Realizing Iterative-Relaxed Scheduler in Kernel Space

Master-Arbeit
Sreeram Sadasivam
2662284

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

Listings

Abstract

Although silicon chip based multiprocessors have become an industry standard, developing parallel applications that target them remains a daunting task. Multithreaded programs are used to realize such parallel applications on these multiprocessors. Concurrency bugs which are often resident in these multithreaded programs are difficult to find and reproduce. The non-deterministic nature of the multithreaded programs causes significant challenges for programmers by hindering their ability to create multithreaded programs with repeatable results. As a consequence, multithreaded programs are significantly harder to debug, test and maintain than sequential application. Thus, outlining the need for software verification techniques in multithreaded programs.

Deterministic Multithreading (DMT) is one approach used to test and debug multithreaded programs for various concurrency bugs. DMT enforces deterministic execution on multithreaded programs by maintaining determinism in the order of executing memory events or acquiring locks. However, DMT solutions cannot enforce a scheduling constraint on a multithreaded program and they do not control the schedule in advance. Thus, making DMT solutions to be better regarded for concurrency testing and not verification. However, currently there are no such techniques that allow to control the schedule of a multithreaded 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. Iterative Relaxed Scheduling (IRS) is one such technique meant to satisfy the above requirement. The existing IRS framework consists of a thread scheduler, a LLVM instrumentor and a verification engine. The thread scheduler implemented in the existing IRS framework 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. This thesis focuses on moving the IRS thread scheduler to kernel space. With such a migration, we expect an improvement in the execution time when multithreaded programs are executed in the IRS environment.

The approach used in the thesis would be benchmarked on various benchmarking programs such as Indexer, Last Zero, Fibonacci and Dining Philosopher's Problem. These experiments are intended to get a better understanding of the strengths and weaknesses of the IRS solutions implemented in the kernel space and in the user space. These experiments provide a comparison between IRS user space and kernel space solutions with execution overhead as the evaluation metric. The execution overhead for each benchmark is realized by computing the overhead in execution time when the benchmark program is executed with and without scheduling constraints. The above experiments are also evaluated with the scaling of processor cores. The scaling experiments are intended to analyze the scalability of IRS user space and kernel space solutions. Such an evaluation is expected to provide some insights into various scalability problems which can reside in various IRS solutions. Additionally, we analyze the number of necessary synchronization calls made by every IRS kernel space solutions. Such an analysis is intended to study the influence of these calls on the execution overhead generated by all IRS kernel space solutions. The approach presented in this work is expected to reduce the execution overhead and also some of the shortcomings generated by its counterpart user space design.

In the past two decades, we have observed a dramatic increase in the computing power of silicon based chips. Heat dissipation and energy consumption now limit the ability of the hardware manufacturers to speed up the chips by increasing their clock rate. Such a phenomenon has led to a major shift in computer architecture, where single-core CPUs have been replaced by CPUs consisting of a number of processing cores. The implication of such a switch is that the performance of sequential applications would no longer increase with each generation of processors, because the individual processing components are not getting faster. On the other hand, applications which are rewritten to use multiple threads can take advantage of these available computing resources to increase their performance by executing their computations in parallel across multiple CPUs.

Unfortunately, writing multithreaded programs are not easy as sequential programs. Multithreaded programs are susceptible to a lot of errors namely concurrency bugs [6]. Concurrency bugs are very hard to find and debug. Some of these bugs are deadlocks, data races. Multithreaded programs exhibit thread interleavings which are non-deterministic. Multithreaded programs that do not properly lock shared data structures can suffer from data races [24]. Chapter 2 provides a technical background of this thesis which highlights the concurrency bugs and various formal methods designed to deal with these bugs.

There are many tools and libraries created to detect and avoid these bugs. Deterministic Multithreading (DMT) is one such solution intended to debug these concurrency bugs [20]. However, DMT based solutions are suitable for concurrency testing and not a realistic solution for verification of the software application. In chapter 3, we discuss more about various DMT based solutions. Iterative Relaxed Scheduling (IRS) is a verification technique used for checking multithreaded programs [23]. In section 2.6, we explain the working of IRS in detail. This thesis is based on the IRS verification technique.

The existing IRS framework has a manual verifier, LLVM instrumentor and scheduler. Manual verifier outputs an execution trace once it is deemed to be safe. For every iteration of IRS, verifier generates an execution trace. The multithreaded program is instrumented with LLVM to incorporate annotations for every shared memory access. IRS specific methods are added before and after every shared memory access in the multithreaded program. Scheduler ensures the instrumented program executes with the scheduling constraints indicated in the execution trace provided by the verifier.

The IRS framework proposed by Metzler et al. [23] has the scheduler implemented entirely in the user space. The user space adaptation of the scheduler focuses on two implementation variants. The first user space implementation focuses on a busy waiting design and the second user space implementation uses a conditional variable setup. Let us call the first implementation as IRS_Sh and second one as IRS_Opt for the sake of simplicity. IRS_Sh uses the pool of user threads created during the multithreaded program to block and signal other threads. It uses the busy waiting design to block a certain thread and use other threads in the pool to signal from the wait. IRS_Opt uses an additional scheduler thread which takes care of requests for blocking or unblocking certain thread. It uses condition variables to address the blocking or unblocking functionality of a given thread. IRS_Sh provides poor performance when the number of threads are more than the number of cores. Busy waiting design performs poorly when the number of cores is less than the number of threads. Whereas in case of IRS_Opt, it uses conditional variables which is a synchronization abstraction provided by the pthread library. In this thesis, we argue that such an implementation is expected to have high execution overhead when the amount of communication using the conditional variables increases. The additional overhead in IRS_Opt is expected to occur due to the library overheads from pthread library. The term ‘amount of communication’ is relative to the number of shared-memory events encountered in the multithreaded program. Both the user space solutions for the scheduler is expected to suffer from poor performance. Moreover, they present additional synchronizations with their user space approach which are meant to impact the scheduling decisions of the OS scheduler. Having these additional synchronizations can present itself with some additional overhead in the overall execution time of the multithreaded program. In thesis, we present a new approach by moving the scheduler logic to kernel space. With such a migration, we expect to have no additional synchronizations in user space and we expect to have lower execution overhead generated by the kernel space adaptation.

We implement IRS scheduler as a Loadable Kernel Module (LKM). LKM implementations are portable and easy to develop, load and debug. However, by moving the IRS scheduler to kernel space we encounter some design changes and challenges with the implementation. In IRS, a thread is allowed to proceed only if the scheduling constraints allow the thread to continue its execution. These constraints in execution trace are based on the number of shared memory events completed by a certain thread. The dependencies of shared memory between threads enforces certain memory constraints among these threads. In IRS before every shared memory access, we check for memory access permission which is based on the trace file or scheduling constraints provided by the verifier. In this thesis, we use two design approaches for implementing IRS scheduler in kernel space. The first approach invokes the kernel space for every shared memory event encountered in the multithreaded program. Whereas, in the second approach we have proxy checking for memory access permission in the user space, when encountered with a shared memory event. We avoid unnecessary invocations to kernel space with every shared memory events encountered in the multithreaded program. In chapter 4, we discuss more about the designs used in this thesis.

We use various benchmarking programs such as Last Zero [1], Indexer [16], Dining Philosopher's Problem [27] and Fibonacci for comparing the IRS user space solution with the kernel space solution used in this thesis. The above comparison is done based on the execution overhead generated by the benchmarking program when its execution is constrained with various IRS solutions. IRS is a software verification approach intended to be used for multithreading programs which use shared memory design. We perform evaluations of the above benchmarks on various IRS implementations by scaling the processor core count. The scaling configurations used for such an evaluation are two, four and eight. The evaluation by scaling the processor cores are expected to reveal any scalability issues in the designs such as false sharing. Detailed analysis and evaluation of these benchmarks are highlighted in chapter 5.

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 such as bugs. 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 [17].

There are two fundamental approaches used in software verification - dynamic and static software verification [17]. Dynamic software verification is performed in conjunction with the execution of the software. This approach consists of two phases which are test and review phases. Initially, the behavior of the execution program is checked for anomalies and then verification is done in the succeeding phase. The verification process handles the test and behavior of the program under different execution conditions. Static software verification is a converse 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 technique which uses static software verification.

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 in the last two decades. Advancements were made in the domain of computer architecture by moving the computing power from single-core to multi-core architectures. With such advancement, there were needs to adapt the programs based on serialized execution to more parallelizable execution. Various parallel programming models were perceived to accommodate the perceived progression. Multithreaded programming was realized as one of the parallel programming designs to utilize the performance power 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 [21]. Some of these concurrency bugs include:

- Data Races
- Order violations
- Deadlocks
- Livelocks

Multithreaded programming yields non-deterministic execution order of a multithreaded program. Non-deterministic execution order indicates different executions order for the same multithreaded program on each execution. These different execution orders can generate different results. Data races occur when the different execution orders generate different results for the same program. Order violations are another problem which occurs due to the non-deterministic execution of the multithreaded programs. Order violations occurs when a multithreaded program depends on a sequence of threads that the scheduler may not provide. Thus, 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 [21] since, they are propagated from same class of problems which involves ordering of events. Whereas, deadlock and livelock are classified as lack of progress bugs [21].

2.3.1 Race Condition

A race condition is one of the most common class of concurrency problems. The problem arises, when there are concurrent reads and writes of a shared memory location. As stated above, the problem occurs with the 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 for y are 0, 1 and 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.

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

The above showcased problem is classified as a race condition bug. An 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.

Deadlocks

A deadlock is a state in which each thread in the thread pool is waiting for some other thread to take action. In terms of the 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].

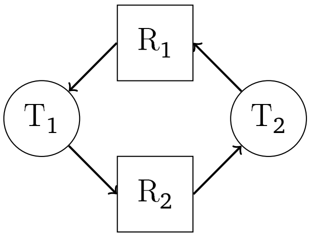


Figure 2.1.: Dead Lock Example

In the example depicted in Fig 2.1, we have two threads T_1, T_2 and two resource instances R_1, R_2. Thread T_1 holds resource R_1 and waits for the acquisition of resource R_2 from thread T_2. Whereas T_2 cannot relinquish resource R_2, unless it acquires resource R_1 from T_1 for T_2's progress. But resource R_1 is locked by T_1 which is waiting for R_2 from T_2. Thus, making a circular wait of resources. This example clearly explains the dependency of resources for the respective thread progress.

A 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 scheduling the threads in a way to avoid the resource contention issue.

Livelocks

Livelocks are similar to deadlocks, except the state of threads change constantly but with none progressing. Livelocks are a 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 [21][7]. The above mentioned situation can be avoided by having one priority process/thread taking up the action.

2.4 Model Checking

Section 2.3 addresses a need for verification in multithreaded programs. The verification solutions range from detecting causality violations to correctness of execution [14]. 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 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 programs and communication protocols [8]. A model checker tries to verify all possible states of a system in a brute force manner [2]. Thus, making state space explosion as one of the major challenges. In section 2.4.1, we discuss more about the state explosion problem. Model checking tools usually verify the partial specification for liveness and safety properties [14]. Model checking tools generate a 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 on them are at-most once. There are two methods commonly used to represent states:

- Explicit-state model checking
- Symbolic model checking

2.4.1 State explosion problem

The state space of a program is exponential in nature when it comes to the number of variables, inputs, width of the data types, etc,. The presence of function calls and dynamic memory allocation makes it infinite [14]. Concurrency makes the situation worse by having an interleaving of the threads during execution. Interleavings generate exponential number of ways to execute a set of statements/instructions thus, having an explosion in the 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 state 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 [14]. 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. 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 [26]. 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 realize it as an interleaving model. In an interleaving model, we have a single linear execution of the program arranged in an interleaved sequence. Concurrently executed events appear to be ordered to each other arbitrarily. Considering all interleaving sequences would lead to extremely large state space. Constructing the full state graph would make fitting the graph 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 reaching the same final state R, shows us that the transitions α_1 and α_2 are commutativity.

The transitions of a system play a major role in POR. POR is based on the dependency relation that exists between the transitions of a systems. A transition $\alpha \in T$ (where T is the set of transitions) is enabled in a state s, if there is a state s' such that $\alpha(s,s')$ holds. Otherwise, α is disabled in s. $\alpha(s,s')$ defines a transition function from state s to 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

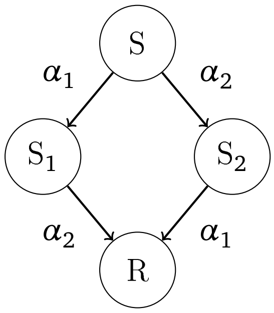


Figure 2.2.: Commutativity Example

s' such that $\alpha(s,s')$. The purpose of POR is to reduce the number of states, while preserving the correctness of the program. A reduced state graph is generated for example using depth-first or breadth-first search methods. A 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 $\alpha(\beta(s)) = \beta(\alpha(s))$.

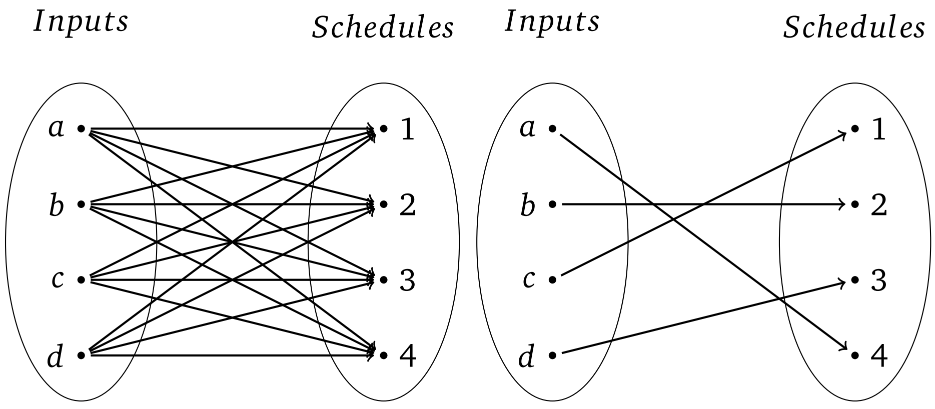
$\alpha(s)$ and $\beta(s)$ indicate different transitions on state s. These functions return a resultant state after the transition.

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, 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 a fewer number of states and retains the correctness property of the model.

The commonly used POR techniques are 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 [16]. These algorithms exploit information about which communication objects in a process gets committed to in a given set of operations in future [16]. 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 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 programs. The non-determinism in the execution of the concurrent programs makes the computation of precision function defined within these implementation 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 [16]. 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. The dynamic POR algorithm by Flanagan and Godefroid [16] explores single transitions and performs recursive calls subsequently. A persistent set is calculated at each state in the state graph of a system.



(a) Traditional Mapping (b) Deterministic Mapping

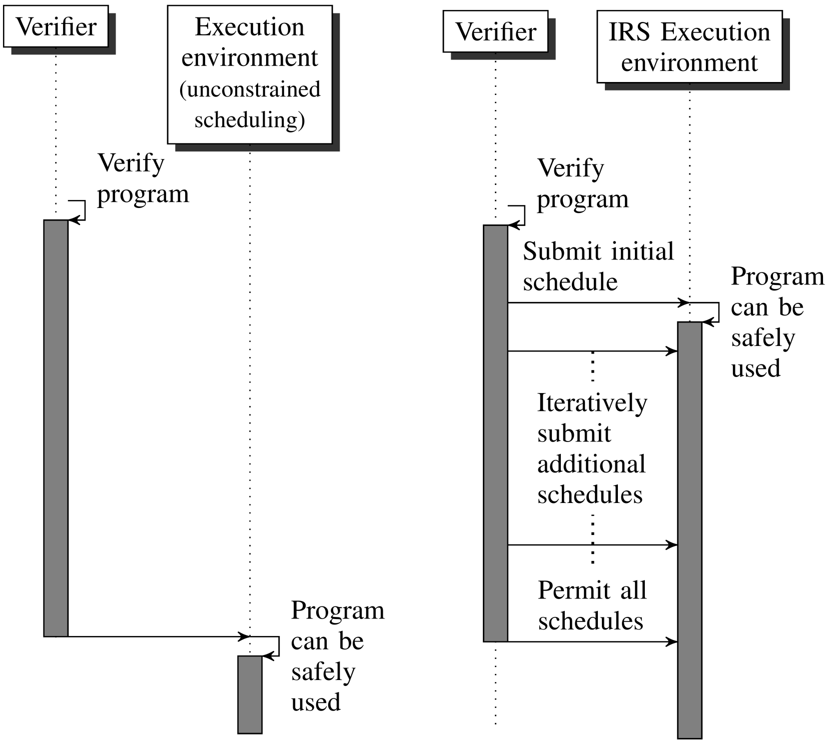
Figure 2.3.: Input to Schedule mappings in multithreaded execution based on the Cui et al. [11]

2.5 Deterministic Multi-Threading

In sections 2.2 and 2.3, we discussed about the non-determinism offered by the multithreaded programs and the bugs associated with them. By adhering to a deterministic execution, a multithreaded program can avoid heisenbugs in these designs. In this section, we discuss about an approach which enforces deterministic execution to concurrent programs. The determinism in the execution of concurrent programs are achieved by enforcing scheduling constraints during shared memory events, synchronization operations or during lock acquisitions. Deterministic Multithreading(DMT) is a technique which addresses the above hypothesis.

DMT solutions present a design methodology for enforcing a single schedule or execution trace on a given input of the program as depicted in figure 2.3b. In the above figure, we have a set of inputs mapped to a set of schedules. Figure 2.3a depicts the non-deterministic execution of the same multithreaded program. Deterministic execution shown figure 2.3b can be achieved by determining the points of synchronization or communication in these programs. Dthreads [20], KENDO [25] and GRACE [5] presents a deterministic solution using synchronization points in the program as an execution point to enforce determinism. COREDET [4] presents solution based on memory level granularity. In COREDET, a deterministic round-robin scheduler is executed at the occurrence of a shared memory event. There are many frameworks - COREDET [4], PARROT [11], KENDO [25], DThreads [20], GRACE [5], which adheres to this principle. Some of these frameworks are discussed further in the next chapter.

DMT solutions can be observed as a solution from the area of concurrency testing because they help to reduce the number of necessary test cases in the multithreaded program. These solutions trade potential execution time overhead(overhead in regard with execution time of unmodified program) by reducing the non-determinism in multithreaded program for simplifying the testing process. DMT solutions do not control the schedule in advance, thus making them unsuitable for automated verification of concurrent programs. Moreover, it is not possible to adjust the level of non-determinism in these solutions. Migrating these DMT solutions to perform program verification is unrealistic because, their scheduling constraints depend on concrete program inputs which, would require to verify all possible inputs separately. DMT solutions provides fixed scheduling constraints and cannot be relaxed during runtime [23]. They also do not provide any fairness in scheduling or for parts of the program by completely unconstrained scheduling.


Verifier
Execution environment (unconstrained scheduling)
Verify program
Program can be safely used
Verifier
IRS Execution environment
Verify program
Submit initial schedule
Program can be safely used
Iteratively submit additional schedules
Permit all schedules

(a) Conventional Verification (b) Iterative Relaxed Scheduling

Figure 2.4.: Conventional verification vs IRS verification adapted from Metzler et al. [23]

2.6 Iterative Relaxed Scheduling

From the previous section, it is abundantly clear in the drawbacks of using DMT approaches for concurrency verification. Iterative relaxed scheduling(IRS) is a program verification technique used for concurrent programs. Figure 2.4a presents a conventional software verification approach for concurrent programs. The main drawback with such a verification technique is that it presents a huge verification delay. Large verification delay is presented from the non-deterministic nature of multithreaded programs. Software verification for concurrent programs present large state spaces thus, leading to state-space explosion problem as discussed in section 2.4.1. POR technique helps to reduce the state space problem by identifying the equivalence classes of program executions such that only one representative of each class needs to be verified. Mazurkiewicz trace [22] is one such equivalence class representation. Verification delays would reach higher values for benchmark programs even if state-of-the-art POR is used [1][18]. The high complexity of state space exploration for concurrent systems is still a concern for large industrial software deployments. However, individual traces can be verified swiftly as observed when relating the verification time to the number of explored traces. From DMT techniques we were able to conjecture the reduction in the number of test cases. DMT solutions do not allow to control the schedule that will occur in advance, which makes them unsuitable for automated verification of multithreaded programs. In chapter 3, we discuss about the various DMT techniques in more detail and how it compares with the IRS solution.

Metzler et al. [23] presents a novel implementation of software verification approach for concurrent programs through a new technique - iterative relaxed scheduling(IRS). IRS provides a way to adjust the amount of non-determinism thus, providing an adjustable verification delay in the system. In IRS instead of waiting for the entire verification to complete, we can use the intermediate verification results that guarantee program correctness with one or more schedules for the execution of the multithreaded program. Mazurkiewicz traces helps to use POR as a verification technique with intermediate verification results. Figure 2.4b depicts the idea of IRS. IRS is in a way "recorder-replay" design. The verifier in IRS submits intermediate schedules for IRS environment to execute, which is similar to the "recorder". The scheduler in the IRS environment reuse the execution traces submitted by the verifier thus, making it "replayer". In chapter 3, we discuss some DMT solutions which use a similar design paradigm. IRS uses LLVM for instrumenting multithreaded programs which use shared memory design. The annotations in the instrumented program are in reference to the shared memory events within the program. Thus, IRS is able to provide fine grained memory level granularity. Mazurkiewicz traces generated by the verifier is obtained in relation with the shared memory events occuring in the mulithreaded program. With the scheduler and the recorded traces, IRS is able to enforce scheduling constraints to the multithreaded program.

The IRS framework presented by Metzler et al. [23] uses two different design variants to realize the scheduler implementation. The first design conceives the scheduling decision within the pool of user threads created in the multithreaded program. The user threads in this design shares scheduling decision thus making them shared schedulers. The blocking and unblocking of the threads in this design is done with a busy waiting implementation. In the second design, we use a seperate thread for handling the scheduling decisions, and we use condition variables for handling the blocking and unblocking of threads. The IRS scheduler in this framework [23] is realized entirely in the userspace. In this thesis, we present an alternative by moving the IRS scheduler to the kernel space. Chapter 4 highlights the need for such migration and later discusses about the kernel space adaptation.

3 Related Work

The previous chapter highlighted the need for debugging tools or methodologies for solving concurrency bugs in multithreaded programming. This thesis is conceived with a solution based on iterative relaxed scheduling(IRS). However, there are other techniques which address various solutions using deterministic multithreading(DMT). In this chapter, we explore various DMT based solutions and the similarities they share with IRS.

3.1 DTHREADS

Liu et al. [20] presents a deterministic multithreading runtime system. It is build on top of PThreads(POSIX Thread) library in Linux. The dthreads library replaces most of the pthread library functions with its own implementation enforcing determinism in execution of threads. All the threads created using dthreads library are created as processes and they use a deterministic memory commit protocol for synchronizing the conceived shared memory state. The idea of using “threads-as-processes” paradigm is motivated from the work of Berger et al. [5]. By moving the design to processes, dthreads eliminates false sharing and provides protection faults. For simplicity we would call these threads as dthreads. “Twining and diffing” technique is used to perform the deterministic memory commit protocol in dthreads. Dthreads are run independently until they reach a synchronization point where the diffing step of the above technique takes place. It compares the modification in the memory page with the twin page which contains the shared state. Each dthread enters the differential comparison(diffing) step based on a token being passed around the dthreads. Dthreads consists of two phases of execution: parallel and serial phases. The twining-diffing step occurs in the serial phase of execution. Transitions to serial phase occurs statically. Any synchronization operation will result in a transition to serial phase.

Dthreads creates private, per-process copies of modified pages between commits. This would increase the program’s memory footprint by the number of modified pages between synchronization operations. In case of IRS, we have do not change the implementation of pthreads. Therefore, the above problem never occurs in IRS however, there is a possibility of false sharing. IRS implementation emphasizes on memory level granularity whereas, dthreads focuses on the synchronization operations encountered in the user program. IRS follows a recorder-replay model whereas, dthreads uses “Twining and diffing” of threads. In IRS, we have a verifier which records the execution traces that are deemed to be safe considering the correctness of the program execution. It also consists of a scheduler which can be considered as the replay part of the model. In this thesis, we migrate the scheduler module to the kernel space from user space. The scheduler would run the instrumented user program with the execution traces generated by the verifier. Dthreads does not have a verifier or a separate scheduler module instead, it has library level support to enforce the determinism in execution. It also does not instrument the user program. Dthreads is C/C++ library support implemented entirely in user space whereas, in this thesis we highlight the IRS scheduler implemented in kernel space.

3.2 GRACE

Berger et al. [5] presents a deterministic library support in Grace. The design is similar to the Dthreads implementation. Grace also uses “threads-as-processes” paradigm. However, it is primarily targeted at fork-join models. Grace focuses on multithreaded designs which highlight thread creation and joining. The reason for the inclusion is that it also falls under the category of DMT based solutions. However, it has lot drawbacks - it focuses on fork-join models only. It is similar to dthreads in a lot of regard. IRS is completely different to the Grace implementation.

3.3 PEREGRINE

Cui et al. [10] conceives an alternative DMT solution with schedule relaxation in PEREGRINE runtime system. It is a record-replay based implementation. It combines two different scheduling designs - sync schedule and memory schedules. The hybrid scheduling design enforces efficiency and determinism in the execution of multithreaded program. Unlike the previously mentioned DMT solutions, PEREGRINE uses an instrumentor in LLVM and a user space scheduler for the replay of execution trace. PEREGRINE executes the multithreaded program with a certain input to generate its execution trace. The recorder records the trace for the given input of the program. Replayer/scheduler reuses the same execution trace for the given input of the program. It enforces the execution trace on the user program for same input thus, providing a level of determinism in its execution. It shares a lot of similarities with IRS. IRS is also record-replay based design. Both these designs provide memory level granularity. However, IRS design generates more traces with less memory level constraints for each iteration therefore, retaining some level of non-determinism in the execution of the program. The memory level determinism in IRS is enforced based on the order of the memory access. Whereas, in case of PEREGRINE it is enforced based on the output of the program. PEREGRINE uses the same execution trace for different inputs to the multithreaded program. Whereas, IRS improves the degree of non-determinism in the execution of multithreaded program with every iteration of verifier.

3.4 KENDO

KENDO is another DMT solution proposed by Olszewski et al. [25], which uses modified Linux kernel to support deterministic logical time. KENDO is a software framework, which enforces weak deterministic execution of general purpose lock-based C/C++ based multithreaded programs. Weak determinism ensures a deterministic order of all lock acquisitions for a given program input. KENDO is a subset of pthreads library. It achieves determinism with the use of deterministic logical time, which is used to track the progress of each thread in a deterministic manner. KENDO has a kernel level implementation to enforce deterministic execution of threads. The IRS implementation highlighted in this thesis focuses on a scheduler implemented in kernel space for enforcing the memory constraints provided in the execution traces by the verifier. KENDO does not have any instrumentation of user program unlike PEREGRINE or IRS. KENDO only focuses on determinism in lock acquisitions and not on all shared memory accesses whereas, IRS addresses memory-level granularity for all shared memory accesses in the multithreaded program.

3.5 COREDET

Bergan et al. [4] came up with a compiler and runtime system enforcing deterministic multithreaded execution in COREDET. It is another runtime implementation based on DMT. COREDET has two phases - parallel and serial phases similar to the Dthreads. It has an instrumentor tool in LLVM for instrumenting memory events similar to PEREGRINE. COREDET is one of the first DMT solution which addressed the shared memory events and provided memory level granularity. COREDET can be executed in two different ways - ownership tracking and store buffering. First approach tracks the ownership of data and serializes the execution whenever threads communicate. Such an approach yields sequentially consistent executions and lower overheads, but lower scalability. Second approach uses memory versioning without any form of speculation and relaxes memory ordering, yielding higher scalability at the cost of higher overheads. It shares a lot of similarities with IRS implementation. Both the solutions use LLVM for instrumenting memory events in the multithreaded program. However, COREDET uses a round-robin scheduling when it enters a serial phase of execution. Whereas in case of IRS on occurrence of a memory event, the scheduler checks for the memory access permission for the given thread with the recorded trace. COREDET does not have any record-replay implementation. It can be conceived as a runtime implementation with emphasis on fine grained memory access with serialized commits.

3.6 PARROT

PARROT is another runtime solution based on DMT from Cui et al. [11]. Compared to other DMT solutions which maps one schedule for one input as depicted in figure 2.3b, PARROT proposes an approach which uses stable multithreading(StableMT). In StableMT, we reuse each schedule on a wide range of inputs, mapping all inputs to a dramatically reduced set of schedules. PARROT is a pthread compactible implementation. PARROT provides weak determinism similar to KENDO but offers stability. PARROT can be integrated with DBUG[28] - an open source model checker in Linux for determining bugs in the schedules. Cui et al. [11] shows us that PARROT-DBUG ecosystem is more effective than either system alone. DBUG checks the schedules that matter to PARROT and the developers. Whereas, PARROT reduces the number of schedules to be checked by DBUG thus, increasing the coverage of DBUG. Compared to IRS, PARROT does not have any static code analysis done inside the multithreaded programs. The determinism provided by PARROT is relative to three factors: external input, performance critical sections, data races with respect to the enforced synchronization schedules. IRS focuses on memory level granularity with a degree of non-determinism whereas, PARROT is focused on weak determinism similar to KENDO. PARROT-DBUG focuses on StableMT with exhaustive testing of all schedules whereas in IRS, the execution of a multithreaded program can be initiated with a single execution trace from the verifier. IRS generates a new trace for every iteration whereas in case of PARROT-DBUG, the execution is blocked until the DBUG checks all the schedules.

Inference

From the above sections, it is abundantly clear that there are not many solutions which come close to the IRS. COREDET is the only implementation which seems to provide memory level granularity similar to IRS. PEREGRINE is another implementation which depicts a record-replay paradigm similar to IRS. PARROT-DBUG presents a StableMT focused on checking a set of reduced schedules for all the provided inputs in the multithreaded program. Other solutions presented in this chapter focus on DMT solutions aimed at synchronization operations rather than memory level accesses. All the DMT approaches listed above support concurrency testing whereas, IRS supports concurrency verification. Adapting the DMT solutions for concurrency verification is difficult as discussed in section 2.5.

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.

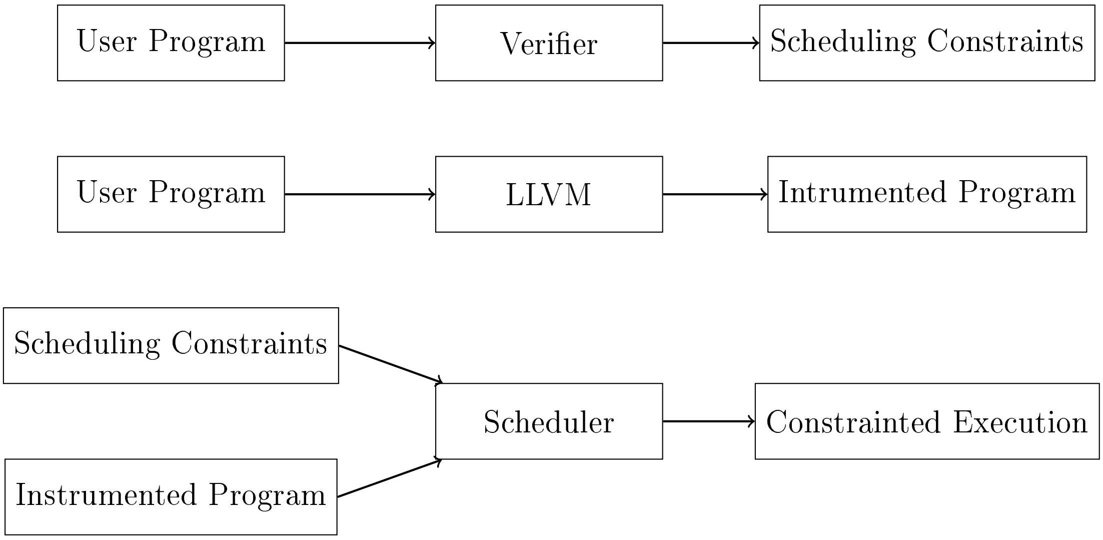


Figure 4.1.: IRS Design Overview

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 trace files. Trace files represent the execution order of shared memory events for the threads in a multithreaded program. The trace in a trace file is 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 [15]. 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.
- $VC(x) < VC(y) \iff \forall a[VC(x)_a \lneqq VC(y)_a] \wedge \exists b[VC(x)_b < VC(y)_b]$
- $x \rightarrow y$ 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 and traces 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

```
Kernel Space:
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 and 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 propagations to yield a good performance compared to the user space solutions.

Listing 4.4: First Approach

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

Listing 4.5: Second Approach

```
User Space:
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 code of this approach is depicted in listing 4.4. Such an approach is suitable when the number of memory constraints are close to the total number of shared memory events in a multithreaded program. 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 code 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:

$$m << t \tag{4.1}$$

m represents the number of memory constraints provided in the trace file for the multithreaded program and t indicates the total number of shared memory events present in the same multi-threaded program. 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.

First Approach	Second Approach
$2 * t$	$t + [0, m]$

Table 4.1.: Number of synchronization calls made to kernel space

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[19]. 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 [29]. 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 4.2 and 4.3 depict this classification.

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

Table 4.2.: Prototypes without proxy checking

	Shared Thread	Separate Thread
Semaphore		Prototype 5
Scheduler APIs		Prototype 6

Table 4.3.: Prototypes with 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.

4.3.1 Trace Registration

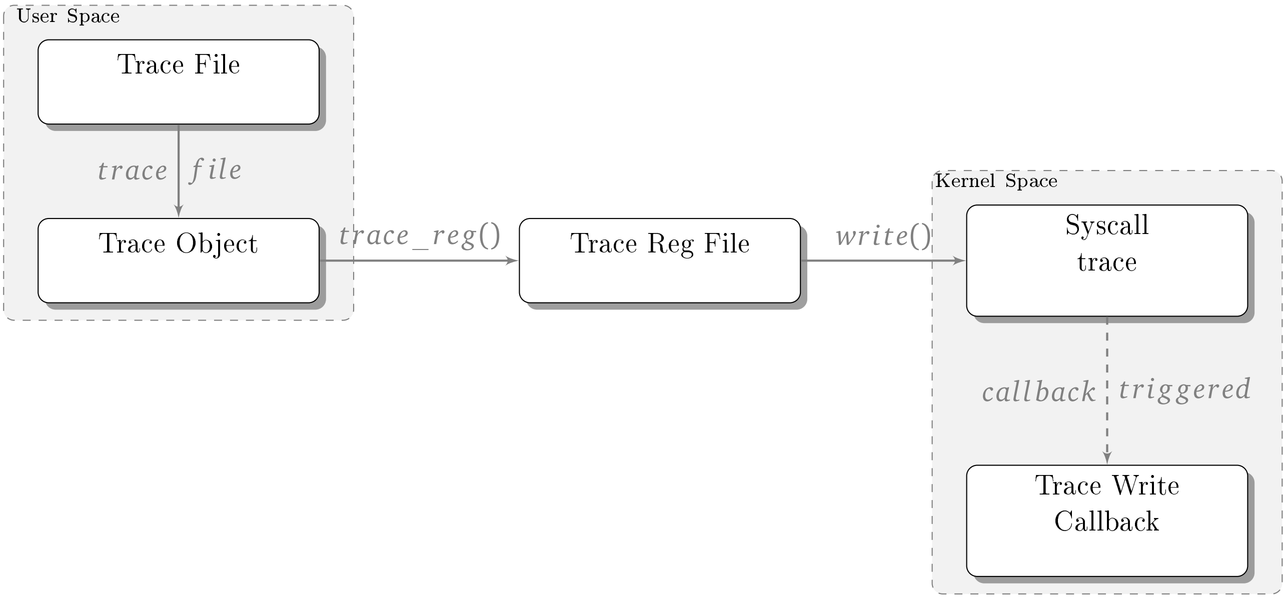


Figure 4.2.: Trace Registration

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

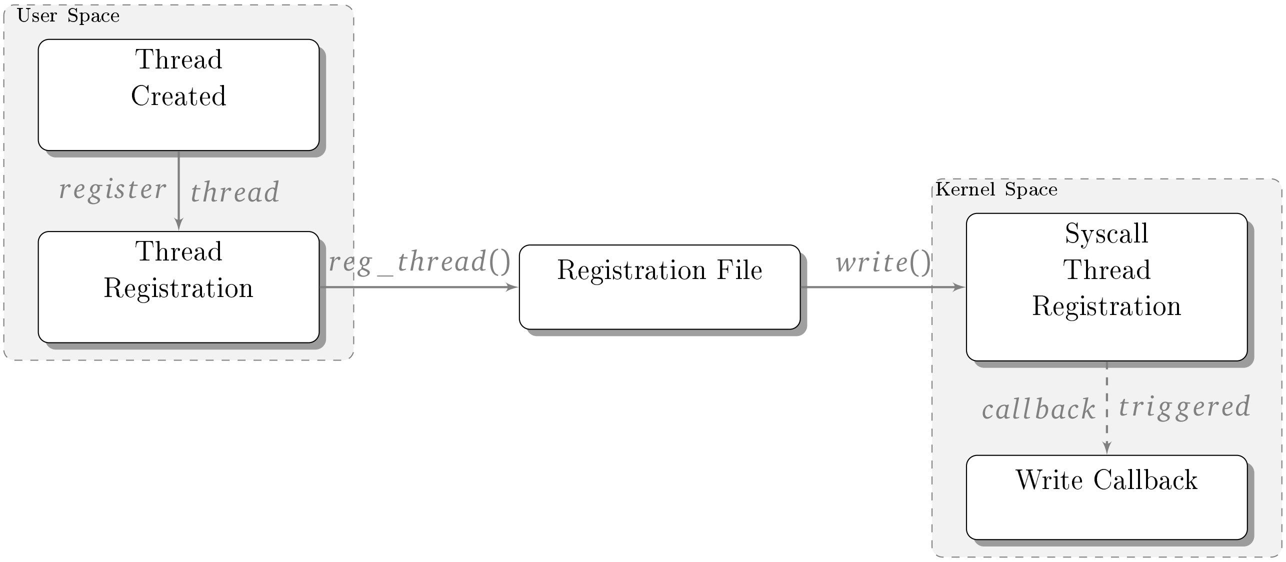


Figure 4.3.: Thread Registration

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.

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;
}
```

4.3.3 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

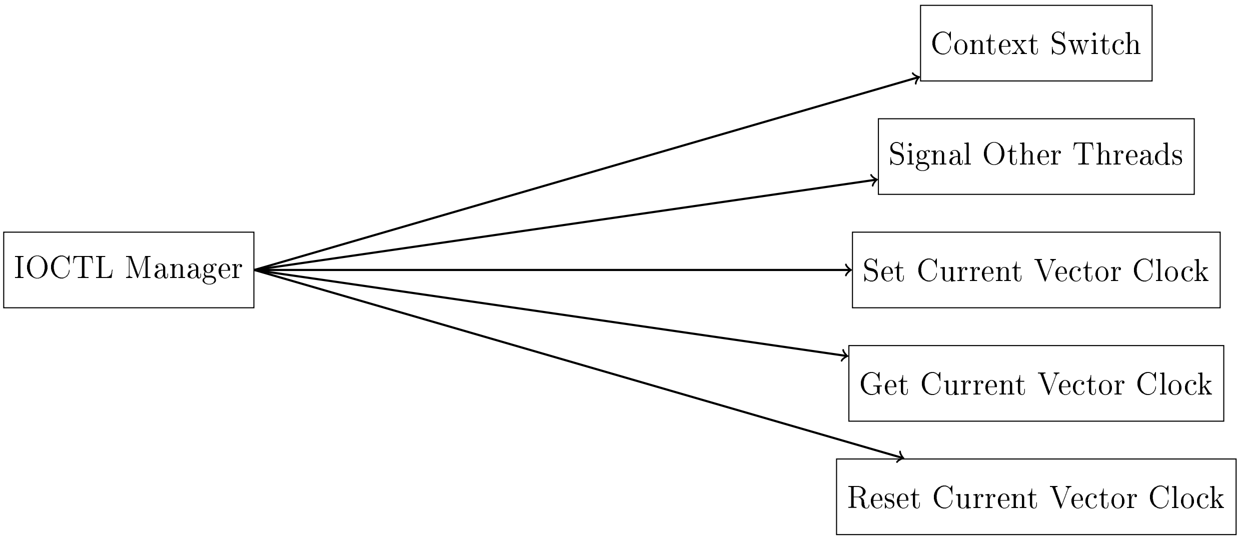


Figure 4.4.: IOCTL Manager

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.

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 code 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 code 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($j \in 1..N$):
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) {
                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 if the condition indicated in equation 4.1.

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:
Run a kernel level thread every 1ms which invokes signal_permitted_threads
    function.
Queue waitqueue={}
ctxt_switch_thread(thread_id tid) {
        if(check_perm(tid)==restricted) {
                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.
```

Additional Note

We have $signal\_all\_other\_threads()$ method invoked within $ctxt\_switch\_thread()$ method in Prototypes 1 and 3. The above call is made from $ctxt\_switch\_thread()$ to ensure that the design does not run into any deadlocks.

A $waitqueue$ is used in all the prototypes. It is used to maintain a bookkeeping for all the blocked threads. A design without $waitqueue$ would require the checking memory access permission with the trace to be done differently. The modified design would have all the signaling threads checking the trace for all threads which could be blocked and signaling all the blocked threads independently whereas, in the design discussed in this thesis we use a $waitqueue$. The waitqueue is implemented as an array of boolean values and with size of total number of threads. To check if a thread is blocked or not, we can access array with the thread's id as its index. Firstly, such a modified design highlights a possibility of a race conditions when removing a memory event from a trace. A memory event is removed from the trace once the event is completed. Secondly, the prototypes using semaphores(Prototypes 1,2 and 5) could not be adapted with such design changes. Let us consider that, there are 4 threads in the multithreaded program. Three of the threads are running, whereas the last thread is blocked. When all first three threads invoke the $signal\_other\_threads()$ method, they would perform the up operation on the semaphore variable meant for last thread(assuming that the fourth thread is allowed to execute). Problem here is that the value of the semaphore say $sema_4$ would be 3 and not one, value 3 would indicate the fourth thread to miss the yield functionality for the next three restricted memory access. As indicated in the appendix C, we use a binary semaphore and not a counting semaphore for the threads in Prototypes 1,2 and 5. To avoid the race conditions, we could have a mutex around the code section which deals with the removal of the memory event from the trace. Invoking mutexes or locks regularly can affect the performance of the multithreaded program [30]. Having a mutex inside this code section would increase the execution overhead drastically. Moreover, such a design with mutex would make the design to be logically equivalent to the one with the $waitqueue$. Therefore, there is no gain in making such a change.

The algorithmic time complexity for running $signal\_all\_other\_threads()$ is $O(n^2)$, whereas $set\_clk()$ is $O(1)$. In $ctxt\_switch\_thread()$ method we have higher algorithmic time complexity in Prototypes 1 and 3, since they invoke $signal\_all\_other\_threads()$ internally. $ctxt\_switch\_thread$ depicts the yield functionality described in this chapter. $signal\_all\_other\_threads()$ or $signal\_permitted\_threads()$ represent the revive threads functionality.

5 Evaluation

In this chapter, we perform various experiments to get a better understanding of the IRS kernel space implementations proposed in the previous chapter. The experiments are intended to provide a comparison between user space and kernel space IRS implementations. The comparison between the above implementations are evaluated across multiple processor cores ranging from two to eight cores. We evaluate kernel space solutions based on the number of calls made to kernel space for synchronization. Four benchmarking programs are used for all the above mentioned evaluations.

5.1 Setup

The evaluation is performed with a virtual machine running on a hardware with Intel Xeon E5-2650 - 2.00 GHz(16 cores) configured with Ubuntu 17.04(64 bit) as the operating system. It is configured with 4GB RAM and 80GB hard disk. The virtual machine(VM) is configured with the LLVM-CLANG 3.9, GCC 4.9 and Boost 1.6.2. We use multiple VM configurations ranging from two cores to eight cores for various evaluations. Configurations by scaling the number of processor cores is mainly intended to perform a scaled evaluation of cores to number of threads across various IRS implementations for a given benchmark.

Note

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

- IRS_Sh - One of the IRS user space implementation discussed in section 2.6. It addresses the use of a busy waiting design for blocking a thread and use the other threads in the thread pool to signal the blocked thread. Thus, making the scheduling decision shared among the threads. This design does not have an additional thread for handling the scheduling decisions.
- IRS_Opt - Another user space IRS solution discussed in section 2.6. It uses an additional thread for handling the scheduling decisions and uses conditional variables to block a certain thread when memory access is restricted.
- Proto_1 - Section 4.3 highlights all the prototypes used in this thesis to provide various solutions addressing the transition of scheduling decision to kernel space. Prototype 1 is the first synchronization discussed in section 4.3 which uses a shared scheduler design similar to IRS_Sh but, the blocking of threads is enforced by using semaphores in kernel space.
- Proto_2 - Prototype 2 is an extension of the previous prototype. Prototype also uses semaphores in kernel space for blocking a given thread but, the signaling the blocked thread is done by an additional thread which is similar to IRS_Opt.
- Proto_3 - Prototype 3 is a lot similar to Prototype 1 in the nature of behavior when it comes to design and its approach to scheduling. Main difference of Prototype 3 compared to Prototype 1 is the use of scheduler APIs instead of semaphores for blocking a given thread.
- Proto_4 - Prototype 4 is a design variant of Prototype 2. It uses the same scheduling approach as Prototype 2 but, differs in the blocking of a given thread. It uses scheduler APIs instead of semaphores.

- Proto_5 - Prototype 5 are an extension of Prototype 2, it is one of the designs which addresses the second approach discussed in section 4.1.2. As discussed in section 4.1.2 it uses a proxy checking of memory access permission in user space.
- Proto_6 - Prototype 6 are an extension of Prototype 4, it is another design which addresses the second approach discussed in section 4.1.2. As discussed in section 4.1.2 it uses a proxy checking of memory access permission in user space.

In short, Prototype 1 , Proto_2, Proto_3, Proto_4 use the first approach discussed in section 4.1.2. Whereas, Proto_5 and Proto_6 use the second approach discussed in section 4.1.2.

5.2 Benchmarks

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

- Fibonacci - Program runs with two threads computing the Fibonacci numbers for 25 iterations per thread.
- Last Zero - Abdulla et al. [1] presents this benchmark for evaluating their work. The benchmark program runs with 16 threads.
- Indexer- Flanagan and Godefroid [16] use this benchmark program for evaluation of their work. The benchmark program runs with 15 threads.
- Dining Philosophers Problem - The benchmark program runs with 16 threads. This benchmark is motivated from the solution presented in Silberschatz et al. [27].

We generate memory constraints(constraints addressing shared memory events) for the above benchmarks in the form of an execution trace stored in a trace file. The trace file is a graphviz representation. The trace file is manually generated based on the benchmark in use. It could be automated in the future by having an automated verifier as discussed in the section 4.1. The graphviz representation is converted manually into a vector clock representation containing only the shared memory dependencies between the threads. The vector clock representation is a string depicting the inter-dependencies between threads on a shared memory event. We use the vector clock representation for the prototypes and graphviz file for the user space IRS solutions. Both the representations are logically equivalent. More details about their representation and conversion can be found in appendix C.

5.3 Evaluation Metrics

5.3.1 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 benchmarking 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 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.

5.3.2 Number of necessary synchronization calls

This metric used to realize the number of necessary calls made to the kernel space for synchronization purposes. The number of synchronization calls are primarily the number of IOCTL calls made under the commands: context_switch, signal_all_other_threads or set_clock. The term 'necessary' is validated based on the number of IOCTL calls which actually performs the synchronization operations for the commands mentioned above. Whereas in case of 'unnecessary' calls, we observe a behavior of IOCTL calls returned to the user space without any major influence or changes in the kernel space. We record the number of IOCTL calls with the command context_switch and determine the prototypes which provide a smaller overhead. We expect some of the prototypes to make unnecessary calls to kernel space when requesting a context switch. These unnecessary calls are expected to generate more overhead on execution time. For multi-threaded programs which have extremely less number of memory constraints compared to the total number of shared memory events is expected to yield poor performance on the prototypes which yield unnecessary calls. Thus, making this evaluation metric a crucial parameter when comparing various prototypes. More explanation about such a behavior is discussed in section 4.1.2. This evaluation metric would also help in understanding the prototypes which would have high and low synchronization overhead. We describe the use of this metric more in this section 5.4.

5.4 Number of synchronization calls

We evaluate the number of necessary calls made to kernel space for synchronization operation. The evaluation is done across all six prototypes described in this thesis. 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 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 the number of memory constraints in the trace is extremely less than the total number of shared memory events in the benchmark as depicted in equation 4.1. Section 4.1.2 provides a detailed explanation of the approach used in prototypes 5 and 6, and the reason for their better performance.

	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

The Fibonacci benchmark 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 for shared memory access 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

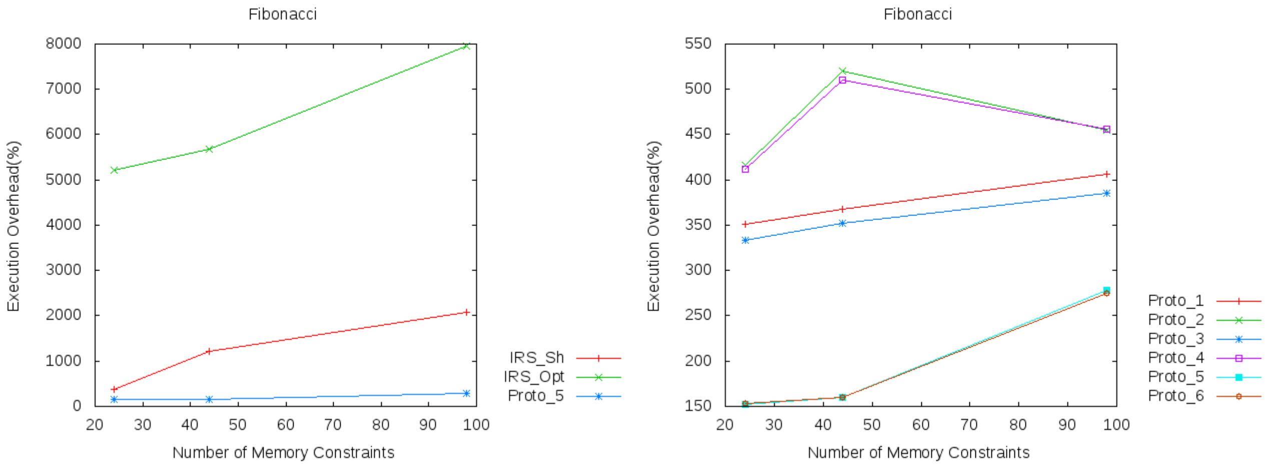
Table 5.3.: Execution overhead(%) when compared with plain execution of Fibonacci across six prototypes

Table 5.3 shows us that the execution overhead is drastically reduced for the prototypes 5 and 6. Reduction in the number of IOCTL calls is reason for such a difference in the execution overhead. Prototypes 5-6 provide lower execution time overhead compared to other prototypes, when the condition indicated in equation 4.1 is satisfied. The evaluation metric ‘number of necessary synchronization calls’ is primarily targeted for IOCTL calls for synchronization purposes. Since Prototypes 5-6 present lower execution time overhead and have less number of synchronization calls, we can safely indicate that they present lowest synchronization overhead among the prototypes for this design.

5.5 Comparison between user space and kernel space IRS solutions

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 four bench-marking programs - Fibonacci, Last Zero, Indexer and Dining Philosophers Problem. We scale the processor configuration from two to eight processor cores and monitor the changes in the performance overhead across the benchmarks - Last Zero, Indexer and Dining Philosophers Problem for the various IRS implementations. Fibonacci benchmark is evaluated only with the configuration of two processor cores because the number of threads configured for the benchmark is two. The best prototype among the six prototypes is chosen and compared with IRS user space implementation for each benchmark. Scaling the number of cores aids in the scalability evaluation of various IRS implementations. Such an evaluation also helps to realize the possibility of false sharing problem in the designs. Execution overhead of the benchmarks are taken after running the programs 1000 times and obtaining the mean of the 1000 runs.

5.5.1 Fibonacci



(a) User space vs Best Prototype

(b) Comparison between prototypes

Figure 5.1.: Comparison of IRS with Fibonacci on two cores

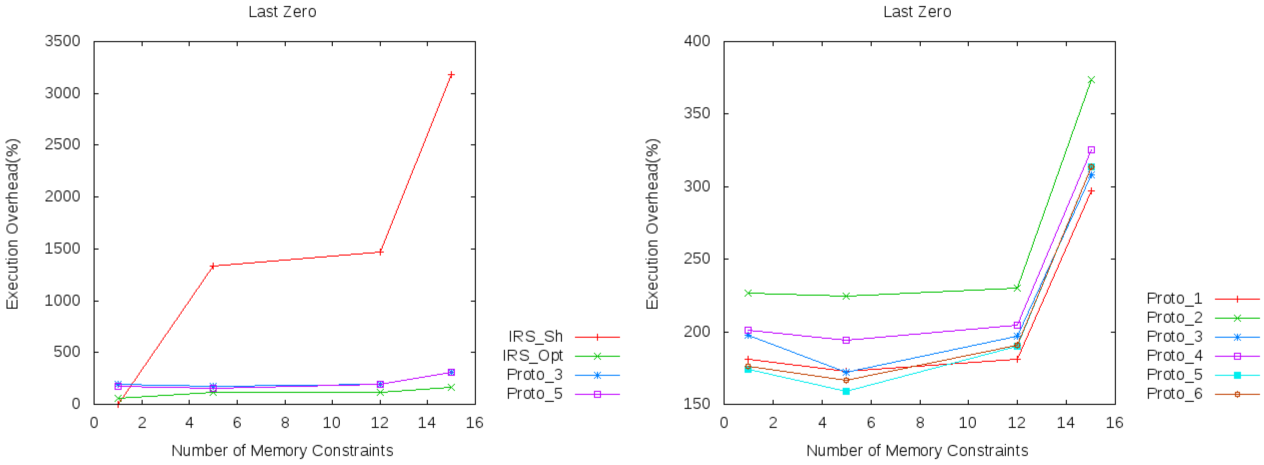
Fibonacci benchmark has two threads at its disposal. The pseudo code of this benchmark is depicted in listing A.4. The trace files for this benchmark provides 98, 44 and 24 memory constraints. The memory constraints indicated in this benchmark are the shared memory dependencies between the two threads in the benchmark. The benchmark is configured to run for 25 iterations. For every iteration as seen in listing A.4, we have three shared memory event. These memory events include two consecutive reads and one final write. There are three memory events for iteration and there are two threads in total for this benchmark. Thus, having a total of 150 shared memory events. The execution overhead is expected to be higher when the number of memory constraints are close to total shared memory events. For the first trace file (98 memory constraints), we expect to have a higher execution overhead in IRS user space implementations and comparatively lower overhead in kernel space implementations. With the third trace(24 memory constraints), we expect the difference in the execution overhead of IRS user space solutions to the prototypes to be smaller. The key reason for such a behavior is that we expect the user space solutions to perform better when the number of memory constraints in the benchmarking programs are far less to the total number of shared memory events in the program.

From figure 5.1b, we observe that Proto_5 and Proto_6 provide the best performance among all the prototypes. The reason for such a behavior is the condition depicted in equation 4.1 satisfies. Section 4.1.2 presents a detailed reasoning for such a behavior. Another interesting observation is the growth rate in relation to execution overhead for Prototypes 5-6 compared to other prototypes when the number of memory constraints are increased. When the number of memory constraints is increased from 44 to 98, in prototypes 5-6 we observe an increase in execution overhead from 160% to nearly 300%. Whereas in case of prototypes 1 and 3, we observe only a small increase from 350% to 380%. From this behavior we can extrapolate that when the number of memory constraints in the benchmark gets closer to the total number of shared memory events, prototypes 1-4 would have comparatively low execution overhead to prototypes 5-6. The reason for such a behavior is explained in section 4.1.2.

From figure 5.1b, we have taken Proto_5 as best prototype for this benchmark. Proto_6 provides nearly the same results but, we have chosen only one of the two. From figure 5.1a, we observe a very low overhead depicted by Proto_5 compared to the two user space solutions. When the number of memory constraints are reduced, we observe a decline in the execution overhead for the user space solutions. It is evident when number of memory constraints is reduced from 44 to 24. IRS_Sh seems to provide the best performance among the two user space solutions of IRS. The reason for such a behavior is that there is no additional thread in IRS_Sh and it is implemented with a busy waiting design. One of the benefits of busy waiting design is that it is very responsive when the number of threads are less than or equal to the number of cores. IRS_Opt depicts the worse performance among all the IRS designs. IRS_Opt utilizes additional thread for signaling the blocked threads. Thus, making a total of 3 threads for this program. For every memory event we have 3 threads running on two cores. The possibility of the scheduler thread(the additional thread used by IRS_Opt for scheduling) getting context switched is higher. The results shown in figure 5.1a validates the above mentioned hypothesis.

5.5.2 Last Zero

Abdulla et al. [1] showcase 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 listing 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 and minimum being 31 shared memory events.

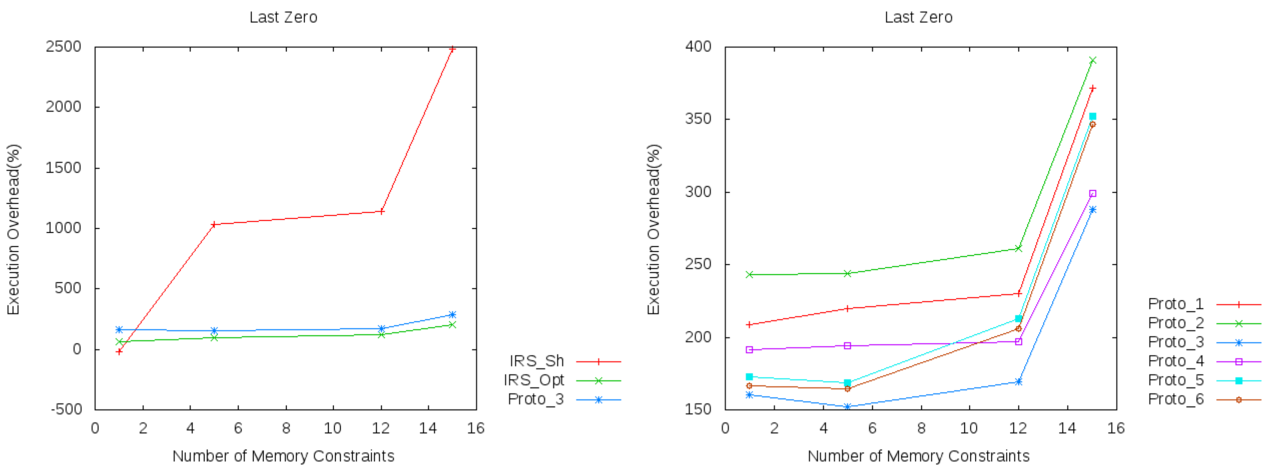


(a) User space vs Best Prototype

(b) Comparison between prototypes

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

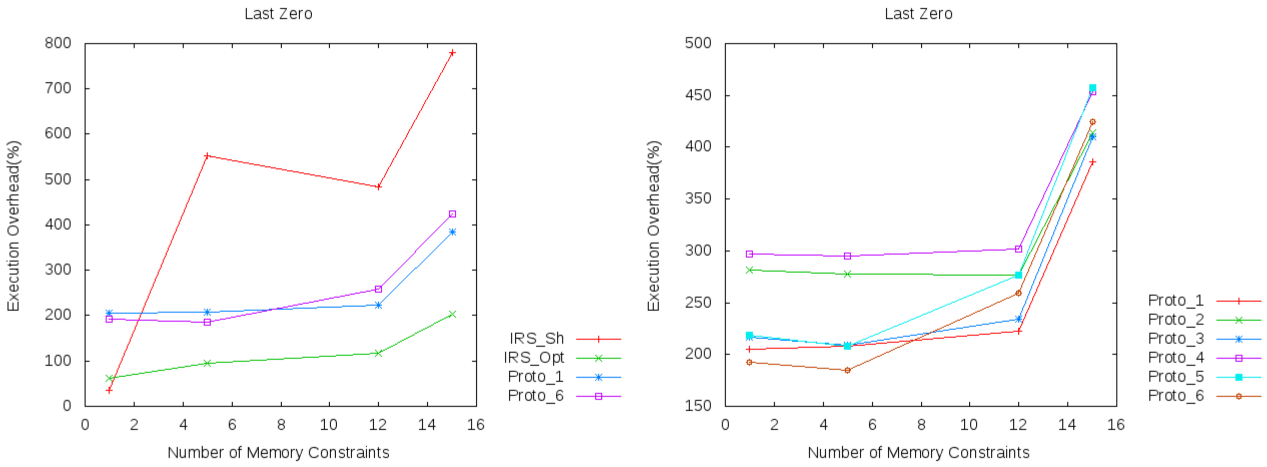
This benchmark has 16 threads in total. In the first trace file(15 memory constraints), we have atleast one dependency for each thread except for the first thread(thread with id as 0). The first trace provides some diversity in the number of memory constraints realized with this benchmark.



(a) User space vs Best Prototype

(b) Comparison between prototypes

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



(a) User space vs Best Prototype

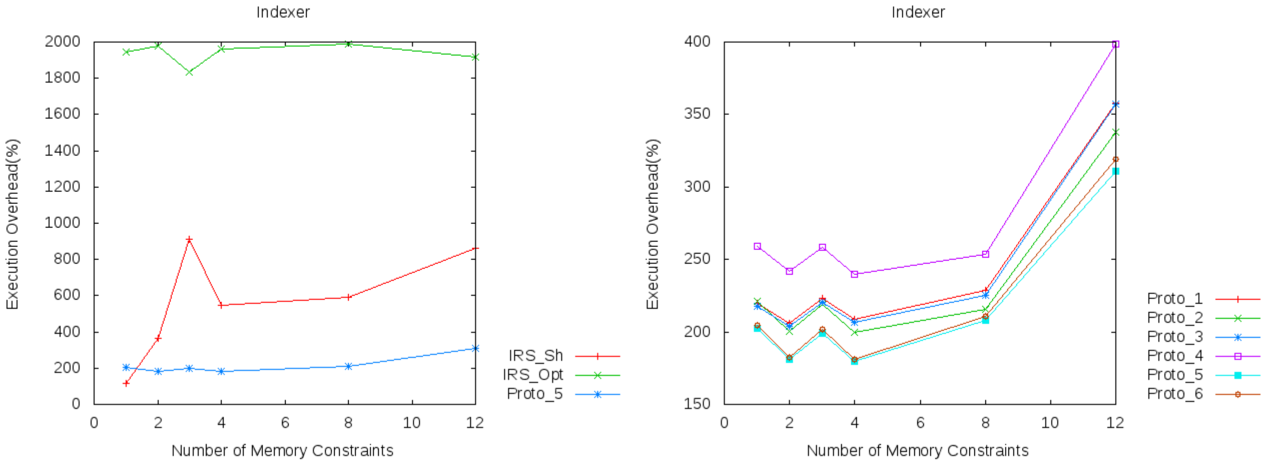
(b) Comparison between prototypes

Figure 5.4.: Comparison of IRS with Last Zero on eight cores

With the evaluation configured for two, four and eight processor cores, we expect the IRS_Sh to provide the worst performance of the all IRS implementations. IRS_Sh is a busy waiting design. Busy waiting design makes the waiting thread to constantly poll one of the cores thus, making it performance inefficient. This benchmark has dependencies across all 16 threads with the first trace file and there are more number of threads to the number of processor cores, thus making the busy waiting design to perform poorly among all the other designs for the first trace. However, it is expected to improve the execution overhead with the scaling of cores. The condition depicted in equation 4.1 satisfies in this benchmark for all the traces thus, making prototypes 5 and 6 to provide the best performance among all the prototypes. Section 4.1.2 highlights this condition in more detail manner.

The results depicted in figures 5.2b, 5.3b, 5.4b show us that Proto_1 and Proto_3 performs better for first trace(number of memory constraints are 15) because the number of memory constraints are closer to the total shared memory events in the benchmarking program. From Fig 5.2b, it is evident that Proto_5 and Proto_6 performs nearly the same when it comes to execution overhead. For traces which have less number of memory constraints(traces 3, 4 have 5, 1 constraints respectively), Proto_5 and Proto_6 performs better among all prototypes in overall. From Figures 5.2a, 5.3a, 5.4a, it is evident that IRS_Sh performs the worst for this benchmark compared to all the other IRS implementations. From Figures 5.2a, 5.3a, 5.4a, we observe that execution overhead for IRS_Sh reduces with the increase in the number of cores. From these results, we can extrapolate that the execution overhead would be much lower for IRS_Sh if the number of processor cores were 16 or more. As explained in the previous paragraph, it is because IRS_Sh is based on a busy waiting design. IRS_Opt provides the best performance among all the IRS solutions under all evaluations of this benchmark. It performs better than IRS_Sh because it uses a condition variable design with an additional thread for handling the scheduling decisions. Based on the results for this benchmark, we can conclude that the overhead generated by the pthread library on the IRS_Opt would be less compared to all the other IRS designs. Thus, making IRS_Opt best choice among all the IRS designs for benchmarks which follow similar conditions and configurations to this benchmark.

5.5.3 Indexer



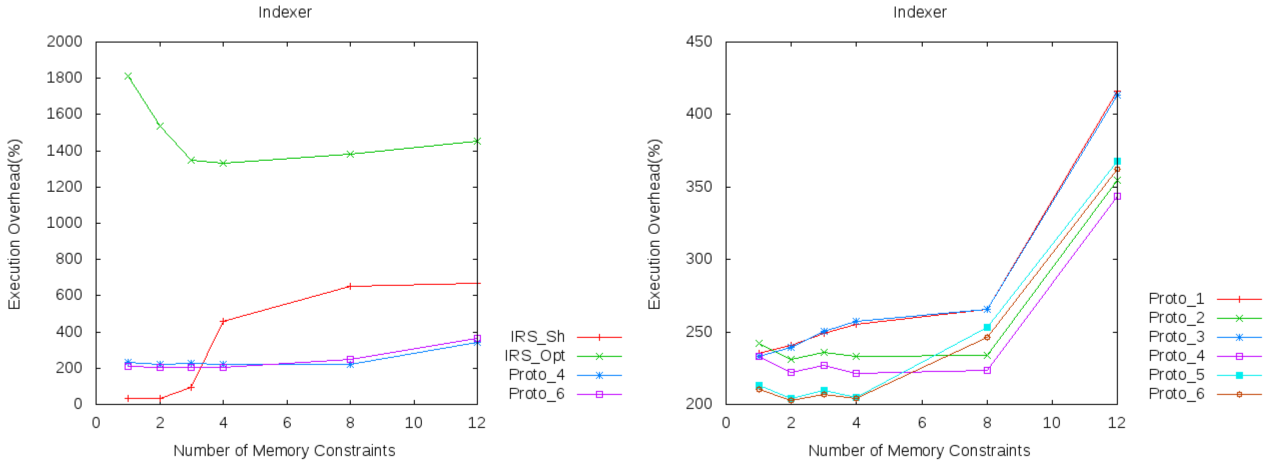
(a) User space vs Best Prototype

(b) Comparison between prototypes

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

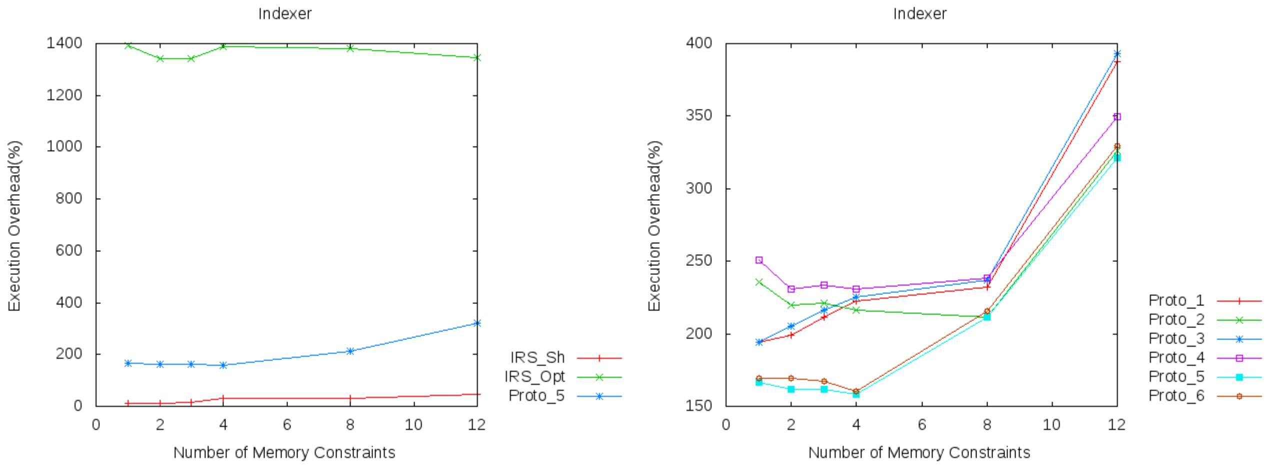
Flanagan and Godefroid [16] utilize this benchmark for evaluating their dynamic POR design. The pseudo code for the indexer program is realized in listing A.2. The indexer program revolves around a hash table, where threads read and write hashed messages on it. Each thread calculates four messages and writes them to a shared hash table. Collisions are detected and avoided using the compare and swap(cas) 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 all the constraints are within a span of four threads and not all 15 threads. For the first trace, we have three memory constraints in each of the four threads. We expect to have good performance for IRS_Sh because the number of constraints are not spread across all 15 threads.



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

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



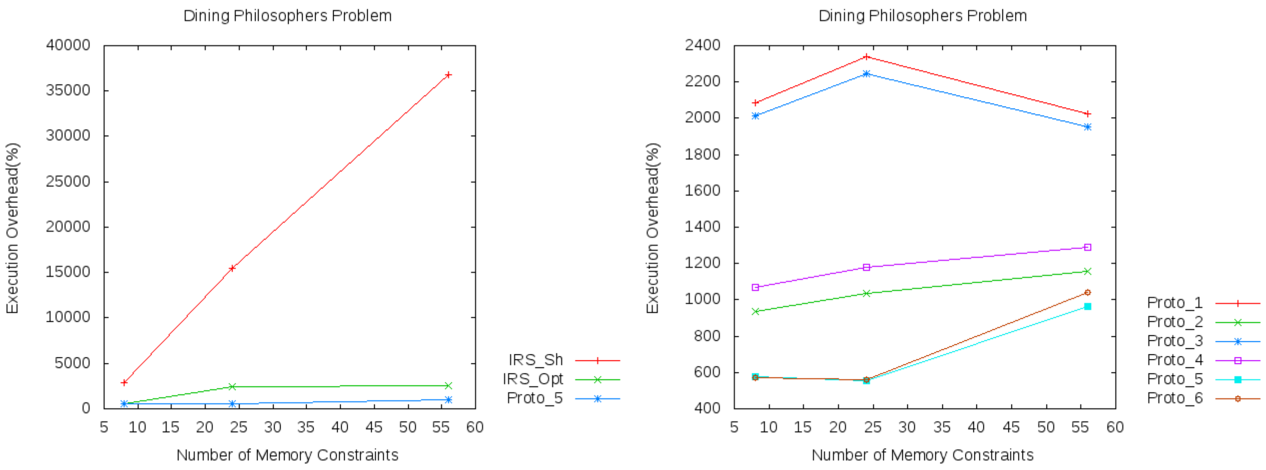
(a) User space vs Best Prototype (b) Comparison between prototypes

Figure 5.7.: Comparison of IRS with Indexer on eight cores

We expect it to perform the best when the core count is 8, even when the number of constraints are set at 12. Proto_5 and Proto_6 are expected perform best among all the prototypes for this benchmark, because the condition indicated in equation 4.1 satisfies.

From figures 5.5b, 5.6b, 5.7b, it is evident that Proto_5 and Proto_6 performs the best among the prototypes in all scenarios for this benchmark. Based on the analysis of figures 5.5a, 5.6a, 5.7a we observe a trend in the improvement of execution overhead in IRS_Sh. As expected IRS_Sh provides the best performance when the number of cores is eight as observed in Figure 5.7a. However, IRS_Opt showcases a huge overhead for this benchmark. One of the reasons for such a poor performance would be the lack of diversity of memory constraints(diversity indicates that the memory constraints are within a span of four threads not all 15 threads). There are few anomalies in the results, such as the increase in execution overhead for IRS_Opt when number of memory constraints is 1 as shown in figure 5.6a. There is no theoretical explanation for such an increase. Another anomaly is when the number of memory constraints is 3 and the configuration for processor count is set to two as shown in figures 5.5b and 5.5a, we observe an increase in execution overhead for all designs except IRS_Opt. The trace file with three memory constraints spans three threads. Thus, making a one thread sleep longer in such a scenario eventually creating a surge in execution overhead.

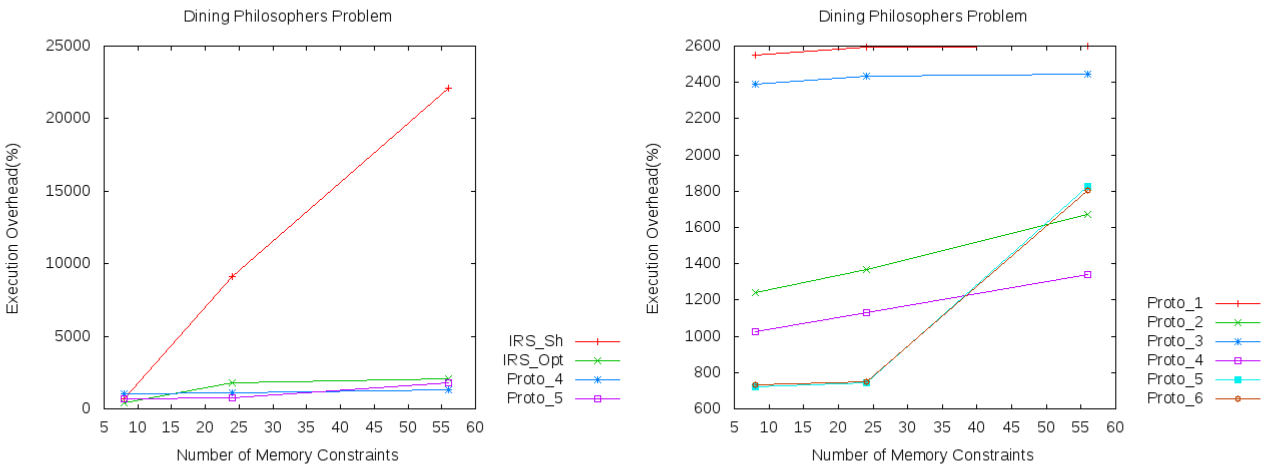
5.5.4 Dining Philosopher's Problem



(a) User space vs Best Prototype

(b) Comparison between prototypes

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

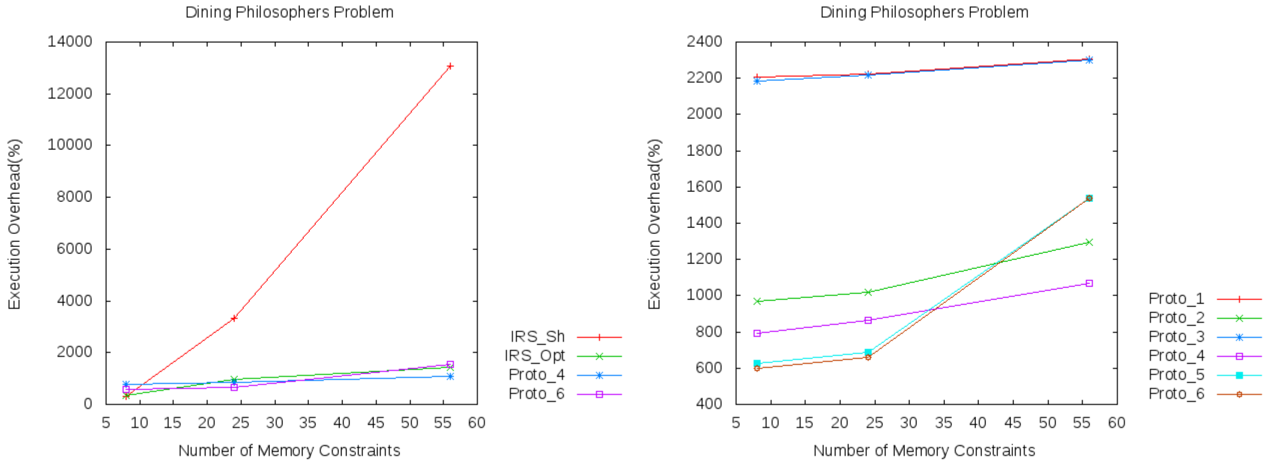


(a) User space vs Best Prototype

(b) Comparison between prototypes

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

The 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. [27] 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 the thread id of the philosopher threads. Every



(a) User space vs Best Prototype

(b) Comparison between prototypes

Figure 5.10.: Comparison of IRS with Dining Philosophers Problem on eight cores

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 listing A.3.

In our experiments, we have adapted the solution to have 16 threads and 10 iterations. Compared to previous benchmarks which had only few constraints, this benchmark has more constraints and spans 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.

This benchmark has 16 threads at its disposal. 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. The number of memory constraints listed for the three traces include: 56, 24, 8. The first trace contains memory constraints spanning all the threads. This benchmark satisfies the condition depicted in equation 4.1 in all trace file conditions. Prototype 1-4 follow the first approach discussed in section 4.1.2. Evaluation on the number of necessary synchronization calls highlighted in section 5.4, infers that the prototypes 1-4 to perform poorly when the condition mentioned in equation 4.1 satisfies. Among prototypes 1-4, we expect prototype 1 and 3 to perform the worst under these conditions. Proto_1 and Proto_3 are kernel space solutions implemented with a shared scheduler in place. Proto_1 and Proto_3, performs $signal\_other\_threads$ call from $AfterMA()$ of every shared memory event. As depicted in section 4.3.4, signaling of other threads is a costly operation compared to updating a vector clock(prototypes 2 and 4 updates vector clock instead of signaling threads). Thus, making Proto_1 and Proto_3 to perform the worst among the prototypes. We expect Proto_5 and Proto_6 to provide the best performance among the prototypes because of the condition mentioned in equation 4.1. IRS_Sh is expected to give away a very poor performance because of the diversity of constraints and since it is based on busy waiting design.

From figures 5.8b, 5.9b, 5.10b, it is evident that Proto_5 and Proto_6 perform the best among the prototypes under all the scenarios of this benchmark. 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 to have the smaller overhead compared to IRS_Sh. From Figs {5.8a, 5.9a, 5.10a}, it is evident that IRS_Sh seems to have the worst performance for this benchmark because of its busy waiting design. However, with the increase in the number for processor cores we can observe a reduction in the execution overhead for IRS_Sh.

5.5.5 Inference on the scaling of processor cores

The experiments performed by scaling the processor cores provided valuable insight into the design problems existent in the IRS implementations. From the results of benchmarks- Last Zero, Indexer, Dining Philosophers Problem, it is evident that the prototypes do not scale well when the number of cores are increased. This situation is evident with the first trace(first trace contains the highest number of memory constraints among all the traces for a given benchmark) of each benchmark as depicted in all the results. In kernel space for all prototypes, we have implemented vector clocks as a C struct containing an array of integer. Such a design was meant to provide good readability and also better memory referencing when invoked from different function. However, such a design drastically suffers from the problem of false sharing. For getting a better understanding of false sharing please refer to appendix C. Such an impact is clearly evident with the scaling of processor cores on the results of all three benchmarks.

5.6 Inference

The Fibonacci benchmark helped us in understanding the behavior of prototypes. The experiments by scaling the processor cores clearly explained the merits and demerits of various IRS designs. All the 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 all prototypes suffer from the problem of scalability. The IRS user space implementations performs better than the IRS kernel space implementations, when the number of memory constraints are in single digits and the number of memory constraints are comparably large.

By observing the above results we can conclude that there is a need of a solution which is the combination of IRS_Opt and Prototypes 5-6. This implementation would have a loadable kernel module similar to Proto_5 or 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 the processor to the OS Scheduler from the thread and also an interface to revive the thread. The check for memory access permission would be implemented entirely in the 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 condition variables in IRS_Opt.

The IRS scheduler designs implemented in this thesis highlights a new approach in designing the scheduler for the IRS framework which involved by moving the scheduler logic to kernel space. We introduced two different design approaches in moving the IRS scheduler to kernel space. The first approach triggers an IOCTL call to kernel space for every shared memory event. In the second approach, we reduce the number of IOCTL calls made to the kernel space for every shared memory event. The reduction in the number of IOCTL calls is achieved by having a proxy checking for memory access permission during a shared memory event. The kernel space solutions presented with six different prototypes. The first four prototypes used the first approach and the last two prototypes were realized with the second approach. All the prototypes presented in this thesis were incremental improvements over its former. We also introduced an alternative representation in the form of vector clocks for realizing the scheduling constraints enforced on threads in a multithreaded program. The vector clock representation was intended to reduce the code complexity and for overcoming one of the design challenges of this thesis.

In our experiments, we compared the IRS user space and IRS kernel space solutions with the benchmarking programs - Fibonacci, Last Zero, Indexer and Dining Philosopher's Problem. In our experiments, we observed that IRS_Sh performed poorly when the number of threads were more than the number of cores and when the memory constraints spanned all the threads in the benchmarking program. We also observed that the IRS_Opt performed better in nearly all the benchmarks but, performed poorly when the memory constraints were spanned to fewer number of threads as observed with Indexer and Fibonacci experiments. Among the kernel space designs, prototypes 5 and 6 performed the best in nearly all the experiments. However, the experiments showed us that when the number of memory constraints came closer to the total number of shared memory events Prototypes 1 and 3 were better as it can be observed with Fibonacci and Last Zero benchmarks. In our experiments, we observed that kernel space designs perform better when number of memory constraints are higher and closer to the total number of shared memory events. However, the prototypes depicted in this thesis suffer from scalability problems as observed with the experiments done on scaling of the processor cores. The prototypes could be optimized by removing the C-structure representation of vector clock, which could help in overcoming a potential false sharing problem.

An experimental study of all these prototypes suggested a new design outlook for moving the IRS scheduler to kernel space. Based on a preliminary analysis, it is evident that there is a need of solution which is a combination of IRS_Opt and Prototypes 5-6. Such a design would require the custom yield and revive functionality from prototypes 5-6 and checking for memory access permission would be handled in the user space entirely. In short, the condition variable design used in IRS_Opt would be replaced by the custom yield and revive functionality from Prototypes 5-6. Such an approach may help to further improve the performance of constrained execution of a multithreaded program on IRS environment.

[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, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pages 337–351. ACM, 2011.

[11] 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.

[12] Edsger W Dijkstra. Cooperating sequential processes. In The origin of concurrent programming, pages 65–138. Springer, 1968.

[13] Edsger W Dijkstra. The structure of the multiprogramming system. In The origin of concurrent programming, pages 139–152. Springer, 1968.

[14] 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.

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

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

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

[18] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In International SPIN Workshop on Model Checking of Software, pages 95–112. Springer, 2007.

[19] Ariane Keller. Tldp - kernel space - user space interfaces, 2008. URL `http://wiki.tldp.org/kernel_user_space_howto`.

[20] 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.

[21] 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.

[22] Antoni Mazurkiewicz. Trace theory. In Advanced course on Petri nets, pages 278–324. Springer, 1986.

[23] Patrick Metzler, Habib Saissi, Péter Bokor, and Neeraj Suri. Quick verification of concurrent programs by iteratively relaxed scheduling. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pages 776–781. IEEE Press, 2017.

[24] Robert HB Netzer and Barton P Miller. What are race conditions?: Some issues and formalizations. ACM Letters on Programming Languages and Systems (LOPLAS), 1(1):74–88, 1992.

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

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

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

[28] Jiří Šimša, Randy Bryant, and Garth Gibson. dbug: systematic testing of unmodified distributed and multi-threaded systems. In International SPIN Workshop on Model Checking of Software, pages 188–193. Springer, 2011.

[29] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In Proceedings of the 9th USENIX conference on Operating systems design and implementation, pages 33–46. USENIX Association, 2010.

[30] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. IEEE Transactions on Computers, 43(6):651–663, 1994.

[31] Linux Torvalds. Scheduler api headerfile in linux kernel, 1991. URL `https://github.com/torvalds/linux/blob/master/include/linux/sched.h`.

[32] Linux Torvalds. Semaphore headerfile in linux kernel, 1991. URL `https://github.com/torvalds/linux/blob/master/include/linux/semaphore.h`.

Appendices

A Benchmarks

A.1 Last Zero

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

```
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;
```

A.2 Indexer

Listing A.2: Indexer Program based on Flanagan and Godefroid [16]

```
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;
}
```

A.3 Dining Philosopher's Problem

Listing A.3: Dining Philosopher's Problem Program

```
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;
}
```

A.4 Fibonacci

Listing A.4: Fibonacci Program

```
Shared Variables: int i=1,j=1;
Local Variables: int k, num_iter=N;
Thread 0:
for (k:=0; k<num_iter; k++) {
        i+=j;
}
Thread 1:
for (k:=0; k<num_iter; k++) {
        j+=i;
}
```

B Experimental Results

B.1 Last Zero

	Prototypes 1-4	Prototypes 5-6
Trace-1	62	39
Trace-2	62	31
Trace-3	62	31
Trace-4	62	31

Table B.1.: Number of IOCTL calls made for Last Zero benchmark

	Prototypes 1-4	Prototypes 5-6
Trace-1	31	8
Trace-2	31	0
Trace-3	31	0
Trace-4	31	0

Table B.2.: Number of context switch calls made for Last Zero benchmark

B.1.1 Two Cores

Plain execution time(Unconstrained execution) is 469.199 μs.

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

B.1.2 Four Cores

Plain execution time(Unconstrained execution) is 595.724 μs.

	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.4.: Execution overhead(%) when compared with plain execution of Last Zero

B.1.3 Eight Cores

Plain execution time(Unconstrained execution) is 595.724 μs.

	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.5.: Execution overhead(%) when compared with plain execution of Last Zero

B.2 Indexer

	Prototypes 1-4	Prototypes 5-6
Trace-1	144	76
Trace-2	144	72
Trace-3	144	72
Trace-4	144	72
Trace-5	143	72
Trace-6	143	72

Table B.6.: Number of IOCTL calls made for Indexer benchmark

	Prototypes 1-4	Prototypes 5-6
Trace-1	72	4
Trace-2	72	0
Trace-3	72	0
Trace-4	72	0
Trace-5	72	0
Trace-6	72	0

Table B.7.: Number of context switch calls made for Indexer benchmark

B.2.1 Two Cores

Plain execution time(Unconstrained execution) is 434.1 μs.

	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.8.: Execution overhead(%) when compared with plain execution of Indexer

B.2.2 Four Cores

Plain execution time(Unconstrained execution) is 554.975 μs.

	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.9.: Execution overhead(%) when compared with plain execution of Indexer

B.2.3 Eight Cores

Plain execution time(Unconstrained execution) is 695.938 μs.

	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.10.: Execution overhead(%) when compared with plain execution of Indexer

B.3 Dining Philosopher's Problem

	Prototypes 1-4	Prototypes 5-6
Trace-1	1923	989
Trace-2	1952	960
Trace-3	1940	961

Table B.11.: Number of IOCTL calls made for Dining Philosophers Problem benchmark

	Prototypes 1-4	Prototypes 5-6
Trace-1	961	32
Trace-2	976	0
Trace-3	970	0

Table B.12.: Number of context switch calls made for Dining Philosophers Problem benchmark

B.3.1 Two Cores

Plain execution time(Unconstrained execution) is 461.486 μs.

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

B.3.2 Four Cores

Plain execution time(Unconstrained execution) is 531.877 μs.

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

B.3.3 Eight Cores

Plain execution time(Unconstrained execution) is 725.887 μs.

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

B.4 Fibonacci

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

Table B.16.: number of IOCTL calls made for Fibonacci benchmark

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

Table B.17.: Number of context switch calls made for Fibonacci benchmark

B.4.1 Two Cores

Plain execution time(Unconstrained execution) is 204.064 μs.

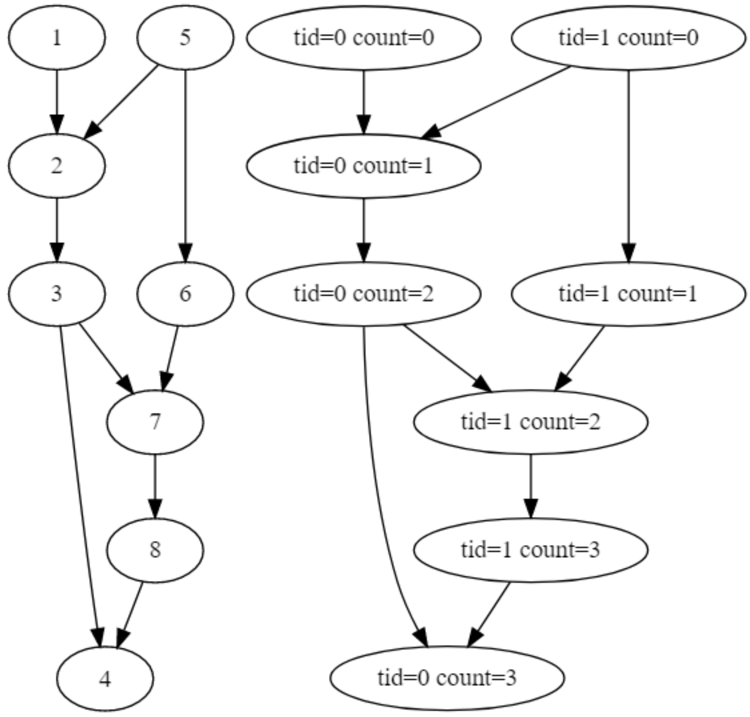
	IRS_Sh	IRS_Opt	Proto-1	Proto-2	Proto-3	Proto-4	Proto-5	Proto-6
Trace-1	2078.221	7944.79	406.833	454.785	385.416	455.745	277.793	275.343
Trace-2	1220.398	5673.862	367.199	520.352	352.506	509.843	160.266	160.307
Trace-3	381.5	5221.922	351.029	416.653	333.704	412.206	152.425	153.06

Table B.18.: Execution overhead(%) when compared with plain execution of Fibonacci

C Additional Explanations and Building Prototypes

C.1 Additional Explanations

C.1.1 Converting Trace to Vector Clock representation



(a) Trace without Annotation

(b) Trace with Annotation

Figure C.1.: Trace representation in Graphviz

Mazurkiewicz trace [22] is an equivalence class representation of a multithreaded program execution. Figure C.1a depicts an execution trace of a multithreaded program with two threads. Figure C.1b is the annotated equivalence of the trace in figure C.1a. The trace file used by the IRS user space implementation is implemented using graphviz library. The figures C.1a and C.1b are PNG representations of the graphviz file used as traces for a multithreaded program. From the trace file representation, we can assume that the multithreaded program uses two threads. There are in total of 4 shared memory events per thread considered in the above trace. However, there are only three memory constraints exhibited in the above trace. The three memory constraints converge at the nodes 2,7 and 4 respectively in the figure C.1a. Memory constraints are mainly the shared memory access dependencies between the threads. From the above trace, we can observe that there are 3 points of these dependencies among two threads.

In this thesis, we move the scheduling decisions to kernel space. The prototypes highlighted in chapter 4 address various scheduler designs in kernel space. The scheduler requires the trace

```
{
  (1,[1:1]),
  (2,[3:2]),
  (1,[3:4])
}
```

Figure C.2.: Vector clock representation of the trace

file as one of the inputs, as depicted in figure 4.1. However, as described in chapter 4 there are few design challenges in migrating the trace file represented as graphviz to kernel space. To overcome the challenge we have used an alternative representation of the trace to realize the scheduling constraints in kernel space scheduler module. Vector clock representation is the alternative representation considered for the above problem. Figure C.3 is the data-type realized in kernel space to store the vector clock representation. The trace representation depicted in figure C.1a would be transformed into a vector clock representation(vector clock representation is actually a string representation).

```
struct vec_clk{
   int clocks[THREAD_COUNT];
};
```

Figure C.3.: Vector clock data type in kernel space

```
struct trace_node{
   thread_id_t thread_id;
   vec_clk clk;
   int valid;
};
```

Figure C.4.: Trace Node Representation in kernel space

In our thesis, we have the thread ids in range from 1 to N. Therefore, in the above example thread id 0 is now 1 and thread id 1 is 2. Figure C.2 represents the vector clock representation of the trace depicted in figure C.1a. This string is generated manually and passed as an input of the trace_reg proc file. Let us disassemble the string to get a better understanding of how it is parsed in kernel space. The curly braces determine the start and end of a given trace file. The values within the parenthesis focuses on a single node in the graph which emphasises on a memory constraint for a certain thread. In our vector representation, we have $(1,[1:1])$ as the first memory constraint. The first value after the opening parenthesis is the thread id, in regard to the first node it is thread id 1. The values within the square brackets are the number of events completed by each thread. These values within the square brackets are separated by a colon symbol. In case of the first node, we have thread id waiting for the completion of an event each in threads with ids 1 and 2. The thread id 1 is only allowed to progress if the vector clock state is equal to or surpassed the above mentioned vectored state. The string is parsed in kernel space when passed as a parameter to the custom proc file trace_reg. Figure C.4 is the data type to which the parsed string is stored in kernel space.

C.1.2 Semaphores

Semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system [12][13]. Semaphores are useful programming abstractions used to prevent race conditions in concurrent programs. There are two types of semaphores - binary semaphore and counting semaphores. Counting semaphores are semaphores which allow an arbitrary resource count. Whereas binary semaphores are restricted to only two values 0 and 1. Binary semaphores are generally used to implement locks. In Linux kernel semaphores are realized under *linux/semaphore.h* [32]. Semaphores are defined using the C struct *struct semaphore* [32]. *sema_init()* is used to initialize a semaphore [32]. In semaphores, we have up and down operations which deal with the increment and decrement of the value stored in the semaphore. In semaphores when the value of the semaphore is 0 and if a down operation is performed, the semaphore would block the thread who called the down operation. The thread is only resumed until an up operation of semaphore is performed by some other thread. In Linux, we have different types of down operations and one up operation. *down()* is the conventional down operation which block the task until the semaphore value is retained to a positive value. *down_interruptible()* is similar to *down()* but, can be unblocked by triggering an interrupt from the user space. *down_try_lock()* is another implementation of down operation in Linux. It uses a busy waiting design to constantly check if the value is positive or not, and it is an interruptible design. Based on the performance and design requirements we use a suitable down operation. In this thesis, considering the debugging purposes we have used *down_interruptible()* as the down operation on the semaphore. In this thesis, we have used an array of binary semaphores, where each semaphore is meant for each thread used in the multithreaded program. The semaphore design is mainly used in Prototypes 1,2 and 5 designs used in this thesis. Listing C.1 provides a sample adaptation of semaphores in Linux kernel. The function *some_dec_operation()* is blocked on *down_interruptible()* call if value of *sem* is still 0 before the call. In *some_inc_operation()* function, we perform an *up* operation on *sem* thus, allowing any thread blocked on *sem* to resume.

Listing C.1: Writing a sample Linux kernel program with semaphores

```
struct semaphore sem;
//kernel module initialization method
module_init() {
        ...
        sema_init(&sem, 0);
}
...
void some_dec_operation() {
        ...
        down_interruptible(&sem);
        ...
}
void some_inc_operation() {
        ...
        up(&sem);
        ...
}
```

C.1.3 Scheduler APIs

Scheduler APIs are a programming abstraction provided by the Linux Kernel for interacting with the OS scheduler. The scheduler APIs are located inside the headerfile $linux/sched.h$ [31]. They provide a lot functions to interact with the scheduler. Some of them include the setting of the scheduler with a specific scheduling policy, setting CPU affinity of certain kernel level task, etc. We are more interested in the APIs which deal with the wait queue and run queue of the OS scheduler. All threads or processes from user space are realized as tasks in kernel space. For every kernel level task there is a $task\ struct$ associated with it. To know more about $task\ struct$ please check the headerfile $linux/sched.h$ [31]. In this thesis, we obtain the $task\ struct$ associated with the thread and push the task to the wait queue, when we need to block the thread associated with the task. The task would be later resumed by some other task by invoking another scheduler API. For the blocking a task, we have a combination of two methods. Initially, we call $set\_current\_task(TASK\_INTERRUPTIBLE)$ which sets the task state of the calling task to wait state(TASK_INTERRUPTIBLE means the task could be revived on some interrupts from the machine). After this step, we call the method $schedule()$ which would relinquish the thread's control over the CPU [31]. In short, with these two steps you have pushed the thread to wait queue of the OS scheduler. If the blocked thread needs to be revived, another task would invoke the method $wake\_up\_process(sleeping\_task)$ [31]. The task which invokes $wake\_up\_process()$ needs to have the $task\ struct$ of the blocked thread. Prototypes 3,4 and 6 mentioned in this thesis use this technique for blocking and unblocking the threads.

C.1.4 False Sharing

False sharing is a performance-degrading usage pattern which can arise in distributed systems [30]. This problem mainly occurs in SMP(Symmetric Multiprocessor) systems where each processor has a local cache. It occurs when threads on different processors modify different memory locations that reside on the same cache line. Since these modifications made to the memory locations are not the same locations on a global viewpoint, the cache invalidation occurs because of these modifications. Thus, leading to a false sharing of the memory locations among threads. Programming constructs such as C-structs are more susceptible to false sharing problems [30]. The vector clock data type depicted in figure C.3 is more susceptible to false sharing problem when the entire array is mapped under a single cache line. Since the C structs are susceptible to false sharing, our vector clock data type is susceptible as well. In chapter 5, we perform a scaling of processor cores and we can observe in those evaluations the prototypes depicted in this thesis suffer from scalability problems. False sharing could be regarded as one of these problems. One way to fix the problem would be removing the C-struct representation and represent the vector clock datatype as normal array of integers.

C.2 Building Prototypes

Prerequisites

The prototypes are only compatible with 64 bit Linux operating systems with kernel version needs to be greater than 3.0 and which support loadable modules. For running the IRS framework, you would require LLVM-CLANG 3.9, GCC/G++ 4.9, BOOST 1.6.2 and latest version of graphviz. You would additionally need the latest version of python and spdlog for generating debug logs.

Folder Structure

All prototypes used in this thesis follow the same folder structure and also the operation of building them. Inside each prototype, we have an *include*, *scheduler* and *Std_Thread_Impl*. *scheduler* folder deals with the business logic of the prototype which includes the scheduler module and processing of the trace. *include* folder contains three headerfiles - *common.h*, *kernel_space.h* and *user_space.h*. The headers inside *include* are mainly common headers used by the designs implemented in their respective scope(implementation scope means implementation is done either in kernel space or user space). *Std_Thread_Impl* contains the benchmarking programs used for the evaluation of the prototypes.

Loading and Unloading the Prototype

Before we load the prototype to kernel space, we need to know the number of threads used in the multithreaded program. Currently, all the prototypes have the *THREAD_COUNT* parameter as statically defined. You have to modify the macro entry *THREAD_COUNT* inside the headerfile *common.h* with the number of threads you have in your multithreaded program. Once that step is complete, you can load the kernel module by running the makefile located in the root directory of the prototype with target load. The command *make load* will compile and load the prototype to kernel space with the number of threads you have indicated in the *common.h* headerfile. The command *make unload* will remove the module from the kernel space and clean up the module versions from your machine.

Invoking Prototype related functions from a Multithreaded Program

We have five main functions which are defined in the headerfile *user_space.h*. These functions are *BeforeMA*(), *AfterMA*(), *initialize_trace*(), *reset_clock*() and *thread_reg*(). Currently, these prototypes are not integrated with LLVM therefore, methods such as *BeforeMA*() and *AfterMA*() needs to be invoked at the point of every shared memory event of the threads manually. But this functionality could be modified once integrated with LLVM. These two function expect the thread id which is value in the range of 1 to N. *initialize_trace*() is the first method which needs to be invoked from the multithreaded program. The method needs to be called before the creation of the threads and it expects a vector clock representation similar to the one shown in figure C.2, but without newline and white space characters. *thread_reg*() needs to be called first within the thread function for registering the thread and it expects a thread id similar to *BeforeMA*(). *reset_clock*() is the final method which needs to be used with the multithreaded program. This method needs to be used once all the threads have completed their execution. The program needs to be run in superuser mode because the IOCTL device created when the module was loaded requires root permission for access. The multithreaded program should only be compiled and executed until the all kernel modules of the prototype are loaded successfully.

- API - Application Programming Interface
- BDD - Binary Decision Diagrams
- CPU - Central Processing Unit
- DMT - Deterministic Multithreading
- IOCTL - Input Output Control
- IPC - Inter Process Communication
- IRS - Iterative Relaxed Scheduling
- IRS_Opt - Another user space IRS solution discussed in section 2.6. It uses an additional thread for handling the scheduling decisions and uses conditional variables to block a certain thread when memory access is restricted.
- IRS_Sh - One of the IRS user space implementation discussed in section 2.6. It addresses the use of a busy waiting design for blocking a thread and use the other threads in the thread pool to signal the blocked thread. Thus, making the scheduling decision shared among the threads. This design does not have an additional thread for handling the scheduling decisions.
- LKM - Loadable Kernel Module
- OS - Operating System
- POR - Partial Order Reduction
- POSIX - Portable Operating System Interface
- ProcFS - Process File System, a virtual file system in Linux Operating System.
- Proto_1 - Section 4.3 highlights all the prototypes used in this thesis to provide various solutions addressing the transition of scheduling decision to kernel space. Prototype 1 is the first synchronization discussed in section 4.3 which uses a shared scheduler design similar to IRS_Sh but, the blocking of threads is enforced by using semaphores in kernel space.
- Proto_2 - Prototype 2 is an extension of the previous prototype. Prototype also uses semaphores in kernel space for blocking a given thread but, the signaling the blocked thread is done by an additional thread which is similar to IRS_Opt.
- Proto_3 - Prototype 3 is a lot similar to Prototype 1 in the nature of behavior when it comes to design and its approach to scheduling. Main difference of Prototype 3 compared to Prototype 1 is the use of scheduler APIs instead of semaphores for blocking a given thread.
- Proto_4 - Prototype 4 is a design variant of Prototype 2. It uses the same scheduling approach as Prototype 2 but, differs in the blocking of a given thread. It uses scheduler APIs instead of semaphores.
- Proto_5 - Prototype 5 are an extension of Prototype 2, it is one of the designs which addresses the second approach discussed in section 4.1.2. As discussed in section 4.1.2 it uses a proxy checking of memory access permission in user space.
- Proto_6 - Prototype 6 are an extension of Prototype 4, it is another design which addresses the second approach discussed in section 4.1.2. As discussed in section 4.1.2 it uses a proxy checking of memory access permission in user space.
- Pthreads- POSIX Threads
- RTID - Real Thread ID which is assigned within the proc file system for any thread created within the user land.
- SMP - Symmetric Multi-Processor

- StableMT - Stable Multithreading
- TaskID - All threads are internally realized as tasks in kernel space and are allocated with an identifier which is task Identifier.
- TLB - Translation Lookaside Buffer
- UTID - User defined thread ID which is relative inside the user program.
- VM - Virtual machine