Realizing Iterative-Relaxed Scheduler in Kernel Space

Master-Arbeit
Sreeram Sadasivam
2662284

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik

Fachgebiet Dependable Embedded Systems and Software
Prof. Neeraj Suri Ph.D

Realizing Iterative-Relaxed Scheduler in Kernel Space
Master-Arbeit
2662284

Eingereicht von Sreeram Sadasivam
Tag der Einreichung: 16. Marz 2018

Gutachter: Prof. Neeraj Suri Ph.D
Betreuer: Patrick Metzler

Technische Universität Darmstadt
Fachbereich Informatik

Fachgebiet Dependable Embedded Systems and Software
Prof. Neeraj Suri Ph.D

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Master-Arbeit 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 dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die schriftliche Fassung stimmt mit der elektronischen Fassung überein.

Darmstadt, den 16. Marz 2018 Sreeram Sadasivam

Contents

List of Figures

List of Tables

Abstract

Concurrency bugs which are often resident in multi-threaded programs with shared memory designs are difficult to find and reproduce. Deterministic multi-threading (DMT) is one such scheme indicated to resolve the above difficulty. But, DMT presents the challenge of having no scheduling constraints. However, currently there are no such techniques that allow to control the schedule of a multi-threaded program on a fine-grained level, i.e, on the level of single memory accesses. A design with a granularity of single memory accesses would help in enforcing the scheduling constraint. This thesis focuses on moving the scheduling decision to kernel space. Thus, improving the execution time of the user program.

In the existing design, we have a thread scheduler and a verification engine. The verification engine primarily focuses on instrumenting the user code and realizing memory accesses made by various user threads. The set of safe schedules are provided by the verification engine for the given user program. The generated execution pattern is later realized with the thread scheduler, when the user program is executed. The scheduler thread is realized in user space. However, there is a problem of the scheduler thread getting context switched when executed in user space. The operating system scheduler might ignore the scheduling constraint set by the user level scheduler. Moving the scheduler task to the kernel space would help to realize the safe scheduling constraints set by the user. With the migration of scheduler module to the kernel space, there arises certain design changes and challenges.

The approach used in the thesis would be bench-marked on various benchmarking programs such as Indexer, Last Zero, Fibonacci and Dining Philosopher's Problem. The evaluation is performed on the execution overhead exerted by the transition to a loadable kernel module. The evaluation will also relate to the number of synchronizations taking place when using the ioctl calls. The above comparison would also cover evaluations across instrumented and un-instrumented code. The scaling of thread count to core count is also considered for the above evaluations. The approach presented in this work is expected to reduce the execution overhead and also some shortcomings generated by its counterpart user-space design.

1 Introduction

—introduction comes here—

2 Background

2.1 Software Verification

Software programs are becoming increasingly complex. With the rise in complexity and technological advancements, components within a software have become susceptible to various erroneous conditions. Software verification have been perceived as a solution for the problems arising in the software development cycle. Software verification is primarily verifying if the specifications are met by the software[14].

There are two fundamental approaches used in software verification - dynamic and static software verification[14]. Dynamic software verification is performed in conjunction with the execution of the software. In this approach, the behavior of the execution program is checked- commonly known as Test phase. Verification is succeeding phase also known as Review phase. In dynamic verification, the verification adheres to the concept of test and experimentation. The verification process handles the test and behavior of the program under different execution conditions. Static software verification is the complete opposite of the previous approach. The verification process is handled by checking the source code of the program before its execution. Static code analysis is one such technique which uses a similar approach.

The verification of software can also be classified in perspective of automation - manual verification and automated verification. In manual verification, a reviewer manually verifies the software. Whereas in the latter approach, a script or a framework performs verification.

Software verification is a very broad area of research. This thesis work is focused on automated software verification for multithreaded programming.

2.2 Multithreaded Programming

Computing power has grown over the years. Advancements are made in the domain of computer architecture by moving the computing power from single-core to multi-core architecture. With such advancement, there were needs to adapt the programming designs from a serialized execution to more parallelizable execution. Various parallel programming models were perceived to accommodate the perceived progression. Multithreaded programming model was one of the designs considered for the performance boost in computing[6].

Threads are small tasks executed by a scheduler of an operating system, where the resources such as the processor, TLB (Translation Lookaside Buffer), cache, etc., are shared between them. Threads share the same address space and resources. Multithreading addresses the concept of using multiple threads for having concurrent execution of a program on a single or multi-core architectures. Inter-thread communication is achieved by shared memory. Mapping the threads to the processor core is done by the operating system scheduler. Multithreading is only supported in operating systems which has multitasking feature.

Advantages of using multithreading include:

- Fast Execution
- Better system utilization
- Simplified sharing and communication

- Improved responsiveness - Threads can overlap I/O and computation.
- Parallelization

Disadvantages:

- Race conditions
- Deadlocks with improper use of locks/synchronization
- Cache misses when sharing memory

2.3 Concurrency Bugs

Concurrency bugs are one of the major concerns in the domain of multithreaded environment. These bugs are very hard to find and reproduce. Most of these bugs are propagated from the mistakes made by the programmer[17]. Some of these concurrency bugs include:

- Data Race
- Order violation
- Deadlock
- Livelock

Non-deterministic behavior of threads is one of the reasons for having the among mentioned bugs. Data race and order violation are classified as race condition bugs. Whereas, deadlock and livelock are classified as lack of progress bugs.

2.3.1 Race Condition

Race condition is one of the most class of common concurrency problems. The problem arises, when there are concurrent reads and writes of a shared memory location. As stated above, the problem occurs with non-deterministic execution of threads.

Consider the following example, you have three threads and they share two variables x and y [6]. The value of x is initially 0.

Thread 1	Thread 2	Thread 3
(1) x = 1	(2) x = 2	(3) y = x

Table 2.1.: Race condition example

If the statements (1), (2) and (3) were executed as a sequential program. The value of y would be 2. When the same program is split to three threads as shown in the above Table 2.1, the output of y becomes unpredictable. The possible values of y = {0,1,2}. The non-deterministic execution of the threads makes the output of y non-deterministic. Table 2.2 depicts possible executions for the above multithreaded execution.

The above showcased problem is classified as race condition bug. Ordered execution of reads and writes can fix the problem.

2.3.2 Lack Of Progress

Lack of progress is another bug class observed in multithreaded programs. Some of the bugs under this class include deadlocks and livelocks.

Execution Order	Value of y
(3),(1),(2)	0
(3),(2),(1)	0
(2),(1),(3)	1
(1),(3),(2)	1
(1),(2),(3)	2
(2),(3),(1)	2

Table 2.2.: Possible executions

Deadlock

Deadlock is a state in which each thread in thread pool is waiting for some other thread to take action. In terms of multithreaded programming environment, deadlocks occur when one thread waits on a resource locked by another thread, which in turn is waiting for another resource locked by another thread. If a thread is unable to change its state indefinitely because the resource requested by it are being held by another thread, then the entire system is said to be in deadlock[7].

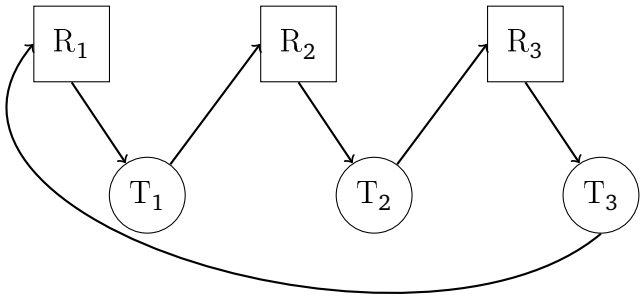


Figure 2.1.: Dead Lock Example

In the example depicted in Fig 2.1, we have three threads T_1, T_2, T_3 and three resource instances R_1, R_2, R_3. The figure depicts hold and wait by each threads. Thread T_1 holds resource R_1 and waits for the acquisition of resource R_2 from thread T_2. T_2 cannot relinquish resource R_2, unless it acquires resource R_3 for its progress. But, resource R_3 is acquired by T_3 and is waiting for R_1 from T_1. Thus, making a circular wait of resources. This example clearly explains the dependency of resources for the respective thread progress.

Deadlock can occur if all the following conditions are met simultaneously.

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

These conditions are known as Coffman conditions[9].

Deadlock conditions can be avoided by having scheduling of threads in a way to avoid the resource contention issue.

Livelock

Livelock is similar to deadlock, except the state of threads change constantly but, with none progressing. Livelock is special case of resource starvation of threads/processes. Some deadlock detection algorithms are susceptible to livelock conditions when, more than one process/thread tries to take action[17][7]. The above mentioned situation can be avoided by having one priority process/thread taking up the action.

2.4 Model Checking

From section 2.3, it is very clear that there needs to be verification for multithreaded programs. The verification solutions range from detecting causality violations to correctness of execution[11]. Model checking is an example of such a technique. It is used for automatically verifying correctness properties of finite-state concurrent systems[8][3]. This technique has a number of advantages over traditional approaches that are based on simulation, testing and deductive reasoning. When solving a problem algorithmically, both the model of the system and the specification are formulated in a precise mathematical language. Finally, the problem is formulated as a task in logic, namely to verify whether a given structure adheres to a given logical formula. The technique has been successfully used in practice to verify complex sequential designs and communication protocols[8]. Model checker tries to verify all possible states of a system in a brute force manner[2]. Thus, making state explosion as one of the major challenges, which is discussed in detail in section 2.4.1. Model checking tools usually verify partial specification for liveness and safety properties[11]. Model checking algorithms generate set of states from the instructions of a program, which are later analyzed. There is a need to store these states for asserting the number of visits made them are at-most once. There are two methods commonly used to represent states:

- Explicit-state model checking
- Symbolic model checking

Advantages of using model checking:

- Generic verification approach used across various domains of software engineering.
- Supports partial verification, more suited for assessment of essential requirements for a software.
- Not vulnerable to the likelihood that an error is exposed.
- Provides diagnostic information thus, making it suitable for debugging purposes.
- Based on graph theory, data structures and logic thus, making it 'sound and mathematical underpinning'.
- Easy to understand and deploy.

Disadvantages of using model checking:

- State explosion problem.
- Appropriate for control-intensive applications rather than data-intensive applications.
- Verifies system model and not the actual system.
- Decidability issues when considering abstract data types or infinite state systems.

2.4.1 State explosion problem

The state space of a program is exponential in nature when it comes to number of variables, inputs, width of the data types, etc,. Presence of function calls and dynamic memory allocation makes it infinite[11]. Concurrency makes the situation worse by having interleaving of threads during execution. Interleaving generates exponential number of ways to execute a set of statements/instructions. Thus, having an explosion in state space. There are various techniques used to avoid the state explosion problem.

2.4.2 Explicit-state Model Checking

Explicit-state model checking methods recursively generate successors of initial states by constructing a state transition graph. Graphs are constructed using depth-first, breadth-first or heuristic algorithms. Erroneous states are determined 'on the fly' thus, reducing the state space. A property violation on the newly generated states are regarded as erroneous states. Hash tables are used for indexing the explored states. If there is insufficient, memory lossy compression algorithms are used to accommodate the storage of hash tables[11]. Explicit-state techniques are more suited for error detection and handling concurrency.

2.4.3 Symbolic Model Checking

Symbolic model checking methods manipulate a set of states rather than single states. Sets of states are represented by formulae in propositional logic. It can handle much larger designs with hundreds of state variables. Symbolic model checking uses different model checking algorithms: fix-point model checking(mainly for CTL), bounded model checking(mainly for LTL), invariant checking, etc,. Two main symbolic techniques used - Binary Decision Diagrams(BDD) and Propositional Satisfiability Checkers(SAT solvers). BDDs are traditionally used to represent boolean functions. A BDD is obtained from a Boolean decision tree by maximally sharing nodes and eliminating redundant nodes. However, BDDs grow very large. The issues in using finite automata for infinite sets are analogous. Symbolic representations such as propositional logic formulas are more memory efficient, at the cost of computation time. Symbolic techniques are suitable for proving correctness and handling state-space explosion due to program variables and data types.

2.4.4 Partial Order Reduction

Partial Order Reduction(POR) is a technique used for reducing the size of state space to be searched by a model checking algorithm[20]. This technique exploits the independence of concurrently executed events. Two events are independent of each other when executing them either order results in the same global state[8]. A common model for representing concurrent software is to have it depicted as interleaving model. In interleaving model, we have a single linear execution of the program arranged in an interleaved sequence. Concurrently executed events appear to be ordered arbitrarily to each other. Considering all interleaving sequences would lead to extremely large state space. Constructing full state graph would make the fitting into the memory difficult. Therefore, a reduced state graph construction is used in this technique.

POR exploits the commutativity of concurrently executed transitions, which would result in the same state. Fig 2.2 depicts the commutativity behavior. S, S_1, S_2 and R are various states of a given program and α_1, α_2 represents various transitions. Consider two paths P_1 and P_2. $P_1 = S \rightarrow S_1 \rightarrow R$ and $P_2 = S \rightarrow S_2 \rightarrow R$. P_1 and P_2 reaches the same final state R. Thus, showing us that commutativity of transitions α_1, α_2 on the given example.

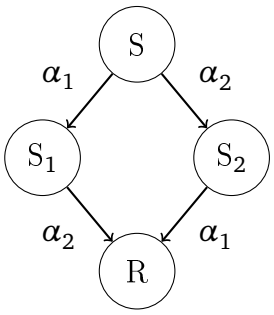


Figure 2.2.: Commutativity Example

Partial order reduction derives its motivation from the early versions of algorithms used for partial order modeling of program execution. POR is described as model checking using representatives[19]. Verification is performed using representatives from equivalence classes of behaviors.

The transitions of a system play a major role in the POR. POR is based on the dependency relation that exists between the transitions of a systems. A transition $\alpha \in T$ is enabled in a state s, if there is a state s' such that $\alpha(s,s')$ holds. Otherwise, α is disabled in s. The set of transitions enabled in s is $enabled(s)$. A transition α is deterministic, if for every state s there is at most one state s' such that $\alpha(s,s')$.

A path π from a state s_0 is a finite or infinite sequence.

$\pi = s_0 \rightarrow s_1 \rightarrow ...$

$\alpha_0(s_0,s_1)$, $\alpha_1(s_1,s_2)$ are transitions on the states in path π such that for every i, $\alpha_i(s_i,s_{i+1})$ holds. If π is finite, then the length of π is the number of transitions in π and will be denoted by $|\pi|$. Purpose of POR is to reduce the number of states, while preserving the correctness of the program. A reduced state graph is generated using depth-first or breadth-first search methods. Model checking algorithm is applied to the resultant graph, which has fewer states and edges.

An independence relation $I \subseteq T \times T$ is a symmetric, anti-reflexive relation such that for $s \in S$ and $(\alpha,\beta) \in I$:

- Enabledness If $\alpha,\beta \in enabled(s)$ then $\alpha \in enabled(\beta(s))$.
- Commutativity $\alpha,\beta \in enabled(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$.

The dependency relation D is the complement of I, namely $D = (T \times T) \setminus I$. The enabledness condition states that a pair of independent transitions do not disable one another. However, that it is possible for one to enable another. Stuttering refers to a sequence of identically labeled states along a path. In fig 2.2, we have two paths P_1 and P_2 which are stuttering equivalent. Thus, the reduced graph would have fewer number of states and retains the correctness property of the model.

Two main POR techniques which are commonly considered: persistent/stubborn sets and sleep sets. Persistent set technique computes a provably-sufficient subset of the set of enabled transitions in each visited state such that unselected enabled transitions are guaranteed not to interfere with the execution of those being selected. The selected set is called a persistent set. Whereas, the

most advanced algorithms are based on stubborn sets. These algorithms exploit information about "which communication objects in a process gets committed to in a given set of operations in future"[13]. Such an information is generally obtained from static code analysis. The sleep set technique exploits information on dependencies exclusively among transitions enabled in the current state along with information recorded about the past of the search. Both the techniques can be used simultaneously and are complementary. Unfortunately, existing persistent/stubborn set techniques suffer from a severe fundamental limitation in the context of concurrent software systems. The non-determinism in the execution of the concurrent programs makes the computation of precision difficult. Sleep sets could be used but, it cannot avoid state explosion. To overcome the above limitations, we have Dynamic POR, which is discussed further in the next section.

2.4.5 Dynamic POR

Dynamic POR is a technique, which dynamically tracks interactions between processes and then exploits this information to identify back tracking points where alternative paths in the state space need to be explored[13]. The algorithm works on depth first search in the reduced state space of the system. Dynamic POR helps to calculate dependencies dynamically during the exploration of the state space. It is able to adapt the exploration of the program's state graph to the precision of having another read or write operation accesses on the same memory location in the same execution path. Dynamic POR algorthm by Flanagan and Godefroid [13], explores single transitions and performs recursive calls subsequently. A persistent set is calculated at each state in the state graph of a system.

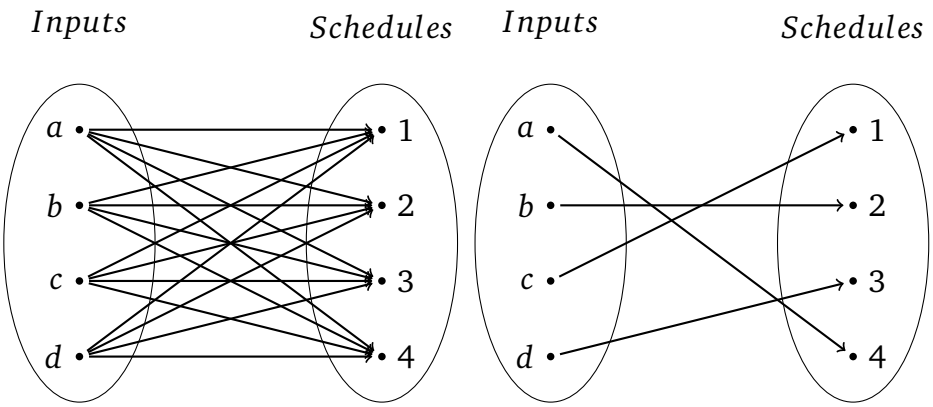
2.5 Deterministic Multi-Threading

In section 2.4, we primarily dealt with various ways to verify a program and suggested various techniques, which could be used in a multi-threaded environment. In this section, we discuss about a different approach to deal with the verification of multithreaded programs. In sections 2.2 and 2.3, we discussed about the non-determinism offered by multithreaded programming designs and the bugs associated with them. One way to detect and avoid bugs is to have a constrained scheduling of threads thus, adhering to deterministic execution. Deterministic multi-threading is an approach used to bring in determinism in the execution of multi-threaded programs.

Fig 2.3a depicts a mapping of inputs to possible scheduling pattern adopted by the multi-threaded program. By bringing in deterministic mapping as shown in fig 2.3b, we have direct mapping between inputs and schedules. Having such mapping provides us the opportunity to determine erroneous executions. Such a mapping facilitates to determine concurrency bugs in the program execution. There are many frameworks - CoreDet[4], Parrot[10], Kendo[18], DThreads[16], Grace[5], which adheres to this principle. Some of these frameworks are discussed further in the next chapter.

2.6 Iterative Relaxed Scheduling

Iterative relaxed scheduling is a program verification technique used for concurrent programs.



(a) Traditional Mapping (b) Deterministic Mapping

Figure 2.3.: Input to Schedule mappings in multithreaded execution based on the Cui et al. [10]

3 Related Work

3.1 COREDET

3.2 PARROT

3.3 KENDO

3.4 DTHREADS

3.5 GRACE

4 Approach

In this chapter, we address the approach used for realizing IRS in kernel space. In the first section, we discuss a theoretical design. In the later sections, we address the potential challenges related to its implementation and the implementation of the prototypes.

4.1 Theoretical Design

Figure 4.1 depicts the design overview of the IRS. Verifier and scheduler are the main components of IRS. Verifier is not an automated software. However, there are some changes in the representation of some components such as the trace/scheduling constraints which are discussed in section 4.1.1.

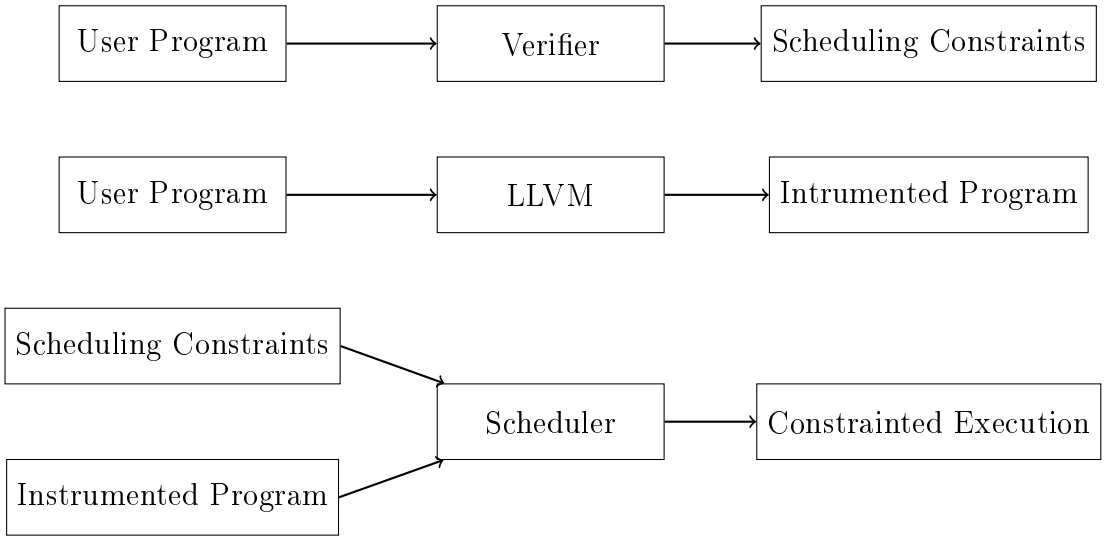


Figure 4.1.: IRS Design Overview

The key component of this thesis is the scheduler. Scheduler handles the scheduling of various user level threads based on their memory access permissions. Memory access permissions are perceived by traces. The traces are realized as simple graph with nodes. Each node denotes a shared memory event for a thread which can be a read or a write event.

Listing 4.1: Uninstrumented User Program

```
Shared Variable: x;
Thread j(j∈1..N):
......
x=0;    //shared memory access
....
```

Listing 4.2: Instrumented User Program

```
Shared Variable: x;
Thread j(j∈1..N):
......
BeforeMA();
x=0;    //shared memory access
AfterMA();
....
```

LLVM is another component part of the IRS framework. It primarily annotates the user program for shared memory events as shown in listing 4.1 and 4.2.

4.1.1 Vector Clock

Vector clock is an algorithmic design motivated from Lamport logical clocks [12]. It is used to detect causality violations and generating a partial ordering of events in a distributed system. A vector clock is an array of N logical clocks corresponding to N processes/threads. Vector clocks allow for the partial causal ordering of events. The following definition holds:

- $VC(x)$ denotes the vector clock of event x, and $VC(x)_a$ denotes the component of that clock for process a.
- $VC(x) < VC(y) \iff \forall a[VC(x)_a \lneq VC(y)_a] \wedge \exists b[VC(x)_b < VC(y)_b]$
- $x \rightarrow y$ indicates event x happened before event y. It is defined as : if $x \rightarrow y$, then $VC(x) < VC(y)$

The approaches discussed in this thesis uses a vector clock implementation for determining the number of memory events completed by a given thread, which is used to block or unblock a given thread. The trace file generated as graph representations are manually converted into strings of vector clock representations for using in this thesis. The vector clock representation only include the threads which are constrained. More details about the vector clock representation can be found in the appendix C.

4.1.2 Design classes

We have two different classes of approach used for this thesis. The two approaches can be classified as: design with proxy and design without proxy checking. Proxy checking referred above is the checking for memory access permission for the thread based on the constraints set in the trace.

Listing 4.3: Yield functionality and thread revival

```
yield(threadid j) {
        if(memory_access_permission(j)==restricted) {
                block_thread(j);
        }
}
reviveotherthreads() {
        for thread j(j ∈ 1..N) except j is not the current thread:
                if(memory_access_permission(j)==allowed) {
                        unblock_thread(j);
                }
        }
}
```

From figure 4.1, it is evident that we require scheduling constraints/traces and the instrumented program for executing with the scheduler. The IRS implementation which this thesis is based upon, uses an LLVM implementation for instrumenting the user program. When instrumenting the user program, LLVM inserts function calls to two methods namely $BeforeMA()$ and $AfterMA()$ which are inserted in places before and after every shared memory access in the program as shown in Listing 4.1 & 4.2. The instrumented program is later, run with the scheduling constraints on the scheduler. The decision to block the thread is handled by the thread itself by invoking the yield functionality. A thread would block itself when it has no permission to access the shared memory. The decision making of the thread and the yield functionality are the only concepts discussed with the prototypes in this thesis.

In this thesis, we propagate the decision making logic and yield functionality to kernel space as shown in Listing 4.3. However, in case of the second approach used in this thesis we have an additional proxy checking for memory permissions in user space. We expect these propogations to yield a good performance compared to the user space solutions.

Listing 4.4: First Approach

```
Thread j(j ∈ 1..N):
//on activating a BeforeMA call
BeforeMA() {
        yield(j);
}
AfterMA() {
        reviveotherthreads();
}
```

Listing 4.5: Second Approach

```
Thread j(j ∈ 1..N):
//on activating a BeforeMA call
BeforeMA() {
        if(memory_access_permission(j
           )==restricted) {
                yield(j);
        }
}
AfterMA() {
        reviveotherthreads();
}
```

First Approach

This approach has no proxy checking in the user space. When there is an occurrence of a shared memory event, this approach communicates to kernel space from the $BeforeMA()$ and $AfterMA()$ callback functions. These functions aid the logic of implementing yield functionality in the thread and also in reviving other threads. The pseudo implementation of this approach is depicted in listing 4.4. There are four design prototypes using this approach. Details related to these prototypes are discussed further in section 4.3.4.

Second Approach

This approach has a proxy checking in the user space. When there is an occurrence of a shared memory event, this approach initially checks for memory access permission in user space and based on the outcome of the check, it communicates to the kernel space. $BeforeMA()$ and $AfterMA()$ functions are used in the same way as in first approach. However, with the only difference of having a proxy check for memory access permission inside the $BeforeMA()$. The pseudo implementation of this approach is depicted in listing 4.5. There are two design prototypes using this approach. More details about these prototypes are discussed further in section 4.3.5. With the proxy checking in place, this design is expected to provide good performance when the following condition occurs: $num\_memory\_constraints << total\_memory\_events$. This approach reduces the number of unnecessary yield calls made to kernel space, which reduces drastic overhead generated by communication and waiting in kernel space. When the above condition fails, this approach would behave like the first approach, but would have higher overhead because of the proxy checking in user space. Under such scenarios, the prototypes under the first approach is expected to provide a good performance.

Note

In the progression of this document, we would be using certain acronyms to indicate certain meanings. Some of them are:

- UTID - User defined thread ID which is relative inside the user program.
- RTID - Real Thread ID which is assigned within the proc file system for any thread created within the user land.
- TaskID - All threads are internally realized as tasks in kernel space and are allocated with an identifier which is task ID.
- API -Application Programming Interface.

4.2 Design Challenges

In this section, we address some of the challenges we might face when moving the scheduling decisions of IRS to kernel space.

4.2.1 Mapping UTID to Task Object

UTID is passed to kernel space via a custom proc file and invoking scheduler API - *get_current_task*(). This function returns a task struct object. In kernel space, we can have a mapping of UTID to the obtained task struct object. User defined thread ID (UTID) is required to be communicated to the scheduler. And the mapping of task struct to UTID needs to be realized, in order for the scheduling to be done right. The custom registration proc file communicates the UTID to the kernel space. The user defined thread writes the UTID in the above proc file, which would trigger a callback to the write function in the kernel space module. Threads will be created based on the user's choice. On thread creation, the threads would invoke the registration module individually. This method would require a definition of synchronization block inside the kernel space since, multiple write function calls are invoked. Multiple threads are accessing the registration module. The synchronization is also required between the scheduler module and registration module.

4.2.2 Data Structure for mapping UTID to Task Object

The mapping of UTID - task object is realized, when the registration of a UTID to the scheduler is done. In the registration, the user thread is required to pass the UTID. The task object is obtained by invoking *get_current_task*() function during registration call. A data structure is created to store the mapping of UTID to task object. An item in the data structure is created whenever a registration of UTID takes place. An item is otherwise accessed during the invocation of *context_switch*() function. In a user space environment, there are solutions such as dictionary mapper or even hash table designs. Since the mapping is coherent in the kernel space, possible design choices include - linked list, arrays. There is a complexity associated in accessing a node in the linked list, which is O(n).

4.2.3 Communication between user thread and kernel space scheduler during context switch

With the transition of scheduler to kernel space, there is a need of having a communication design to interact between the user program and kernel space scheduler. The communication can be dealt in many ways[15]. Some of them are:

- ProcFS - Virtual file system for handling process and thread information base. Useful for small and short communications.
- Netlink - Special IPC scheme between kernel space and user space which uses sockets. However, extremely expensive when opening and closing of sockets.
- System call - Functional implementation mainly meant to communicate some data or perform a specific service in kernel space. It requires a static implementation in the kernel source tree. Extremely hard to develop and debug the implementation.
- CharacterDevice - Special buffering interface provided for communicating with character device driver setups.
- Mmap - Fastest way of copying data between kernel space and user space without explicit copying. Useful for large transactions of data.
- Signals - Unidirectional communication. Communicated from kernel space to user space.
- Upcall - Execute a certain function defined in the user space from kernel space.
- IOCTL - Used primarily for input and output operations in between user space and kernel space. It is an extension of character device implementation. It uses simple read and write system calls for communication purposes. It can be realized as an alternative for system call.

Assessing the requirements for the implementation, IOCTL seems to be a perfect fit for all the interactions required for a scheduler module. System call implementation requires the building of the entire kernel source tree and they are very difficult to debug and develop. IOCTL provides the possibility for a plug and play design.

4.2.4 Mapping the trace object to kernel space

The proposed design uses vector clocks as an outline for trace implementation. The traces generated as graphs are mapped to kernel space as a struct object. Graphs are realized as graphviz files. Parsing of graph is required before they are mapped to the kernel space. The parsed graph is passed to the kernel space as a long string via a custom proc file. Currently, there is no automated method existent in this thesis to generate a graph string from a graphviz file. We generate the trace string manually and pass it as an input to the custom proc file when the user program starts.

4.2.5 Trace verification inside user program vs kernel space scheduler

On occurrence of a shared memory event, the respective callbacks (BeforeMA() & AfterMA() - Before memory access and After memory access) from the user program would trigger an IOCTL call to the kernel module. Such a design would facilitate towards a non-preemptive scheduler. By overcoming the additional synchronization overhead existent in the user space design, we encounter the problem of invoking IOCTL calls for accessing the kernel module. In a monolithic kernel architecture, most of the IOCTL calls are blocking synchronous calls to the kernel space. Having too many IOCTL calls would increase the scheduler overhead on the program execution.

One solution is to make IOCTL calls when there is an actual need of a context switch. The user space threads would assess the trace based on which the IOCTL calls for the kernel space scheduler would be made. We discuss about such a solution in two prototypes used in the implementation section.

Note - Non-preemptiveness indicated in this section and the rest of the document is in regard to the scheduler implemented in this thesis and not the OS Scheduler.

4.2.6 Yield to scheduler vs Preemptive scheduler

The current implementation uses a non-preemptive design for the scheduler. The design uses the verification of memory access event and performs yield to scheduler when the access to memory is not permitted. A preemptive design would reduce the communication between user space and kernel space during context switch but, would increase the same for every memory access events. With such an implementation, it would require the kernel space to be able to detect the memory access events of the global memory used by the user space threads. Considering the complexity of its implementation and lack of existing solutions such a design would be not feasible to implement.

4.2.7 Vector clock design for finding the event in the trace

Before a shared memory access is made, the user thread triggers a callback - BeforeMA() (in short before memory access). The callback internally triggers a yield to scheduler if the memory access is not permitted. The memory access permission is determined by checking the trace object. The timeline of the event is required to be addressed during the checking with the trace. The event timeline can be determined by having a vector clock design. The same vector design needs to be used inside the kernel space as well, for its trace verification function.

4.3 Synchronization Designs

As discussed in the previous sections, we classify the designs into two classes. The classification is based on the checking for memory permission in user space. The first class has no checking for memory permission in the user space and the second has a proxy checking in user space for memory permission. Tables 4.1 and 4.2 depict this classification.

	Shared Thread	Separate Thread
Semaphore	Prototype 1	Prototype 2
Scheduler APIs	Prototype 3	Prototype 4

Table 4.1.: Prototypes without proxy checking

The yield functionality depicted in all the prototypes either use a semaphore based solution or scheduler APIs. To get a better understanding about semaphores and scheduler APIs please refer to appendix C.

The implementation of all prototypes comprise of three loadable kernel modules: thread registration, trace control and scheduler. The scheduler module is the only module implemented differently for all the prototypes.

	Shared Thread	Separate Thread
Semaphore		Prototype 5
Scheduler APIs		Prototype 6

Table 4.2.: Prototypes with proxy checking

4.3.1 Trace Registration

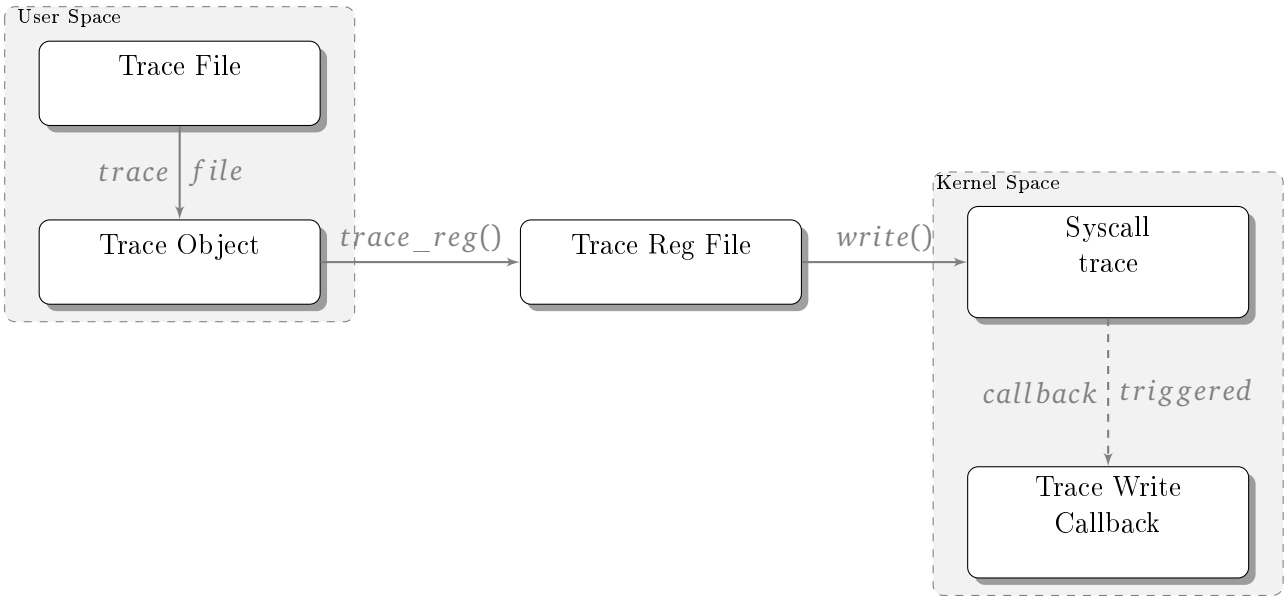


Figure 4.2.: Trace Registration

The trace file is passed on as an input for the scheduler. In figure 4.2, the trace file is read by the main user thread at the start of its execution. It parses the file, creates and passes the trace object to the kernel space as string via a custom file created in the proc file system. Trace registration is implemented within the trace control module. The trace control module deals with the manipulation of trace object in the kernel space. Trace control module is imported into the scheduler module for obtaining the trace control functionality and also to access the trace itself.

4.3.2 Thread Registration

In figure 4.3, the registration block happens when a user thread is created. The registration happens via a custom proc file system. For convenience a *thread_reg()* method is invoked in the user space, which internally registers the thread to the kernel space. The thread registration implementation is done entirely in thread registration kernel module. This registration is used to allocate memory space for bookkeeping the thread entry in scheduler.

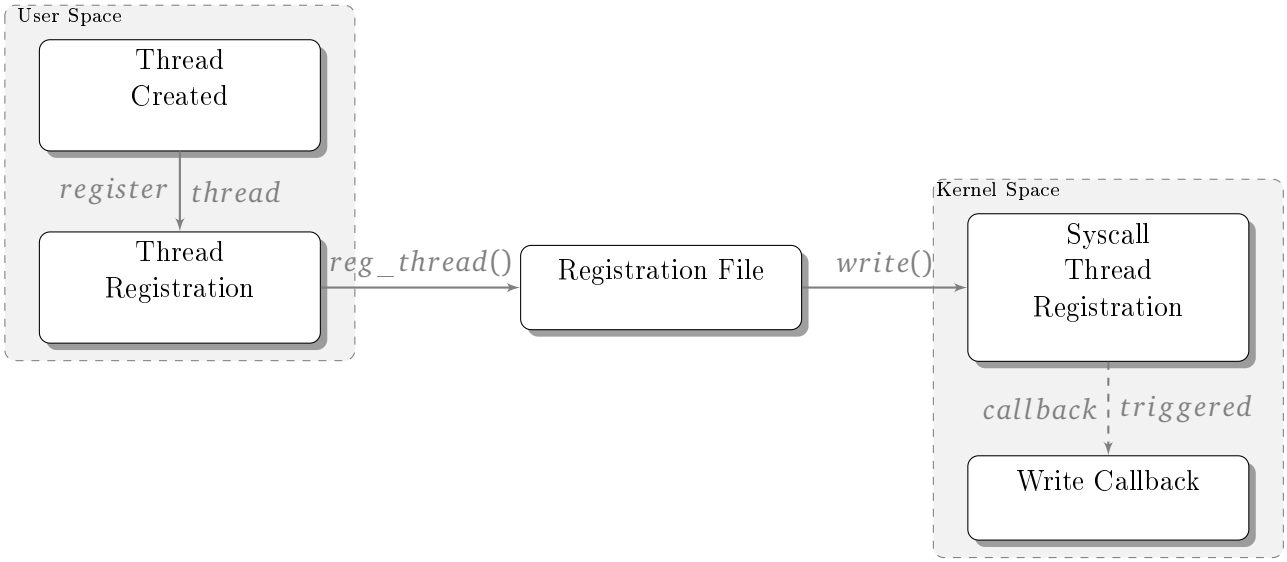


Figure 4.3.: Thread Registration

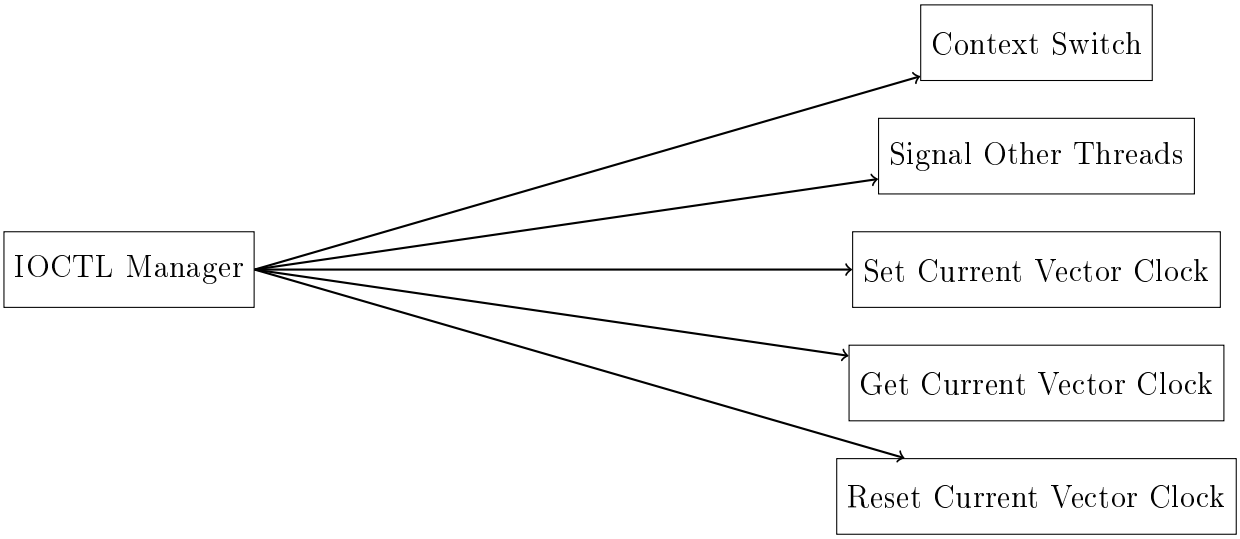


Figure 4.4.: IOCTL Manager

4.3.3 IOCTL Manager

IOCTL Manager is an abstraction layer in the kernel space used to contain any commands triggered from the user space. The manager acts as an interface for the commands requested by user space for kernel level services. It provides five different commands of operation as shown in figure 4.4. Among the five operations, three operations deal with the vector clock manipulation. The commands *context_switch* and *signal_other_threads* are generally very expensive commands when invoked. You can observe their implementation in later sections of this chapter. *context_switch* command is used by all prototypes when there is a need to yield to the scheduler. *signal_other_threads* command is used by prototypes 1 and 3, the threads signal among themselves. In case of prototypes other than 1 and 3, *set_clk* command is used more frequent for updating a given thread's memory event. Updating the vector clock is a short operation when compared to signaling other threads. The prototypes 1 and 3 are expected showcase poor performance when

encountered with the condition: $num\_memory\_constraints << total\_memory\_events$. These prototypes invoke *signal_other_threads* command for every shared memory event thus, having a poor performance. Reset vector clock is invoked at the completion of the user program for resetting the clock and using it for the next execution. Get current vector clock is used to obtain the entire vector clock during the time of invocation. This command is primarily used for debugging purposes.

Listing 4.6: Pseudo Code for checking memory access permission

```
mem_access = {allowed, restricted};
check_mem_acc_perm(curr_vec_clk, traceobj, thread_id tid) {
        if(clock[tid] is same in traceobj and curr_vec_clk) {
                ∀thread_id i in {{1...N}-{tid}}:
                if clock[i] in traceobj is greater to the one in curr_vec_clk) {
                        return restricted;
                }
        }
        else if(clock[tid] in traceobj is lower than one curr_vec_clk) {
                return restricted;
        }
        return allowed;
}
```

4.3.4 Design with no checking in user space

In the following designs, we address the use of check for memory access permission method entirely in kernel space. The pseudo implementation for the checking for memory access permission is depicted in Listing 4.6. All the prototypes in this thesis would be using the implementation shown in Listing 4.6 for checking the memory access permission.

Design with no additional scheduler thread

The design described in this section addresses the use of no additional scheduler thread. Prior to any global memory access, the given design would invoke IOCTL command with *context_switch* and thread id of the thread which addressed the memory event as its parameters. Prototypes 1 and 3 primarily works with a similar implementation. These prototypes defer in place of blocking and unblocking of the threads. Listing 4.7 depicts the pseudo implementation of prototype 1.

Listing 4.7: Pseudo Code for Prototype 1

```
User Space:
Thread j(j∈1..N):
BeforeMA() {
        ioctl(CONTEXT_SWITCH, thread_id);
}

AfterMA() {
        ioctl(SIGNAL_OTHER_THREADS, thread_id);
}
```

```
Kernel Space:
semaphore threads_sem[1..N] = {0...0};
Queue waitqueue={}
ctxt_switch_thread(thread_id tid) {
        if(check_perm(tid)==restricted) {
                signal_all_other_threads(tid);
                waitqueue.push(tid);
                down(threads_sem[tid]);
        }
}
signal_all_other_threads(thread_id tid) {
        ∀ thread_id i in {{1...N}-{tid}}:
                if(i in waitqueue and check_perm(i)==allowed) {
                        waitqueue.remove(i);
                        up(threads_sem[i]);
                }
}
```

Design with an additional scheduler thread

In this design, we have an additional scheduler thread which addresses the signaling mechanism pertained in the previous design. By having an additional scheduler thread, we move the entire signaling system to the scheduler thread. Thus, reducing the execution overhead encountered in the pool of user space threads for signaling other threads. The major changes are in kernel space code. However, there are minor variations in the $AfterMA()$ in user space. Inside the $AfterMA()$ call, we invoke the set_clk command. It is used to update the memory event for the thread which encountered the memory event. Prototypes 2 and 4 follow similar implementation. However, they defer in the blocking and unblocking mechanism of the thread.

Listing 4.8: Pseudo Code for Prototype 2

```
User Space:
Thread j(j ∈ 1..N):
BeforeMA() {
        ioctl(CONTEXT_SWITCH, thread_id);
}

AfterMA() {
        ioctl(SET_CLK, thread_id);
}

Kernel Space:
semaphore threads_sem[1..N] = {0...0};
Queue waitqueue={};

Run a kernel level thread every 1ms which invokes signal_permitted_threads
    function.

ctxt_switch_thread(thread_id tid) {
        if(check_perm(tid)==restricted) {
                signal_all_other_threads(tid);
```

```
                waitqueue.push(tid);
                down(threads_sem[tid]);
        }
}
set_clk(thread_id tid) {
        vec_clk[tid]++;
}
signal_permitted_threads() {
        ∀ thread_id i in {1...N}:
                if(i in waitqueue and check_perm(i)==allowed) {
                        waitqueue.remove(i);
                        up(threads_sem[i]);
                }
}
```

4.3.5 Design with proxy checking in user space

In the following designs, we address the use of checking for memory access permission both in user space and kernel space.

Design with no additional scheduler thread

Without an additional thread in kernel space, the design would require a signaling function inside $AfterMA()$, similar to the one used in Design 4.3.4 with no additional scheduler thread. Triggering a signaling mechanism is an additional overhead on the thread calling the $AfterMA()$. Therefore, such a design is not a wise choice when considering the performance metrics such as execution time.

Design with an additional scheduler thread

The scheduler implementation is similar to one defined in the section 4.3.4 with an additional scheduler thread. Key difference is the additional checking for memory access permissions in the user space. The major changes are in user space code. Prototypes 5 and 6 use a similar setup as shown in Listing 4.9. However, they differ in their blocking and unblocking mechanism of threads. As explained in the previous sections, this design is expected to perform better under the following condition: $num\_memory\_constraints << total\_memory\_events$.

Listing 4.9: Pseudo Code for Prototype 5 and 6

```
User Space:
Thread j(j ∈ 1..N):
BeforeMA() {
        if(check_perm(j)==restricted) {
                ioctl(CONTEXT_SWITCH, thread_id);
        }
}

AfterMA() {
```

```
        ioctl(SET_CLK, thread_id);
}
//Rest of the code remains the same.
```

4.3.6 Variant in blocking implementation

In the previous designs, the blocking was done using semaphores. In the variant design, we use the combination of *schedule*() and *wake_up_process*() functions provided by the Linux scheduler APIs. The kernel level tasks associated for the provided user level threads are moved from running queue to wait queue by initially setting the task status as *TASK_INTERRUPTIBLE* and yielding the processor by invoking *schedule*(). The task added in wait queue is later resumed, when *wake_up_process*(*sleeping_task*) is invoked by another task(primarily another thread). On calling the *wake_up_process*(*sleeping_task*), the task status for *sleeping_task* is set as *TASK_RUNNING*. It would be pushed to run queue and executed in future by the operating system scheduler on the basis of scheduler class and priority of tasks in run queue. Prototype 3,4 and 6 utilizes this design. Listing 4.10 and 4.11 depicts Prototypes 3 and 4.

Listing 4.10: Pseudo Code for Prototype 3

```
//rest remains the same.
Kernel Space:
Queue waitqueue={}
ctxt_switch_thread(thread_id tid) {
        if(check_perm(tid)==restricted) {
                signal_all_other_threads(tid);
                waitqueue.push(tid);
                set_current_task_state(TASK_WAIT);
                schedule();
        }
}
signal_all_other_threads(thread_id tid) {
        ∀ thread_id i in {{1...N}-{tid}}:
                if(i in waitqueue and check_perm(i)==allowed) {
                        waitqueue.remove(i);
                        wakeup_process(taskforthreadid(i));
                }
}
```

Listing 4.11: Pseudo Code for Prototype 4

```
//rest remains the same.
Kernel Space:
Queue waitqueue={}
ctxt_switch_thread(thread_id tid) {
        if(check_perm(tid)==restricted) {
                signal_all_other_threads(tid);
                waitqueue.push(tid);
                set_current_task_state(TASK_WAIT);
                schedule();
        }
```

```
}
signal_permitted_threads() {
        ∀ thread_id j in {1...N}:
                if(j in waitqueue and check_perm(j)==allowed) {
                        waitqueue.remove(j);
                        wakeup_process(taskforthreadid(j));
                }
}
//rest remains the same.
```

5 Evaluation

In this chapter, we provide an evaluation proof of using IRS in kernel space. Experiments are setup for understanding the behavior of the six prototypes proposed in this thesis. Additionally, they are also used to provide a comparison with the user space implementation of IRS.

5.1 Setup

The evaluation is performed with a virtual machine running on a hardware with maximum of 16 cores. The virtual machine can use from two to eight processor cores which is later used for the scaled up evaluation. The virtual machine is based on Intel Xeon E5-2650 - 2.00 GHz configured with Ubuntu 17.04 as the operating system. It is configured with 4GB RAM and 80GB hard disk. The virtual machine is configured with the LLVM-CLANG 3.9, GCC 4.9 and Boost 1.6.2.

Note

We would be using the following abbreviations in the rest of the evaluations for simplicity of expression.

- IRS_Sh - IRS user space implementation using shared scheduler design.
- IRS_Opt - IRS user space implementation using additional scheduler with conditional variable design.
- Proto_1 - Prototype 1 discussed in the previous chapter.
- Proto_2 - Prototype 2 discussed in the previous chapter.
- Proto_3 - Prototype 3 discussed in the previous chapter.
- Proto_4 - Prototype 4 discussed in the previous chapter.
- Proto_5 - Prototype 5 discussed in the previous chapter.
- Proto_6 - Prototype 6 discussed in the previous chapter.

5.2 Evaluation Metrics

5.2.1 Execution Overhead

Evaluation is done between the IRS user space solutions vs kernel space solutions. Execution overhead is calculated for each solution with respect to the plain execution of the bench-marking program. Unconstrained execution of the program is considered as plain execution.

$$ExecutionOverhead = (T_{constr} - T_{plain})/T_{plain} * 100$$

T_{constr} is the execution time of the bench-marking program when executed with the scheduling constraints. T_{plain} is the plain execution time of the same bench-marking program. We expect to monitor the performance of various IRS implementations by checking this metric. If execution overhead for a certain IRS design is the smallest, that design is considered to give the best performance.

5.2.2 Number of valid synchronization calls

It is used to realize the number of voluntary calls made to kernel space for synchronization purposes. It is primarily the number of IOCTL calls made under the command - context_switch, signal_all_other_threads or set_clock. We also find the number of voluntary context switch calls made to kernel space to determine the prototypes which can provide a smaller overhead. We observe the use of this metric more in this section 5.4.

5.3 Benchmarks

We use four different bench-marking programs for the evaluation of this thesis. The bench-marking programs include:

- Fibonacci - Program runs with two threads computing Fibonacci numbers for 25 iterations per thread.
- Last Zero - Program runs with 16 threads [1].
- Indexer- Program runs with 15 threads [13].
- Dining Philosophers Problem - Program runs with 16 threads. This benchmark is motivated from the solution presented in Silberschatz et al. [21].

5.4 Voluntary kernel level calls

We evaluate the number of voluntary calls made to kernel space for synchronization. The evaluation is done across all six prototypes. The benchmark used for the evaluation is Fibonacci. The Fibonacci benchmark presents different levels of memory constraints via its traces. It has three traces providing 98 constraints, 44 constraints and 24 constraints respectively.

	Prototype 1-4	Prototype 5-6
Trace-1	300	175
Trace-2	300	150
Trace-3	300	150

Table 5.1.: Number of IOCTL calls

From the tables 5.1 and 5.2, it is really evident that prototypes 5 and 6 reduce the number of calls made to kernel space. Prototypes 5 & 6 are expected to provide better performance compared to other prototypes, when there are less dependencies between threads.

	Prototype 1-4	Prototype 5-6
Trace-1	150	27
Trace-2	150	0
Trace-3	150	0

Table 5.2.: Number of context switch calls

Let us consider the Fibonacci benchmark, it has two threads with a total of 75 shared-memory events per thread. Thus, making a total of 150 memory events. For every shared-memory event, prototypes 1-4 trigger IOCTL calls to kernel space for context_switch, signal_all_other_threads or set_clock. Therefore, having a total number of IOCTL calls as 300. In case of prototype 5-6, we have a proxy checking in user space which drastically reduces the calls to kernel space for additional synchronization. The set_clock ioctl command is the only call made consistently for every memory access when using prototypes 5-6.

	Proto-1	Proto-2	Proto-3	Proto-4	Proto-5	Proto-6
Trace-1	406.833	454.785	385.416	455.745	277.793	275.343
Trace-2	367.199	520.352	352.506	509.843	160.266	160.307
Trace-3	351.029	416.653	333.704	412.206	152.425	153.06

Table 5.3.: Execution overhead(%) when compared with plain execution of Fibonacci

Table 5.3 presents the reasoning of using prototypes 5-6. It shows the execution overhead is drastically reduced for the above mentioned prototypes. Reduction in the number of IOCTL calls makes a huge difference in the execution overhead. Prototypes 5-6 performs the best, when the following condition holds: $num\_memory\_constraints << total\_memory\_events$.

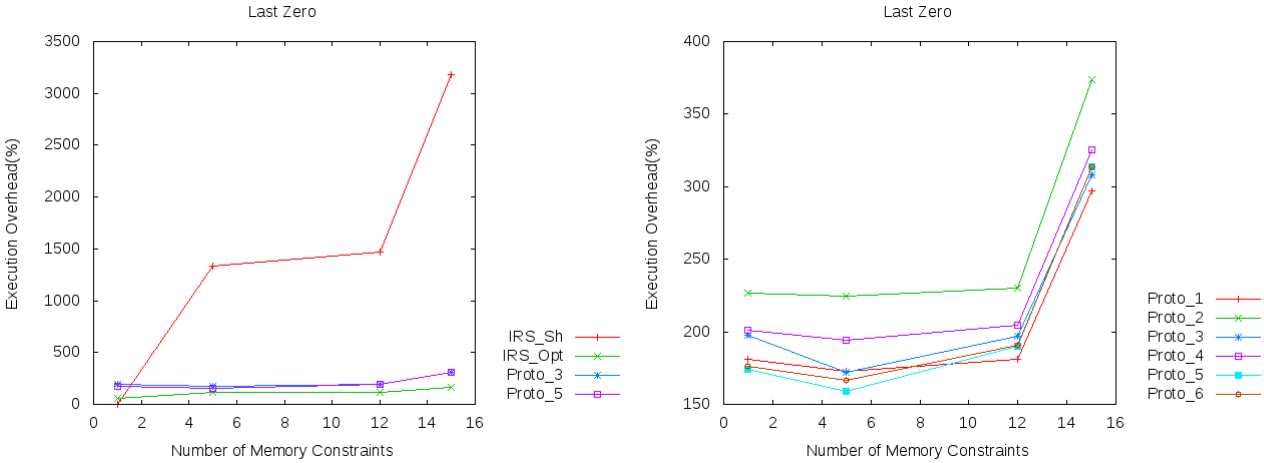
5.5 Scaled-up Evaluation

In this evaluation, we understand the merits and demerits in the performance of the six prototypes and the two user space IRS implementations. For this evaluation, we use three bench-marking programs - last zero, indexer and dining philosophers problem. We scale the core count from two to eight processor cores and monitor the changes in the performance overhead across the three benchmarks for the various IRS implementations. For comparing the designs with IRS user space implementations, we have chosen the best prototype among the six for comparison.

5.5.1 Last Zero

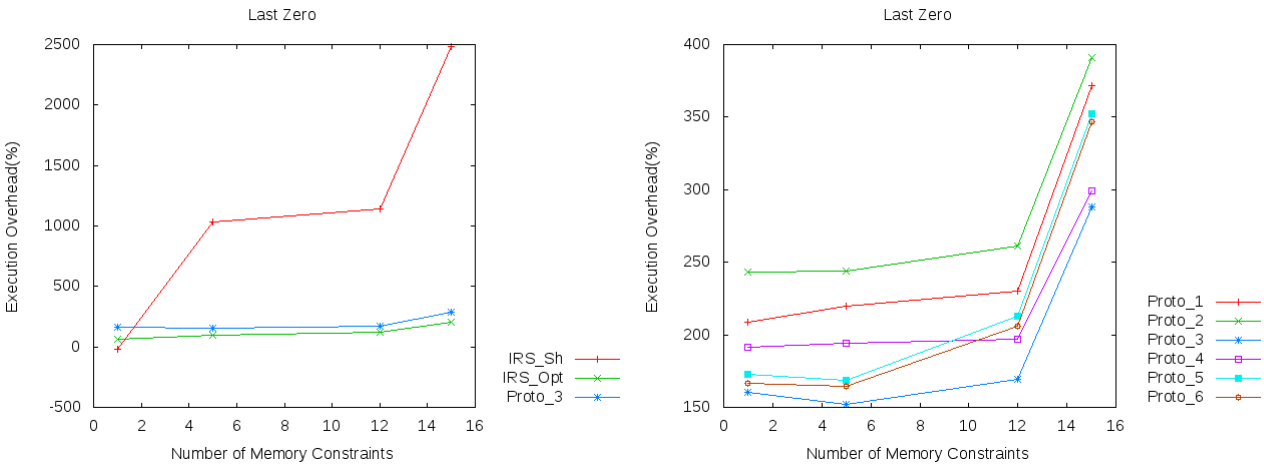
Abdulla et al. [1] showcases this benchmark for the evaluation of their dynamic POR. The last zero program has 16 threads at its disposal. The pseudo code of this benchmark is depicted in Fig A.1. This benchmark is meant to provide memory constraints spread across different threads rather than being in two threads. The number of memory constraints provided in the trace files include: 15, 12, 5, 1 respectively. The maximum number of possible shared-memory events is 46.

The benchmark has 16 threads and the memory constraints are spread across threads. Thus, making it expensive in regard to execution time. IRS_Sh is expected to be the worst of all designs considering the fact that it is a busy waiting design. Busy waiting design makes the waiting thread to constantly poll one of the cores thus, making it performance inefficient. However, it is expected to improve the execution overhead with the scaling of cores. IRS_Opt is designed using conditional variables and it is expected to perform better than IRS_Sh for scaled evaluations. IRS_Opt is expected to provide a better performance among all the designs because the $num\_memory\_constraints << total\_memory\_events$. Proto_5 and Proto_6 are also



(a) User space vs Best Prototype (b) Comparison between prototypes

Figure 5.1.: Comparison of IRS with Last Zero on two cores

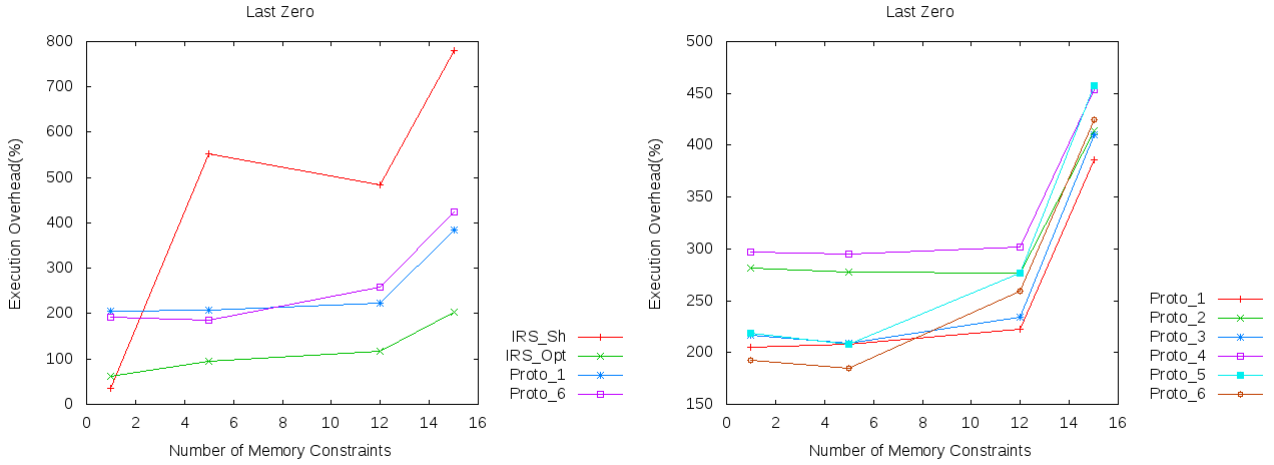


(a) User space vs Best Prototype (b) Comparison between prototypes

Figure 5.2.: Comparison of IRS with Last Zero on four cores

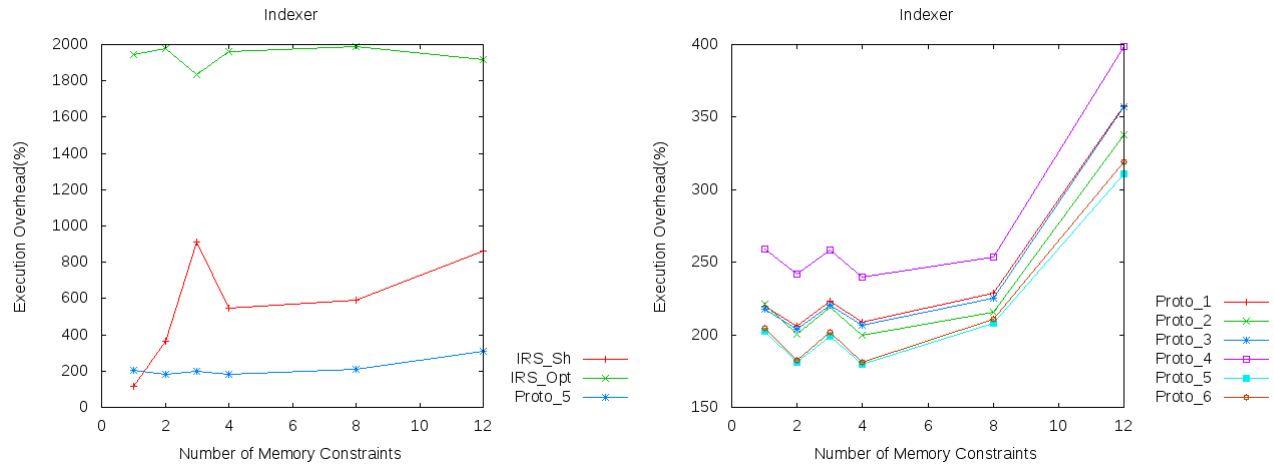
expected to provide better results with this benchmark, since the above condition holds good for these prototypes.

From Fig 5.1b, it is evident that Proto_5 and Proto_6 performs nearly the same when it comes to execution overhead. Surprisingly, Proto_3 performs better in these evaluations because they are expected to provide good performance when the number of memory constraints are higher in the bench-marking program. It is nearly true in case of evaluations with four cores and eight cores from Fig 5.2b and Fig 5.3b. From Figures {5.1a, 5.2a, 5.3a}, it is evident that IRS_Sh is the worst implementation for this benchmark among all the other implementations. From the above figures, the validity of improvement in execution overhead with scaling cores for IRS_Sh holds.



(a) User space vs Best Prototype (b) Comparison between prototypes

Figure 5.3.: Comparison of IRS with Last Zero on eight cores



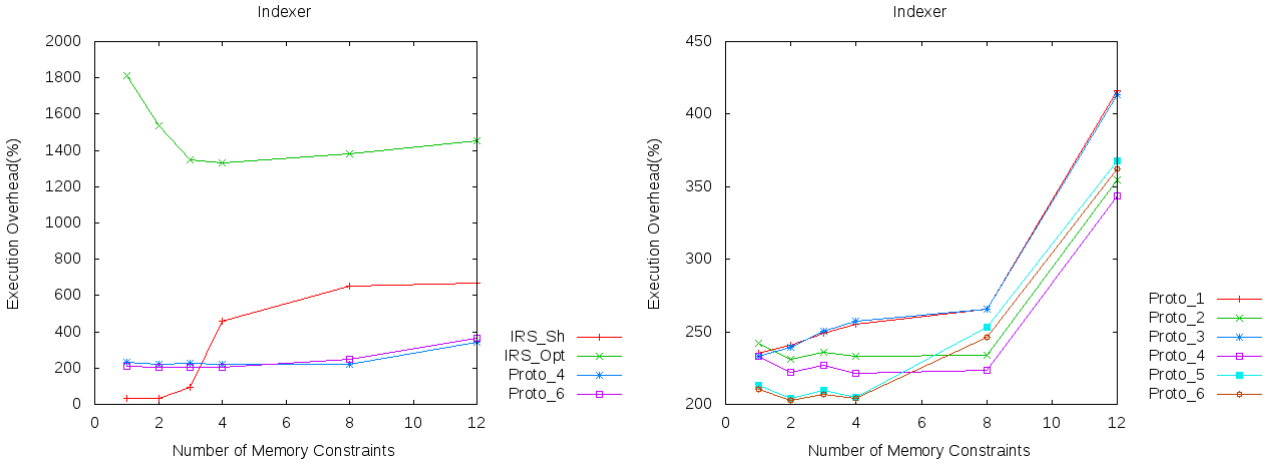
(a) User space vs Best Prototype (b) Comparison between prototypes

Figure 5.4.: Comparison of IRS with Indexer on two cores

5.5.2 Indexer

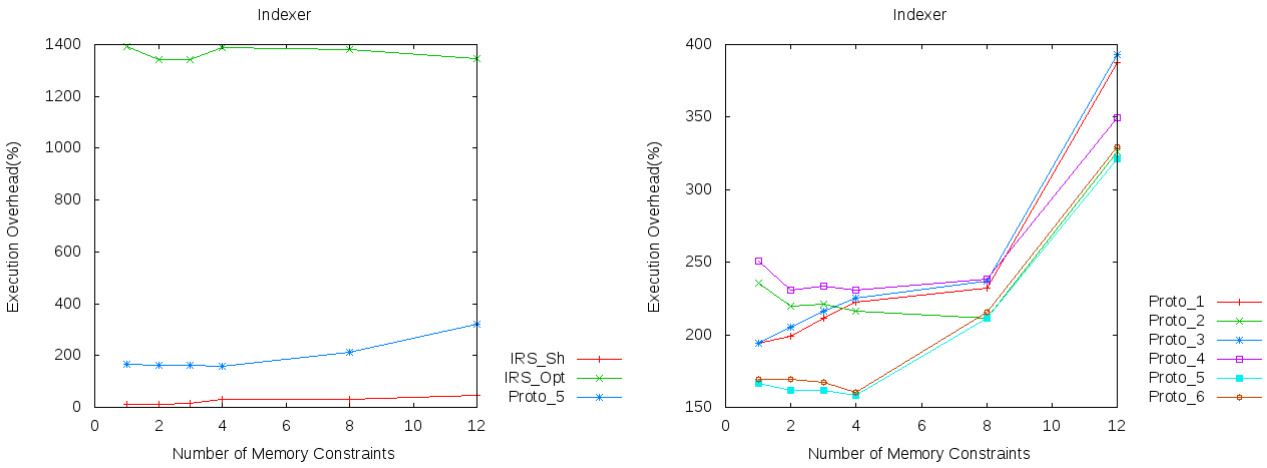
Flanagan and Godefroid [13] utilizes this benchmark for evaluating their dynamic POR design. The pseudo code for the indexer program is realized in Fig A.2. The indexer program revolves around a hash table, where threads read and write hashed messages on it. Each thread calculates four messages and writes them to a shared hash table. Collision are detected and avoided using compare and swap statement. The message values depend on the thread id.

For our experiments, we have used 15 threads with the indexer program. The benchmark contains approximately 60 shared-memory events in total. Six traces are used with the following as the number of memory constraints: 12, 8, 4, 3, 2, 1. The memory constraints are set in a way that most of the constraints are within a span of few threads and not all 15 threads. We expect to have good performance for IRS_Sh because the number of constraints are not spread across all 15 threads. We expect it to perform the best when the core count is 8, even when the constraints



(a) User space vs Best Prototype (b) Comparison between prototypes

Figure 5.5.: Comparison of IRS with Indexer on four cores



(a) User space vs Best Prototype (b) Comparison between prototypes

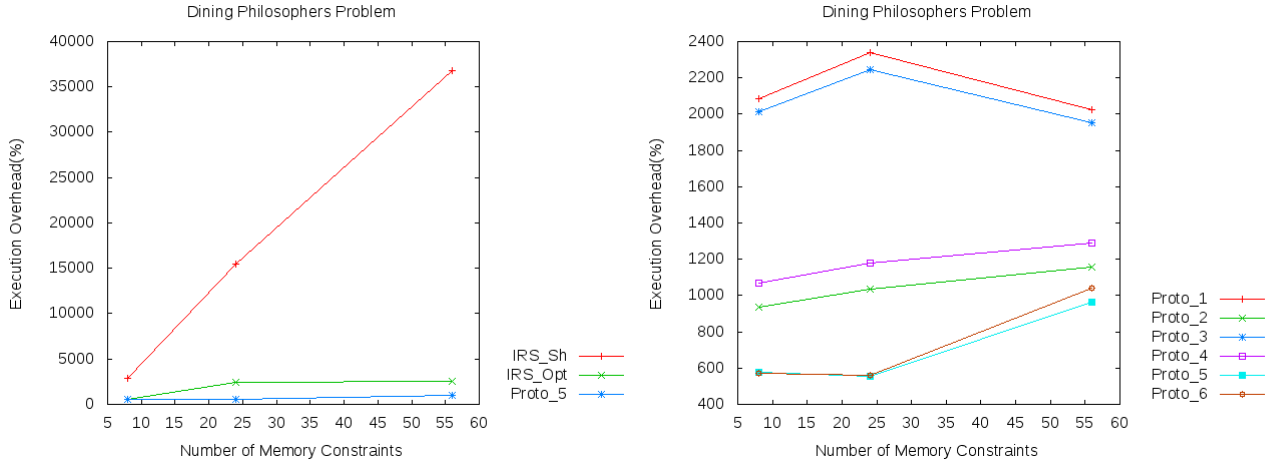
Figure 5.6.: Comparison of IRS with Indexer on eight cores

are set at 12. Proto_5 and Proto_6 are expected perform better for this benchmark, since the condition $num\_memory\_constraints << total\_memory\_events$ holds.

From Figs {5.4b, 5.5b, 5.6b}, it is evident that Proto_5 and Proto_6 performs the best in almost all scenarios. Thus, maintaining the above condition. Based on the analysis of Figs {5.4a, 5.5a, 5.6a}, we observe a trend in the improvement of execution overhead in IRS_Sh. As expected IRS_Sh seems to give the best performance when the number of cores is eight as observed in Figure 5.6a. IRS_Opt showcases a huge overhead for this benchmark, because of the lack of diversity in the memory constraints(spread of memory constraints across all threads).

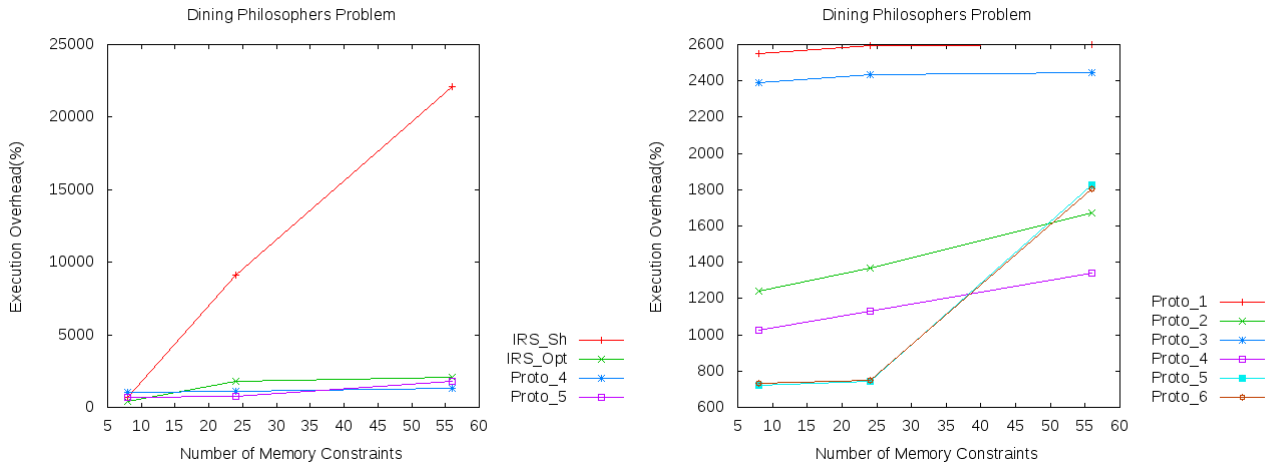
5.5.3 Dining Philosopher's Problem

Dining Philosopher's Problem is a well known synchronization problem in the domain of concurrency problems. There are many solutions adhered to overcome the problem. Silberschatz et al.



(a) User space vs Best Prototype (b) Comparison between prototypes

Figure 5.7.: Comparison of IRS with Dining Philosophers Problem on two cores



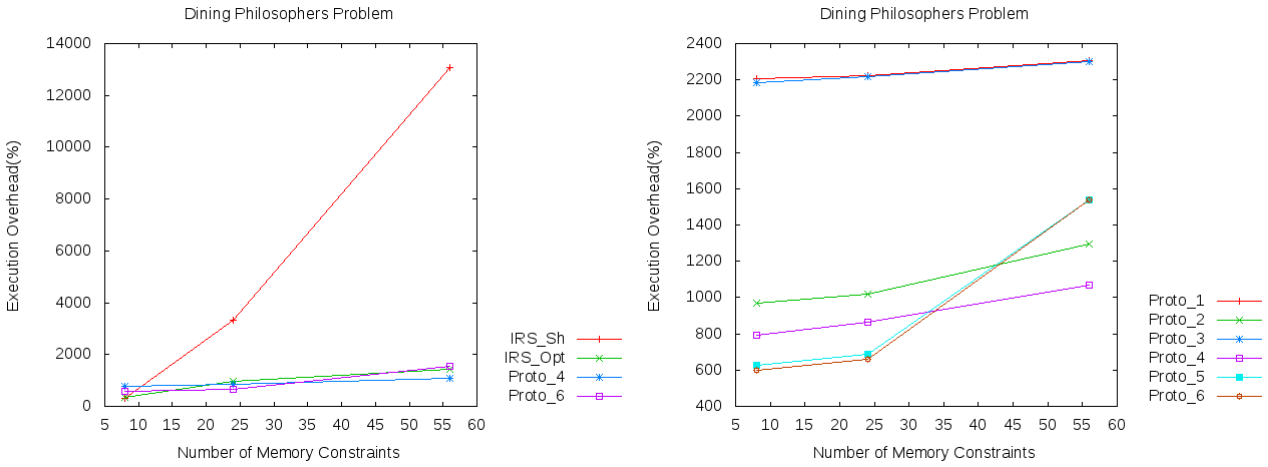
(a) User space vs Best Prototype (b) Comparison between prototypes

Figure 5.8.: Comparison of IRS with Dining Philosophers Problem on four cores

[21] have addressed many solutions in their book for the above problem. We are using one of the solutions proposed in their book. The solution uses two classes of philosophers - odd and even philosopher. The classification is based on their thread id. Every odd philosopher checks the chopstick on the left before checking on the right for availability. The opposite in case of even philosopher. The solution for this problem is showcased in Fig A.3.

In our experiments, we have adapted the solution to have 16 threads and number of iterations as 10. Compared to previous benchmarks which had only few number of constraints. This benchmark has more constraints spread across all 16 threads. This benchmark is expected to run in milliseconds time range rather than microseconds which was evident with respect to the previous benchmarks. For getting a detailed understanding of the experimental results, please refer to appendix B.

There are nearly 60 shared-memory events per thread. Thus, making a total of 960 shared-memory events in total. The benchmark has at-most eight threads running at any point of time.



(a) User space vs Best Prototype (b) Comparison between prototypes

Figure 5.9.: Comparison of IRS with Dining Philosophers Problem on eight cores

The number of memory constraints listed for the three traces include: 56, 24, 8. The number of memory constraints is too small compared to the total number of shared memory events in the bench-marking program. Proto_1 and Proto_3 are kernel space solutions implemented with a shared scheduler in place. Because of constant kernel synchronization calls during shared-memory events, Proto 1-4 are expected to provide a high overhead. However, the Proto_2 and Proto_4 uses a separate thread for scheduling, the IOCTL call made from $AfterMA()$ only increments the vector clock. Whereas in case of Proto_1 and Proto_3, we have calls made from $AfterMA()$ which performs signalling of other threads. Signalling of other threads is a costly operation compared to updating a vector clock. Thus, making Proto_1 and Proto_3 to perform the worst among the prototypes. The condition $num\_memory\_constraints << total\_memory\_events$ holds for this benchmark, we expect the Proto_5 and Proto_6 to provide the best performance among the prototypes. IRS_Opt is expected to have a good performance since the above condition holds true for the design. IRS_Sh is expected to give away a very poor performance because of the diversity of constraints.

From Figs {5.7b, 5.8b, 5.9b}, it is evident that Proto_5 and Proto_6 performs the best in all the scenarios. Proto_1 and Proto_3 as expected came up with the worst performance among the prototypes. In case of the comparison with the user space implementation, IRS_Opt seems have smaller overhead compared to IRS_Sh. From Figs {5.7a, 5.8a, 5.9a}, it is evident that IRS_Sh seems to give away the worst performance for this benchmark because of its busy waiting design.

5.6 Summary

The Fibonacci bench-mark helped us in understanding the behavior of prototypes. The scaled up experiments clearly explained the merits and demerits of various IRS designs. The scaled up experiments show that Proto_5 and Proto_6 seem to perform the best among the prototypes in nearly all the scenarios given in the bench-marking programs. IRS_Sh implementation seems to give away the worst performance in nearly all benchmarks. From the experiments, it is evident that Prototypes 1-4 suffer from the problem of scalability. The performance of user space imple-

mentation overshadows the kernel space implementation when the number of memory constraints are in single digits.

By observing the above results we can conclude that we need to have a solution which is combination of IRS_Opt and Prototypes 5-6. This implementation would have a loadable kernel module similar to Proto_5 and Proto_6 however, without any checking for memory permissions in kernel space for the threads. The kernel module would help in providing as a forced yield of processor to the OS Scheduler from the thread and also an interface to revive the thread. The check for memory permission would be implemented entirely in user space and calls would be made to the above mentioned kernel space solution for yielding the processor. In short this kernel space interface would be used instead of using a condition variables in IRS_Opt.

6 Conclusion

—conclusion comes here—

Bibliography

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. ACM SIGPLAN Notices, 49(1):373–384, 2014.

[2] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. Principles of model checking. MIT press, 2008.

[3] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. Systems and software verification: model-checking techniques and tools. Springer Science & Business Media, 2013.

[4] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In ACM SIGARCH Computer Architecture News, volume 38, pages 53–64. ACM, 2010.

[5] Emery D Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In ACM sigplan notices, volume 44, pages 81–96. ACM, 2009.

[6] Richard H Carver and Kuo-Chung Tai. Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs. John Wiley & Sons, 2005.

[7] Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. Concurrent software verification with states, events, and deadlocks. Formal Aspects of Computing, 17(4):461–483, 2005.

[8] Edmund M Clarke, Orna Grumberg, and Doron Peled. Model checking. MIT press, 1999.

[9] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. ACM Computing Surveys (CSUR), 3(2):67–78, 1971.

[10] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A Gibson, and Randal E Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 388–405. ACM, 2013.

[11] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27(7):1165–1178, 2008.

[12] Colin Fidge. Logical time in distributed computing systems. Computer, 24(8):28–33, 1991.

[13] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In ACM Sigplan Notices, volume 40, pages 110–121. ACM, 2005.

[14] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. Fundamentals of software engineering. Prentice Hall PTR, 2002.

[15] Ariane Keller. Tldp - communication between user space and kernel space. URL `http://wiki.tldp.org/kernel_user_space_howto`.

[16] Tongping Liu, Charlie Curtsinger, and Emery D Berger. Dthreads: efficient deterministic multithreading. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pages 327–336. ACM, 2011.

[17] Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. A study of concurrency bugs and advanced development support for actor-based programs. arXiv preprint arXiv:1706.07372, 2017.

[18] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. ACM Sigplan Notices, 44(3):97–108, 2009.

[19] Doron Peled. All from one, one for all: on model checking using representatives. In International Conference on Computer Aided Verification, pages 409–423. Springer, 1993.

[20] Doron Peled. Ten years of partial order reduction. In Computer Aided Verification, pages 17–28. Springer, 1998.

[21] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating system concepts essentials. John Wiley & Sons, Inc., 2014.

Appendices

A Benchmarks

A.1 Last Zero

```
Variables: int arr[0...N] := {0,0..,0}, i;
Thread 0: for (i:=N; array[i]!=0; i--);
Thread j(j∈1..N): arr[j] := arr[j-1] + 1;
```

Figure A.1.: Last Zero Program based on Abdulla et al. [1]

A.2 Indexer

```
Thread-global (shared) variables:
const int size = 128;
const int max = 4;
int[size] table;

Thread-local variables:
int m = 0, w, h;

Code For thread tid:
while (true) {
        w := getmsg();
        h := hash(w);
        while (cas(table[h],0,w) == false) {
                h := (h+1) % size;
        }
}
int getmsg() {
        if (m < max ) {
                return (++m) * 11 + tid;
        } else {
                exit(); // terminate
        }
}
int hash(int w) {
        return (w * 7) % size;
}
```

Figure A.2.: Indexer Program based on Flanagan and Godefroid [13]

A.3 Dining Philosopher's Problem

```
Thread-global (shared) variables:
const int size = THREAD_COUNT;
const int num_iter = N;
int[size] chopsticks;

Thread-local variables:
int i;

Code For thread id:
for(i=0; i<num_iter; i++) {
        while((chopsticks[id%THREAD_COUNT]!=0) && \\
         (chopsticks[(id-1)%THREAD_COUNT]!=0);
        if(id%2==0) {
                chopsticks[id%THREAD_COUNT] = 1;
                chopsticks[(id-1)%THREAD_COUNT] = 1;
        }
        else {
                chopsticks[(id-1)%THREAD_COUNT] = 1;
                chopsticks[id%THREAD_COUNT] = 1;
        }
        chopsticks[id%THREAD_COUNT] = 0;
        chopsticks[(id-1)%THREAD_COUNT] = 0;
}
```

Figure A.3.: Dining Philosopher's Problem Program

B Experimental Results

B.1 Last Zero

B.1.1 Two Cores

	IRS_Sh	IRS_Opt	Proto-1	Proto-2	Proto-3	Proto-4	Proto-5	Proto-6
Trace-1	3183.115	168.069	297.304	373.623	308.218	325.577	313.979	313.569
Trace-2	1472.212	119.385	180.815	230.234	196.77	204.881	190.245	190.703
Trace-3	1335.806	117.766	172.643	224.606	172.096	194.399	159.221	166.818
Trace-4	2.373	59.937	181.197	226.441	197.909	201.305	174.226	176.522

Table B.1.: Execution overhead(%) when compared with plain execution of Last Zero

B.1.2 Four Cores

	IRS_Sh	IRS_Opt	Proto-1	Proto-2	Proto-3	Proto-4	Proto-5	Proto-6
Trace-1	2485.820	204.205	371.688	390.946	288.458	299.136	352.202	346.665
Trace-2	1138.293	117.764	229.857	260.937	169.108	196.685	212.92	206.062
Trace-3	1030.858	95.448	219.563	243.64	152.027	193.9	168.338	164.249
Trace-4	-19.370	62.22	208.598	243.083	160.428	191.643	172.793	166.349

Table B.2.: Execution overhead(%) when compared with plain execution of Last Zero

B.1.3 Eight Cores

	IRS_Sh	IRS_Opt	Proto-1	Proto-2	Proto-3	Proto-4	Proto-5	Proto-6
Trace-1	779.417	204.205	385.556	413.958	409.856	453.227	457.043	424.486
Trace-2	483.096	117.764	222.77	276.71	234.497	301.796	276.798	259.555
Trace-3	553.460	95.448	207.627	277.427	209.038	294.96	207.559	184.733
Trace-4	36.206	62.22	204.648	281.475	216.443	296.927	218.399	192.43

Table B.3.: Execution overhead(%) when compared with plain execution of Last Zero

B.2 Indexer

B.2.1 Two Cores

	IRS_Sh	IRS_Opt	Proto-1	Proto-2	Proto-3	Proto-4	Proto-5	Proto-6
Trace-1	863.217	1914.739	357.769	337.66	356.85	398.296	310.709	319.146
Trace-2	591.346	1990.388	228.661	215.401	225.404	253.299	208.094	210.911
Trace-3	549.424	1961.126	208.925	199.812	206.814	239.773	179.803	181.235
Trace-4	913.520	1835.159	223.274	219.309	220.361	258.238	198.842	201.541
Trace-5	366.838	1979.13	206.164	200.645	203.99	241.717	181.155	182.518
Trace-6	114.149	1942.555	219.88	220.863	217.671	259.263	202.801	204.868

Table B.4.: Execution overhead(%) when compared with plain execution of Indexer

B.2.2 Four Cores

	IRS_Sh	IRS_Opt	Proto-1	Proto-2	Proto-3	Proto-4	Proto-5	Proto-6
Trace-1	668.686	1454.444	416.435	354.394	413.523	343.415	367.727	362.178
Trace-2	654.411	1382.204	265.879	233.669	265.585	223.591	253.029	246.531
Trace-3	461.305	1331.296	255.422	232.805	257.612	221.685	204.7	204.444
Trace-4	95.79	1350.194	249.06	236.008	250.734	227.052	209.682	206.575
Trace-5	35.645	1534.044	240.435	231.331	239.469	221.838	203.924	203.073
Trace-6	32.884	1813.431	235.273	242.227	232.892	233.356	213.21	210.276

Table B.5.: Execution overhead(%) when compared with plain execution of Indexer

B.2.3 Eight Cores

	IRS_Sh	IRS_Opt	Proto-1	Proto-2	Proto-3	Proto-4	Proto-5	Proto-6
Trace-1	45.890	1345.816	387.794	325.782	392.818	349.26	321.113	329.815
Trace-2	31.410	1380.798	232.334	211.399	237.239	238.239	211.39	215.601
Trace-3	31.062	1386.956	222.408	216.232	225.199	231.08	158.417	160.585
Trace-4	14.880	1341.635	211.633	221.182	216.122	233.326	161.897	167.565
Trace-5	13.408	1341.658	199.179	219.984	204.912	231.014	161.455	169.477
Trace-6	9.726	1392.006	194.167	235.813	194.466	250.578	166.475	169.62

Table B.6.: Execution overhead(%) when compared with plain execution of Indexer

B.3 Dining Philosopher's Problem

B.3.1 Two Cores

	IRS_Sh	IRS_Opt	Proto-1	Proto-2	Proto-3	Proto-4	Proto-5	Proto-6
Trace-1	36848.947	2562.658	2021.564	1155.722	1953.05	1289.845	962.768	1040.711
Trace-2	15456.130	2449.932	2338.214	1034.048	2246.695	1181.131	555.116	558.851
Trace-3	2842.118	588.281	2085.458	936.671	2012.523	1070.232	576.78	569.479

Table B.7.: Execution overhead(%) when compared with plain execution of Dining Philosophers Problem

B.3.2 Four Cores

	IRS_Sh	IRS_Opt	Proto-1	Proto-2	Proto-3	Proto-4	Proto-5	Proto-6
Trace-1	22108.586	2054.342	2597.968	1672.907	2445.464	1339.063	1827.013	1801.944
Trace-2	9111.364	1817.914	2596.101	1368.239	2436.708	1132.18	742.664	748.36
Trace-3	697.035	436.169	2549.619	1240.308	2388.429	1027.419	720.767	733.722

Table B.8.: Execution overhead(%) when compared with plain execution of Dining Philosophers Problem

B.3.3 Eight Cores

	IRS_Sh	IRS_Opt	Proto-1	Proto-2	Proto-3	Proto-4	Proto-5	Proto-6
Trace-1	13078.157	1439.948	2307.765	1293.606	2301.262	1070.26	1538.427	1536.78
Trace-2	3339.345	976.687	2225.813	1016.723	2216.829	864.586	685.367	661.549
Trace-3	301.401	365.057	2207.008	966.478	2185.629	791.868	625.616	598.105

Table B.9.: Execution overhead(%) when compared with plain execution of Dining Philosophers Problem