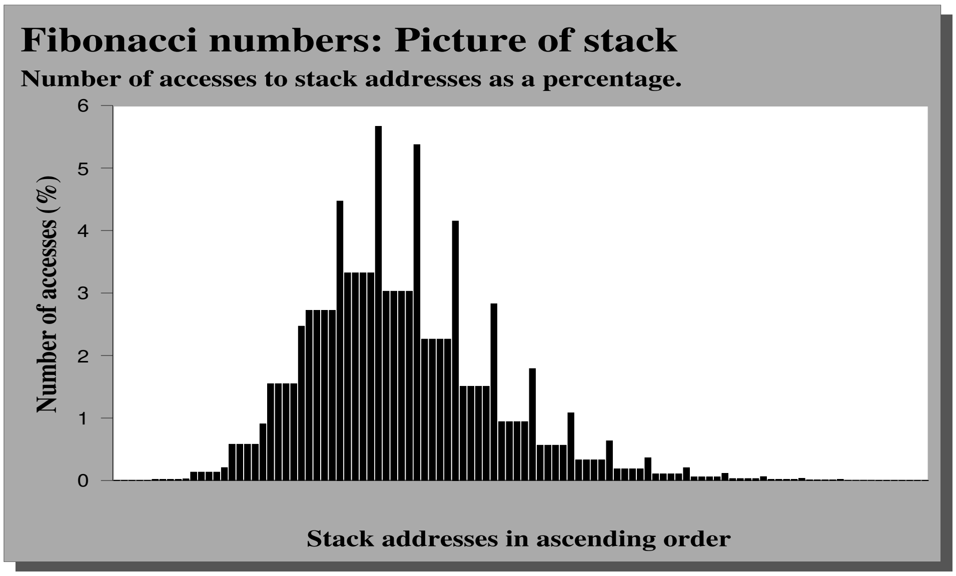

Figure 48:

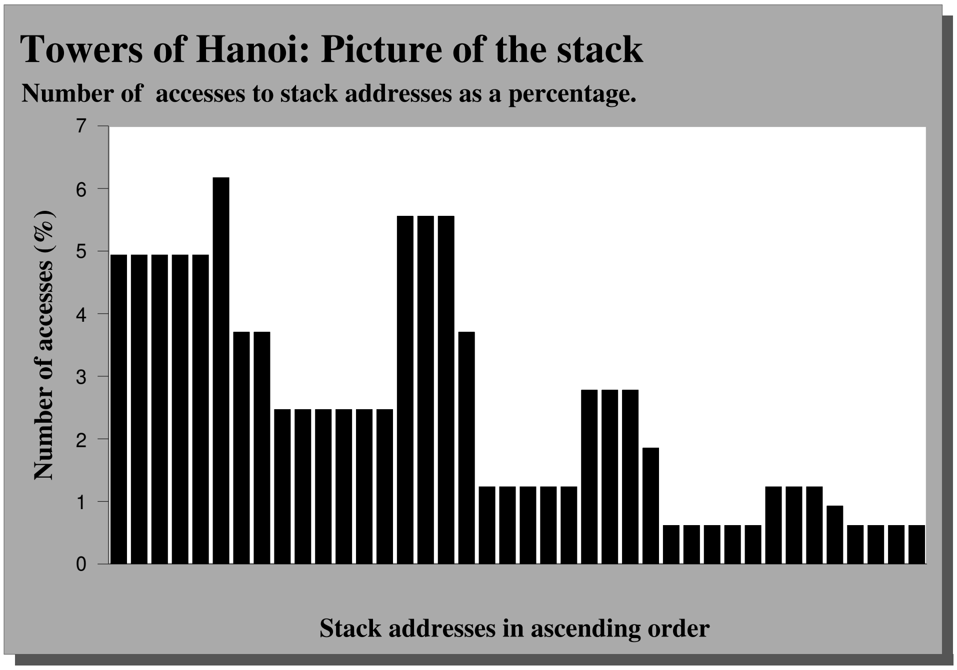

Figure 49:

3. Finally let us consider the programs that have heap access. These are

   - the **balanced binary tree insertion** program which shows the creation of a balanced binary tree to insert a set of random numbers in a sorted fashion and the traversal of the tree,
   - **a simple word processor** which shows linked list creation and traversal
   - and finally a **binary search algorithm with heap sort** program.

   From the graphs for these tests (Figs. 26 to 44), the following observations can be made. When heap space is used by doing a memory allocation in the program, it is seen that there is less reuse on the heap. There are a large number of heap locations that are accessed, especially when memory is not freed, as is the case here. The results obtained show that stack accesses are more numerous than heap accesses, but more locations on the stack are reused. Typically, thousands of heap locations could be used even for fairly small programs, whereas only a few hundred stack addresses may be accessed.

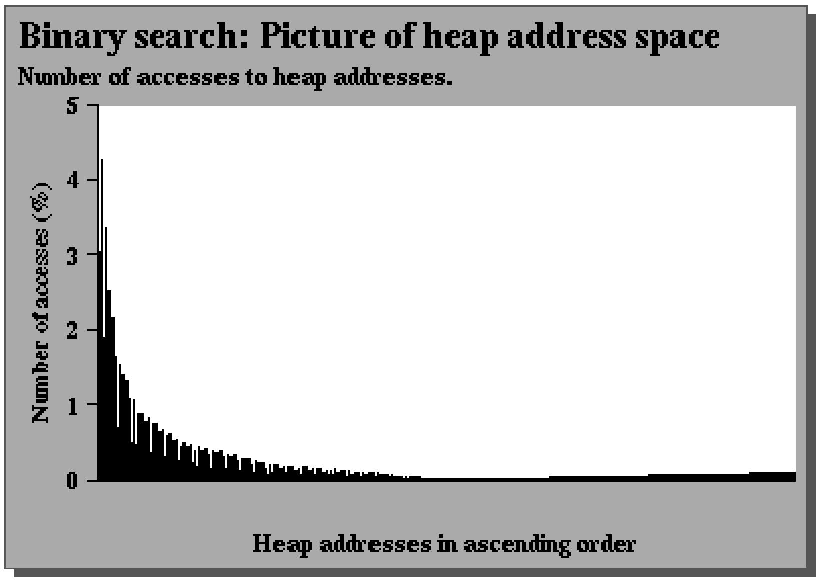

Figure 50:

For a program such as the **binary search with heap sort** which uses a linear array that resides on the heap, lower heap addresses (the beginning of the heap space) are used more than the higher addresses. In the case of the binary search, this is expected because there is a single memory allocation done for the size of the array and the simulator may have put this in the beginning of the heap. Access to these locations is more as seen from Fig. 50.

Since binary search and heap sort are done, we can see a curve where the percentage of accesses to some locations decreases. It is difficult to say whether this decrease was caused by the **heap sort** routine that sorted the array, or if it was by the **binary search** routine, since both these routines

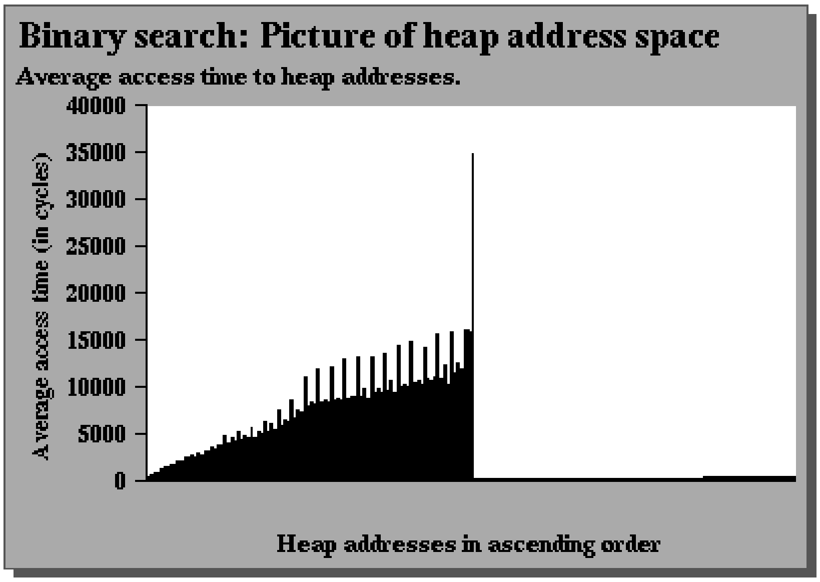

Figure 51:

used the same array in the heap. It could also be because of the nature of the test data given to the binary search routine, where all the search keys were in the first half of the heap. There are a few temporary and local variables used during the heap sort phase in the program and these are given space in higher heap locations. This is the fairly flat collection of undivided lines we see in Fig. 50 in the higher heap addresses. Fig. 51 shows the average access cycles to heap locations, and we can see that accesses took less cycles between each, to locations that were used often, as would be expected.

The word processor test is a program that creates a linked list to store a text that is given to it as explained earlier. This test is different from the binary search test discussed above because in the latter, there is one single memory allocation done in the beginning to an array and all the activity takes place with it. In the word processor test however, there are many memory allocations done using the function “malloc”, and the number that is done varies with the input text. Hence it would be interesting to study the behavior of such random memory allocation programs with **DLXProf**. The **balanced binary tree creation and insertion** is also one such test where a tree is created dynamically, and so these two tests can be compared in their behavior. One interesting observation made from the picture of the heap space (Fig. 52) is that equal access is made to all addresses on the heap. We can see that the number of accesses to all heap addresses are almost the same, around 1.5 percent. This is because, in this program, all the elements of the linked list are accessed uniformly. While the lists are being created, all heap locations are accessed, and when the lists are traversed to process the text, all the elements of all the linked lists (there are many rows of them, one for each line of input text as mentioned earlier) are traversed the same number of times.

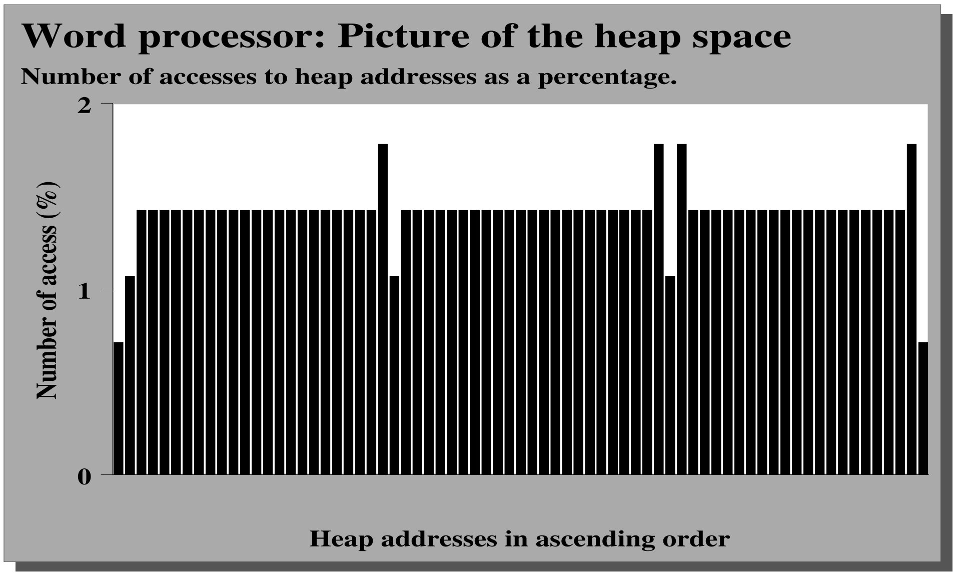

Figure 52:

In the balanced binary tree creation and insertion program (Fig. 53) however, even though the whole tree is accessed there is more access towards the beginning of the heap than the higher locations. This is because the tree is more often traversed in the higher parts to check the keys of the nodes to find the right place to insert the next element, and these addresses are usually in the lower addresses of the heap. Only when the right subtree is found, the subtree may be traversed to the depth needed to insert the element. It would not be always be necessary to go up to the leaves of the tree to insert the elements, whereas the root and other higher parts of the tree are always traversed to insert new nodes or look for some, once the tree has been built to a certain depth. The root is always accessed at least once. Hence the beginning of the heap space has the highest percentage of accesses. One other interesting observation is that for this kind of dynamic memory allocation programs, where the allocation is done in pieces rather than as a large block, the average access cycles to heap addresses (Figs. 54 and 55) is **not** the inverse of the number of accesses to those locations. The two graphs show a gradual decrease in the average access cycles with increasing heap addresses. This means that the number of cycles that elapse between accesses to higher heap addresses are much less than the cycles that elapse between accesses to lower heap addresses. Fig. 54 shows this kind of behavior.

Let us say that the next node to be inserted is at the bottom of the tree. Then the root is traversed first, let us say, at cycle 0, and then the other nodes are visited, and let us say it takes about 100 cycles to get to the spot where the memory allocation is to be done for the node to be inserted. This allocation is expensive ("mallocs" are expensive) and we could assume it takes about 200 cycles to complete it. Then, the next time the root is accessed, it has been at least 300 cycles since the last time it was accessed. So when we go back up the tree, the root is the last to be accessed, and since the root and higher parts of the tree are in the lower addresses of the heap, there is a greater

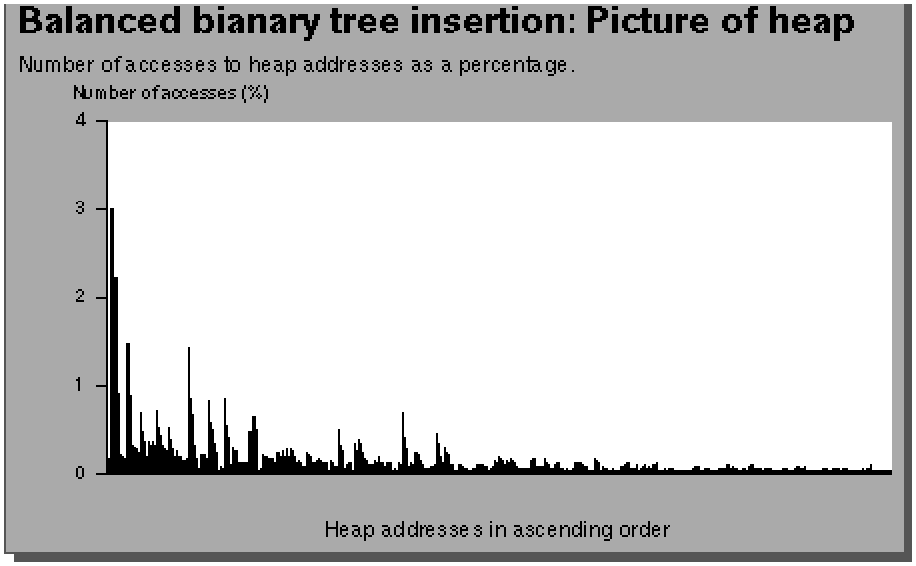

Figure 53:

difference in the cycles elapsed between accesses to them. Now considering the Fig. 55, there is a fall in the cycles between accesses as the heap addresses increase. This test is a linear linked list creation and traversal and we should expect a linear behavior like with arrays. To explain this variation in behavior, it is necessary to take a look again at what the program does. It reads in a text from a file and separates the words from it. It then creates one element of the linked list for each word. Once an element has been created, it is never traversed in a random fashion again. In other words, all elements (and so all heap locations) are accessed the same number of times. Hence from Fig. 52 we can see that the curve is almost flat, showing equal percentage of access to all locations. The small spikes are accesses to the heads of each row of the linked list more times than the other elements of the same list. This behavior is as expected. Now coming back to Fig. 55, let us assume for the sake of simplicity that there is only one row of text, so there is only one linked list. Let us assume the first element of the linked list is allocated memory at cycle 0, and the next one at cycle 100 (because "mallocs" are expensive), and the third at cycle 200 and the fourth at 300. After the last element has been allocated memory, the list is printed by traversing the elements one at a time. This traversal does not take as many cycles as a "malloc". So the first element is traversed at cycle 310, the second at 320, the third at 330 and the fourth at 340. Now if we calculate the difference in cycles for each address, the first element was traversed first at cycle 0 and next at cycle 310 so the number of cycles elapsed between two consecutive accesses to it is 310, the second was traversed first at cycle 100 and next at cycle 320, so the difference is 220, and calculating similarly for the third, the difference is 70 and for the fourth it is 30. Hence we see that there is a decrease in the cycles from the first element to the last, and this is why as we traverse

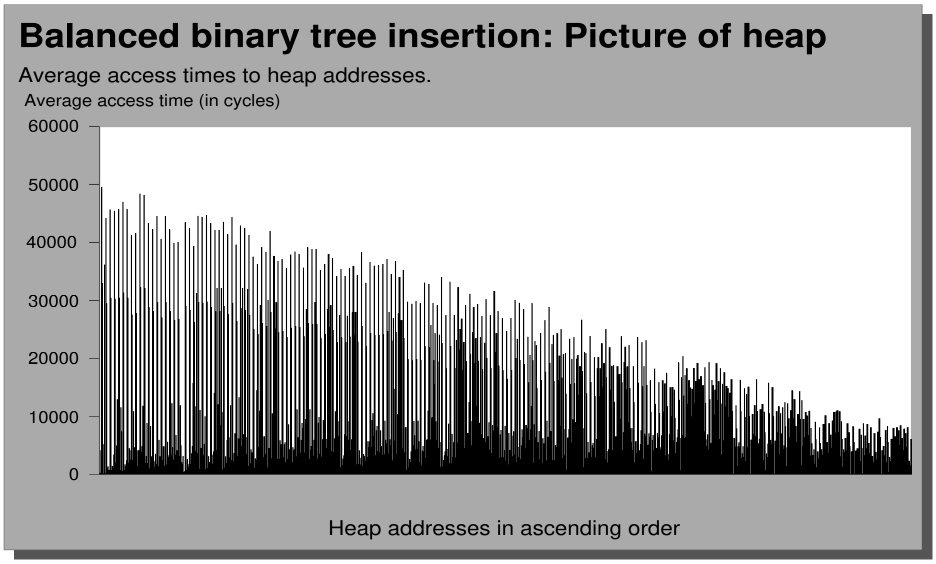

Figure 54:

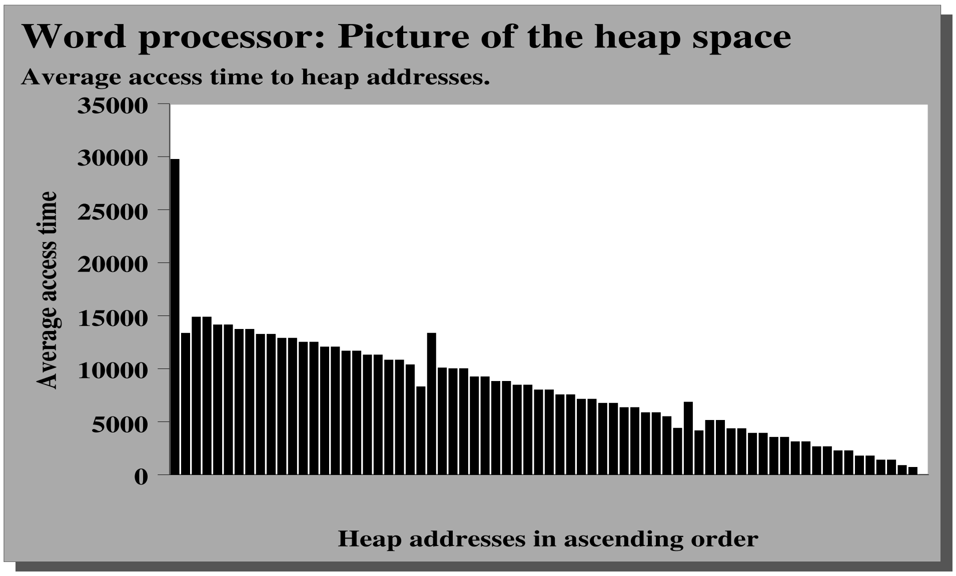

Figure 55:

the heap from the lower to the higher addresses, we find a decrease in the access cycles. It would be interesting to study this behavior in the cache, to see which addresses should reside in the cache longer to reduce the misses, the higher heap addresses, or the lower. Some larger programs can be studied in a similar manner, for a better understanding.

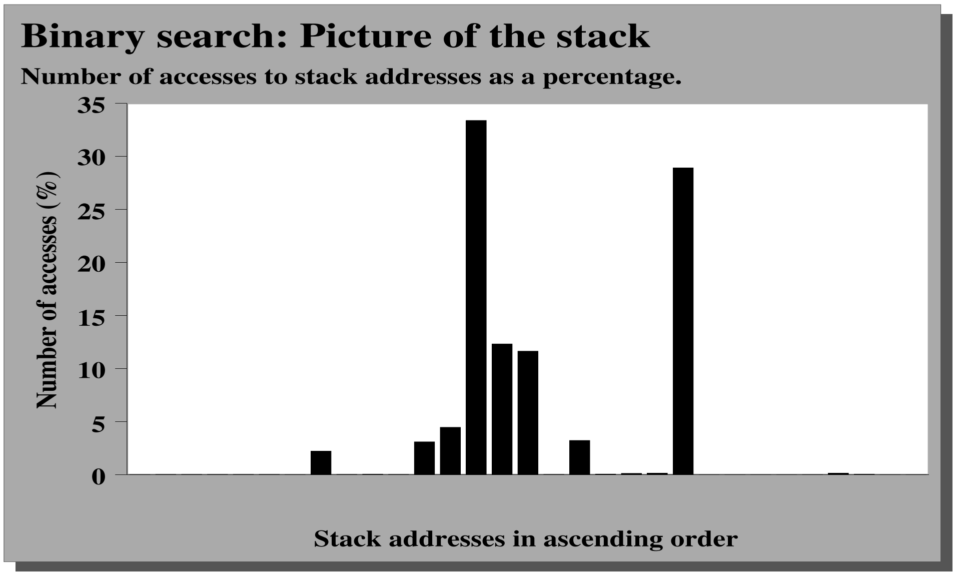

Figure 56:

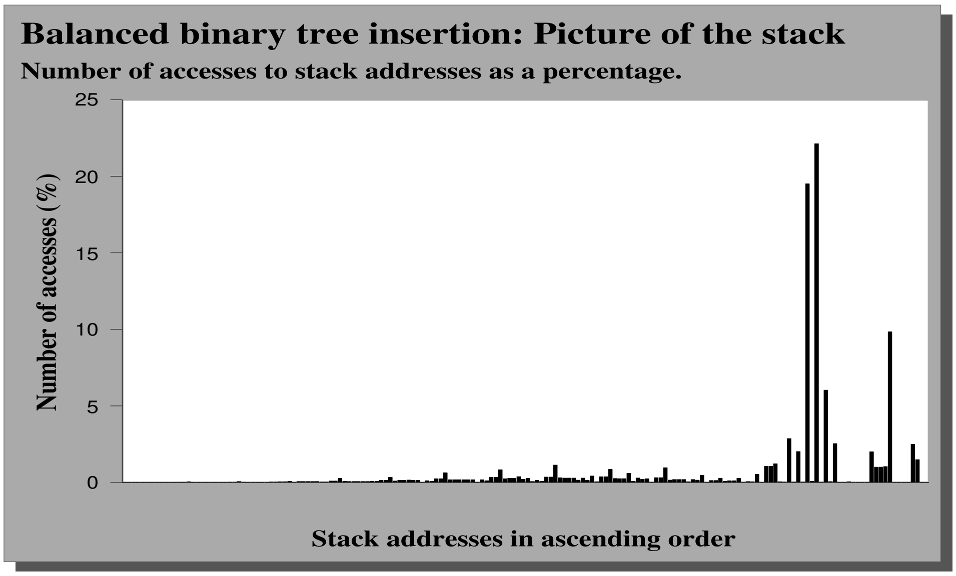

Figure 57:

Pictures of the stack for all the tree tests (Figs. 56, 57 and 58) show that there are more accesses closer to the top of the stack, since the functions that do most of the important work are invoked last in all the

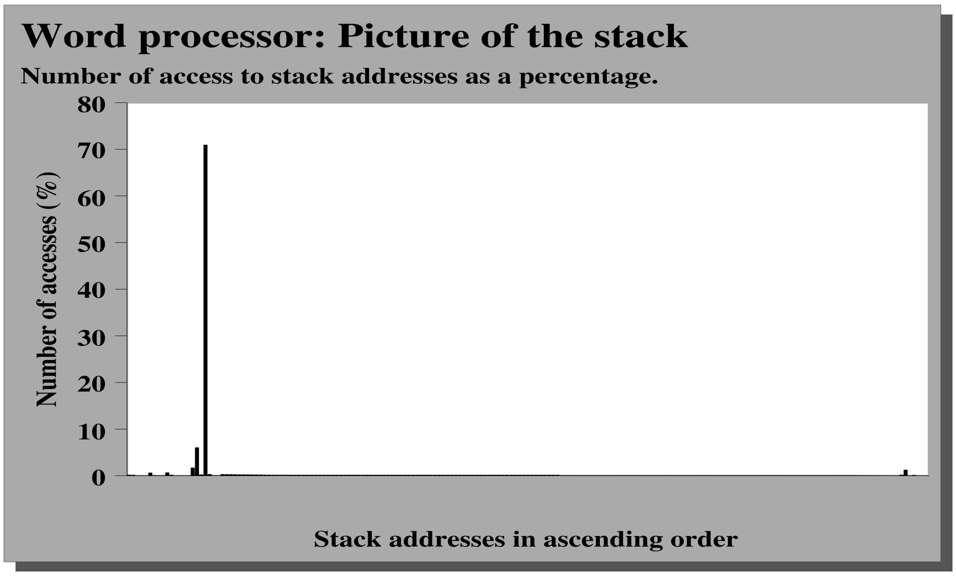

Figure 58:

three tests. In all the stack access pictures of all the tests, we can see that in the lower addresses of the stack there are very few accesses. These do not actually form part of the main algorithm of the programs that are being tested. They pertain to some system calls that always take place during execution. Also, in programs that show memory allocation to the heap, we can find that more cycles elapse between two accesses to a location. We find typically that most accesses are at least a 1000 cycles apart. This is true whether it is an access to the stack or to the global memory space. For instance, the Fig. 22 shows that for the **fibonacci numbers** test, 70% of accesses were spaced within 0 to 50 cycles, and only 10% of them fell above 1000 cycles. Similarly Fig. 46 shows that for the **Bubble Sort** test, more than 50% of the accesses were within 0 to 101 cycles apart. On the other hand, when we compare the figures for the word processor test, Fig. 40 shows that only about 0.005% of the accesses were within 3 cycles. more than 40% of them were around 1000 cycles (Fig. 41 and 43). This clearly shows that heap accesses, and especially memory allocations waste a lot of machine cycles.

Not only are heap accesses expensive, so are global memory accesses. Fig. 20 shows that there are no accesses that are more than 700 cycles apart. But in all the other tests, there are invariably at least some accesses that are separated by more than 1000 cycles, even if there are no heap accesses. This is because, all these tests have some global memory access. Hence another useful conclusion one could arrive at is to try and reduce remote accesses, or to try and use caches to trap the remote accesses in the cache itself. There is an ongoing research that is being done for multi-threaded architectures to cache remote accesses, which will be discussed in a later section of this report.

6 Conclusion

We have seen the important role played by performance analysis tools, and the motivation, design and implementation of one such tool **DLXProf** was discussed. There were two phases to this project. The first was at the **GNU C** compiler level where information was collected at the **AST** and **RTL** levels to produce a modified assembly code with annotations containing some extra information needed for profiling memory references. The second phase described two parts of **DLXProf**, one that would be used to collect information from target programs about the number of pointer, array, local and global variable information using the modified assembly code, and the other that collects information about memory references to the stack, heap and global memory. The reasons for the implementation at the **GNU C** compiler level being incomplete at the time of submission of this report were discussed. Although, results were successfully obtained using the second profiler, and attempt was made to see how useful **DLXProf** actually is in studying the behavior of programs in caches, since this was the long term goal of the project. It was found from even the few small tests that were conducted that the tool is useful for this purpose. For example for programs with arrays in global memory, there are a lot of global memory access and having the array in cache will improve the performance, and reduce cache misses. Behavior of programs with heap accesses was also studied, and a uniform access to the heap space was found when the program allocated large chunks of memory at a time in the heap. Hence it would be useful to have this part of the heap in the cache, since a lot of it was accessed. An interesting behavior was detected with programs that did memory allocation in the heap space in smaller quantities at a time with respect to the memory access cycle that elapse between accesses, and this behavior needs further analysis to see how it could affect the behavior of these programs in caches.

Memory access behavior studied using **DLXProf** could be useful in studying methods to improve memory latency with caches in multi-processor system. Research is being done to see if it is possible to cache remote memory accesses in multi-threaded systems locally, so that the number of remote accesses are reduced, thereby improving the overall memory latency and reducing memory traffic [17]. **DLXProf** has proved useful in this attempt to analyze program locality in single processor systems, especially for heap and remote memory accesses. These observations could prove as a basis for analyzing shared memory multiprocessor systems.

We can see from the growing importance of execution profiling, that it will become an indispensable component of future programming systems. One of the main problems with all the known profilers is that they are either intrusive to a certain extent, or they operate as separate threads as in the **Parasight** if they are to be totally non-intrusive. But the profiling approach that has been followed with **DLXProf** eliminates intrusion with the target programs, while being interactive at the same time due to the availability of the **DLX** simulator. Also, this technique gives some high-level information about the program at run-time, which is a novel approach in itself. When larger programs are run through the **DLX** simulator, **DLXProf** can be a very useful tool to study their run-time behavior and to further develop this project to study cache behavior.

7 Future Directions

DLXProf is not complete yet. There are several additions and enhancements that are to be done. Hence it is necessary to give some future directions here.

First, it is necessary to have a good user interface with the simulator for several reasons. The interface that comes with the simulator is a simple command line interface. This may be enough with just the simulator, but with tools like performance analysis tools added to it, this is inadequate. It is especially necessary to have an utility to plot graphs with the results of the performance analysis. Without graphs, the results obtained are just big files with numbers which cannot be understood easily. Hence a graphical user interface has been developed for **DLXProf** using the **NeXT** Interface Builder, which is a good user interface development package that exists on the **NeXT** workstations. For further details on the development and features of the interface, refer elsewhere [18]. The graph plotting package that has been included with the interface is the **GNU** Plot software. Another user interface that exists with the **DLX** simulator is the **xdlxsim** which is an interface developed with **X** windows at McGill University, by Russel Olsen of the department of computer science. But one disadvantage of these user interfaces is that the **GNU** Plot is the graph plotting software that has been interfaced here as with the interface on the **NeXT** interface builder, and this tool does not plot bar charts. Bar charts are some of the most expressive charts for performance analysis with **DLXProf**. The bar charts that have been used in this report to show the results obtained on programs with **DLXProf**, have been developed using the Lotus Improv package that exists on the **NeXT** machines. But this software has a limit to the number of points that can be plotted on a graph and the package is not very reliable when the graph has more than 800 points. Hence it is a very time consuming process to use it, and cannot be incorporated into other programs easily. As a possible solution to this problem, there is a package called "xgraph" which can be used with X windows, and can plot bar charts. This can be interfaced easily to programs and so it can be incorporated this with the "xdlxsim" if bar charts are needed.

The second addition that has to be done to **DLXProf** is that the part of the profiling that is incomplete at the **GCC** level has to be completed. For this, it is necessary to have a simpler intermediate representation than the Register Transfer Language (**RTL**) of the **GCC**. The implementation for capturing information from the Abstract Syntax tree has already been done. The captured information has to be passed down to the assembly level in some other way than through the **RTL**.

The third addition is that once the information has been passed down to the assembly code, when the simulator gets the file to execute, the extra information from it has to be made use of to do profiling. The simulator has already been modified to read in a modified assembly file with annotations. It can load the extra information into its memory array. It also does some profiling such as counting the number of memory references caused by pointer type references, or array references, or local or global references. But it could be made to collect more information about these references as desired.

References

[1] Agarwal A., Sites R.L., and Horowitz M. ATUM:Microcode. *Proceedings of 13th Annual International Symposium on Computer Architecture*, pages 119–127, June 1986.

[2] Levy A., Katwijk Van J., Pavlides G., and Tolsma F. A Support environmant for prototyping distributed systems. *Proceedings of the first International Conference on System Integration*, April 1990.

[3] Malony A. Cedar Performance Measurements. Technical report, CSRD Report No.579, Center for Supercomputing Research and development, University of Illinois, June 1986.

[4] Levy A.M. and Pavlides G. Simulation vs. Prototype Execution:A Case Study. *COMPEURO '90 Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering*, pages 428 – 436, May 1990.

[5] Stunkel Craig B. and Fuchs Kent W. TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation. *Performance Evaluation Review*, 17(1):70–78, May 1989.

[6] Mukerji Chandrika. Instruction Scheduling at the RTL Level. Technical report, McGill University School of Computer Science Technical Note 28, May 1991.

[7] Clark D.W. Cache Performance in the VAX-11/780. *ACM Transactions on Computer Systems*, 1:24–37, February 1983.

[8] Allen Frances, Burke Michael, Charles Philippe, Cytron Ron, and Ferrante Jeanne. An overview of the ptran analysis system for multiprocessing. *Proceedings of the 1987 International Conference on Supercomputing*, 1987.

[9] MIPS Computer Systems Inc. *MIPS Language Programmer's Guide*, 1988.

[10] Dunwoody Craig J and Linton Mark A. Tracing Interactive 3D Graphics Programs. *Computer Graphics:Symposium on Interactive 3D Graphics*, 24(2):155 – 163, 1990.

[11] So K., Darema-Rogers F., George D.A., Norton V.A., and Pfister G.F. PSIMUL - A System for Parallel Execution of Parallel Programs. Technical report, Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1986.

[12] Graham Susan L., Kessler Peter B., and McKusick Marshall K. gprof: a Call Graph Execution Profiler. *SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices*, 17(6), June 1982.

[13] Hennessey L. and Patterson D. A. *Computer Architecture:A Quantitive Approach*. Morgan Kaufman, 1990.

[14] Hendren Laurie, Donawa Chris, Emami Maryam, Gao Guang R., Justiani, and Sridharan Bhama. Designing the McCAT Compiler based on a Family of Strucured Intermediate Representations. Technical report, Advanced Computer Architecture and Program Structures Group, School of Computer Science, McGill University, Technical Memo 46, May 1992.

[15] Larus James R. Abstract Execution:A Technique for Efficiently Tracing Programs. *Software – Practice and Experience*, 20(12), December 1990.

[16] Covington R.C., Madala S., Mehta V., Jump J.R., and Sinclair J.B. The Rice Parallel Processing Testbed. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 4–11, May 1988.

[17] Vugranam C. Sreedhar. Improved Memory Latency and Network Traffic for a Multithreaded System by Exploiting Program Locality. Advanced Computer Architecture and Program Structures Group, June 1992.

[18] Jagannath Sumithra. Profiling on the DLX Simulator. Submitted as project report for course 308-573B at the school of Computer Science, McGill University, Canada, April 1992.

[19] Axerold T.S., Dubois P., and Elgroth P. A Simulator for MIMD Performance Prediction - Application to the S-1 Mkiia Multiprocessor. *Parallel Computing*, 1:237–274, 1984.

[20] Sarkar Vivek. Determining Average Program Execution Times and their Variance. *SIGPLAN '89 Conference on Parallel Language Design and Implementation, SIGPLAN Notices*, 24(7), July 1989.

[21] Wen-Hann Wang and Baer Jean-Loup. Efficient Trace Driven Simulation Methods for Cache Performance Analysis. *Performance Evaluation Review*, 18(1):27–36, May 1990.

[22] Aral Ziya and Gertner Ilya. Non-Intrusive and interactive profiling in Parasight. *SIGPLAN '88 Symposium on PPEALS Parallel Programming:Experiencewith Applications, Languages and Systems*, 23, September 1988.

APPENDIX A

```
/*
                                    TOMCATV
*/
/* tomexp.f -- translated by f2c (version of 8 April 1991  13:00:55).
   You must link the resulting object file with the libraries:
        -lF77 -lI77 -lm -lc   (in that order)
*/

#include "f2c.h"

/* Common Block Declarations */

struct {
    doublereal s1alt;
} zeit_;

#define zeit_1 zeit_

/* Table of constant values */

static integer c__1 = 1;

/* ************************************************************** */

/*     MESHGENERATION WITH THOMPSONS SOLVER, VECTORIZED VERSION */

/*     SET  N = 33, 65, 129 */

/*     DO  SCALAR AND VECTOR  RUNS */

/*     W.GENTZSCH   FH REGENSBURG   F.R.G. */

/* *************************************************************** */

/* Main program */ main()
{
    /* Format strings */
/*    static char fmt_1100[] = "(\0021\002,2x,\002*** 2-D ITERATION BEHAVIOR *\
**\002,/)";
```

```
    static char fmt_1200[] = "(/,1x,\002IT   I   J   X-COR      I   J   Y-CO\
R      I   J  \002,\002 X-RES      I   J   Y-RES\002,/)";
    static char fmt_1300[] = "(i3,i4,i4,e11.4,i4,i4,e11.4,i4,i4,e11.4,i4,i4,\
e11.4)"; */

    /* System generated locals */
    integer i__1, i__2;
    doublereal d__1;

    /* Builtin functions */
/*    integer s_wsfe(), e_wsfe(), do_fio();  HAND-CONVERTED TO PRINTFs
      int s_stop();  */

    /* Local variables */
    static doublereal alfa, dmax_, dxcm, dycm;
    static integer ixcm, lmax, jxcm, iycm, jycm, irxm, jrxm, irym, jrym;
    static doublereal a, b, c, d[81]    /* was [9][9] */, h;
    static integer i, j, k, l, m;
    static doublereal r, x[81]  /* was [9][9] */, y[81] /* was [9][9] */,
            relfa, aa[81]       /* was [9][9] */, dd[81]        /* was [9][9]
            */;
    static integer im, jm, ll, ip, jp;
    static doublereal qi, qj, rx[81]    /* was [9][9] */, ry[81]        /*
            was [9][9] */, xx, yx, xy, yy;
    static integer i2m, j2m, i1p, j1p;
    static doublereal abx, aby, rel, eps, rxm, rym, pxx, qxx, pyy, qyy, pxy,
            qxy;

    /* Fortran I/O blocks */
/*    static cilist io___11 = { 0, 6, 0, fmt_1100, 0 };
    static cilist io___12 = { 0, 6, 0, fmt_1200, 0 };
    static cilist io___58 = { 0, 6, 0, fmt_1300, 0 }; */


    zeit_1.s1alt = (float)0.;

    alfa = (float).1;
    relfa = (float).98;
    lmax = 100;
    eps = (float)5e-8;
    h = (float).125;
```

```
    rel = (float)2. / relfa;

/*     INITIAL FIELDS FOR X(I,J) AND Y(I,J) WITH STRECH FACTOR ALFA */

    for (i = 1; i <= 9; ++i) {
/* L4: */
        x[i - 1] = (real) (i - 1) / (float)8.;
    }

    for (i = 1; i <= 9; ++i) {
        x[i + 71] = x[i - 1];
        x[i * 9 - 9] = (float)0.;
        x[i * 9 - 1] = (float)1.;
        y[i - 1] = (float)0.;
        y[i * 9 - 9] = x[i - 1];
        y[i * 9 - 1] = alfa * x[i - 1];
/* L1: */
    }
    for (i = 1; i <= 9; ++i) {
/* L2: */
        y[i + 71] = ((float)1. - x[i - 1]) * y[72] + x[i - 1] * y[80];
    }
    for (j = 2; j <= 8; ++j) {
        for (i = 2; i <= 8; ++i) {
            x[i + j * 9 - 10] = x[i - 1] * (float).9;
            y[i + j * 9 - 10] = (((float)1. - x[i - 1]) * y[j * 9 - 9] + x[i
                    - 1] * y[j * 9 - 1]) * (float).9;
/* L3: */
        }
    }
/*     WRITE (6,1400) ((X(I,J),I=1,N),J=1,N) */
/*     WRITE (6,1400) ((Y(I,J),I=1,N),J=1,N) */
/* Produced by:      WRITE (6,1100) */
/*                   1100 FORMAT(1H1,2X,30H*** 2-D ITERATION BEHAVIOR ***,/) */
printf("  *** 2-D ITERATION BEHAVIOR ***\n");
/*    s_wsfe(&io___11);
    e_wsfe(); */
/* Produced by:      WRITE (6,1200) */
/*                   1200 FORMAT(/,1X,50HIT   I   J   X-COR      I   J   Y-COR      I   J  ,
 *                       125H X-RES      I   J   Y-RES,/) */
printf(" IT   I   J   X-COR      I   J   Y-COR      I   J   X-RES      I   J   Y-RES\n\n");
```

```
/*    s_wsfe(&io___12);
    e_wsfe(); */

    i1p = 2;
    j1p = 2;
    i2m = 8;
    j2m = 8;


/*     START ITERATION LOOP 190 */

/*     CALL TIMEB(D1,D2,D3,D4,D5) */
    ll = 0;
L190:
    ixcm = 0;
    jxcm = 0;
    dxcm = (float)0.;
    iycm = 0;
    jycm = 0;
    dycm = (float)0.;
    irxm = 0;
    jrxm = 0;
    rxm = (float)0.;
    irym = 0;
    jrym = 0;
    rym = (float)0.;

/*     J-LOOP */

    m = 0;
    i__1 = j2m;
    for (j = j1p; j <= i__1; ++j) {
        jp = j + 1;
        jm = j - 1;
        ++m;

/*     I-LOOP */

        i__2 = i2m;
        for (i = i1p; i <= i__2; ++i) {
            ip = i + 1;
```

```
            im = i - 1;
            xx = x[ip + j * 9 - 10] - x[im + j * 9 - 10];
            yx = y[ip + j * 9 - 10] - y[im + j * 9 - 10];
            xy = x[i + jp * 9 - 10] - x[i + jm * 9 - 10];
            yy = y[i + jp * 9 - 10] - y[i + jm * 9 - 10];
            a = (xy * xy + yy * yy) * (float).25;
            b = (xx * xx + yx * yx) * (float).25;
            c = (xx * xy + yx * yy) * (float).125;
            qi = (float)0.;
            qj = (float)0.;
/*     QI = A*0.5 */
/*     QJ = B*0.5 */
            aa[i + m * 9 - 10] = -b;
            dd[i + m * 9 - 10] = b + b + a * rel;
            pxx = x[ip + j * 9 - 10] - x[i + j * 9 - 10] * (float)2. + x[im +
                    j * 9 - 10];
            qxx = y[ip + j * 9 - 10] - y[i + j * 9 - 10] * (float)2. + y[im +
                    j * 9 - 10];
            pyy = x[i + jp * 9 - 10] - x[i + j * 9 - 10] * (float)2. + x[i +
                    jm * 9 - 10];
            qyy = y[i + jp * 9 - 10] - y[i + j * 9 - 10] * (float)2. + y[i +
                    jm * 9 - 10];
            pxy = x[ip + jp * 9 - 10] - x[ip + jm * 9 - 10] - x[im + jp * 9 -
                    10] + x[im + jm * 9 - 10];
            qxy = y[ip + jp * 9 - 10] - y[ip + jm * 9 - 10] - y[im + jp * 9 -
                    10] + y[im + jm * 9 - 10];

/*     CALCULATE RESIDUALS ( EQUIVALENT TO RIGHT HAND SIDES OF EQU
S.) */

            rx[i + m * 9 - 10] = a * pxx + b * pyy - c * pxy + xx * qi + xy *
                    qj;
            ry[i + m * 9 - 10] = a * qxx + b * qyy - c * qxy + yx * qi + yy *
                    qj;
/* L250: */
        }
/* L310: */
    }

/*     DETERMINE MAXIMUM VALUES OF RESIDUALS */
```

```
    i__1 = m;
    for (j = 1; j <= i__1; ++j) {
        i__2 = i2m;
        for (i = i1p; i <= i__2; ++i) {
            if ((d__1 = rx[i + j * 9 - 10], abs(d__1)) < abs(rxm)) {
                goto L262;
            }
            rxm = rx[i + j * 9 - 10];
            irxm = i;
            jrxm = j;
L262:
            if ((d__1 = ry[i + j * 9 - 10], abs(d__1)) < abs(rym)) {
                goto L270;
            }
            rym = ry[i + j * 9 - 10];
            irym = i;
            jrym = j;
L270:
            ;
        }
    }

/*     SOLVE TRIDIAGONAL SYSTEMS IN PARALLEL */

    if ((i__2 = m - 1) < 0) {
        goto L601;
    } else if (i__2 == 0) {
        goto L201;
    } else {
        goto L301;
    }
L201:
    i__2 = i2m;
    for (i = i1p; i <= i__2; ++i) {
        rx[i - 1] /= dd[i - 1];
        ry[i - 1] /= dd[i - 1];
/* L102: */
    }
    goto L601;
L301:
    i__2 = i2m;
```

```
    for (i = i1p; i <= i__2; ++i) {
        d[i - 1] = (float)1. / dd[i - 1];
/* L103: */
    }
    i__2 = m;
    for (j = 2; j <= i__2; ++j) {
        i__1 = i2m;
        for (i = i1p; i <= i__1; ++i) {
            r = aa[i + j * 9 - 10] * d[i + (j - 1) * 9 - 10];
            d[i + j * 9 - 10] = (float)1. / (dd[i + j * 9 - 10] - aa[i + (j -
                    1) * 9 - 10] * r);
            rx[i + j * 9 - 10] -= rx[i + (j - 1) * 9 - 10] * r;
            ry[i + j * 9 - 10] -= ry[i + (j - 1) * 9 - 10] * r;
/* L401: */
        }
    }
    i__1 = i2m;
    for (i = i1p; i <= i__1; ++i) {
        rx[i + m * 9 - 10] *= d[i + m * 9 - 10];
        ry[i + m * 9 - 10] *= d[i + m * 9 - 10];
/* L411: */
    }
    i__1 = m;
    for (j = 2; j <= i__1; ++j) {
        k = m - j + 1;
        i__2 = i2m;
        for (i = i1p; i <= i__2; ++i) {
            rx[i + k * 9 - 10] = (rx[i + k * 9 - 10] - aa[i + k * 9 - 10] *
                    rx[i + (k + 1) * 9 - 10]) * d[i + k * 9 - 10];
            ry[i + k * 9 - 10] = (ry[i + k * 9 - 10] - aa[i + k * 9 - 10] *
                    ry[i + (k + 1) * 9 - 10]) * d[i + k * 9 - 10];
/* L501: */
        }
    }

/*     ADD CORRECTIONS */

    l = 0;
    i__2 = j2m;
    for (j = j1p; j <= i__2; ++j) {
        ++l;
```

```
        i__1 = i2m;
        for (i = i1p; i <= i__1; ++i) {
            x[i + j * 9 - 10] += rx[i + l * 9 - 10];
            y[i + j * 9 - 10] += ry[i + l * 9 - 10];
/* L290: */
        }
    }

/*      PREPARE OUTPUT OF CONVERGENCE BEHAVIOUR */

L601:
    ++ll;
/*     s_wsfe(&io___58);
    do_fio(&c__1, (char *)&ll, (ftnlen)sizeof(integer));
    do_fio(&c__1, (char *)&ixcm, (ftnlen)sizeof(integer));
    do_fio(&c__1, (char *)&jxcm, (ftnlen)sizeof(integer));
    do_fio(&c__1, (char *)&dxcm, (ftnlen)sizeof(doublereal));
    do_fio(&c__1, (char *)&iycm, (ftnlen)sizeof(integer));
    do_fio(&c__1, (char *)&jycm, (ftnlen)sizeof(integer));
    do_fio(&c__1, (char *)&dycm, (ftnlen)sizeof(doublereal));
    do_fio(&c__1, (char *)&irxm, (ftnlen)sizeof(integer));
    do_fio(&c__1, (char *)&jrxm, (ftnlen)sizeof(integer));
    do_fio(&c__1, (char *)&rxm, (ftnlen)sizeof(doublereal));
    do_fio(&c__1, (char *)&irym, (ftnlen)sizeof(integer));
    do_fio(&c__1, (char *)&jrym, (ftnlen)sizeof(integer));
    do_fio(&c__1, (char *)&rym, (ftnlen)sizeof(doublereal));
    e_wsfe();   */
/* Produced by
      WRITE (6,1300)LL,IXCM,JXCM,DXCM,
     1                 IYCM,JYCM,DYCM,
     2                 IRXM,JRXM, RXM,
     3                 IRYM,JRYM, RYM
 1300 FORMAT(I3,I4,I4,E11.4,I4,I4,E11.4,
     1             I4,I4,E11.4,I4,I4,E11.4)     */
printf("%4d%4d%4d%11.4e%4d%4d%11.4e%4d%4d%11.4e%4d%4d%11.4e\n",
       ll,ixcm,jxcm,dxcm,iycm,jycm,dycm,irxm,jrxm,rxm,irym,jrym,rym);
    abx = abs(rxm);
    aby = abs(rym);
    dmax_ = max(abx,aby);
    if (ll < lmax && dmax_ > eps) {
        goto L190;
```

```
    }

/*     CALL TIMEB(T1,T2,T3,T4,T5) */
/*     END OF ITERATION LOOP 190 */

/*     WRITE (6,1400) ((X(I,J),I=1,N),J=1,N) */
/*     WRITE (6,1400) ((Y(I,J),I=1,N),J=1,N) */
/*     WRITE(6,1414) T1 */
/* 1414 FORMAT(1X,' TIME = ',F12.6) */
/* L1400: */
/*    s_stop("", 0L); */
} /* main */
```

```
/*
                                    BUBBLE SORT
*/
/* This program implements a bubble sort */

#define  nil            0
#define  false          0
#define  true           1
#define  bubblebase     1.61
#define  dnfbase        3.5
#define  permbase       1.75
#define  queensbase     1.83
#define  towersbase     2.39
#define  quickbase      1.92
#define  intmmbase      1.46
#define  treebase        2.5
#define  mmbase         0.73
#define  fpmmbase       2.92
#define  puzzlebase     0.5
#define  fftbase        1.11
#define  fpfftbase      4.44

    /* Sorts an array using bubblesort */
#define SORTELEMENTS     5000
#define SRTELEMENTS      500

int    seed;

int    sortlist[SORTELEMENTS+1],
    biggest, littlest,
    top;


Initrand () /* Supplies the seed for bubble sort */
    {
    seed = 74755;
    };

int Rand () /* Supplies the random numbers using the seed for bubble sort */
    {
    seed = (seed * 1309 + 13849) & 65535;
```

```
    return( seed );
    };

    bInitarr() /* Creates and sorts the array */
        {
        int i, temp;
        Initrand();
        biggest = 0; littlest = 0;
        for ( i = 1; i <= SRTELEMENTS; i++ )
            {
            temp = Rand();
            sortlist[i] = temp - (temp/100000)*100000 - 50000;
            if ( sortlist[i] > biggest ) biggest = sortlist[i];
            else if ( sortlist[i] < littlest ) littlest = sortlist[i];
            };
        };

main() /* Main program */
    {
    int i, j;
    bInitarr();
    top=SRTELEMENTS;

    while ( top>1 ) {

        i=1;
        while ( i<top ) {

            if ( sortlist[i] > sortlist[i+1] ) {
                j = sortlist[i];
                sortlist[i] = sortlist[i+1];
                sortlist[i+1] = j;
                };
            i=i+1;
            };

        top=top-1;
        };
    if ( (sortlist[1] != littlest) || (sortlist[SRTELEMENTS] != biggest) )
        printf ( "Error3 in Bubble.\n");
    else for (i =0; i < SRTELEMENTS; i += 99) printf("%d\n", sortlist[i]);
```

```
};
```

APPENDIX B

```
/*
                                   TOWERS OF HANOI
*/
/* Towers of Hanoi program */
main()
{
    register int i;
    /* i = getchar() - '0'; */
    Move(3, 1, 3, 2);
    }

Move(disks,from,to,with) /* Solves the problem recursively */.
int disks,from,to,with;
{
    if(disks == 0) return;
    Move(disks-1,from,with,to);
    printf("move from %d to %d\n",from,to);
    Move(disks-1,with,to,from);
    }
```

```
/*
                                FIBONACCI NUMBERS
*/
/* This program generates the first 20 fibonacci numbers */

fib(n) /* Generates the fibonacci numbers recursively */
int n;
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

int n = 20;
int x[20];

main() /* Main program */
{
    register int i;

    for (/* 1798 */ i = 0; /* 1800 */ i < n; /* 1847 */ i++) {
        printf("i = %d, n = %d: %d \n", i, n, x[i] = fib(i));
    }
    printf("\n");
}
```

APPENDIX C

```
/*
                                BINARY SEARCH
*/
/* This program implements a binary search algorithm with heap sorting over an
array in the heap space */

#include <stdio.h>
#define NUM_ELEM  100

typedef struct info table;  /* Structure of each element of the array */
struct info {
            unsigned int        address;
            unsigned short  refs;
            unsigned short  last_cycle;
            table           *next;
           };

table  *stack_table_ptr;
int     counterst;
main () /* Main program that fills the array with random numbers, then calls
the heap sort routine to sort it and then the binary search function to search
for a key. This is done NUM_ELEM times */
{
    unsigned int address = 74755;
    int i, j;
    FILE   *out_file;
    stack_table_ptr = (table *) malloc ((NUM_ELEM-1)*sizeof(table));
    for (i=1; i<NUM_ELEM; i++)
    {
       stack_table_ptr[i].address = address;
       stack_table_ptr[i].refs = 0;
       stack_table_ptr[i].last_cycle = 0;
       sort_the_list (i, stack_table_ptr);
       counterst = binary_search(i, address, stack_table_ptr);
       address = (address * 1309 + 13849) & 65535;
    }
    printf ("DONE\n");
    out_file = fopen ("out-file", "w");
    for (i=1; i<NUM_ELEM; i++)
```

```
      fprintf (out_file, "%x\n", stack_table_ptr[i].address);
   fclose (out_file);
}

binary_search (num_elem, address, head_ptr) /* This does the actual binary
search. It returns 0 if not found and 1 if found. These return values are not
used. The algorithm is taken from the book The art of Computer Programming vol
3. By Donald E. Knuth. Published by Addison Wesley */
int num_elem;
unsigned int address;
table    *head_ptr;
{
    int l1 = 0, u=num_elem, index=2, found=0;

    /* Do binary search */
       while (1)
       {
         if (u < l1)
         {
            found = 0;
            break;
         }
         index = ((l1+u)/2);
         if (address < head_ptr[index].address)
            u = index-1;
         else if (address > head_ptr[index].address)
             l1 = index+1;
         else if (address == head_ptr[index].address)
         {
            found ++;
            break;
         }
       }
       if (!found)  /* Insert to end of the table */
         return 0;
       else
         return 1;
}
int sort_the_list (num_elem, head_ptr) /* The heap sort algorithm. Taken from
the book: The art of Computer Programming Vol 3. by Donald E. Knuth; Published
by Addison Wesley */
```

```
int   num_elem;
table *head_ptr;
{
    int          l=((num_elem-1)/2)+1, r=(num_elem-1), i, j;
    table        *the_record = (table *) malloc(sizeof(table));
    unsigned     the_key=0;
    int          counter = num_elem-1, iter_cnt=0, cnt=0, o;

    /* Use heap sort */
    if (counter >= 2)
        while (1)
        {
            iter_cnt ++;
            if (l > 1)
            {
                l --;
               the_record->address = head_ptr[l].address;
               the_record->refs = head_ptr[l].refs;
               the_record->last_cycle = head_ptr[l].last_cycle;
               the_key = head_ptr[l].address;
            }
            else
            {
               the_record->address = head_ptr[r].address;
               the_record->refs = head_ptr[r].refs;
               the_record->last_cycle = head_ptr[r].last_cycle;
               the_key = head_ptr[r].address;
               head_ptr[r].address = head_ptr[1].address;
               head_ptr[r].refs = head_ptr[1].refs;
               head_ptr[r].last_cycle = head_ptr[1].last_cycle;
               r --;
               if (r == 1)
               {
                   head_ptr[1].address = the_record->address;
                   head_ptr[1].refs = the_record->refs;
                   head_ptr[1].last_cycle = the_record->last_cycle;
                   break;
               }
            }
            j = l;
            while (1)
```

```
            {
                i = j;
                j = 2*j;
                if (j < r)
                {
                   if (head_ptr[j].address < head_ptr[j+1].address)
                   {
                      j++;
                   }
                }
                if (j > r)
                {
                   head_ptr[i].address = the_record->address;
                   head_ptr[i].refs = the_record->refs;
                   head_ptr[i].last_cycle = the_record->last_cycle;
                   break;
                }
                if (the_key >= head_ptr[j].address)
                {
                   head_ptr[i].address = the_record->address;
                   head_ptr[i].refs = the_record->refs;
                   head_ptr[i].last_cycle = the_record->last_cycle;
                   break;
                }
               head_ptr[i].address = head_ptr[j].address;
               head_ptr[i].refs = head_ptr[j].refs;
               head_ptr[i].last_cycle = head_ptr[j].last_cycle;
            }
        }
}
```

```
/*
                 BALANCED BINARY TREE CREATION AND INSERTION
*/
/* This program is an implementation of a balanced binary tree. It creates the
tree by inserting new elements. Then the tree is traversed once to print out
the elements in sorted order and another time to free the tree*/

#include <stdio.h>
#define NUM_ELEM 200

typedef struct tree_st node; /* Structure of each node of the tree */
struct tree_st {
                unsigned int        address;
                long                last_cycle;
                long                tot_cycles;
                long                num_filled;
                long                balance_factor;
                node                *llink;
                node                *rlink;
            };
node    *tree_ptr;
int     traverse_tree();
int     my_free();
main () /* This is the main function that supplies the random numbers to
insert into the tree and call the other functions */
{
    unsigned long  address = 74755;
    int  i;
    FILE   *out_file;
    tree_ptr = (node *) malloc (sizeof (node));;
    tree_ptr->rlink = NULL;
    tree_ptr->llink = NULL;
    for (i=0; i<NUM_ELEM; i++)
    {
        make_table (i, address);
        address = (address * 1309 + 13849) & 65535;
    }
    out_file =
  fopen ("/labs/acaps/acaps/u1/sumi/gccdir/dlxsim/BENCHMARKS/HEAP/bin-heap/out",
        "w");
   if (tree_ptr->rlink != NULL)
```

```
      traverse_tree (tree_ptr->rlink, out_file);
   fclose (out_file);
   my_free (tree_ptr->rlink);
   free (tree_ptr);
}
my_free (tree) /* Traverse the tree using in-order traversal and free it */
node *tree;
{
    if (tree->llink != NULL)
       my_free(tree->llink);
    if (tree->rlink != NULL)
       my_free(tree->rlink);
    free (tree);
}
traverse_tree (tree, handle) /* Traverse the tree using in-order traversal
and print the nodes in ascending order */
node  *tree;
FILE  *handle;
{
    if (tree->llink != NULL)
        traverse_tree (tree->llink, handle);
    if (tree->num_filled)
        fprintf (handle, "%x   %f\n", tree->address,
                (float)(tree->tot_cycles/tree->num_filled));
    if (tree->rlink != NULL)
       traverse_tree (tree->rlink, handle);
}
int make_table (insn_num, address) /* This function actually creates and
inserts items into the balanced binary tree */
int      insn_num;
unsigned int address;
{
    node        *head_ptr = tree_ptr, *t, *s, *p;
   if (head_ptr->rlink == NULL)
   {
       head_ptr->rlink = (node *) malloc (sizeof(node));
       head_ptr->rlink->address = address;
       head_ptr->rlink->num_filled = 1;
       head_ptr->rlink->last_cycle = insn_num;
       head_ptr->rlink->tot_cycles = 0;
       head_ptr->rlink->balance_factor = 0;
```

```
      head_ptr->llink = 0;
      head_ptr->rlink->rlink = NULL;
      head_ptr->rlink->llink = NULL;
      return;
   }
   { /* Create and insert into balanced binary tree. This algorithm is taken
from the book: The art of Computer Programming Vol.3 by Donald E. Knuth.
Published by Addison Wesley */
      node        *t, *s, *p, *q, *r;
      int         a=0;
      t = head_ptr;
      s = p = head_ptr->rlink;
      while (1)
      {
          if (address == p->address)
          {
             p->last_cycle = insn_num;
             p->num_filled ++;
             return;
          }
          else if (address < p->address)
          {
              q = p->llink;
              if (q == NULL)
              {
                 q = (node *) malloc (sizeof (node));
                 p->llink = q;
                 break;
              }
              else if (q->balance_factor)
              {
                 t = p;
                 s = q;
              }
          }
          else /* if address > p->address */
          {
             q = p->rlink;
             if (q == NULL)
             {
                q = (node *) malloc (sizeof (node));
```

```
            p->rlink = q;
            break;
         }
         else if (q->balance_factor)
         {
            t = p;
            s = q;
         }
      }
      p = q;
   }
   q->address = address;
   q->tot_cycles = 0;
   q->last_cycle = insn_num;
   q->num_filled = 1;
   q->rlink = NULL;
   q->llink = NULL;
   q->balance_factor = 0;
   if (address < s->address)
      r = p = s->llink;
   else
      r = p = s->rlink;
   while (p != q)
   {
      if (address < p->address)
      {
         p->balance_factor = -1;
         p = p->llink;
      }
      else if (address > p->address)
      {
         p->balance_factor = 1;
         p = p->rlink;
      }
      else
         break;
   }
   if (address < s->address)
      a = -1;
   else
      a = 1;
```

```
if (s->balance_factor == 0)
{
   s->balance_factor = a;
   return;
}
else if (s->balance_factor == -a)
{
   s->balance_factor = 0;
   return;
}
else if (s->balance_factor == a)
{
   if (r->balance_factor == a)
   {
      p = r;
      if (a == -1)
      {
         s->llink = r->rlink;
         r->rlink = s;
      }
      else if (a == 1)
      {
         s->rlink = r->llink;
         r->llink = s;
      }
      s->balance_factor = r->balance_factor = 0;
      goto finish;
   }
   else if (r->balance_factor == -a)
   {
      if (a == -1)
      {
         p = r->rlink;
         r->rlink = p->llink;
         p->llink = r;
         s->llink = p->rlink;
         p->rlink = s;
      }
      else if (a == 1)
      {
         p = r->llink;
```

```
            r->llink = p->rlink;
            p->rlink = r;
            s->rlink = p->llink;
            p->llink = s;
         }
         if (p->balance_factor == a)
         {
             s->balance_factor = -a;
             r->balance_factor = 0;
         }
         else if (!(p->balance_factor))
         {
            s->balance_factor = 0;
            r->balance_factor = 0;
         }
         else if (p->balance_factor == -a)
         {
            s->balance_factor = 0;
            r->balance_factor = a;
         }
      }
      finish:
         if (s == t->rlink)
            t->rlink = p;
         else
            t->llink = p;
     }
  }
}
```

```
/*
                                 WORD PROCESSOR
*/
/*
 * Test for pointers (heap references). This program creates and traverses a
 * linked list.
 */
#include <stdio.h>

typedef struct word word_struc;
struct word {
              char       *word;
              word_struc *next_word;
            };
typedef struct var_t line_struc;
struct var_t{
              word_struc     *word_ptr;
              line_struc     *next_line;
            };

main () /* This function does all the processing of reading in the text,
separating it into words and lines and putting them into the linked list. */
{
   line_struc  *line_head, *tmp_line, *prev_line=NULL;
   word_struc  *tmp_word, *prev_word=NULL;
   char        line[80], word[80];
   FILE        *handle;
   int          done=0, len=0,j;

   handle = fopen ("/u1/grad/sumi/IBDLX/Tests/tfile", "r");
   line_head = tmp_line = (line_struc *) malloc (sizeof (line_struc));
   while (!done)
   {
       tmp_word = tmp_line->word_ptr = (word_struc *) malloc (sizeof
                                               (word_struc));
       tmp_word->word = NULL;
       tmp_word->next_word = NULL;
       for (j=0; j<80; j++)
          line[j] =0;
       if (fgets (line, 80, handle) == NULL)
          done ++;
```

```
else
{
   int   eol=0,i=0;
   while (!eol)
   {
       int end_of_str=0;
       for (j=0; j<80; j++)
          word[j]=0;
       j=0;
       while (!end_of_str)
       {
          if (line[i] == '\n')
          {
             eol ++;
             break;
          }
          word [j] = line [i];
          switch (line[i])
          {
             case ';':
             case ' ':
             case '.':
             case ',':
             case '?':
             case '/':
             case ':':
             case '[':
             case ']':
             case '"':
             case '-':
             case '!':
             case '(':
             case ')':
             case '\t':
                end_of_str ++;
                break;
             default:
                break;
          }
          i++;
          j++;
```

```
            }
            len = strlen(word);
            tmp_word->word = (char *) malloc (len+1);
            strcpy (tmp_word->word, word);
            prev_word = tmp_word;
            tmp_word->next_word = (word_struc *) malloc (sizeof(word_struc));
            tmp_word = tmp_word->next_word;
            tmp_word->word = NULL;
            tmp_word->next_word = NULL;
        }
        prev_word->next_word = NULL;
        prev_line = tmp_line;
        tmp_line->next_line = (line_struc *) malloc (sizeof (line_struc));
        tmp_line = tmp_line->next_line;
        tmp_line->next_line = NULL;
        tmp_line->word_ptr = NULL;
     }
  }
  prev_line->next_line = NULL;
  print_the_list (line_head);
}

print_the_list (head) /* Print the text line by line */
line_struc  *head;
{
     line_struc *tmp_line = head;
     word_struc *tmp_word;

     while (tmp_line != NULL)
     {
        tmp_word = tmp_line->word_ptr;
        while (tmp_word != NULL)
        {
           printf ("%s", tmp_word->word);
           tmp_word = tmp_word->next_word;
        }
        tmp_line = tmp_line->next_line;
        printf ("\n");
     }
}
```