Replay of Concurrent Shared-Memory Programs

Diplomarbeit

von

Oliver Schuster

aus

Wesseling

vorgelegt am

Lehrstuhl für Praktische Informatik IV

Prof. Dr. Wolfgang Effelsberg

Fakultät für Mathematik und Informatik

Universität Mannheim

April 1999

Betreuer: Prof. Dr. Wolfgang Effelsberg

Acknowledgements

This thesis would not exist without the help and support of several people. First of all, I want to thank my supervisor at the University of Waterloo, Canada, Professor Dr. Peter Buhr, for his outstanding support and guidance. He was a resource for technical discussions and taught me how to approach and realise a scientific project.

I thank Professor Dr. Wolfgang Effelsberg who coordinates the Waterloo-Mannheim exchange and keeps it alive. He is also my supervisor at the University of Mannheim, Germany.

I also thank my readers, Professor Dr. Ric Holt and Professor Dr. Stephen Mann, and Professor Dr. Gordon Cormack for their valuable comments.

Furthermore I thank all my friends in Canada and Germany for their constant support and encouragement. Ashif Harji provided me with a constant flow of technical information as well as a lunch partner. Charles Fortin was willing to show me that I can still improve my Squash skills and taught me a different perspective of Canada. Dorothee Wilbs believed in me and reminded me that there is a world outside of computer science, which is quite fascinating.

I also thank my lab mates, Ming, Dorota and Tom, and the exchange student gang from Mannheim, Ralf, Achim, Jan Peter and Michael.

All this would not have been possible without my parents, who encouraged me to see the world and supported me while doing so.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Püfungsbehörde vorgelegen.

Waterloo, den 14. April 1999 Oliver Schuster

Abstract

Concurrent programs are non-deterministic. This is a significant obstacle during the debugging cycle, since an error might not reoccur in a subsequent execution.

In this thesis, approaches for replaying concurrent programs are examined and the design and implementation of SMART, the Shared Memory Application Replaying Tool for μC++ programs is presented.

Replaying is a useful tool because it allows the cyclical debugging approach to be applied to concurrent applications. Since concurrent applications are non-deterministic, it may be hard to recreate an error situation. If the source for non-determinism is recorded and subsequently reproduced, the exact order of execution can deterministically be recreated, and multiple executions of an application for the purpose of inspection and debugging is possible.

SMART is a tool which records the scheduling activities and exact position of timeslices in a μC++ application. Based on these data, the execution of a uniprocessor program can be recreated. Input and time queries can be an additional sources of non-determinism. Currently, the user of SMART needs to ensure that these sources behave in a deterministic fashion.

Trademarks

Abbreviations

AI	Artificial Intelligence
CREW	Concurrent-read-exclusive-write
DCE	Distributed Computing Environment
FIFO	First In, First Out
GDB	The GNU Source-Level Debugger
GUI	Graphical User Interface
I/O	Input / Output
ID	Identifier
IDL	Interface Description Language
ISO	International Standardisation Organisation
KDB	Kalli's DeBugger
OSF	Open Software Foundation
PC	Program Counter
POET	Partial Order Event Tracer
RPC	Remote Procedure Call
SIC	Software Instruction Counter
SMART	Shared Memory Applications Replaying Tool
STL	Standard Template Library
μC++	Micro C++

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> Debugging is the dirty little secret of computer science. Despite the progress we have made in the past 30 years – faster computers, networking, easy-to-use graphical interfaces, and everything else – we still face some embarrassing facts about software development. Henry Lieberman [41, page 27]

1.1 The Debugging Scandal

When reading the above paragraph and the article that goes along with it, it becomes clear that debugging is a neglected field in computer science. In the modern world of today it is possible to use software in building cars without any human interaction, but the tools that exist for helping programmers find bugs in their software still need a high level of interaction and attention. Many programmers do not even bother to learn how to use a debugger or other tools for searching for bugs. Print statements still seem to be the most popular debugging method, even though it is tedious to use them and far better interactive tools exist.

> Some people express despair of any possibility of making significant improvements in the debugging process. Debugging is just plain hard, they say. Many programmers display a macho attitude, saying real programmers don't need debugging tools. Henry Lieberman [41, page 28]

Lieberman and several other experts in the field disagrees with this pessimistic statement. They believe that good tools can significantly help detect the cause for bugs and write better software. In this thesis, I throw my lot into the pool of tools and attempt to make the scandal less scandalous.

A lot of progressive debugging research originated in functional languages like Lisp or Scheme [52, 70] because of the dynamic nature of these languages. Dynamic binding and typing result in significant information about the program being necessary at runtime, which can then be used to facilitate debugging.

Statically bound and typed languages normally do not have this information and the imperative nature of these languages results in significant state that must be examined. I will to present a new tool named SMART for statically bound and typed concurrent languages. SMART stands for “Shared Memory Application Replaying Tool”.

1.2 Debugging Approaches

There are several approaches to debugging with varying degrees of sophistication. The approaches taken vary among programmers and each has a preferred method.

In a survey done by Eisenstadt [21], four significant categories were found. The first category and the one used most often is called “Gather Data”. It consists of the `print` statement added to the source code as well as using tools to control and analyse the resulting executable to obtain more information about the bug. Stepping through the code and setting conditional and unconditional breakpoints using a debugger is part of this category as well as using a profiler or wrapper function to analyse the code. More sophisticated tools to detect memory leaks and other error conditions can be found in this group, too.

The second category, “Inspeculation”, is a hybrid of “inspection” of code, “simulation” by hand and plain “speculation”, otherwise called trial-and-error. Even though this method largely depends on a sudden moment of enlightenment or plain luck, it is used often and successfully. The next category is called “Expert Recognized Clichés”. In this case, the problem is presented to a fellow programmer who recognises it and explains the bug and a possible solution. The last category, “Controlled Experiments”, applies if the root problem is already identified and needs to be verified and clarified. Specific input is fed to the application to verify the problem and obtain more details about the cause to clarify the problem.

In all categories, especially in the first, second, and last one, debugging is a cyclic activity. McDowell and Helmbold [53] call the classical approach to debugging “cyclical debugging”.

The cycle of debugging consists of the two phases [35]:

1. Developing a hypothesis about faulty behaviour of the program.

2. Verifying the hypothesis with successive confirmation or rejection.

1.3 Concurrent Debugging

Debugging of a concurrent program is, compared to debugging a sequential program, significantly more complicated, because of the move from a single thread to multiple threads. In general, a human being is able to concentrate only on a small number of things at any given time. It is nearly impossible to consciously follow different threads in parallel. This complexity is the main reason why developing concurrent programs is hard. But other reasons exist, and they expand into the field of debugging.

A sequential program is deterministic in the order of execution if the same input is provided, i.e., each run of a sequential application yields the same results. Concurrent programs are non-deterministic, because the execution path of each individual thread depends on the execution path of the other threads in the system. Due to timeslicing and non-deterministic scheduling decisions, the execution path of one thread might change, and therefore change the execution path of other threads and the result of the whole application.

The non-deterministic execution and parallel access to resources can cause different kinds of errors that do not exist and cannot occur in sequential programs. Among them are race conditions, deadlocks and livelocks.

A race condition is the result of a lack of synchronisation. Deadlocks and livelocks result in a lack of progress made by the involved threads. In the deadlock case, all involved tasks are blocked forever, in livelocks, a selection algorithm postpones the progress of the involved tasks. A more detailed discussion of these errors is presented in section 4.1 and in [8, chapter 7].

It is nearly impossible to use the cyclical debugging approach to detect and analyse these kind of errors. In addition, a debugger changes the timing behaviour significantly by inserting additional code into the executable and stopping individual threads while the programmer analyses them, which might prevent the error from occurring at all. This additional effect is called the “Probe Effect” [25] or the “Heisenberg Uncertainty Principle” applied to software [39, page 98]. Bugs that are altered when using a debugger or other tool are sometimes called “Heisenbugs”.

If it is possible to recreate an execution path exactly, including the potential error, and to re-examine the execution again and again, using the cyclical debugging approach, then Heisenbugs, race conditions, deadlocks and livelocks can be detected and eliminated.

This thesis attempts to solve the dilemma by recording an execution path, and thus providing the possibility to replay an execution path multiple times.

1.4 Outline

In chapter 2, related work in the field of replaying is presented and a short summary of each approach is presented. The work is categorised based on the communication mechanisms used.

Chapter 3 gives an introduction to μC++, a user-level thread library for C++ and to KDB, a concurrent debugger for programs written in μC++. The implementation of this thesis is incorporated into these two tools.

Some design considerations are discussed in chapter 4 explaining the background for elemental decisions made during the design and implementation phase. Causes for non-determinism in concurrent programs are analysed, and the choice for a restriction to uniprocessor programs is explained. Also, the design goals and restrictions are established and presented.

The recording side is presented and explained in chapter 5. The implementation of the software instruction counter is discussed and the data that needs to be recorded is considered. The points at which the recording should occur are discussed.

Chapter 6 presents the replaying side. Two cases, replaying of blocking due to yielding and replaying of blocking due to timeslices, are discussed. The interaction of the schedulers in the μC++ kernel and the debugger KDB are presented and the need for this hybrid approach is explained.

A detailed example of a debugging session using SMART is presented in chapter 7. The necessary preparation before recording, the recording run itself, and the replaying part of a debugging cycle with SMART is shown and the steps are explained in detail.

In chapter 8, preliminary work is presented concerning replaying to real-time programs. The real-time extensions of μC++ are shown as well as the possibilities and limitations of replaying real-time programs.

A conclusion and an outlook to potential future work is presented in chapter 9.

Chapter 2

Related Work

Research in replaying spans more then twenty years. Several approaches have been tried. The approaches can be grouped depending on the underlying communication mechanisms. Concurrent programs communicate by either exchanging messages or by accessing shared memory [71]. The mechanism used is essential in determining what data must be recorded for future replaying.

A shared-memory system might be implemented on a distributed environment as it is in Agora [5]. Relevant for the following categorisation is the mechanism that is most significant for replaying. In Agora, the messages exchanged to synchronise the shared-memory are important. Because of that, Agora is sorted into the category of message-passing systems. The following categorisation is therefore not necessarily based on the categorisation of the underlying communication model.

The following sections discuss various approaches to replaying and contrast each one of them to SMART.

2.1 Message Passing Systems

Event based replaying systems can be based on the messages exchanged among nodes. The communication among different nodes determines the flow of the program.

2.1.1 BugNet

BugNet [15] is a debugging system written for MICRONET, a cluster of sixteen loosely coupled microcomputer nodes. MICROS [82] is an example of a distributed system where the nodes communicate via message-passing. The MICROS project uses Modula-2 [81] for both system and application development. A debugger provides a non-graphical user interface to control the execution of each process. Basic debugging features such as starting, stopping and stepping are supported. Special functionality is provided so that a user can control the amount of message traffic displayed and monitor the information flow among the threads. Examination and manipulation of local variables and other entities is also possible. BugNet is able to reproduce an execution environment based on the interprocess communication. Exact reproduction cannot be guaranteed because of possible non-deterministic selection features.

A central database exists to support the distributed debugging in MICOS. This database monitors all interprocess communication and queries the event monitors, which exist on each node. The event monitors provide information about the local state. The user interface is connected to the central database.

Each local event monitor records all messages and periodically checkpoints all variables into a local circular buffer. Because of the circular structure of the buffer, older information is eventually overwritten when newer information has to be stored.

Upon encountering an error, the user may select to replay the events leading towards the error. Each node selects the oldest available checkpoint and restarts execution from there. Tasks with newer checkpoints are suspended until they are synchronised with the other nodes.

BugNet bases recording on the messages exchanged among nodes, which is characteristic for a message-passing system.

Since μC++ provides mechanisms based on shared-memory for synchronisation and communication among tasks, it is impossible to implement the ideas realised for BugNet in μC++.

Due to the circular buffer used for checkpointing in BugNet, older information might get lost. It might not be possible to replay the complete execution path of an application.

In SMART, a complete execution is replayable.

2.1.2 Agora

The Agora system [5, 23] has been used in a large, heterogeneous system [1] at Carnegie Mellon University. It is a distributed shared-memory system. A shared data structure is kept consistent across the nodes in the network and is converted for the different machine architectures, e.g., the byte ordering is switched.

The Agora debugger uses checkpoints and events as the mechanism for replay. The shared-memory is checkpointed initially and again periodically. The programmer needs to specify when checkpoints should be set depending on the application. On the initial checkpoint, the full shared-memory is dumped to disk, on each subsequent one, only the accessed memory pages are saved. The underlying Mach [2] operating system supports the capabilities to replace the routines that write or read memory to disk, allowing Agora to determine the accessed pages.

For exact replay, events based on the interaction among processes are defined. The events are activation, and read and write accesses on the shared data structures. Using Lamport's [37] clock algorithm, these events are put in partial order. Stopping a process that produces events preceding events in other processes stops these other processes, too.

Agora is an implementation for a specific shared-memory system that relies on the messages exchanged to synchronise the memory. Checkpoints have to be specified by the user. This demands a certain amount of interaction from the user and a deeper understanding of the Agora system.

SMARTdoes not demand a deep understanding of it by the user. It is usable without having to configure it first for each individual program to be debugged. No special features of the operating system are used; a standard UNIX system is sufficient.

2.1.3 POET

POET [77], the "Partial Order Event Tracer", is a tool developed at the University of Waterloo by David Taylor, Jay Black, Thomas Kunz and others. It collects and displays event traces of a distributed application. POET is written for multiple systems but the most extensive support is for the Open Software Foundation's (OSF) Distributed Computing Environment (DCE) [59].

POET has been augmented with a Replay Facilitator [84,85] to provide replaying for DCE applications. The communication among the participating machines in DCE is done via remote procedure calls (RPCs). These are recorded by augmenting the stub code generated by the interface description language (IDL) compiler. The captured events are stored in disk servers. A primary disk server stores the execution history during recording, a secondary server observes it during replay.

The replay facilitator uses the data recorded and stored in the primary disk server to create a replay control structure, which is used to manage replay. Based on the recorded partial order of events, the replay facilitator communicates with agents in the application on the distributed machines.

By delaying threads and controlling the execution of RPCs, the replay facilitator recreates the partial order of events as provided from the primary disk server. The events occurring during replaying are send to the secondary disk server. Based on the progress of the application, queried by the replay facilitator from the secondary disk server and the events stored during the recording phase, the replaying progresses.

As with other approaches based on message passing, it is impossible to implement this approach for μC++. Although there are mechanisms to use POET with μC++ [78], replaying a μC++ application based on events would not be sufficiently accurate because other communication and synchronisation mechanisms might be used that are not recorded nor monitored.

2.2 Shared Memory Systems

Two different approaches exist for recording execution paths in shared-memory systems. The first technique comes from the underlying concept of replaying in message-passing systems. Every access to shared-memory is considered to be an event, and as such, is recorded. This method is **data** oriented. The second approach is based on the fact that the **order** of execution determines the order of access to shared-memory.

2.2.1 Data

For the **data** oriented approach, events have to be generated, which determine the synchronisation among threads. Memory access is a good candidate for the source of the events, but synchronisation mechanisms such as semaphores are feasible as well.

2.2.1.1 Recap

Recap [60] is a tool that provides the illusion of reverse execution for parallel programs. Its design is independent of the operating system. Recap is part of the Allegro [43] programming environment developed at Stanford University. It was intended to integrate Recap into Ispy, a window-oriented debugger. The following discussion is based on the design of Recap.

Each process has a local event log, which logs external events such as system calls and shared-memory access. Shared-memory accesses are determined by augmenting load instructions in the assembler code. Three cases have to be differentiated, either a non-shared variable is loaded and no additional code is necessary, or a shared variable is loaded and an event has to be generated to log the access, or it is unclear if the access is shared or not. In the last case, the compiler generates a conditional check, which dynamically determines if the access is shared or not based on a table that indicates which memory pages are shared.

Simulating the logging by hand based on a VAX-11/780, 1Mb of data per process per second is generated. Since this is a huge amount of data, a mechanism had to be found to reduce it. A sliding window of log events is suggested, which represents the last N seconds from the current execution time. The severe cutoff is not considered significant since a processor can perform substantial amounts of computation in a small number of seconds, so sufficient events exist in most situations to allow replay.

During the monitoring phase, the events are logged, during the replay phase external events are provided by the event log and internal events are checked and synchronised with the log. At regular intervals during the monitoring phase, a process is checkpointed. The original process is forked and the forked process continues execution while the original process is suspended and stored for use during replay.

Reverse execution is realised by letting the user scroll to a certain point of execution. The nearest checkpoint before this point is taken and simulated up to the point of execution using the stored events. Signals are recreated by utilising the program counter and measuring the execution time. This method is not precise, but is assumed to be close enough in practice.

As mentioned, Recap produces about 1 Mb of data per second. The storage and administration of this vast amount of data takes time and space. Recording only short windows for reverse execution does not seem to be acceptable since it may be necessary to replay longer running programs.

SMART only yields small amounts of data. The data can be stored in fast memory during recording, keeping the time for administration of the data small. No expensive access to slower, larger memory is necessary during recording. The complete execution path is recorded with SMART.

The implementation of checkpoints in the Recap approach is also resource intensive. A thread is duplicated at regular intervals and multiplies the allocated memory resources.

SMART does not depend on checkpoints and does not need a checkpoint mechanism.

Fundamentally, the operational and memory overhead necessary for the Recap approach is too large. It has a significant impact on the usability of the program itself during recording. Therefore, a different approach is needed.

2.2.1.2 Igor

Igor [22] is a system that traces accesses to memory based on write accesses to memory pages. It is built for the DUNE [61] operating system at Bellcore. The prototype implementation was written for C programs on a 68000 hardware platform. It only supports debugging of single threaded programs. Nevertheless, interesting approaches are considered and realised in Igor.

The initial idea for Igor came from trying to revive a program after it terminated and dumped core. A tool was developed that enables a user to restart a program based on its core image. This capability is not always possible, but in most cases it yields reasonable results. The program has to be augmented with a routine to support restarting after abnormal termination. The program is continued after the C statement that caused the core dump.

The next extension lead to the introduction of checkpoints. The execution state is written out periodically to a set of files during execution. To support this feature, DUNE was augmented to provide a command named `pagemod`, which returns all memory pages that changed since the previous call to `pagemod`. These pages are written to disk with a timestamp, along with the contents of the registers and the program counter.

When a program executes, three different times, `real`, `system`, and `user` time, are normally available. The first value gives the total time of execution. The system time is the time spend in system calls and other operating system functions. The user time is the time spend executing the application. For Igor, DUNE is augmented with a new system call, `ualarm`, which provides an interrupt after a certain amount of user time elapsed. The normal UNIX `alarm` [44] system call provides the same functionality based on the `real` time[1].

In DUNE, the `alarm` system call only provides a granularity of 1.0 seconds. The `ualarm` call has the advantage of 0.01 second granularity.

The new `ualarm` system call is used to trigger the creation of checkpoints in Igor. Therefore, checkpoints are triggered based on the user time, which reflects the execution more accurately, and checkpoints are triggered with a finer granularity.

The resulting program had an execution time overhead of between 43 and 280 percent, depending on the checkpoint intervals and the application.

The Igor system has similar disadvantages to Recap. The memory usage might be less, but execution time overhead of up to 280 percent is unacceptable. A user would probably not use a tool regularly that slows the normal execution down by up to four times.

SMART only slows execution by about 10 percent during normal execution. This figure is considered to be acceptable.

Besides, Igor is unable to recreate the exact execution to a sufficient level of granularity. It is only possible to recreate the execution up to a checkpoint, but execution points between checkpoints cannot be accurately reproduced. This lack of functionality defeats the purpose of replaying.

SMART replays execution exactly down to assembler-code granularity. This granularity

[1]A standard UNIX system provides the `setitimer` [45] command, which provides a possibility to set an interval timer based on either the real time, the process time (virtual time), or the combined time the process executes and the system is executing on behalf of the process. This system call does not seem to exist in DUNE.

is sufficient to replay race conditions and provides the possibility to detect Heisenbugs.

2.2.2 Order

Recording the order of execution is an approach that is more complicated but it yields significantly less data, resulting in better usability. However, it is nontrivial to record and replay based on order in a shared-memory system. Several approaches try to solve this problem, but none of them are completely successful on a multiprocessor shared-memory system.

2.2.2.1 Deterministic Execution

The order of execution of a concurrent program depends on the synchronisation among the different tasks. Semaphores [17] and monitors [30] are examples of primitives that provide the ability to control ordering.

Carver and Tai [12] present an approach implemented for UCSD Pascal and Ada, which records the utilisation of synchronisation primitives. The first step in their approach is to define the format of a so called synchronisation sequence consisting of the process in which the synchronisation occurs, the type of synchronisation event and the synchronisation primitive utilised. An example for the synchronisation primitive is a semaphore, and synchronisation events are opening and closing of the semaphore.

In the second step, the synchronisation sequence has to be recorded, which is done by altering the synchronisation primitive. The information is stored as a *triple*.

In the third and last step, the recorded synchronisation sequence is replayed. The synchronisation primitives are altered again to obtain a *permit* before performing any synchronisation event. This permit is granted if the event is supposed to occur according to the recorded data.

The presented approach relies on proper utilisation of synchronisation primitives. Mutually exclusive access of shared variables is assumed to be guaranteed, meaning that shared variables are only accessed in a critical section, which is protected by synchronisation primitives like a semaphore or a monitor.

The approach by Carver and Tai assumes a race free program. This assumption is too restrictive considering that the tool is used for debugging. Race conditions are common errors in concurrent programs. Replaying and reverse execution is possibly an excellent tool to detect these conditions. It should not be assumed that they do not exist in the program to be debugged.

SMART does not assume a race free program. It records race conditions and replays them exactly as recorded.

2.2.2.2 Instant Replay

Developed for the Butterfly parallel processor, Instant Replay [38] is a system that records the order of execution. The implementation extends the Chrysalis [55] operating system, which was written for the Butterfly machine.

Instant Replay requires a protocol that ensures a valid serialisation of accesses to shared objects. One possibility is the concurrent-read-exclusive-write (CREW) protocol [14].

If a process wants to read a shared object, it must use a certain entry procedure for read access and an exit procedure after finishing the read. For writing, similar wrapper methods exist. The implementation of the wrapper methods is based on the CREW protocol, which solves the reader-writer problem. Code is added to write read-write access information to a file during monitoring.

The protocol data is augmented with a version number, which is monotonically incremented after each write access to a shared object. Since multiple readers but only one writer can access a shared object in the CREW protocol, the total number of readers in a read group are counted and added to the protocol information. A writer logs the version number and the number of readers in the previous read group; a reader only logs the version number on each access.

In replay mode, a reader blocks when the version number of an object does not correspond to the version number of the next read access in the history file. A writer blocks as well, based on the version number. If the object has the correct version number, the writer gets the number of readers previously recorded and waits until this amount of read accesses has occurred. Afterwards it changes the shared object.

The Chrysalis operating system provides operations on a set of data types that can be shared among all processes executing on the machine. The operating system provides primitives to access these data types as well. Instant Replay was implemented by changing these primitives to conform to the CREW protocol. Some of the primitives already provided mutual exclusion. Since mutual exclusion is stricter than the CREW protocol, it is unnecessary to augment these primitives.

Instant Replay requires little memory. Each log entry to the history file requires four bytes. An example program solving the knight's tour problem, which calculates a sequence of moves for a knight through which the figure touches each square on a chess board exactly once, produced between 250 and 300 communication events per second. The program ran less than a minute to solve the problem, the exact execution time depends on the number of processors used. The total amount of space needed for recording did not exceed 72 kilobytes.

For Instant Replay it is necessary to heavily augment the operating system. This capability is unavailable in many cases and makes the resulting replay mechanism platform dependent.

SMART is implemented for conventional non-research platforms. It is unnecessary to augment the operating system or run in any privileged execution modes.

Further, Instant Replay requires that all access to shared objects conforms to the CREW protocol or is mutually exclusive. This restriction precludes race conditions since all shared objects are in an adequately protected critical section. The programmer is restricted by a system that does not provide sufficient freedom for parallel programming. Forcing only the CREW protocol or mutual exclusion on the access to all shared variables potentially inhibits concurrency.

μC++ [11], the programming language which SMART augments, provides no restrictions to the programmer in implementing a concurrent application.

2.2.2.3 Repeatable Scheduling Algorithm

The Repeatable Scheduling Algorithm by Russinovich and Cogswell [64] is implemented for the Mach 3.0 [62] operating system. It concentrates on uniprocessor concurrent systems.

The key to the repeatable scheduling algorithm is the fact that the order of scheduling is the only factor that determines the order of execution in a uniprocessor system. Because of that, recording the activities of the scheduler is sufficient to recreate the exact path of execution.

To record the position at which a thread is scheduled, the PC register and a software instruction counter value are logged. Few processors [29] provide hardware instruction counters. Therefore, it is normally necessary to implement an instruction counter in software.

The approach is effective and yields less than a kilobyte of logging data per second, depending on the scheduling activities of the program. On average, the time overhead during recording is about ten percent, and during replay between eight and fifteen percent.

While the Repeatable Scheduling Algorithm only applies to uniprocessor systems, it produces a small amount of logging data, is able to reproduce race conditions exactly and does not slow the system down significantly. However, the software instruction counter is implemented in the Replaying Scheduling Algorithm by augmenting a program at the assembler level making the approach highly platform dependent.

SMART heavily leans on the ideas presented by Russinovich and Cogswell; however, the software instruction counter is implemented at the source code level, which provides better portability.

For the Repeatable Scheduling Algorithm, the scheduler is replaced and augmented during recording and replaying. In μC++, scheduling has been generalised by allowing ready queue management to be replaced at the user-level. This capability provides high flexibility to the programmer, e.g., for implementing real-time scheduling algorithms. In SMART, the scheduler specified by the programmer or the default scheduler is used during recording. Only during replaying is the scheduler replaced. This mechanism preserves the flexibility of μC++.

2.2.3 Real-Time Replaying

Tsai et al [79] present an interesting approach to concurrent replaying, which takes the specific features of a real-time system into account. Since a real-time application is highly time-sensitive, a traditional software approach to replaying is infeasible because it interferes with the execution and therefore alters it.

To provide a non-interfering recording and replaying system for a real-time application, Tsai et al decided to implement a hardware approach. Their system is build for a 68000 architecture. The system records the execution history by monitoring the address, data, and control busses between the central processing unit and the processor memory. The processor state is replicated in a so called DPU (dual processor unit). The data obtained from the various busses are stored in a separate memory unit.

After the recording is finished, the recorded data is analysed and unnecessary events are removed. Two groups of events are significant for the replaying phase: system calls and shared memory access. To recreate the system calls, the mechanism as described for the Repeatable Scheduling Algorithm is used, except a hardware instruction counter is used instead of a software one. The exact position of a signal is recreated during the replaying phase based on the hardware instruction counter by using conditional breakpoints. Shared memory access consists of access to global variables or process synchronisation primitives such as semaphores. During replaying, the access to shared memory is monitored, and compared to the access previously recorded. If the access to shared memory during replaying does not correspond to the one recorded previously, the access is postponed and the process causing the access is blocked.

Tsai et al's approach is hardware dependent. It relies on the existence of a hardware instruction counter. Furthermore, the programming language of the application to be recorded is restricted: It needs to be a block structured language, a block being a function or a procedure. Their approach is designed and implemented for a uniprocessor system only. Because the recording is done on a low level, the amount of data recorded is significant.

2.2.4 Summary

From the presented approaches, some do not apply because they are specifically for a message-passing system. The approaches that are feasible for shared-memory systems mostly are inefficient. They either produce a large amount of data or slow the execution down significantly.

Only the approach by Russinovich and Cogswell seems feasible, since it has low interference on normal execution. The lack of portability for the software instruction counter can be eliminated. I present a platform independent approach in chapter 5. The restriction to uniprocessor platforms is more significant. Arguments considering this restriction are presented in section 4.4.2.

Chapter 3

μC++ and KDB

This thesis is written with a practical purpose in mind. The goal is to improve an existing concurrent programming environment and its debugging tools and provide a useful extension to the debugging tools. In the following two sections, the environments and tools to be enhanced are introduced. Only those details relevant to this thesis are presented.

3.1 μC++

μC++ [11] is an extension of the object-oriented C++ [72, 83] programming language. μC++ was developed by Peter Buhr, Richard Stroobosscher, and Robert Zarnke and is used as a tool to teach advanced control structures with an emphasis on concurrency[1] to computer science students at the University of Waterloo, Canada and development of reliable concurrent programs on microprocessors[2] at the University of Toronto, Canada. It is also used as an experimental platform for scientific research. New and interesting ideas in the field of programming languages are tested by implementing them in μC++.

μC++ is a concurrent language. It provides light-weight concurrency on uniprocessor and parallelism on multiprocessor computers. The library approach is used to provide light-weight threads. Higher level control structures, e.g., coroutines [51], monitors [9, 28, 30], and tasks as well as lower level mechanisms, e.g., spin locks [33] and semaphores [17], are provided as tools to realise concurrent applications.

These additions to C++ are implemented using a translator. A program, containing μC++ language extensions, is read in by the μC++ translator and transformed into C++ statements. The term translator is used instead of preprocessor because the input is parsed and symbol

[1] CS 342: Control Structures

[2] CSC372: Microprocessor Software

tables are built, whereas a preprocessor normally only substitutes strings. The resulting C++ program is compiled using a conventional C++ compiler. The GNU C++ compiler is used since it is freely available and exists on a wide variety of platforms.

To realise user level tasks, μC++ provides a kernel, which controls the scheduling, and hence, the order of execution of the tasks. The kernel resides in a runtime library, which is linked to the object files that are produced by the C++ compiler.

Figure 3.1, taken from [11], shows the runtime structure of a μC++ program. The runtime system is divided into several clusters. The system cluster and user cluster are always present. Programmers can create additional clusters. On each cluster, at least one μC++ processor is shown; on the user cluster, multiple processors exist. μC++ light-weight tasks are executing on each cluster, and coroutines and monitors exist on the user cluster. The terms processor, tasks, cluster, and scheduler are explained and specific μC++ implementation details are presented in the following sections.

3.1.1 Processor

In μC++, the notion of a virtual processor exists. A processor is a mechanism for executing a μC++ task.

μC++ operates in uniprocessor or in multiprocessor mode. A μC++ processor is a simulated processor. The simulation details depend on the mode μC++ runs in. In uniprocessor mode, all μC++ processors are simulated in one UNIX process. In multiprocessor mode, each μC++ processor corresponds to a UNIX process. This structure implies that in uniprocessor mode all the scheduling among μC++ processors is done by the μC++ kernel. In multiprocessor mode, the UNIX scheduling mechanism applies among the used UNIX processes and the μC++ scheduling mechanism is used within a μC++ processor. In both modes, the scheduling of tasks on a processor is controlled by μC++.

3.1.2 Coroutines

A coroutine is an object with its own state of execution. It is possible to suspend the execution in such an object and resume it at a later point. The suspend and resume operations are implemented using context switches.

The concept of a coroutine is implemented as a class. The essential characteristics of a coroutine are given by Marlin in [51]:

1. the values of data local to a coroutine persist between successive occasions on which control enters it (that is, between successive calls), and

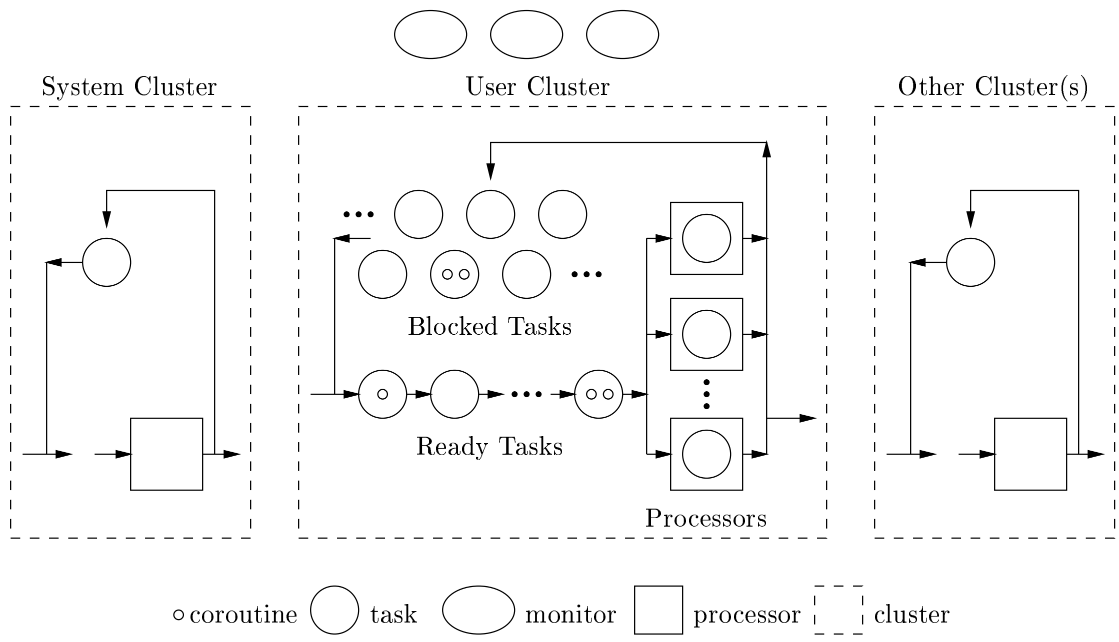

Figure 3.1: Runtime Structure of a μC++ Program

2. the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage.

μC++ provides a coroutine construct following this definition. It provides the language primitives `uSuspend` and `uResume` to suspend and resume the execution of a coroutine, respectively.

3.1.3 Tasks

μC++ implementes user-level threads. A thread is embedded in a task, which is a C++ object. A task has a `main()` method, which is the body of the task. The thread of the task starts execution in this method and terminates when the method returns. As an object, a task has a constructor and a destructor. The constructor is called before the tasks thread starts, the destructor after the thread is terminated. The creation, execution, and destruction of threads is controlled by the μC++ kernel.

A task executes on a μC++ processor and is preemptively time sliced. The time slicing is triggered using signals. If a signal occurs, a check is made to determine if the currently executing task that receives the signal is performing μC++ kernel operations. If this is the case, the signal is postponed, since the μC++ kernel is, in general, not reentrant. A roll-forward mechanism is used to manage this case by simulating the effect of a signal as soon as it is safe to do so. Otherwise, a context switch occurs immediately. Since the signals are triggered depending on a preset timer interrupt, the time slices are non-deterministic.

3.1.4 Cluster

Clusters are provided as a high-level structuring mechanism to bind a group of tasks with a group of processors. Each cluster needs at least one processor to execute its tasks. A cluster has its own scheduler, which determines the order tasks are distributed among its available processors. The scheduler, and with it the scheduling policy, can be defined by the user. Only one scheduler exists for all processors on the cluster. Depending on the scheduler, this construct can provide automatic load balancing among processors. Neither a task nor a processor is bound to one cluster forever. It is possible to migrate either among clusters. For a more detailed discussion of clusters, see Buhr [11, page 9].

3.1.5 Scheduler

A scheduler in μC++ manages a list of tasks. The list is often a queue called a "ready queue". A scheduler must supply a minimum set of methods. One method provides a mechanism to add tasks to the list, another one removes them from the list. Further, a method exists indicating if the list is empty. Finally, two methods are called whenever a new task is added to the cluster and when a task is removed from the cluster, respectively.

Two of the more popular scheduling mechanisms are round robin and priority scheduling. The round robin mechanism iterates over the tasks in a system and gives each one a fixed amount of execution time. For priority scheduling, a priority is assigned to each task. The task with the most significant priority in the ready queue gets scheduled. For an introduction to scheduling and other schemes see [76, pages 61–69].

In μC++, the user may specify a scheduler for a newly created cluster. If no scheduler is specified, μC++ provides a default, which is round robin. The concept of real-time exists in μC++. A cluster on which real-time tasks execute uses a priority scheduler. A task can only be in one scheduling queue at any given time, which guarantees that it is only selected by one processor for execution.

3.1.6 Roll-Forward Mechanism

To ensure mutual exclusion without blocking the signal handler, a roll-forward mechanism [3, 42, 56] is implemented in μC++.

Interrupts, more specifically SIGALRMs [44], are used in μC++ to provide timeslices. Four different cases are considered when a timeslice occurs. The timeslice occurs while

1. executing in user code, interrupts are enabled, and not in a spinlock,
2. executing in user code, interrupts are disabled, and not in a spinlock,
3. executing in library code, interrupts are disabled, and not in a spinlock or
4. executing in library code and either in a spinlock or interrupts are enabled.

The first case is the general case. The code executing just before the interrupt does not require any special consideration. The interrupt does not pose any problems, and therefore, it is possible to block the interrupted thread, and schedule a different one.

In the second and third case, the code executing before the interrupt may not be thread safe, indicated by the fact that interrupts are disabled. Therefore, it could lead to inconsistencies if the executing thread is blocked. Therefore, the blocking is postponed using the roll-forward mechanism. The thread will block and provide a possibility for other threads to run when it enables interrupts again.

In the fourth case, the current execution is not interruptible, e.g., in a system call that is not reentrant. Therefore, the interrupt is pushed forward and execution of the interrupted thread is continued.

3.2 KDB

KDB [35,65] is a concurrent shared-memory debugger developed by Martin Karsten, Jun Shih, and Peter Buhr. It is a tool to debug μC++ programs. KDB stands for “kalli’s debugger”, taking the pattern form GDB.

3.2.1 Features

KDB uses a graphical user interface (GUI) to support the debugging of multiple threads. As mentioned in [27], a GUI is needed to provide adequate usability when debugging multiple threads.

KDB provides the standard features of a debugger. Nexting and stepping is possible. Nexting is realised by setting a temporary breakpoint at the next source line that is executed. Furthermore, if a return instruction is detected in the next source line, a breakpoint is set in the caller’s function after the call. If a branch instruction is detected in the next line, and the target address of the branch instruction is beyond the scope of the current line, a breakpoint is set at the target address. For stepping, an additional breakpoint is set if a call instruction is detected in the current line. The additional breakpoint is set at the target address of the call. After the temporary breakpoints are set, the program execution is continued until one of the temporary breakpoints is hit. Breakpoints are realised in an efficient way, which is explained in section 3.2.3.

Variable lookup and execution of a restricted set of GDB commands is possible. The variable lookup is done in cooperation with GDB and is discussed in more detail in section 3.2.5.

It is possible to attach to a running application. Attaching stops the executing application and provides the debugger with the status of all necessary variables and the program counters of the various threads in the application. The user of the debugger can then examine details of the application and run it in debugging mode as if the application was initially started through the debugger. This feature is useful for applications that seem not to be making progress (deadlock) or spinning (livelock). The program status can be examined and the cause for the deadlock or livelock can be determined.

3.2.2 Partitioned Design

KDB is a partitioned debugger, consisting of a global debugger, which is an application itself, and a local debugger, which is part of the application being debugged. The partitioning is shown in figure 3.2. Traditional debuggers do not have any code in the application itself.

The global debugger provides the GUI and its functionality, and controls the application. The local debugger monitors the application and informs the global debugger of the progress of the application. This partitioned design has several advantages, which are mentioned in the following sections.

3.2.3 Breakpoints

One of the important features of KDB is the implementation of fast and restricted fast conditional breakpoints. A conventional breakpoint implementation uses traps to trigger the debugger whenever a breakpoint is hit. This implementation results in an operating-system level context switch from the program to the debugger, and therefore in a long delay.

Since the application code is shared among threads in μC++, the debugger has to test if a triggered breakpoint applies to a thread. The fast breakpoint implementation inserts code into the executable to perform this check locally rather than have the check performed by the global debugger. Hitting a breakpoint now only results in a function call; the function returns immediately if the breakpoint does not apply.

The breakpoint implementation in μC++ is based on ideas presented by Kessler [36]. The fast breakpoints are more than 2000 times faster than the conventional approach of using a trap and checking the condition in the global debugger.

Conditional breakpoints are implemented by augmenting the inserted breakpoint function being called with a check for a condition. Again, checks are executed by the task to enhance performance. The debugger is only notified if the breakpoint applies to the thread in which it is triggered and the condition is true. Because of the local check, the form of the condition is restricted; only integer and pointer variables or constants can be compared.

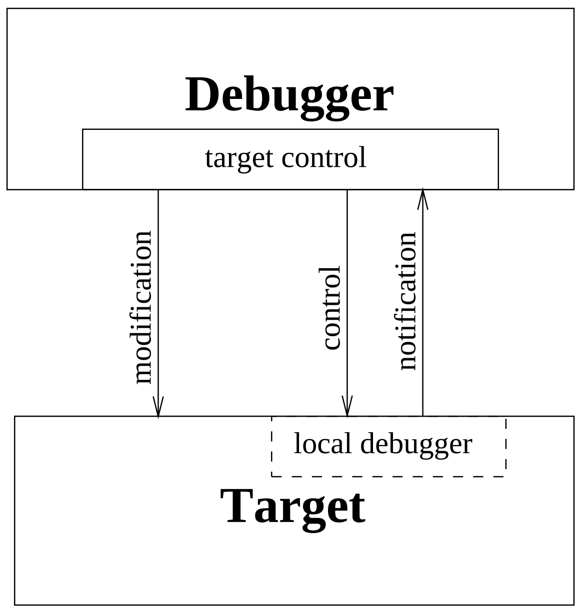

Figure 3.2: Partitioning of Work between Debugger and Target

3.2.4 Communication Mechanisms

Conventional sequential debuggers communicate with the application that is being debugged via synchronous mechanism. An operating system normally provides some mechanism to achieve this capability. For example, a UNIX operating system provides either the `ptrace` system-call [16, 46, 73] or access to the `/proc` file-system [47, 75] or both. Whenever the application is accessed by the debugger, the application stops execution. When the application runs, the debugger usually pauses.

This synchronous mechanism is unacceptable for the light-weight threads in μC++. Debugging a single thread would stop the whole application. Because of that, an asynchronous communication mechanism is used.

A UNIX socket is used for the asynchronous communication channel. The global debugger communicates bidirectionally with a local debugger, which is an independent thread in the application, through the socket. The resulting communication scheme is presented in figure 3.2, taken from [35].

Since an application cannot manipulate its own code, it is necessary to use the synchronous communication channels for setting breakpoints. The synchronous channel is also used for variable lookups. The asynchronous communication channel is used for communicating informations from the local debugger to the global debugger, like hitting a breakpoint, task creation, task destruction, etc. The global debugger communicates operation requests to the local debugger, like setting the condition in a conditional breakpoint.

3.2.5 GNU DeBugger GDB

To look up a variable, it is necessary to know at which position in the data segment of the application it is stored. This information is determined using symbol tables and information in the executable. The access mechanism is split into two parts. The first part deals with the different file formats for executables like `a.out`, `elf`, `coff`, and others. `Libbfd` [13] provides this functionality.

During compilation of an application, debug information can be generated from the symbol table, which is the second part. The format of the symbol table depends on the source language, the compiler and the assembler used. GDB [69] provides access to the symbol table.

Since GDB is available in a source code distribution, it is possible to compile the needed functionality from GDB into a library named `libgdb` and link it to the rest of the KDB source code. GDB is a sequential debugger, which is not thread-safe. Therefore it is necessary for KDB to wrap GDB inside of a monitor.

3.3 Summary

In this section, the tools that are augmented to provide replay are introduced. SMART is used to enhance μC++ with replay capabilities. The concurrent debugger KDB is used to provide a comfortable user interface during replaying and control of the application execution.

The following aspects of μC++ are necessary to understand how SMART works and is implemented:

- The details of virtual processors in μC++ and of the basic implementation of coroutines and tasks,

- the concept of a cluster, a mechanism to bind tasks and processors,
- and the functionality of a scheduler within a cluster.

The following aspects of KDB are necessary to understand how SMART works and is implemented:

- The basic design of KDB,
- details of the breakpoint implementation and the communication mechanisms,
- and the interaction between KDB and GDB.

Chapter 4

Design

This chapter presents the theoretical design goals and issues that lead to the practical design and implementation restrictions. These restrictions force certain constraints on the subsequent implementation.

4.1 Concurrent Errors

To understand how best to debug a concurrent program, it is essential to understand the errors specific to concurrent programs.

4.1.1 Race Condition

A race condition [58] is a time-dependent failure caused by a lack of synchronisation or mutual exclusion. For synchronisation, a race condition occurs if two tasks, which depend on each other, are not properly synchronised. If a task **B** needs to base its execution on results from a task **A**, some synchronisation mechanism needs to be in place. Otherwise, it might happen that **B** executes before **A** provides its results; **B** might base its execution on old or undefined data. For mutual exclusion, two or more tasks enter a critical section at the same time. The result is access to a resource from several tasks simultaneously that should only be accessed synchronously by one task. Corruption of shared data is the eventual, undesired result.

A race condition is an error that is hard to debug, since it is nondeterministic. It might be, that the error only occurs in one out of a thousand executions. The error is subtle, and in many cases does not occur if the timing behaviour of the program is changed by adding print statements or using a debugging tool. A replay facility provides a possibility to eliminate the non-determinism during debugging.

4.1.2 No Progress

The following two error types are different but both result in a lack of progress by the tasks involved. The two cases are illustrated using the Dining Philosopher Problem:

> A number of philosophers, in this case five, sit around a table and eat and think and get hungry in between. The constraint is that they only have five forks, and each one needs two forks at the same time to eat. Therefore, it is impossible for all to eat at the same time. To guarantee that each of the philosophers is able to eat eventually, and therefore, does not starve to death, it is necessary to provide a mechanism that regulates the usage of the forks.

Computer Scientists seem to have a soft spot for philosophers. Much research has been done to solve the stated problem, e.g., by Dijkstra [19], Lynch [49], or Lehmann and Rabin [40].

4.1.2.1 Deadlock

In a deadlock situation, none of the tasks involved is able to make any progress because the resources necessary are used by other tasks. The situation is illustrated in figure 4.1.

In the example, each of the philosophers is hungry and therefore acquires a fork. Now, each philosopher has a fork and waits for its neighbour to give up its fork but no philosopher can acquire a second fork and subsequently eat. Since all philosophers are in the same waiting state, and none is willing to give up its own fork, the system is deadlocked.

A deadlock situation can often be detected using a debugger that can attach to a running program; each task is examined and those in a deadlock can usually be located quickly. Such debuggers are not always available nor does such a debugger explain *how* the tasks entered the deadlock. Attaching and debugging a program in such a situation only reveals that there is a deadlock. Replaying allows a programmer to determine the precise sequence of events that lead to the formation of the deadlock. Only print statements can supply the equivalent information at the cost of potentially changing the execution behaviour.

4.1.2.2 Livelock

In a livelock situation, none of the tasks involved make progress because they are negotiating with each other as to which one is allowed to make progress. In the philosopher example provided in figure 4.2, the neighbours Jill and Jeff are hungry. Jeff already grabbed his left fork, and Jill has her right fork. Jeff now tries to get his right fork, but realises that Jill is

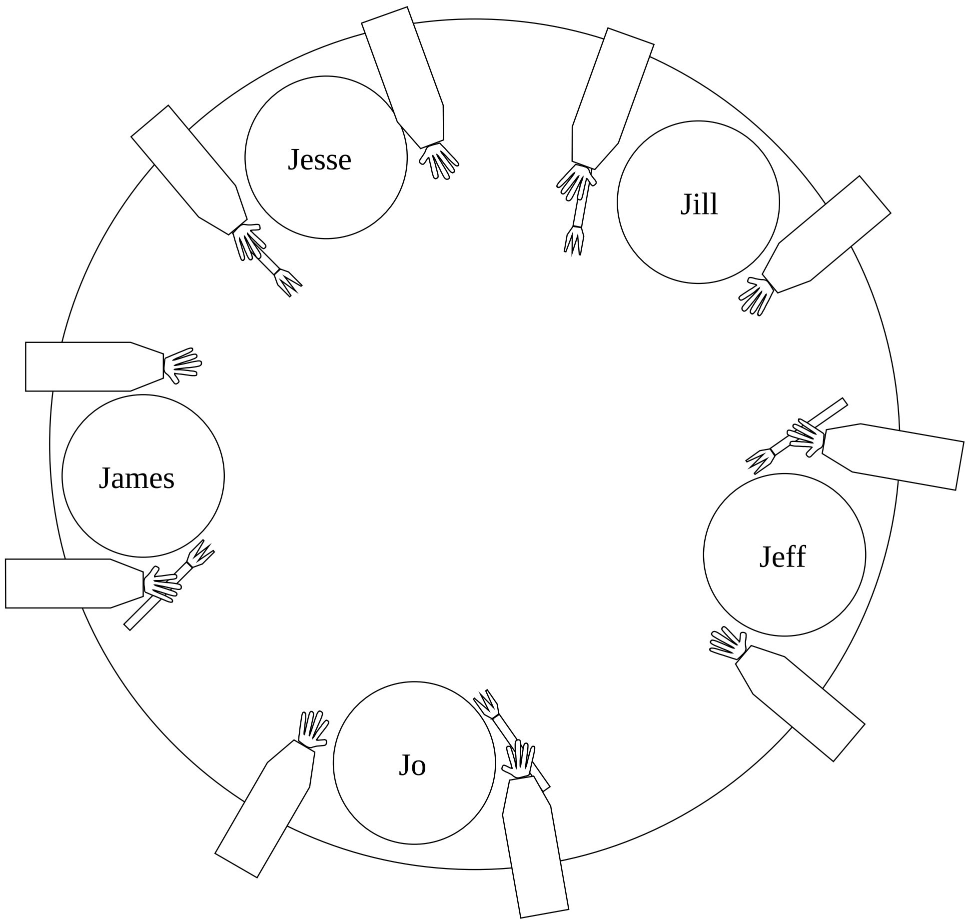

Figure 4.1: Example of a Deadlock

trying to get the same fork. As a gentleman he backs down. But Jill is in the same situation, and she behaves like a gentlewoman, and backs down to give Jeff an opportunity to take the fork. Both realise that both backed down and both try to get the fork again. The polite backing down occurs again and again, forever.

For debugging, the livelock situation is similar to the deadlock one. Most of the time it is also possible to detect a livelock by attaching a debugger to the running program. Occasionally it might be that while the tasks negotiate who is allowed to make progress, they keep on allocating resources, e.g., memory. At some point, no more resources are available and the program crashes. In these cases, the user has to attach the debugger in a reasonable amount of time, e.g., before the resources are all used up. Again, replaying can help to demonstrate the cause for the livelock, not just the presents of a livelock.

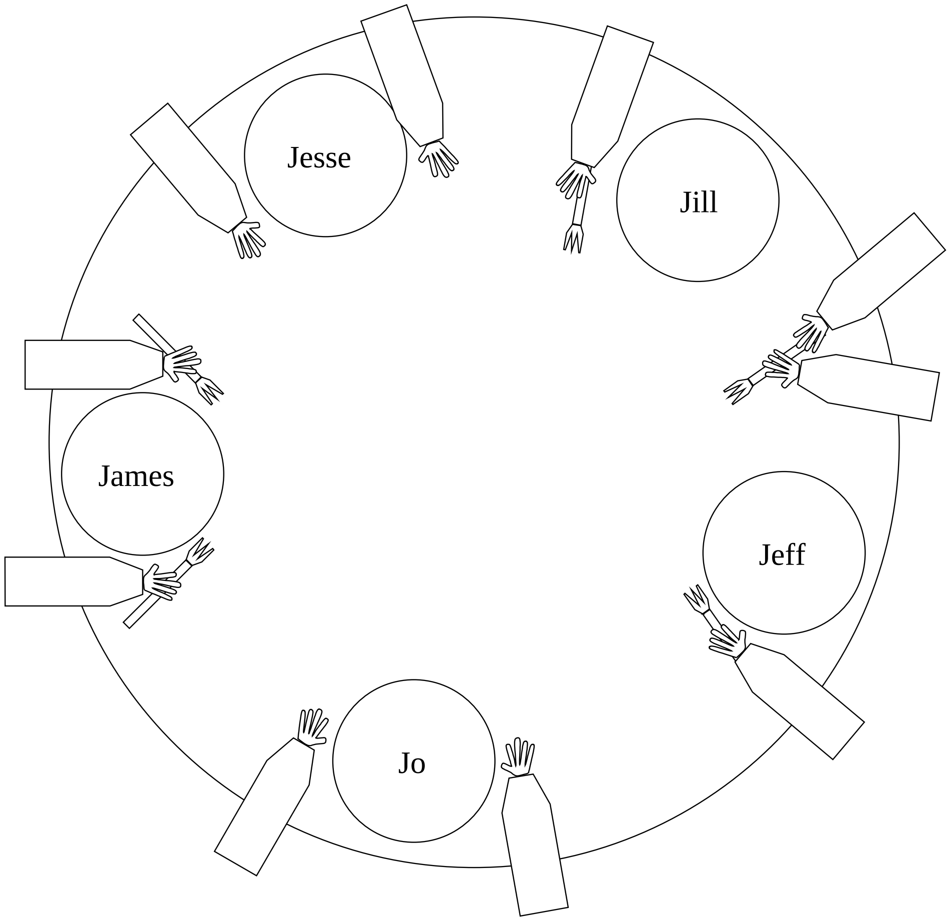

Figure 4.2: Example of a Livelock

4.2 Design Goals

The main goals for the design of the replayer are usability, inter-operability, accuracy, and extendibility.

4.2.1 Usability

A replayer is probably used by a frustrated programmers, desperate for a trace of an error that occurs for no apparent reason in presumably sound code. It should be easy for such a programmer to record and replay a concurrent program. Preferably, no significant additional learning effort should be necessary.

Using a known environment is one of the design objectives. Activating recording and replaying should take no more effort than an additional option on the command line. It should be unnecessary to augment the source code.

Normal execution should be influenced as little as possible by recording activities. Recording should happen transparently. It should be possible to record every execution during a testing phase and to use the recorded data of only a few of these executions. Therefore, recording has to be fast and have a low overhead. The memory usage must be low and a fast memory must be used. This requirement implies that accesses to disk or other slow storage mediums during normal execution is impossible.

4.2.2 Inter-Operability

The replaying facility should be usable with existing tools, like μC++ and KDB. Integrating replay into an existing environment simplifies the usability, making additional learning effort unnecessary for the user.

The tools that are being augmented should not be altered in functionality. Providing an unchanged environment to the programmer is an important design goal. Restricting the functionality of the tools used by a programmer is counterproductive.

4.2.3 Platform Independence

μC++ and KDB exist on various platforms. Therefore, it is necessary to keep the replayer free from platform dependent code.

On some platforms, special debugging support might exist such as a hardware instruction counter. In this case, consideration should be given to using the special support. However, it is necessary to implement similar features for other platforms as generic solutions, even if the cost is higher.

4.2.4 Accuracy

The recorded execution should be replayed with a high accuracy. A possible accuracy metric would be to guarantee that replaying is accurate modulo a function call, a source code statement, an assembler statement or a microprocessor command. In general, it is necessary to provide the highest possible accuracy that can be achieved with software tools. For SMART, the execution needs to be recorded accurately based on the assembler statement being executed and this same granularity needs to be provided during replaying.

Recording should be non-intrusive. Normal execution should not be different, within the limits of the Probe Effect [25], if recording is switched on. No variable alterations and significant timing differences should occur.

4.2.5 Extendibility

μC++ contains real-time functionality. Support for time defined delays and periodic, sporadic and aperiodic tasks exist. The real-time support in μC++ is being extended further to accomplish more sophisticated constructs and provide better usability for the implementation of real-time systems with μC++.

Recording and replaying is potentially a powerful instrument for debugging real-time programs. This possible extension should be kept in mind for the design and implementation of SMART. To facilitate real-time, the following problem is addressed. Since recording has to occur in parts of the kernel, it has to be done in constant time. Otherwise kernel operations might have an incalculable overhead and be infeasible for real-time applications.

4.3 Design Restrictions

The replayer is intended to augment the development toolkit surrounding μC++. Therefore the following restrictions apply:

shared-memory μC++ is a concurrent language for shared-memory systems. Therefore, the replayer has to support shared memory.

platform independent μC++ is supported on a variety of different platforms. The replayer should not establish any restrictions that would prevent μC++ with replaying from being ported to any of the platforms on which it exists today or in the future. Platform dependent code may only be used if an independent, and possibly less efficient, alternative is provided.

language dependent Since μC++ is the language of choice to be augmented, the design can be done specifically for this language and the mechanisms, structures, and constructs used and provided by it, which includes the constructs provided by C++. If specific GNU C++ constructs are used, an alternative for an ISO C++ compatible general compiler needs to be specified, as well.

UNIX operating systems μC++ only exists on operating systems that are derivates of UNIX such as Solaris, Linux, or Irix. Therefore, it can be assumed that the underlying operating system adheres to UNIX standards.

KDB If KDB is augmented with replaying facilities, the replayer should not establish any restrictions on KDB functionality, but provide additional, powerful capability.

4.4 Other Factors

In the following sections, the rationale behind the Repeatable Scheduling Algorithm is explained and an argument is made for restricting it and SMART to uniprocessor platforms.

4.4.1 Cause for Non-Determinism

The cause for non-determinism in concurrent shared-memory programs comes from competing access to shared variables. The example in figure 4.3 demonstrates the phenomenon.

A global integer variable named `shared` is shared by direct reference in `Task`. The program creates two tasks and passes an integer to the constructor. The first task gets the number 1, the second task gets number 2. The constructor of the tasks stores the value in a local variable. In the main routine, the local task specific value is written into the global shared variable.

Several possible scenarios exist. Two possible execution paths are presented in table 4.4 as Alternative **A** on the left and Alternative **B** on the right.

In alternative **A**, `uMain` is started as the first task. Task `one` is created and a context switch occurs. Task `one` is executed next, changing the value of `shared` to 1 and terminating. After the termination, `uMain` is scheduled as the only remaining task in the system at this point. Task `two` is created and executes. It changes the value of `shared` to 2 and terminates. Finally `uMain` terminates. The value of `shared` at the end of the execution is 2.

Alternative **B** presents a different path of execution. `uMain` is started and creates both tasks `one` and `two`. At the end of `uMain`'s main method, it waits for the destruction of the two newly created tasks and blocks. The resulting context switch schedules task `two` as the next task to execute. Task `two` changes `shared` to 2 and terminates. Since `uMain` is still blocked, task `one` is the next task to execute. Task `one` changes `shared` to 1 and terminates. Finally `uMain` terminates. The value of `shared` is 1.

Both alternatives are possible and can be generated by different possible execution sequences. Hence, the final value in the shared variable depends on the order in which the tasks execute. The execution order depends on the scheduling order, which depends on the implementation of the scheduler and subsequent available processor time.

The effect shown is a **race condition**. This effect might be intended, but more often it is caused by lack of mutual exclusion.

```
#include <...>

int shared;

uTask Task {
    int val;
    Task( int val ) : val( val ) {}
    void main() {
        shared = val;
    }
};

void uMain::main() {
    Task one( 1 );
    Task two( 2 );
}
```

Figure 4.3: Non-Determinism

A		B	
Task executing	Value of `shared`	Task executing	Value of `shared`
uMain	uninitialised	uMain	uninitialised
one	1	two	2
main	1	one	1
two	2		

Figure 4.4: Non-Determinism: Alternative A and B

4.4.2 Uniprocessor vs. Multiprocessor

On a multiprocessor platform, the scheduler is not the only mechanism that determines the order of a concurrent execution. No single sequential order of execution exists, because several tasks might execute in parallel. It might be possible that both task `one` and task `two` of the example in figure 4.3 execute simultaneously. Assuming that the process of assigning an integer value is a mutually exclusive operation, a true race between both tasks occurs and the winner manipulates the content of `shared` first, only to get overwritten by the loser, who changes the just written value to its value.

In uniprocessor mode, only one thread executes at any given time, and hence, changes values in the shared-memory. Thus, only the order of scheduling determines the path of execution. Recording this order and replaying it exactly should provide the possibility of recreating an execution.

μC++ is a language that produces parallel and pseudo-parallel code. Every race condition that occurs in parallel execution mode also occurs in pseudo-parallel execution mode, albeit with less frequency of occurrence. Therefore, the restriction to uniprocessor is not that severe.

4.5 Design

The design of SMART is based on the approach by Russinovich and Cogswell. As mentioned in section 2.2.2.3, the Repeatable Scheduling Algorithm recreates the order of scheduling in a uniprocessor system. The algorithm consists of two parts, the recording part and the replay part. During recording, scheduling activities are captured and uniquely recorded by storing the program counter and the value of an instruction counter. The recorded order is recreated during replaying by replacing the scheduler with a replay scheduler. The exact position of a task switch is triggered using breakpoints, which are set based on the recorded PC (program counter) and SIC (software instruction counter) values.

SMART follows this design in both the recording and the replaying phase.

4.5.1 Recording

During recording, the scheduling order of the threads is recorded. The order of scheduling determines the access to the shared-memory and the synchronisation among the various threads. Recording and recreating this exact order is equivalent to reexecuting the application with the same execution path.

For exact replaying, the exact scheduling position has to be recorded. Therefore, the value of the PC has to be recorded. In addition, the value of a SIC needs to be saved if a timeslice is the cause for the scheduling event. If a task is scheduled because of a normal yield of the processor (e.g., blocking on a semaphore), only the scheduling event itself needs to be recorded.

The data of the recording is stored in a file *after* execution for further processing through a replaying mechanism. The format of the file is strictly defined so that the replayer is able to interpret the information correctly.

During recording, the additional processor time and memory space should be kept to a minimum, especially if a time restriction must be honoured. Therefore, the amount of data recorded must be kept to a minimum.

During the developing, testing, and bug fixing phase of the application development process, it should be possible to record every single execution just in case an error happens to occur. Then, it is always possible to replay the last execution accurately to recreate any erroneous situation. The alternative is to rerun[1] the program, hoping for the error to reoccur.

To provide the possibility to record every execution, recording may only have a small disk storage overhead.

4.5.2 Replaying

For replaying, the scheduler is replaced with a special replay scheduler. The replay scheduler determines the recorded order of scheduling events based on the data file created during the recording phase.

An exact reproduction of the timeslices needs to be done. The values of the PC and the SIC determine the exact position at which a timeslice occurred during recording. The task has to block during replaying at exactly the same position, which can be ensured by using conditional breakpoints.

A program is usually only replayed if an error occurred. As such, replaying is part of the interactive debugging process. Therefore, it may be relatively slow.

A tool for the examination of the execution path has to be provided to the programmer via a comfortable interactive debugging environment to allow the programmer to examine data that could lead to the particular execution sequence.

[1] Experience shows that the number of reruns can be a value above 10000.

4.6 General Applicability

The presented approach is applicable for other systems. To apply the approach on which SMART is based, the following components of a system need to be present and alterable:

scheduler All scheduling activities need to be recorded and the scheduler needs to be replaced to recreate them during replaying. If only one scheduler exists, which is true for many uniprocessor systems, this task is easier. As SMART proves, it is possible to use the approach for a system with multiple schedulers.

timeslice A way to record and recreate the timeslices must exist. Therefore, the position at which the timeslices are handled needs to be accessible and timeslice handling needs to be augmented. If a debugger capable of conditional breakpoints is present, that tool can be used to recreate the timeslices.

SIC The SIC needs to be implemented. It is possible to implement the SIC for virtually all procedural and object oriented languages; all constructs which might cause an iteration need to be augmented.

4.7 Summary

This chapter presents the general replay design and some of the circumstances leading to it. Several important design goals are established. High usability of the tool is important. This goal can be met by providing an easy tool that does not require much interaction by the user and runs transparent to the user. Inter-operability with existing applications is important to reduce learning effort. Augmenting existing tools like μC++ and KDB provides a way to meet this goal. Furthermore, platform independence is crucial so that the augmented tools are not restricted in their functionality and availability.

All the design restrictions are enumerated. SMART needs to run in a shared memory environment. It has to be platform independent but may be language dependent. The underlying operating system can be assumed to be a UNIX derivate. Furthermore, KDB is the debugger used and augmented.

Other factors influencing the design are introduced. Causes for non-determinism are analysed, and it has been found to be the scheduling order of the executing tasks.

A discussion of reasons to restrict the implementation to the uniprocessor case is presented. In the multiprocessor environment, scheduling can occur as low as the bus architecture level for simultaneous assignment, making it largely impossible to capture such events without special hardware support. Because it is possible to simulate the multiprocessor case in a uniprocessor environment, it is usually sufficient and not overly severe to restrict the design of SMART to the uniprocessor case.

Chapter 5

Recording

The next two chapters describe the implementation of SMART. Before an execution can be replayed, it has to be recorded. Therefore, the first phase of the implementation is the recording phase.

5.1 Design Considerations

The recording phase must be non-intrusive and passive. The execution path should be minimally altered and side effects should be prevented. The central decision is determining what data to record.

Recording is done without user interaction. The user only switches on recording by compiling with a `-record` flag and calling the application via a shell script named `record`. Everything else is done automatically and transparently with minimal side effects in the execution path.

Figure 5.1 shows the execution path of a concurrent application in uniprocessor mode. The application has three tasks, with execution proceeding horizontally. Time progresses from the top of the figure to the bottom. Task 1 is the first task to be created. The second task waits for the first task to block, since only one task can execute at any given time in a uniprocessor system. After the first task blocks, the second task executes. It is interrupted by a timeslice, which forces it to block. The third task is waiting to be executed, and does so after the second task gives up the processor. At this point, all three tasks exist, task three is executing, and the first two task are blocked. Whenever a task blocks and provides the possibility for a subsequent task to execute, a context switch occurs. The context switches are shown as the curly lines in the figure.

The third task blocks due to a yield. The yield might result because the third task waits for I/O to occur or the programmer explicitly specified that the third task is supposed to yield the processor at this point. Another possibility is that the third task yields because it needs to wait for another task to continue execution, e.g., task one in the example.

After the third task blocks, the first task is scheduled again. It blocks at some point due to a yield and provides the second task the possibility to execute. The second task finishes its execution and terminates, freeing the processor for another task to execute. The first task executes again, being interrupted by a timeslice, which forces the first task to block. The third task is scheduled, and executes for a while before terminating. The last task remaining in the system is the first task, which executes until termination.

The determination of which task is to be executed next is done by a scheduler. Different scheduling mechanisms are mentioned in section 3.1.5.

5.2 Task Creation and Destruction

Task scheduling activities have to be recorded, which includes creation and destruction as the initial and final scheduling of a task. It is reasonable to record the name of the task and the address as a unique identifier during creation. For all subsequent recording, the name can be omitted since the address is unique to the task and it is possible to extract the name of the task based on its address if needed.

The task creation can be recorded upon construction and initialisation of the task, the task destruction can be recorded in the destructor of the task. The recording may not occur earlier than the constructor or later than the destructor since the unique identifier of the task is its address, which does not exist beyond these two points.

5.3 Blocking

A task blocks for two different reasons. The first reason is direct yielding of the processor or a command that results in an indirect yield. A yield forces the task to block in a deterministic manner — it blocks at exactly the same position in all runs.

When the time quantum of a task has expired, the scheduler timeslices the task to provide processor time to other tasks, which is the second reason for blocking. In a subsequent run, the timeslice might occur at a different point of execution or not at all because the timing is different.

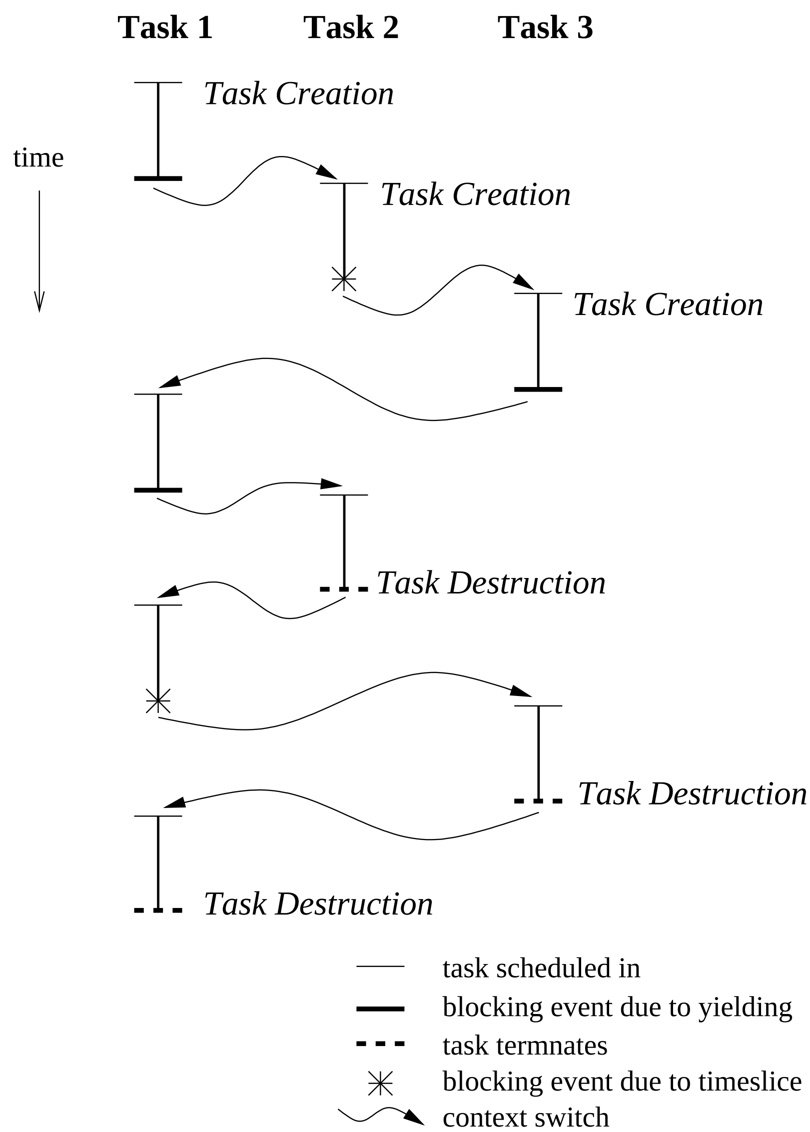

Figure 5.1: Execution of Three Threads

5.3.1 Yielding

The first kind of blocking is deterministic, and thus occurs in subsequent runs at the same position. The point of execution does not have to be recorded, as neither the PC value nor the SIC value is needed. It is not even relevant which task is being blocked because it happens deterministically. It is more significant which task is scheduled next since the scheduler itself might be non-deterministic. Therefore, each scheduled task is recorded with its unique identifier address.

5.3.2 Timeslicing

The timeslices have to be recorded, too. Of interest, besides the unique identifier of the task, is the current position of execution. The current position is determined by the content of the program counter register. However, the PC value alone is insufficient. To illustrate this, consider the short program shown in figure 5.2. If a timeslice occurs during the execution of the `cout` statement, the position can be identified as being in line six. However, this position identification does not uniquely determine the position of the execution. Line six is executed a total of ten times, and all of these ten times the identifier would be six.

While the illustration is based on the source code level, the same argument applies for assembler code. The PC value is insufficient to uniquely identify a position in an execution path of a program. An additional counter is needed to identify repeated execution.

```
  #include <iostream>
  int main() {
      int i;
      for( i = 0; i < 10; i++ ) {
5         cout << i << endl;
      } // for
  }
```

Figure 5.2: Position based on PC value - C++ program

One potential counter is an instruction counter, counting each executed instruction. The counter is incremented after every assembler instruction. This capability could easily be done in hardware by incrementing a dedicated register every time a new instruction is executed by the central processing unit. However, hardware support does not exist in the form of an instruction counter on any of the major hardware platforms [31,32,34,66][1]. Thus, the counter

[1]The Mips R10000 [29] chip is an exception to this rule. It provides a "Performance Counter", which can be customised to count instructions.

has to be implemented in software. Mellor-Crummey and LeBlanc suggest an implementation of a software instruction counter at the assembler level [54]. Since the recording should work on a diverse set of platforms, such a highly platform dependent solution is not the best option. If the programming language is specified, a specific solution for this language can be implemented. I will present such a solution in section 5.5.1.

Based on the above discussion, it is necessary to record the address of the task as a unique identifier for each timeslice, and the PC and SIC. It is possible to replay a timeslice based on these two values. The PC is used for setting a breakpoint at the address specified by it. The breakpoint is to be triggered whenever the SIC has the recorded value. This makes the breakpoint a conditional breakpoint.

μC++ uses the roll-forward mechanism as discussed in section 3.1.6 to guarantee mutual exclusion but still perform the signal handler, albeit slightly delayed. To provide sufficient information to recreate the actions taken after a timeslice occurs, it is necessary to record an indicator showing under which circumstance the timeslice occurred. This indicator is a simple integer variable with values ranging from one to three, corresponding to the first three cases presented in section 3.1.6.

5.4 System Tasks

From the user's perspective, the approach as described so far is sufficient. All relevant tasks are being recorded transparently. Nevertheless, the approach lacks some details that depend on the specific implementation of the thread support in the runtime environment.

As mentioned in chapter 3, the goal of this thesis is to augment μC++ and deal with any peculiarities associated with it. μC++ differentiates between system tasks, used solely for administration work by the runtime environment, and user tasks. A user should only see the user tasks and normally does not have to care about system tasks. Nevertheless, the scheduler has to deal with the system tasks. Interestingly, the system tasks do not access the shared-memory areas that the user accesses, and therefore they are irrelevant for the recording and are filtered to further reduce the recorded data.

An easy way to differentiate system tasks from user tasks is at the point when a task is created. Before a μC++ program is started, some global constructors are called. In these constructors, some system tasks are created. For the sake of this implementation, all tasks that are created before the construction of the recording object are considered to be system tasks. The recording object is a global object in the μC++ kernel. It is constructed as one of the last objects in the kernel.

The recorder registers every task that is created after its own creation, and maintains a list of the tasks that are registered. The recorder is called for all scheduling activities that occur. The recorder determines, according to an internal list, if the task that caused the scheduling activity is a user task and needs to be recorded. If it is a system task, it is ignored with respect to recording but scheduled normally.

5.5 Implementation

Two independent aspects of recording have to be implemented, the SIC and the recorder itself. The code to support the software instruction counter is added to the μC++ translator, the recording object is added to the μC++ kernel.

5.5.1 Software Instruction Counter

Figure 5.3 shows the steps from the source code to the executable file for a μC++ program. The first step is the C++ preprocessor. The resulting output of the preprocessor is passed to the μC++ translator, which parses the file and replaces μC++ primitives with calls to the appropriate functions in the μC++ libraries. The result is standard C++ code, which is compiled and linked to build an executable. After the μC++ translator, the steps are no different from an ordinary C++ program.

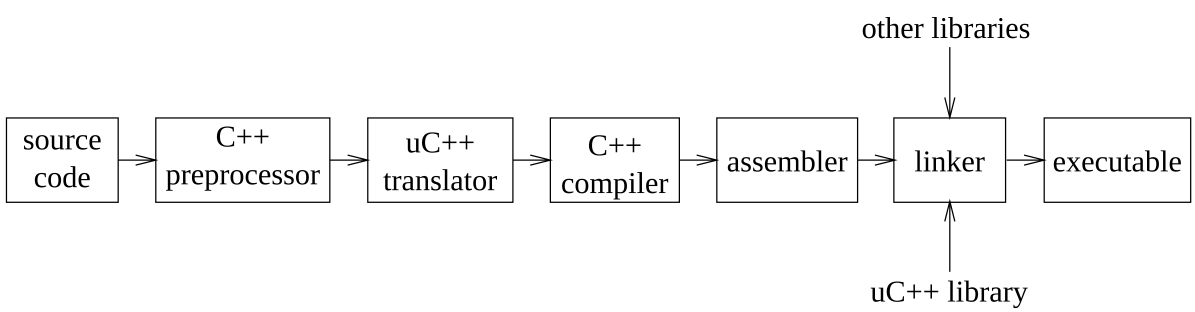

Figure 5.3: Generating a μC++ Program

Since the μC++ translator analyses and parses the source file, it is a good candidate for inserting code to increment the SIC. Therefore, the translator is augmented with a command line option `-record`, which switches on insertion of SIC code.

The simplest way to implement a SIC is to add an increment command of a counter after every line of source code. The drawback is that this would roughly double the execution time, which is unacceptable. The minimal necessary incrementing has to be found.

Remember, the purpose of the SIC is to uniquely identify a position in the source code, including the number of iterations necessary to reach this point. During each and every iteration, the SIC has to be incremented at least once. It is possible to generate jumps to previous addresses (iteration) in several different ways in C++. The obvious ones are `for`, `while`, and `do` loops. If such a loop does not contain any unstructured back jumps, only one increment is necessary. Examples for the loop construct and an arrow marking the back jump are shown in figures 5.4(a), 5.4(b), and 5.4(c). It is sufficient to augment the body of the loop with an increment statement as shown by the line `SIC += 1;` in the figure. This technique also handles the use of `break` and `continue` in a C++ loop.

```
int j = 0;
for ( int i = 0; i < 10; i++ ) {
    SIC += 1;
    j += 10;
}
```

(a) For Loop

```
int j = 0;
do {
    SIC += 1;
    j += 10;
} while ( j < 100 );
```

(b) Do Loop

```
int j = 0;
while ( j < 100 ) {
    SIC += 1;
    j += 10;
}
```

(c) While Loop

```
int j = 0;
startinc:
SIC += 1;
j += 10;
goto startinc;
```

(d) Goto

```
int j = 0;
f() {
    SIC += 1;
    j += 10;
    if ( j < 100 ) {
        f();
    }
}
```

(e) Recursive Function Call

Figure 5.4: Back Jumps in a C++ Program

Another possibility for a back jump in a C++ program is the `goto`[2] statement. An example is shown in figure 5.4(d). For loop constructs, a SIC incrementing command is inserted at the beginning of the appropriate block. (If the block only consists of a single statement, i.e., no braces exist, the braces are inserted.) A `goto` command does not start a new block. Therefore, the increment of the SIC is done after the label to which execution jumps due to the `goto` command. This approach may lead to unnecessary increments, specifically if the `goto` transfers in the forward direction. Only back jumps are an issue, but the μC++ trans-

[2]The opinion about the `goto` statement are diverse. As example for the two extremes, two quotes from famous computer scientists: "You can't live without GOTOs. Death. Taxes. GOTOs." [7], "the go to statement should be abolished from all 'higher level' programming languages" [18]

lator does not provide sufficient information to differentiate. The unnecessary increments only affect overhead introduced to the program, not the correctness and uniqueness of the recorded SIC. Finally, there are few `goto`s in most C++ programs.

The last possibility to generate iterations is by using function calls. Figure 5.4(e) shows a recursive function, which calls itself. Calling a function several times is also considered to be a form of iteration. If the code of the function is not inlined, the PC value does not indicate which of the calls is being executed. To uniquely identify an execution position in a function, the SIC is incremented at the start of the body of every function. In the case of an inlined function, such an increment is unnecessary and introduces additional overhead. Unfortunately, it is impossible to determine which functions are inlined during preprocessing of the source code. Even if a function is defined as inlined by the programmer the compiler does not necessarily follow this suggestion, and might not inline a specified function. On the other hand, the compiler might consider a function appropriate for inlining even if the programmer did not suggest it to be inlined. Adding an increment command to every function body again introduces additional overhead, but does not influence the uniqueness of the SIC.

The method to add the SIC to the appropriate position in the source code is shown in figure 5.5. It is called while parsing a label, the `while`, `do`, or `for` primitive and a body of a method if the `-record` flag is set.

```
void gen_addsic( token_t * before ) {
    gen_code( before, " uKernelModule :: uActiveTask -> uSIC += 1 ;" );
} // gen_verify
```

Figure 5.5: Inserting the SIC

The variable `uKernelModule::uActiveTask` is a pointer to the task that is active and is managed by the μC++ runtime system. It is usually obtained by calling the function `uThisTask()`. However, eliminating the function call gives a speed up of about 50 percent, based on measurements on a Linux Pentium 133 PC. The details of these measurements are given in section A.1.

Here are two properties of the SIC that can be varied in the implementation: SIC size and location. The SIC size indicates the maximum value of the SIC before it overflows. The SIC location indicates where the SIC variable is stored in the application. Both of these issues are discussed in detail.

5.5.1.1 SIC Overflow

In a sufficiently long running application, it is possible for the SIC to overflow and generate non-unique values for recording.

The SIC is an integer value. Since a lot of iterations might occur, the value should be in an adequately large variable. An `unsigned int` has been chosen, which has a maximum of 4,294,967,295 [24, 74] on any four byte integer computer, e.g., Intel i586 or Sun Sparc machines. Interestingly, overflows are not handled, for the following reasons.

It does not make too much sense to handle a potential overflow by increasing the size of the variable or by setting a flag since this only changes the magnitude of the problem, not the problem itself. It is hoped that the chosen storage is sufficiently large to prevent overflows in most reasonable cases. Furthermore, overflow handling would introduce additional overhead for the incrementing of the SIC, which slows down the program. Since the SIC is already generating a decrease in runtime of about 10 percent, additional delays should be prevented.

The following argument shows that the overflow issue can be ignored. To be more precise, it is not the overflow that is a problem, but an ambiguous SIC value within a timeslice. A timeslice is recorded with a specific SIC value so that it can be recreated by setting a conditional breakpoint with the SIC value as the condition during replaying. As long as this SIC value is unique for the specific timeslice, overflows are not a concern. An example execution in figure 5.6 explains why many overflows can be ignored. Three tasks are shown. The execution segments of the tasks are lettered A to H. Next to each execution segment, the SIC value upon scheduling and blocking is noted. For this example, it is assumed that the maximum SIC value is 128.

Two SIC overflows occur in the example, one in segment G of task one, the other one in segment F of task two. Since no timeslices are recorded after the overflow in segment G, because task 1 terminates, no further complications may occur. The overflow in segment F is more worrisome. The first timeslice for task two occurs at a SIC value of 96. The value is unique, it has not occurred before. The second timeslice occurs at SIC value 21. The value occurs earlier in the execution in segment B. Nevertheless, the SIC has not been 21 since the last timeslice, which occurs at SIC value 96. Therefore, the value for the second timeslice is unique for this specific timeslice.

As argued and seen in the example, it is important that between timeslices no SIC value occurs twice. The default value for a timeslice in μC++ is approximately 0.1 to 0.01 seconds, depending on the execution environment. A simple experiment shows that it takes about three minutes and 15 seconds on a Pentium 133 to get an overflow if an unsigned integer is just incremented. (The details of these experiments are shown in section A.2.) In general,

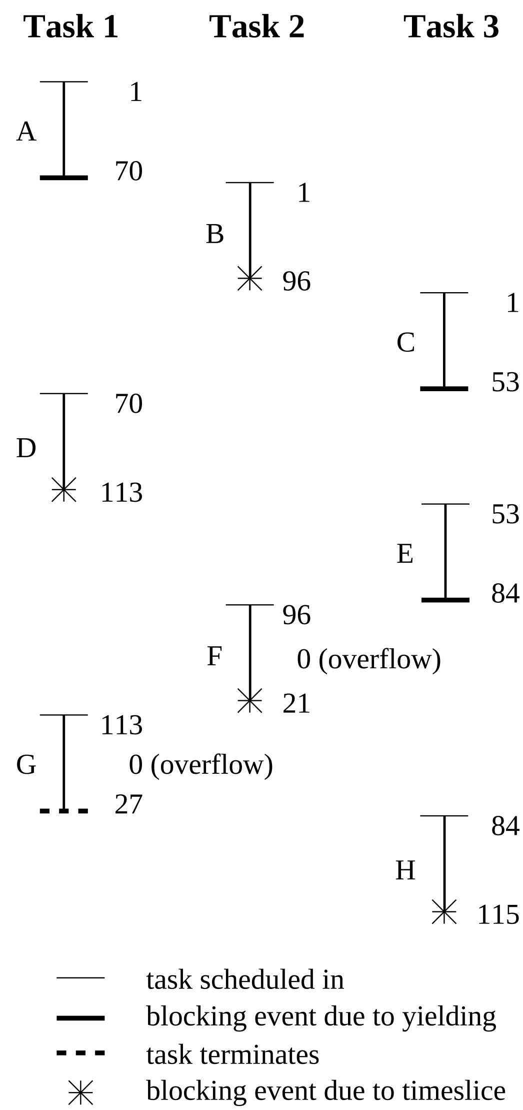

Figure 5.6: SIC Overflow Example

the incrementing of the SIC is embedded in many other lines of user code, so the SIC is incremented much slower than this upper bound. Unless the default value for a timeslice is changed by several magnitudes, at least on a Pentium 133, no significant overflow and repetition of a SIC can occur between timeslices.

5.5.1.2 SIC Location

The SIC can be stored in different places. For example, it is possible to use one global variable or a local variable per task.

Sharing the SIC as a global variable among tasks is a valid solution, because SMART only operates in uniprocessor mode and integer assignments are atomic on the used platforms.

Therefore, the SIC always yields a unique value for each iteration in each task. However, a global SIC value increases faster, and hence, increases the potential for overflow. One major problem exists though: increment of a shared variable is not atomic on most modern RISC computers. For example, the following steps are normally required:

1. Fetch the variable form shared memory into a register,
2. increment the register contents,
3. and store the variable in shared memory.

Multiple tasks can be executing this sequence concurrently resulting in inconsistent incrementing of the global SIC.

Mellor-Crummey and LeBlanc suggest using a register for the storage of the SIC so the increment is atomic. The GNU compiler gcc [67] and egcs [68], the "Experimental GNU compiler system", the newest generation of the GNU compiler system [6], support such a feature by associating a variable with a register. Since the SIC is accessed often, this method would provide a significant increase in speed.

However, assigning a register to a variable is also a bit dangerous. It has to be guaranteed that the specific register is not used for anything else. This requirement implies that all used libraries have to be checked for potential usage of the register and that optimisation by the compiler may be impeded due to the loss of the register for normal computation.

Another disadvantage is that using a register is a highly platform dependent solution. For each used platform, an eligible register, preferably with the same size as on other platforms, has to be found.

Another reason to make the SIC local, rather than global, is that it is not affected by the additional tasks during replay. If the SIC is global, the additional system tasks would have to be identified and no SIC inserted.

Because of the disadvantages of a global SIC, SMART uses a SIC stored on a per task basis as a local variable. This solution provides less interdependencies among tasks. Also, the SIC does not overflow as often as if stored globally, and it does not matter that the access to the SIC is not atomic because the SIC is no longer shared.

5.5.1.3 Optimisation and the SIC

All modern compilers can perform multiple levels of optimisation during compilation, such as dead code removal, code reordering, storing variables in registers, etc. Debugging optimised applications is problematic, since the resulting assembler code from the compiler does not necessarily map to the source code anymore. These transformations result in confusing behaviour during an interactive debugging session. For gcc, GDB only handles some optimisations and disables others.

Replaying an optimised μC++ program might fail since the optimisation could eliminate or significantly change the behaviour of the source level SIC. For example, in:

```
for ( int i = 0; i < 10; i ++) {
    SIC += 1;
    ...
}
```

the compiler could hoist the incrementing of the SIC outside the loop and simply add 10 to it. This optimisation is possible because there are no references to the SIC in the loop body. In general, such aggressive optimisation is not handled by most debuggers and would be turned off when the application is compiled with debugging support.

5.5.2 Recording

Since recording is triggered from inside the kernel during scheduling, only elemental functions may be used. Blocking operations are not allowed during the actual recording, since the scheduler may not be blocked. Therefore, recording is implemented with low overhead and no blocking calls.

During the startup phase of the kernel, a recording object is created if the recording is switched on by calling the executable using the `record` shell script. The object is a normal C++ object. It provides methods that are called to add the specific events to the list of recordings and to initialise and save the list. The class definition is shown in figure 5.7.

Unfortunately, certain crucial information needed by the replay object is only available after the global `main` routine is called implicitly at application startup. This information is the shell arguments passed to the application from the command line. Once this information is available in `main`, a call is inserted to the `specifyFileName()` method of the recorder object. The name of the executable, which in C++ is the first (implicit) command line option, and the name of the file in which the recorded data should be stored is passed to the recorder object and stored. Also, the date and time at which the execution occurs is stored. These data are used during replaying to verify that the recorded data is generated from the executable.

```
class uRecorder {
    // Protect the list
    uSpinLock uRecorderLock;

    // name of the file to hold recording information
    string recordFileName;

    // recorded data
    list< string > recordList;

    // list of tasks to be recorded
    set< unsigned int > taskList;

    // switch recording on / off (do not record tasks created before uMain)
    bool record;

  public:
    static uRecorder *uRecorderInstance;

    uRecorder();
    ~uRecorder();
    void specifyFileNames( char *exe_name, char *record_name);
    void scheduledTask( uBaseTask &t );
    void addNewTask( uBaseTask &t );
    void taskFinished( uBaseTask &t );
    void cxtSwt( uBaseTask &t, int action, void *sp, void *pc );
    void saveInfo();
    void addToList( const string &name );
}; // uRecorder
```

Figure 5.7: Class Definition of the recorder

5.5.2.1 Re-Entrant Problem

If a timeslice occurs during calls to the recording object, the order of the recorded data will be incorrect. Imagine the following situation: while recording a scheduling event `A`, a timeslice occurs and causes the scheduling event `B`. Scheduling event `A` clearly occurred before scheduling event `B`. Nevertheless, the recording of `A` was interrupted and the event `B` is stored before `A`.

This problem is called the re-entrant problem [10, page 27]:

> A routine ... is said to be non-re-entrant when its correctness relies on its uninterrupted execution and yet this uninterrupted execution is not guaranteed by its implementation.

It is necessary to protect the recording and prevent the above mentioned situation from happening. Therefore, interrupts are disabled upon entering a recording object method, and enabled again upon leaving it.

5.5.2.2 Memory Management

It is infeasible to write the events to disk directly, since this might potentially cause blocking. Therefore, all events are buffered in a list. Adding elements to the end of a list is an operation that can be done in constant time. Therefore, the time overhead is constant and minimal, independent of the size of the list or the size of the data to be added. To prevent delays during execution due to memory allocation, a large chunk of memory is reserved on startup to provide space for the list entries. If all the reserved memory is filled, recording is stopped and a message is printed upon termination of the application. The size of the memory is configurable by specifying the `-m` option to the `record` shell script.

Before the application terminates, the `saveInfo()` method of the record object is called and the list is written to a file. The filename is either a default value depending on the name of the executable or a name provided by the user.

5.5.3 Format of Recorded Data

The data that is recorded and saved by the `saveInfo()` method is ASCII text. Different events are recorded with different data:

The creation of a new task is recorded in the format:

```
T < task name > < task id > < cluster id >
```

with the task name as a string without spaces and the task id and cluster id as an integer.

The creation of a new cluster is recorded in the format:

```
C < cluster name > < cluster id >
```

with the cluster name as a string and the cluster id as an integer.

A scheduling event due to a yield is recorded as:

```
S < task id > < cluster id >
```

with the task id and cluster id as an integer.

A scheduling event due to a timeslice is recorded as:

```
W < task id > < action > < SIC > < PC >
```

with all parameters saved as integers.

Furthermore, the name of the executable and the time of the recording are stored in the format:

```
Executable: < name of executable > Time: < time of recording >
```

with the name of the executable being a string and the time stored in the UNIX integer time format [48].

Finally, the termination of a task is recorded as:

```
TaskFinished: < task id >
```

with the task id being an integer.

5.6 Summary

This chapter presents the recording part of SMART. The first part considers design aspects and determines which events and activities have to be recorded. Task creation and destruction have to be recorded as the end points for subsequent scheduling activities. Blocking is considered, as it leads to scheduling activities. Different types of blocking are determined: blocking caused by yielding and timeslices. For blocking caused by yielding, only the task scheduled after the blocking needs to be recorded. To provide sufficient data for the replay of timeslices, it is necessary to record the PC and SIC value. Recording system tasks is unnecessary as they do not affect replay, which reduces the amount of data.

The implementation consists of two parts. The first is the implementation of the SIC and the second is the recording of the data itself. Recording of the SIC can be optimised as only looping needs to be recorded. The sources for looping are: the loop constructs, recursive function calls and goto statements, are identified and accordingly augmented.

For recording, a new object is created, which stores the data in a temporary list and saves the list at the end of the application. This approach reduces the overhead during execution.

Chapter 6

Replaying

Replaying provides a way of using the cyclical debugging approach with concurrent programs. The second phase of cyclical debugging consists of further examination of the executable. A debugger provides functionality to control and inspect the execution path. Therefore, the replaying side of SMART is integrated into KDB. KDB provides a graphical debugging environment, which gives the user a certain amount of comfort and ease of use to accomplish the debugging task.

The main goal to be achieved during replaying is the accurate recreation of the recorded execution path with a sufficiently fine granularity. Therefore, all context switches have to be reproduced exactly as they happened in the previously recorded run. Since context switches result in scheduling activities, it is equivalent to reproduce these activities.

As determined in the previous chapter, two cases for blocking exist: blocking due to yielding and timeslices. The blocking due to yielding occurs deterministically and therefore does not need to be recreated. Timeslices are recreated, as well as the scheduling of a task.

6.1 Hybrid Approach

The presented approach and implementation of SMART utilises features of the local debugger and the global debugger. Furthermore, it augments the μC++ kernel. KDB is used to replay timeslices, parts of μC++ are augmented to provide replaying of scheduling events. This hybrid approach is not just a simple way to implement SMART, it is also efficient and necessary.

The chosen distribution of work simplifies the implementation, since existing features are reused. Conditional breakpoints, which are used to simulate timeslices, are not reinvented but rather the existing ones from KDB are used and augmented as necessary. Furthermore, it is unnecessary to reinvent a graphical debugging environment, since KDB provides that.

It is impossible for an application in UNIX to change its own executable code. Therefore, it is necessary to have an external application, which provides a facility to replay timeslices. KDB is used as this external application.

Replaying each scheduling event via breakpoints, including the ones caused by normal blocking, is possible, but highly inefficient. It is much faster to replace the scheduler in the application with a replay scheduler. Conditional breakpoints introduce overhead, which is eliminated for normal blockings by using a replay scheduler.

6.2 Preparations

Before replaying can start, some preparations have to be done. First of all, it is necessary to enable replaying, which is done via a command line option. During startup of the application itself and of KDB, it is necessary to prepare the recorded data and store them in an adequate data structure.

6.2.1 Starting Replaying

KDB is started via two scripts. The procedure is shown in figure 6.1. The first script, named `kdb`, starts a script named `kdb_target` and the debugger itself, which resides in the executable `kdb_server`. `kdb_target` subsequently starts the application to be debugged.

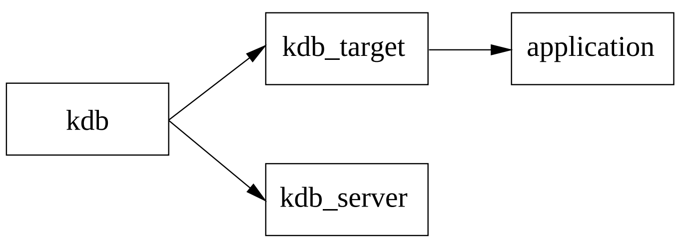

Figure 6.1: Starting up KDB

To enable replaying, a new command line option, `-replay [filename]`, is added to `kdb`. The filename is the name of the file that contains the recording data. If the filename is omitted, a default value is constructed by the script by augmenting the filename of the application with `.rec`. `kdb` passes the replay parameter as a command line option to `kdb_server`, and sets an environment variable named `U_REPLAY` with the name of the file containing the recorded data.

Before execution of the `main()` routine of the application being replayed, the constructors for the global variables are called. The replayer object is one of the global variables. It checks for the environment variable `U_REPLAY`, and if present, reads the file name. If the variable is not present, replaying is not activated.

The file with the recorded data is completely read and some data structures are built, which are explained in more detail in section 6.2.2.

The data file contains a line with the time of creation of the file and the name of the executable from which it was created. If the name in the data file and the name of the executable reading the data file are not the same, the execution is aborted. The execution is also aborted if the timestamp of the executable running is younger than the timestamp recorded in the data file.

As mentioned, the name of the file cannot be checked before the execution of the `main()` function of the application to be replayed, since it is necessary to retrieve the name as an option implicitly provided on the command line. Therefore, the check is not done during the creation of the replay object but later during execution of `main()`.

6.2.2 Preparing Recorded Data

Before the program can be replayed, the recorded data has to be analysed and prepared. Since the kernel and the global debugger both participate in the replaying activities, both sides read the recorded file.

In the following sections, several example tables are shown. The data in the tables was obtained by running the test program presented in appendix B.

6.2.2.1 Global Debugger

The global debugger reads the complete file in and builds two tables out of it. The first table is a translation table of task identifiers, the second table contains the timeslicing events.

Task Name	recorded ID	ID during replay
uMain	0x161f20	0x17cb88
CountTask	0x16a720	0x187588
CountTask	0x172f20	0x18fd88

Table 6.1: Example Translation Table for Tasks

Table 6.1 shows an example of a complete translation table. Each row in the table represents one recorded task. The table consists of three columns. The first column contains the task name, the second column the address of the task during recording and the third column contains the address during replaying.

The task name may not be a unique identifier of the specific task; it might only be the name of the class from which the task is an instantiation. In the example shown, two tasks have the same name, `CountTask`. Therefore, it is necessary to record the address as a unique identifier.

However, the address of a task changes from recording to replaying, because different objects in μC++ are created during replaying. As well, a recording run is normally done with the local debugger switched off. The local debugger is active during replaying. The presence of these additional objects during replay forces application objects to appear at different addresses.

During the preparation phase, only the first and second column of the translation table can be filled. The third column is built during execution. Whenever a new task is created, the local debugger informs the global debugger and registers it. Part of the registration of the task with the global debugger is checking if it was recorded, i.e., if it is a user task, and augmenting the translation table with the new address.

The global debugger needs to recreate the recorded timeslices. Therefore, it builds a table with the recorded timeslicing information. The rows contain the unique identifier and the PC and SIC values. The order in which the timeslices were recorded is relevant only on a task basis. Therefore, it is possible to construct a list for each task individually. Such a construct simplifies the retrieval of the next timeslice that needs to be replayed for a given task. The resulting data structure is shown in table 6.2.

Task ID	PC	SIC
0x161f20	0xc81ec	333
0x16a720	0xda4c4	299139
	0xda460	472978
0x172f20	0xda428	172305
	0xda428	347317
	0xda460	487552

Table 6.2: Example of a Timeslice Data Structure

6.2.2.2 Replayer Object

The replayer object, which is part of the application being replayed, also prepares the recorded data for easier handling during the actual replaying phase. It is also necessary to build a translation table for the task addresses as in table 6.1. The third column is filled in when a new task is created.

Cluster Name	recorded ID	ID during replay
uSystemCluster	0x11ccd8	0x11ccd8
uUserCluster	0x14ddd8	0x14ddd8
joe	0x16b870	0x186090

Table 6.3: Example Translation Table for Clusters

In addition to the task translation table, a cluster translation table needs to be built. An example is shown in table 6.3. It is similar to the task translation table: each row contains the data for one cluster, the first column contains the cluster name, the second the recorded address of the cluster and the third is filled with the address of the cluster during replaying. In general, the cluster translation table contains significantly fewer entries than the task translation table, since fewer clusters exist in a normal μC++ application than tasks.

The shown example is not the best one, since the address of the `uUserCluster` and `uSystemCluster` does not change from recording to replaying. Theses two clusters are created early in the execution, and therefore are not affected by the additional replay objects. However, the address of the cluster `joe` does change. This cluster is created for no other purpose than to demonstrate the address shift.

Task ID	Cluster ID
0x161f20	0x14ddd8
0x16a720	0x14ddd8
0x172f20	0x14ddd8
0x16a720	0x14ddd8
0x172f20	0x14ddd8
0x16a720	0x14ddd8
0x172f20	0x14ddd8
0x161f20	0x14ddd8
0x161f20	0x14ddd8
0x161f20	0x14ddd8
0x161f20	0x14ddd8
0x161f20	0x14ddd8

Table 6.4: Example Table of Scheduling Events

The replayer object contains the data needed to schedule the tasks of the application being replayed in the order in which they were recorded. These data are held in a table that is built based on the recorded data. An example of such a table is shown in table 6.4. Each row contains one scheduling activity. The first column of the table contains the recorded address of the task to be scheduled. The second column contains the address of the cluster on which the task executed during the recording phase. It is important that the order of the scheduling events is maintained.

6.3 Scheduling

The scheduling events are replayed by replacing the schedulers on every clusters with a replay scheduler. The default behaviour of the replay scheduler imitates the μC++ default scheduler. Therefore, it does not matter if the default scheduler is replaced on a cluster that was not recorded. Such a cluster would be used for system related tasks, e.g., the system cluster.

6.3.1 Replay Scheduler

The blocking events are reproduced by controlling the scheduling order of the tasks. The replay scheduler returns the tasks in the order they were recorded for execution.

Figure 6.2 shows a schematic view of the modified scheduler. Since only the user tasks were recorded, only these tasks are replayed. Therefore, a priority system is established to take care of unrecorded tasks. The replay scheduler has two queues, one for the tasks that are not recorded, shown in the figure on the left side, and one for the tasks that were recorded, shown on the right side. The first group corresponds to the tasks referred to as system tasks, the second group corresponds to the tasks referred to as user tasks.

The priority scheduler first checks for system tasks on the system queue. It is necessary to run system tasks first because they establish any necessary administration properties to make the system work. The system queue is a simple FIFO queue.

If the user queue is not empty, the next task to execute is selected according to the list built from the recorded data. It is necessary to ensure that the replayed task is scheduled on the cluster where it was recorded. The cluster information was stored during recording.

A scheduler can only schedule tasks on the cluster it is associated with. Therefore, it is necessary to check if the next task to be scheduled is recorded on the cluster that is active. If this is the case, the task is removed from the user queue, otherwise nothing is to be scheduled for the replay scheduler on the current cluster.

This does not imply that the system is deadlocked. The next task might need to be scheduled on a different cluster or the task might be blocked in the local debugger while it is examined by the user via KDB.

The replay scheduler provides the following methods:

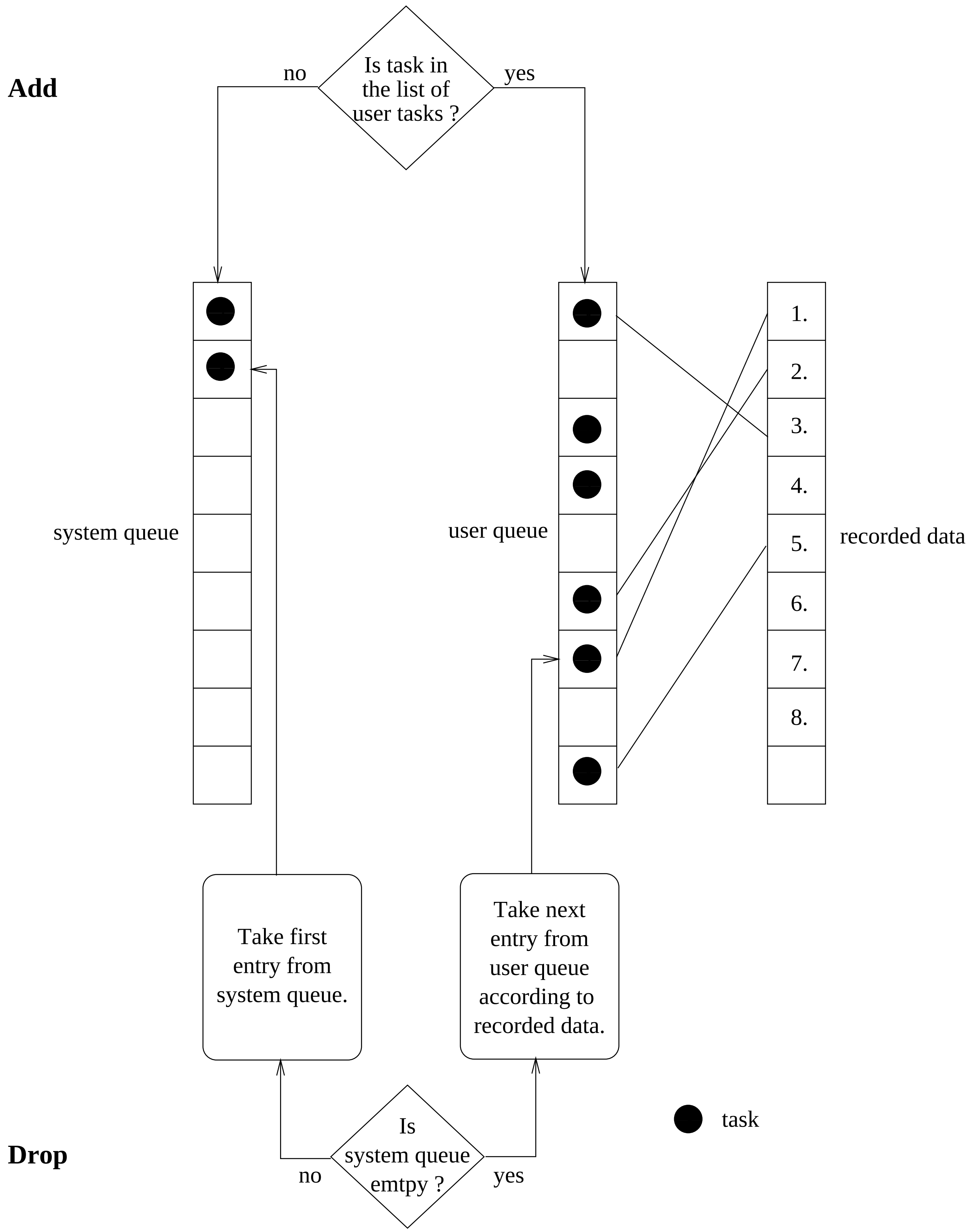

Figure 6.2: The Replay Scheduler

`uEmpty()` returns true, if no task can be scheduled at the moment on the cluster on which the replay scheduler resides. If a task can be scheduled, false is returned.

`uAdd()` adds a task to the appropriate queue of the replay scheduler. It is determined if the task is a system task or a user task and the task is accordingly added to the system queue or the user queue.

`uDrop()` returns the next task to be scheduled. The mechanism is described above.

6.3.2 Local Debugger during Replaying

During recording no local debugger existed; therefore, any additional blocking of the replayed tasks in the local debugger must be dealt with in a special way. The cluster mechanism is used to solve this problem. Tasks migrate from the user cluster to the system cluster when they enter the local debugger. The system cluster is the only cluster not involved in replaying, since it is only scheduling system tasks. The replay task may now block in the local debugger, but the blocking event does not count towards the recorded blockings.

Blocking occurs in the local debugger if a call cannot be accepted immediately or in mutex member routines where execution cannot continue due to blocking. Therefore, a non-mutex wrapper routine as shown at the bottom in figure 6.3 is provided for each mutex member of the local debugger. At the beginning of the wrapper, a `uDebuggerWrapper` object is created. The constructor of this object checks if replaying is active and subsequently migrates the task to the system cluster. The mutex member is called from inside the wrapper. The migration is reversed in the destructor of `uDebuggerWrapper` upon leaving the wrapper routine and return values are passed on.

If the local debugger handled a replay related conditional breakpoint, the task is supposed to block. This case is handled by the `if` statement checking `migrateUser` in the destructor of `uDebuggerWrapper`. The breakpoint handler sets `migrateUser`. The flag is checked in `uDebuggerWrapper` and the status of the task is accordingly set.

It is necessary to set a flag in the task if it was migrated through the wrapper, since migration does cause blocking. The task gets a status of `tempsystem`, which means that it is temporarily handled as a system task, and the status is supposed to be changed to the normal `user` the next time it is scheduled. The replay scheduler acts accordingly.

```
uDebuggerWrapper::uDebuggerWrapper() {
    uThisTask().tempClusterCnt += 1;
    if ( uReplayer::uReplayerInstance
          && uReplayer::uReplayerInstance->getReplay() ) {
        uThisTask().tempCluster = &uThisTask().uGetCluster();
        uThisTask().uMigrate( *uKernelModule::uSystemCluster, true );
        uThisTask().migrateUser = false;
    } // if
} // uDebuggerWrapper

uDebuggerWrapper::~uDebuggerWrapper() {
    if ( uThisTask().tempCluster && uThisTask().tempClusterCnt == 0 ) {
        if ( uThisTask().migrateUser ) {
            uThisTask().uTaskGroup = uBaseTask::user;
        } else {
            uThisTask().uTaskGroup = uBaseTask::tempsystem;
        } // if
        uThisTask().uMigrate( *uThisTask().tempCluster, true );
        uThisTask().tempCluster = NULL;
    } // if
} // uLocalDebugger::exitWrapper


int uLocalDebugger::breakpointHandler( int no ) {
    uDebuggerWrapper tempWrapper;
    int temp = breakpointHandlerMX( no );
    return temp;
} // uLocalDebugger::breakpointHandler
```

Figure 6.3: Wrapper Routine and Helper Methods

6.4 Timeslices

Timeslices are replayed using conditional breakpoints. A timeslice is recorded with the PC and SIC value. Setting a conditional breakpoint at the address provided by the PC value with the SIC value as the condition interrupts the execution at the position where a timeslice occurred. During replaying as well as during recording, the SIC value is incremented if the program is compiled with the `record` flag as described in chapter 5. Therefore, it is possible to use the SIC value during replaying.

Since the PC value is an address, it is necessary to set a breakpoint based on this address. However, KDB only supported breakpoints based on line numbers and/or source code file names, which does not provide sufficient granularity. As discussed in section 4.2.4, it is desirable to replay code exact on an assembler instruction level. Therefore, I augmented KDB with the ability to set breakpoints based on an address.

To ensure that the PC values remain consistent between recording and replay, it is necessary to link all libraries statically with the executable to be replayed. If they are linked dynamically, the address of the code could be different between recording and replaying. Therefore, it would be impossible to use this approach, because the PC positions would require some form of complex adjustment.

6.4.1 Implementation

Whenever a new user task is created in the application being replayed, the task is registered with KDB via the local debugger. For replaying, a new task is created in KDB, which controls the replaying of timeslices. It is called during the registration process of a new user task.

The replay task in KDB checks if the task is in the list of recorded tasks. If this is the case, the task address is stored in the translation table. At this point, it is possible to map the recorded task identifiers to the replay identifiers.

The breakpoints need to be set well in advance of the point at which the actual timeslice occurs. One possibility is to set the first breakpoint for the first recorded timeslice at task creation. Subsequent breakpoints are set whenever the previous breakpoint is hit. This approach was successfully implemented. Nevertheless, the approach has a significant flaw, although it never occurred in practice. Consider the situation in figure 6.4.

In the example, four tasks exist. Since they run in uniprocessor mode, only one task executes at any given time. Time starts at the top and runs toward the bottom of the figure. Next to each execution segment, the values of the SIC at the start and end of this segment are shown. For simplicity, the maximum value for the SIC in the graphic is assumed to be

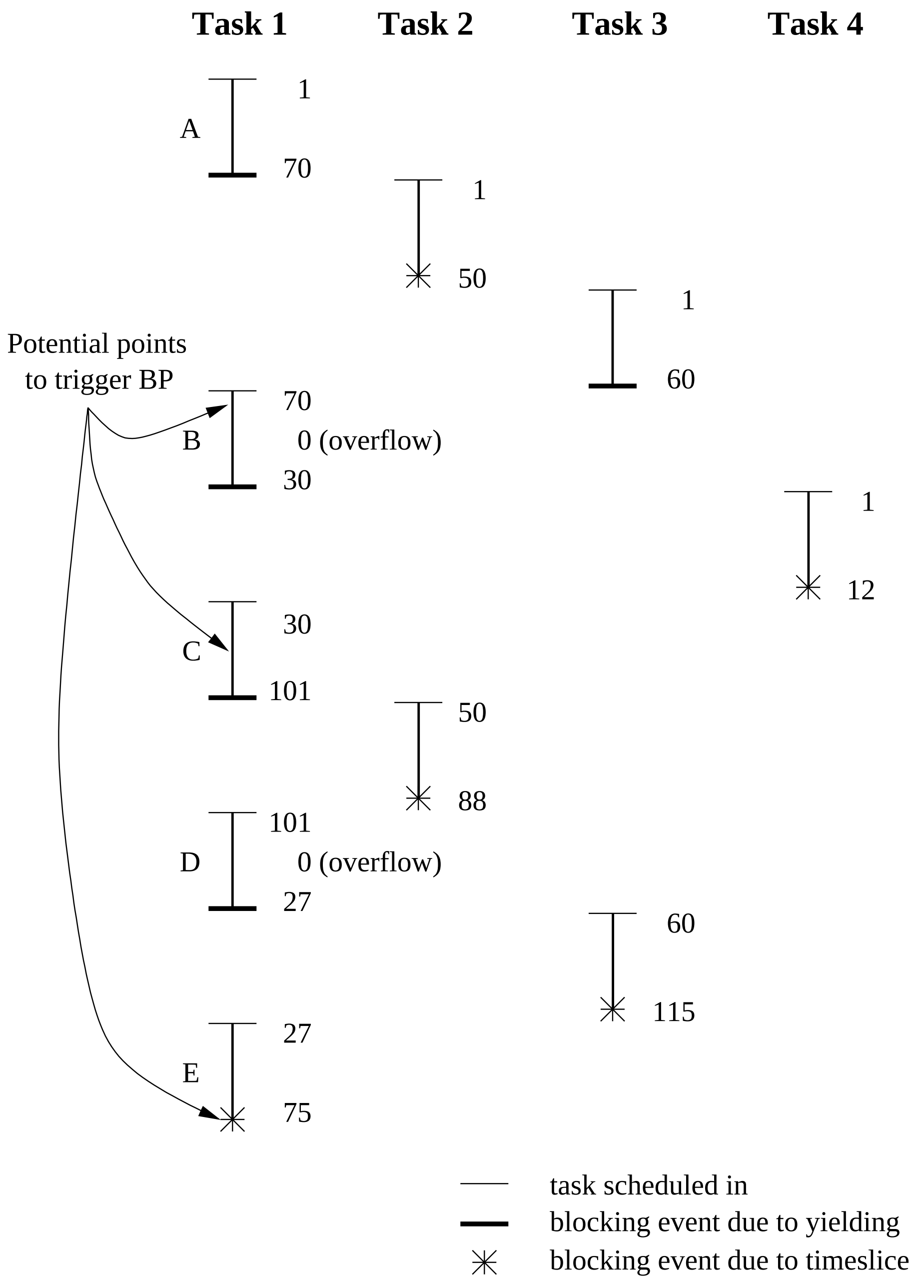

Figure 6.4: Potential Problems while Replaying Timeslices

128. The SIC of task one overflows twice. The execution segments for task one are lettered A to E.

The first timeslice for task one occurs at a SIC value of 75. If the conditional breakpoint for this timeslice is set upon creation of the task, the breakpoint is active through all five execution segments shown and would be triggered at the end of segment E. However, since the SIC overflows twice before the actual timeslice, it has the value 75 at two other positions on the execution path of task one. Segments B, C and E are the positions at which the SIC has the value 75. If by coincidence the PC values are the same at the point at which the SIC hits 75 in segment B and E or C and E, the breakpoint is triggered too early. Therefore, the timeslice is not replayed at the correct place and the whole replaying gets out of synchronisation. This scenario is not only possible for the first timeslice being replayed but also for all subsequent timeslices.

Therefore, setting the breakpoint on task creation and resetting it when hitting the breakpoint is insufficient. Alternatively, it is possible to set the breakpoint at the scheduling event that immediately precedes the position at which the timeslice needs to be recreated. The breakpoint needed to replay task 1's timeslice in segment E is set at the point at the start of segment E. During the execution of the segments A to D, no replaying related breakpoint is set for task 1.

This approach is implemented in SMART. During the preparation phase of the replaying object, it is determined which recorded scheduling events are succeeded by recorded timeslices. Accordingly, a flag is set in the list containing the recorded scheduling events. When a new task is scheduled, the flag is checked. If the flag is set, the global debugger is requested to set the next breakpoint. The local debugger is augmented to provide this functionality.

The user task is blocked in the local debugger while the global debugger sets the breakpoint. KDB retrieves the information for the next timeslice out of the table of timeslices and implements the breakpoint. Upon completion, a message is sent to the local debugger to unblock the user task. Normal execution is resumed afterwards.

As a side effect, this approach has other advantages. A breakpoint set in an executable slows every task down encountering the breakpoint, even if it is not applicable. Tasks for which the breakpoint is not set execute an additional `if` statement. The task for which the breakpoint applies also checks the condition. By setting the breakpoint at the scheduling event which immediately precedes the position of the timeslice, only the task for which the breakpoint applies suffers additional overhead. Other tasks do not execute while the breakpoint is set. Also, the task for which the breakpoint is set checks the breakpoint fewer times than in the first approach. Going back to figure 6.4, task 1 only needs to check the breakpoint in segment E of the execution instead of during all sections. The other tasks are not influenced by the breakpoint set for task 1 at all.

6.4.2 Replaying Roll-Forward

The roll-forward mechanism used in μC++ to ensure mutual exclusion is presented in section 3.1.6. The four different cases are recorded by storing an integer value indicating the case.

It is necessary to replay the behaviour in each of the four cases after a timeslice. The easiest case from a replaying standpoint is the fourth one – since the interrupt is ignored, it is unnecessary to replay it. This case is already handled during recording by simply not recording it at all.

The first case does not require any special attention beyond the presented mechanisms. Only the second and third case need to be considered, and are easily handled by explicitly setting the roll-forward flag, as occurred during recording.

6.4.3 Timeslices during Replay

Timeslices that occur during replaying are ignored. Any replay timeslices would cause unpredictable blockings that did not occur during recording. Therefore, timeslicing is disabled during replaying.

6.5 Performance during Replaying

The slowdown during replaying is hard to summarise in a single percentage, since it depends on the program being replayed. Therefore, the following analysis is based on a single program, which is close to a worst-case example.

The results in the table 6.5 are generated by executing the program in appendix B several times under different circumstances. If the application is run using KDB without any user interaction (i.e., no breakpoints are set), it takes about nine seconds. The detailed numbers are shown in the second column of the table. If the program is replayed, it takes about

	KDB	Replayed	KDB with cond bp
real	0m8.678s	9m14.883s	6m23.728s
user	0m0.670s	8m42.190s	4m47.040s
sys	0m0.080s	0m14.820s	0m38.010s

Table 6.5: Speed Comparison

nine minutes and 15 seconds. This significant slowdown is mainly due to the usage of the conditional breakpoints during replaying. To appreciate this cost, the application was run via KDB without enabling replaying but setting a single conditional breakpoint similar to one set during replay. The results are shown in the fourth column. The application runs for six minutes and 24 seconds before terminating. The reason the execution during replaying is slower are the 15 additional conditional breakpoints being set during replaying.

Therefore, the significant slow down is the result of the implementation of the conditional breakpoints in KDB. It is important to note that the conditional breakpoints in KDB are approximately 30 times faster than watchpoints used in GDB.

6.6 Summary

The replaying part of SMART is presented. Upon startup of replaying, the recorded data is read and stored in an easy accessible data structure, which is done in the application as well as in the debugger KDB. All schedulers are replaced in the application by a special replay scheduler, which schedules data according to a priority (system tasks first) and recorded order.

Execution of additional code needs to be control. Therefore, it is necessary to migrate tasks to a cluster that is not replayed if the task executes in the local debugger. Timeslices are replayed by inserting conditional breakpoints into the application, normal scheduling events are replayed using the scheduler.

Chapter 7

Sample Session

To use SMART requires little work for a user. An additional flag needs to be set during compilation of the program, recording has to be enabled during execution of the application and KDB needs to be told that it is supposed to replay the application to be debugged. This chapter demonstrates the usage of SMART by replaying the program shown in appendix B.

7.1 Preparation

Prior to execution, it is necessary to augment the program with the SIC and insert the statements that increment the SIC at the appropriate places. μC++ is extended with an additional flag `-record`, which needs to be added to the command line during compilation, plus the `-g` flag is necessary for symbol debugging with KDB, e.g.:

```
u++ test8.cc -g -record -o test8
```

This command compiles the program `test8.cc`, which is shown in appendix B, with debugging support and a SIC. The resulting executable is stored in the file `test8`. It does not make much sense to compile a program with recording support but without debugging support, so a warning is issued if this occurs.

7.2 Recording

To execute the program and enable storing of the recorded data, it is necessary to use the script command `record`. The exact format of the command is as follows:

```
record [ -f recordfile ] [ -m memory-size ] target [target-args]
```

With the `-m` option it is possible to specify how much memory should be reserved for the recording data. It is unnecessary to provide the `-f` option. If a name for the `recordfile` is not provided, the recorded data is stored in a file with the name `target.rec`, i.e., the name of the application that is recorded is extended with a `.rec` suffix and that name is used for the name of the file in which the recorded data are stored. For example,

```
record ./test8
```

runs the program and the recorded data is stored in the file `test8.rec`, since no other filename is provided.

`test8` is a non-deterministic program. It starts two tasks, which each add 25 strings to a shared list. Each string contains a monotonically increasing number and an identifier of the task generating the string. The resulting list contains 50 strings. Since the program is non-deterministic and no specific order of access to the string is enforced, the order of the strings in the list changes between runs of the application. Sample output from two different runs of the program are shown in figure 7.1 and 7.2. The output is arranged in seven columns for easier readability and to save space.

```
first 0    second 4   first 8    second 9   first 19   second 17  second 23
first 1    second 5   first 9    second 10  second 13  second 18  second 24
first 2    first 4    first 10   second 11  second 14  second 19
second 0   first 5    first 11   second 12  first 20   second 20
second 1   second 6   first 12   first 15   second 15  second 21
second 2   second 7   first 13   first 16   second 16  first 23
first 3    first 6    first 14   first 17   first 21   first 24
second 3   first 7    second 8   first 18   first 22   second 22
```

Figure 7.1: Sample Output One of `test8`

```
first 0    second 4   second 8   second 13  second 19  first 15   first 23
second 0   first 4    second 9   second 14  second 20  first 16   first 24
second 1   first 5    second 10  second 15  second 21  first 17
second 2   first 6    second 11  second 16  second 22  first 18
first 1    first 7    first 8    second 17  second 23  first 19
first 2    second 5   first 9    second 18  second 24  first 20
first 3    second 6   first 10   first 11   first 13   first 21
second 3   second 7   second 12  first 12   first 14   first 22
```

Figure 7.2: Sample Output Two of `test8`

The content of the file `test8.rec` is shown in figure 7.3. The file is shortened at the end, indicated by the dots and shown in two columns. The output of the execution of `test8` indicates that fourteen timeslices occurred during the run, since fourteen times the output switches from the first task to the second task or vice versa as can be seen by the fourteen lines that start with a `W`.

```
C uSystemCluster 1151440
C uUserCluster 1350648
Executable: ./test8 Time: 922411920
T uMain 1442616 1350648
C joe 1473944
T CountTask 1475208 1350648
W 1475208 1 301469 883448
T CountTask 1474024 1350648
W 1474024 1 334168 883440
S 1475208 1350648
W 1475208 1 472177 883444
S 1474024 1350648
W 1474024 1 677877 883436
S 1475208 1350648
W 1475208 1 642981 883416
S 1474024 1350648
W 1474024 1 849370 883460
S 1475208 1350648
W 1475208 1 1505235 883468
S 1474024 1350648
W 1474024 1 1366288 883460
S 1475208 1350648
W 1475208 1 2022241 883480
S 1474024 1350648
W 1474024 1 1537688 883460
S 1475208 1350648
W 1475208 1 2182557 883436
S 1474024 1350648
W 1474024 1 1709099 883416
S 1475208 1350648
W 1475208 1 2353292 883436
S 1474024 1350648
W 1474024 1 2224488 883440
S 1475208 1350648
TaskFinished: 1475208
S 1474024 1350648
TaskFinished: 1474024
S 1442616 1350648
S 1442616 1151440
.
.
.
S 1442616 1350648
S 1442616 1151440
S 1442616 1350648
TaskFinished: 1442616
```

Figure 7.3: Sample Recording File for Execution of `test8`

7.3 Replaying

A program is replayed via KDB, the debugger for μC++ applications.

KDB can be called with the following options:

```
kdb [ -help ] [ -version ] [ -x | -f logfile ] [ -api ] [ -nointerface ]
[ -debug flags ] [ -replay [ filename ] ] [ target  [ target-args ] ]
```

Each option is explained in more detail:

`-help, -h` prints out a short explanation of the command line options of KDB.

`-version, -v` prints out the version number of KDB.

`-x` displays debug information in a new xterm.

`-f logfile` write debug information into a file named `logfile`.

`-api` control KDB via API (application programming interface) calls.

`-nointerface` do not show graphical user interface, which is useful if KDB is controlled via the API.

`-debug flags` prints out debugging information. The flags specify the group of debugging information to be printed. The `-x` and `-f` option control where the debugging information is printed. If `+` is specified as flag, all debugging information available is printed. A list of other possible values can be found in the file `DebugFac.h` in the KDB source directory.

`-replay [ filename ]` replay the target with data recorded in the file with the name `filename`. If the filename is not specified, the name of the target is augmented with the suffix `.rec` and the resulting string is used as the file name. If the `-replay` option is specified, a target name has to be provided.

`target [ target-args ]` name of the application to be debugged and options for the application. The target name is mandatory if the `-replay` option is specified.

For replaying, only the `-replay` option is relevant. A sample call to replay an application is

```
kdb -replay ./test8
```

This call replays the program `test8` with the recorded data stored in the file `test8.rec`.

The figures 7.4, 7.5 and 7.6 show a debugging session with KDB. The program presented in appendix B is running. The window in figure 7.4 is the main window of KDB, the two windows in figure 7.5 and 7.6 show the two `CountTask` tasks. The result of the execution via KDB with replaying enabled is exactly the same as the one during recording.

By setting additional breakpoints and stepping through the execution, a user can attempt to determine the cause of an error. The user may have to replay the application multiple times, examining different tasks and variables to determine the cause of the problem.

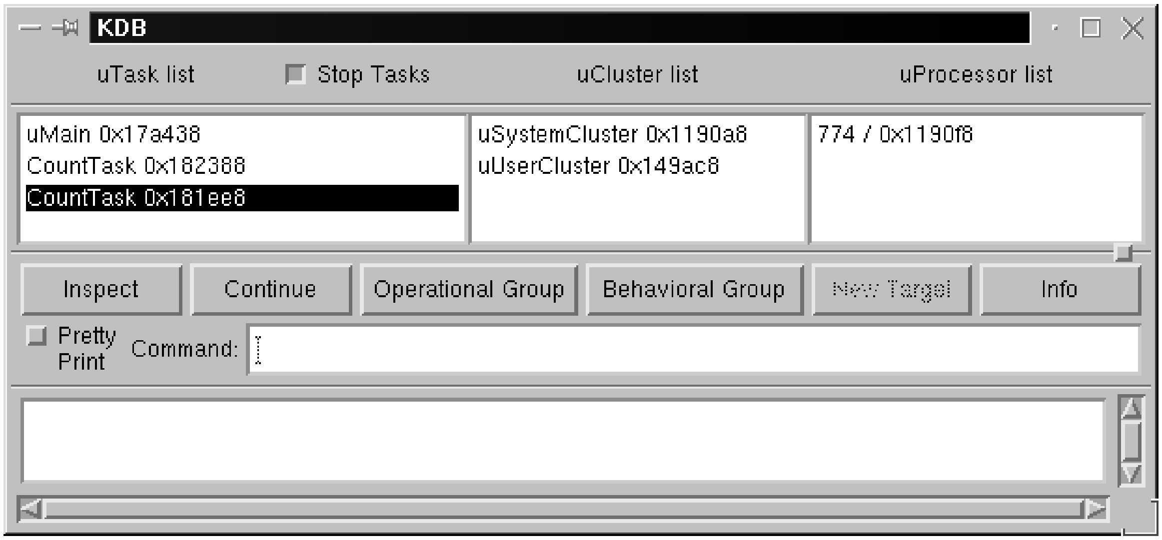

Figure 7.4: KDB Main Window

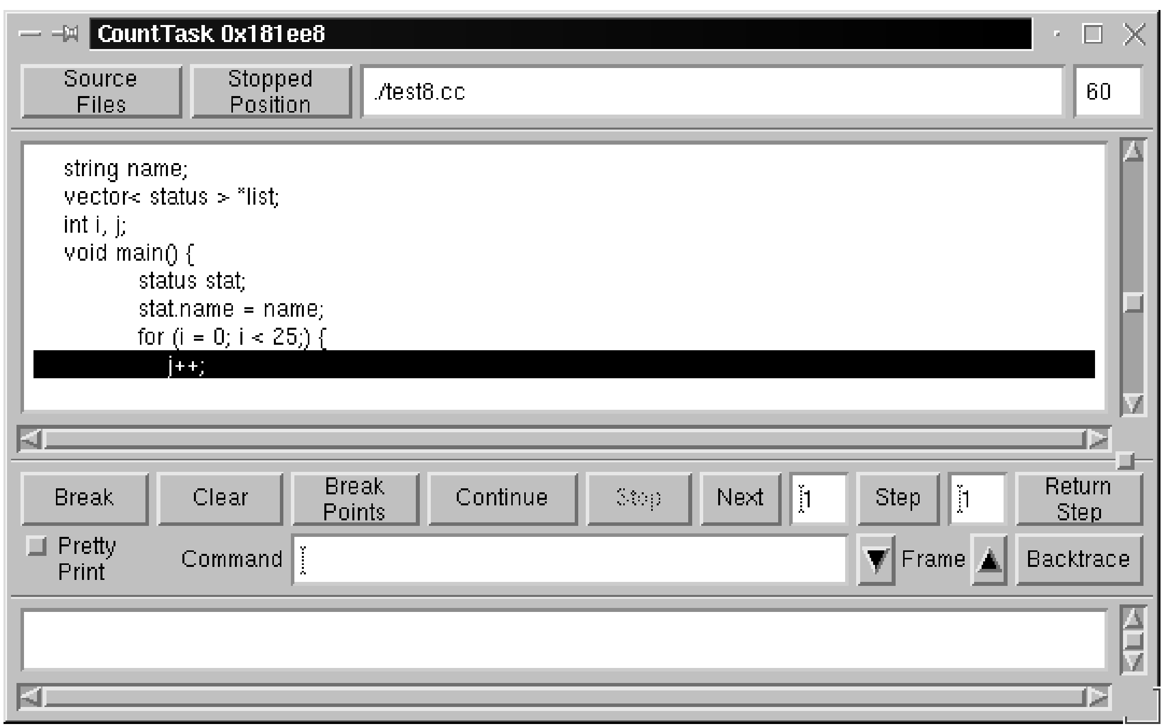

Figure 7.5: Task One

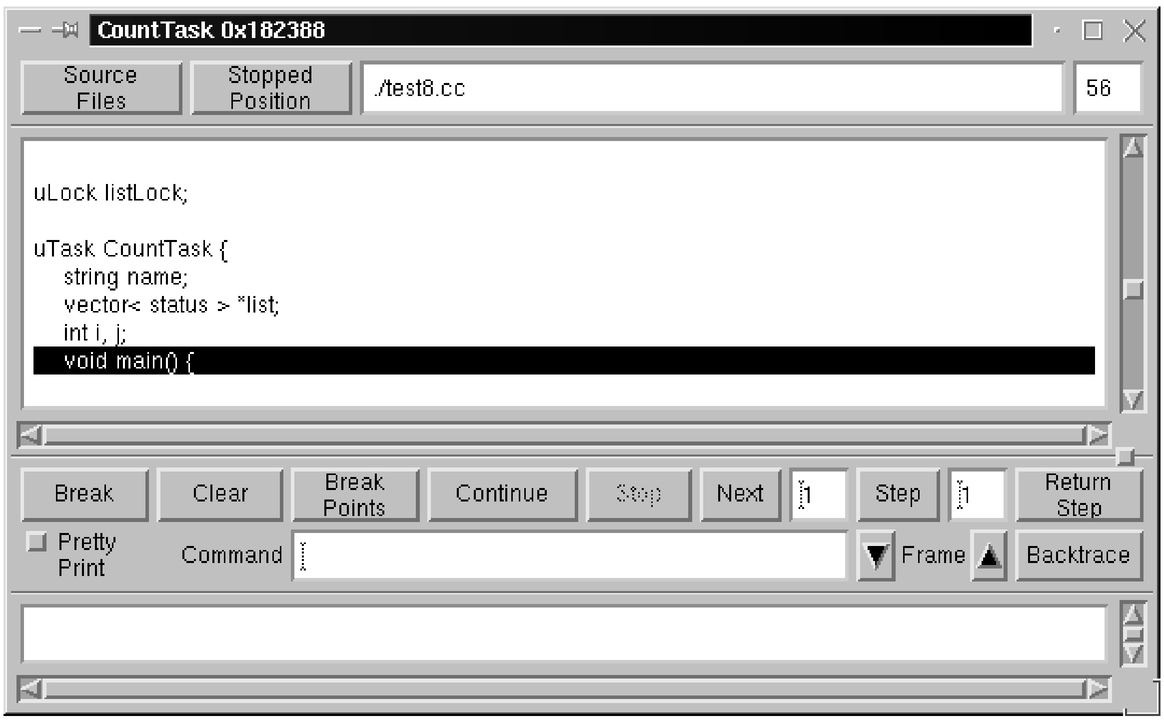

Figure 7.6: Task Two

Chapter 8

Replaying Real-Time

SMART, as described in the previous chapters, is unable to record and replay real-time programs. The current design and subsequent implementation were carefully thought out so they do not hinder the possibility of extending SMART to handle real-time applications.

In this chapter, suggestions are made for extending SMART for real-time applications. The limitation of the current implementation are discussed and the changes for recording and replaying of real-time applications are presented.

8.1 Debugging Real-Time Applications

Debugging real-time applications is a difficult undertaking since many debugging methods interfere with the application and cause the Probe Effect [25] to occur, which is already a severe problem in non-real-time applications. In real-time applications, a change to the timing behaviour of the execution can be catastrophic — it might cause tasks to miss their deadline and, in a waterfall effect, change the whole execution path of all tasks. Therefore, an important aspect of the debugging approach for real-time applications is non-interference. One of the more popular approaches to debugging real-time applications, therefore, is monitoring. DCT [4] is a system providing practically non-intrusive monitoring by using special hardware to access the bus between the processor and the memory. Monitoring an application provides valuable information about the execution — nevertheless it does not provide a way to interactively debug a program nor use the traditional debugging cycle.

Little research has been done concerning the application of replaying to real-time programs. Tsai et al [79] implemented a system using a hardware approach to prevent interfering with the execution. (Their approach is presented in section 2.2.3.) Mueller and Whalley [57] present an approach for a real-time system. HMON [20] is a system to monitor and debug distributed real-time programs.

8.2 Limitations

As Tsai points out, real-time debugging must be non-interfering. Every interference of a real-time application changes the timing behaviour of that application and with it the execution path. Changes in the timing of a real-time application in the magnitude of milliseconds or even microseconds might have a significant influence on the application.

Since SMART is a software approach to replaying, it does introduce additional delay and therefore interference with the execution of the real-time application. An additional constant time overhead is introduced during recording. This overhead cannot be eliminated for a software approach to real-time replaying. It is possible to constantly record the application, which would not eliminate the overhead but integrate it into the application as normal functionality.

A real-time application is often closely coupled with external sensors and the results of the real-time application often provide steering information to external machinery. If SMART is used for replaying real-time programs, it is necessary to simulate these external influences. Because replaying is much slower than recording, it is assumed that a significant number of deadlines will be missed. Nevertheless, in a simulation, it should be possible to provide less sensitive simulated sensors and machinery.

8.3 Possibilities

It is impossible to debug real-time programs using a traditional debugger. A real-time application cannot just be stopped to provide the user with the possibility to query the content of a variable or take the time to understand a certain execution path decision. Stopping execution changes the timing behaviour in real-time programs, as the clock continues to tick. During replay, it is possible to delay the execution.

SMART provides the possibility to debug a real-time application using KDB. This provides the possibility to use the cyclical debugging approach for μC++ real-time programs.

In addition, and as mentioned in previous chapters, SMART provides a way to deterministically replay applications. Since real-time applications are concurrent in μC++, non-determinism and the associated problems during debugging also apply to real-time applications. SMART accurately replays the execution path and provides a way to deterministically replay a real-time application.

8.4 Real-Time in μC++

Real-time programs depend on time. A language in which real-time applications can be written needs to provide some notion of time. μC++ is no exception – it presents different interfaces that provide easy access to clocks and the manipulation of time and duration objects, called `uClock`, `uTime` and `uDuration` respectively.

8.4.1 Notion of Time

A `uClock` is a dynamic object providing a monotonically increasing time value. It is designed to provide the possibility of generating multiple, different clock environments. The current implementation uses POSIX system calls to update the internal time of a `uClock` object.

A `uTime` object stores a time value. It therefore stores a static value. It can be created by either querying a `uClock` or by explicitly setting a specific time by providing values for the date and time or a subset of them.

A `uDuration` object is a static object storing a span of time. It can be set providing a span in seconds and/or nanoseconds.

Arithmetic operations are defined for `uDuration` and `uTime` objects. With these it is possible to calculate the difference between two `uTime` objects and store it in a `uDuration`. Other operations are defined, as well.

8.4.2 Time-Defined Delays

It is possible to suspend a μC++ task or delay its execution for a certain amount of time. To provide a way to trigger an event at a specified time, a time-ordered event queue is necessary. Each entry contains a `uTime` object, which specifies the time the event occurs. Furthermore, it might contain a `uDuration`, providing the possibility for periodic events.

The events are triggered using a SIGALRM. Since timeslices are implemented by using SIGALRMs, too, a timeslice is considered to be a periodic event. Each node contains a signal handler, which is called if the node is removed from the event queue when the actual time is later than or equal to the time at which the event is to occur.

8.4.3 Real-Time Tasks

Several new task types are introduced to μC++ to provide the special needs of a real-time tasks. These are the types periodic, sporadic and aperiodic tasks. A periodic task is a task specified to execute at a certain periodic frequency. Furthermore, a start time, at which the execution of the task starts, and an end time, at which the task terminates, can be provided.

A sporadic task is not as predictable as a periodic task. It only specifies the minimum time between the arrival of that task. Start and end time can also be specified. The readiness of an aperiodic task is not deterministic. It does not have a period nor a minimal time between arrivals, it just appears with a particular deadline.

8.4.4 Scheduler

Real-time is to a large degree the art of scheduling tasks in the right order according to different goals, i.e., meeting the deadline or having the shortest delay. To execute real-time tasks in the order that best fulfils these goals, it is necessary to provide a scheduler that is aware of these goals and tries to fulfil them. The standard round-robin scheduler of μC++ is not capable of that task. To provide greater flexibility, it is necessary to provide a modular way to change the scheduler on a per cluster basis. It is possible to specify a scheduler upon cluster creation and this capability is used during replay to replace the cluster scheduler. A deadline monotonic scheduler has been implemented as the first real-time scheduler. Other real-time scheduling approaches are being developed.

8.5 Design of Replaying Real-Time

Two key aspects need to be considered if replaying is to be extended for real-time applications. The first is the real-time scheduler used for real-time applications and the fact that the scheduler might change in the future, the second is the event-queue and the SIGALRMs that trigger events from the queue.

8.5.1 Recording

Due to the dynamic real-time scheduler, recording must be independent of the scheduler used in the application. It is necessary to record the scheduling of a task outside of the scheduler itself. This can be done by finding every place where the scheduler `uDrop()` function is called. These calls are not centralised and might occur in different contexts. A different possibility is to record the scheduling events at the position where a task starts executing after being scheduled. This point can only occur at one specific location, which is split up into four different cases in the kernel. Recording at these positions provides the ability to centralise the recording of scheduling events without augmenting the scheduler and keeping the code maintainable. This is already done in SMART – scheduling events are recorded at the kernel level, one level below the scheduler.

It is necessary to record the additional SIGALRMs and the consequences. All SIGALRMs are already recorded with the PC and SIC values at which they occur in the SIGALRM handler. As mentioned in section 3.1.6, four different cases exist. The first three are recorded and the fourth one can be ignored, since the SIGALRM is ignored.

A SIGALRM results in a check of the event queue. It is necessary to additionally record the events removed from the event queue and associated the SIGALRMs preceding the event.

In a real-time system, it is necessary for the kernel to execute in constant time. This requirement is maintained by recording, since every recording activity is done in constant time.

8.5.2 Replaying

During replaying, all SIGALRMs need to be recreated accurately. Furthermore, the events associated with the SIGALRMs need to be recreated. The recreation of a SIGALRM can be done by using a conditional breakpoint, as described in section 6.4. Whenever a replaying related conditional breakpoint is hit, further action is necessary.

Each event in the event queue contains an event handler, which is called when the event occurs. Four different types of event handlers exist:

`uCxtSwtchHndlr` This handler is the normal context switch handler, which causes a timeslice to occur by yielding the task that is interrupted. It is not associated with a specific task.

`uWakeupHndlr` This event handler is associated with `uTimeout` statements, which allow a task to block for a specified duration. It is called when a task is to be woken up. The task status is set to ready and it is added to the cluster's ready queue.

`uTimeoutHndlr` This event handler calls into a monitor and indicates that a timeout on a `uAccept` statement occurred. The event is associated with a specific monitor.

`uSelectTimeoutHndlr` This event handler is called whenever a timeout is specified with an I/O operation. This event is associated with the task waiting for I/O.

It is necessary to recreate these handlers and replay them in the conditional breakpoint handler.

8.6 Summary

This chapter presented the pros and cons of extending replaying for real-time applications. The bottom line is that replaying is a useful tool to debug real-time applications if certain restrictions are kept in mind. These restrictions are tougher than for non-real-time applications.

A short introduction to the specific changes needed for μC++ to accommodate real-time applications is given and the changes needed for the recording and replaying process are presented. Time constraints precluded implementing and testing these changes.

Chapter 9

Conclusion

The goal of this thesis is to investigate replay in concurrent shared memory applications that are developed with μC++. To accomplish this goal required finding a feasible solution for replay of shared memory concurrency, and providing an easy to use interface. Subsequent replaying must occur transparently.

After investigating the existing approaches for replaying concurrent shared memory applications it turns out all had significant flaws. Some of them introduce high overhead, others do not accurately replay the executable or do not provide a sufficient granularity for replaying. Others assume a race free program. Only approaches for message passing systems seemed to be feasible and fulfilled the demand of transparency and ease of use without significant user interaction.

The approach presented by Russinovich and Cogswell was the only approach for a shared memory system which seemed to fulfil the stated requirements. However, implementing their idea in μC++ nevertheless proved complex. μC++ supports clusters, which proved to be a difficulty, since tasks needed to be recorded on different clusters. The main simplification that I made was restricting the implementation to uniprocessor. The implemented approach takes the peculiarities of μC++ into account. Recording is possible with low overhead and replaying is highly accurate. Therefore, race conditions and deadlocks can be easily reconstructed.

SMART is integrated in μC++ and KDB. KDB provides a graphical debugging environment that is enriched by SMART. The functionality of μC++ is preserved and not restricted during recording. Every non-real-time μC++ program can be recorded and replayed. The additional work necessary to record and replay real-time programs is sketched out. Extending SMART to support real-time programs provides an interesting, new possibility for debugging real-time applications in general. The limitations and possibilities as well as the design are presented and discussed.

9.1 Future Work

Only the fundamental form of replaying has been implemented in SMART. Other possible extensions and improvements exist.

9.1.1 Effective Replaying of Timeslices

Replaying in the implemented version highly relies on conditional breakpoints in KDB, which have been implemented by Shih [65]. Conditional breakpoints are slow, even though the implementation in KDB is faster than the conventional implementation used by GDB and other debuggers. Speeding up conditional breakpoints or finding a different mechanism that does not depend on them would speed up replaying considerably. Russinovich and Cogswell [64] suggest a more efficient mechanism, which reuses the SIC.

In their approach, the SIC is started at the value at which the timeslice occurred and it is decremented during replay whenever an increment occurred during recording. When the SIC reaches zero, an interrupt is triggered which is caught. Depending on the hardware support, e.g., usage of dedicated registers or a hardware instruction counter as provided by the Mips R10000 chip [29], this might provide a possibility to speed replaying up significantly.

9.1.2 Reverse Execution

An interesting feature would be the ability to run programs backwards. This feature exists for some functional languages such as Lisp in the form of ZStep 95 [80]. It is potentially interesting to have a feature to run a program backwards, even only for a few instructions, providing an undo functionality for the debugger if the user steps too far.

For example a potential approach could monitor all accesses to memory and keep the values before the access in a limited circular buffer. Therefore, the buffer would contain the changes of the last instruction executed. Upon an `undo` command, the value of the buffer would be written back to the addresses that were changed last and the PC would be reset to its previous position. Recap, which was introduced in section 2.2.1.1, shows another potential way of presenting the illusion of reverse execution.

9.1.3 Input Replaying

SMART does not record input to the executable. Recording input data from different potential sources like keyboard, disk, or network activities would potentially be useful. The input could be stored and simulated during a later replaying run. If the input is stored independently of the execution path, it would even be possible to use the same input for different iterations in the debugging and testing cycle of the development phase of an application.

9.1.4 Race Detection

One of the sources for errors in concurrent programs are race conditions. Much research has been done in the field of race detection. If races in a μC++ program could be detected and marked visually, e.g., in KDB, it would be beneficial for the developer. It could be decided if the race is a result of a lack of synchronisation or if it is a desired effect. Interactively changing the outcome of the race would provide the possibility to simulate all different potential execution paths of a concurrent program.

9.1.5 Debugging Real-Time

Other approaches exist to ease debugging of real-time applications. The most common one is monitoring of the application with as little overhead as possible.

A potentially interesting approach is to get hold of the time. This approach has been done in different manners by either manipulating the real-time clock [50] or by altering timeout values [63]. Since μC++ uses its own clocks as a basis for real-time, it might be possible to slow these clocks down or stop them completely to provide a possibility to inspect the real-time application during execution. It is not known if this is a feasible approach, but it has potential for debugging real-time applications.

Appendix A

Experimentation Results

The following sections show details about the various timing experiments done.

A.1 Function Call

The time to execute a function has been tested comparing the execution time of the programs shown in figure A.1 and A.2.

Each program is run ten times. The results for the program in figure A.1 are shown in table A.2, the results for the program in figure A.2 are shown in table A.3. The experiment was performed on a Pentium 133 PC, running Linux and the KDE window manager.

Table A.1 shows a summary of the results. The execution time of the version without function call is about twice as fast as the version with an additional function call.

time	with call	without call	difference in seconds	factor
real	12.9288	6.8555	6.0733	1.8859018306
user	14.1520	6.8200	7.3320	2.0750733138
system	0.0060	0.0050	0.0010	1.2

Table A.1: Comparison of average execution times

A.2 Capacity of the SIC

The SIC is implemented as an unsigned int. In the context of the overflow discussion, it is tested how long it takes to cause such an event. The program in figure A.3 terminates after an overflow occurs. The table A.4 shows test results of ten executions, showing the execution time. The test results are achieved on a Pentium 133 PC running Linux.

```
int j;

void inc() {
    j += 1;
}

int main() {
    for ( int i = 0; i < 100000000; i ++)
        inc();
    return 0;
} // main
```

Figure A.1: Changing a variable in a function

time	1	2	3	4	5	
real	13.063	12.938	12.910	12.909	12.917	
user	12.870	12.860	12.870	12.860	12.870	
sys	0.020	0.020	0.000	0.010	0.000	
	6	7	8	9	10	avg
real	12.908	12.906	12.910	12.906	12.921	12.9288
user	12.860	12.870	12.860	12.860	12.880	14.1520
sys	0.010	0.000	0.010	0.010	0.000	0.0060

Table A.2: Changing a variable in a function, times in seconds

```
int j;

int
main() {
    for ( int i = 0; i < 100000000; i ++)
        j +=1;
    return 0;
} // main
```

Figure A.2: Changing a variable directly

time	1	2	3	4	5	
real	6.937	6.844	6.848	6.851	6.838	
user	6.840	6.830	6.820	6.810	6.810	
sys	0.010	0.000	0.000	0.010	0.010	
	6	7	8	9	10	avg
real	6.848	6.840	6.859	6.849	6.841	6.8555
user	6.820	6.820	6.820	6.800	6.830	6.8200
sys	0.000	0.000	0.010	0.010	0.000	0.0050

Table A.3: Changing a variable directly, times in seconds

```
int
main() {
    unsigned int i = 1;
    for (; i > 0;)
        i += 1;
    return 0;
} // main
```

Figure A.3: Testprogram to achieve an overflow of an unsigned int

time	1	2	3	4	5	
real	3m15.290s	3m15.195s	3m15.207s	3m15.644s	3m15.200s	
user	3m15.170s	3m15.180s	3m15.170s	3m15.380s	3m15.160s	
system	0m0.010s	0m0.010s	0m0.010s	0m0.000s	0m0.010s	
	6	7	8	9	10	average
real	3m15.202s	3m15.195s	3m15.210s	3m15.222s	3m15.206s	3m15.2571s
user	3m15.170s	3m15.170s	3m15.180s	3m15.170s	3m15.170s	3m15.192s
system	0m0.000s	0m0.010s	0m0.000s	0m0.000s	0m0.010s	0m0.006s

Table A.4: Testresults for achieving an overflow

Appendix B

Test Program

The following listing presents the program used to obtain the example data in chapter 6 and 7.

```
// ./test8.cc –
//
// This programm starts two tasks which update the same list without
// explicit coordination (scheduling).

#include <uC++.h>
#include <uIOStream.h>
#include <string>
#include <vector>

// Routines needed to do I/O on strings from uC++
uOStream &operator<<( uOStream &os, const string s ) {
    uIosWrapper wrapper( os );
    (ostream &)os << s;
    return os;
}

// set timeslicing to very fast
int uDefaultPreemption() {
    return 1;
} // uDefaultPreemption

struct status {
    string name;
    int count;
}; // struct status

uLock listLock;

uTask CountTask {
    string name;
    vector< status > *list;
```

```
        int i, j;
        void main() {
            status stat;
            stat.name = name;
            for (i = 0; i < 25;) {
                j++;
                if ( ( j % 100000 ) == 0 ) {
                    stat.count = i;
                    listLock.uAcquire();
                    list->push_back( stat );
                    listLock.uRelease();
                    j = 0;
                    i += 1;
                } // if
            } // for
        } // main
      public:
        CountTask( const string &name, vector< status > *list ) : name( name ), list( list ) {}
    }; // CountTask


    void uMain::main() {

        vector < status > statlist;

        {
            CountTask one( "first", &statlist );
            CountTask two( "second", &statlist );
            uCluster test("joe");
        }

        vector< status >::iterator pos;

        for (pos = statlist.begin(); pos != statlist.end(); pos++ ) {
            uCout << (*pos).name << " " << (*pos).count << endl;
        } // for

    } // uMain::main()
```

Bibliography

[1] Adams, D., and Bisiani, R. The carnegie-mellon university distributed speech recognition system. *Speech Technology 2*, 3 (Apr. 1986).

[2] Baron, R., Rashid, R., Siegel, E., Tevanian, A., and Young, M. Mach-1: An operating system environment for large scale multiprocessor applications. *IEEE Software*, Special Issue (July 1985).

[3] Bershad, B. N., Redell, D. D., and Ellis, J. R. Fast mutual exclusion for uniprocessors. *SIGPLAN Notices 27*, 9 (Sept. 1992), 223–235.

[4] Bhatt, D., Ghonami, A., and Ramanujan, R. An instrumented testbed for real-time distributed systems development. In *Proceedings Real-Time Systems Symposium* (1730 Massachusetts Avenue, N.W., Washington, D.C. 20036-1903, USA, Dec. 1987), vol. 8, IEEE, Computer Society Press of the IEEE, pp. 241–250.

[5] Bisiani, R., and Forin, A. Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions on Computers* (Aug. 1988).

[6] Buck, J. *G++ FAQ, Frequently asked Questions about the GNU C++ compiler*, June 1998. Available at http://www.cygnus.com/misc/g++FAQ_toc.html.

[7] Buhr, P. A. Profquotes. *mathNEWS 79*, 2 (Jan. 1998), 5.

[8] Buhr, P. A. *Understanding Control Flow with Concurrent Programming using μC++*. to be published, 1999. A Draft is available at http://www.undergrad.math.uwaterloo.ca/~cs342/documents/uC++book.ps.

[9] Buhr, P. A., Fortier, M., and Coffin, M. H. Monitor classification. *ACM Computing Surveys 27*, 1 (Mar. 1995), 63–107.

[10] Buhr, P. A., and Mok, W. Y. R. Advanced abnormal event handling mechanisms. submitted.

[11] Buhr, P. A., and Stroobosscher, R. A. μC++ annotated reference manual. Tech. rep., University of Waterloo, Canada, Waterloo, Ontario, N2L 3G1, Canada, 1998. Available via ftp from plg.uwaterloo.ca/pub/uSystem/uC++.ps.gz.

[12] Carver, R. H., and Tai, K.-C. Replay and testing for concurrent programs. *IEEE Software 8*, 2 (Mar. 1991), 66–74.

[13] Chamberlain, S., and Cygnus Support. *libbfd, The Binary File Descriptor Library*, First for BFD version < 3.0 ed., Apr. 1991. Distributed with libbfd.

[14] Courtois, P. J., Heymans, F., and Parnas, D. L. Concurrent control with "readers" and "writers". *Communications of the ACM 14*, 10 (Oct. 1971), 667–668.

[15] Curtis, R., and Wittie, L. Bugnet: A debugging system for parallel programming environments. In *Proceedings of the 3rd International Conference on Distributed Computing Systems* (Miami / Ft. Lauderdale, Florida, USA, Oct. 1982), IEEE, pp. 394–399.

[16] Digital Equipment Corporation. *Ptrace(2), Ultrix documentation*, Mar. 1990.

[17] Dijkstra, E. W. Cooperating sequential processes. Tech. rep., Technological University, Eindhoven, The Netherlands, 1965. Reprinted in [26] pp. 43–112.

[18] Dijkstra, E. W. Go to statement considered harmful. *Communications of the ACM 11*, 3 (Mar. 1968), 147–148.

[19] Dijkstra, E. W. Hierarchical ordering of sequential processes. *Acta Informatica 1* (1971), 115–138.

[20] Dodd, P. S., and Ravishankar, C. V. Monitoring and debugging distributed real-time programs. *Software — Practice and Experience 22*, 10 (Oct. 1992), 863–877.

[21] Eisenstadt, M. My hairiest bug war stories. *Communications of the ACM 40*, 4 (Apr. 1997), 30–37.

[22] Feldman, S. I., and Brown, C. B. Igor: A system for program debugging via reversible execution. *ACM SIGPLAN Notices 24*, 1 (Jan. 1989), 112–123.

[23] Forin, A. Debugging of heterogeneous parallel systems. *ACM SIGPLAN Notices 24*, 1 (Jan. 1989), 130–140.

[24] Free Software Federation. limits.h. Standard C header file, 1992.

[25] Gait, J. A probe effect in concurrent programs. *Software - Practice and Experience 16*, 3 (Mar. 1986), 225–233.

[26] Genuys, F., Ed. *Programming Languages.* Academic Press, New York, 1968. NATO Advanced Study Institute, Villard-de-Lans, 1966.

[27] Goldszmidt, G. S., and Yemini, S. High-level language debugging for concurrent programs. *ACM Transactions on Computer Systems 8*, 4 (Nov. 1990), 311–336.

[28] Hansen, P. B. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1973.

[29] Heinrich, J. *MIPS R10000 Microprocessor User'sManual*, second ed. MIPS Technologies, 2011 North Shoreline, Mountain View, California 94039-7311, USA, Oct. 1996.

[30] Hoare, C. Monitors: An operating-system structuring concept. *Communications of the ACM 17*, 10 (Oct. 1974), 549–557.

[31] Intel Corporation, Ed. *Intel Architecture Software Developer's Manual.* Intel Corporation, P.O. Box 7641, Mt. Prospect, IL 60056-7641, USA, 1997.

[32] International, S., Ed. *The SPARC Architecture Manual.* Prentice-Hall, Englewood Cliffs, New Jersey 07632, USA, 1992.

[33] Jones, A. K., and Schwarz, P. Experience usinge multiprocessor systems – a status report. *ACM Computing Surveys 12*, 2 (June 1980), 121–165.

[34] Kane, G., and Heinrich, J. *MIPS RISC Architecture.* Prentice-Hall, Englewood Cliffs, New Jersey 07632, USA, 1992.

[35] Karsten, M. A multi-threaded debugger for multi-threaded applications. Diplomarbeit, Universität Mannheim, Schloss, 68131 Mannheim, Germany, Aug. 1995.

[36] Kessler, P. B. Fast breakpoints: Design and implementation. *ACM SIGPLAN Notices 25*, 6 (June 1990), 78–84.

[37] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* (July 1978), 558–564.

[38] LeBlanc, T. J., and Mellor-Crummey, J. M. Debugging parallel programs with instant replay. *IEEE Transactions on Computers 36*, 4 (Apr. 1987), 471–482.

[39] LeDoux, C. H., and Parker, Jr., D. S. Saving traces for ada debugging. In *Ada In Use: Proceedings of the Ada International Conference* (Cambridge, Great Britain, Sept. 1985), J. G. Barnes and G. A. Fisher, Jr., Eds., ACM, Cambridge University Press, pp. 97–108.

[40] LEHMANN, D., AND RABIN, M. O. On the advantage of free choice: A symmetric and fully distributed solution to the Dining Philosophers problem (extended abstract). In *Proc. Eighth Ann. ACM Symp. on Principles of Programming Languages* (1981), pp. 133–138.

[41] LIEBERMAN, H. The debugging scandal. *Communications of the ACM 40*, 4 (Apr. 1997), 27–29.

[42] LIM, JR., P. E. Real-time in a concurrent, object-oriented programming environment. Master's thesis, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada, 1996.

[43] LINTON, M. A. Distributed management of a software database. *IEEE Software 4*, 6 (Nov. 1987), 70–76.

[44] LINUX PROGRAMMER'S MANUAL. Alarm(2): alarm - set an alarm clock for delivery of a signal. Manual Page, July 1993.

[45] LINUX PROGRAMMER'S MANUAL. Getitimer(2): getitimer, setitimer - get or set value of an interval timer. Manual Page, Aug. 1993.

[46] LINUX PROGRAMMER'S MANUAL. Ptrace(2): ptrace - process trace. Manual Page, July 1993.

[47] LINUX PROGRAMMER'S MANUAL. Proc(5): proc - process information pseudo-filesystem. Manual Page, July 1996.

[48] LINUX PROGRAMMER'S MANUAL. Time(2): time - get time in seconds. Manual Page, Sept. 1997.

[49] LYNCH, N. A. Fast allocation of nearby resources in a distributed system. In *Conference Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing* (Los Angeles, California, 28-20 Apr. 1980), pp. 70–81.

[50] Maio, A. D., Ceri, S., and Reghizzi, S. C. Execution monitoring and debugging tools for Ada using relational algebra. *ACM Ada Letters 5*, 2 (1985), 109–123.

[51] Marlin, C. D. *Coroutines : a programming methodology, a language design, and an implementation.* Springer-Verlag, Berlin, Germany, 1980.

[52] McCarthy, J. History of LISP. *ACM SIGPLAN Notices 13*, 8 (Aug. 1978), 217–223.

[53] McDowell, C. E., and Helmbold, D. P. Debugging concurrent programs. *ACM Computing Surveys 21*, 4 (Dec. 1989), 593–622.

[54] Mellor-Crummey, J. M., and LeBlanc, T. J. A software instruction counter. *ACM SIGPLAN Notices 24*, Special Issue (May 1989), 78–86.

[55] Milliken, W. *Chrysalis Programmer's Manual.* BBN Laboratories, June 1985.

[56] Mosberger, D., Druschel, P., and Peterson, L. L. A fast and general software solution to mutual exclusion on uniprocessors. Tech. Rep. TR94-07, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA, June 1994.

[57] Mueller, F., and Whalley, D. B. On debugging real-time applications. In *Workshop on Language, Compiler, and Tool Support for Real-Time Systems* (June 1994), ACM SIGPLAN.

[58] Netzer, R. H., and Miller, B. P. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems 1*, 1 (Mar. 1992), 74–88.

[59] Open Software Foundation. *Introduction to OSF / DCE.* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1993.

[60] Pan, D. Z., and Linton, M. A. Supporting reverse execution of parallel programs. *ACM SIGPLAN Notices 24*, 1 (Jan. 1989), 124–129. SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, Madison, WI, May 1988.

[61] Pucci, M. F., and Alberi, J. L. Experiences with efficient interprocess communication in DUNE. In *Distributed and Multiprocessor Systems Workshop Proceedings, October 5–6, 1989. Fort Lauderdale, FL* (Berkeley, CA, USA, Oct. 1989), USENIX Association, Ed., USENIX, pp. 349–360.

[62] Rashid, R., Baron, R., Forin, A., Golub, D., Fones, M., Julin, D., Orr, D., and Sanzi, R. Mach: A foundation for open systems. In *Workstation Operating Systems: Proceedings of the 2nd Workshop on Workstations and Operating Systems* (1730 Massachusetts Avenue, N.W., Washington, D.C. 20036-1903, USA, Sept. 1989), IEEE, Computer Society Press of the IEEE, pp. 109–113.

[63] Rowe, P. K., and Pagurek, B. Remedy: A real-time, multiprocessor, system level debugger. In *Proceedings Real-Time Systems Symposium* (1730 Massachusetts Avenue, N.W., Washington, D.C. 20036-1903, USA, Dec. 1987), vol. 8, IEEE, Computer Society Press of the IEEE, pp. 230–240.

[64] Russinovich, M., and Cogswell, B. Replay for concurrent non-deterministic shared-memory applications. *ACM SIGPLAN Notices 31*, 5 (May 1996), 258–266.

[65] Shih, J. Debugging concurrent programs. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1, 1996.

[66] Sites, R. L. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[67] Stallman, R. M. *Using and Porting GNU CC*, updated for version 2.7.2.1 ed. Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, June 1996. Distributed with gcc.

[68] Stallman, R. M. *Using and Porting GNU CC*, updated for egcs-1.1.1 ed. Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, Mar. 1998. Distributed with egcs.

[69] Stallman, R. M., and Cygnus Support. *Debugging with GDB, The GNU Source-Level Debugger*, 4.12 for gdb version 4.16 ed. Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, Jan. 1994. Distributed with gdb.

[70] Steele, Jr., G. L., and Gabriel, R. P. The evolution of lisp. *ACM SIGPLAN Notices 28*, 3 (Mar. 1993), 231–270.

[71] Stotts, Jr., P. D. A comparative survey of concurrent programming languages. *ACM SIGPLAN Notices 9*, 17 (Sept. 1982), 76–87.

[72] Stroustrup, B. *The C++ Programming Language*, third ed. Addison-Wesley, Reading, Massachusetts 01867, USA, 1997.

[73] Sun Microsystems. *Ptrace(2), SunOS Reference Manual, Vol. II*, Jan. 1990.

[74] Sun Microsystems. limits.h. Standard C header file, 1996.

[75] Sun Microsystems. proc(4): /proc, the process file system. Manual Page, Feb. 1997.

[76] Tanenbaum, A. S. *Modern Operating Systems*, first ed. Prentice-Hall, Englewood Cliffs, New Jersey 07632, USA, 1992.

[77] Taylor, D. A prototype debugger for Hermes. In *Proceedings of the 1992 CAS Conference* (Nov. 1992), pp. 29–42.

[78] Taylor, D., and Buhr, P. A. *POET with μC++*. University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada, Sept. 1997.

[79] Tsai, J. J. P., Fang, K.-Y., Chen, H.-Y., and Bi, Y.-D. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering 16*, 8 (Aug. 1990), 897–916.

[80] Ungar, D., Lieberman, H., and Fry, C. Debugging and the experience of immediacy. *Communications of the ACM 40*, 4 (Apr. 1997), 38–43.

[81] Wirth, N. *Programming in Modula-2*, fourth ed. Springer-Verlag, Berlin, Germany, 1988.

[82] Wittie, L., and van Tilborg, A. Micros, a distributed operating system for micronet, a reconfigurable network computer. *IEEE Transactions on Computers C-29*, 12 (1980), 1133–1144.

[83] X3 Secretariat, Ed. *Draft Standard - The C++ Language*. Information Technology Council, Washington, DC, USA, 1997. X3J16 / 97 - 14882.

[84] Yong, Y. Replay and distributed breakpoints in an osf dce environment. Master's thesis, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada, 1995.

[85] Yong, Y. M., and Taylor, D. J. Performing replay in an OSF DCE environment. In *Procedings of the 1995 CAS Conference* (Toronto, Ontario, Canada, 1995), IBM Canada Ltd. Laboratory, Centre for Advanced Studies, pp. 52–62.