Program Execution Monitoring:

Software Structures and

Architectural Support

Thesis submitted in accordance with the requirements of the University of Liverpool for the Degree of Doctor in Philosophy by David Wilkinson September 1990

ACKNOWLEDGEMENTS

I would like to thank, first and foremost, my supervisors, Paul Leng and Colin Charlton, for their help and encouragement throughout the project; the Science and Engineering Research Council for providing a grant; Tim Robinson and High Level Hardware Ltd. for their help with microcode problems; Ken Chan and Keith Halewood for their knowledge and advice on all things computing; and finally my parents for their support during my academic studies.

Abstract

The work described in this thesis addresses the problems of monitoring the execution of modern, high-level language software and, in particular, problems associated with the provision of high-level control over target program execution with an acceptable performance degradation.

The state of the art indicates that the development of execution monitoring tools has largely avoided the issue of higher-level control, probably due to the lack of machine support for these facilities and the inevitable excessive performance overhead which would result. Only in the field of real-time monitoring has suitable machine support been described, usually consisting of specialised electronics.

We describe a monitoring environment which lends itself to the monitoring of high-level software, and enables monitoring software to provide the level of control required through the use of appropriate software structures and a set of monitoring primitives, for which suitable support can be provided.

A set of abstract-level events is introduced which can be monitored by the use of a single type of monitoring primitive and the inspection of the target process state at single instants in time. A notation is introduced for representing sequences of these events as a directed graph, where the arcs indicate a chronological ordering, enabling the monitoring of higher-level concepts and information which is not, in general, preserved during execution. To provide these facilities at an acceptable level of performance degradation, means of implementing architectural support is examined.

An experimental implementation of the directed graph mechanism and a microcoded version of architectural support for a virtual memory machine is outlined, as is an analysis of the performance of the system. The performance figures obtained indicate that it is possible to provide monitoring facilities for high-level software which performs with an acceptable performance overhead and is applicable to a wide range of machines.

Table of Contents

1. Introduction

1.1. Software Development

The software development cycle is usually represented as consisting of six stages: specification, functional decomposition, coding, testing, debugging and evaluation [Johnson78b]. At all stages of the cycle reliability of the final product is an important factor.

The specification of a system attempts to prepare, in a complete and consistent manner, the intentions of the user, thus providing a basis for the design and implementation of the system [Martin88] [Gerrard90]. A number of techniques have been developed to aid in this specification stage of the development cycle. Since the first complete technique for system analysis, SOP [IBM61], many others have followed, including methods which use languages to describe relationships between objects and activities. Examples of such languages include TAG [Head71], JSP [Jackson75], PSL [Teichrow77], RSL [Alford77], SADT [Ross77] and GIST [Balzer81]. Details of these methods can be found in [Fairley85]. The reliability of a system can be better guaranteed if a complete specification is provided, which is produced more easily and clearly if a formal specification technique is used [Quirk85].

The second and third stages of the development cycle, functional decomposition and coding, have, perhaps, had most attention paid to them. Software reliability has been improved through the advocacy of structured programming and the design of high-order languages.

Many authorities have noted that the latter stages of the development cycle have not kept up with the pace of change of the former stages [Evans66] [Zelkowitz71] [Glass80] [Plattner81] [Deutsch82] [Gramlich83] [Plattner84]. One reason for the apparent lack of interest in the testing and debugging stages is the research into formal proofs for program correctness. Such proofs include symbolic execution and evaluation [King76] [Cheatham79] [Kishimoto83a] [Young88] and the use of flow expressions [Shaw78]. However, it can be assumed, from the literature, that it will be some time before formal proofs are ready to replace the testing and debugging phases in the software development cycle [Kopetz79] [Lauesen79] [Plattner81] [Deutsch82]. Thus, it is still important for all programs to enter the testing and, if necessary, the debugging stages of development.

Testing techniques attempt to execute the code with enough sets of input data to infer a degree of confidence in the code. In all but the simplest cases it would be impossible to test all possible sets of input data and so the strategy is to develop a system which generates sets of input data which adequately test the program [Liskov86] [Weyuker86]. There are many criteria for the selection of test data. One method relies on programmer experience to choose input data which tests known problem areas [Bauer77]. Another method, the acceptance test, uses 'real' input data, assuming that the user, being the only one who has authentic knowledge of the intended use of the software, is best able to pick test cases from the actual use of the system [Kopetz79]. The former method does not take into account the intended use of the product and the thoroughness of the test is likely to vary from one programmer to another. The acceptance test provides only statistical data on the reliability of the software and is more likely to test only main execution paths, thus neglecting the more obscure conditions. A further testing strategy is structural testing, where the input data is chosen from examination of the program structure [Fairley85] [Ntafos88]. Knowledge of the program structure, together with the results of the test, builds the basis of inductive evidence for the correctness of software [Kopetz79]. Structural testing occurs in a number of forms: for example, all instructions of the program are executed at least once, every branch point is tested, in each direction, at least once, or all control paths are tested. Research into methodologies for test data generation can be found in [Howden75] [Ramamoorthy76] [Clarke83] [Girgis85] [Cantone87] [Clarke89]. The ability of such testing tools to produce adequate testing strategies can be shown using symbolic traces through program instrumentation [Huang78] [Huang80].

Testing reveals the *point of detection* of an error, or "bug". This recognition of a bug, however, does not immediately reveal the *point of origin* of the bug; it is the debugging stage of the development cycle which performs the necessary diagnosis [Brown73] [Johnson78b] [Fairley85]. It is also suggested that the debugging process incorporates the correction of any bugs. To aid in the debugging of software many systems provide some sort of debugging tool. However, advances in structured programming techniques and high-level languages have increased the semantic gap between debugging tools and the view the programmer holds of the software. Most debugging tools of today still lend themselves more to the debugging of assembly language code than high-level language programs. This means the high-level language programmer is required to know extra information such as the operating system and compiler memory allocation techniques. The inadequacy of tools [Gramlich83] has resulted in the continual advocacy of the insertion of special probe statements, into the software source, as a debugging method [Bauer77] [Deutsch82].

Once all the known bugs have been removed and the program has been tested to the required degree of confidence then, if necessary, the evaluation stage is entered to highlight areas of inefficient code. The three software performance evaluation methods are: selection evaluation, in which performance is included as a criterion in the decision to obtain a system; performance projection, in which performance is estimated for a system which does not yet exist; and performance monitoring, which provides data on the actual performance of a system [Lucas71]. Performance monitoring is the only technique which applies to an existing piece of software. Historically, only hardware performance was evaluated but with larger and more complex software systems residing on the hardware it is important that the software is as efficient as possible.

1.2. Execution Monitoring

One requirement which appears to be common to the testing, debugging and evaluation stages is the ability of the software developer to be able to view the internal execution steps of a process.

> "Program testing with execution monitoring is not only used to verify the correct

> operation of a program, but is also very useful when the exact cause of an observed malfunction must be determined, when internal details must be understood by somebody who is not yet familiar with a program, or when software must be optimised to increase execution speed." [Tiberghien86:384]

The technique employed to enable the viewing of the internal steps of a process is known as **execution monitoring** and is based on observability and controllability [Kopetz79] [Plattner84]. Observability implies that the internal steps of execution, or the flow of both control and data, can be viewed by the programmer. Controllability allows the programmer to specify conditions which, when they arise, enable interaction with the executing program [Seidner83]. Both observability and controllability should be defined in terms of the semantics of the language in use [Evans66].

Execution monitoring tools are required because it is difficult to visualise the steps involved in program execution [Gramlich83].

> "-Whilst we, on the one hand, write and percept our programs as linear sequences of text, the actual path taken through the program by control during its execution is far from straightforward. It is often difficult to foresee all the possible ways in which control can sweep through the program and at the same time see all the consequences this can have for program results.
>
> -Not only that the actual development of program execution depends on the state of its data, it is the manipulation of data alone which is the main and only task of the program execution. Because of the great variety of ways in which the respective data items can be influenced, it is very difficult to assess the actual impact of individual

operations." [Babcicky80:11]

The implementation of execution monitoring tools can be divided into two broad categories: those developed for real-time software and those developed for non real-time software. In the case of real-time software, where the code is dependent on time, special hardware is used to extract information from the executing program and to exert control over it, without upsetting the time dependency. This hardware can take the form of simple machine bus probes [Fryer73] [Gentleman83] [Tsai90], complex circuitry which allows some form of monitoring at the abstract level of the high-level language [Goossens83] [Small85] [Rijks87], or even the use of a second computer, faster than the target machine [Plattner81] [Plattner84].

The implementation of execution monitoring tools for non real-time software is based on the fact that there is usually little or no support provided by the machine. Historically, execution monitoring was performed via post-mortem dumps, snapshot dumps and simple trace facilities [Evans66]. These often provided masses of data in hexadecimal, requiring knowledge of the machine and, in the case of high-level languages, the compiler. Even today interpretation [Evans66], preprocessors [Balzer69] [Cohen77] [Foxley78], or simple break-and-examine tools [Pierce74] [Atkinson78] provide the basis for most execution monitoring tools.

The high-level language programmer views his program at the abstract level of the language in use and, for this reason, any tool designed to aid in the development of a program should operate at that same abstract level. Simply allowing the programmer to use an assembly language tool via source symbols does not create high-level language tools. It is thus necessary to incorporate features such as recursion, procedure calling chains, execution paths and dynamic variables into development tools, and especially the execution monitor. Those systems which do offer a degree of high-level abstraction do so at the expense of execution speed, something which is not always tolerated by the programmer. On the other hand, real-time execution monitors do not delay target process execution but the cost of such implementations is usually prohibitively high.

1.3. Aims of this thesis

The aim of the work described in this thesis has been to investigate methods of providing execution monitoring facilities at the abstract level of the language in use, but without incurring either heavy execution overheads or requiring extensive hardware support.

The evolution of execution monitoring tools from early hexadecimal dumps and traces is given in chapter 2, as is a set of requirements for the monitoring of software and, in particular, high-level language software.

Chapter 3 describes a monitoring environment for software monitoring at the level of the source. This includes the variety of commands which can arise when monitoring high-level language software and also a description of high-level language features, such as procedures and recursion, and their effect on monitoring. A set of monitoring primitives is introduced which, with appropriate monitoring software, is capable of implementing high-level monitoring.

The software structures required by monitoring software and the algorithms employed to implement the execution monitoring facilities outlined in chapter 3 are discussed in chapter 4. A number of examples of possible monitoring scenarios are also given.

Chapter 5 reviews the current state of architectural support for execution monitoring and describes possible methods of implementation of the monitoring primitives of chapter 3. A technique for the implementation of the monitoring primitives which makes use of the virtual to physical translation mechanism, common to many multitasking machines, is described and its effect on execution speed discussed.

An experimental implementation of the monitoring software and architectural support described in chapter 4 and chapter 5 is described in chapter 6. This also includes a section devoted to the analysis of the performance interference caused by monitoring activity. Methods of overcoming much of this degradation are also described.

2. Execution Monitoring

2.1. Introduction

Execution monitoring is the viewing of the internal steps taken during program execution [Plattner81] [Burkhart84] [Plattner84]. This involves both the execution of instructions and the accessing and updating of program variables. Without the use of a technique to aid in the process of execution monitoring a programmer can only infer the internal workings of an executing program from the output it produces. For example, output during program execution indicates that the process has reached a particular point in the code, and the output of program variables gives their values at that particular time, but the workings of the rest of the process, the intermediate steps which make up the observable effects, are often a complete mystery.

In the testing, debugging and evaluation of programs the execution path is often required, something which is impossible with most programs which output only a fraction of the required information.

Many current testing methods involve some sort of structural testing [Howden75] [Clarke83] [Weyuker86] [Cantone87] [Ntafos88] or data flow examination [Huang79] [Girgis85] [Frankl88] [Clarke89] [Weyuker90]. In both these cases the adequacy of the testing technique can be determined by "watching" the internal steps taken during program execution. During structural testing, questions which require answering include: How many times did a particular loop iterate, which branch did a conditional take or, how many times was a particular procedure called recursively ? Data flow examination, on the other hand, requires knowledge of the sequence of assignment of values to variables.

The process of debugging asks very similar questions to that of testing and, again, these can be answered by an execution monitor. For example: When does a variable get assigned an incorrect value or, when does execution proceed along an incorrect path ? Knowing exactly where an error occurs usually leads to the reason why.

The evaluation stage of program development is concerned with frequency counts and timing statistics about the executing code. For example: How many times does the memory allocation routine get called or, which routine is responsible for the greatest amount of execution time ?

Execution monitoring is also useful when a programmer needs to understand a piece of code that he wrote some time ago or code which was written by someone else. It has been suggested that the best way of presenting the programmer with a complete understanding of a piece of code is via static analysis of the program [Tischler83] [Fairley85]. The reasoning behind this is that dynamic information concerns only one specific execution run, whereas static analysis explores and summarises all possibilities. However, all source code must be available to make use of this method and bugs in compilers or system software cannot be detected so easily. Thus, it is the view of the author and others [Barra83] [Ambras88b] that dynamic information has a role to play in the conveying of program workings.

Perhaps the most obvious and easiest implementation of an execution monitor is the insertion of monitoring statements into the program source code. This method of monitoring is proposed as early as 1951 [Wilkes51] and still has a following today [Winder88]. It has a number of applications, including testing [Huang79] [Lauesen79] [Huang80], debugging [Ferguson63] [Mann73] [Aral88a] [Winder88] and, program profiling and evaluation [Huang80]. The manual insertion of monitoring statements requires no special machine features or additions to the language translator/interpreter. Further to this, the programmer requires no extra knowledge of a separate monitoring language and can insert just enough code to perform the necessary tasks. The extra code added by the programmer for monitoring purposes would take the form of print statements, giving a trace of the flow of control, or values of variables. Performance evaluation is possible by the addition of extra variables, called *monitoring variables,* which can be used as frequency counters or timers.

Whilst the method of inserting extra code, as described by the above authors, is an effective and adequate monitoring technique it does have its flaws and limitations. In order to add monitoring statements the programmer must have access to the original source code and the resources for recompilation of the amended program. It is thus not possible to monitor library routines or production code for which the source is not available to the programmer, or code on machines for which the language translator is not provided. Secondly, because this method involves the insertion of code prior to recompilation and execution there is no provision for the flexibility of responding to earlier monitoring output. This flexibility means that a programmer, during the monitoring of execution, can, if he wishes, alter the objects being monitored based on the results obtained from previous monitoring statements. Whilst this may be feasible in an interpretive environment it is non-trivial for compiled code, although this approach has been implemented through the use of an incremental compiler [Fritzson83].

Like any other code the extra monitoring statements take CPU time to execute and thus the overall effect is to increase the execution time of the monitored, or *target,* program. Extensive monitoring, requiring the addition of many monitoring statements, can seriously degrade performance leading to user disapproval of the method [Johnson82]. More seriously, any method which leads to a relatively large performance degradation cannot be used in conjunction with programs which are in any way time dependent. Also, from the viewpoint of the user, the performance degradation must also include the cost of recompilation.

Apart from a performance degradation, the addition of monitoring statements can have other side-effects on the target program, not least of which is the introduction of errors. For example, it is possible, in some languages, to reference variables *anonymously.* This can occur, for instance, when array indexing is not checked for bound violations at run-time. Thus, accessing an element of the array outside of the declared bounds will anonymously reference other areas of memory. *Wild* addressing occurs when the correct value is obtained but from the incorrect location; thus, the program appears to function correctly but is logically incorrect. The insertion of extra monitoring variables can thus alter the functionality of the target program, which is to be avoided if the monitoring system is to be non-interfering.

The occurrence of the, perhaps, obscure conditions described above is not necessary for the introduction of errors. Simple typing errors can ruin a whole run, resulting in the inconvenience of corrections and recompilation. Similarly, the programmer, when inserting statements, must be careful to adhere to the syntax and semantics of the programming language in use. For example, amending the code fragment in figure 2.1 may be performed erroneously, giving the code fragment in figure 2.2. Although the print statement in this amended fragment appears to belong to the conditional unit, the actual structure of the code is shown in figure 2.3, with the print statement being executed regardless of the state of the boolean in the conditional.

```
if( boolean )
   statement1 ;
statement2 ;
```

Figure 2.1

When the errors outlined above are disregarded the method of inserting monitoring statements into the target source still has its limitations. Any code which passes through the language translator/interpreter must conform to the scope rules of the programming language in use; this restriction limits the kind of information which can be extracted from the executing program.

```
if( boolean )
   statement1 ;
   printf("branch taken") ;
statement2 ;
```

Figure 2.2

```
if( boolean )
   statement1 ;
printf("branch taken") ;
statement2 ;
```

Figure 2.3

Further to the above flaws and limitations the technique described is also prone to voluminous output. Monitoring statements in loops and frequently called procedures can lead to a lot of output, swamping the programmer. The code required to limit the amount of monitoring output can often be complex and involve the addition of code in more than one place. The addition of more code increases the chance of errors and also makes the cleaning up process more difficult and error prone. This cleaning up process involves the removal of the additional monitoring statements in order to create either a production version of the program, or a starting position to monitor other aspects of the program.

Some of the problems associated with the insertion of monitoring statements can be avoided if the target language provides the extensions for monitoring. Variable associations [Hanson76] and event associations [Hanson78] provide language extensions for SNOBOL4 enabling user-defined functions to be associated with the act of referencing a variable or the occurrence of an event respectively.

It can be concluded that the insertion of monitoring statements into the target source is far from adequate. There is need for a tool which gives the programmer an insight into the internal states of a process, but without the restrictions and problems outlined above. The tool must, therefore, work without the inconvenience of user-addition of statements and recompilation stage, and work outside of the normal restrictions of the specific language in use.

2.2. Classical Tools

2.2.1. Background

The first computer programs were written in assembly language; this having a direct relationship to the code executed on the machine. The method, outlined in the previous section, of inserting monitoring statements into the source code is, in general, highly unsuitable when appiied to assembly language programs. The first reason for this is that at the assembly level of programming the simplest high-level language construct (for example, a print statement) can be made up of many instructions. Secondly, there is often a requirement that certain registers and condition flags are preserved between statements; this is sometimes made even more complicated due to an often complex programming style.

Both the above make it difficult for a programmer (especially a programmer who needs to monitor code he did not write) to modify source in order to insert monitoring code. A tool is required which inflicts no interference on the preservation of information between instructions and is invoked simply and quickly. The first monitoring tools, developed primarily to aid in the debugging of assembly language programs, are now referred to as *classical* monitoring tools. Whilst primitive in their operation they implemented a much needed software development tool, and many of the monitoring systems available on present computers are still based heavily upon them.

In the early days of computing programming was performed at the machine console in an interactive manner. With the introduction of multi-user systems the programmer interacted with the machine via a job queue in a batch processing approach. Thus, two styles of monitoring systems evolved for the two differing methods of operation. *Post-mortem* monitoring systems were developed for the programmer in a batch processing environment where the program is submitted along with a complete set of input data to the job queue. Results from execution are delivered back to the programmer once execution has completed. *Conversational* monitoring systems were developed for the other mode of operation, that is, interactive at the machine console or, as developed later, in a time-sharing environment. Results from execution, in the interactive mode, are supplied to the programmer as they are produced and the input data given on demand. Thus, the input data can be modified according to program events.

2.2.2. Conversational monitoring system

The conversational, or break-and-examine, tool allows the programmer to interact with and extract information from the executing program. The same tools are described as "on-line" tools by Evans and Darley [Evans65] [Evans66] to distinguish them from tools for batch processing environments.

Conversational tools were first developed for monitoring at the computer console and evolved from the use of switches and lights [Evans66] [Johnson77]. The method of utilising console switches and lights involved altering machine memory and registers by setting the switches appropriately and observing execution through the console lights. A relatively modern approach to using this method is described by Hurst [Hurst84]. In this system the lights indicate execution within a range of addresses, and the switches allow the programmer to alter the ranges associated with the lights. Thus, the programmer is able to monitor program behaviour dynamically.

Classical conversational software tools often offer the following facilities [Evans65] [Bauer77]:

- Setting/resetting of code breakpoints.
- Examination/modification of memory/registers.
- Insertion/deletion of source lines.
- Search of memory for bit pattern.

Code breakpoints cause suspension of the user's program and return control to the software monitor when a specified location is reached during execution. The programmer is thus able to control program execution by the setting of code breakpoints at appropriate points in his code. Once control is passed to the monitor the examination and modification of memory and registers can be used to locate bugs or test that routines are functioning as expected.

Conversational systems differ in the way that commands are entered and in the control of the display. MONITOR [Gladstone76] allows memory addresses to be entered as hexadecimal numbers or ASCII characters with the added benefit of expression evaluation. Further to this is the ability to use indirect addresses and the user-defined symbols found in the assembly language source. The display, however, consists of only hexadecimal or ASCII values of machine registers and main memory. The programmer is restricted to the use of only four breakpoints, which the designer of the tool states is sufficient because multiple breakpoints only confuse the programmer. A facility not always found in conversational systems is provided by the trace command. With this operating, the values of the machine registers are saved after the execution of each instruction. Thus, a history of execution is available after execution has completed.

DDS [North77] is a similar tool to that of MONITOR but allows the user to specify that instructions are to be displayed in the mnemonic form as used in the assembly source. The display in DDS is updated automatically during execution with the update step defined by the user. This also displays the currently executing instruction. Both DDS and MONITOR offer a single step facility whereby the programmer regains control via the conversational tool after the execution of only a single instruction. This facility is only practical for examining small segments of a program due to the time involved; it is assumed that the programmer has narrowed down the area of the program to be monitored using the breakpoint facility. DDS also offers the programmer a facility that is not often found in assembly language monitors; namely a breakpoint on the accessing or updating of memory locations.

A tool called DEBUG [Evans65], based on TIC and DDT, offers facilities as described above but attempts to overcome problems associated with patching code in symbolic assembly language and the production of a "clean" version logically equivalent to the patched program.

Assuming that the machine hardware provides no support for the implementation of code breakpoints there are two main methods of implementation:

1. Replacement of the instructions at the specified address with a jump to subroutine instruction, the call address being that of the debugging routine.

2. Replacement of the instructions at the specified address with a trap instruction which causes an interrupt. The address of the debugging routine is picked up from the interrupt vectors, the transfer of control performed by the hardware or the operating system.

It is important that the execution of the program is not altered in any way by the interference of the monitor execution. For this reason all machine registers, which must be preserved between between instructions, are saved to a scratch area of memory prior to the transfer of control to the monitor. On specifying continuation of the program, control is transferred back to the calling point by restoring the saved resources from the scratch area and then executing the replaced instruction(s) before performing a return from subroutine or interrupt.

2.2.3. Post-mortem monitoring system

Post-mortem monitoring tools, developed primarily for batch processing environments, provide execution information for analysis when program execution has either ended normally or with a fatal error. Because program execution is activated as a job on a queue any input must be supplied prior to job initiation. This also applies to user-specified input for the monitoring routines. The transfer of control at code breakpoints is performed in the same way as described in the last section for conversational tools. However, instead of a routine which converses with the programmer, the post-mortem routines use the pre-specified commands from the programmer to extract the required execution information.

There are four main tools developed within this class of monitoring system [Ferguson63] [Bauer77]:

1. The post-mortem dump or system dump is probably the simplest of the tools to implement, and is usually used to gain information at the point of a fatal error. When a fatal error occurs during program execution, control is passed to the post-mortem dump routine which prints the values of machine registers and memory locations of the monitored

program at the point of the error. The output, however, is usually produced in either hexadecimal or octal with no attempt at conciseness.

2. The snapshot dump is similar to the post-mortem dump above, except that the output occurs as soon as a programmer-specified address is reached and not upon process termination. Usually the programmer is able to specify as many snapshot points as required and also the information to be placed in the dump. When a snapshot point is reached during execution a jump to the appropriate monitoring routine occurs and the desired information is extracted from the process memory and is output. Execution of the monitored program commences with the instruction replaced by the code breakpoint instruction. This tool relieves the programmer of the need to insert monitoring statements into the program source.

3. A trace facility produces output on execution of each instruction, usually within a programmer-specified range. The information produced in this case would typically consist of the program counter, other important registers and the instruction currently being executed. This facility is typically implemented by entering an interpretive mode which extracts the necessary information before realising the instruction in software. Interpretation is resorted to in this case as otherwise the monitoring system would need to set code breakpoints on every instruction.

4. The traceback monitoring tool indicates how control reached the current point of execution, the output triggered by an error. This facility

requires that appropriate information is extracted at instructions where a transfer of control occurs. The information required is usually the values of the machine registers. The traceback tool could also show where and to what value memory locations are altered.

2.2.4. Assessment

In giving a degree of insight into an executing program the classical monitoring tools release the programmer from the burden of inserting monitoring statements into the source code but do not solve all the problems associated with execution monitoring.

One of the greatest drawbacks to the classical tools is the amount and format of the generated output. This is often in hexadecimal or octal and can be extremely voluminous, making analysis of it difficult. Unless the programmer is able to specify exactly what information is to be displayed in a dump facility then the output will consist of the working space of the program. Even if the data generated in a dump can be tailored to the user's requirement, it is quite easy for the amount of output to exceed that which can be easily analysed. This can happen, for instance, with the snapshot dump, when the snapshot points, specified by the programmer, are executed many times, for example, in a loop or frequently used subroutine. This can be overcome by a condition attached to the snapshot point which indicates either a maximum number of times it can be acted upon or how many times it is ignored before being recognised. A similar method of reducing the volume of output can be applied to the trace tool, where only a specified number of executions of a particular location cause any output. This would be in addition to restricting the trace to between two specified locations.

The use of an interpreter for the trace facility causes severe performance degradation as the execution of each instruction is performed by software. This can be reduced significantly if the range specified by the programmer is the only area to be interpreted, with execution outside this area proceeding as normal. The post-mortem dump adds no overhead to program execution times and the snapshot dump can add very little if used sparingly.

The displaying of memory and addresses as hexadecimal or octal values can be very tiresome for the programmer as the view of the program during the monitoring stage is different to that of the coding stage, where symbols are used. It is thus helpful if instructions in memory can be displayed as mnemonic values and addresses be displayed as symbols used by the programmer in the assembly language source. The lack of readable output is even more confusing for the high-level language programmer [Satterthwaite72]. An increase in the use of structured high-level languages has led to an increase in the number of programmers who know little about the compiler for memory allocation and instruction generation, and even less about the workings of the underlying machine. It is also apparent that these programmers are not prepared to learn the skills necessary to use classical tools for high-level language monitoring [Ferguson63].

Perhaps the simplest approach to making the classical monitoring tools more appealing to the high-level language programmer is to use program source symbols in commands and output. Post-mortem dumps for high-level languages [Bayer67] [Satterthwaite72] show the failure point in terms of a "stack" of procedure calls and the values of variables local to each procedure invocation. The latter system for the programming language ALGOL W [Satterthwaite72] also provides a high-level language trace facility which displays the current point of execution as a source code line number. Also displayed are the values of any variables used in expressions. The performance is quoted as 50-150 times slower when tracing. Another approach to the problem of providing post-mortem dumps for high-level languages is performed by STABDUMP [McGregor80], a symbolic dump interpreter. The STABDUMP dump analysis program picks up the monitored program's symbol table information and the store image of the program at the point of failure, and returns values for all variables including the contents of data structures such as arrays and records. This is performed in addition to the "unwinding" of the procedure call stack. One advantage of this method over previous methods is that interpretation of the classical dump does not incur a performance overhead during program execution.

The use of compile-time symbol tables for the production of symbolic monitoring output has also been applied to conversational systems. DEBUG [Atkinson78] for the programming language BCPL is a break-and-examine tool which allows symbols to be used for breakpoints on entry to program functions. However, breakpoints of a finer granularity require a machine address, as does the examination and alteration of program variables. Thus, the programmer must have knowledge of the compilers memory allocation routine. DDS [Pierce74] is a similar break-and-examine tool with the extra feature of allowing symbolic names to be used to refer to program variables and line numbers to indicate particular source statements. The system, however, compromises its high level language functionality by regarding multidimensional arrays as vectors, the programmer having to work out the correct subscript to access an element. DDS also allows symbolic patching of the original source code.

The classical monitoring tools have also been improved upon for assembly language debugging and testing. ALADDIN [Fairley79] allows the assembly language programmer to set assertions that describe logical relations among various components of the program state. This feature is described as a "location independent breakpoint facility". Because ALADDIN must take control between execution of successive instructions, execution of the monitored program is interpreted, leading to a performance degradation of 100 or more. FADEBUG-I [Itoh73] is a module testing facility for assembly language programs. The programmer uses SET statements to prepare a set of input data and uses a CHECK statement to ensure that the correct results are obtained after module execution. The programmer is also able to obtain a listing of all the physical routes in a module from entry to exit. Graphbug [Davies86] provides a graphical display to a conventional conversational tool. The display can show areas of memory, register values and the next instruction to be executed; this is updated during execution but all values are in hexadecimal.

The classical monitoring tools, even with the modifications described above, are of little use to the high-level language programmer writing programs with high-level concepts such as procedures, recursion, local variables and dynamically allocated memory.

2.3. High-Level Language Monitoring

2.3.1. Requirements

Many of the requirements of monitoring systems were identified as early as 1966 [Evans66]. From this, and also subsequent studies including [Bauer77] [Lauesen79] [Tratner79] [Glass80] [Seidner83] and [Burkhart84], the foll^owing major themes emerge:

1. The monitoring system must provide the user with full flexible control over the execution of his program. This requirement is perhaps best described using the method of Plattner and Nievergelt [Plattner81]. A process, created by program execution, can be thought of as the trajectory of a point moving through space. The space, through which the point moves, is the state space of the process, and is a cartesian product defined by the program being monitored and the semantics of the programming language it is written in. This state space is a set of states which includes all potential states of the process, and deliberately includes states which will never be reached, as it is, in general, impossible to decide whether a given process will ever reach a particular state. Each state of the process state space consists of the following two components:

(i) A control component which reflects the active points of control. This ranges from the program counter in the simplest of languages to a dynamic stack structure for a more complex language incorporating procedures and recursion.

(ii) A data component which consists of all input data and internal data currently belonging to the process. Again, the complexity of this process state component depends on the complexity of the programming language.

The point describing a trajectory through the process state space indicates the current process state and moves through the state space according to the statements within the target program.

The requirement that the user is allowed full, flexible control maps onto the concept described above as the provision of a facility for the highlighting of a set of states. This highlighting produces two regions within the state space of the process. In one region, the monitoring predicate which defines the two regions, is false and in the other region it is true. Monitoring is thus the "watching" of the trajectory of the point as it moves through the state space of the process; the monitoring action is performed when the point crosses the boundary of the region yielding true for the monitoring predicate.

2. In order for the programmer to be able to fully observe the execution steps of the monitored program the entire working space of the process must be visible to him. This includes being able to see the current point

of control in terms of which source statement is currently being executed and the procedure invocations undertaken in order to reach this point; all program data must be accessible, even program variables which are currently out of scope and thus not visible to the executing code.

It may be the case that program execution up to the desired point of observation is expensive in execution time. The process of debugging, in such a situation, would require that as many bugs as possible were found in each run [Lauesen79]. For this reason it is desirable that the programmer can alter the values of erroneous program variables in order that execution can continue normally to find the next bug. This altering of the process workspace has other uses, including the setting up of variables for testing and evaluation purposes. As with the examination of the process workspace it must be possible for the programmer to alter any location within the entire state space including values of variables not visible to the executing code.

Further to the above facility of examination and alteration of program variables is the examination and alteration of the program source code. This can be implemented easily within an interpretive environment, as the source code provides the executable code directly. The same facility is obviously more difficult to implement when a translation phase is used, as the source code is not directly executed but produces executable code via the compiler. Within earlier monitoring systems it was often impossible to alter the code executed unless the programmer was prepared to delve into the assembler version of the high-level source

[Evans66]. A primitive facility would allow the programmer to insert patches into the executable code using source language terms [Pierce74]. This, however, does not, in general, give executable code equal to the code which would be generated from the amended source code. The use of an incremental compiler, where each source statement is compiled separately, allows the programmer to make alterations to the program source which are reflected immediately in the executable code [Fritzson83]. The automatic updating of the source is also discussed by Ferrante [Ferrante83]. Another method of implementing dynamic changes is to alter the source code, recompile it using the standard compiler and then change the current core image to incorporate the new code and data [Cook83].

3. Due to the proliferation of high-level languages and common run-time environments found on many machines it is possible to write program modules in different languages and then link them together to form a multilingual process. It would thus be beneficial to the programmer to have a single monitoring system to service all languages [Elliott82] [Beander83]. It would also prove more economical to write and maintain a single system, with the programmer having the benefit of a consistent interface [Victor77] [Hart79].

   The diversity of languages means that a language independent monitoring system in an interpretive environment is infeasible [Johnson78b]. Language independence is achieved to a certain degree through the use of compiled code, as the object code produced is the same for all languages. However, in order that the programmer can monitor programs using the symbols found in the source code the monitoring system must have access to the symbol table generated during translation. There are two approaches to supplying a language independent monitoring system with the necessary generated symbol tables. Either a common format symbol table is generated for all languages enabling the monitoring system to access them in a consistent manner [Beander83] [Cardell83] [Walter83] or else, the monitoring system has a number of language interfacers, one for each language it supports, which access the symbol table in the required way and pass the information to the monitoring system in a consistent manner [Victor77] [Johnson78b].

Language independence by itself, however, is of most benefit to the designer and developer of the monitoring system rather than to the programmer performing the monitoring. In order to specify monitoring commands for a number of different languages the programmer would need to use a monitoring language which looks like none of the individual languages but incorporates features from all of them [Elliott82]. This would be both unnatural for the programmer (writing programs in one language and monitoring them in another) and also require the learning of the new monitoring language. A better technique would be to make the monitoring system language independent but appear language dependent to the programmer; this we call language sensitive [Johnson77] [Goodman82] [Beander83]

[Gramlich83]. For each language supported the monitoring system would need to know the syntactic and semantic rules of the language including: scope rules, referencing of structures, and the procedural, arithmetic and conditional logic which makes up language expressions.

A problem arises when programming languages are used as the command language for the monitoring system in that most languages do not possess features to enable the specification of, for example, breakpoints. Because of this the language sensitive command language of the monitoring system needs extensions for unsupported monitoring facilities [Ashby73]. It is possible and, in fact, desirable that these extensions are common to the entire set of supported languages, giving a natural and consistent view of the system to the programmer. The two extremes when providing a set of language extensions for monitoring are reflected by the UNIX debugger cdb [Cdb(1)], which provides a large number of single and two letter commands, each providing a different monitoring facility, and DISPEL [Johnson81] which looks like an algorithmic programming language and is provided for the writing of routines from primitives, which are then called upon when required.

4. Program optimisation during program translation retains the functionality of the original program but can alter the structure or intermediate results to save time and/or space [Hennessy82]. The ability to monitor optimised code is advantageous for a number of reasons. Firstly, it can be the case that it is impossible to obtain a working, unoptimised version of the program. Reasons for this include

the compiler performing certain optimisations during normal operation, even with no optimisation specified, and the code being optimised because of timing or size constraints. Secondly, even if it is possible to generate a special unoptimised version of the code it may be the case that the error is no longer apparent, due to timing or structural differences.

Optimisation occurs in many forms but usually involves the elimination, duplication or relocation of code; the elimination or relocation of variables; or the simplifying of subexpressions [Ferrante83] [Seidner83] [Zellweger83] [Richardson89]. It is the altering of the code in this way that affects source level monitoring; the correspondence between the source program and the optimised executable version is often very complex. Problems occurring during the monitoring of optimised code include:

- Trying to set breakpoints on removed code (monitor would respond with no such code).
- Tracing of relocated code (trace would show code in different order).
- Resuming execution at duplicated code (execution can resume at a number of different points).
- Tracing relocated or eliminated variables (expected references and/or updates would be missing from the trace).

There are two methods of overcoming the problems of source level monitoring of optimised code. Either the monitor exhibits: (i) correct behaviour, or (ii) transparent behaviour. Correct behaviour means that the monitor responds with, in source program terms, the relevant changes caused by the optimisation at the execution point. The programmer would, in this case, receive messages from the monitoring system informing him of, for example, code movement or the removal of variables. A better solution, transparent behaviour, responds as if the program were compiled without optimisation. Thus, the programmer does not see the effects of optimisation during monitoring. Navigator [Zellweger83] attempts to provide transparent behaviour for inline procedure expansion (code duplication) and cross-jumping (code elimination). It does this by replacing the usual tables generated during translation with two tables; one mapping source code to object code and, the other, object code to source code. The problems of optimisation and, in particular, code motion and variable relocation are discussed by Hennessy [Hennessy82], as are algorithms for overcoming them.

The objectives of the above requirements are to create a monitoring system which is able to monitor in terms of high-level concepts but does not require the programmer to learn a new language or large set of commands. Other features which can improve a moniioring system include a user interface driven by windows and menus and the ability of the monitoring system to be applied to a program at any point, that is, before execution has started, during execution, or even after a fatal error has caused termination.

2.3.2. Monitoring with interpreters

Interpreter based systems have been, and still are, a means of implementing monitoring facilities for high-level languages. The reason behind the relative ease of implementation lies in the fact that execution of the target program is performed by software. Thus there is no complicated hardware to interact with in order to provide even simple facilities, such as breakpoints. The monitoring software required in order to implement monitoring tools can be inserted into the language interpreter at the appropriate place. For example, a facility for the tracing of updates to a particular variable can be implemented simply by checking identifiers within the code which emulates variable assignment. This same feature in a translator environment is non-trivial to implement as many machines do not provide facilities in hardware for the trapping of updates to memory locations and, consequently, many monitoring systems resort to either machine instruction single stepping or interpretation.

In the 1960's debugging features appeared for on-line LISP implementations for the MAC time sharing system, the M-460 system, the SDC time sharing system, the Berkeley system, the DEC PDP-6, and the DEC PDP-1. The QUICKTRAN system implements debugging tools for interpreted FORTRAN programs [Evans66].

The MAC and M-460 systems made the tracing facilities of batch processing systems available to on-line users, with an extended capability of making the tracing conditional. Along with this was a program which allowed the editing of the internal representation of LISP programs, permitting the modification of the program, and thus, by way of inserting control transfers, implementing breakpoints. This editor uses the LISP language itself for adding extra code making the addition of complex conditional monitoring events easy compared to assembly language systems.

The QUICKTRAN system allows modification of the FORTRAN program and facilitates non-conditional breakpointing by including a statement which, when reached, transfers control to the user. As with the LISP systems tracing is featured; in this system on assignment and control transfer. Further to this are diagnostics to inform when code is never executed, variables never set, or variables never used.

Monitoring systems implemented in an interpretive environment do not all date from the 1960's. More recently interpretation has been used to implement a debugging facility for the Chill Compiling System by providing virtual machines through high-level emulation of hardware independent code trees [Goodman82]. MicroScope [Ambras88a] [Ambras88b] builds a data base on a LISP program and allows the programmer to monitor execution, display data changes and control flow dynamically with execution performed via interpretation.

Two major problems arise when monitoring is implemented using an interpretive environment: the introduction of errors and the degradation of execution performance. Both of these are, however, of minimal importance if the environment used to implement monitoring facilities is the same environment which is used to run production versions of the program. Thus, the problems only manifest themselves to a significant degree if interpretation is resorted to for monitoring only.

The switching from a translator environment to one involving interpretation introduces a significant performance overhead because the desired effects are realised in software rather than directly in hardware. This is something which the programmer may not be willing to endure and is unacceptable for software which must conform to timing constraints.

The second problem which arises with the use of interpreters is the introduction of erroneous behaviour. The writing of an interpreter for a given language is a non-trivial task and there is a chance that programs interpreted will vary in functionality from the corresponding translated code. Errors occurring in program execution may thus be limited to only one of the environments, leading to mistrust in the system [Mikelsons83].

2.3.3. Monitoring with preprocessors

A high-level language monitoring system can be implemented by passing the source code through a preprocessor which inserts monitoring statements at the appropriate places. The monitoring statements which are added can perform simple tasks such as recording entry to a particular procedure or more complex actions requiring conditionals based on program and monitor variables.

There are two approaches to monitoring with automatically inserted monitoring statements via a preprocessor. Either the monitoring statements output information during execution, which are then analysed by the programmer in order to find program bugs or ascertain correctness, or the extra monitoring statements output information to a database which can then be interrogated by the programmer, using an inquiry language, after execution has terminated.

Ctrace [Steffen84], BUGTRAN [Ferguson63] and two other systems [Arisawa80] [Clark83] are examples of the first variety. Ctrace is a preprocessor for the C programming language which prints the executing statement, in the form of a source statement, and also the values of variables the current statement uses or modifies. This particular tool is proposed as a portable monitoring tool because all monitoring code passes through the normal C compiler and is thus translated with the rest of the program into appropriate machine dependent code. A graphical system for Pascal [Clark83] performs essentially the same function as Ctrace except the information is presented in the form of diagrams based on Nassi-Shneiderman charts.

BUGTRAN is a tape-to-tape prepass debugging aid for FORTRAN. This system differs slightly from the previous monitoring tools in that the user inserts monitoring commands into the source, which are translated into appropriate FORTRAN statements by the preprocessor. The programmer, using BUGTRAN, can specify the type of checking or information required and also restrictions on when the BUGTRAN statement is effective. Information available includes the printing of variables updated in expressions, statements executed, snapshot dumps or entry and exit to and from subroutines or functions. This can be restricted to a region of the program, defined by statement numbers; at a particular depth of subroutine call; and whilst an arithmetic expression yields a value true. A similar system called STAR [Arisawa80] uses language extensions to implement recursive subroutines, as well as assertion, variable dump and snapshot facilities. Particular attention is paid to the reporting of messages in terms of the original source program and not the version after the preprocessor stage.

Of the systems which collect information for presentation upon termination, one of the simplest is probably SCAMP [Foxley78]. This is a profiling system for ALGOL68 R programs. Given a syntactically correct program an amended program is produced which when executed gives frequency counts for five ALGOL68 constructs: blocks, routine bodies, **if**, **case** and **do** statements. Restrictions placed on the source by the preprocessor include: **if** and **case** clauses must be written in the full form; externally defined and library routines must be called indirectly, via dummy procedures; but implicitly called system routines cannot be monitored. The increase in execution is estimated to vary from 5 per cent to 100 per cent. Frequency counts can be a help in the improvement of efficiency in code but do not show any kind of control or data flow from the executing program.

Another approach to providing monitoring facilities via a preprocessor is to create a database or history tape of a program's execution and then allow the programmer to access it using an inquiry language. One such system for ALGOL60-like programs [Cohen77] uses a source translator with the resulting instructions interpreted. The database created consists of an array of *label trajectories* which hold the information on the label code, the variables updated and the operators used. The inquiry language can then be used to answer questions such as: what is the value of a variable at a particular label and at what label does a variable attain a certain value ?

A similar system, EXDAMS [Balzer69], was developed to satisfy three requirements: to test proposed, but unimplemented, debugging and monitoring facilities; as an extendable facility to which new monitoring aids could be added easily; and as a system to provide independence of a particular machine, the implementation of the language, and also the source language used.

EXDAMS is a four-phase system: program analysis, which creates a model of the program source (that is, all the static information), and inserts the necessary monitoring statements to create a history tape at run-time; compilation of the modified source by the standard translator; run-time history-gathering where the compiled code is executed (the inserted statements building the history tape); and debug-time history-playback to respond to programmers monitoring requests.

Included in EXDAMS is *flowback analysis,* a facility to aid in the debugging of programs. Given a particular value this shows how execution proceeded to produce the specified value; appearing in the form of a tree, each node representing a source language assignment, the links giving the nodes of sub-expressions.

The addition of new monitoring aids requires only an addition to the command language, the appropriate code to request information from the *information retrieval* routine, and process the information for displaying to the programmer. Balzer attributes the attractiveness of EXDAMS to the ability to run programs at variable speed, in either direction, together with the ability to stop execution, switch monitoring tools, and continue.

Systems like EXDAMS which create history tapes or databases are I/O bound; the considerable amount of information being placed in these tapes causing a severe performance degradation. A solution put forward by Cohen and Carpenter [Cohen77] is for the programmer to specify which program variables or operators need to be stored in a given run. EXDAMS tries to reduce I/O by using the model to interpret what has been stored in the history tape as opposed to storing both values and identifiers of variables.

The problems associated with monitoring systems, based on preprocessors, are basically the same as those for the manual insertion of monitoring statements. The use of preprocessors does, however, reduce the problem of errors in the actual monitoring statements and also solves the problem of removing monitoring statements after a session of monitoring. A restriction which arises with any system that applies pre-specified monitoring requests to an execution run is the inability to alter monitoring requests, variable values or the point of control during execution.

2.3.4. Conversational monitoring systems

Simple monitoring systems developed for high-level languages implemented on small machines [Pierce74] [Atkinson78] are little more than the classical conversational tool described in section 2.2.1. These systems allow code breakpoints on source statements or program procedures and enable the programmer to examine or alter program variables. However, it is often the case that the programmer must know the memory allocation and code generation techniques of the compiler. Due to the decrease in cost of computer memory it has become more and more feasible, over the years, for larger, more complex monitoring systems to reside on machines and also for the storage of translation information generated by the compiler, which would normally be discarded once translation had completed.

Source level, conversational monitoring systems, allowing programmer interaction with an executing program are now commonplace on many machines. Dbx [Dbx(1)], found on 4.2BSD UNIX machines, provides symbolic debugging for C and FORTRAN programs. The programmer is able to trace source statements, procedure or function entry, and variable update. It is also possible to restrict the tracing to a particular procedure or function invocation, as well as associating a condition with the monitoring command which must evaluate to true for the information to be reported. Halting of the target process is specified in terms of a source statement, procedure or function entry, or variable update, all with an optional condition which must yield true if a halt of the target process is to be performed. Other commands enable the single stepping of source statements, as well as the displaying and altering of program variables. A similar system to Dbx, called Cdb [Cdb(1)], provides essentially the same facilities as Dbx, with extra facilities for the monitoring of procedure invocations at different depths on the procedure call stack. However, the procedure invocation on the stack is specified by an absolute value which restricts the ability of the programmer to perform complex monitoring associated with procedure calling chains. Most commands have an optional command list associated with them, enabling commands to specify further commands, and thus allowing more complex monitoring, although the programmer must handle all the levels of control himself. The command language of Dbx uses meaningful words such as "stop" and "trace", whereas cdb uses only single or two letter commands which can, in the opinion of the author, be easily forgotten making the system tiresome.

It is the author's experience that one of the easiest and most useful monitoring systems is VAX DEBUG [Digital86]. The command language consists of meaningful words with consistent qualifiers. As with cdb, commands can be associated with breakpoints, thus allowing complex monitoring predicates, although again, the programmer must handle housekeeping duties. For example, local variable monitoring requires that a breakpoint is set on entry to the appropriate procedure. The action of this trap is to calculate the absolute address of the required variable and to set a trap on updates to this location. At the same time a breakpoint must trap the exit from the procedure so that the variable update trap can be deleted. VAX DEBUG allows for the accessing of out of scope variables via a facility which qualifies identifiers with their enclosing blocks. None of the above systems, however, provide easy to use facilities for the required monitoring control described in section 2.3.1.

A monitoring tool which has a significant following is *reverse execution*. The programmer using a reverse execution tool is able to execute up to a point in the program where a fault or other event becomes apparent and then execute the code in reverse to see exactly how the current state came about. This is similar to flowback analysis except the program states are not simply recorded for analysis but are stored so that a previous state can be restored in order that reverse execution can be performed interactively during a program run.

IGOR [Feldman88], Recap [Pan88] and the reverse execution tool described by Zelkowitz [Zelkowitz71] use checkpointing to implement reverse execution. This technique involves saving the current process state at regular intervals which can be restored in order to achieve a "backing-up" of the process. For this to work only those parts of the process address space which have changed need be checkpointed, thus saving memory space.

A different approach is discussed by Kishimoto [Kishimoto83a] [Kishimoto83b]. Here a programming environment provides a reverse execution facility by obtaining the necessary information from a database which allows the different tools of the environment to communicate with each other, and is updated by a technique called data-driven symbolic execution which supplies both normal and symbolic execution results. The use of a relational database to support software development is also described by Powell [Powell83]. Debugging of programs is thus reduced to the performing of queries on the database, where all execution information is stored.

Recap, mentioned above, is primarily for use with parallel programs. This is an area of program monitoring which has received increased attention in recent years. The Parasight debugging system [Aral88a] [Aral88b] monitors programs in a shared-memory multiprocessor by creating observer programs, called "parasites", which dynamically patch jump operations, called "scan points", into the executing binary, to bring about the necessary transfers of control. GRIP [Venables89] enables the "watching" of Occam channels, displaying the results graphically using a folding display. This in effect provides a method of altering the area, and consequently the granularity, of the view. The Parallel Program Debugger (PPD) [Miller88] uses incremental tracing to provide a system based on flowback analysis. Incremental tracing is achieved by two logs generated during execution: a prelog and a postlog. These are implemented by appropriate code generated by the compiler/linker and indicate those variables which will be accessed in the block about to be executed, and the values of those variables changed during execution of the block.

A debugger for the MuTEAM language [Baiardi83] uses behavioural expressions constructed from event specifications to allow the programmer to monitor concurrent programs. The event specifications enable the programmer to specify communication events, process termination events, or variable update events, for which monitoring action is required. Events are also used in DISDEB [Lazzerini86] to enable interactive debugging on a multi-microprocessor system constituting a node of the Selenia Mara architecture. The event specification identifies a process and a memory location or input/output channel for which a supplied value range must hold true for the monitoring action to be performed. A different approach is undertaken by Voyeur [Socha88]. The Voyeur prototype supports graphical visualisations of parallel program executions by allowing the parallel programmer to easily construct application-specific, visual views of parallel programs. *Algorithm animation* [Feldman89] also provides facilities for an application-specific execution view.

Research into the area of program monitoring has not focused entirely on the problem of providing high-level control over program execution. One area of research into making monitoring systems more useful and easier to use looks at multilingual capability. Possibly the easiest approach to multilingual monitoring is to provide a *new* monitoring language which possesses features from many languages but resembles no single language in particular. This is the technique used by Elliott for the monitoring of PL/I, FORTRAN and BASIC [Elliott82]. However, the monitoring system appears more natural to the programmer if the monitoring language changes to reflect the programming language being monitored. Systems which provide a language sensitive facility include RAIDE [Johnson78b], AIDS [Hart79] and VAX DEBUG [Beander83] [Digital86]. The first of these, RAIDE, enables language interfacers to be attached to the monitoring system so that further programming languages can be monitored. An algorithmic monitoring language, called DISPEL [Johnson81], allows complex monitoring procedures to be written from a minimal set of primitives, and via a virtual machine called SPAM. AIDS and VAX DEBUG also provide language sensitive monitoring but executing compiled code on the actual target machine. Other multilingual systems which monitor compiled code without the need for specially altered source includes: DELTA [Walter83], SWAT [Cardell83] and DAD [Victor77]. Of these DELTA and SWAT achieve this via a common symbol table format, whereas DAD provides language interfacers along the lines of RAIDE.

Research into graphical interfaces has attempted to show that current monitoring facilities are approximately the correct ones, despite being low level ones, and that a good interface is a necessity for effective monitoring systems [Winder88]. Joff [Cargill83] [Cargill85] is a graphical debugger for C programs on the Blit, a multi-processing bitmap terminal. Windows, or *layers*, are associated with different processes and are used to separate the different classes of information; for example, source code, program output, monitoring output. In addition a pop-up menu system allows the programmer to interact with joff with little need to resort to use of the keyboard. The processes which control terminal activity run asynchronously to the target program on the host machine and thus receive information by downloading.

The graphical interface to the Ups debugger [Bovey87], for C and FORTRAN, provides a window display with a set of "buttons" for activities such as quitting the debugger or obtaining help. The menu system is a *postfix* system whereby the menu of available commands depends on the object selected. This helps to reduce the number of commands in the menu.

Dbxtool [Adams86], a frontend interface to dbx, provides five windows and six command "buttons". The five windows provide the current state of the debugging process (for example, target file name and line number), the current locus of execution, the current setup for the command "buttons" (which can be altered to suit a particular monitoring task), a command dialogue area, and a display of values of selected variables. A similar approach is taken by JDB [Winder88]. Again this is a frontend interface, but in this case to sdb. The window system offers very much the same facilities as dbxtool, the difference between the two systems being in the adaptability of JDB. There are three levels to the system: beginner, intermediate and expert. At the beginner level all interaction is via the menu system with a tree structure of commands, the object being that the programmer is led, by the system, through a hierarchy of options. At the other extreme, the expert level, commands are entered via the command line window with the menu offering a fast access facility to the most common commands.

2.4. Summary

The execution monitoring tool provides controllability and observability over the target process, thus enabling the programmer to view the internal execution steps taken by a program. Execution monitoring has applications in program testing, to establish test data coverage and to uncover data flow anomalies; in program debugging, to locate and identify program bugs; and in performance analysis.

Implementations of execution monitors have varied over the years. Still widely in use today is the method of inserting monitoring statements, written in the target language, into the program. This, when compiled and executed, outputs the desired information. However, problems and limitations have led to other methods of monitoring.

The classical monitoring tools, developed primarily for assembly language programmers, enabled the programmer to "dump" the required process state information and, when using conversational tools, to execute the target program with traps which cause suspension of the target process and the transfer of control to an interactive facility allowing the examination of the process state. Although these tools are satisfactory for assembly language programs they are unsuitable for high-level language programmers.

Methods of implementing high-level language monitoring systems have included the adaptation of interpreters, the use of preprocessors to automatically insert monitoring statements into the target program, and the use of a second process to monitor the target process. Problems and limitations are to be found with all of these methods, but we believe that the use of a second monitor process controlling the target process offers the possibility of high-level and non-intrusive monitoring. Current systems which use a monitor process have attempted to increase their usefulness by making them easier to use.

Because many programming environments allow multilingual processes, monitoring systems have been designed to be language independent, thus allowing a single consistent system to be used for all languages. An additional approach to language independence is language sensitivity where the monitoring system is language independent but appears language dependent. Systems now exist which provide a language sensitive facility either by a common symbol table format or via a set of language interfacers.

To avoid the learning of an often large and complex command language graphical user interfaces have been added to monitoring systems. These consist of windows to split the different classes of monitoring and program information, and the use of menus and buttons to allow the easy entering of monitoring commands. However, adding a graphical user interface to a simple break-and-examine monitoring system is not sufficient for the monitoring of high-level language concepts such as procedures and dynamic variables. Rather than placing the onus of translating high-level monitoring requests into machine-level traps onto the programmer this could be made part of the control structures of the monitoring system. Most current systems still only offer a break on source statement execution or entry to a specified procedure. More complex monitoring requirements must be handled by the programmer. An assertion facility is to be found on a few monitoring systems, allowing the programmer to monitor full expressions involving program variables. These are often implemented, however, at a performance cost by a mechanism such as machine instruction emulation or the trapping of execution after every instruction.

The problem of performance degradation is one which few researchers, outside of the real-time environment, have confronted. However, if a practical high-level language monitoring system is to be implemented then it is essential that the system does not incur an unacceptable performance overhead.

From the above, the state of the art in execution monitoring is defined by conversational tools which require no special software *hooks* within the target source, assertion facilities allowing quite complex monitoring expressions, and a graphical interface which shields the user from the command language. The significant issues not fully addressed in current systems include the performance degradation incurred by complex monitoring expressions, completeness in dealing with language constructs, and language sensitivity. We will go on to address these issues by considering the structure of a generalised high-level monitoring environment.

3. Monitoring Environment

3.1. Introduction

In the previous chapter (section 2.3.1) the requirements of an ideal program monitoring system were identified. Basically, these requirements consist of a high-level, language sensitive monitoring command which provides full control over target program execution, and the displaying of the target process state in a high-level, language sensitive manner. The ability to port the monitoring system to different machines would benefit both the implementor and user; the system implementor need write only one portable version of the system and the user benefits from a consistent monitoring system across all machines. Whilst the implementation of a fully portable system is unrealistic, the implementation of a modular design would enable as much of the system as possible to be ported to another machine.

In this chapter we examine a possible organisation for a monitoring environment with these aims. We postulate a division of the environment into three units: The *user interface* unit (UIU), *process control* unit (PCU), and *machine control* unit (MCU). These units interact as shown in figure 3.1.

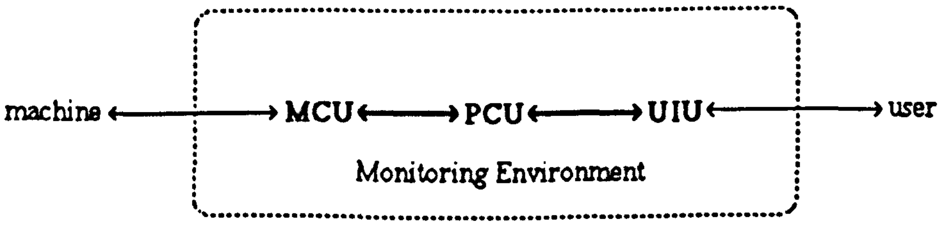

Figure 3.1

The UIU performs two functions: the interconversion of information between a language sensitive form and a language independent form, and the exchange of information with the user. Because compiler writers introduce non-standard and machine dependent facilities, the implementation of language sensitive monitoring commands requires a separate language interfacer for each language compiler supported [Victor77]. Using the appropriate language interfacer the UIU transforms language sensitive commands into a corresponding language independent version, which is passed to the PCU. Similarly, process state information obtained from the target process is passed, in a language independent form, to the UIU from the PCU. This can be converted into a language sensitive form by, again, use of the appropriate language interfacer (figure 3.2).

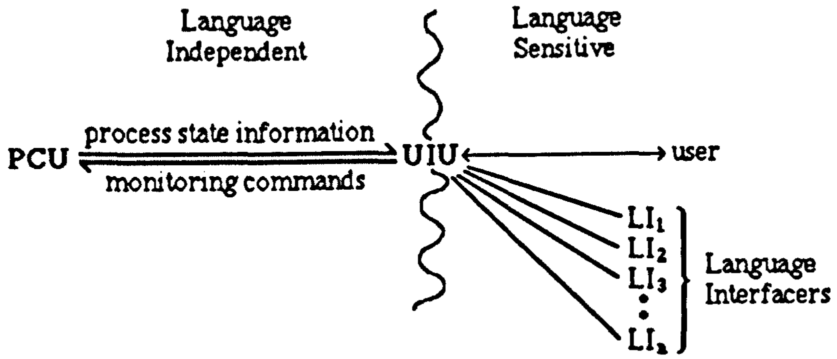

Figure 3.2

The second function of the UIU, the exchange of information with the user, includes the output of program source statements and variable values, the output from the executing target program, and the input of monitoring commands from the user. Modern graphical displays can greatly simplify the task of using a monitoring system through the use of windows for the partitioning of different classes of information, and the displaying of options in menus.

The PCU performs two functions. Firstly, it takes language independent monitoring predicates from the UIU and controls the execution of the process, "watching" for the predicate arising, using functions in the MCU. A second task of the PCU is to extract required information from the current process state and pass it to the UIU in a high-level but language independent form.

The first task, "watching" for high-level monitoring predicates, involves following target program execution which, in general, requires duplication, within the PCU, of those control structures of the target process which affect the monitoring predicate. It is the PCU of the environment which performs translation of program symbols into machine addresses. It is thus necessary that all relevant compile-time information be retained [Johnson79]. The four classes of information required are:

(i) descriptions of all symbolic data

(ii) descriptions of all code segments

(iii) descriptions of all optimisations

(iv) the source program conveniently, but not necessarily, broken into lexical tokens.

The generation of an internal symbol table via a common format compile-time table is largely infeasible due to the use of compilers from many different sources, for which the compiler writers adhere to no standard symbol table format. It must therefore be a function of the UIU to allow the accessing of the required symbol table information via the appropriate language interfacer.

Because the target program executes directly on the target machine, via a translation phase, all control of the target process must be performed at this same machine level. Thus machine information obtained from the program symbol table is used to invoke monitoring primitives within the MCU.

The second task of the PCU is the extraction of process state information from the executing target program and also the altering of that state information. At a higher level this is the examination and alteration of program variables. Because the method of one process extracting process state information from a second process varies between machines, portability requires modules for interfacing machine dependent portions of the monitoring environment. In most cases this will be the relevant operating system calls.

The MCU takes machine addresses from the PCU and applies them to the requested monitoring primitive, which "watches" either control flow or data flow. The interaction between the units is shown in figure 3.3.

Whilst the above discussion of the monitoring environment units is implementation independent, particular implementations appear more appropriate than others. The implementation is also influenced by the necessity for performance degradation to be kept to a minimum.

The functions of the UIU favour an implementation in software, with any

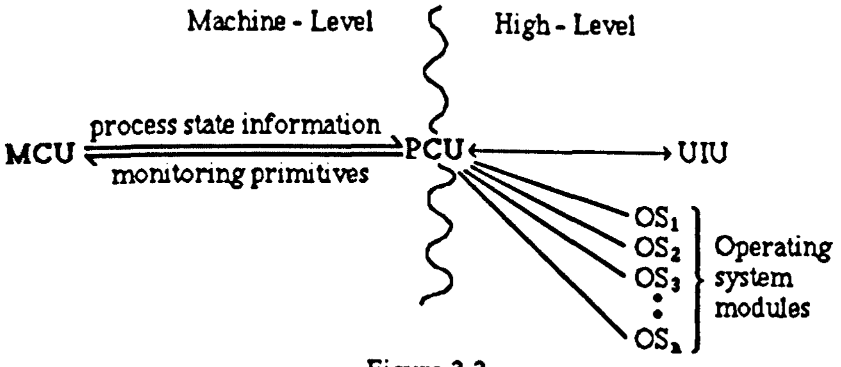

Figure 3.3

graphical facilities undertaken in an appropriate graphical management system. The problem of performance degradation is not critical for the user interface; in most cases user input will govern the performance overhead of the UIU, and the cost of performing the language conversion will be negligible in relation. The functions of the PCU also suggest an implementation in software as the duplication of the control structures of the target process, in hardware, would be prohibitively expensive. The implementation of the MCU, however, is more critical. The machine level traps which arise from the functions of the MCU incur a continuous monitoring overhead. This is the performance overhead imposed upon each statement executed, and is required to be kept to a minimum. It would be therefore be advantageous for the MCU to be implemented in hardware or firmware. This would also be advantageous because the process interactions which the MCU is attempting to "watch" also occur at the hardware or firmware level. However, machines without the necessary monitoring primitives implemented in hardware or firmware would still be able to carry the monitoring system if the appropriate functions were available in software (figure 3.4).

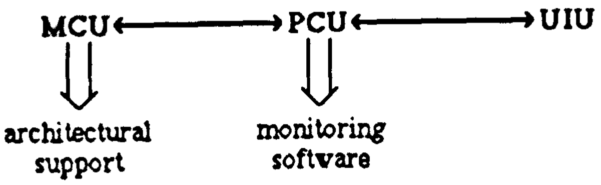

Figure 3.4

Modularisation of the monitoring environment can also ease the problem of monitor interference. The target process may, for example, be running on a relatively small microprocessor with limited memory, and by moving as much of the above monitoring environment to a connected machine there will be less target machine memory claimed by a monitor. Similarly, the running of the target program and the monitor in parallel, on two processors, would decrease the execution overhead of monitoring. The running of the two processes in parallel is, however, limited by the necessity for synchronisation between the processes. The setting of traps and the inspection of the target process state, resulting from previously set traps, should occur with the target process halted, thus preventing a state change before the required action. If the connection between the two processes was asynchronous then incorrect values could be extracted from variables or traps set too late to catch a particular machine-level event.

On: method for the implementation of parallel monitoring makes use of a FIFO queue and a phantom memory [Plattner84]. Memory transactions are recorded in the FIFO queue, which connects the target processor with the monitor processor. A phantom memory duplicates the target memory by accessing the FIFO queue for memory changes. Any access of the target process state, by the monitor system, is performed on the phantom memory whilst the FIFO queue is locked. Once monitoring activity has completed, the queue is unlocked and the phantom memory once again updated. This method obviously restricts the monitor to examination of the process state only, preventing the alteration of the target memory space. A further problem with the above method is the need for a monitoring machine which is at least as fast as the target machine and with enough memory to implement the phantom memory.

In the following sections we examine each of the units of the monitoring environment described above.

3.2. User interface

The user interface exists to enable communication between the user and the PCU or monitoring software; taking commands, in the notation of the target programming language, from the user and supplying the monitoring software with a language independent version.

In general the monitoring software system requires a language independent monitoring command of the form:

WHEN <monitoring condition> **PERFORM** <monitoring action>

The *monitoring action* indicates the user requirements when the monitoring predicate is satisfied. It may specify, for example, statistics gathering, data value recording, user notification of predicate satisfaction, or the halting of the target process and the transfer of control to a conversational tool allowing the user to examine and possibly alter the current state of execution.

The *monitoring condition* defines the point of execution at which the monitoring action is to be performed. In general the point of execution involves a control component which specifies the necessary flow of control through the target program, and a data component which indicates the required values of program data. In this section we examine the range of predicates the user may wish to specify and suitable notation for the language independent command. We do so not in order to define a monitoring language as such, but rather so as to identify language-related problems which will influence the design of the process control unit.

At one extreme, the monitoring predicate could be specified by the user giving ranges of values for all of the N variables in the N dimensional state space. This is, however, totally impractical for the large values of N arising from even modestly sized programs written in a modern high-level language. A more practical approach is to identify those constructs, within high-level languages, which require support for monitoring, and to provide a suitable notation with which they can be incorporated into the monitoring condition of the above WHEN command.

There are two categories of constructs which require monitoring support: textual or static constructs, and dynamic constructs. The textual problems include uniquely identifying all variables, data structures, arrays and pointers, and the dynamic constructs consist of procedure-calling chains, recursion and unpreserved state information.

The principal problem in defining static constructs for monitoring in a high-level program is in distinguishing program objects with the same identifier. The occurrence of the variable identifier "x" in a monitoring condition involves a degree of ambiguity. There may be many instances of the variable "x", declared globally, local to procedures or in some languages any program block, all quite distinct from one another. In a broader sense this is a problem of specifying out of scope variables or uniquely identifying all program variables.

It may be observed that the correct variable is always used during execution and so the language compiler must "know" which variable to use. However, compilers usually only refer to variables currently in scope, with the declaration of two variables with the same identifier, in the same block, causing an error. If a monitoring system were to adopt this approach then all references to variables could be unambiguously specified simply by the appropriate identifiers. However, a variable which is not currently in scope is still part of the programmer's abstract view of execution and thus, in general, it would not be unusual for the user to want to examine, alter or involve in a monitoring condition variables which are out of scope.

Specifying out of scope objects can be achieved by qualifying each program identifier with its surrounding block. The usual cause of scope restrictions is the inclusion of procedures in a program and so the identifier of the procedure becomes the obvious choice of qualifier [Bruegge83a]

[Bruegge83b] [Plattner84]. For example, a variable **X** local to a procedure **procA** is denoted by **procA/X.**

The textual nesting of procedures, giving rise to procedures local to other procedures can also be denoted by the above mechanism. **procA/procB/Y** specifies the variable **Y** local to procedure **procB** which, in turn, is local to the procedure **procA.**

Languages also exist which allow objects to be declared local to unnamed blocks, some allowing declarations at any point in the code. In these cases the unnamed part of the unique identifier is filled by a line number/statement offset pair. The line number relates to the position within the source code of the beginning of the region of scope; the statement offset is appropriate when more than one statement occurs on the line in question. This assumes that translation phase information includes the relevant object code mappings for these high-level concepts. A shorthand notation could be adopted to reduce the length of the monitoring command. One approach would be that simply specifying the variable identifier refers to the variable "nearest" in scope.

Other language dependent problems which occur at the textual level include the handling of data structures, arrays and pointers. A single language independent syntax for dealing with fields of structures, elements of arrays and referenced objects of pointers is required which encompasses all operations allowed in the target programming languages. For example, array slicing is possible in languages like Algol68 and must therefore exist in the language independent syntax even though languages such as C and Pascal do not have this facility. Using the above notation all program objects can be unambiguously identified at the textual level.

The second category of monitoring support deals with dynamic constructs. Specifying a source statement, as a monitoring predicate, for which some monitoring action is to be performed when execution reaches that statement does not necessarily define the path of execution followed to reach that particular point. An extreme approach would require the user to specify as a predicate the exact sequence of statements which define a path of control through the program. As before, this extreme approach is impractical and so we attempt to identify the dynamic aspects of program execution which may be of interest to the monitoring system user. The first of these is the notion of a procedure-calling chain. In general it is possible to reach a source statement by a number of different execution routes; these differing by the sequence of procedure calls made. A user of the system may wish to halt and inspect the target process when a procedure is entered, but only if called from a second specified procedure. Thus a notation is required which defines sequences of calls through the program.

Plattner [Plattner84] uses the structure of the process state space to indicate a sequences of calls. A graphical representation, in the form of a multiway tree, has procedure calls as nodes. If the edges to child nodes are numbered sequentially starting at one, the flow of control through procedure calls can be denoted by an appropriate list of numbers relating to the edges traversed of the state space tree. This method not only defines the sequence of calls but also distinguishes between sequences where calls originated in different places. For example, the multiway tree shown in figure 3.5 is derived from the code segment in figure 3.6.

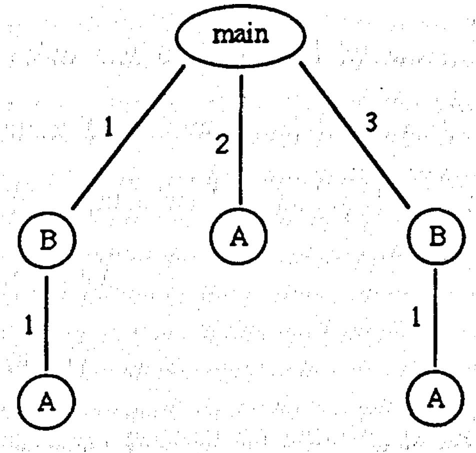

Figure 3.5

```
main()
{
   B() ;
   A() ;
   B() ;
}

A()
{

}

B()
{
   A() ;
}
```

Figure 3.6

The statement list 3:1 indicates entry to procedure B, from the second call in the main program, followed by the call to procedure A in procedure B. However, in most cases indicating the source of a call would be of a finer granularity than that required by the user. A mechanism is needed whereby a procedure calling chain can be specified with an optional facility for indicating the source of the call.

Cohen and Carpenter [Cohen77] develop an inquiry language for use with a history database which can be searched by label, such that specific labels occur before and after it. For example, L1:L2:L3 searches for label L2 with previous label L1 and next label L3. Using this notation a procedure calling chain could be denoted by

procA:procB:procC

This would halt the target process on entering procedure **procC** called from procedure **procB,** which, in turn, was called from procedure **procA.** If the source of the call is required then an optional line number-statement offset pair (that is, the source statement of the call) would be associated with the appropriate procedure identifier.

Path expressions [Bruegge83a] [Bruegge83b] use a similar mechanism for accessing specific instances of variables. For example,

M.P>M1.foo.i

denotes the variable **i** in the routine **foo** in module **M1** called from routine **P** in module **M.**

Recursion exists in most modern high-level languages and adds an extra level of complexity in specifying monitoring conditions. Both notations by Bruegge [Bruegge83a] [Bruegge83b] and Plattner [Plattner84] can handle recursion, but in a long winded way.

As an extension to specifying *n* procedure identifiers in a procedure calling chain, the user can specify just the identifier with a depth of recursion indicator *n*. A syntax for the referencing of variables local to recursive procedures is probably most useful if it allows them to be referenced as an offset from the current depth of recursion. For example, **procA/x(0)** = **procA/x(1)** specifies a condition which is satisfied when the value of the current variable **x** local to procedure **procA** is equal to the variable **x** local to the procedure **procA** which called the current invocation.

Information concerning procedure calling chains and recursion is preserved from one call to the next whereas other monitoring conditions, supplied by the user, may not always possess this property. This would occur, for example, if monitoring were required through a series of conditional statements. A monitoring command of the form:

WHEN <statement A>; <statement B> **PERFORM...**

would perform the associated monitoring action when statement B is executed after the execution of statement A. However, once execution has reached statement B it is, in general, impossible to tell whether statement A was executed or not. This chronological ordering of sub-predicates could also be applied to the flow of data as well as the flow of control.

A further monitoring mechanism, which can be used in conjunction with the features described above, is the ability to restrict monitoring predicates to a particular region of the target program. This can be used for a number of reasons, including the monitoring of variables whilst a particular statement is executing and the monitoring of the number of times a particular statement is executed whilst execution remains in a specified region of the program. This latter example could be used for the monitoring of loop iterations, with the first statement of the loop monitored whilst control remains in the scope of the loop. DISPEL [Johnson77] has a mechanism, as does Bruegge [Bruegge83a] [Bruegge83b], for accessing, within a debugging procedure, the number of times a routine or statement has been executed. These can be used to implement monitoring facilities similar to those above.

A rather contrived example, which involves some of the mechanisms outlined above, is given below.

WHEN procA:procB{20;24;28}:(40)procC{procB/y=procA/x} **PERFORM...**

This "watches" for an entry to procedure procA, followed by a call to procedure procB, in which, the statements on lines 20,24 and 28 are executed in that order. This followed by a call to procedure procC, from the call at source line forty, in which the variable y local to procedure procB attains the same value as the variable x local to procedure procA.

3.3. Software monitor

The monitoring software takes language independent monitoring commands and performs the necessary tasks to monitor the target process in the required way. The complexity of monitoring high-level language programs is related to the complexity of the programming language and, in particular, the complexity of the structure of the process states arising from that language. Thus, an inspection of process states arising from a variety of programming languages is required prior to the definition of the structures needed to monitor them.

Listed by Plattner [Plattner84] are the structures of the control and data components generated by a variety of types of programming language.

A simple programming language which has no concept of procedures or dynamic allocation of variables exhibits a static structure in both the control and data components of the process state. The control component is simply the current point of execution in terms of source statements, and the data component is the set of variables declared. The structure of the process state can be determined by examination of the program text.

A more complex language allowing procedures, but denying recursion extends the control component of the process state into a stack-like structure which is bounded in size by the number of procedure declarations. The data component, however, remains a static structure with variables retaining values across procedure calls.

A language with procedures and associated dynamic local variables, but still denying recursion, extends the data component into a stack-like structure which is again bounded in size. In this type of language the values of local variables are not retained across procedure calls. Allowing recursion produces unbounded stack structures in both the control and data components, which grow in proportion to the number of procedure invocations.

Some high-level languages allow the user to dynamically allocate and free memory during execution (HEAP in Algol68 or malloc in C). These *user controlled dynamic variables* release the data component from its stack-like structure, which occurs in the control component when the language has a concurrent computation facility.

From the above it is obvious that modern high-level languages, which offer procedures, recursion and user controlled dynamic variables, require more complex monitoring structures than those found in classical monitoring systems. For the remainder of this work we assume the process state structure is of the more complex type, although we assume the language does not provide a concurrent computation facility.

Basically, there are two methods of providing high-level language monitoring facilities. The first method involves trapping only part of the monitoring predicate and then, once this trap occurs, the rest of the predicate can be evaluated by reconstruction of the high-level image from the process state. In the case of a chronological ordering of sub-predicates (for example, a procedure calling chain), the part trapped would be the final part in the list. There are, however, three main problems with this approach. Firstly, it can be difficult to reconstruct the high-level image from a process state, resulting in quite complex monitoring routines. Secondly, not all information is available prior to execution, making it difficult to set traps. Examples of this include addresses of dynamic variables local to procedures, and the address of any memory space allocated to user controlled dynamic variables. Dynamic variables are usually referred to as offsets, in the program symbol table; these are added to the base address of the procedure invocation environment at run-time. The address of the memory space that user controlled dynamic variables reference is, by their very nature, under the control of the user and is again assigned at run-time. The third problem which is encountered with the above method of reconstructing the high-level image is the inability to monitor information which is not preserved during state changes. For example, the monitoring of the path through a series of program statements cannot be performed by simply trapping the final statement in the path; the required information (that is, the sequence of statements executed) is, in general, lost by the time execution reaches the final statement. The second and third problems added to the performance degradation caused by unnecessary trapping makes this method unsuitable for the monitoring of predicates outlined in the previous section.

We thus examine a second method of providing the required high-level language monitoring facilities. Instead of attempting to reconstruct the high-level image in order to evaluate the monitoring predicate, execution is mirrored within the monitoring software. Thus, the monitoring software recognises each part of the predicate as it is satisfied. The main difference between the two methods described is the need for the monitoring software, in the second method, to be able to dynamically set and reset machine-level traps, enabling each part to be monitored in turn.

We next examine how the predicates outlined in the previous section can be monitored using the method of mirroring execution. Procedure calling chains can be monitored by "watching" for the entry to each procedure, in the chain, in turn. As each one is satisfied then the focus of attention moves to the next. One important point to note is that the return from the procedure must also be "watched". This enables the focus of attention to revert to the previous procedure in the chain. There is also the possibility of intermediate calls, whereby a call to an unspecified procedure occurs between two procedures in the chain. One method of overcoming this problem is to "watch" the entry point of all procedures in the target program, so that any unspecified ones occurring can be recorded by the monitoring software. This method, however, can be extremely inefficient if many "invalid" procedure calls occur. A better method "watches" for entry to a procedure at a particular depth of procedure calls. Thus, in a procedure calling chain each procedure is required to occur at a depth one greater than the previous procedure. Consequently an indicator is required of the current depth of procedure calls. This is also useful for restricting predicates to a particular region of code. The monitoring software, in this case, "watches" the entry and exit points of the region; once execution is within the region then the specified predicate can be "watched", but required at a particular depth of procedure call to avoid intermediate procedure calls.

The monitoring of unpreserved information is easily performed using the idea of monitoring sub-predicates in sequence. For example, the monitoring of a predicate where the execution of one statement follows another is performed by first "watching" the earlier statement and then, when this is executed, the second statement. Although information about the execution of the first statement may no longer be available in the process state once it has occurred, the monitoring software has already recorded the fact and can proceed with the rest of the predicate.

The idea of monitoring one part of the predicate after another can also be applied to the monitoring of program data flow. Dynamic variables local to procedures can only be monitored once the appropriate procedure has been entered and the address of the variable calculated. Similarly, user controlled dynamic variables can only be monitored when the memory space is assigned to the variable. The method of following target program execution, in the monitoring software, allows the moment, when the above variables become active, to be monitored, prior to the monitoring of the variables themselves.

Because of the stack-like structures of the control and data components (with the extension of the data component where user controlled dynamic variables are defined) it is possible for more than one line of execution to be satisfying the monitoring predicate. For this reason the monitoring software must also be able to stack the different lines of execution, such that, if any of them satisfy the predicate then the monitoring action is performed.

The monitoring method outlined above will be examined in greater detail in chapter 4.

3.4. Architectural support

The target program is compiled and executed directly on the target machine, and thus all monitoring of the target program must also be performed at the same machine level. It is possible to design and implement abstract level monitoring facilities in hardware [Goossens83] [Rijks87] but they are often complex and specialised towards a particular monitoring feature. The environment described in this chapter requires a set of monitoring primitives which are minimal, yet sufficient, to enable the monitoring software to monitor the predicates described in section 3.2. For reasons of performance it would be advantageous for the monitoring primitives to have architectural support.

Examination of the machine level reveals three main operations: execution of a machine instruction, reading of a memory location, and writing to a memory location. We denote the primitives which trap the above operations as: the code breakpoint, the data breakpoint, and the watchpoint, respectively. It is assumed that these primitives are invoked with an absolute machine address and respond, when trapped, with that same absolute address and type of primitive. The monitoring software of the previous section is responsible for interpreting this at a higher level. It is possible to implement other primitives [Johnson82] but the three primitives described above are sufficient for monitoring all the high-level constructs which we have identified. Architectural support for the three primitives is examined, and possible forms of implementation described in chapter 5.

4. Monitoring Software

4.1. Introduction

4.1.1. General

Monitoring software performs two functions. Firstly, it must take commands supplied at the user level and translate these into controls over target program execution. It does this by use of the monitoring primitives in such a way as to "watch" for user-supplied conditions arising. For example, at the user level the tracing of variables is specified by the variable identifier, whereas at the machine level, this translates to the tracing of an absolute memory location, using a watchpoint primitive. The second task of monitoring software is the reverse of the above and involves the translation of execution information into a symbolic high level form for presentation to the user. For example, the displaying of a variable involves the extraction of information from a memory location in the form of a bit pattern. It is the task of the monitoring software to use any symbolic information supplied, to display this bit pattern in an appropriate form. Similarly, requesting the current point of execution should not result in a list of program counter values, which have been stored as a result of procedure calls. As in the above example, symbolic information is used to effect a transformation from the machine level to a higher level representation which, in this example, might result in a procedure-calling chain being displayed.

As stated earlier, monitoring software is located between the user level and the machine level of the monitoring environment. The high level command, issued from the user level, is discussed in section 3.2 and is assumed to take the form of a generalised WHEN statement.

Each WHEN statement is a predicate-action pair; the action being performed when the predicate is found to be true. The strength of the monitoring software lies in the diversity of types of predicate which can be monitored. From examination of structured programming languages a list of programming concepts which might be allowed in the predicate can be made. In section 3.2 we identified the following as some of the concepts for which monitoring support is required: procedures, variables local to procedures or program blocks, procedure calling chains, recursion, user-controlled dynamic variables and also control and data flow which is not generally preserved within the target process state. The action associated with the predicate takes the form of a recording operation, a notification operation or a target process halt operation, from which a conversational tool allowing target process state examination can be entered.

In section 3.4 we identified a minimal set of monitoring primitives for use by the monitoring software in controlling the target process. Monitoring the flow of control requires the code breakpoint which monitors the execution of an instruction at a particular memory location. Monitoring the flow of data requires two primitives: the data breakpoint, which monitors accesses (reads) of memory locations, and the watchpoint, which monitors updates (writes) to memory locations.

4.1.2. Basic features of a software monitor

Monitoring software is concerned with the use of the monitoring primitives to bring about monitoring of the user-specified high-level condition. We define a basic software monitor as one which performs simple monitoring by a minimum amount of processing. The facilities available thus resemble the monitoring primitives more closely than the high-level facilities outlined in chapter 3.

The minimum amount of processing required of high-level language monitoring software is the translation of user-supplied program identifiers to machine-level addresses. This allows the user to state watchpoints and breakpoints in terms of the high-level source code. Translation of identifiers to addresses is simple given that the information produced at compile time is not discarded. Most symbol tables which are built during compilation give machine addresses for global variables, procedures and labels but only offset values for local variables. The basic monitoring software described requires that, not only is this information retained, but also machine addresses for each program line or even program statement are available. Using this table the monitoring software can "watch" for the execution of a program line or statement and the update or access of global variables.

The mechanism, within the monitoring software, for transferring control between the monitoring process and the target process is also simple. After setting the appropriate monitoring primitives the monitoring software resumes execution of the target process and waits for one of the traps to be taken. When a trap is taken an interrupt-like mechanism halts the target process and informs the "sleeping" monitoring process. The monitoring software can now enter a conversational tool which allows the state of the target process to be examined and monitoring primitives to be set or removed.

To provide additional features within the monitoring software requires an increase in the amount of processing performed. The monitoring software still sets primitive monitoring traps statically (the user explicitly sets and resets them) but now has the ability to access target process state information on transfer of control. As an example consider the command:

WHEN X = 0 ...

For this condition the monitoring software must invoke monitoring primitives to "watch" for updates to the variable X, and keep a record of the expression for checking during execution. When a trap is taken and control is passed from the target process to the monitoring process the monitoring software uses the stored expression to access variable values and check if the expression is satisfied. If the expression is satisfied then the respective monitoring operation is performed, otherwise control is passed back to the target process and the monitoring process waits once more for a trap to occur. Using this system it is possible to monitor expressions involving global variables, execution of procedures or source statements, and also procedure-calling chains. Conditions involving chains of procedure calls can be monitored as the necessary information is available in the target process state as a set of return addresses. A breakpoint on the entry point to the final procedure in the chain is set and then, when taken, the monitoring software can check the stack of return addresses for the correct calling chain. The user is never notified of transfers of control for which the monitoring record was not satisfied, and thus only sees the conditions specified in the original command.

This basic monitoring software does, however, have its limitations and problems. The major limitation is linked to the fact that all primitive monitoring functions are invoked statically and so traps cannot be added or removed during the course of execution unless explicitly done so by the user. This means that the system is incapable of monitoring local variables or user controlled dynamic variables (variables which are assigned space via the use of, for example, HEAP in Algol68 or malloc in C). Conditions involving information which is not preserved within the target process state are also incapable of being monitored using this basic monitoring software.

Apart from the above limitations there is also the possibility of a performance loss due to transfers of control. This arises, particularly with procedure calling chains, when the condition being "watched" by the primitive monitoring functions occurs many times, causing a transfer of control, but the monitoring record stored within the monitoring software is not satisfied.

4.1.3. <u>Higher-level monitoring</u>

Predicate-action monitoring systems allowing the monitoring of abstract high-level concepts are not new. DISPEL [Johnson81] is an event-action language providing primitives and a mechanism to define debugging procedures. The primitives from which the debugging procedures are built consist of keyboard interrupts, run-time errors, entry and exit of statements or routines and the access or update of variables. Two system functions indicate the number of times a variable has been accessed and the number of times a particular code segment has been entered. It is not clear that DISPEL has the ability to monitor complex predicates involving, for example, paths and local variables at differing depths of recursion.

Generalised Path Expressions [Bruegge83a] [Bruegge83b] is the result of a modification of a system designed for the synchronisation of concurrent processes. Path expressions are specified by the operators repetition, sequence and exclusive selection; and operands called *path functions*. For example,

```
    Path(Open;(Read|Write)*;Close)End;
```

specifies that a file has to be opened first before an arbitrary sequence of alternating read and write operations, followed by a close operation.

The history variables REQ, ACT and TERM can be applied to any path function and indicate the number of times the function has attempted to be performed, started to be performed and terminated respectively. Debugging with generalised path expressions takes one of two forms: either the specified execution sequence is looked for and the path action defines what to do if the sequence occurs, or the execution sequence is enforced and the path action defines what to do if a violation occurs. For example,

```
FINDPATH BeginLoop
  WhileLoop [ACT(PostLine) = N and ACT(PlaceLines) = 1]
```

looks for activation of WhileLoop when PostLine has been called N times and PlaceLines once.

Alternatively,

```
CHECKPATH Loop
  { WhileLoop [ACT(WhileLoop) < 6] | PlaceLines }*
```

enforces that WhileLoop should not be executed more than six times before a call to PlaceLines occurs.

The implementation of generalised path expressions uses a prologue/epilogue approach. If a path function is called then a prologue is entered which updates the history variable ACT. It is then determined whether the call is an allowed transition in the execution sequence by checking the current path expression state. Upon exit an epilogue updates the TERM history variable and state transition is checked again.

Whilst the monitoring of the flow of control allows the inclusion of abstract high-level concepts such as procedure calling and paths, generalised path expressions are less able to monitor the flow of data during execution.

An event definition language (EDL), described by Bates and Wileden [Bates82] [Bates83], provides users with a means of obtaining a behavioural abstraction from a distributed system's event stream. The user can hierarchically construct events on top of primitive events or earlier event definitions, but EDL cannot distinguish between the request, activation and termination of primitive events. There is also only constructs for event detection and not for actions.

To improve on the basic monitoring software described and the above event-action languages a mechanism is required whereby the monitoring software can mirror the run-time calculations performed by the machine during execution. Using this system both local variables and user controlled dynamic variables can be monitored. Local variables are stored as offsets in the program symbol table, and the absolute address is calculated by adding this offset to the base address of the current procedure activation record on the stack. Monitoring of local variables thus requires the monitoring software to perform this same calculation which can only be done on entry to the appropriate procedure. Similarly user controlled dynamic variables have no address until the space allocation routine is called and an area of memory is set aside for them. Again, calculation of machine addresses for these variables must be mirrored by the monitoring software. The monitoring of information which is not generally preserved within the target process state is slightly different, in that inspection for calculation is not needed but some sort of record within the monitoring software, which can be appropriately updated as the sequence of execution points are reached; that is, a generalisation of the history variables of Bruegge.

In the basic monitoring software all primitives are invoked statically thus leaving the user in charge of setting and resetting them. However, if the monitoring software was able to dynamically set and reset primitive monitoring functions then the intermediate points of control, highlighted above, could be trapped, calculations performed and then execution resumed.

Plattner [Plattner84] describes monitoring software which is able to include local variables in monitoring expressions. Each monitoring statement is given a unique identification and stored in the *monitoring statement list.* This consists of three fields: the predicate, the action and a boolean value which indicates whether the monitoring statement is available for evaluation. The *monitoring statement evaluation list* contains, for each state variable reference occurring in a predicate, a list of identifications of monitoring statements that must be checked when the corresponding state variable is written to. Thus each entry possibly references into many entries of the monitoring statement list. The structure which is used to monitor procedure calls and returns is the *potential procedure activation tree.* This takes the form of a multiway tree created by the recursive algorithm:

> "starting at the current node, scan the program text of the associated procedure. If this procedure calls another procedure, create a new successor node, label it with the name of the called procedure, and make this new node the current node. Then execute this algorithm again." [Plattner84:758]

Execution of the target process can be viewed as a tree walk in the multiway tree described. Using a pointer to point to the current procedure activation node of the tree a procedure call is the moving of this pointer to the appropriate child node and a procedure return is the moving back to the parent node.

On procedure calls any local variables have their absolute addresses calculated and appropriate breakpoints are set. Also monitoring statements are marked available for evaluation if the evaluation condition of their predicates is fulfilled. The evaluation condition of a predicate is satisfied when the node pointer is on or beyond the node which guarantees that all the necessary procedures have been called for the local variables in the predicate to exist.

The above proposal, however, does not solve the problem of monitoring user controlled dynamic variables or the inclusion of information which is not, in general, preserved within the target process. For example, the monitoring of a sequence of source statements requires more than examination of the target process at a single instant in execution.

4.1.4. Levels of monitoring

None of the systems studied offers a complete solution to the general problem of execution monitoring. To help analyse this problem, we identify three levels at which monitoring predicates may be specified:

Primitive level

At this level monitoring predicates take the form of the execution of instructions at machine addresses or the access/update of memory locations.

Abstract level

This level builds on the primitive level by introducing high-level language concepts. Examples of predicates at this level are the call of a procedure or the assignment to a variable. Only one type of primitive is required to monitor each abstract level predicate but further information may be necessary to check whether it is satisfied.

Conditional level

This is the level at which the user specifies monitoring commands; it differs from the previous level in that it cannot, in general, be monitored by simple inspection of the target process at a single instant as it may imply a sequence of abstract level predicates. For example:

WHEN 11;12;13 PERFORM <monitoring action>

which performs the monitoring action when the source statements on lines 11, 12 and 13 are executed.

The monitoring software must break the user specified commands (at the conditional level) into the necessary sequence of abstract level predicates. A representation of the conditional level is thus required which shows the sequence of abstract level predicates or **events** needed to bring about satisfaction of the user-specified condition. A facility is also required alongside this which enables the mirroring of the run-time calculation of machine addresses. Rather than just following a sequence of events a conditional level predicate often requires a "going back in time" facility. This occurs when an event is monitored until a second event occurs. At this point all monitoring primitives must be set to look as they did before monitoring of the first event commenced. The representation of the conditional level predicate must be able to show this "going back in time."

We define an **event-graph** as a directed graph representation of the sequence of events needed. The nodes of the graph represent the events of the monitoring condition and the arcs indicate the sequence of events. An arc which forms a cycle in the event-graph represents the "going back in time" to a previous state. As execution of the target program proceeds and traps are taken in the target process, then the monitoring software traverses the event-graph, setting and resetting monitoring primitives accordingly. The conditional level predicate is satisfied when the terminal node of the graph is traversed. However, it is not necessary for all of the nodes to have been visited for the monitoring condition to be satisfied.

For example, consider a monitoiing condition which is satisfied if an assignment occurs and a specified procedure is in the current procedure calling chain. The event-graph representation for this is shown in figure 4.1.

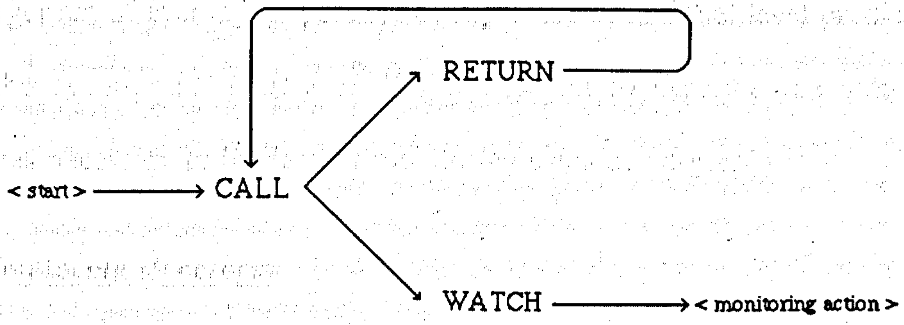

Figure 4.1

The CALL, RETURN and WATCH events represent procedure entry, procedure exit and variable assignment respectively. From the above representation monitoring begins with the CALL event. When this is satisfied the event-graph is traversed by following the outgoing arcs, giving the next events in the sequence to be monitored. In this case monitoring switches to the exit point of the procedure and the assignment of the variable. If the WATCH event is satisfied then the event-graph has been fully traversed and the monitoring operation associated with the condition is performed. However, the cycle around the CALL and RETURN events indicates a "going back in time" and so when the RETURN event is satisfied the arc going backwards in the sequence requires that the monitoring primitives are set just as they were when the CALL event was first visited. That is, we no longer require monitoring of the abstract level events found further along the sequence than the CALL event. From the representation for this example it can be seen that under no circumstances could the monitoring operation be performed unless the exact sequence of program events required for condition satisfaction had occurred. By the suitable use of a larger set of events it is possible to represent conditional level predicates using the event-graph structure. We will go on to describe the software structures necessary to implement monitoring using event-graphs.

4.2. Monitoring structures

4.2.1. Overview

In this section we describe data structures sufficient to represent each of the nodes or events of the event-graph defined in the previous section. Information within the event structure falls into two categories, the first of which is concerned with housekeeping. Fields in this category are not directly responsible for representing high-level concepts but are used to aid in the representation of a conditional-level predicate as an event-graph. The second category of field information is used directly to describe high-level abstraction.

In total we will define eight fields associated with an event structure, shown in figure 4.2.

Identifier	Type	Nesting Level Control	Enable Control	Local Variable Stack	Event Expression	Local Action	Action

Figure 4.2

The housekeeping fields are the *identifier, action* and *enable control* fields. The *identifier* field, which can be a simple integer, enables the monitoring software to associate monitoring primitives with events and also locate a particular event within the event-graph. The relationship between machine-level primitives and events of the event-graph is maintained by the monitoring software using the event identifier. Machine-level primitives are invoked with an absolute machine address and are independent of the event which invoked them. When traps are taken, control is passed to the monitor process, which is informed of the type and machine address of the primitive responsible. It is thus a function of the monitoring software to associate the relevant events with the traps taken. One possible method of doing this is by keeping a list of event identifiers which correspond to particular primitive types and machine addresses. One consequence of this is that only one primitive of any one type and address need be set for all events which require it. Events which need to invoke primitives for which the machine trap is already set, simply have their event identifiers appended to the list of identifiers (held by the monitoring software) which correspond to the trap. The removal of monitoring primitives is also performed via the list of event identifiers, with the identifier removed from the list, but the machine trap is only removed if no further event identifiers are associated with it. The addition and removal of identifiers from the lists can be thought of as the setting and resetting of *logical* primitives.

The *action* field provides the means by which the event-graph takes its structure. This field implements the arcs in the event-graph by referencing successor events. A further field (not shown) is used to allow multiple outgoing arcs. By using this field to link common successors all "actions" of an event can be found by firstly, following the action field to reference the first successor and then within the successor events by following this extra field.

The third field in the housekeeping category is the *enable control* field. This field is included primarily for reasons of efficiency. After visiting a node of the event-graph and before looking at any successor nodes the usual course of action is for the monitoring software to remove any primitives associated with the node. However, it may be the case that the event is required to remain active even when satisfied. An example of this would be the tracing of a variable or entry to a procedure, where all instances need to be monitored, not just the first. The easiest and most efficient method of implementing this requirement is a simple boolean field which indicates it.

Fields in the second category are true event fields, in that they are incorporated into the event structure for the purpose of implementing abstract programming concepts rather than implementation of the representation.

The *type* field indicates which abstract programming concept the event is monitoring. For example, CALL, RETURN and WATCH in the previous section are all types of events. In total we will define thirteen different types of event required for monitoring purposes. This field informs the monitoring software how to interpret the rest of the event structure, and in particular the event expression field, and is also used to determine which monitoring primitive to invoke to monitor the event. Only one type of monitoring primitive is ever used with a particular type of event.

To enable the run-time calculation of addresses the *local action* field contains the local variable offset found in the target program symbol table. The calculation is performed when the associated event is satisfied.

The nesting level of a process state refers to the depth of procedure calls. Thus the *nesting level control* field allows events to be tied to certain nesting levels.

The *event expression* field references the machine information necessary for the invocation of monitoring primitives, and the *local variable stack* is used to store this information. In the simplest of cases the local variable stack is superfluous to requirements but is included in the event structure for more complex conditions.

4.2.2. Event types

The type field of the event structure indicates the function of the abstract level event thus determining how the rest of the fields of the event structure are to be interpreted. When invoking monitoring primitives this field is accessed to find which of the three primitives is appropriate. Each of the different types of event monitors using just one of the three types of primitive and in most cases using cnly a single primitive.

There are thirteen types of event falling into three categories. The first category monitors the flow of control through the target program and comprises five types. The CALL and RETURN events are most commonly used as a pair ensuring that successor events in the event-graph are monitored only when a specified procedure is active. The usual arrangement of these two types in the event-graph is shown in figure 4.3.

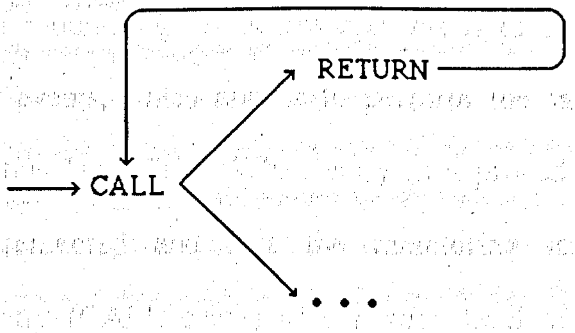

Figure 4.3

A CALL event, which monitors procedure entry, is succeeded in the event-graph by a RETURN event and the rest of the predicate representation. When entry to the procedure occurs, the rest of the predicate is monitored, as is procedure exit. If a return from the procedure occurs then the monitoring software reverts to monitoring the procedure entry and stops monitoring the rest of the predicate.

Similar to the CALL and RETURN events are the ENTRY and EXIT events (figure 4.4).

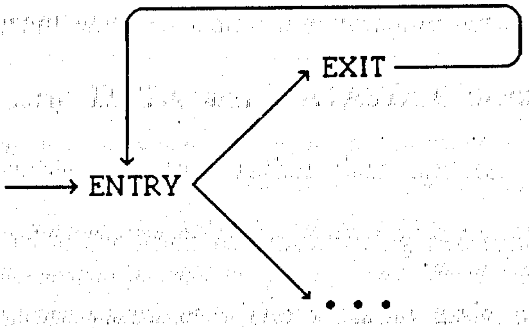

Figure 4.4

These events are used when a code segment, other than a procedure, is to be active before monitoring of successor events is to occur. The necessity for two pairs of events, which appear to perform the same function, is linked with the nesting level field. When entry to a procedure occurs, the nesting level is incremented and thus the monitoring software must monitor procedure entry (CALL event) at a nesting level one greater than for an ENTRY event. This will become clearer when examples are described in a later section.

To allow simple source statement monitoring an event of type CODE is provided. All the events in this category make use of the code breakpoint monitoring primitive "watching" for instruction execution.

The second category consists of six types of event and facilitates the monitoring of data flow. The flow of data during execution includes variable assignment and variable reference, both with either local, global or user controlled dynamic variables. Consequently there is an event type for each possible combination. WATCH events monitor updates to local variables and WATCHSTAT for global variables. Two different events are required here because of the different ways in which the machine address to be monitored is calculated. Similarly DATA and DATASTAT monitor both types of variable but for variable reference rather than update. DATAUCDV and WATCHUCDV provide facilities for monitoring references and updates to user controlled dynamic variables. The variable assignment types use the watchpoint primitive whereas the reference types use the data breakpoint.

The third category of events enables the monitoring of expressions. An expression consists of operators, variables and constants. A type EXPRESSION is used to root an expression tree with constants held in events of type CONSTANT and variables included in types WATCH or WATCHSTAT. As an example consider the expression: $X+Y=Z-1$ where X and Y are local variables and Z is globally declared. An expression tree for this appears in figure 4.5 and the corresponding event-graph structure in figure 4.6.

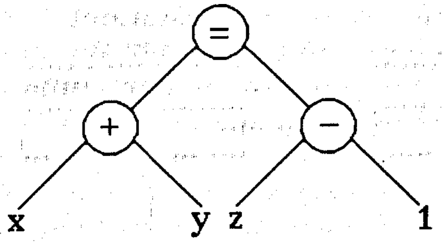

Figure 4.5

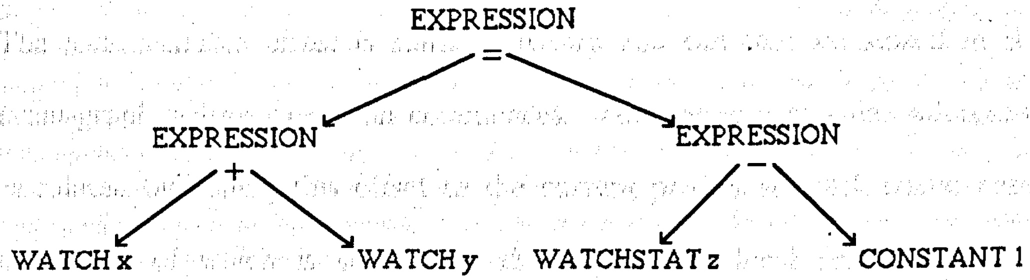

Figure 4.6

4.2.3. Run-time calculations

The local action field is used to mirror the run-time calculations performed by the machine. Primarily for use in calculating addresses for local variables, local actions are most often used in conjunction with events of type CALL and ENTRY. This is the case because CALL and ENTRY events monitor entry to code segments and thus the activation of new local variables. The local action field contains a list of local action structures, which consist of a local variable offset (found in the program symbol table) and a stack structure. This stack structure is used to hold the machine addresses calculated at run-time (figure 4.7).

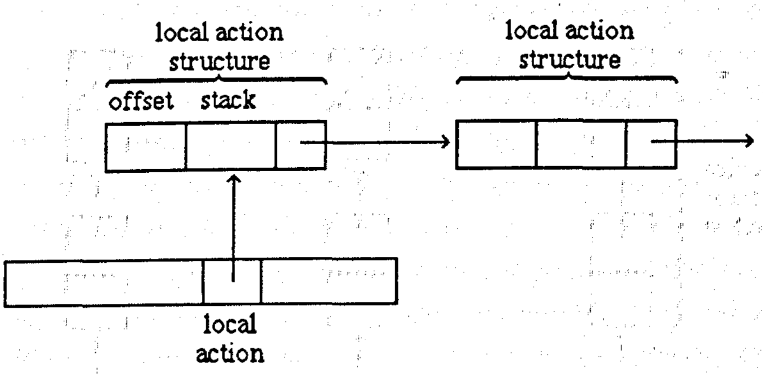

Figure 4.7

The local variable offset is static in nature and can thus be stored in the event-graph before execution commences. The absolute machine address is calculated by adding this offset to the current procedure stack frame base; the result of which is stacked in the appropriate local action structure. Maintaining a list of local action structures allows more than one local variable, declared in the same program block, to be monitored. The storage space for the calculated address is a stack structure due to the possibility of recursion. Each time the code segment is entered recursively a new address is calculated and stacked, and popped off again when the segment is exited. Successor events can then access the appropriate local action structure for the machine address to set monitoring primitives (figure 4.8).

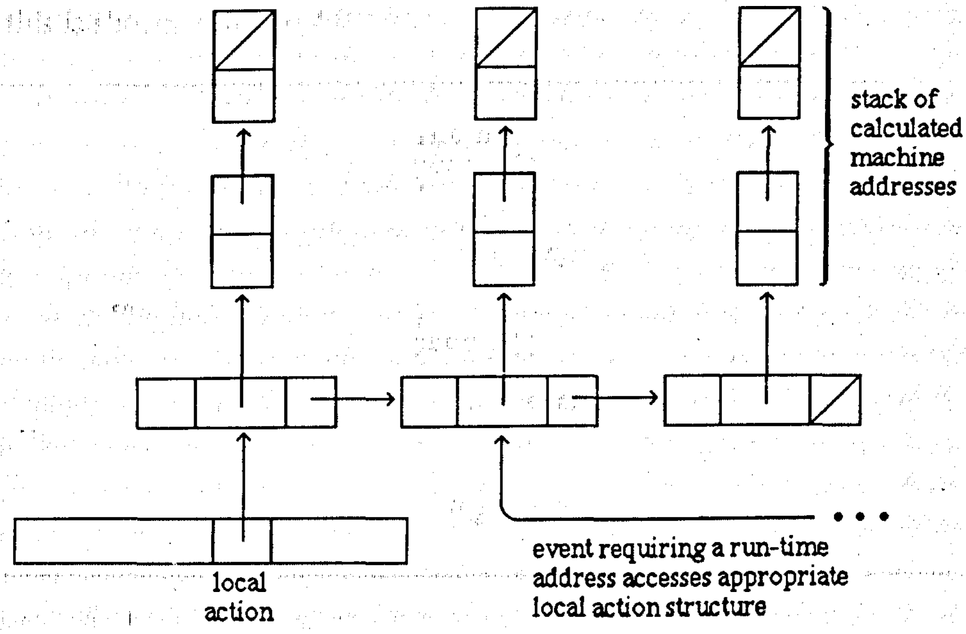
stack of calculated machine addresses
local action
event requiring a run-time address accesses appropriate local action structure
• • •

Figure 4.8

4.2.4. Nesting level

The nesting level control field consists of a boolean which indicates whether the field is to be used for this particular event and a nesting level value. The nesting level of a process refers to the depth of procedure calls; that is, the number of procedures currently active. Thus any active procedure is at a nesting level one greater than the routine that called it. By storing a nesting level value in the nesting level control field, and then checking this against the current nesting level value when traps are taken, events can be tied to particular nesting levels thus removing the problem of intermediate procedure calls.

As an example consider the event-graph representation for the assignment to a variable while textually confined to a particular procedure. Example code for this is shown in figure 4.9.

```
int x ;
proc A()
{
    B() ;
    x = expr ;
    A() ;
}
proc B()
{
    x = expr ;
}
```

Figure 4.9

The event-graph representation for the monitoring of updates to the global variable X, but only those confined to the textual region of procedure A, is given in figure 4.10.

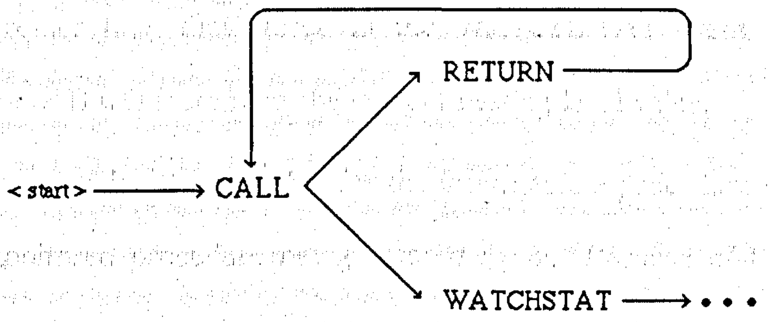

Figure 4.10

The CALL and RETURN events monitor entry to and exit from procedure A and the WATCHSTAT event monitors the global variable X. If the nesting level field were omitted from the event structure then the update to variable X, in procedure B, when called from A, would also result in the monitoring action. By placing the nesting level value which occurs on entry to procedure A in the nesting level control field of the WATCHSTAT event, and then checking this value, against the current nesting level value, when the variable is updated, the update in procedure B can be ignored (figure 4.11).

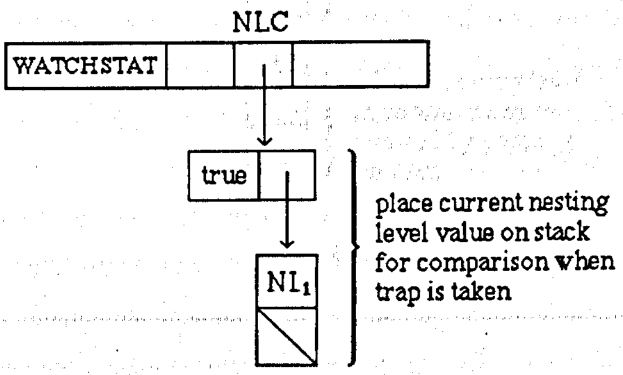
NLC
WATCHSTAT
true
NI_1
place current nesting level value on stack for comparison when trap is taken

Figure 4.11

This is so because the update in procedure B will execute at a nesting level one greater than that stored in the WATCHSTAT event, as the call to procedure B will increment the current nesting level value.

Due to recursion it is possible for the intermediate procedure call to result in the monitored procedure being entered again. Because of this, the nesting level value must be stored in a stack structure in the nesting level control field. This allows the WATCHSTAT event to monitor updates to the variable X for the recursive call to procedure A, but enables the nesting level value of the first call to be restored when the recursive call exits (figure 4.12).

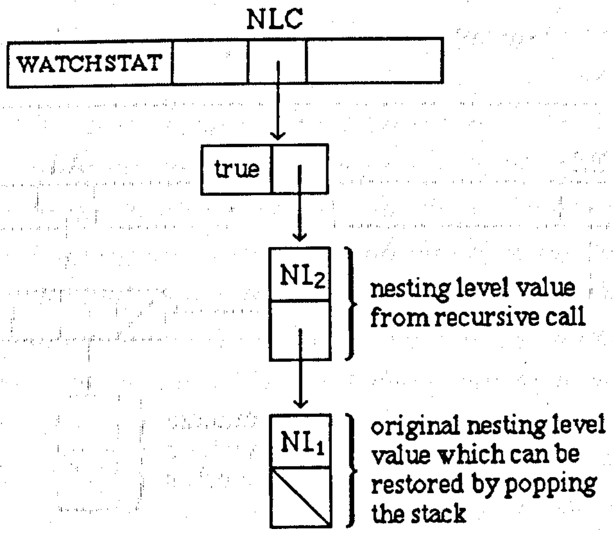
NLC
WATCHSTAT
true
NI_2
nesting level value from recursive call
NI_1
original nesting level value which can be restored by popping the stack

Figure 4.12

4.2.5. Machine information

The event expression field holds all the machine level information needed for invoking monitoring primitives. The type field indicates how this field is interpreted. In the simplest case static information, which is known before execution, is referenced by the event expression field (figure 4.13).

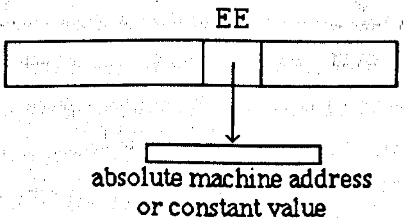

Figure 4.13

In the cases where machine addresses are calculated at run-time (DATA and WATCH) the event expression field references the structure created by the local action field of the appropriate event (figure 4.14). The event expression field of WATCHUCDV and DATAUCDV events reference a second event, which will be explained by an example later in this chapter.

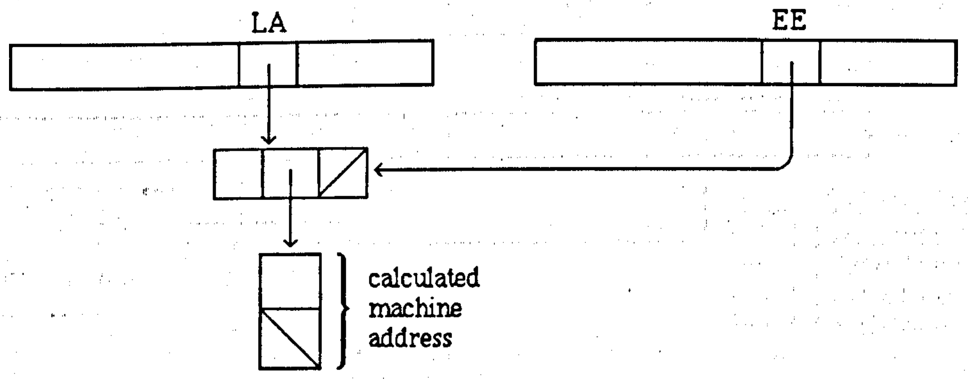

Figure 4.14

Events of type EXPRESSION have an event expression field which references an expression structure, consisting of an operator and pointers to two other events, thus forming an expression tree (figure 4.15).

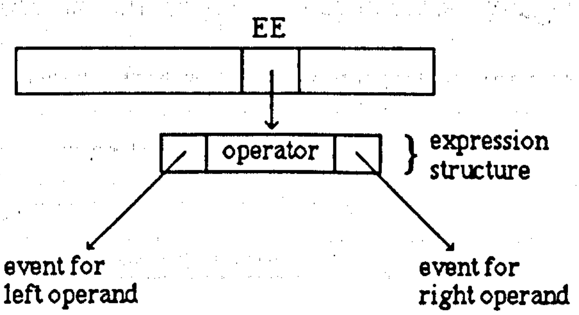

Figure 4.15

The local variable stack is used to hold machine information for *consumer* events of addresses calculated at run-time. We refer to the event which gives rise to the stack of calculated addresses as the *producer* event and the event which accesses the addresses via the event expression field as the *consumer* event.

Consumer events use the addresses accessed through the event expression field to invoke monitoring primitives and also copy them to the local variable stack (figure 4.16).

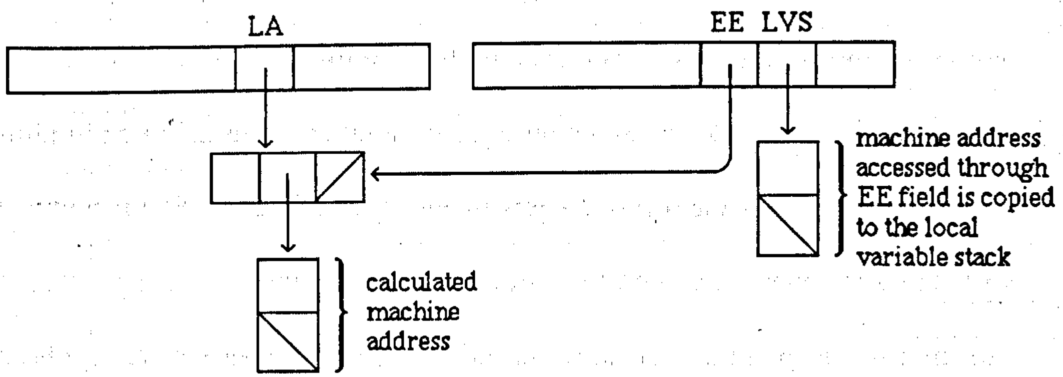
LA
EE LVS
machine address accessed through EE field is copied to the local variable stack
calculated machine address

Figure 4.16

The necessity for the local variable stack arises when events are placed between the producer and consumer events in the event-graph and consequently the local action stack does not necessarily indicate the addresses monitored by the consumer event. This will be further investigated in the next section with reference to an example.

4.3. Monitoring with a directed graph

Monitoring software handles each event in a uniform manner with the overall structure of the event-graph having no influence over the way in which individual events are monitored. The monitoring of a conditional level predicate is, however, determined by the event-graph structure and the degree to which the predicate is satisfied is indicated by how much of the event-graph has been traversed. Traversal is based on the traps taken and the information contained within the event structures. The predicate specified in the monitoring condition is found to be satisfied without any user interaction being necessary.

We identify two phases in the monitoring of individual events which will be referred to as **examination** and **evaluation.** In the simplest terms **examination** of an event results in traps being set on the target process and **evaluation** which occurs in response to traps taken results in the examination of successor events. More fully, **examination** of an event occurs as a result of evaluating its predecessor or, in the case of the *start* event, when the examination phase is explicitly applied to the event. There are two functions that the examination phase must perform; firstly, a nesting level value is stacked in the nesting level control field, whether the field is marked active or not. The nesting level value stored is the value which is required for the event to be satisfied during evaluation. The second function of the examination phase is different for different types of event but, in general, involves accessing the event structure to get machine information for the setting of monitoring primitives.

Evaluation of an event accesses the event structure information to check whether it is satisfied or not. Satisfaction of an event occurs if the trap was taken at the correct nesting level. If the event is found not to be satisfied then no further action is taken, except a transfer of control back to the target process. If an event is found to be satisfied then appropriate traps are removed and stacks within the event structure popped as required. A final function of evaluation is the examination of successor events.

The detailed working of the examination and evaluation phases for the different event types is best described with a set of examples. Where necessary a C-like language will be assumed, although the monitoring is not language-specific.

Example 1

Consider the monitoring command:

WHEN <line 6> PERFORM <monitoring action>

When applied to the code fragment in figure 4.17 the monitoring action is performed when the specified source statement is reached in execution. The event-graph representation of the above monitoring command is shown in figure 4.18.

```
line
nos

 5   statement1 ;
 6   statement2 ;
 7   statement3 ;
```

Figure 4.17

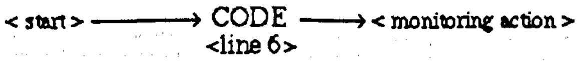

Figure 4.18

An event of type CODE indicates that a code breakpoint is to be set at the location specified by the event expression field; the trap taking effect when execution reaches it. The location stored in the event expression field is taken from the symbol table, held by the software monitor, which lists addresses for source statements. The successor event of the CODE event is the monitoring action event.

There are two other fields within the CODE event which alter the way in which monitoring proceeds. The nesting level control field, if set, requires that the machine-level trap is taken at a particular depth of procedure call for the event to be satisfied during **evaluation.** Traps taken at any other nesting level are ignored. The other field which alters the effect of the event-graph is the enable control field. If this is set then all occurrences of the event result in the monitoring action, otherwise only the first satisfaction performs this and then the event becomes inactive. Without further qualification from the user the default setup for the event-graph would monitor all occurrences of the condition and at any nesting level.

Monitoring of this event-graph commences with **examination** of the CODE event. The event expression field is accessed for a machine address and a code breakpoint is set at the address. Control is then passed to the target process and execution continues normally until the breakpoint is reached. At this point control is passed to the monitor process which relates any machine-level traps to abstract-level events. **Evaluation** of the CODE event now takes place. With a default setting the event is always satisfied and the monitoring action is performed with the event and associated trap left active.

Global variable monitoring would use a similar graph but with an event of type WATCHSTAT and an event expression field containing the address of the variable. The resultant trap from **examination** of this event would be a watchpoint for monitoring updates to machine locations.

Example 2

WHEN 10{11;16} PERFORM <monitoring action>

The monitoring condition in the above command requires that the statement at line 11 is executed, followed by execution of the statement at line 16, with monitoring restricted to the block beginning at line 10. When applied to the code fragment in figure 4.19 the monitoring action is performed when control firstly passes through <statement1> (expression-A results true) and then <statement4> (expression-B results false) for any one execution of the conditional block. Sequences such as: execution of line 11, line 15, then an iterative execution of line 12, followed by line 16 do not result in the monitoring action.

```
line
nos

10   if( expressionA )
11      statement1 ;
12   else statement2 ;
13
14   if( expressionB )
15      statement3 ;
16   else statement4 ;
```

Figure 4.19

The event-graph representation of the above monitoring command is shown in figure 4.20.

The inclusion of the ENTRY/EXIT event pair enables monitoring to be restricted to the code fragment. The event expression field of the ENTRY

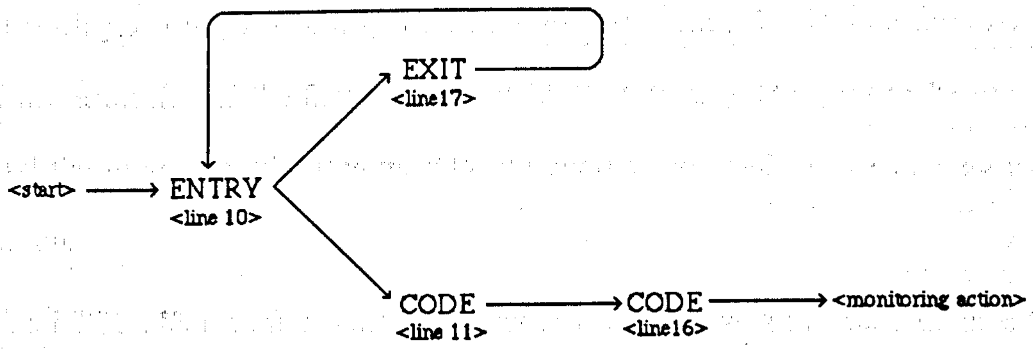

Figure 4.20

event is the address of the first instruction of the code fragment and the same field of the EXIT event holds the address of the instruction executed following the code fragment. The event expression fields of the two CODE events hold the appropriate addresses for the specified source statements. As in the previous example the state of the nesting level and enable control fields can alter the effect of the event-graph. If the user requires the monitoring action for all occurrences of the predicate then the enable control field of the ENTRY event is set, otherwise only the first instance of the predicate will result in the monitoring action. The state of the nesting level control field of the ENTRY event determines whether the monitoring predicate is restricted to the nesting level applying when the command was specified, or not; the latter case allowing intermediate procedure calls. In the following we assume the event-graph monitors all occurrences at any nesting level.

The ENTRY/EXIT event pair prevents iteration of the code fragment from causing invalid monitoring actions. This is because the EXIT event restores the state of the event-graph as if the code fragment had not been entered, removing any traps on the specified source statements. Iteration cannot now cause execution of the first source statement <statement1> on one iteration and the execution of <statement4> on another, to result in the monitoring action.

The EXIT event must monitor the flow of control leaving the code fragment at a particular nesting level (nesting level control field is active), with only the first instance monitored (enable control field is not active). The two CODE events also have the nesting level control field active and the enable control field not active. This enables the event-graph to monitor the correct flow and distinguish between recursive calls. The effect is that on any one pass through the code fragment, <statement1> must be executed followed by <statement4>, for the monitoring action to be performed.

Traversal of the graph commences with the **examination** of the ENTRY event. This results in a code breakpoint being set at the entry point to the code fragment. Control is now transferred to the target process, which executes normally until the instruction at the code breakpoint is executed. Control now passes back to the monitoring process which relates the trap taken to the event which set it. This results in the **evaluation** of the ENTRY event. Successful **evaluation** of the ENTRY event results in the **examination** of its two successor events.

Examination of the EXIT event places the current nesting level value in the nesting level control field and sets a code breakpoint at the exit point of the code fragment. Similarly, **examination** of the first CODE event places the current nesting level value in the nesting level control field and a code breakpoint at the address of the source statement <statement1>. Assuming a current nesting level value of NL_1, the resulting state of the event-graph is shown in figure 4.21.

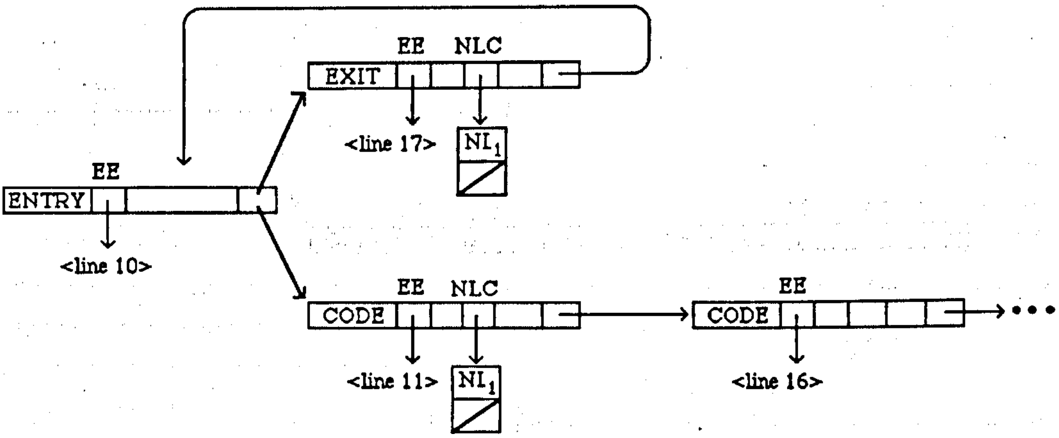

Figure 4.21

Execution of the target program now continues until a trap occurs, causing a transfer of control to the monitoring process.

Assuming that the execution of <statement1> caused the transfer of control then the CODE event is **evaluated,** which is satisfied if the current nesting level value is still NL_1. If the trap occurred after a recursive call then **evaluation** fails and target program execution continues. If, however, **evaluation** succeeds then, because the enable control field is not active, the code breakpoint, set by this event, is removed and the nesting level control stack popped, and the successor event is **examined.**

This successor event is the second CODE event, the **examination** of which stacks the nesting level value in the nesting level control field and sets a code breakpoint at the address of the source statement <statement4> (figure 4.22).

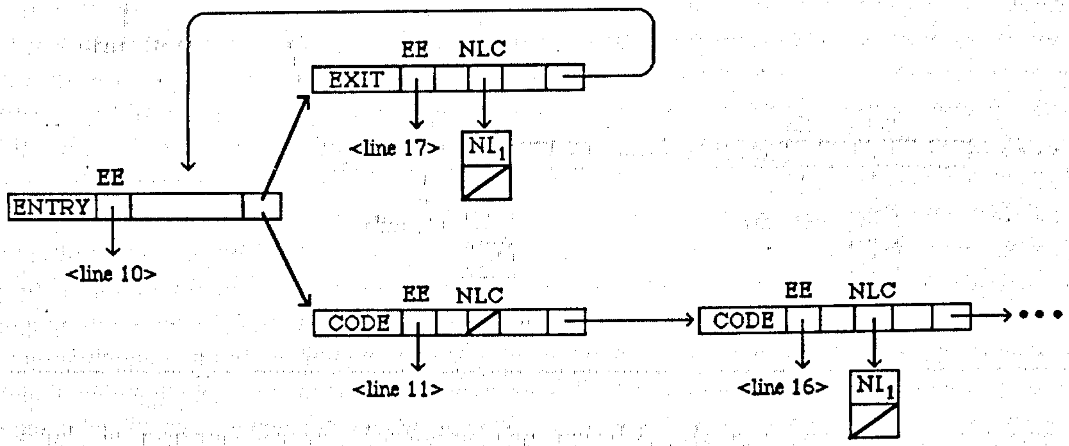

Figure 4.22

Evaluation of the second CODE event would occur in a similar way to the first CODE event and, if satisfied, would result in the monitoring action.

If the ENTRY event is **evaluated** again, following a recursive invocation of the sequence, then its two successor events are **examined** for a second time. This proceeds in a similar way to the first time, with the new nesting level value (say NL_2) stacked in the nesting level control field (figure 4.23).

One difference does exist between the two **examination** phases. Because all addresses in the event expression fields of the events are static, only one monitoring primitive is needed for each event. Thus, each recursive call which results in the code fragment being executed can use the previously set trap. The removal of the primitive only occurs when the nesting level control

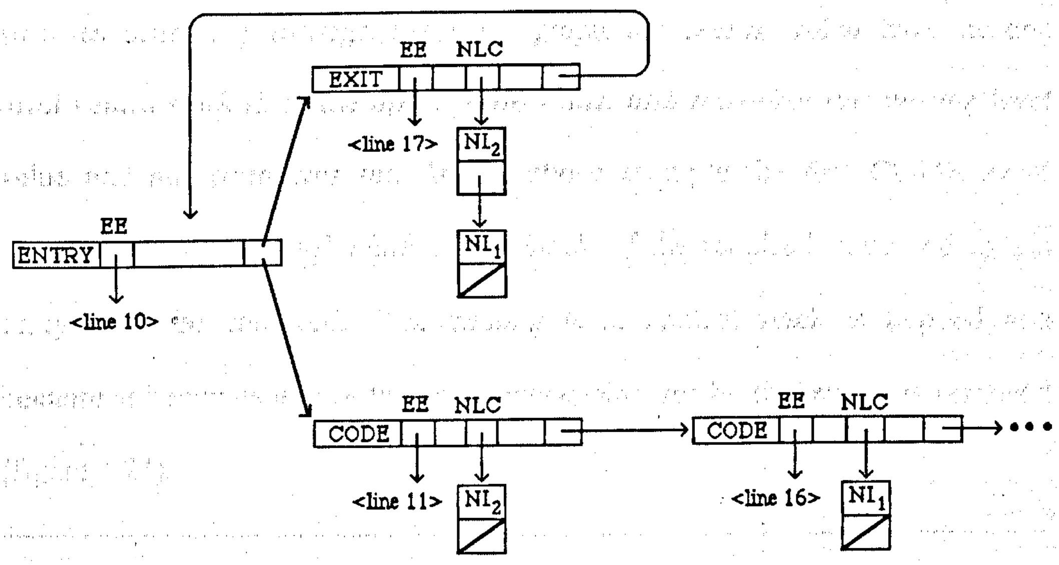

Figure 4.23

stack is popped empty, implying the primitive is not required any more. Similarly, primitives need only be set when a nesting level value is stacked on an empty nesting level control stack. Thus, in the above example, a second primitive is not required on the EXIT point of the code fragment.

The stacking of the nesting level enables the monitoring software to follow multiple traversals of the graph. The values stacked indicate the extent to which the predicate is satisfied at each of the levels.

If, in the above state of affairs, the code fragment is exited, then **evaluation** of the EXIT event takes place. For successful **evaluation** the trap must have occurred at a nesting level value of NL_2. If this is the case then the nesting level control stack is popped, but the primitive set on the exit point of the code fragment is not removed as the stack is not empty. A further function of **evaluation,** in the case of EXIT events, is the restoration of the event-graph to the state applying before the ENTRY event had occurred. This involves searching through the event-graph for events which have nesting level values stacked of the appropriate value, and removing this nesting level value and any primitives set. In the above example the first CODE event does have a nesting level value, on the stack, of the required value and so this entry must be removed. The nesting level control stack is popped and because it becomes empty the code breakpoint, set by this event, is removed (figure 4.24).

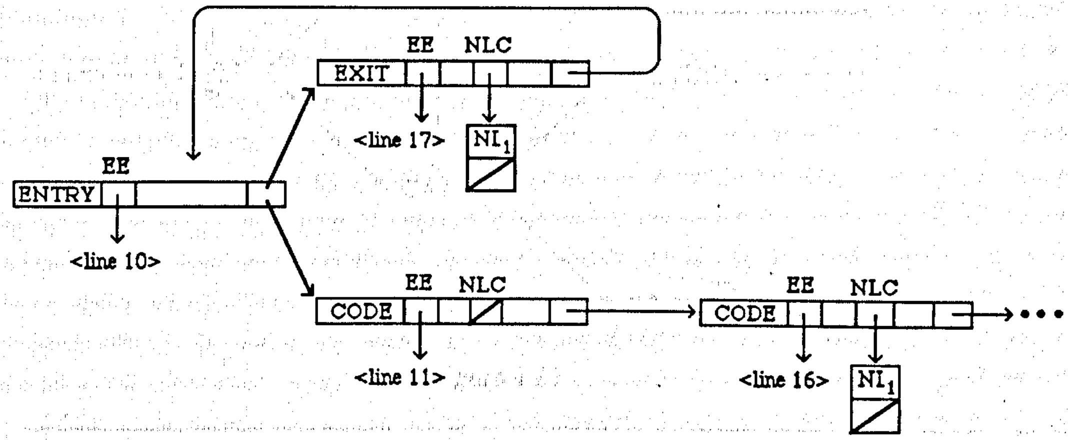

Figure 4.24

Example 3

WHEN B:A{/x} PERFORM <monitoring action>

When applied to the code fragment in figure 4.25 the monitoring action is performed when the global variable x is updated, within procedure A, but only if this is called from procedure B.

The event-graph representation of the above monitoring command is shown

```
int x ;
proc A()
{
   x = expr ;
}
proc B()
{
   A() ;
}
```

Figure 4.25

in figure 4.26.

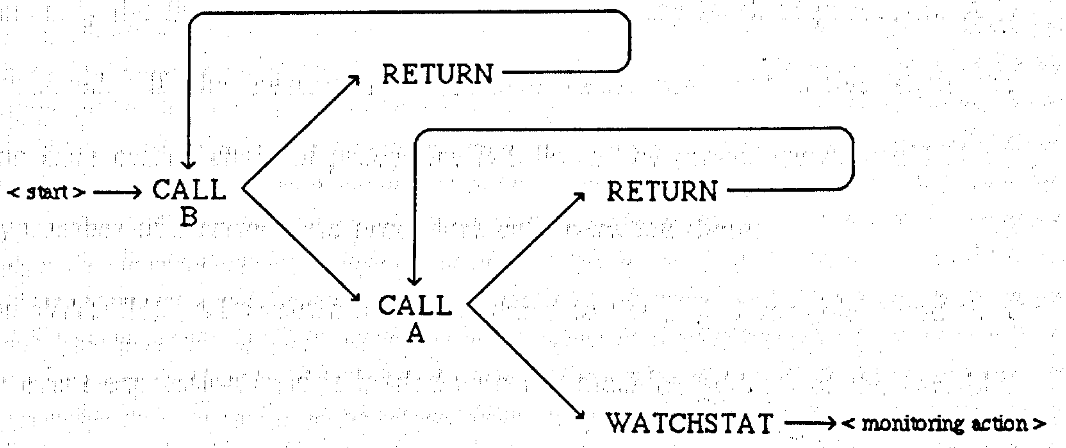

Figure 4.26

The CALL and RETURN events act in a similar way to the ENTRY and EXIT events of the previous example. Different types are used for procedures, however, as the nesting level is influenced by procedure calls and returns. Thus, whereas ENTRY and EXIT events "watch" for a nesting level of, for instance NL, CALL events are "watching" for $NL+1$ and RETURN for $NL-1$.

The first CALL event, which will be referred to as CALL-B, is used to monitor entry to procedure B. The event expression field of this event is loaded with the address of the first instruction of procedure B. The nesting level control and enable control fields produce the same effect as in the previous example for the ENTRY event.

The second CALL event, which will be referred to as CALL-A, is assigned the first instruction of procedure A to the event expression field. To preserve the procedure calling chain of procedure B followed by procedure A, the nesting level control field is active and the enable control field not active. Thus only the first occurrence, at a particular nesting level, of procedure A is monitored. If the nesting level control field were not active then the procedure calling chain of procedure B followed by procedure A would allow any number of intermediate procedure calls between them.

The WATCHSTAT event monitors updates to the global variable x and thus the event expression field is loaded with the machine address of the variable. Different monitoring effects are again seen depending on the setup of the enable control and nesting level control fields.

The nesting level control field of the WATCHSTAT event has a rather subtle effect on monitoring. If it is set, thus requiring a particular nesting level for successful **evaluation** of the event, then the monitoring action is performed for updates to the variable x, but only if they occur at the nesting level of procedure A; that is, the statement which causes the update of the variable is within the textual scope of procedure A. The other effect, caused by the field not being set, results in the monitoring action if procedure A is called from procedure B, and possibly followed by any number of intermediate procedure calls.

The two RETURN events are each set up to monitor only one occurrence of procedure return at a particular nesting level. In this example the RETURN events will always occur at the correct nesting level (assuming the code generated by the compiler adheres to the nesting of procedure calls) but, in general, CALL/RETURN pairs "watching" the same procedure can occur any number of times within the event-graph and so it is essential that the correct RETURN for a particular CALL is recognised. The RETURN events are required to monitor exits from the procedure identified in the associated CALL event. This could be achieved by monitoring the final instruction of the procedure. However, there may be many "final" instructions if the target programming language allows returns from procedure to be explicitly included as program statements. A better method may be to monitor the first instruction after the return from procedure has occurred. This enables the monitoring of a single machine address, but means there is no static machine address which can be assigned to the event expression field of the RETURN event. Instead, the return address is, in most architectures, to be found in the procedure stack frame environment. The access of the return address would thus be a function of the operating system modules (introduced in chapter 3) attached to the process control unit, or monitoring software, of the monitoring environment.

Traversal of the event-graph commences with the **examination** of the CALL-B event, resulting in a code breakpoint at the entry point to procedure B. Execution of the target program now resumes until the machine-level trap is taken and control passes, once again, to the monitor process. The CALL-B event is evaluated as a result of this leading to **examination** of the two successor events. **Examination** of the RETURN event, paired with the CALL-B event, places the current nesting level value (say, NL_1) minus one on the nesting level control stack, and sets a code breakpoint at the return address (say, RA_1) found in the stack frame environment. To allow for recursion the return address, at which the code breakpoint is set, must be stored in the event structure. The reason for this will become apparent later in this example. The local variable stack is used to hold the procedure return addresses (figure 4.27).

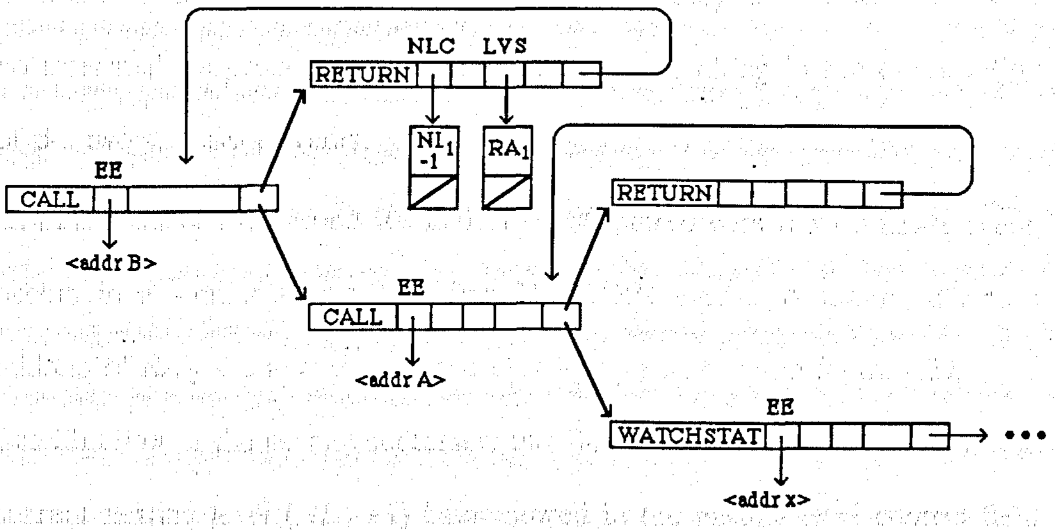

Figure 4.27

Examination of the second successor of the CALL-B event, the CALL-A event, places the current nesting level value plus one on the nesting level control stack, and sets a code breakpoint at the entry point to procedure A (figure 4.28).

If procedure A is entered then **evaluation** of the CALL-A event takes place,

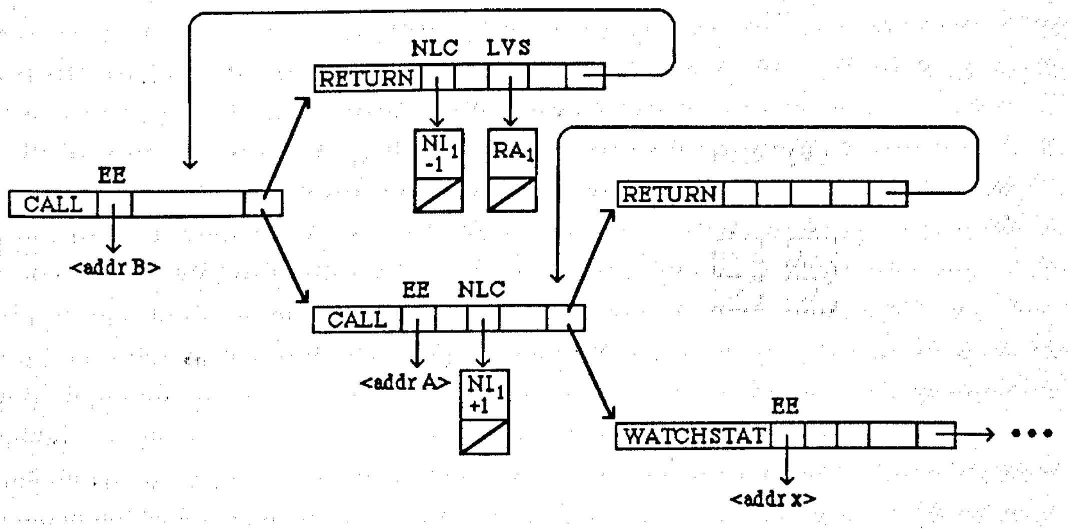

Figure 4.28

which checks the current nesting level value against that stored in the nesting level control field. **Evaluation** is only satisfied if they are the same and thus no intermediate procedure calls have occurred, resulting in the **examination** of the two successor events.

Examination of the second RETURN event, paired with the CALL-A event, occurs in a similar way to the first RETURN event. Assuming a return address of RA_2 the new state of the event-graph is shown in figure 4.29.

Examination of the second successor, the WATCHSTAT event, results in the current nesting level (NL_1+1) being stored in the nesting level control field and the address in the event expression field (machine address of global variable x) used to invoke a watchpoint primitive (figure 4.30). This traps execution when the address is updated.

There are now four traps set, one for each event except for the CALL-A event. Assuming a recursive call to procedure B occurs, then **evaluation** of

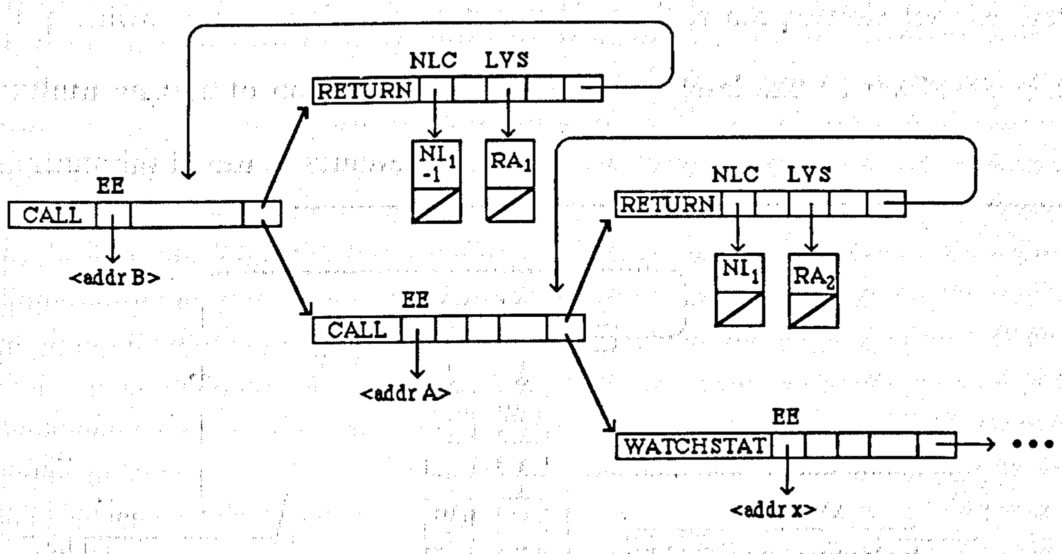

Figure 4.29

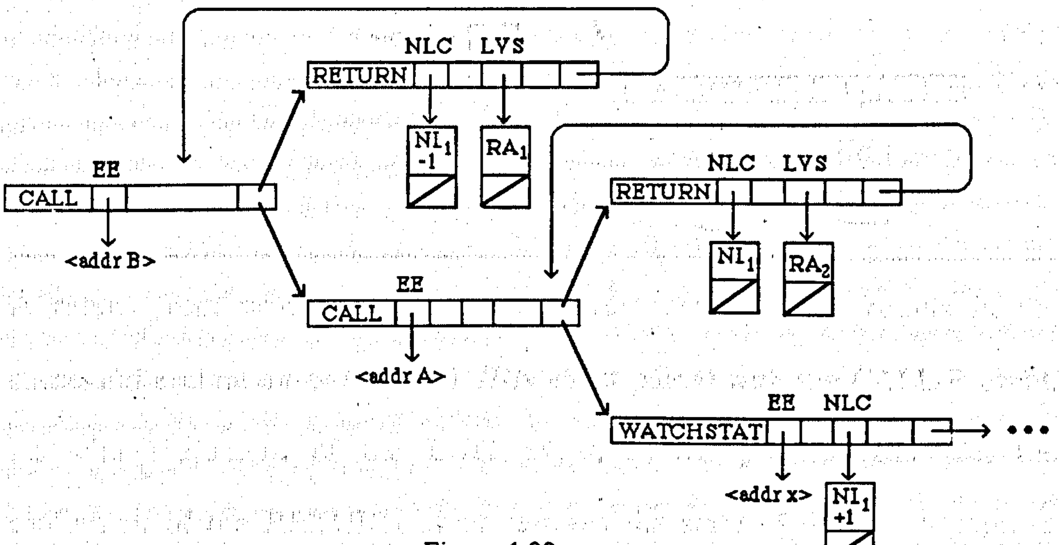

Figure 4.30

the CALL-B event occurs. This results in the **examination** of the successor events once again. This occurs in a similar way to the first time. Assuming a current nesting level value of NL_2 the state of the event-graph is shown in figure 4.31. As in the previous example the two threads of monitoring can be seen by inspection of the nesting level control field. However, a difference between the two **examination** phases of the RETURN event exists. Although a primitive is set on the new return address, it is not possible for the first return address to occur at the correct nesting level and so the breakpoint monitoring it, can be removed.

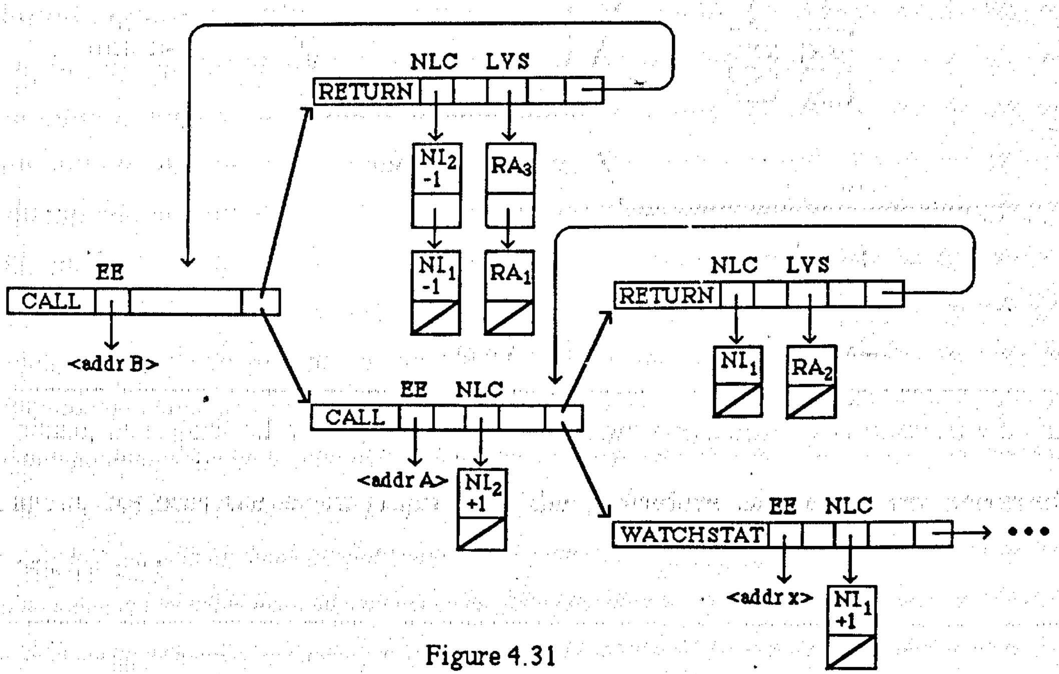

Figure 4.31

A return from procedure B, at the correct nesting level, results in the successful **evaluation** of the RETURN event paired with the CALL-B event. As with **evaluation** of the EXIT event, in the previous example, the **evaluation** of the RETURN event restores the state of the event-graph to that applying before the procedure call. Thus all events succeeding the CALL event, paired with the evaluated RETURN event, are visited and if they possess nesting level values of the appropriate value then these are removed, as are any primitives set by them (figure 4.32).

A return from procedure A at the correct nesting level would bring about **evaluation** of the RETURN event paired with the CALL-A event. A similar

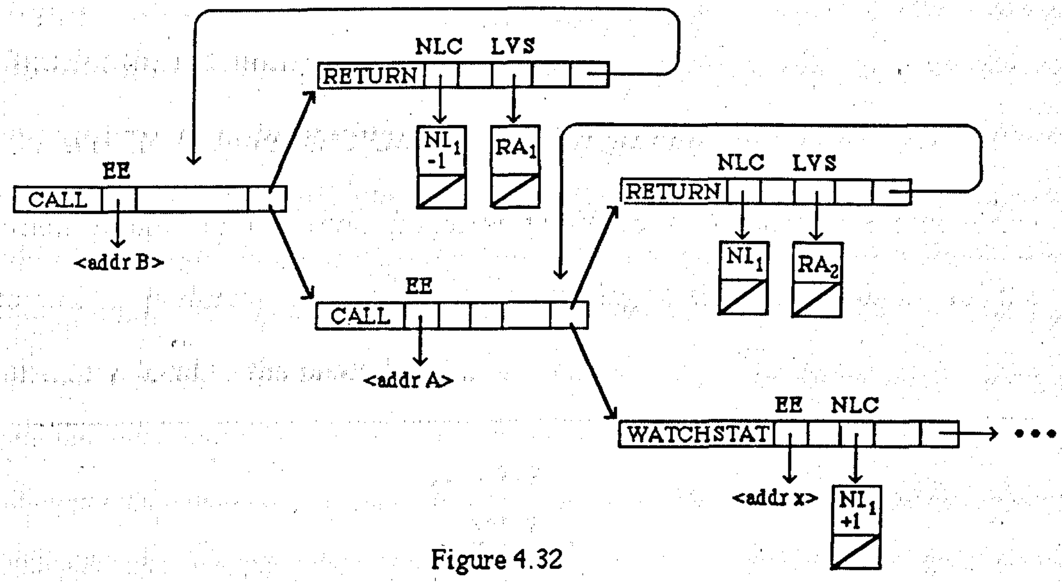

Figure 4.32

phase, as described above, for the return from procedure B is entered which, again, restores the event-graph as if the procedure call had never occurred (figure 4.33).

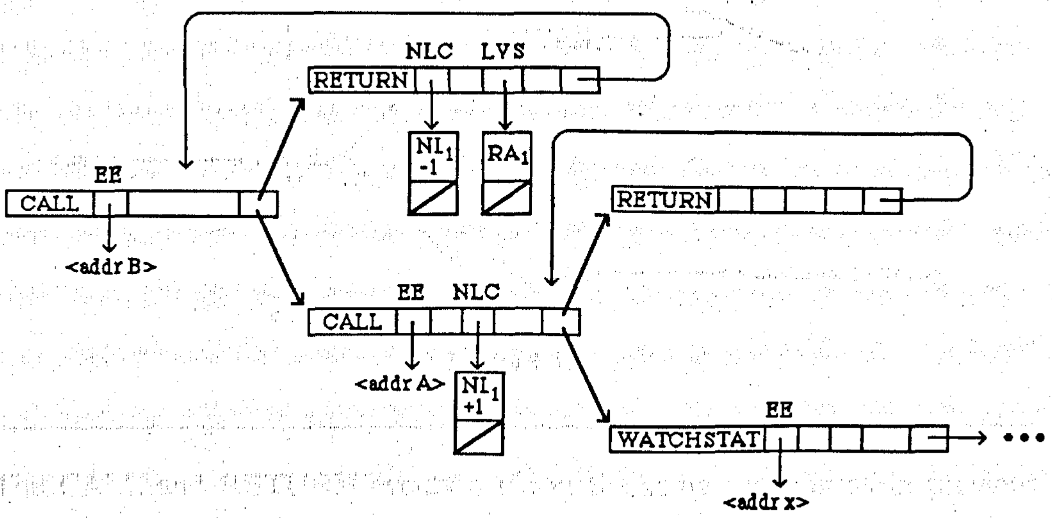

Figure 4.33

Example 4

Consider the monitoring command:

WHEN /X/y PERFORM <monitoring action>

When applied to the code fragment in figure 4.34 the above monitoring command performs the specified monitoring action for updates to the variable y, local to the procedure X.

```
proc X()
{
   int y ;
   y = expr ;
}
```

Figure 4.34

The event-graph representation of this command is given in figure 4.35.

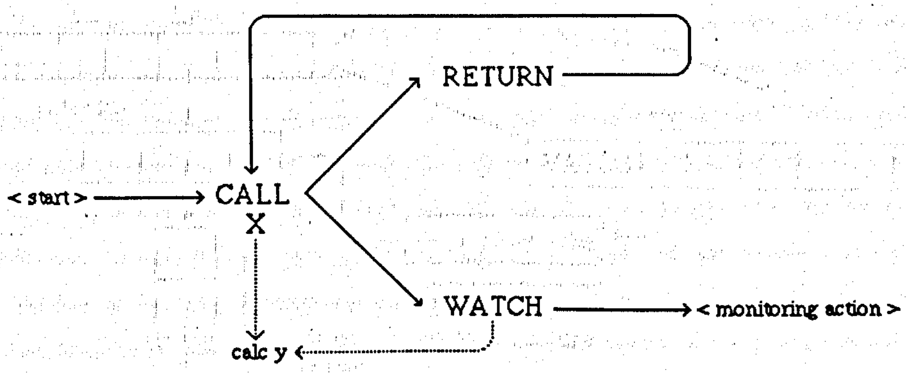

Figure 4.35

The CALL and RETURN events perform the same function as in previous examples; that is, restricting the rest of the predicate to the scope of the specified procedure. In addition to this the CALL event performs a *local action.* This is performed on each successful **evaluation** of the CALL event, and in this example the machine address of the local variable y is calculated and stacked for use by the WATCH event.

The WATCH event is similar to the WATCHSTAT event except that the event expression field holds a pointer to the stack of calculated addresses, and not a static machine address. A structure located between the WATCH event and the local action structure, called the *local variable access* structure, enables the user to specify options appropriate to monitoring local variables (figure 4.36). This structure consists of an *offset*, indicating how far down the stack of addresses the WATCH event looks for an address, thus allowing variables from a specific instance of a recursively called procedure to be included in a monitoring predicate; and a flag which indicates whether only a single instance of the variable is to be monitored, or all instances.

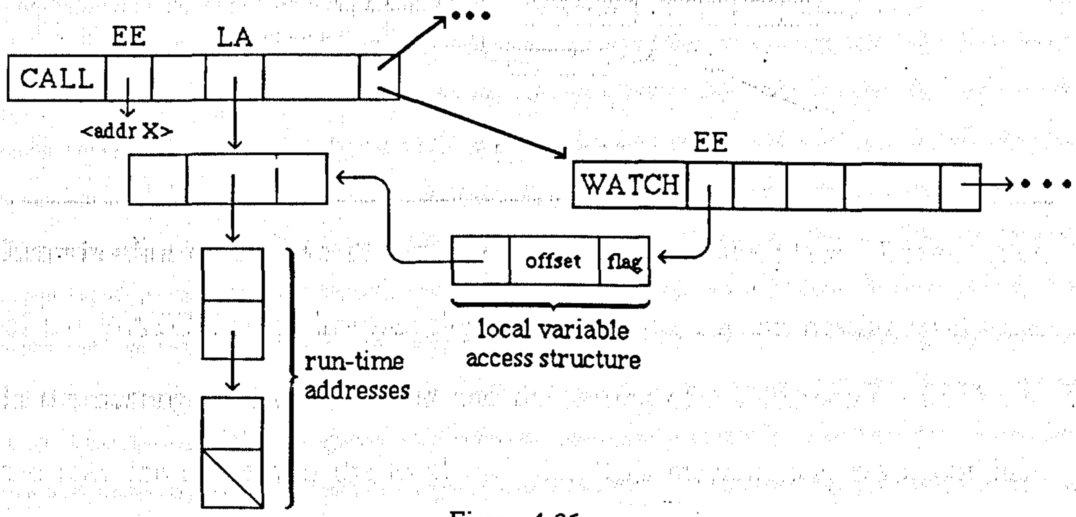

Figure 4.36

Traversal of the condition graph commences with the **examination** of the CALL event, which results in a code breakpoint at the entry point to procedure X. When the CALL event is evaluated, and before **examination** of the successor events takes place, the local action is performed. In this example, the offset in the local action structure (variable offset obtained from the symbol table) is added to the current stack frame pointer, and the result (say, LV_1) stacked in the same local action structure (figure 4.37). It may be the case that more than one variable, local to a procedure, is found in the monitoring predicate, in which case, the local action performs the above calculation and stacking function for each local action structure in the list.

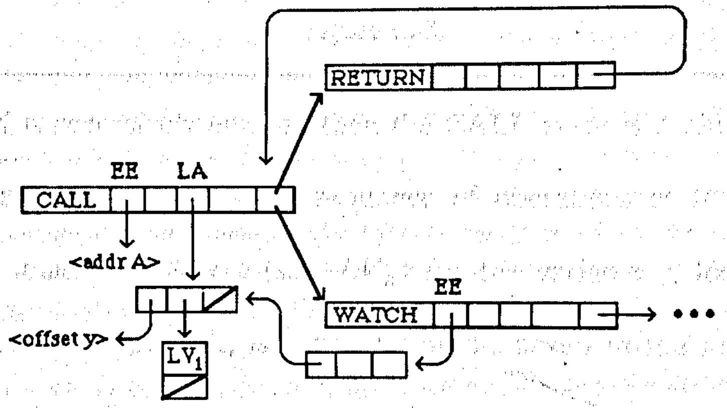

Figure 4.37

Examination of the two successor events can now take place. **Examination** of the WATCH event involves the stacking of the current nesting level value in the nesting level control field and the setting of a watchpoint primitive at the machine address of the local variable. This machine address is obtained from the local action stack via the local variable access structure. The offset in this structure is used as an offset from the top of the local action stack. Thus, if this offset was anything other than zero then no machine address would be available and consequently no primitive set. A final function of **examination** is the copying of any machine address accessed, to the local variable stack. The resultant event-graph is shown in figure 4.38.

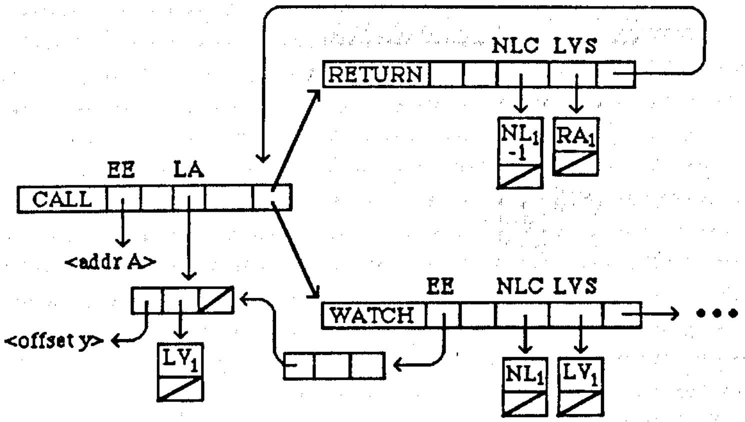

Figure 4.38

If procedure X is recursively entered then the CALL event is evaluated for a second time, resulting in the same sequence of operations as for the first **evaluation.** A machine address (say, LV_2) for the variable y, local to the recursively called procedure X, is calculated and stacked in the local action stack. **Examination** of the two successor events results in the **examination** of the RETURN event which proceeds as in the previous example. The **examination** of the WATCH event, for the second time, stacks the nesting level value and accesses a machine address via the local variable access structure. If the flag, in this structure, indicates that only one instance of the variable is to be monitored, then the primitive set on LV_1 is removed and a watchpoint set on the new machine address LV_2. If, however, the flag indicates that all instances of the local variable are to be monitored, then the first primitive is left active. The event-graph resulting from the above and assuming a new nesting level of NL_2 and return address of RA_2 is shown in figure 4.39.

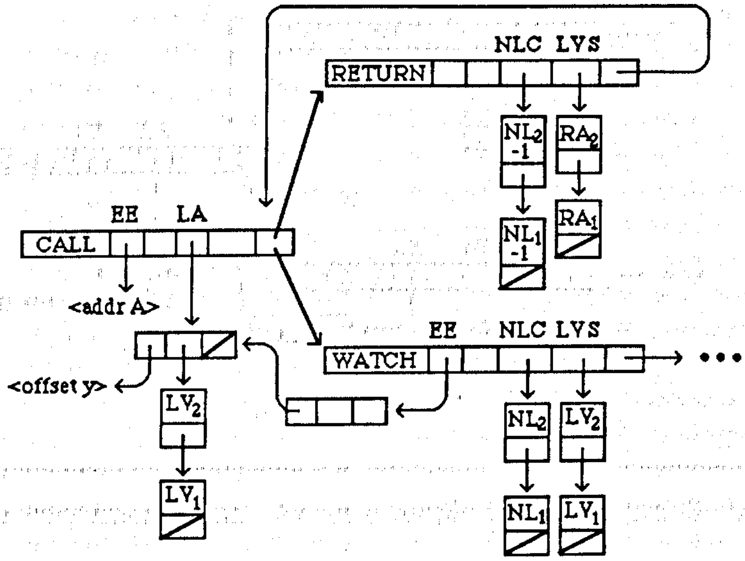

Figure 4.39

If an update to the monitored variable occurs then the WATCH event is evaluated, and if successful, results in the monitoring action. Successful **evaluation** of the WATCH event occurs in the same way as for other events, in that, the trap must be taken at the correct nesting level, if the nesting level control field is active.

Evaluation of the RETURN event occurs when a return from procedure X occurs. The reverting of the event-graph to a previous state must now occur, and in a similar way to the previous examples (figure 4.40).

The necessity for the copying of the machine address, of the local variable, is not apparent in this example, as the local variable stack is a direct copy of the local action stack. In general, however, this is not the case. Any events between the producer event (CALL) and the consumer event (WATCH) can cause the stacking of machine addresses at the producer event which are

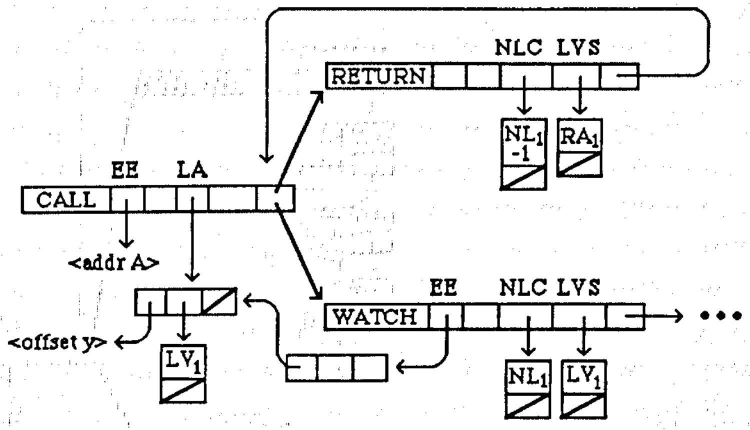

Figure 4.40

irrelevant to the consumer event. As an example the event-graph shown in figure 4.41 represents local variable monitoring where an event occurs between the consumer event and the producer event.

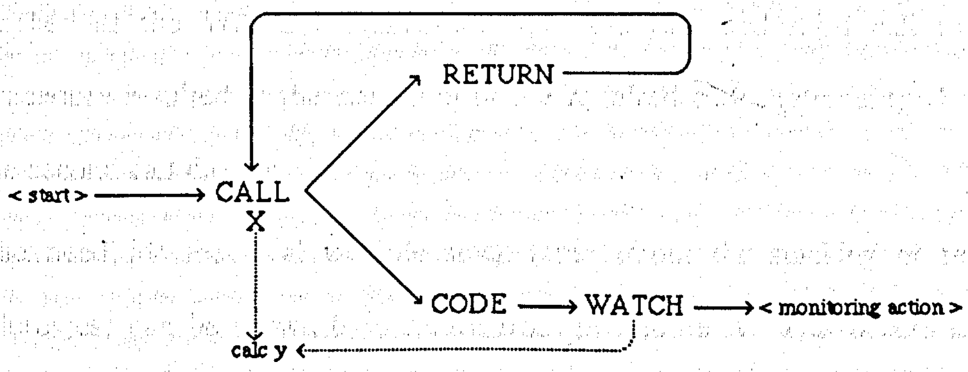

Figure 4.41

Figure 4.42 shows the above event-graph part traversed.

From the state of the stacks in this event-graph it is possible to infer the flow of execution (as it pertains to the monitoring command) which led to the current state. The three machine addresses stacked at the CALL event show that procedure X was twice called recursively. The two machine addresses

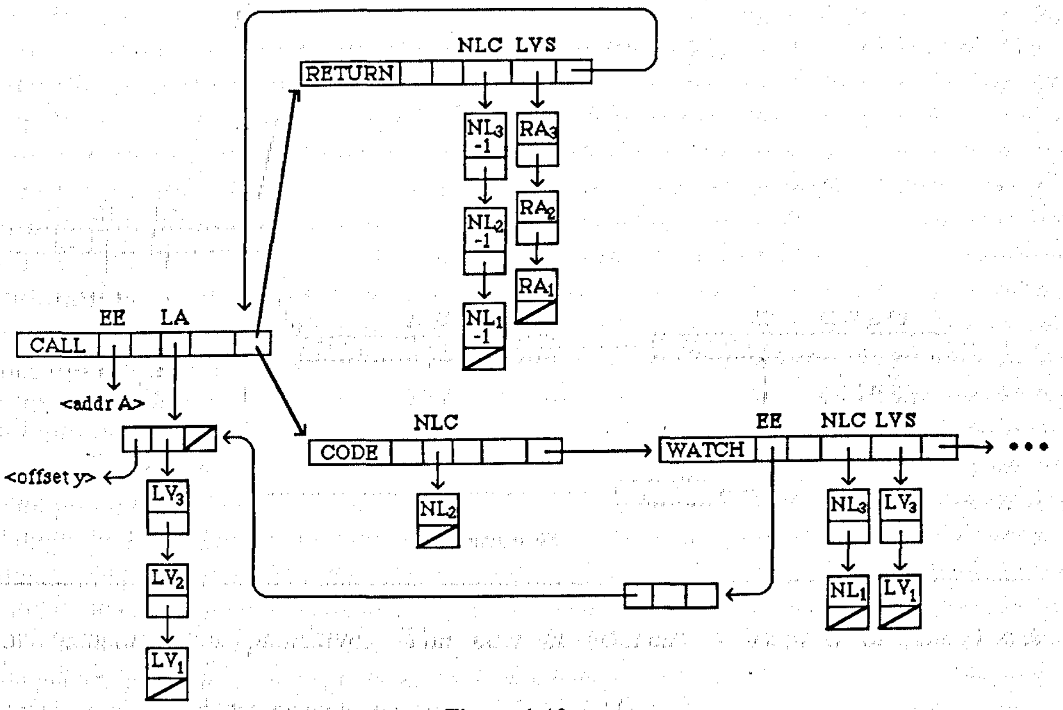

Figure 4.42

stacked at the WATCH event show that the first and third call to the procedure resulted in the execution of the required code statement, whereas the second call did not.

The need for the local variable stack (apart from the stacking of return addresses) can be shown by a return from procedure X. This occurs if the trap at return address RA_3 is taken resulting in the popping of stacks and removing of primitives, to give the event-graph shown in figure 4.43.

If the flag, in the local variable access structure, indicates that only one instance of the variable is to be monitored at any one time then, when removal of the primitive at address LV_3 occurs, a primitive must be reinstated on a former address. As can be seen from the event-graph in figure 4.43 the local action stack does not give the correct machine address

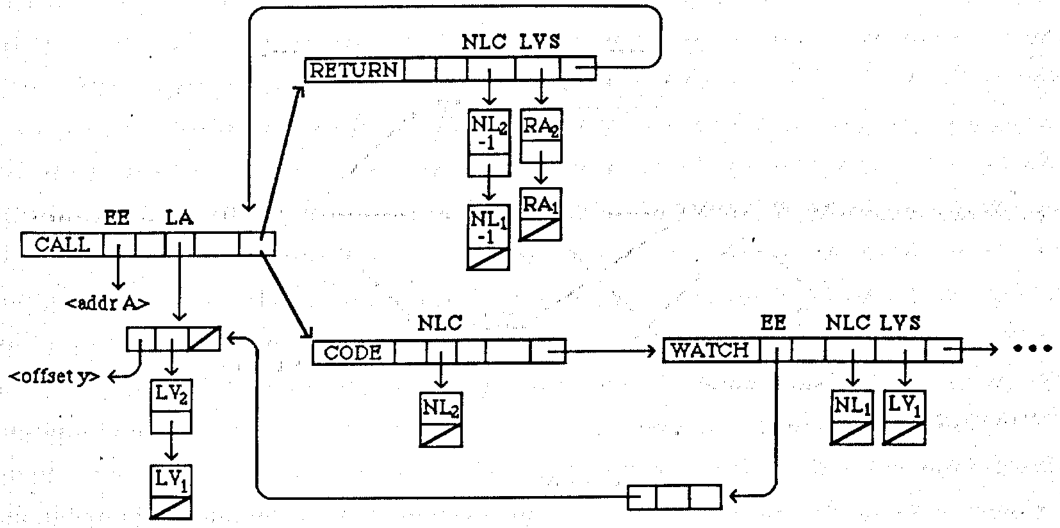

Figure 4.43

for reinstating a primitive. The correct address is located on the local variable stack of the consumer event (WATCH).

Example 5

Consider the monitoring command:

WHEN /A/x = /y + 1 PERFORM <monitoring action>

When applied to the code fragment in figure 4.44 the monitoring action is performed when the variable x, local to procedure A, is equal to the global variable y plus one. The event-graph representation of this is shown in figure 4.45.

```
int y ;
proc A()
{
   int x ;
   x = expr ;
   y = expr ;
}
```

Figure 4.44

The CALL, RETURN, WATCH and WATCHSTAT events examine and

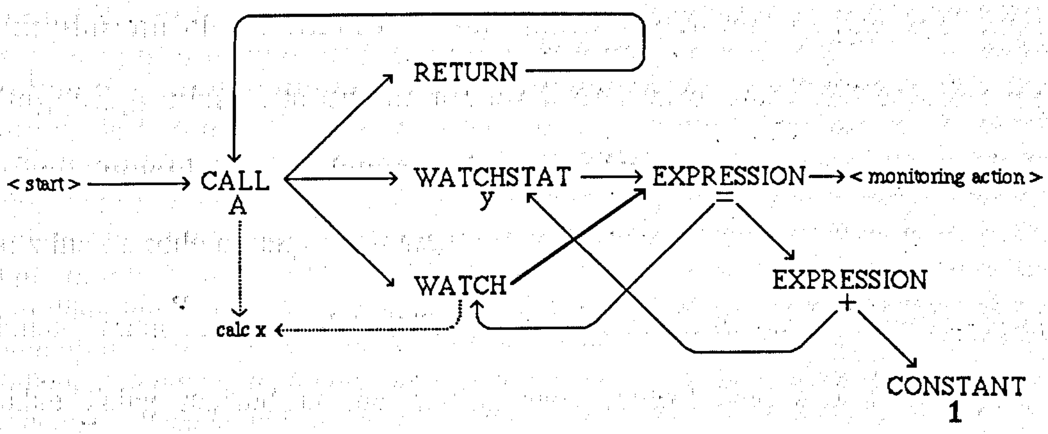

Figure 4.45

evaluate in the same way as previous examples. The EXPRESSION and CONSTANT events arc, however, processed in a different way to the other event types. Neither of these events set monitoring primitives and are thus never evaluated. **Examination** of the EXPRESSION event occurs when a variable is updated, and results in the checking of the expression in the monitoring predicate (that is, the expression tree in the event-graph). If the expression tree yields a result of *true* then the monitoring action is performed, otherwise control is passed back to the target process and execution continues.

Traversal of the event-graph commences with the **examination** of the CALL event, which results in a code breakpoint at the entry point to procedure A. When this trap is taken the CALL event is evaluated, which is successful if the trap is taken at the required nesting level, or if the nesting level control field is not active. Successful **evaluation** causes the stacking of a machine address (say, LV_1) in the local action stack and leads to the **examination** of the three successor events.

Examination of the RETURN event places the current nesting level value (say, NL_1) minus one on the nesting level control stack and accesses the current procedure stack frame environment for a return address (say, RA_1), at which a code breakpoint is set.

Examination of the WATCHSTAT event results in the current nesting level value being stacked in the nesting level control field and a watchpoint primitive being set on the static address in the event expression field.

Examination of the WATCH event involves the stacking of the current nesting level value in the nesting level control field and the setting of a watchpoint primitive at the machine address of the local variable, which is obtained from the local action stack via the local variable access structure. The resultant event-graph is shown in figure 4.46.

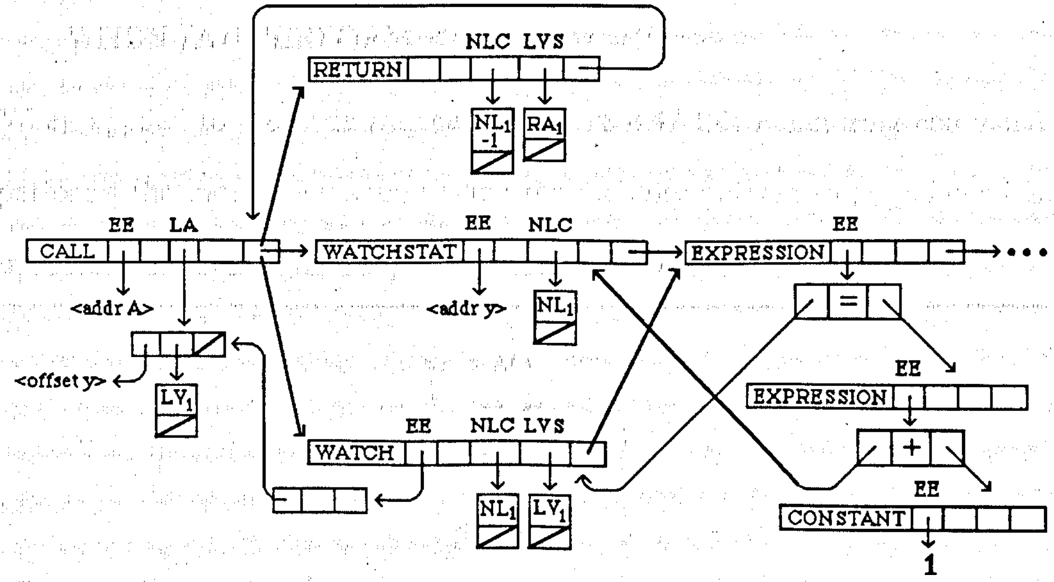

Figure 4.46

A return from, or recursive call to, procedure A is handled in exactly the same way as in the previous example. However, an update to either the global variable y or the local variable x results in the **examination** of the first EXPRESSION event. This involves traversing the expression tree, checking each of the logical or arithmetic subexpressions. Constant events return the constant value stored in the event expression field; WATCHSTAT and WATCH events return the value stored in the machine address of the event expression field or local variable stack, respectively; and EXPRESSION events return the result of applying the operator to the operands. In the above example the **examination** of the EXPRESSION event checks the value of the local variable x against the value of the global variable y plus one. Only if the first EXPRESSION event returns *true* is the monitoring action performed.

Example 6

WHEN /A/i PERFORM <monitoring action>

When applied to the code fragment in figure 4.47 this monitoring command performs the monitoring action when the memory pointed to by pointer i is updated. The event-graph representation of this is shown in figure 4.48.

```
proc A()
{
    int *i ;
    i = malloc() ;
    *i = expr ;
}
```

Figure 4.47

The CALL, RETURN and WATCH events examine and evaluate in the

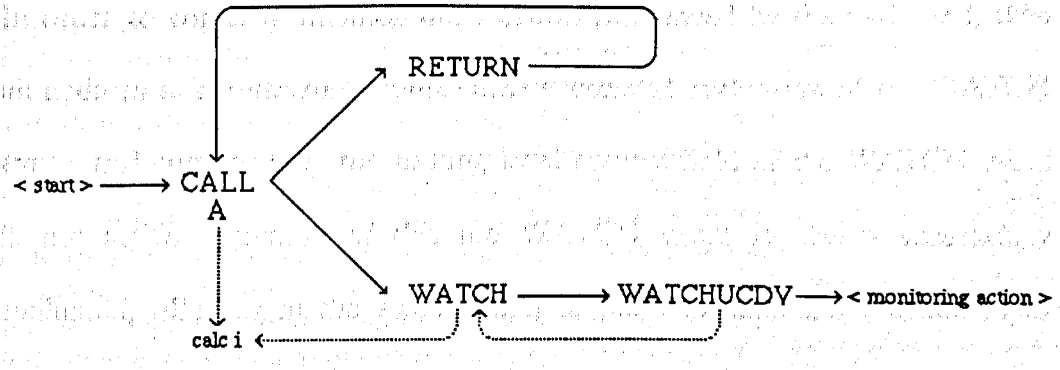

Figure 4.48

same way as previous examples. The WATCHUCDV event monitors updates to the memory location referenced by the address, obtained via the event expression field.

Traversal of the event-graph commences with the **examination** of the CALL event, which results in a code breakpoint at the entry point to procedure A. When this trap is taken **evaluation** of the CALL event occurs, resulting in the same sequence of events as the previous examples. After **examination** of the two successor events the state of the event-graph is shown in figure 4.49.

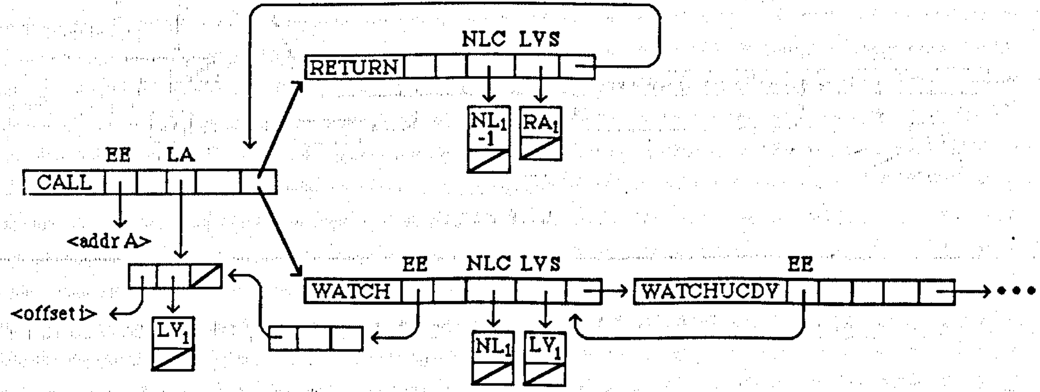

Figure 4.49

An update to the variable i results in the **evaluation** of the WATCH event. In order to correctly monitor the location referenced by the variable i, then all updates to the variable i must cause successful **evaluation** of the WATCH event, and consequently the nesting level control field of the WATCH event is not active. Because of this the WATCH event is always successfully evaluated, resulting in the **examination** of the WATCHUCDV event. This involves storing the current nesting level value in the nesting level control field, and accessing a machine address via the event expression field. The machine address accessed is the top of the local variable stack of the event referenced by the event expression field. However, this address is the address of the pointer and must be dereferenced to get the location referenced by the pointer (say, $*LV_1$). A watchpoint is set on this location, which is also stored on the local variable stack (figure 4.50).

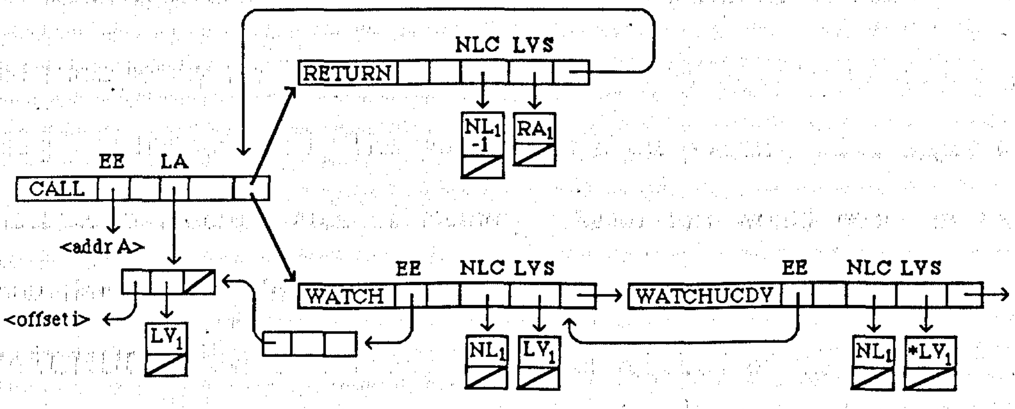

Figure 4.50

A recursive call to procedure A causes a stacking of the new machine address for the variable i and also the **examination** of the two successor events. The state of the event-graph after this is shown in figure 4.51.

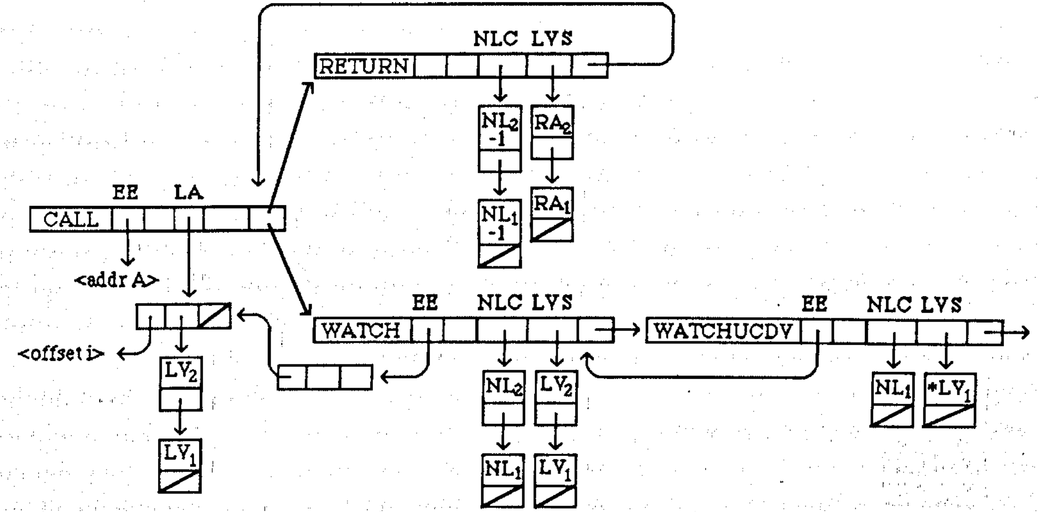

Figure 4.51

Assigning memory to the latest instance of the variable i causes the **evaluation** of the WATCH event. This results in the **examination** of the WATCHUCDV event which, in turn, leads to the stacking of the new location referenced by the address in the WATCH event (figure 4.52).

The traps set by the WATCH event are left active in order to catch any changes to the variable i. This would occur if the pointer i was changed to reference a different area of memory. Again this would result in the **evaluation** of the WATCH event and the subsequent **examination** of the WATCHUCDV event. However, the effect required of the **examination** phase is different in this case. Instead of stacking another address, one of the addresses on the stack must be changed. This occurs for user controlled dynamic variables because, as described in chapter 3, they release the data component, of the process state, from a purely stack-like structure.

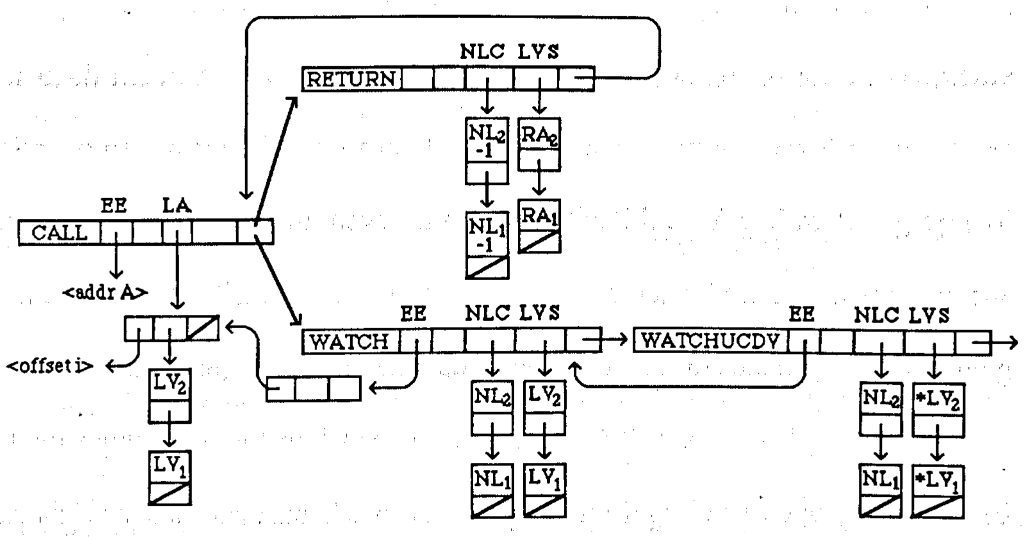

Figure 4.52

Examination of the WATCHUCDV event compares the number of addresses stacked in the local variable stack of each of the WATCH and WATCHUCDV events and, if they are equal then the **evaluation** of the WATCH event was due to the reassignment of the pointer, and not the introduction of a new pointer. However, if the stacks are unequal in size then it is a recursive call to procedure A, and the introduction of a new variable, which has caused the **evaluation** of the WATCH event.

4.4. Summary

In this chapter we have described three levels of monitoring: the primitive level, the abstract level, and the conditional level. Predicates at the conditional level define the monitoring of high-level concepts such as procedure-calling chains, local variables, user controlled dynamic variables and also process state information which is not generally preserved within the process state. A conventional method of implementing the conditional level would probably involve checking the predicate at regular intervals such as after each machine instruction. This can, however, result in an unacceptable performance overhead. In order to monitor a conditional level predicate as efficiently as possible we have introduced the idea of a directed graph of abstract level predicates or **events.** These events, when monitored in the specified sequence, monitor the conditional level predicate. By defining thirteen types of event and two phases: examination and evaluation, we have been able to demonstrate the monitoring of the high-level concepts indicated above. Event t.pes, primitives used and the actions taken during examination are summarised in table I.

The types of predicate which the event-graph supports are to a certain extent supported by an abstraction mechanism described by Lazzerini and Lopriore [Lazzerini89]. This enables a programmer to construct abstract predicates, for monitoring, from a number of simple predicates. Simple predicates can include target variables and *instruction address* (_ia) variables. The debugging system associates an _ia variable with a monitored block of the target program; blocks being denoted by a colon separated list of identifiers, indicating the nesting structure of the block. During execution the _ia variable is assigned the statement label (automatically assigned by the system and denoted by $1,$2..$^) of the currently executing statement. When execution leaves a particular block the _ia variable retains the label of the final statement to be executed. This enables control flow to be monitored where the path taken cannot be determined by simple inspection of the

Table I

Event Type	Primitive Used	Action During Examination
ENTRY	code breakpoint	Store *NL*, uses 1 primitive
EXIT	code breakpoint	Store *NL*, uses 1 primitive
CALL	code breakpoint	Store $NL+1$, uses 1 primitive
RETURN	code breakpoint	Store $NL-1$, uses multiple primitives
CODE	code breakpoint	Store *NL*, uses 1 primitive
WATCH	watchpoint	Store *NL*, uses multiple primitives depending on flag in local variable access structure
DATA	data breakpoint	Store *NL*, uses multiple primitives depending on flag in local variable access structure
WATCHSTAT	watchpoint	Store *NL*, uses 1 primitive
DATASTAT	data breakpoint	Store *NL*, uses 1 primitive
WATCHUCDV	watchpoint	Store *NL*, uses multiple primitives
DATAUCDV	data breakpoint	Store *NL*, uses multiple primitives
EXPRESSION	-	Do not store *NL*, Uses no primitives
CONSTANT	-	Do not store *NL*, Uses no primitives

process state. As an example consider the monitoring of two statements L1 and L2 which must be executed in that order for the predicate to be satisfied. The required monitoring commands are:

conditional CL1 = (B_ia == L1)
conditional CL2 = (B_ia == L2)
conditional on CL1L2 = CL1 & CL2
origin CL1L2 at CL1
break on CL1L2

The above commands set simple predicates "watching" the execution of each of the two statements, and a compound predicate which links these simple predicates. The **origin at** command prevents CL1L2 from becoming true if an event connected with CL2 occurs before an event connected with CL1.

In order to update the _ia variables a transfer of control to the monitor is required after the execution of each source statement, in each of the blocks involved in a predicate. Additionally the monitoring of a program involving recursion, and consequently the following of multiple instances of a predicate, appears not to be supported.

There are problems with the event-graph implementation, the first and not least of which, is the possibly large numbers of primitives which can arise from even a simple graph. This requires an efficient implementation of the monitoring primitives if execution rates are not to be lowered to the point where the system becomes unusable.

A second problem which involves performance degradation is that of unnecessary transfers of control. This occurs, for example, when monitoring procedure calling chains and also arises in a basic monitoring system. The problem arises, in the event-graph system, when the first procedures in the chain are frequently called but do not call the next procedure in the list. Unnecessary transfers of control arise with the basic monitoring system when the final procedure in the chain is called frequently but the sequence of calls, to fulfil the procedure calling chain, have not occurred. If the performance degradation associated with the event-graph system were to become intolerable then it might be possible to combine both methods in order to minimise performance degradation.

We now examine the possible implementations of the three monitoring primitives, attempting to minimise the performance overhead associated with them, thus allowing the event-graph system to invoke as many primitives as required.

5. Architectural Support

5.1. Introduction

Architectural support for monitoring implies using existing architectural features or adding extra hardware and/or firmware to decrease the performance degradation caused by monitoring. Much of the literature proposing architectural support for monitoring addresses the problem of performance interference in a real-time environment. Real-time software differs from other software in that it is functionally dependent on time, usually interacting with external devices or objects. For this reason real-time monitoring systems must not introduce delays into target process execution. There are, however, two types of delay: bounded and unbounded [Plattner81]. A *bounded* delay occurs when the delay is independent of the number of target program statements executed, but may well depend on the number of predicates to be monitored. An *unbounded* delay, however, is one which grows indefinitely with the duration of the target process, each predicate evaluation adding to the total delay. Although the monitoring system must not incur unbounded delays it is possible for real-time monitoring to endure bounded delays.

Because the target process is shared between the monitor processor and target processor the system cannot allow mutually exclusive access of the target process [Plattner84], as this incurs unbounded delays. For this reason monitoring systems for real-time software access the target process state, without performance interference, by capturing process state information at some point in the hardware where it is available (for example, the pins of the CPU or memory bus). However, it is the interpretation of the hardware activities in terms of source-level events that introduces delays. This is tackled, in the literature, by specialised hardware support for logic analysers [Plattner84] [Bemmerl86] [Rijks87], and is discussed in a later section.

Software which does not have strict timing constraints (as real-time software does) does not require a monitoring system which adheres to bounded delays. Because of this, the interpretation of hardware activities to source-level events can be relegated to software and architectural support provided only for monitoring primitives. However, monitoring must incur a minimal performance overhead if the system is not to become unusable, and so the implementation must strive to incur a delay, associated with a monitoring function, only if that function is active [Johnson82]. We also wish to minimise the *continual* performance overhead imposed by the presence of an active monitoring function, such as a breakpoint; the intermittent cost of responding to a breakpoint invocation is, conversely, not so significant in a non-real-time system.

In this chapter we examine possible methods of implementing support for monitoring, with different architectural resources, and describe an implementation for use in a virtual memory environment that incurs a low continual overhead.

5.2. Implementations

5.2.1. Simulation

The three primitives of the monitoring environment outlined in chapter 3 are the code breakpoint, the data breakpoint and the watchpoint. The code breakpoint traps to the monitor process when the instruction at a specified location is executed. A similar trap occurs for the data breakpoint when a specified memory location is accessed and for the watchpoint when the location is updated.

It is possible to implement the three primitives with no architectural support for program monitoring. In this case the implementor can resort to simulation of the underlying machine [Huang84]. Predicate evaluation for the primitives is thus performed in software as is emulation of machine instructions. Simulation is the most flexible method of implementation [Saal72] allowing easy tailoring to a specific need [Melvin86]. There are, however, disadvantages to implementation via simulation. The performance overhead can be considerable; Agarwal, Sites, Horowitz [Agarwal86] and Melvin [Melvin86] quote a possible 1000:1 execution overhead. The production of a functionally accurate simulator is also non-trivial and is thus a source of expense. The serious drawback, however, is the performance degradation because the user will dispense with the monitoring system if it runs too slowly [Johnson82].

5.2.2. Non-intrusive hardware monitoring

It is obvious from the previous section that simulation is not usually a practical implementation proposal and is useless in a real-time environment. The monitoring of real-time software requires some form of non-intrusive hardware support so that monitoring can proceed with no delays incurred in the target process. Thus the timing dependencies with the target program are not disturbed.

For some time logic analysers have been used to extract information from executing programs by collecting machine state information on the machine bus via a set of signal probes [Fryer73] [Lloyd80] [Gentleman83]. The logic analyser is then able to compare or store the detected signals as appropriate. The collection and storage of machine state information is performed with no delay to the executing program. Hamilton [Hamilton83] describes a system whereby both event occurrences and duration for blocks of memory and sections of code can be measured. It allows a number of modes of operation including program activity, memory activity and module duration. A debugging monitor for the Bell System 1A processor [Witschorik83] allows real-time monitoring by taking snapshot views of the processor state information. Storage in the system enables the recording of 512 snapshots. However, both of these systems record and present to the user information at the machine level rather than at the program language level.

The ASDS memory bus monitor [Lyttle90] provides a symbolic debugging system for real-time embedded Ada software. The halting of the target process is performed by the bus monitor placing appropriate signals on the memory bus. Real-time constraints are met, however, by restricting the monitoring of variables to statically declared variables only. Thus, ASDS cannot monitor variables local to procedures or dynamic variables under the control of the user.

As it becomes more feasible and economical to code real-time systems in a high level language [Hill83] the real-time monitor must incorporate high-level monitoring techniques. In a lot of cases monitoring at the level of the basic block is sufficient to infer program activity, thus avoiding the re-creation of the high level view, which introduces delays for logic analysers [Plattner81]. RED [Hill83] is a real-time monitoring system for use with a high-level statement orientated language. During execution the entry to each basic block causes an entry into a history record indicating the basic block and the exact time of entry. After execution the history record together with compile-time information can be used to create a source level display showing which blocks were executed and at what time. A hardware probe implementation is described whereby the target memory is expanded to contain tag bits denoting which instructions begin basic blocks.

The SOVAC system [Lemon79] provides breakpoints on access of location, value in location and event counter reaching pre-defined value. High speed data selection and logging places information in a FIFO queue for later access by a software front end. As an instruction executes, the results of the previous instruction are stored in the FIFO, thus the monitoring processor must be able to store all relevant information in the time it takes to execute the shortest instruction.

The detection of a high-level event often requires the detection of a number of sub-events [Gentleman83]. This is a similar principle to that described in chapter 4 where high-level events are broken up into a directed graph of sub-events. When monitoring variables local to procedures it is the entry to and exit from the particular procedure which constitute the extra sub-events, whilst for user controlled dynamic program variables it is the call to the memory allocation routine. The HP64340A [Small85] from Hewlett Packard is a software analyser add-on which can follow the execution of a program by using cross-reference data, special hardware and post-capture data reduction. Low-level address and data recognisers are armed with, for example, the procedure entry and stack-setup instruction address to monitor a local variable. The analyser allows nine hardware breakpoints and the counting of 256 different events.

Circuitry for the monitoring of local variables is proposed by Goossens, Rijks, Tiberghien and Vermeesch [Goossens83] [Rijks87]. The idea is to provide a tag memory which removes the need for a second filter processor which has a speed an order of magnitude higher than the target processor. This tag memory is a normal memory of the same length as the memory of the system under test. Each entry of this tag memory can hold an operation code which indicates the action to be performed when the target system's processor places the particular address on the bus. When monitoring local variables entry to the appropriate procedure is "watched" for, resulting in the stack base register being stored for later use. Subsequent accesses of memory result in this stack base register being subtracted from the address on the bus. A relative tag memory in the analyser circuitry indicates any operation to be performed for the particular access. To enable the monitoring of variables local to recursive procedures the storage for the base register is a stack structure and the activate/deactivate block, which indicates when execution is within the specified procedure, is transformed into a counter which is incremented on procedure entry and decremented on procedure exit. However, the monitoring of variables in different procedures requires that the circuitry be replicated for each block with a variable to be monitored. A tag system is also used for the monitoring of user controlled program variables. An appropriate operation is associated with the tag entry corresponding to the locations of the instructions for the calls to the memory allocation routine. When this call occurs the bus is monitored for an assignment to the dynamic variable. This assignment associates storage area to the dynamic variable thus providing the monitor with the memory location and thus the tag location of the variable.

The monitoring of instruction execution and variable update is also examined by Bemmerl [Bemmerl86]. Circuitry is described which provides code breakpoints and data breakpoints without slowing the target process. Local and dynamic variables are monitored by storing the run-time address in the hardware monitor when the procedure prologue instruction is executed and removed when the corresponding procedure epilogue instruction is executed.

The approach adopted by Plattner and Nievergelt [Plattner81] [Plattner84] also uses a second memory with size equal to the target memory. However, in this case the memory is a copy of the target memory and is termed the phantom memory. To decouple the target processor from the monitor processor and consequently the target memory from the phantom memory a FIFO queue is inserted between them. A conventional low-level "bus listener" is used to detect signals on the target bus and places the necessary information in the FIFO. At the other end of the FIFO is the monitor processor which rebuilds an image of the target state in the phantom memory. Predicate evaluation by the monitor processor results in the locking of the FIFO queue whilst the phantom memory is accessed. During this interval any memory transactions on the target system bus are queued in the FIFO. Thus, the queue must be large enough to hold the information queued whilst the FIFO is locked for a reasonable period of time. Secondly, the monitor processor must be of a speed which is capable of clearing the queue once it is unlocked. To speed up the monitor process a breakpoint bitmap is connected to the output of the FIFO, which reports to the monitor any memory transactions referencing a location belonging to a previously defined set of memory locations.

5.2.3. Built-in hardware support

Simulation and logic analysers represent the extremes of support for program monitoring. In the following we look at ways of implementing architectural support for the three monitoring primitives without the cost of expensive circuitry. However, the implementation of primitives which trap absolute addresses requires that the high-level functionality is provided by the monitoring software. The architectural support thus reduces the performance degradation associated with predicate evaluation at the machine level.

The primitive requiring the least sophisticated architectural support is probably the code breakpoint. If the implementor has the ability to alter the process instruction space code breakpoints can be provided which allow all other regions of the program to execute at a normal rate. The location at which the breakpoint is required is accessed, the instruction found there saved within the software monitor, and a "jump" instruction stored in its place [Johnson82] [McLear82] [Abramson83]. The destination of the "jump" is the monitor routine. This method, however, requires that the monitor is a part of the target process instruction space. To avoid the interference that this causes, operating system support is required which allows the monitor to be executed as a separate process. This support takes the form of a trap instruction which causes suspension of the target process and a transfer of control to the monitor process. There are drawbacks to this method: it must be possible to modify the target instruction space and consequently it is not possible to set code breakpoints in ROM; sections of code cannot be shared amongst users, and the monitoring software must emulate the instruction replaced by the trap instruction.

The importance of the code breakpoint has led to many machines being built with architectural support for them. The IBM System/370 and the SPAM architecture [Johnson82] allow groups of instructions to be specified, resulting in a trap if an instruction in the group is executed. This facility is often implemented using bounds registers which are checked on instruction execution. Using only one set of bounds registers limits the user to one breakpoint region leaving the monitoring software to check traps to enable a finer granularity.

Extending main memory to incorporate a trap bit can be used to implement code breakpoints but this requires either special memory or a reduction in the useable instruction length. The COBOL virtual machine implementation on the NCR Criterion supports this facility but only allows breakpoints of paragraph granularity [Johnson82]. Associative memories and bitmaps may also be used to implement breakpoints but are likely to be costly and are thus not to be found in common use.

It is possible to implement code breakpoints using data breakpoints [Abramson83]. In this implementation it is the access of the instruction which causes the trap and not its execution. The problems of monitoring ROM and sharing code amongst users are removed for this implementation but other disadvantages take their place. In particular, accessing an instruction does not necessarily mean that it will be executed. This occurs, for example, in instruction caching where a number of instructions are loaded with one access. For this reason, and because the problems of sharing code and monitoring ROM are unlikely to be of significance at the development stage, the simpler method of replacing instructions by trap instructions appears to be the most satisfactory implementation for the code breakpoint primitive. Means of overcoming the performance degradation associated with the emulation of the replaced instruction will be examined in a later section.

Implementation of data breakpoints and watchpoints is not as simple as code breakpoints as it is not possible to replace the location contents with a trap value. Because of this many debugging and monitoring systems either cannot monitor data flow or have to resort to the use of code breakpoints and single stepping.

An often used implementation of breakpoints is the trapping to monitoring software after the execution of each instruction [Groll78]. A list of breakpoints is thus checked after each instruction. The VAX T-bit facility [Digital82] and the 68000 trace bit [Motorola82] cause a trap after the execution of each instruction allowing the monitoring software to take control and perform predicate evaluation. An interrupt-driven facility for trapping execution is described by Smith [Smith82]. The performance degradation associated with the single stepping of machine instructions approaches that of simulation.

Another approach to the implementation of data traps is through the use of instruction counters [Cargill87] [Mellor-Crummey89]. The basic idea is to stop the target process periodically by loading the instruction counter with a predetermined instruction step and then checking the "watched" location. If the value has changed since the last trap occurred then the data trap must have occurred in that time span. The target process is restarted, either from scratch or from a checkpoint, and executed to a point midway in the above region. By performing this procedure enough times the region under observation is reduced until the instruction causing the memory update is found. When using this method a compromise must be found between small instruction steps, giving frequent transfers of control but fewer restarts to find the update, and large instruction steps, giving fewer transfers of control but more restarts to find the update. The performance overhead with the above method can be considerable if the program is restarted many times. The use of checkpoints reduces restart execution time somewhat, but update information must be stored resulting in both memory and execution rate interference. A further disadvantage with the use of instruction counters is that only watchpoints can be detected and not data breakpoints.

The periodic transferral of control to the monitor process also enables the implementation of other tools. The Mesa Spy [McDaniel82] provides a performance analysis toolkit via a technique based on PC sampling. This method "grabs" control at regular intervals and extracts information from the process state, enabling the monitor to determine for what execution time routines are responsible.

Possibly the simplest hardware support for data breakpoints and watchpoints is the provision of a machine register which is checked on each memory reference; a match of register value and memory reference location causes an interrupt-like trap of the kind described previously. The AIDS monitoring system [Hart79] allows for a single watchpoint to be implemented in hardware using a reserved data update register. If the system is used on a machine which does not support a data update register or more than one watchpoint is required then interpretation of the target program is used. Machines which support a data update trap include SYMBOL and the IBM System/370 [Johnson82]. Performance degradation associated with a single watchpoint using a data update register would be negligible as the comparison could in most cases be performed in parallel with the memory access. However, it is apparent from the discussion in chapter 4 that more than one watchpoint will often be required in a high-level monitoring condition. For example, the monitoring of an array, a variable local to a recursive procedure, or an expression involving more than one variable will all require more than one watchpoint and consequently more than one data update register.

One solution to this problem is to increase the number of registers available to the monitoring system. However, there is usually a limited number of registers available in a machine and having dedicated data update registers may not be practical as "spare" registers are often allocated to program variables. To avoid conflict for machine registers an additional bank of registers in the form of an associative memory can be added to the system. Performance degradation is still minimal because all elements of the memory are compared in parallel. Thus each memory reference still has only a performance overhead of a single register comparison which as in the single register case may be performed in parallel with the memory access. However, only small true associative memories are available and these are expensive. Manufacturers adding a register array would thus be more likely to use it to improve performance than to implement monitoring facilities.

An inexpensive solution to the provision of a data update memory is to use a section of normal main memory. Performance degradation with this method is increased considerably. If watchpoints are stored in no particular order in the memory then a sequential search is required, needing, on average, a number of memory accesses equal to half the watchpoints currently set. Sorting the watchpoints in the memory will require fewer additional accesses in general but adds the overhead of implementing a search routine for the sorting method used. If the area of main memory is only reserved when the monitoring system is invoked then this is also a source of interference to the target program execution. A separate, possibly faster, block of memory would reduce the performance overhead and remove the memory interference problem.

Some machines have existing features which can be used to implement watchpoints and data breakpoints. Descriptor based machines [Bishop81] perform all memory references via descriptors. This alone encourages software reliability [Johnson82] as the descriptor contains attribute information concerning the entity it describes. In most cases the descriptor can be easily extended to indicate a data trap on the data item. Performance degradation is minimal as the descriptor is referenced whether monitoring is being performed or not. The overhead is simply a comparison on the data trap flag in the descriptor. However, this implementation is restricted to those machines which reference memory via descriptors.

VAX DEBUG [Digital86] implements data breakpoints and watchpoints using exception handlers combined with the memory protection mechanism [Beander83]. An exception handler is simply a routine which is executed whenever an exception is raised. A system of priority levels of exception handlers means that the monitoring exception handlers can always "grab" control when the need arises. Data breakpoints are implemented by invalidating the protection status of the page of memory in which the location resides. Any reference to this page will raise an exception and the monitoring software can take control and check for a breakpoint. Watchpoints are implemented in a similar way but the page is write-protected so only memory location updates cause an exception to be raised. The performance degradation associated with this implementation can be considerable. A single watchpoint causes the software monitor to be entered whenever a location on the same page as the watchpoint is referenced. With a common page size of 512 words a reference to any one of up to 512 variables will raise an exception and cause the software monitor to be invoked.

High-level language computer architectures are designed with a particular language or type of language as the target language of the machine, and it might be expected that such computers would be more effective at monitoring programs written in that target language. However, Ditzel and Patterson [Ditzel80] concludes that the machine organisation itself does not necessarily help in the implementation of high-level monitoring facilities and that "the goal should be to provide machines that allow the creation of efficient systems with excellent diagnostics".

All the methods of implementing data breakpoints and watchpoints described in this section are not entirely satisfactory for one reason or another. Either the expense involved is too great or the performance degradation is impractical, or the implementation is restricted in other ways. Many of the methods examined do not allow data breakpoints or do not differentiate between data breakpoints and watchpoints.

In the following sections we describe a method of implementing code breakpoints, data breakpoints and watchpoints in a virtual memory machine which is relatively inexpensive but incurs an acceptably low *continual* overhead.

5.3. Virtual to Physical Translation

5.3.1. Background

In the early days of computers it was the programmer's job to divide his program up into a number of small pieces, or **overlays** which would fit into the available memory. It was also the responsibility of the programmer to store each overlay in an appropriate place in secondary memory, and arrange for the loading of overlays from secondary memory into main memory and vice versa. Using this method programs could be written which were larger than the available memory in the computer.

In 1961 a group at Manchester University devised a method whereby the above process of breaking large programs into overlays and transporting them between main and secondary memory was performed automatically. This method is now called **virtual memory.** The idea behind virtual memory is to allow the programmer to program in terms of **virtual address space** which is independent of the size of the actual memory in the machine but does depend on the size of the address field of the machine. Thus a machine with a 16 bit address field can reference 2^{16} or 65536 words no matter what the size of the actual machine memory. The illusion of a large memory exists because the virtual memory technique always loads the correct chunk of program from secondary memory into main memory when a memory reference occurs.

One common technique for the implementation of virtual memory is called **paging.** Equal sized chunks of program, called **pages,** are read in from secondary memory and placed in similar sized pieces of main memory called **page frames.** Pages are chunks of virtual address space and page frames or blocks are chunks of physical address space or main memory. The mapping of virtual address space onto physical address space is performed by means of the **memory map.** In the paging technique this mapping is performed by means of a **page table.** The page table for a given program has as many entries as there are pages in the virtual address space. Common page sizes are 512, 1024 or even 4096 words, but always a power of two. This means that the size of the page table is equal to the virtual address space size divided by the size of each individual page, and is thus, itself, a power of two in size. From this an example 16 bit machine with pages of 1024 words has a virtual address space of 65536 words and 64 page table entries.

The reason for all sizes being restricted to a power of two is for ease of translation, which becomes apparent when virtual addresses are examined. Using the above example machine it can be seen that the top six bits of a virtual address indicate the page number and the bottom ten bits the word offset within that page. When translating virtual addresses to physical addresses the page number is calculated simply by taking the top six bits of the address and this value is then used to index the page table to find the corresponding page frame value or block number in actual machine memory. The bottom ten bits of the address or word offset are carried across unchanged to form the within block offset.

The page table used in the translation process must contain at least three fields: a flag to indicate whether the page is currently loaded into a physical address space page frame, the page frame number if the flag in the previous field indicates that the page is loaded, and a secondary memory address which gives the location of the page in secondary memory. The size of the page frame field depends on the amount of actual memory in the machine. For example, a machine with 16K of memory split into page frames of 1024 words would have sixteen page frames and a page frame field width of four bits. A virtual to physical address translation for this example machine is shown in figure 5.1.

Multitasking machines use a set of page tables, one for each process residing on the machine. In practice page table entries of real machines have some sort of protection status field, which may indicate, for example, page is read-only, page is out of bounds, or page is not loaded into memory. Other information held within the page table entries is used to speed up housekeeping duties. An example of this is the **modified** flag. This is set if the page loaded into main memory is updated during program execution. When the page frame is overwritten by another page being loaded into memory it need only be written out to secondary memory if the modified flag

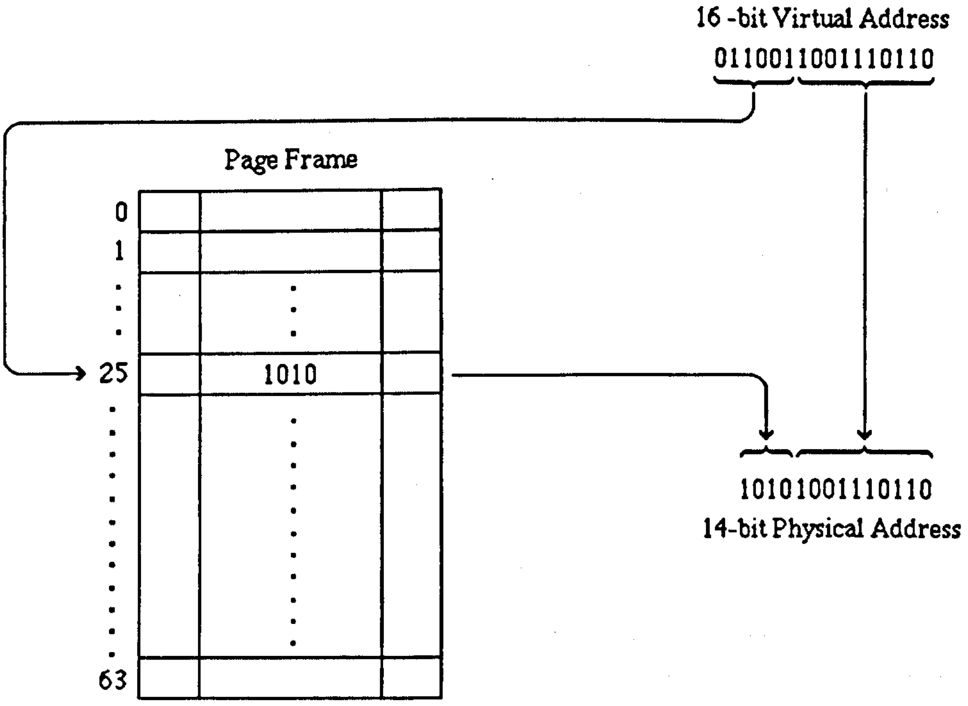

Figure 5.1 Virtual to Physical Translation

is set.

To achieve a fast translation the page tables can be implemented as a fast associative register array. However, large register arrays are expensive and therefore impractical, so page tables are usually held as software structures. If the translation were performed entirely in software the performance degradation would make the system intolerably slow and so modestly sized associative register arrays and fast caches are often employed to hold the most recently used entries of the page tables. Using a fast access memory means that the translation can be overlapped with other CPU activity to make the translation time acceptable.

The use of a caching technique or **translation buffer** introduces a two-level page fault mechanism. A page fault interrupt occurs if the page table entry protection is in some way violated. Operating system intervention is required in this case to either load the page from secondary memory, acquire more memory for the process, or signal an illegal memory reference. A lower level page fault can also arise. This occurs if the page table entry is not cached in the translation buffer. The caching of the page table entry is performed at the architectural level either by hardware or firmware. For a more detailed study of virtual memory techniques refer to [Watson70] [Lorin81] [Tanenbaum84] [Maekawa87].

5.3.2. Implementation of breakpoints in a virtual memory architecture

5.3.2.1. Data breakpoints and watchpoints

In this section a method for providing architectural support for data breakpoints and watchpoints in a virtual memory machine is described. This takes the form of additional hardware in the virtual to physical translation unit which causes an interrupt-like trap, similar to that caused by a code breakpoint, which results in a transfer of control to the monitor process.

A method of this nature is used by Abramson and Rosenberg [Abramson83] for the MONADS II computer. The address translation unit of MONADS II is rather unusual, consisting of a hash table held in very fast addressable memory, a hashing unit and a comparator. Each virtual address to be translated is used by the hashing unit to find a chain of cells in the hash table. The hardware searches this chain of cells until either the page number is found or the end of the chain is reached. The added architectural support consists of an extra field of the hash table and a fast breakpoint memory. The extra field of the hash table is used to index the breakpoint memory which holds lists of within-page displacements.

Each virtual address translation consults the extra link field of the hash table. If this field is found to be empty then the translation proceeds normally, otherwise the memory reference is suspended and the breakpoint memory accessed at the location indicated by the link field. The chain of within-page displacements is now searched until a match is found or the end of the chain is reached. If a match is found then an interrupt occurs. Fields also exist within the breakpoint memory to implement breakpoints on ranges of locations, breakpoints associated with particular processes and also to indicate whether the breakpoint is a watchpoint or a data breakpoint.

The address translation mechanism of the MONADS II computer is unusual in that the hardware translator is not a cache for the most recently used addresses but holds all the translation information for main memory resident pages. As described in the previous section the more commonly found mechanism is that of an associative store or fast memory which caches virtual to physical translations with appropriate protection bits. In this section we describe support for the data breakpoint and watchpoint which will be more widely applicable than the more specific implementation described for MONADS II.

The architectural support we describe here consists of an extra flag within the protection status of the page table entry, and a breakpoint memory. The additional protection status bit, termed the **monitor** bit, causes a memory access fault to be raised in much the same way as for an invalid or illegal reference. The low-level page fault mechanism tests the protection bits in the page table entry to either cache the page table entries from the software structures or cause an interrupt to the operating system for software intervention. Additional architectural support is inserted into the low-level caching mechanism to test the monitor bit. If the cause of the access fault is not the monitor protection bit then the access fault mechanism proceeds as normal. However, if the monitor bit is the reason for the access fault then the additional breakpoint memory is accessed.

The breakpoint memory should preferably be a block of fast memory but could equally be implemented in main memory with a slight performance loss. Access to the breakpoint memory is made using the word-in-page offset of a virtual address, thus requiring the memory to have as many entries as there are words in a page. However, to accommodate breakpoints on the same word-in-page offset but different page numbers extra entries are required to store any conflicts for entries.

Each entry of the breakpoint memory consists of five fields: an entry enabled flag, a page number, a data breakpoint flag, a watchpoint flag, and an overflow link field (figure 5.2).

The **entry enabled** field indicates whether the particular breakpoint memory entry contains valid information. The **page number** field holds the page number of the data breakpoint or watchpoint, and is checked when a trap

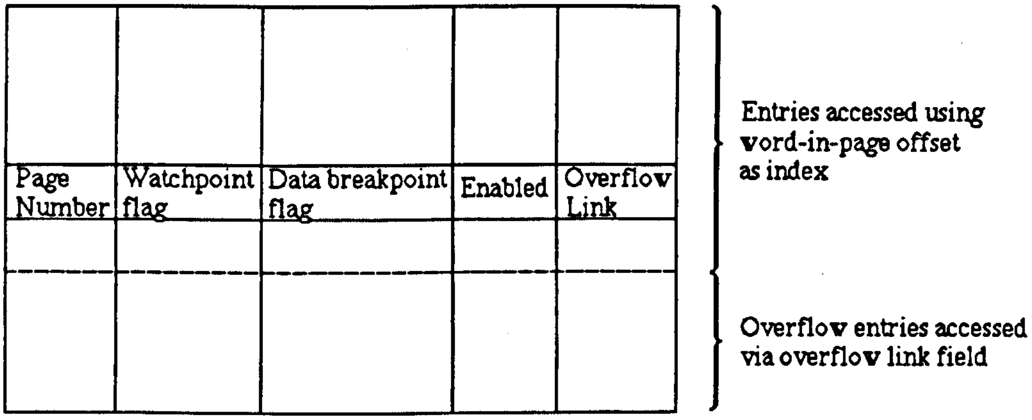
Page Number
Watchpoint flag
Data breakpoint flag
Enabled
Overflow Link
Entries accessed using word-in-page offset as index
Overflow entries accessed via overflow link field

Figure 5.2 Breakpoint Memory

occurs to ensure that the entry is the correct one for that particular breakpoint. All entries can be uniquely identified in this way as the page number field, together with the word-in-page index, reconstructs the original virtual address. The **data breakpoint flag** indicates that a memory access to the particular location is to be trapped and the **watchpoint flag** indicates that a memory update is to be trapped. The **overflow link** field links together all breakpoints which are currently set on any one particular word-in-page but on different pages.

The setting of a data breakpoint or watchpoint is very simple. The word-in-page offset of the breakpoint address is used as an index into the breakpoint memory. In most cases the indexed entry of the breakpoint memory will not contain a previously set breakpoint and will thus have the enabled flag not set. If this is the case then the page number of the breakpoint is stored in the page number field and the appropriate flag indicating a data breakpoint or watchpoint is set. The enabled flag for this entry can now be set activating the breakpoint. Figure 5.3 shows the setting of a data breakpoint on location 6676H in an imaginary 16 bit machine.

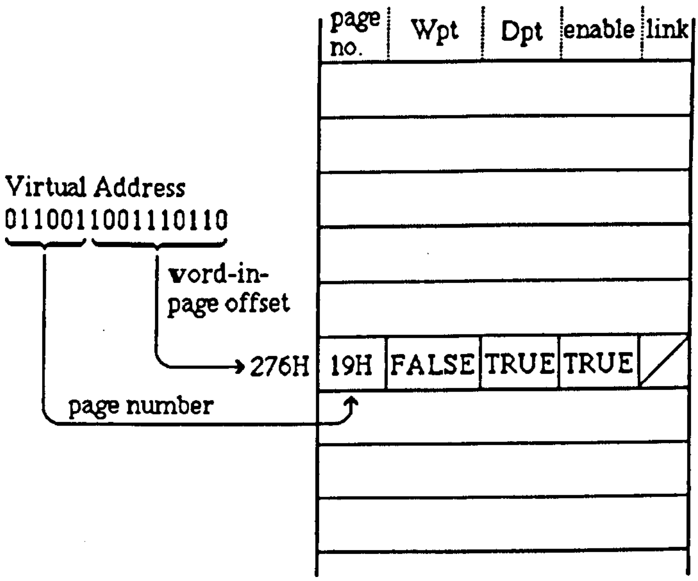

Figure 5.3 The setting of a breakpoint

The situation may arise, however, that the entry indexed by the word-in-page offset is already in use, leading to a conflict for the particular entry. It may be the case that a number of breakpoints have been set at this word-in-page offset and so the generalised process for the setting of a breakpoint involves the indexing of the breakpoint memory and the subsequent following of the overflow link until either an entry is found with the enabled flag not set or a null overflow link. Entries with the enabled flag not set occur in the chain of entries when a breakpoint has been removed. This method of deleting breakpoints simply by removing the enabled flag allows time consuming housekeeping duties such as reorganisation of the overflow chain to be performed at non-critical times.

When an entry is found with the enabled flag not set then the sequence of events for setting a breakpoint is performed on this empty breakpoint memory slot. If, however, the end of the chain is reached and no entries have been found which are not in use then a new entry must be linked to the end of the chain to hold the relevant information. Figure 5.4 shows the state of the breakpoint memory before a watchpoint is set and figure 5.5 the new state of the memory.

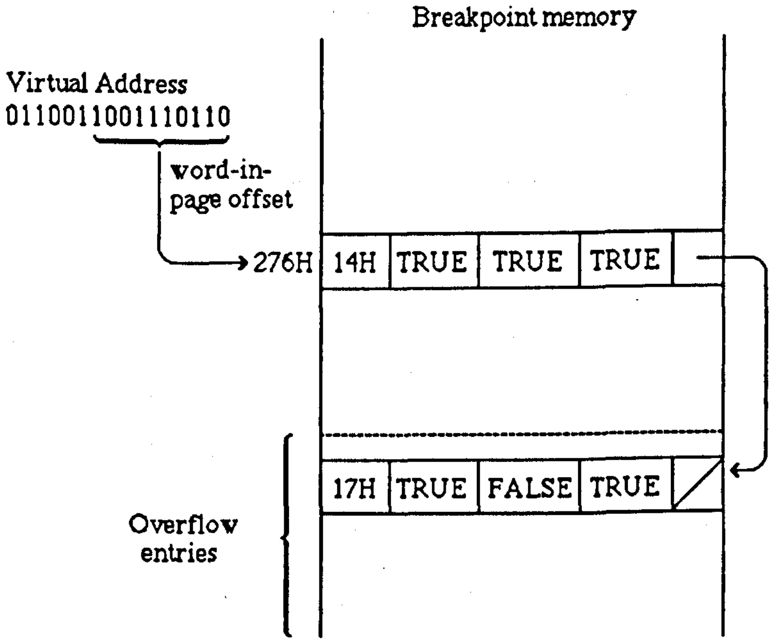

Figure 5.4 Example state of breakpoint memory

In addition to the updating of the breakpoint memory the monitor bit of the page table entry for the page on which the trap is to be set must be set to force the access fault when a reference occurs.

References to locations in pages with no data breakpoints or watchpoints will

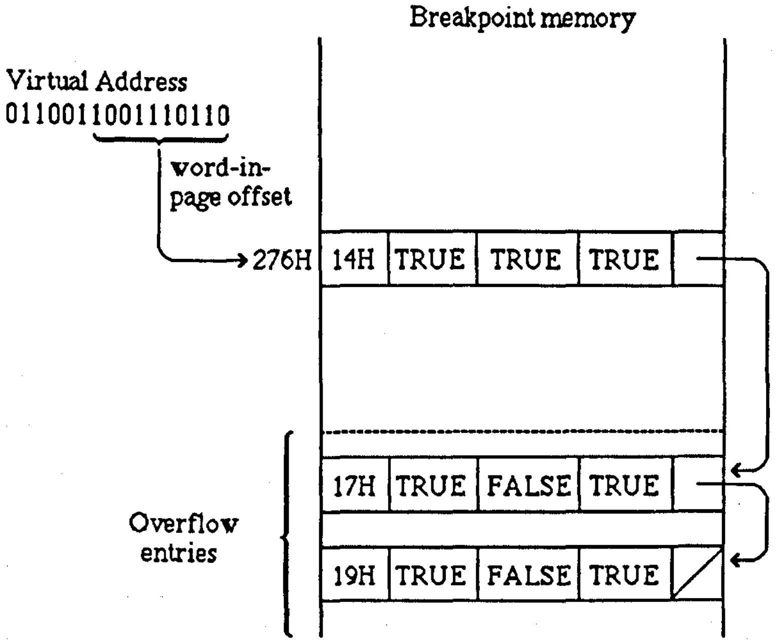

Figure 5.5 Setting of a watchpoint with contention

thus proceed at a normal rate. However, references to locations in pages *with* traps will be slowed in proportion to the number of traps with the same word-in-page displacement. The breakpoint memory is accessed using the word-in-page offset as an index and the overflow link followed as necessary until a match is found between the page number field and the page number of the referenced location. When a match is found the data breakpoint and watchpoint flags are checked against the mode of access and an interrupt-like trap forced if appropriate. The trap is required, however, at the end of the executing instruction as a context switch to the monitor process would not be allowed by most machines midway through an instruction.

In most cases only one breakpoint will be found per entry of the breakpoint memory thus contributing only a slight overhead, this being reduced still further if the first breakpoint memory access is performed in parallel with the translation buffer look-up. The architectural support described above is shown in figure 5.6.

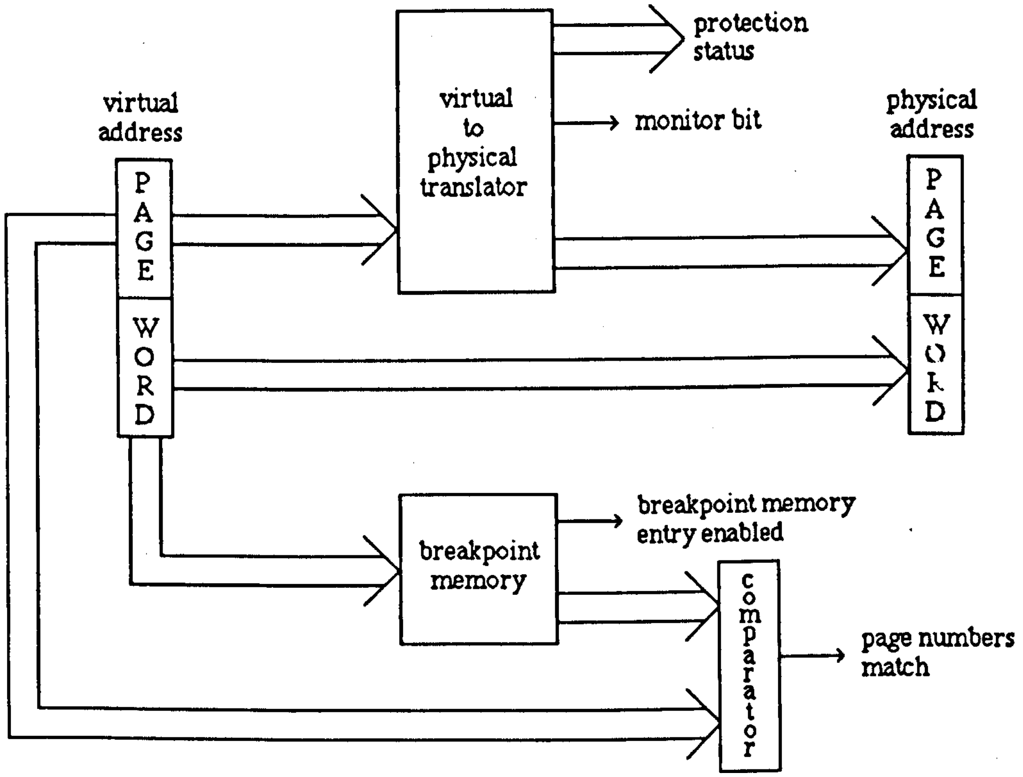

Figure 5.6 Architectural support for breakpoints

5.3.2.2. Code breakpoint

As previously discussed, the simplest and most effective implementation of a code breakpoint is achieved by the instruction at the appropriate location being replaced by an instruction which causes an interrupt-like trap to the operating system. This section is concerned with reducing the execution overhead associated with this form of code breakpoint. Due to the method of implementation there is no continual overhead on program execution; unaltered regions of the instruction space execute at normal rates and only those locations replaced by trap instructions cause a performance degradation. Aside from software evaluation of monitoring predicates there is a performance degradation more directly associated with code breakpoints. This is the cost in execution speed of the time it takes to either emulate the replaced machine instruction in software, or else put back the original instruction, single step it and then replace it again by the trap instruction. Support to reduce the time involved with either of the above two procedures is not to be found in the literature. This is possibly because the interference of the monitoring software overshadows the degradation caused by the implementation of the breakpoint. However, a high-level monitor such as that described in chapter 4 may return control to the target process almost immediately. This would occur, for example, if the code breakpoint caused a trap at an undesired nesting level. In this case the performance degradation associated with executing the original instruction may contribute considerably to the overall degradation and so support is required.

The architectural support described in this section consists of a breakpoint memory similar to that used for data breakpoints and watchpoints, and a special machine instruction to be used as the trap instruction. Each entry of the breakpoint memory consists of four fields: an entry enabled flag, a page number, a machine instruction, and an overflow link field (figure 5.7).

The **entry enabled** field, the **page number** field, and the **overflow link** field perform the same functions as in the previous section, for data breakpoints

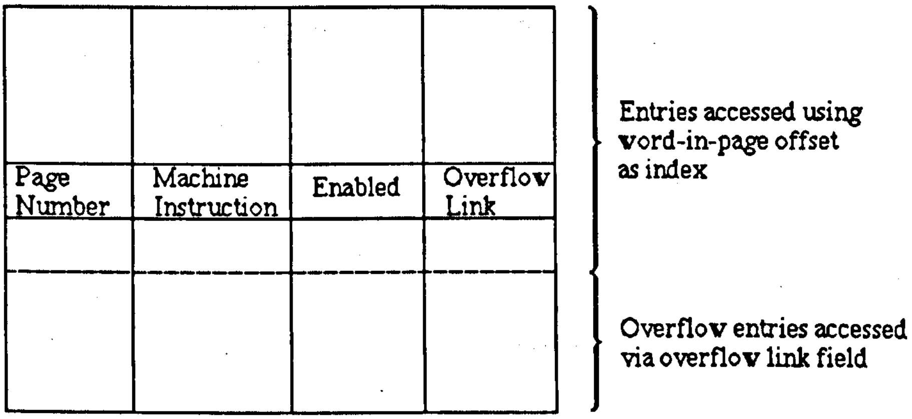
Page Number
Machine Instruction
Enabled
Overflow Link
Entries accessed using word-in-page offset as index
Overflow entries accessed via overflow link field

Figure 5.7 Breakpoint Memory

and watchpoints. The **machine instruction** field, however, is unique to the code breakpoint case, and holds the instruction which is replaced by the trap instruction. This is the instruction which, in conventional monitoring systems, is emulated or temporarily reinstated at the location of the code breakpoint.

The setting of a code breakpoint occurs in the same way as for data breakpoints and watchpoints except that, instead of setting flags to indicate the type of reference, the machine instruction at the breakpoint location is accessed and stored in the appropriate field of the breakpoint memory.

The breakpoint instruction is a special instruction which performs a breakpoint memory access prior to causing an interrupt-like trap to the operating system. The access of the breakpoint memory is performed using the word-in-page offset and involves following the overflow link field, in much the same way as when setting breakpoints, until an enabled entry is found with an appropriate page number field, that is, one which matches the page number of the currently executing location. On finding the entry which matches this breakpoint the original instruction for that location can be extracted from the appropriate field. Rather than reinstating the instruction in the target program instruction space the instruction can be loaded directly into the processor instruction register for execution. Thus, when the monitoring process is ready to transfer control back to the target process it need not perform any instruction emulation or "juggling".

In most cases only a single breakpoint memory access will be required to find the instruction replaced by the code breakpoint trap instruction. Also contributing to the speed of the breakpoint memory access is the method of access. Extracting the word-in-page offset from the virtual address is a quicker generator of an index than the use of a hash unit, for example.

5.3.2.3. Firmware monitoring

There is scope in the methods described in this chapter for a firmware implementation. Thus, it may be possible to implement the scheme without adding hardware to the machine, but by simply changing the firmware of the machine. A requirement of this process is a writeable control store which has enough extra RAM for the additional microcode.

Firmware monitoring systems have been described in the literature over the past twenty years but these have been orientated to low-level monitoring.

Tracing of instructions [Barnes74] [DeBlasi77] [Agarwal86], opcode counts [Saal72] and sampling systems [Armbruster79] all create large machine-level records. The advantages of using a firmware monitor include the flexibility, small expense and high speed associated with microprogramming. There are, however, problems associated with microprogramming: the useage of control store is often complicated [Grätsch81], it may not be possible to modify all sections of the standard instruction set due to memory limitations or timing restrictions [Agarwal86], many firmware resources are often global across the machine resulting in only one user being able to operate the system at any one time [Melvin86], and tools for the development of microprograms are often at the assembly language level, making the effort involved in developing and debugging microcode quite high [Grätsch81] [Melvin86].

A microcoded implementation of the above proposals has been performed on the HLH Orion, the procedure and results of which are described in the next chapter.

5.4. Summary

The monitoring of real-time software must not introduce unbounded delays and consequently it relies on expensive circuitry and monitoring processors to incur only bounded delays. Software which does not necessarily have to adhere to real-time rules must also be monitored with as small a performance overhead as possible. However, the use of expensive electronics to incur only bounded delays is not feasible and so less expensive methods must be sought whilst keeping the performance overheads to a minimum. To this end an economical implementation of the three monitoring primitives is sought.

The virtual to physical translation method proposed in this chapter attempts to do this. It has many advantages over other implementations, such as those involving the use of register arrays, bitmaps and descriptors. A major advantage is that only memory references to pages which contain breakpoints are delayed. It is only these pages which cause the low-level access fault mechanism to be invoked other than normally occurring access faults such as uncached entries or page faults. References to pages without breakpoints proceed as normal. Other methods which perform a memory look-up or some sort of check when a memory reference occurs do so for every memory reference.

It is unlikely that many breakpoints will be set with the same word-in-page displacement and so references to pages which do contain breakpoints will only be delayed for a short time. Thus very little time is wasted traversing the list of breakpoints if a breakpoint is not found. It may also be possible to perform the first index into the breakpoint memory and the original memory reference in parallel thus incurring a negligible delay if only a single breakpoint is located on the particular word-in-page displacement. This is in contrast to the implementation proposed by Abramson and Rosenberg [Abramson83] where the translation cache must be accessed to obtain the breakpoint memory index, and the two accesses cannot be overlapped.

In contrast to other look-up methods, such as a breakpoint bitmap implementation, the index into the breakpoint memory is not calculated specially for monitoring purposes. The index used for the access of the breakpoint memory is the word-in-page displacement which is calculated by the virtual to physical translation mechanism in a normal virtual to physical translation.

Implementations such as the descriptor-based method can restrict the class of objects which can be monitored. The method proposed in this chapter allows any virtual address to be monitored including the instruction space, data stack and heap.

Finally, an inexpensive implementation of the proposed mechanism can be applied to most virtual memory machines. The cost of the implementation is further reduced if a microcoded version is possible.

There are, however, some disadvantages and problems to be found with the proposed implementation of the monitoring primitives. Firstly, unlike [Abramson83] breakpoint ranges cannot be supported and so breakpoints must be set on each element of arrays or structures for example. Thus the monitoring of large arrays or structures leads to the first indexed elements of the breakpoint memory being used and consequently gives rise to many more conflicts and overflow entries. However, it would still be the case that only those pages which contain breakpoints would be delayed and so the problem is not considered a serious one.

Perhaps a more serious problem is associated with the inclusion of performance enhancing features on many machines. These take the form of fast memory caches for accessing memory. By placing the most recently used variable in a fast cache the machine does not need to access slower main memory as often. If the caching occurs on physical addresses then no problem arises but if it is virtual addresses which are cached then the virtual to physical translation may not take place. An extreme case of this occurs in RISC machines where large register arrays are used to hold program variables. The problem arising from the use of a data cache can be overcome if the caching mechanism prevents monitored locations from being cached. However, the problem is more serious in a RISC machine as performance may suffer considerably if the reference of program variables in the register array is performed in main memory. To overcome this the machine must feature the ability to trap on access or update of a register in the register array. The transfer of program variables between main memory and the register array must be "watched" so that appropriate registers can be monitored when necessary.

The practicality and performance of the monitoring methods outlined in this chapter are further examined in an experimental system described in the following chapter.

6. Implementation and Analysis

6.1. Monitoring Software

6.1.1. Requirements

There are three basic requirements of the monitoring software apart from the implementation of the software structures and graph traversal algorithms described in chapter 4. Firstly, two processes are required: the monitor process and the target process. The target process should be created exactly as it would be if no monitoring were specified. Thus it should require no more resources than any other process on the machine or cause execution of the target program to differ from a normal execution. Secondly, the monitoring software must provide the monitor process with full control over the target process. This consists of starting the target process when the monitoring software requires it and suspension of the target process through the use of monitoring primitives. Finally, the monitoring software must invoke two way communication between the monitor process and the target process. This involves communication of variable values and trap locations.

An experimental system of this form has been implemented to validate and study the performance of the monitoring structures and primitives introduced in chapters 4 and 5. The machine used to implement the experimental system is the High Level Hardware Orion running 4.2BSD UNIX. In order to make full use of the facilities offered by the UNIX operating system the implementation language used is the C programming language [Kernighan78].

Input from the user is parsed using a set of YACC grammar rules [Johnson78a] with actions in C which perform appropriate functions or build the necessary directed graph for the monitoring software. Selected procedures from the implementation can be found in appendix A.

6.1.2. Processes and Control

The creation of the target process and control over it by the monitor process is achieved by UNIX system calls from the monitor process, resulting in a parent-child relationship between the two processes. The four system calls used to invoke the two processes with the appropriate control and synchronisation are the fork, exec, ptrace and wait system calls.

The *fork* system call [Fork(3)] invokes an exact copy of the calling process resulting in two processes executing the same code at the same time. The replacement of one process by another is performed by the *exec* family of system calls [Exec(3)]. These two calls allow the monitoring software to firstly replicate itself and then overlay one copy with a process of the executing target program. However, the execution of two processes in parallel is not sufficient to meet the requirements of the monitoring system. The monitor process must have control over the target process, halting it via the use of monitoring primitives and restarting it when appropriate. A degree of synchronisation is also required, such that only one of the two processes is executing at any one time. Thus, the suspension of the target process causes the monitor to restart and vice versa.

Control and synchronisation of the two processes is achieved through the use of UNIX signals [Signal(3)] and the two system calls: *ptrace* and *wait.* The ptrace system call [Ptrace(2)] allows the monitor to start the target process at will, whilst the wait system call [Wait(2)] causes suspension until an appropriate signal is generated.

The code fragment in figure 6.1 constitutes the basis for synchronised monitoring with the required control.

```
if( (pid = fork()) == 0 ) {
   /* executed by target process */
   ptrace( PT_SETTRC,0,0,0) ;
   exec( filename ) ;
} else {
   /* executed by monitor process */
   do {
      wait( &status ) ;
      stopped() ;
   } while( ptrace( PT_CONTIN,pid,(int*)1,0 ) != -1 ) ;
}
```

Figure 6.1 Skeleton routine for synchronised monitoring

The first line of the code segment invokes a copy of the current process. At this point both processes are executing in parallel and perform the test in the conditional. The test against zero is true in the case of the child or target process and false in the case of the parent or monitor process. It is this test which allows the two processes to be separated. The code between the first pair of braces is executed by the child process, while the parent process executes the code between the second pair. The variable *pid* acquires the child process identification number for use in later system calls.

The child process executes the ptrace system call which informs the operating system of the parent-child link. The second statement, the exec system call, replaces the current process with the execution of the target program. At the same time as this the parent process is executing the *do* loop in the second part of the conditional. The *wait* statement suspends this monitor process until the target process suspends itself and generates a signal. Examples of such a signal include the execution of an illegal instruction, an interrupt generated at the keyboard or a memory fault. Because of the parent-child link made by the child process earlier the target process does not terminate but merely suspends itself. Thus when a signal is generated the monitor process resumes execution and enters the procedure *stopped.*

This procedure contains all the necessary monitoring software to perform the high-level monitoring described in chapter 4. The *status* variable which acquires the signal identifier by the wait system call can be tested to determine the reason for the target process suspension. The monitor process transfers control back to the target process when ready by returning from the procedure *stopped.* The ptrace system call in the *while* statement resumes execution of the child at the point where it halted and the monitor process executes the loop to wait once again for a generated signal.

Using this system only one process is active at any one time, thus satisfying the synchronisation condition. Further, the monitor process is able to perform any number of monitoring functions before it passes control back to the target process, thus satisfying the conditions of control.

6.1.3. Communication

Communication between the two processes must be two way. The monitor process must be able to communicate locations and types of monitoring primitive to the target process to enable the setting of traps within the executing program. Similarly the target process must be able to communicate locations and types of monitoring primitive back to the monitor process in order that it can determine the cause of suspension. As well as the need to communicate trap information the monitoring process must be able to update target process memory and also receive target process memory values. This implements, at the user level, the setting and examination of target program variables.

The reading and writing of target process memory is implemented using the ptrace system call, which also allows the reading and writing of target process registers. The communication of monitoring primitives also uses the ptrace system call. By assigning an area of memory for the purposes of communication between the two processes we can realise memory mapped monitoring primitives. Effectively, the writing to one area of memory sets monitoring primitives and the reading of another area allows the identification of traps which have occurred since the last transfer of control, which will be the traps which caused the current transfer of control. In the cases where a special, fast memory is used for monitoring purposes it is necessary for the monitoring process to be able to read and write to it. This will in most cases require the implementation of customised instructions.

6.1.4. Monitoring Functions

The user of the monitoring system operates it via an interface constructed from YACC grammar rules. On the experimental implementation this is restricted to the setting and displaying of program variables, the tracing of procedure calls during execution, the single stepping of source statements, and the building of directed graphs according to specified WHEN commands.

All program objects, such as variables and procedures, are specified using the source code symbols with the extra option to use line numbers for the setting of code breakpoints. The facility of using symbols to refer to program objects is provided via the program symbol table. This is stored as part of the executable file when the *debug* option is given at compile time. It is stored in the file as symbol-type-address triples enabling the monitor to print information in the correct format and also allowing monitoring functions which perform operations on groups of objects of the same type. For example, accessing all procedure names allows the monitor to trace procedure calls during execution, or accessing all parameter variables allows the monitor to print the values passed to a particular procedure. The relevant information is extracted from the file and stored internally to the monitor process.

6.2. Architectural Support

6.2.1. Requirements

In chapter 5 it was stated that a feasible implementation of the required architectural support for monitoring is possible using microcode. This section describes such an implementation for the HLH Orion, a user microprogrammable machine.

The Orion CPU consists of the following main components [HLH84]:

- Control Store
- Microprogram Sequencer
- Map Tables
- Arithmetic and Logic Unit (ALU)
- Cache Memory
- Virtual Memory Translation Buffer

All but the ALU, eight AMD AM2901C 4-bit bitslice microprocessors linked in parallel, have some bearing on the implementation.

To enable a realistic implementation of the required support the control store (that is, the high speed memory in which the microprogram resides) must be large enough to hold the standard microcode together with any additional code required for monitoring purposes. Many microprogrammable machines have only a limited free space in the control store thus requiring very careful coding [Agarwal86]. However, the control store in the Orion CPU, which was designed for microcode development, consists of 32K 64-bit words of RAM divided into 4K word pages. This is easily sufficient for the implementation of the support described in chapter 5.

There are four main areas where the microcode is altered or added to:

(i) A code breakpoint instruction is required, which causes an interrupt-like trap to the operating system, after obtaining the original instruction from the breakpoint memory. This original instruction is placed in the instruction register prior to the transfer of control so that it is the first instruction to be executed on a return. A further action of the code breakpoint instruction is the placing of the location and type of primitive in the reserved area of memory for communication to the monitor process (section 6.1.3).

(ii) Because the top of scalar stack is cached in a fast memory, separate from main memory, it is necessary to alter the instructions which access this cache so that they access the appropriate area of main memory as well, thus making all variable accesses use the virtual to physical translation mechanism.

(iii) The use of an area of memory reserved for communication with the monitoring system was mentioned above, and will be discussed further in a later section.

The area of memory is reserved during the process start-up microcode and is not visible to the executing process. The reserved memory is allocated from the bottom of the vector stack, chosen because of the ease of implementation.

The final action of the revised start-up microcode causes an interrupt-like trap back to the operating system so that the monitor process can take control and prompt the user prior to target process execution.

(iv) To enable the monitoring of memory references the code which handles access faults during references is altered. Firstly, the extra protection bit, or *monitor* bit (section 5.3.2.2), is implemented which causes the necessary access faults whenever monitored pages are accessed. The extra code to access the breakpoint memory can now be added at this point in the access fault code to determine if a primitive exists on the referenced location. If this is the case then the location and type of trap is stored in the reserved area of memory. Once this has been performed an interrupt-like trap to the operating system is required and is achieved by loading an illegal instruction opcode into the instruction register. When this is decoded the interrupt mechanism saves the context of the executing process and restores the kernel context with the appropriate trap code. Because of the parent-child link made by the monitoring software the operating system sends the illegal instruction signal generated to the monitor process which performs a return from the wait system call it is currently executing.

6.2.2. Microprogram Sequencer

The microprogram sequencer controls the order in which microinstructions are fetched from the control store and executed. Most of the sequencer functions perform some kind of a control transfer such as *jump* or *jump to subroutine*. One of the sequencer functions performs the decoding of machine instruction opcodes which introduces a problem for the implementor on the HLH Orion and similar machines. The implementation requires that an interrupt-like trap to the operating system occurs when a code breakpoint, data breakpoint or watchpoint is encountered and, whilst this is easily achieved for code breakpoints through the use of a special instruction, data breakpoints and watchpoints must cause a transfer of control to the operating system after the execution of the machine instruction performing the memory reference. However, in most cases the offending memory reference will occur in the middle of a machine instruction. To overcome this problem the microcode which "watches" for references to monitored memory locations sets a flag, the *data trap* flag, which indicates that a monitoring primitive has been observed. This flag can then be tested after the execution of each instruction emulation routine.

The use of a sequencer function to decode opcodes along with instruction caching means that there is no single routine which performs the usual fetch-decode-execute phase for each instruction. Instead, up to four instructions may be executed before the microcode to perform another fetch is entered. The sequencer function in question takes the value in the instruction register and performs a look-up in a fast internal memory which holds machine instruction opcodes and their respective emulation routine addresses in the control store. The reloading of the instruction register from the instruction cache is performed by a line of microcode which is found in every emulation routine.

In most cases there will be two solutions to the above problem. The first involves changing the microcode such that all machine instruction opcode decoding performs a check on the data trap flag. The HLH Orion has sufficient control store to facilitate the altering of the required code but the task is still non-trivial due to the, often, complicated structure of microcode routines. Implementations on machines without the free control store space would have to compromise and insert the code to check the data trap flag in the most frequently executed piece of code. In most cases this would be the routine which performs the instruction caching. The granularity of the trap mechanism is thus determined by the number of instructions cached.

A multiple instruction bank feature of the HLH Orion is utilised in the experimental implementation to realise a routine, which is executed after the execution of each emulation routine, and requires only minimal changes to the existing microcode but does not incur a heavy execution overhead. This mechanism is described in the next section.

6.2.3. Map Tables

The map tables provide a mechanism by which abstract machine opcodes can be quickly decoded. An instruction register is loaded with the opcode to be decoded, after which a specific sequencer function causes this value to be used as an index into the map tables. The value found in this fast memory is the address of the first microinstruction in the machine instruction emulation routine. This mechanism is shown in figure 6.2.

An instruction set occupies a pair of map tables, each allowing the decoding of 256 opcodes. Current implementations of the HLH Orion have four such pairs of map tables, allowing entirely independent instruction sets to reside in the machine simultaneously.

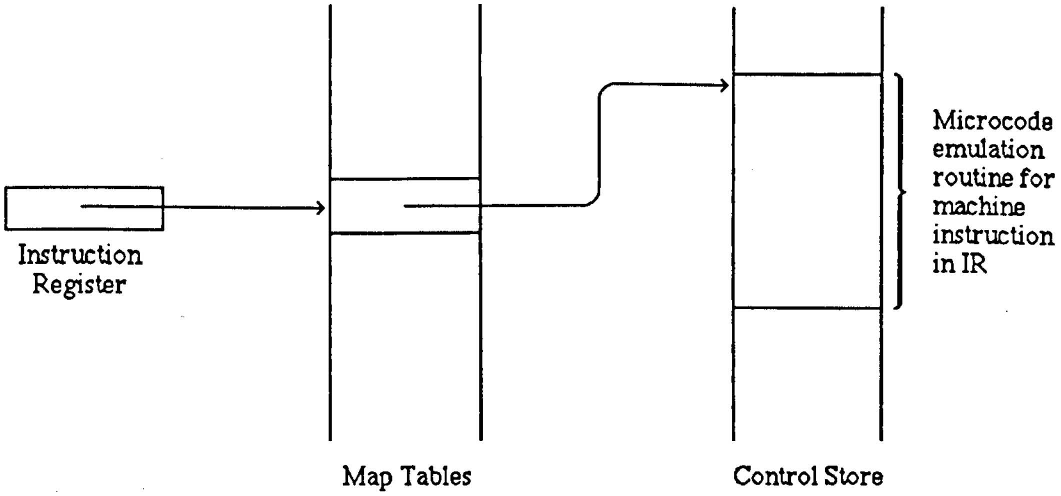

Figure 6.2 Decoding of opcodes via map tables

The multiple instruction bank feature enables the implementation of a microcode routine which is executed after every instruction emulation routine. This in turn enables the implementation of the data trap flag and the trap back to the operating system. One pair of map tables is initialised to the same control store address, to which all opcodes will decode. The routine stored at this particular control store address performs the check on the data trap flag and also decodes the original opcode in the correct instruction bank (figure 6.3).

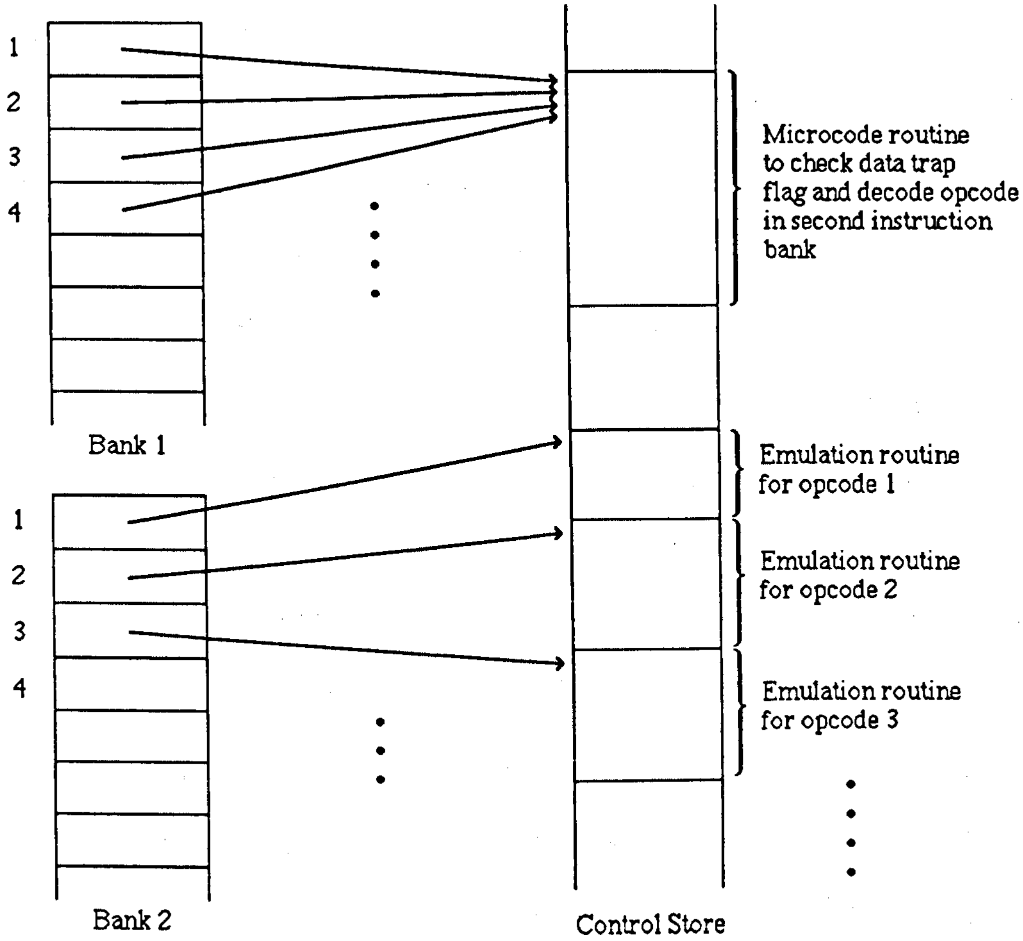
1
2
3
4
Bank 1
1
2
3
4
Bank 2
Control Store
Microcode routine to check data trap flag and decode opcode in second instruction bank
Emulation routine for opcode 1
Emulation routine for opcode 2
Emulation routine for opcode 3

Figure 6.3 Use of multiple instruction banks to implement data trap flag

6.2.4. Cache Memory

The cache memory on the HLH Orion provides a large bank of fast registers internal to the CPU. However, at the microcode level the cache is simply a randomly addressable memory, separate from the main system memory. It presents itself as an obvious implementation for the breakpoint memory introduced in chapter 5.

The cache memory is divided into pairs of 512 word sections with the current implementation of the machine having two such pairs. The standard instruction set makes use of one cache bank, leaving the second at the disposal of the microprogrammer.

The use of a cache memory as a fast access memory for the top of stack introduces a problem for the implementor, as discussed in section 5.4. By using a cache the usual virtual to physical translation required to access program variables is not performed, thus bypassing any microcode which is added to "watch" for accesses to monitored locations. Fortunately the caching mechanism on the HLH Orion is performed entirely in microcode and so the problem can be resolved by the microprogrammer.

There are two solutions to the above. Firstly, all the microcode which accesses the fast cache memory could be removed and replaced with the corresponding code which accesses main memory. This would involve the rewriting of large amounts of microcode. A simpler method, and the one used in the final experimental implementation, is to continue to use the top of stack cache, thus leaving the original microcode intact, but to add code which performs the corresponding virtual to physical translation. The actual main memory reference is not required for the generation of the necessary access faults and because the translation can, in most cases, be performed in one cycle the performance overhead is not excessive.

The second cache bank is used to contain the breakpoint memory. However, the cache is not large enough to hold any overflow entries or entries for code breakpoints and so these must be relegated to storage in the main system memory. The provision of a much larger fast memory would improve the implementation of the breakpoint memory by allowing overflow entries and code breakpoint entries to be stored in the fast memory. Because, for code breakpoints, replaced instructions are stored in the breakpoint memory the 32 bit entries in the cache memory are not, in this case, large enough to hold all of the associated fields. The whole of the breakpoint memory for code breakpoints is therefore implemented in main memory. This situation is not as detrimental to the performance as it appears to be. Each breakpoint memory entry requires only two words of main memory which can be accessed with a routine only one cycle longer than for a single word access. It was also found that with the less cramped breakpoint memory structure, fields could be extracted with less code than for the corresponding fast memory entries.

6.2.5. Virtual Memory Translation Buffer

The primary component in the HLH Orion memory management hardware is a fast memory, internal to the CPU, known as the *translation buffer*. In the standard system the translation buffer is treated purely as a cache for the memory-based page tables. The selection of an appropriate microoperation uses the virtual address in the virtual address register as an index into the translation buffer, resulting in the output of the corresponding physical address and six bits of protection information (figure 6.4).

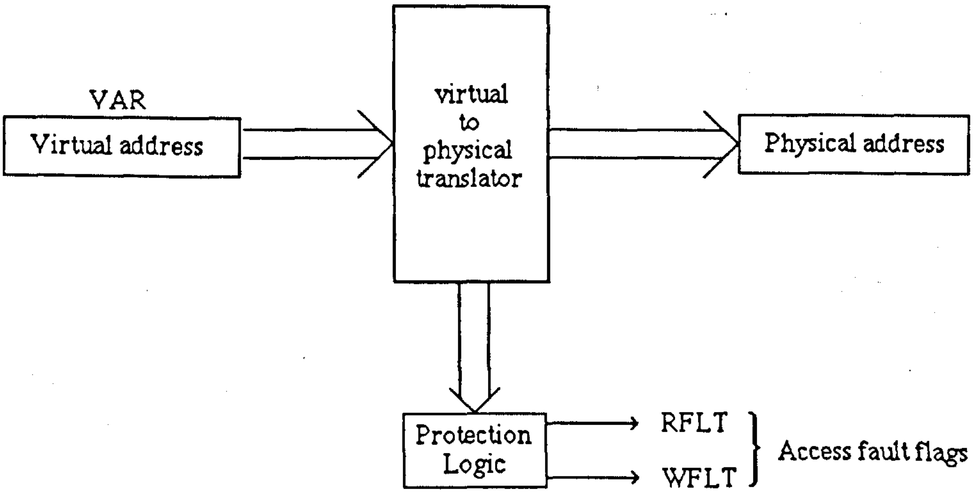

Figure 6.4 Virtual to physical translation

The states of these six protection bits determine the states of two condition flags, one registering a read fault and the other a write fault. It is these flags which are tested, by microcode, during a memory reference to check for an illegal translation. If the relevant flag is set then a microcode library routine is entered which attempts to solve the problem or, if this is not possible, it resorts to high level intervention by the operating system. For those cases which can be solved by microcode intervention alone, the library routine restarts the memory reference and returns to the original code as if nothing had happened.

The translation buffer and associated fault mechanism lend themselves to alteration for monitoring purposes, and in particular the monitoring of references to "watched" memory locations via data breakpoints or watchpoints. The translation mechanism operates at the page level as described in section 5.3.1 and so alterations are made to enable the trapping of references to "watched" pages, with the breakpoint memory employed to reduce this to a word granularity. To achieve this the translation buffer and memory-based page tables effectively require an extra protection bit which, when set, indicates a monitoring primitive is currently active on that particular page. As all six protection bits are required for other purposes, the effect of an extra bit is reproduced by an area of memory 6K words in size, forming a bitmap representation of the extra protection bit. The translation of addresses which have monitoring primitives set on them must, however, cause the two fault flags to be set and is something which cannot be accomplished with a bitmap. To overcome this one of the existing protection bits, the *accessed* bit, takes on the extra role of indicating a "watched" page. The standard use of this protection bit is to indicate a translation buffer entry which is invalid due to it not being cached. This situation arises because of the many-to-one mapping of the caching process, causing an entry of the translation buffer to not correspond to the supplied virtual address but to one of the other pages which cache onto it.

The library routine which is called when a translation fault occurs attempts to rectify the fault, using microcode, so that software intervention is not required. It is at this point that the bitmap entry for the referenced page is checked to determine if monitoring is the cause of the fault. However, even if the monitoring bit is set in the bitmap an uncached buffer entry cannot be ruled out. Thus the bitmap check is performed after the caching of the translation buffer entry. If the extra protection bit, accessed in the bitmap, is set then a monitoring primitive is known to exist on that particular page and so the accessed bit of the memory based tables and the translation buffer is reset. This results in all subsequent references to the page causing another access fault. At this point the breakpoint memory must be accessed to determine whether monitoring is active on the referenced location rather than just the page. If the breakpoint memory indicates that the location is being monitored and the type of primitive active matches the initial mode of reference (whether read or write) then the data trap flag is set, which will ultimately result in a transfer of control to the monitoring process.

6.3. Single Instruction Degradation

In this section we examine the effect of the additional monitoring microcode on the performance of the machine. It is assumed that the microcode uses only main memory storage and does not make use of fast access memory internal to the CPU. The effect of using fast memories and other features which aid the architectural support is examined in a later section.

The cost of the code breakpoint primitive is negligible when compared with the monitoring software overhead, which is incurred on transfer of control, and so only data breakpoint and watchpoint primitives are considered. We begin by calculating the cost of the monitoring support when executing a typical machine instruction. Whilst the results may not reflect the performance degradation during program execution they will indicate monitoring cases which require hardware support for a more efficient implementation.

The instruction chosen, on which to base the performance figures, is the "load word from memory" (ll_w) instruction [HLH85]. This takes a word operand, accesses it as a memory location and places the value found there on the top of the stack. The ll_w instruction is one of the more commonly found instructions, and is of medium length, and so may be taken to be typical.

Performance figures were obtained manually by adding up the timings for the individual microinstructions in each of the routines. The problem of conditionals and iterations in the microcode routines was resolved by taking the average time.

In total seven different monitoring conditions were examined and, in all cases, it is assumed that a monitoring primitive is never active on the actual location referenced by the instruction. Thus, we are measuring the interference of the monitoring microcode when a transfer of control will not occur; that is, the *continual* overhead. The timings of the seven cases are based on the following routine timings:

a) 1260ns for the instruction emulation routine.

b) 1125ns for the checking of the data trap flag between instructions.

c) 6450ns for the library routine which handles access faults.

d) 12775ns for the access of the bitmap.

e) 2925ns for the resetting of the accessed bit & access of the breakpoint memory.

f) 1950ns for the access of overflow entries in the breakpoint memory.

The cases:

(i) The instruction is executed in the standard instruction set with no additional code for monitoring purposes. It is also assumed that all memory references performed by the instruction proceed with no faults, thus not making use of the access fault code. This case is simply the execution time of the emulation routine, ie. 1260ns, giving a base for examining degradation in other cases.

(ii) The instruction is executed in the instruction set modified for monitoring purposes, resulting in a timing of 2385ns, which includes the routine which checks the data trap flag after each instruction. This case indicates that the execution time of the instruction is almost doubled even when monitoring primitives have not been set and access faults do not arise. Support for the checking of the data trap flag could thus halve the execution time of the individual instruction and would greatly improve execution times of target programs.

(iii) This case is the same as the previous case except that a monitoring primitive is active, but on a page other than that referenced by the instruction. This does not add further degradation to the performance and the total execution time is again 2385ns. It is a feature of the monitoring system not to increase target process performance overheads for monitoring primitives active on pages which are not referenced.

(iv) This case is the same as case (iii) except that a monitoring primitive is now located on the page referenced by the instruction. This is the first case where the access fault code is entered, resulting in the access of the bitmap and the breakpoint memory. The total execution time is thus 2385+6450+12775+2925ns, giving 24535ns. It is apparent that the referencing of locations on monitored pages gives rise to an immense overhead. This is due mainly to the implementation of the extra protection bit as a bitmap representation. Support for this by the provision of a monitoring bit in the page table entries would reduce this to a more practical level.

(v) This case is a generalisation of case (iv). A monitoring primitive is located on the page referenced by the instruction and X monitoring primitives are located on the same word-in-page as the referenced location. This generalised case incurs the overhead of case (iv) with the additional calculation of an overflow entry for each of the X primitives on the same word-in-page. The execution time is obtained as 24535+1950*X*ns. This shows that the performance overhead of a breakpoint memory contention is relatively small, thus allowing the monitoring software of the monitoring environment to set as many primitives as required.

The next two cases highlight the effect on performance of having to cache a translation buffer entry in the monitoring environment.

(vi) This is the same as case (i) except that the memory reference causes an accessed fault due to an uncached translation buffer entry. The standard instruction set emulation routine involving the access fault code executes in 7710ns.

(vii) This is the same as case (ii) except that, as above, the memory reference causes an accessed fault due to an uncached translation buffer entry. This involves checking the data trap flag and the bitmap representation, and totals 21610ns. Access faults in a monitoring environment thus almost triple the execution time. This is due to the checking of the bitmap, but is made more serious because of the number of access faults which occur during program execution. Context switches and the implementation of a LRU paging algorithm both reset the accessed bit, causing many more faults than would occur due to tag mismatches. As with case (iv) a monitor bit in the protection field of the page table entries would reduce this overhead considerably.

The calculated execution times for the single instruction cases described above are summarised in table II.

Table II

Case	Time(ns)	Degradation(%)
i	1260	-
ii	2385	89
iii	2385	89
iv	24535	1847
v	24535 + 1950X	1847 + 155X
vi	7710	-
vii	21610	180

6.4. Program Degradation

In this section we examine the effect of the microcode monitoring support on overall program execution times. The program constructed to obtain the timings was written specifically for this purpose, and is shown in figure 6.5.

The program was run under nine different monitoring conditions, with the timing obtained, in each case, by taking the average of ten runs. The first five cases are illustrations of cases (i) to (v) in section 6.3. The remaining four cases show the effect of monitoring predicates on the execution time of the monitoring software.

(i) The program was compiled to run in the standard instruction set and was thus not affected by the monitoring microcode.

(ii) The program was compiled to run in the modified instruction set and

```
#define PAGE 4096
#define FILLER 1588

char array0[FILLER] ;
char array1[PAGE] ;
char array2[PAGE] ;
char array3[PAGE] ;
char array4[PAGE] ;
char array5[PAGE] ;
char array6[PAGE] ;

int i,j,k ;

main()
{
   for( i = 0 ; i < 500000 ; i++ )
     array1[i] = 0 ;
   for( i = 0 ; i < 100 ; i++ )
     array2[i] = 0 ;

   proca() ;

   for( i = 0 ; i < 100 ; i++ )
     procb() ;
}

procb()
{
   int r ;
}

proca()
{
   int x,y,z ;

   for( i = 0 ; i < 500000 ; i++ ) x = 0 ;
   for( i = 0 ; i < 100 ; i++ ) y = 0 ;
}
```

Figure 6.5

was thus affected by the additional microcode, but was run with no monitoring predicates and thus no active primitives.

(iii) This case was similar to (ii) except that a watchpoint was set on element zero of the variable array6. This variable is not referenced by the program, thus showing the effect of monitoring a page other than that referenced.

(iv) The primitive set in case (iii) was now set on a page which is referenced. This was achieved by setting a watchpoint on element zero of the variable array1. The page, on which this element is stored, is accessed 500000 times by the loop on line 16.

(v) This case shows the effect of contention for breakpoint memory entries on the execution time of the target program. Watchpoints were set on element zero of the variables array1, array3, array4, and array5. Because each of these arrays begins on a page boundary and is declared to be exactly one page in length, element zero of each array is located on the same word-in-page. Again array1 causes the access faults due to the loop on line 16, and so the page is accessed 500000 times with four entries contending for the same ıreakpoint memory location.

The next four monitoring conditions show the impact of the monitoring software on performance. All transfers of control to the monitoring software which do not interact with the user produce a non-negligible performance degradation. Examples of conditions which produce such transfers of control include the monitoring of local variables, where the address of the variable is calculated by the monitoring software at run-time, and the monitoring of expressions, where this must be evaluated at run-time due to any updates to the specified variables, whether local or global.

(vi) A monitoring predicate was specified which monitors the value of elem.ent zero of the variable array2. The user is notified when it attains the value one. This predicate causes 100 transfers of control because of the loop on line 18, but never actually becomes true.

(vii) This monitoring case produces figures to show the cost of monitoring local variables. A predicate was specified which builds a graph to monitor the variable r local to the procedure procb. This variable is never accessed and so only the cost of calculation of the run-time location of the variable is obtained. The calculation is performed by the monitoring software on entry to the procedure through the use of code breakpoints as described in chapter 4. The loop on line 23 means this is performed 100 times.

(viii) Case (vii) was performed again but on the variable z local to procedure proca. The performance figures show the effect of monitoring local variables, as both the variables x and y are located on the same page. There will, thus, be one transfer of control to calculate the address of the variable z and then 500100 access faults because of the loops on line 36 and 37.

(ix) The final monitoring case examines the cost of predicates which monitor the value of local variables. A monitoring predicate was specified which monitors the value of variable y local to the procedure proca. The user is notified when it attains the value one. This predicate causes one transfer of control to calculate the run-time address of the variable, 100 transfers of control to check the value of the variable when it is updated on line 37, and 500100 access faults because of the loops on line 36 and 37.

The execution times (in seconds) obtained are shown in table III.

Table III

Case	Monitoring S/w	Target program
i	-	12.30
ii	-	28.65
iii	-	28.70
iv	-	40.35
v	-	40.40
vi	1.50	29.30
vii	6.40	31.75
viii	0.10	39.85
ix	1.60	40.55

By comparing each of the above cases against either the standard instruction set case (case (i)) or the monitoring microcode case with no active primitives (case (ii)), the performance degradation caused by that particular monitoring case can be determined. These are examined below.

1) A comparison of case (i) with case (ii) shows the degradation produced by the monitoring microcode when no primitives are active. This degradation is approximately 133%, which is caused by the checking of the data trap flag after each machine instruction and the accessing of the bitmap during access faults. This overhead will vary in proportion to the number of access faults arising.

The clock granularity of one-sixtieth of a second made it not possible to get meaningful timings for the monitoring software. This is also the case when constructing up to twenty directed graphs. Thus the degradation produced by the monitoring software on start-up may be assumed to be negligible.

2) The single instruction calculations of the last section indicate that cases (ii) and (iii) should record the same timings. That is, a primitive set on a page other than those referenced by the target program does not affect the performance. The recorded target program timing for case (iii) shows only a 0.17% increase on the timing recorded for case (ii) which can be attributed to the slight differences in execution runs and the granularity of the timings.

3) The comparison of case (iv) and (ii) shows the performance degradation associated with referencing a page which contains an active monitoring primitive. An increase in execution time of 11.70 seconds is accumulated over 500000 access faults, which is an increase of 23400ns per fault; c.f. the expected value of 22150ns calculated in the previous section.

4) The comparison of cases (v) and (ii) indicates the effect of contention for breakpoint memory entries. The increase, per fault, calculates to 23500ns, and the overall degradation rises from 228% to 228.5%. Thus, as expected, the degradation due to contention for breakpoint memory entries is negligible.

5) Case (vi) shows the effect of transfers of control between monitor

process and target process, with the monitoring software performing the expression evaluation each time. The resulting increase in execution time of 0.65 seconds corresponds to a 6.5ms per fault increase. This relatively large rise is due to the transfers of control and an increase in the number of access faults. The rise in the number of access faults occurs because the translation buffer is cleared on each transfer of control thus undoing the caching mechanism during normal operation.

The monitoring software execution time is approximately 1.5 seconds more than the negligible setup and graph construction time. It can be assumed therefore, that the 1.5 seconds is attributable to the 100 transfers of control and expression evaluations. This is likely to remain the same for most evaluations, but increasing for more lengthy expressions involving program variables.

6) The results obtained in case (vii) show the setup overhead of monitoring local variables. This setup time involves a transfer of control on procedure entry, followed by the calculation of the variable address and the setting of monitoring primitives, on this address and the return address of the procedure call. The measured execution times reveal an approximate 6.4 second increase for the 100 transfers of control and address calculations.

The slight increase in target program execution time is probably caused by references to the stack frame environment (stack pointers and return addresses) of the monitored procedure.

7) Examination of the results for case (viii) indicates an 11.2 second increase on case (ii) which, because it is over 500100 access faults, gives a 22396ns increase per fault, in line with the predicted value of 22150ns.

   The software degradation represents the setup time of the graph and the calculation of a local variable address during a single transfer of control. A value of 0.1 seconds compared to the previous case appears to indicate a higher initial overhead with a somewhat lower figure for any subsequent transfers of control.

8) Case (ix) shows the effect on performance of monitoring expressions with local variables. The overhead cons'sts of 1.5 seconds for the 100 transfers of control to check the value of the variable when it is updated and 100ms for the calculation of the run-time address.

The examination of results in this section has verified that performance figures determined for the single instruction case are confirmed in program execution, within the bounds of experimental error. However, the transferring of control does affect this generalisation by causing extra access faults due to uncached translation buffer entries. A further point indicated by the figures is that the affect of the monitoring software is also fairly predictable, with standard degradation factors for each monitoring event.

6.5. Monitoring Hardware

Examination of the figures, both calculated and generated, in the two previous sections highlight the areas which contribute most to performance degradation. In this section we examine the impact of possible hardware support for these areas.

The figures for the hardware cases introduced below are calculated in the same way as for the single instruction cases in section 6.3, with estimates for the routines which cannot be implemented.

The two main causes of degradation are the checking of the data trap flag after every machine instruction emulation routine and the accessing of the bitmap representation of the monitor bit in the protection status of a page table entry.

Hardware 1

The checking of the data trap flag almost doubles the execution time of a target program, even if no monitoring primitives are active. In order to create an efficient monitoring system it is essential that hardware support is provided to implement the data trap flag.

The provision of a flag which can be set and used as a condition code in conjunction with the microprogram sequencer functions will remove most, if not all, of the performance degradation associated with the checking of the data trap flag.

The microcode version with no primitives in section 6.4 (case ii) increased the execution time by 133%. This would be reduced to only 44% if the above hardware were provided. Similarly the degradation for case (iv), in section 6.4, where 500000 access faults occur, is reduced from 228% to 139%.

Hardware 2

The above hardware proposal is most effective in those monitoring cases where few access faults arise. However, for those cases where many access faults occur it can be seen that the prime cause of the performance degradation is the checking of the bitmap representation of the extra protection bit.

Reserving a bit in the protection status of a page table entry for monitoring purposes removes the need for a bitmap and consequently the code to access it. The extra monitor bit in the protection status is tested for in the same way as the other protection status bits are tested, to determine the reason for an access fault. Performance savings are made in two areas. Firstly, access faults due to an uncached translation buffer entry or a tag mismatch do not need to access and check the bitmap, and secondly, monitored pages which are cached do not need to perform the caching mechanism unnecessarily.

Applying these figures to the program execution in section 6.4 gives an overall degradation, for case (iv), of 156% instead of 228%.

Hardware 3

By applying both of the above hardware proposals a system can be constructed which incurs a negligible performance overhead in all cases except when a monitored page is referenced.

Applying these figures to the program in section 6.4 gives a zero performance overhead for the cases where a monitored page is not referenced. The case where monitored pages are referenced (case iv) incurs only an 18% overhead, in comparison with 228% for the standard microcode implementation.

Hardware 4

The performance degradation of hardware 3, above, is now solely based on the breakpoint memory access and the checking of the contents of appropriate entries. This access can be made faster through the use of a fast memory.

Applying this to the program of section 6.4 gives a target program increase, for case (iv), of 1.8 seconds, resulting in an overall degradation of only 15%.

However, the fast memory available on the Orion is a global resource, requiring either a reload on a context switch or the limiting of the system to one user at a time. If the reloading option is taken then this will affect the performance advantage of using the fast memory.

Table IV shows the calculated timings for the single instruction case and the corresponding timings for the hardware introduced, and table V gives the figures for the percentage degradation.

Table IV

	Timings(ns)				
Case	Microcode	Hardware 1	Hardware 2	Hardware 3	Hardware 4
(i) Base	1260	1260	1260	1260	1260
(ii) Mon. O/head	2385	1260	2385	1260	1260
(iii) Unref. prim	2385	1260	2385	1260	1260
(iv) Ref prim	24535	23410	6835	5710	4835
(v) Contention	24535+1950X	23410+1950X	6835+1950X	5710+1950X	4835+275X
(vi) Base fault	7710	7710	7710	7710	7710
(vii) Mon. fault	21610	20485	8835	7710	7710

This examination of possible architectural support has shown that with minor hardware modifications it is possible to implement monitoring primitives which incur a very modest overhead. One of the main requirements for architectural support, outlined in chapter 5, is that monitoring must not incur a performance overhead until a monitoring primitive is active. This is satisfied by the architectural support outlined in this section.

In the worst case, when a monitoring primitive is active on the page referenced, then the performance degradation is still quite modest, incurring less than a 300% overhead for a typical instruction performing the reference. All other cases, including access faults, incur either zero or negligible overheads.

Table V

	Degradation(%)				
Case	Microcode	Hardware 1	Hardware 2	Hardware 3	Hardware 4
(i) Base	-	-	-	-	-
(ii) Mon. O/head	89	0	89	0	0
(iii) Unref. prim	89	0	89	0	0
(iv) Ref prim	1847	1758	442	353	284
(v) Contention	1847+155X	1758+155X	442+155X	353+155X	284+22X
(vi) Base fault	-	-	-	-	-
(vii) Mon. fault	180	166	15	0	0

6.6. Case Studies

In this section we examine the impact of monitoring on the execution of two programs. This will include the performance degradation imposed by the experimental monitoring system, implemented on the HLH Orion, and also the estimated impact if the hardware, described in the previous section, was available.

Case 1

A cross-referencer program is executed with the following six different monitoring conditions.

(i) Execution under the standard instruction set with no support for monitoring.

(ii) Execution under the monitoring microcode with no monitoring primitives active. Performance is affected by the bitmap access on access faults and the checking of the data trap flag.

(iii) As case (ii) but a global variable is monitored. This variable is never updated but does cause 93712 access faults, due to references to the monitored page.

(iv) A global variable is monitored as part of a monitoring expression. This requires monitoring software intervention to perform the expression evaluation each time the variable is updated. The monitored page is accessed 423565 times causing the same number of access faults, whilst the monitored variable is updated 208 times.

(v) A local variable is monitored, requiring monitoring software intervention to calculate the address of the local variable on procedure entry. This occurs 178 times with 19704 access faults.

(vi) A local variable is monitored as part of a monitoring expression. The variable is updated only once during execution of the procedure but the procedure is called 138 times, and 417 access faults take place.

As in previous sections the degradation can be explained by the number of access faults for the target program and the number of transfers of control for the monitoring software. Table VI shows the above results and the estimated timings assuming the availability of architectural support described as "hardware 4" in the previous section.

Table VI

		Microcode			Hardware		
Case	Software (sec)	Target (sec)	Target deg (%)	Overall deg (%)	Target (sec)	Target deg (%)	Overall deg (%)
i	-	6.30	-	-	6.30	-	-
ii	-	13.45	113	113	6.30	0	0
iii	-	15.90	152	152	6.65	6	6
iv	4.50	23.35	271	342	7.80	24	95
v	14.55	16.65	164	395	6.35	1	232
vi	11.35	15.50	146	326	6.30	0	180

The above figures show that the execution time of the target program can be lowered to an acceptable level through the use of hardware support (a maximum increase of only 24%). However, the influence of the monitoring software becomes the dominant factor in the overall degradation. Thus the performance is determined by the number of transfers of control to the monitoring software and not the execution time of the target program.

Case 2

A benchmark program is executed with the same six conditions as for the first program above. However, the benchmark program consists of fewer variables updated in loops, thus causing a higher number of access faults per second of execution. The six monitoring conditions are given below.

(i) Execution under the standard instruction set with no support for monitoring.

(ii) Execution under the monitoring microcode with no monitoring primitives active.

(iii) As case (ii) but a global variable is monitored. This variable is never updated but does cause 3601265 access faults, due to references to the monitored page.

(iv) A global variable is monitored as part of a monitoring expression. The monitored page is accessed 300008 times causing the same number of access faults, whilst the monitored variable is only updated once.

(v) A local variable is monitored, requiring monitoring software intervention to calculate the address of the local variable on procedure entry. This occurs only once with 8821267 access faults.

(vi) A local variable is monitored as part of a monitoring expression. The variable is updated 150 times during execution of the procedure, which is called only once, and 8821561 access faults take place.

Table VII shows the above results and the estimated timings assuming the availability of architectural support described as before, i.e. "hardware 4".

Table VII

		Microcode			Hardware		
Case	Software	Target	Target deg	Overall deg	Target	Target deg	Overall deg
	(sec)	(sec)	(%)	(%)	(sec)	(%)	(%)
i	-	37.30	-	-	37.30	-	-
ii	-	90.05	141	141	37.30	0	0
iii	-	171.60	360	360	50.15	34	34
iv	0.05	97.60	162	162	38.35	3	3
v	0.11	283.30	660	660	68.85	85	85
vi	2.90	284.35	662	670	68.85	85	92

In contrast to the first case the impact of the monitoring software is negligible and the overall degradation is due to the high number of accesses to monitored pages, resulting from the clustering of target program variables and the number of references to them.

6.7. Cost of Monitoring

The microcoded implementation described in this chapter has shown a monitoring overhead of between 113% and 141% even with no monitoring primitives active. For the same case studies this overhead increases to between 152% and 360% when a primitive is active. The use of architectural support, as described in chapter 5, removes any overhead when monitoring primitives are not active and reduces the overhead for the case when a primitive is active. For the case studies in this chapter the overhead was reduced to between 6% and 34%. This level of performance degradation is likely to be acceptable in most non real-time environments.

Architectural support for the three monitoring primitives does not, however, affect the overhead incurred by the monitoring software. One of the case studies showed a performance degradation of over 200% for local variable monitoring, even after the assumption of architectural support for the monitoring primitives. A similar monitoring predicate for the other case study showed an overhead of less than 100%, and so it is difficult to predict a general percentage overhead for the monitoring software. This overhead, however, is incurred only when primitives are invoked, and almost any system of monitoring (without the use of expensive parallel hardware) is likely to lead to significant overheads at this stage.

7. Conclusions

The research described in this thesis has been directed towards the design of software tools and hardware support for program execution monitoring. Execution monitoring has applications in program testing, to establish test data coverage and to uncover data flow anomalies; in program debugging, to locate and identify program bugs; and in performance analysis. In recent years, however, the emphasis has been on the writing of correct code, with an increase in the use of specification aids, structured programming techniques, and research into formal methods of establishing program correctness.

Despite these developments it would appear that it is still necessary for software to enter the testing, debugging and evaluation stages of development. However, there has been less research into these stages than other stages have enjoyed. Due to an increase in the use of modern high-level languages the semantic gap between the programmer's abstract view of the software and the execution monitoring tools available has increased. Monitoring systems developed for assembly language programs are now totally unsuitable for high-level language monitoring.

In chapter 2, the requirements for an execution monitoring tool, operating at the level of modern software, were identified. These include the provision of full control over target program execution, the ability to view the entire working space of the target process, the ability to monitor in a language independent but language sensitive manner, and the ability to monitor programs which have had optimisation techniques applied to them during compilation. Research into the latter two requirements has produced systems which are language sensitive (for example, RAIDE and VAX DEBUG) and can monitor optimised versions of programs (for example, NAVIGATOR). However, the level of control and observability offered by many monitoring systems still resembles that of the classical monitoring tools, developed for assembly language programs. Monitoring methods still involve the setting of simple code breakpoints on source statements and the tracing of global variables. Systems which do offer monitoring facilities for high-level software often resort to extremely inefficient modes of operation such as simulation or the single-stepping of machine instructions.

Facilities for the kind of control required have been implemented to some extent for software in a real-time environment. Due to the nature of this software it is important that monitoring is non-intrusive, and so this often results in specialised parallel hardware, which is prohibitively expensive in most circumstances.

The aim of the work described in this thesis was to examine ways of providing monitoring facilities allowing the required level of control over the the target process and with an acceptable performance overhead, and without the need for extensive hardware support.

In chapters 3 and 4, the software structures required to implement a monitoring environment with these requirements were examined. The general requirement for user-level interaction was postulated to take the form of a WHEN command which performs some monitoring action when a monitoring predicate is satisfied. To enable the monitoring of a general predicate, three levels were identified at which monitoring predicates may be specified. These are the primitive level, expressed in terms of the execution steps taken at the machine level; an abstract level, expressed in terms of the notation and semantics of the high-level language in use; and a conditional level, which describes a process state involving sequences of abstract level predicates.

At the primitive level it was found sufficient to provide a set of three monitoring primitives: the code breakpoint, the data breakpoint and the watchpoint. A predicate at the abstract level can be implemented simply by translating it into one or more of the monitoring primitives. This requires information defining the relationship between source and object program and also in certain cases the mirroring of the stack operation of the target process. At the conditional level, the translation is further complicated by the absence, in general, of any representation of the condition as a single monitoring primitive or set of primitives. For this reason we introduced the idea of an event-graph for monitoring conditional level predicates. Each node of the graph represents an event, or specific state of the process control and data space, which can be recognised by simple inspection of the target process state using one of the monitoring primitives. The arcs of the event-graph indicate the chronological sequencing of the events, enabling predicates to be monitored which do not, in general, preserve the process state information during execution. It has been shown that the form of event-graphs introduced in this work enables a representation of the monitoring of program conditions involving procedures, recursion, dynamic variables and other features of high-level language programming.

To enable a practical implementation of the above system it was necessary for the monitoring primitives to incur only a small performance overhead. Methods of implementing architectural support for the primitives were examined in chapter 5 and a method described for use with a virtual memory architecture. This method involved altering the virtual to physical translation mechanism to "watch" for monitored locations. An additional monitor bit was added to the page table entries of the virtual memory management system which indicates a primitive located on that particular page. A breakpoint memory could be accessed if this was the case to determine whether a breakpoint was active on the word of the page, and if so a trap-like interrupt was caused to enable the transfer of control to the monitor process.

An experimental system to determine the effectiveness of the ideas introduced was implemented on the High Level Hardware Orion, a user-microprogrammable machine. The monitor and target process synchronisation was provided by fairly conventional means, using UNIX system calls. The monitoring software structures and associated algorithms were implemented in the C programming language and the support for the monitoring primitives provided in microcode.

Evaluation of the effectiveness of the methods implemented was performed for a typical machine instruction; an actual program, run under the experimental system, to determine whether the results from the single instruction case could be generalised to program execution; and two case studies.

The results obtained were encouraging, showing that the experimental implementation offers high-level monitoring for a reasonable performance cost. The figures, however, showed that performance degradation could be substantially reduced by the provision of simple hardware support, consisting of: an extra condition flag, indicating a trap is to be taken; an extra monitor bit in the page table entries; and a fast memory implementation of the breakpoint memory. The typical "background" interference (caused by references to monitored pages), with this support in place, is estimated at only 15%. This figure might even be tolerated in a real-time environment where a slight performance degradation during development is usually acceptable.

The design of the high-level notation required for interaction with a program for the purpose of execution monitoring was largely outside the scope of the research described here. However, the implementation of a complete monitoring system using the structures developed in this work requires the definition of a suitable form of user interface. Particular issues in the design of this include the use of graphical images, and the method of making the system available to a number of high-level languages.

Appendix: Selected code fragments

This appendix contains selected procedures from the experimental system described in chapter 6.

The procedure 'take_primitive' is called due to a transfer of control from the target process. The address and type passed to this procedure is obtained via the area reserved for communication between the two processes (see section 6.1.3). The list of *logical* traps associated with the actual trap is traversed, the trap identifier accessed and passed to a procedure 'taketrap' which performs the necessary actions within the event-graph.

The procedure 'set_primitive' takes as parameters the trap identifier, address and type. A *logical* primitive is set by adding the identifier to the list of *logical* traps. Only if necessary is an actual primitive set.

The procedure 'databrkpt' takes an address and updates the bitmap representation of the additional *monitor* bit, and adds an entry to the breakpoint memory.

take_primitive

```
take_primitive( address,type )
unsigned address ;
int type ;
{
   /* This procedure attempts to find a trap taken by the machine in the
      list constructed by software.  If no trap exists then we cannot
      continue as machine and software are out of sync * /.

    int index ;
    int found ;
    struct trap *temp ;

    found = FALSE ;
    index = address & WORD_IN_PAGE ;
    temp = table_of_traps[type][index] ;

    while( temp != null_trap ) {
        /* Search through list of traps with same word-in-page for the
            address given. Any found are passed to the procedure 'taketrap' * /
        if( ((*temp).address == address ) {
            taketrap( (*temp).trap_number,address ) ;
             found = TRUE ;
        }
        temp = (*temp).overflow ;
    }
    if( !found ) {
        printf("***FATAL ERROR in take_primitive***\n") ;
        printf("Took machine trap for which no software trap existed\n") ;
        printf("address = %x\ttype = %s\n",address, print_type[type]) ;
    }
}
```

```
set_primitive( trap_no,address,type )
int trap_no ;
unsigned address ;
int type ;
{
    int index ;
    int set ;
    struct trap *temp ;

    /* Set a primitive at the address given and of the type given.
       This primitive is identified by the number given.
       An actual primitive is only set if one does not already exist,
       otherwise the identifier is simply added to the list of
       identifiers for that primitive */

    index = address & WORD_IN_PAGE ;
    set = TRUE ;
    temp = table_of_traps[type][index] ;

    while( (temp != null_trap) && set ) {
        /* Search the list of identifiers to see if a trap is already
           active at that address */
        if( (*temp).address == address ) set = FALSE ;
        temp = (*temp).overflow ;
    }

    /* Add the identifier to the list of identifiers */
    temp = ( struct trap * ) malloc ( sizeof( struct trap ) ) ;
    (*temp).address = address ;
    (*temp).trap_number = trap_no ;
    if( type == PRIM_CODE )
        (*temp).instrs = ptrace( MRD,pid,address,0 ) ;
    (*temp).overflow = table_of_traps[type][index] ;
    table_of_traps[type][index] = temp ;

    if( set ) {
        /* A primitive is required at the location specified */
        switch( type ) {
        case PRIM_CODE :
                /* set a code breakpoint and replace instruction by the code
                   breakpoint instruction */
                codebrkpt( address,ptrace( MRD,pid,address,0 ) ) ;
                ptrace( MWR,pid,address,TRAP ) ;
                break ;
        case PRIM_DATA :
                /* set a data breakpoint */
                databrkpt( address ) ;
                break ;
        case PRIM_WCH :
                /* set a watchpoint */
                watchpt( address ) ;
                break ;
        }
    }
}
```

databrkpt

```
databrkpt(address)
unsigned address ;
{
    unsigned index ;
    unsigned entry ;
    unsigned memval ;
    unsigned bitmap ;
    unsigned mask ;

    /* Set a data breakpoint at the address given */

    /* Place segment bits (2 MSBits) at top of page */
    if( address & 0x80000000 )
        bitmap = address | 0x08000000 ;
    else
        bitmap = address & ~0x08000000 ;
    if( address & 0x40000000 )
        bitmap = bitmap | 0x04000000 ;
    else
        bitmap = bitmap & ~0x04000000 ;

    bitmap = bitmap >> 10 ;  /* Remove word-in-page bits */
    mask = bitmap & 0x1f ;  /* 'mask' contains bottom 5 bits of page number */
    bitmap = bitmap >> 5 ;  /* Remove these 5 bits */
    bitmap = bitmap & 0x1fff ;  /* 'bitmap' contains offset into the bitmap
                                          representation */
    bitmap = bitmap + BITMAP ;  /* Add 'bitmap' to the address of the bitmap
                                          representation */
    mask = 1 << mask ;  /* 'mask' is a 32-bit w^rd with the appropriate bit set
                                  for accessing the bitmap representation */
    ptrace(MWR,pid,bitmap,ptrace(MRD,pid,bitmap,0) | mask) ;  /* Set this bit
                                                                  using the
                                                                  'ptrace' system
                                                                  call */


    index = address & WORD_IN_PAGE ;  /* 'index' contains word-in-page of
                                                  original address */
    entry = DATATABLE + index ;  /* 'entry' contains the address of the
                                            breakpoint memory entry to access */
    memval = ptrace(MRD,pid,entry,0) ;  /* 'memval' contains the breakpoint
                                                     memory entry at this address */

    while( ( memval & ACTIVE ) != 0 ) {
        /* Search breakpoint memory (following o.erflow chain if necessary)
           for either the end of the chain or an entry which is not active */
        if( ( memval & OVF_ACTIVE ) == 0 ) break ;
        entry = DATAOVFLOW + ( ( memval >> 1 ) & OVF_MASK ) ;
        memval = ptrace(MRD,pid,entry,0) ;

    }
    if( ( memval & ACTIVE ) == 0 ) {
        /* If an entry is found which is not active then insert the new
           entry here */
        memval = memval | ACTIVE | DATABIT | (address & PAGE) ;
        ptrace(MWR,pid,entry,memval) ;
    }
    else {
        /* Add a new link to the overflow chain and insert the new entry */
        memval = memval | ( datastackptr << 1 ) | OVF_ACTIVE ;
        ptrace(MWR,pid,entry,memval) ;
        entry = DATAOVFLOW + datastackptr ;
        ptrace(MWR,pid,entry,(ACTIVE | DATABIT | (address & PAGE))) ;
        datastackptr++ ;
    }
}
```

References

[Abramson83]

D. Abramson & J. Rosenberg "Hardware Support for Program Debuggers in a Paged Virtual Memory" *ACM Computer Architecture News* Vol. 11 No. 2 June 1983 pp. 8-19

[Adams86]

E. Adams and S.S. Muchnick "Dbxtool: A Window-Based Symbolic Debugger for SUN Workstations" *Software - Prac.ice and Experience* Vol. 16 No. 7 1986 pp. 653-669

[Agarwal86]

A. Agarwal, R.L. Sites & M. Horowitz "ATUM: A New Technique for Capturing Address Traces Using Microcode" *Proceedings of the 13th Annual Symposium on Computer Architecture* 1986 pp. 119-127

[Alford77]

M. Alford "A Requirements Engineering Methodology for Real-Time Processing Requirements" *IEEE Transactions on Software Engineering* Vol. SE-3 No. 1 January 1977 pp. 60-69

[Ambras88a]

J. Ambras and V. O'Day "MicroScope: A Knowledge-Based Programming Environment" *IEEE Software* May 1988 pp. 50-58

[Ambras88b]

J.P. Ambras, L.M. Berlin, M.L. Chiarelli, A.L. Foster, V. O'Day, and R.N. Splitter "MicroScope: An Integrated Program Analysis Toolset" *Hewlett-Packard Journal* August 1988 pp. 71-82

[Aral88a]

Z. Aral and I. Gertner "High-Level Debugging in Parasight" *ACM Workshop on Parallel and Distributed Debugging* May 1988 pp. 151-159

[Aral88b]

Z. Aral, I. Gertner, and G. Schaffer "Efficient Debugging Primitives for Multiprocessors" *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* 1988 pp. 87-95

[Arisawa80]

M. Arisawa and M. Iuchi "Debugging Methods in Recursive Structured FORTRAN" *Software - Practice and Experience* Vol. 10 1980 pp. 29-43

[Armbruster79]

C.E. Armbruster,Jr. "A Microcoded Tool to Sample the Software Instruction Address" *ACM Sigmicro Newsletter* No. 10 1979 pp. 68-72

[Ashby73]

G. Ashby, L. Salmonson, and R. Heilman "Design of an Interactive Debugger for FORTRAN: MANTIS" *Software - Practice and Experience* Vol. 3 1973 pp. 65-74

[Atkinson78]

M.P. Atkinson and M.J. Jordan "An Effective Program Development Environment for BCPL on a Small Computer" *Software - Practice and Experience* Vol. 8 1978 pp. 265-275

[Babcicky80]

J. Babcicky "Some Thoughts on Debugging Tools" *SIMULA Newsletter* Vol. 8 No. 4 November 1980 pp. 11-13

[Baiardi83]

F. Baiardi, N. de Francisco, E. Matteoli, S. Stefanini, and G. Vaglini "Development of a Debugger for a Concurrent Language" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 98-106

[Balzer69]

R.M. Balzer "EXDAMS - Extendable Debugging and Monitoring System" *AFIPS Conference Proceedings* Vol. 34 1969 pp. 567-580

[Balzer81]

R. Balzer "GIST Final Report" *Information Sciences Institute, University of Southern California* February 1981

[Barnes74]

D.H. Barnes and L.L. Wear "Instruction Tracing via Microprogramming" *ACM 7th Annual Workshop on Microprogramming* October 1974 pp. 25-27

[Barra83]

K. Barra and H.P. Dahle "SIMOB - A Portable Toolbox for Observation of SIMULA Executions (Extended Abstract)" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 121-122

[Bates82]

P.C. Bates and J.C. Wileden "EDL: A Basis for Distributed System Debugging Tools" *Proceedings of the 15th Hawaii International Conference on System Sciences* 1982 pp. 86-93

[Bates83]

P.C. Bates and J.C. Wileden "An Approach to High-Level Debugging of Distributed Systems (Preliminary Draft)" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 107-111

[Bauer77]

F.L. Bauer "Software Engineering: An Advanced Course" *Springer, New York* 1977

[Bayer67]

R. Bayer, D. Gries, M. Paul, and H.R. Wiekle "The ALCOR Illinois 7090/7094 Postmortem Dump" *Communications of the ACM* Vol. 10 No. 12 December 1967 pp. 804-808

[Beander83]

B. Beander "VAX DEBUG: An Interactive, Symbolic, Multilingual Debugger" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 173-179

[Bemmerl86]

T. Bemmerl "Realtime High Level Debugging in Host/Target Environments" *Microprocessing and Microprogramming* 18, 1986 pp. 387-400

[Bishop81]

J.M. Bishop and D.W. Barron "Principles of descriptors" *Computer Journal* Vol. 24 No. 3 August 1981 pp. 210-221

[Bovey87]

J.D. Bovey "A Debugger for a Graphical Workstation" *Software - Practice and Experience* Vol. 17 No. 9 1987 p. 647-662

[Brown73]

A.R. Brown and W.A. Samps›n "Program Debugging" *New York: American Elseveir and MacDonald* 1973

[Bruegge83a]

B. Bruegge and P. Hibbard "Generalized Path Expressions: A High-Level Debugging Mechanism" *The Journal of Systems and Software* Vol. 3 1983 pp. 265-276

[Bruegge83b]

B. Bruegge and P. Hibbard "Generalized Path Expressions: A High-Level Debugging Mechanism" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 34-44

[Burkhart84]

H. Burkhart and R. Millen "High-Level Language Monitoring: Design Concepts and Case Study" in *Advances in Microprocessing and Microprogramming,* B. Myhrhaug and D.R. Wilson (eds) 1984 pp. 177-186

[Cantone87]

G. Cantone, A. Cimitile, and U. de Carlini "Testability and Path Testing Strategies" *Microprocessing and Microprogramming* 21, 1987 pp. 371-382

[Cardell83]

J.R. Cardell "Multilingual Debugging with the SWAT High-Level Debugger" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 180-189

[Cargill83]

T.A. Cargill "The Blit Debugger (preliminary draft)" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 190-200

[Cargill85]

T.A. Cargill "Implementation of the Blit Debugger" *Software - Practice and Experience* Vol. 15 No. 2 February 1985 pp. 153-168

[Cargill87]

T.A. Cargill and B.N. Locanthi "Cheap Hardware Support for Software Debugging and Profiling" *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems* October 1987 pp. 82-83

[Cdb(1)]

"HP-UX Reference, Volume 1" *Hewlett-Packard Co., Fort Collins, Colorado* 1988

[Cheatham79]

T.E. Cheatham,Jr., G.H. Holloway, and J.A. Townley "Symbolic Evaluation and the Analysis of Programs" *IEEE Transactions on Software Engineering* Vol. SE-5 No. 4 July 1979 pp. 402-417

[Clark83]

B.E.J. Clark and S.K. Robinson "A Graphically Interacting Program Monitor" *The Computer Journal* Vol. 26 No. 3 August 1983 pp. 235-238

[Clarke83]

L.A. Clarke and D.J. Richardson "The Application of Error-Sensitive Testing Strategies to Debugging" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 45-52

[Clarke89]

L.A. Clarke, A. Podgarski, D.J. Richardson, and S.J. Zeil "A Formal Evaluation of Data Flow Path Selection Criteria" *IEEE Transactions on Software Engineering* Vol. 15 No. 11 November 1989 pp. 1318-1332

[Cohen77]

J. Cohen and N. Carpenter "A Language for Inquiring About the Run-Time Behaviour of Programs" *Software - Practice and Experience* Vol. 7 1977 pp. 445-460

[Cook83]

R.P. Cook and I. Lee "Dymos: A Dynamic Modification System" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 201-202

[Davies86]

A.C. Davies and A.S. Goussous "Graphbug - A Microprocessor Software Debugging Tool" *Microprocessors and Microsystems* Vol. 10 No. 4 May 1986 pp. 195-201

[Dbx(1)]

"Volume 1 UNIX Programmers Manual, Orion Time Sharing", release 2 *High Level Hardware Ltd., Windmill Road, Oxford.* January 1986

[DeBlasi77]

M. DeBlasi, N. Fanelli, G. Giannelli, and G. Degli Antoni "Profile Finder, A Firmware Instrument for Program Measurements" *Euromicro Journal* Vol. 3 1977 pp. 27-33

[Deutsch82]

M.S. Deutsch "Software Verification and Validation: Realistic Project Approaches" *Prentice-Hall, Englewood Cliffs* 1982

[Digital82]

Digital Equipment Corporation "VAX-11 Architecture Reference Manual" *Digital Equipment Corporation, Maynard, Mass.* 1982

[Digital86]

Digital Equipment Corporation "VMS Debugger Manual" *Digital Equipment Corporation, Maynard, Mass.* 1986

[Ditzel80]

D.R. Ditzel and D.A. Patterson "Retrospective on High-Level Language Computer Architecture" *Computer Architectural News* Vol. 8 No. 3 1980 pp. 97-104

[Elliott82]

B. Elliott "A High-Level Debugger for PL/I, FORTRAN and BASIC" *Software - Practice and Experience* Vol. 12 1982 pp. 331-340

[Evans65]

T.G. Evans and D.L. Darley "DEBUG - An Extension to Current Online Debugging Techniques" *Communications of the ACM* Vol. 8 No. 5 May 1965 pp. 321-326

[Evans66]

T.G. Evans and D.L. Darley "On-Line Debugging Techniques: A Survey" *AFIPS Conference Proceedings* Vol. 29 1966 pp. 37-50

[Exec(3)]

"Volume 1 UNIX Programmers Manual, Orion Time Sharing", release 2 *High Level Hardware Ltd., Windmill Road, Oxford.* January 1986

[Fairley79]

R.E. Fairley "ALADDIN: Assembly Language Assertion Driven Debugging Interpreter" *IEEE Transactions on Software Engineering* Vol. SE-5 No. 4 July 1979 pp. 426-428

[Fairley85]

R.E. Fairley "Software Engineering Concepts" *McGraw-Hill Book Co.* 1985

[Feldman88]

S.I. Feldman and C.B. Brown "IGOR: A System for Program Debugging via Reversible Execution" *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* 1988 pp. 112-123

[Feldman89]

M.B. Feldman and M.L. Moran "Validating a Demonstration Tool for Graphics-Assisted Debugging of Ada Concurrent Programs" *IEEE Transactions on Software Engineering* Vol. 15 No. 3 March 1989 pp. 305-313

[Ferguson63]

H.E. Ferguson and E. Berner "Debugging Systems at the Source Language Level" *Communications of the ACM* Vol. 6 No. 8 August 1963 pp. 430-432

[Ferrante83]

J.Ferrante "High Level Debugging with a Compiler" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 123-129

[Fork(3)]

"Volume 1 UNIX Programmers Manual, Orion Time Sharing", release 2 *High Level Hardware Ltd., Windmill Road, Oxford.* January 1986

[Foxley78]

E.Foxley and D.J. Morgan "Monitoring the Run-Time Activity of Algol68-R Programs" *Software - Practice and Experience* Vol. 8 1978 pp. 29-34

[Frankl88]

P.G. Frankl and E.J. Weyuker "An Applicable Family of Data Flow Testing Criteria" *IEEE Transactions on Software Engineering* Vol. 14 No. 10 October 1988 pp. 1483-1498

[Fritzson83]

P. Fritzson "A Systematic Approach to Advanced Debugging Through Incremental Compilation (preliminary draft)" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 130-139

[Fryer73]

R.E. Fryer "The Memory Bus Monitor - A New Device for Developing Real-Time Systems" *AFIPS Conference Proceedings* Vol. 42 1973 pp. 75-79

[Gentleman83]

W.M. Gentleman and H. Hoeksma "Hardware Assisted High Level Debugging (preliminary draft)" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 140-144

[Gerrard90]

G.P. Gerrard, D. Coleman, and R.M. Gallimore "Formal Specification and Design Time Testing" *IEEE Transactions on Software Engineering* Vol. 16 No. 1 January 1990 pp. 1-12

[Girgis85]

M.R. Girgis and M.R. Woodward "An Integrated System for Program Testing using Weak Mutation and Data Flow Analysis" *IEEE Proceedings of the 8th International Conference on Software Engineering, London* August 1985 pp. 313-319

[Gladstone76]

B. Gladstone "MONITOR/DEBUGGER Saves Time when Checking μP Software" *EDN (USA)* Vol. 21 No. 17 1976 pp. 69-80

[Glass80]

R.L. Glass "Real Time: The "Lost World" of Software Debugging and Testing" *Communications of the ACM* Vol. 23 No. 5 May 1980 pp. 264-271

[Goodman82]

R.S. Goodman "A Multilingual High Level Debugging Facility" *IEEE Phoenix Conference on Computers and Communication* 1982 pp. 182-185

[Goossens83]

M. Goossens and J. Tiberghien "High Level Debugging Aids for Real Time Software" *Microcomputers: Development in Industry, Business and Education, Euromicro* 1983 pp. 71-78

[Gramlich83]

W.C. Gramlich "Debugging Methodology" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 4-8

[Grätsch81]

W. Grätsch and H. Kästner "Firmware Monitoring - History and Perspective" *Microprocessing and Microprogramming* 8, 1981 pp. 237-246

[Groll78]

K.H. Groll and J.U. Petrie "Program Tracing" *IBM Technical Disclosure Bulletin* Vol. 21 No. 3 August 1978 pp. 1187-1188

[HLH84]

High Level Hardware Ltd. "Orion Microarchitecture Reference Manual", second edition *High Level Hardware Ltd., Windmill Road, Oxford.* 1984

[HLH85]

High Level Hardware Ltd. "Orion Macroarchitecture Reference Manual", second edition *High Level Hardware Ltd., Windmill Road, Oxford.* 1985

[Hamilton83]

G. Hamilton "Logic Analyzer gives Programmers Real-Time View of Software Performance" *Electronics* May 5, 1983 pp. 117-122

[Hanson76]

D.R. Hanson "Variable Associations in SNOBOL4" *Software - Practice and Experience* Vol. 6 1976 pp. 245-254

[Hanson78]

D.R. Hanson "Event Associations in SNOBOL4 for Program Debugging" *Software - Practice and Experience* Vol. 8 1978 pp. 115-129

[Hart79]

J.J. Hart "The Advanced Interactive Debugging System (AIDS)" *ACM SIGPLAN Notices* Vol. 14 No. 12 1979 pp. 110-121

[Head71]

R.V. Head "Automated Systems Analysis" *Datamation* Vol. 17 1971 pp. 22-24

[Hennessy82]

J. Hennessy "Symbolic Debugging of Optimized Code" *ACM Transactions on Programming Languages and Systems* Vol. 4 No. 3 July 1982 pp. 323-344

[Hill83]

C.R. Hill "A Real-Time Microprocessor Debugging Technique" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 145-148

[Howden75]

W.E. Howden "Methodology for the Generation of Program Test Data" *IEEE Transactions on Computers* Vul. C-24 No. 5 May 1975 pp. 554-559

[Huang78]

J.C. Huang "Program Instrumentation and Software Testing" *IEEE Computer* Vol. 11 No. 4 1978 pp. 25-32

[Huang79]

J.C. Huang "Detection of Data Flow Anomaly Through Program Instrumentation" *IEEE Transactions on Software Engineering* Vol. SE-5 No. 3 May 1979 pp. 226-236

[Huang80]

J.C. Huang "Instrumenting Programs for Symbolic-Trace Generation" *Computer Magazine* December 1980 pp. 17-23

[Huang84]

J.C. Huang, M. Ho, and T. Law "A Simulator for Real-Time Software Debugging and Testing" *Software - Practice and Experience* Vol. 14 No. 9 September 1984 pp. 845-855

[Hurst84]

A.J. Hurst "A Source Language Performance Monitoring Facility for the B1800 Modula Interpreter" *ACM Sigmicro Newsletter* September 1984 pp. 28-36

[IBM61]

IBM "Study Organization Plan (SOP)" *Form No. C 20-8075* White Plains, NY. 1961

[Itoh73]

D. Itoh and T. Izutani "FADEBUG-I, A New Tool for Program Debugging" *IEEE Symposium on Computer Software Reliability* 1973 pp. 38-43

[Jackson75]

M.A. Jackson "Principles of Program Design" *Academic Press* 1975

[Johnson77]

M.S. Johnson "The Design of a High-Level, Language-Independent Symbolic Debugging System" *Proceedings of the ACM Annual Conference* 1977 pp.315-322

[Johnson78a]

S.C. Johnson "YACC: Yet Another Compiler-Compiler" *Volume 2 UNIX Programmers Manual, Orion Time Sharing, release 2* High Level Hardware Ltd., Windmill Road, Oxford. January 1986

[Johnson78b]

M.S. Johnson "The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment" *Technical Report* 78-6 August 1978

[Johnson79]

M.S. Johnson "Translator Design to Support Run-Time Debugging" *Software - Practice and Experience* Vol. 9 1979 pp. 1035-1041

[Johnson81]

M.S. Johnson "DISPEL: A Run-Time Debugging Language" *Computer Languages* Vol. 6 1981 pp. 79-94

[Johnson82]

M.S. Johnson "Some Requirements for Architectural Support of Software Debugging" *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* 1982 pp. 140-148

[Kernighan78]

B.W. Kernighan and D.M. Ritchie "The C Programming Language" *Prentice-Hall, Englewood Cliffs, New Jersey* 1978

[King76]

J.C. King "Symbolic Execution and Program Testing" *Communications of the ACM* Vol. 19 No. 7 July 1976 pp. 385-394

[Kishimoto83a]

Z. Kishimoto "A System for Program Validation, Revalidation and Debugging" *IEEE Phoenix Conference on Computers and Communications* 1983 pp. 442-446

[Kishimoto83b]

Z. Kishimoto "An Experimental Debugger in a Limited Programming Environment (extended abstract)" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 63-66

[Kopetz79]

H. Kopetz "Software Reliability" *The Macmillan Press Ltd* 1979

[Lauesen79]

S. Lauesen "Debugging Techniques" *Software - Practice and Experience* Vol. 9 1979 pp. 51-63

[Lazzerini86]

B. Lazzerini and C.A. Prete "DISDEB: An Interactive High-Level Debugging System for a Multi-Microprocessor System" *Microprocessing and Microprogramming* 18, 1986 pp. 401-408

[Lazzerini89]

B. Lazzerini and L. Lopriore "Abstraction Mechanisms for Event Control in Program Debugging" *IEEE Transactions on Software Engineering* Vol. 15 No. 7 July 1989 pp. 890-901

[Lemon79]

L. Lemon "Hardware System for Developing and Validating Software" *Proceedings of the 13th ASILOMAR Conference on Circuits, Systems and Computers* Pacific Grove, 1979 pp. 455-459

[Liskov86]

B. Liskov and J. Guttag "Abstraction and Specification in Program Developmen." *The MIT Electrical Engineering and Computer Science Series, MIT Press McGraw-Hill* 1986

[Lloyd80]

R. Lloyd, H. Ovies, J.L. Rosado, and D.J. Willson "Programmable Map and Trace Instrument" *IBM Technical Disclosure Bulletin* Vol. 23 No. 5 October 1980 pp. 2075-2078

[Lorin81]

H. Lorin and H.M. Deitel "Operating Systems" *Addison-Wesley Publishing Co.* 1981

[Lucas71]

H.C. Lucas, Jr "Performance Evaluation and Monitoring" *Computing Surveys* Vol. 3 No. 3 September 1971 pp. 79-91

[Lyttle90]

D. Lyttle "A symbolic Debugger for Real-Time Embedded Ada Software" *Software - Practice and Experience* Vol. 20 No. 5 May 1990 pp. 499-514

[Maekawa87]

M. Maekawa, A.E. Oldehoeft, and R.R. Oldehoeft "Operating Systems Advanced Concepts" *The Benjamin/Cummings Publishing Company Inc.* 1987

[Mann73]

G.A. Mann "A Survey of Debug Systems" *Honeywell Computer Journal* Vol. 7 No. 3 1973 pp. 182-198

[Martin88]

J. Martin and C. McClure "Structured Techniques: The Basis for CASE" *Prentice-Hall, Englewood Cliffs NJ* 1988

[McDaniel82]

G. McDaniel "The Mesa Spy: An Interactive Tool for Performance Debugging" *Performance Evaluation Review* Vol. 11 No. 4 1982 pp. 68-76

[McGregor80]

D.R. McGregor and J.R. Malone "Stabdump - A Dump Interpreter Program to Assist Debugging" *Software - Practice and Experience* Vol. 10 1980 pp. 329-332

[McLear82]

R.E. McLear, D.M. Scheibelhut, and E. Tammaru "Guidelines for Creating a Debuggable Processor" *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* 1982 pp. 100-106

[Mellor-Crummey89]

J.H. Mellor-Crummey and T.J. LeBlanc "A Software Instruction Counter" *Computer Architecture News* Vol. 17 No. 2 April 1989 pp. 78-86

[Melvin86]

S.W. Melvin and Y.N. Patt "A Microcode-Based Environment for Non-Invasive Performance Analysis" *The 19th Annual Workshop on Microprogramming* 1986

[Mikelsons83]

M. Mikelsons "Interactive Program Execution in Lispedit" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp.71-80

[Miller88]

B.P. Miller and J. Choi "A Mechanism for Efficient Debugging of Parallel Programs" *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* 1988 pp. 141-150

[Motorola82]

"MC68000 16-Bit Microprocessor User's Manual" *Prentice-Hall 1982*

[North77]

S. North "A Dynamic Debugging System" *Creative Computing (USA)* Vol. 3 No. 5 1977 pp. 26-28

[Ntafos88]

S.C. Ntafos "A Comparison of Some Structural Testing Strategies" *IEEE Transactions on Software Engineering* Vol. 14 No. 10 October 1988 pp. 868-874

[Pan88]

D.Z. Pan and M.A. Linton "Supporting Reverse Execution of Parallel Programs" *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* 1988 pp. 124-129

[Pierce74]

R.H. Pierce "Source Language Debugging on a Small Computer" *Computer Journal* Vol. 17 No. 4 November 1974 pp. 313-317

[Plattner81]

B. Plattner and J. Nievergelt "Monitoring Program Execution: A Survey" *IEEE Computer* Vol. 14 November 1981 pp. 76-93

[Plattner84]

B. Plattner "Real-Time Execution Monitoring" *IEEE Transactions on Software Engineering* Vol. SE-10 No. 6 November 1984 pp. 756-764

[Powell83]

M.L. Powell and M.A. Linton "A Database Model of Debugging" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp.67-70

[Ptrace(2)]

"Volume 1 UNIX Programmers Manual, Orion Time Sharing", release 2 *High Level Hardware Ltd., Windmill Road, Oxford.* January 1986

[Quirk85]

W.J. Quirk "Verification and Validation of Real-Time Software" *Springer-Verlag* 1985

[Ramamoorthy76]

C.V. Ramamoorthy, S.F. Ho, and W.T. Chen "On the Automated Generation of Program Test Data" *IEEE Transactions on Software Engineering* Vol. SE-2 No. 4 December 1976 pp. 293-300

[Richardson89]

S. Richardson and M. Ganapathi "Code Optimization Across Procedures" *Computer* Vol. 22 No. 2 February 1989 pp. 42-49

[Rijks87]

E. Rijks, J. Vermeesch, M. Goossens, and J. Tiberghien "Integration of a Hardware Module for Tracing Local Variables in Realtime Software" *Microprocessing and Microprogramming* 21, 1987 pp. 639-646

[Ross77]

D.T. Ross and K.E. Shoman "Structured Analysis for Requirements Definition" *IEEE Transactions on Software Engineering* Vol. SE-3 No. 1 1977 pp. 6-15

[Saal72]

H.J. Saal and L.J. Shustek "Microprogrammed Implementation of Computer Measurement Techniques" *ACM 5th Annual Workshop on Microprogramming* 1972 pp. 42-50

[Satterthwaite72]

E. Satterthwaite "Debugging Tools for High Level Languages" *Software - Practice and Experience* Vol. 2 1972 pp. 197-217

[Seidner83]

R. Seidner and N. Tindall "Interactive Debug Requirements" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 9-22

[Shaw78]

A.C. Shaw "Software Descriptions with Flow Expressions" *IEEE Transactions on Software Engineering* Vol. SE-4 No. 3 May 1978 pp. 242-254

[Signal(3)]

"Volume 1 UNIX Programmers Manual, Orion Time Sharing", release 2 *High Level Hardware Ltd., Windmill Road, Oxford.* January 1986

[Small85]

C.H. Small "Software Analyzer Traces High-Level-Program Execution in Real-Time" *EDN (USA)* Vol. 30 No. 9 1985 pp. 83-84

[Smith82]

M.F. Smith "Debugging with Filtered Traces" *Computer Design* April 1982 pp.143-146

[Socha88]

D. Socha, M.L. Bailey, and D. Notkin "Voyeur: Graphical Views of Parallel Programs" *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* 1988 pp. 206-215

[Steffen84]

J.L. Steffen "Experience with a Portable Debugging Tool" *Software - Practice and Experience* Vol. 14 No. 4 April 1984 pp. 323-334

[Tanenbaum84]

A.S. Tanenbaum "Structured Computer Organization", second edition *Prentice-Hall, Inc. Englewood Cliffs NJ.* 1984

[Teichrow77]

D. Teichrow and E. Hershey "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems" *IEEE Transactions on Software Engineering* Vol. SE-3 No. 1 January 1977 pp. 41-48

[Tiberghien86]

J. Tiberghien "Development Tools" *Microprocessing and Microprogramming* 18, 1986 pp. 383-386

[Tischler83]

R. Tischler, R. Schaufler, and C. Payne "Static Analysis of Programs as an Aid to Debugging" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 155-158

[Tratner79]

M. Tratner "A Fundamental Approach to Debugging" *Software - Practice and Experience* Vol. 9 1979 pp. 97-99

[Tsai90]

J.J.P. Tsai, K. Fang, and H. Chen "A Non-Invasive Architecture to Monitor Real-Time Distributed Systems" *Computer* Vol. 23 No. 3 March 1990 pp. 11-23

[Venables89]

P.J. Venables and H. Zedan "Debugging and Monitoring Highly Parallel Systems with GRIP" *Microprocessing and Microprogramming* Vol. 28 1989 pp. 79-84

[Victor77]

K.E. Victor "The Design and Implementation of DAD, a Multiprocess, Multimachine, Multilanguage Interactive Debugger" *Proceedings of the 10th Hawaii International Conference on System Science* 1977 pp. 196-199

[Wait(2)]

"Volume 1 UNIX Programmers Manual, Orion Time Sharing", release 2 *High Level Hardware Ltd., Windmill Road, Oxford.* January 1986

[Walter83]

C.K. Walter "DELTA - The Universal Debugger for CP-6" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 203-205

[Watson70]

R.W. Watson "Timesharing system design concepts" *McGraw-Hill Computer Science Series* 1970

[Weyuker86]

E.J. Weyuker "Axiomatizing Software Test Data Adequacy" *IEEE Transactions on Software Engineering* Vol. SE-12 No. 12 December 1986 pp. 1128-1138

[Weyuker90]

E.J. Weyuker "The Cost of Data Flow Testing: An Empirical Study" *IEEE Transactions on Software Engineering* Vol. 16 No. 2 February 1990 pp. 121-128

[Wilkes51]

M.V. Wilkes, D.J. Wheeler, and S. Gill "The Preparation of Programs for an Electronic Digital Computer" *Addison-Wesley* 1951

[Winder88]

R. Winder and J. Nicolson "JDB: An Adaptable Interface for Debugging" *Software - Practice and Experience* Vol. 18 No. 3 1988 pp. 221-238

[Witschorik83]

C.A. Witschorik "The Real-Time Debugging Monitor for the Bell System 1A Processor" *Software - Practice and Experience* Vol. 13 1983 pp. 727-743

[Young88]

M. Young and R.N. Taylor "Combining Static Concurrency Analysis with Symbolic Execution" *IEEE Transactions on Software Engineering* Vol. 14 No. 10 October 1988 pp. 1499-1511

[Zelkowitz71]

M. Zelkowitz "Reversible Execution as a Diagnostic Tool" *PhD Thesis, Cornell University* January 1971

[Zellweger83]

P.T. Zellweger "An Interactive High-Level Debugger for Control-Flow Optimized Programs" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* 1983 pp. 159-171