MÄLARDALEN UNIVERSITY
SWEDEN

Efficiency of LTTng as a Kernel and Userspace Tracer on Multicore Environment

Master Thesis in Software Engineering
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

June, 2010

Romik Guha Anjoy
romik.03305@gmail.com

Soumya Kanti Chakraborty
soumyakanti.chakraborty@gmail.com

Supervisors:
Alf Larsson, Ericsson AB and Daniel Flemström, Mälardalen University

Examiner:
Daniel Sundmark, Mälardalen University

Preface

This document is a Master Thesis in the field of Software Engineering. The work described here has been accomplished at the Industrial Research and Innovation Laboratory (IRIL), School of Innovation, Design and Engineering, Mälardalen University in Västerås and at Ericsson AB in Stockholm in a time span between January 2010 and June 2010.

We are grateful to Mathieu Desnoyers and Pierre-Marc Fournier from the *ltt-dev* team for providing us with valuable information about LTTng and for fixing the reported bugs quickly. We are also grateful to Francois Chouinard from Ericsson Canada and the *eclipse-dev* team for helping us.

We are thankful to Sten Ekman for his *Innovation Science& Management* course which equipped us with skills to deal with the research work efficiently. We are also thankful to Daniel Sundmark for reviewing the initial draft of our thesis document and providing his valuable feedback.

Special thanks go to our supervisor Daniel Flemström who dedicated much time supervising us and pointing us in the right directions. Alf Larsson, our supervisor at Ericsson AB, also deserves special thanks for his invaluable feedback, insight and orientations in the field.

Examiner at Mälardalen University

Daniel Sundmark
daniel.sundmark@mdh.se

Supervisor at Mälardalen University

Daniel Flemström
daniel.flemstrom@mdh.se

Supervisor at Ericsson AB

Alf Larsson
alf.larsson@ericsson.com

Abstract

With the advent of huge multicore processors, complex hardware, intermingled networks and huge disk storage capabilities the programs that are used in the system and the code which is written to control them are increasingly getting large and often much complicated. There is increase in need of a framework which tracks issues, debugs the program, helps to analyze the reason behind degradation of system and program performance. Another big concern for deploying such a framework in complex systems is to the footprint of the framework upon the setup. LTTng project aims to provide such an effective tracing and debugging toolset for Linux systems. Our work is to measure the effectiveness of LTTng in a Multicore Environment and evaluate its affect on the system and program performance. We incorporate Control and Data Flow analysis of the system and the binaries of LTTng to reach for a conclusion.

Thesis Summary

The Goal of the thesis is to analyze the performance of LTTng kernel and Userspace tracer in a multicore environment under various load configurations. Control and Data Flow analysis of the system and the application binaries is carried out to evaluate the performance measurements of the tracing tool. With Control Flow analysis, we annotate source code of application binaries, measure the internal CPU cycles usage, perform a call graph analysis to draw a picture about the necessary calls made by the program and the tool during testing. Data Flow analysis helps us to find out the memory performances of the tracing utility and its memory leaks under different load configurations.

The experiments we performed in course of finding the efficiency of LTTng kernel tracer and the userspace tracer are:

- **Experiment 1** – Determination of load configuration parameters for System Under Test (SUT)
- **Experiment 2** – Measuring the efficiency of LTTng Kernel Tracer
- **Experiment 3** – Measuring the efficiency of LTTng Userspace Tracer
- **Experiment 4** – Measuring the impact on System as well as Traced Application when LTTng Kernel Tracer and Userspace Tracer are executed together
- **Experiment 5** – Running load program and tbench on LTTng Kernel with Non Overwrite and Flight Recorder tracing modes
- **Experiment 6** – Running UST tracing on load and tbench program each instrumented with 10 markers under different load configurations
- **Experiment 7** – Running the Kernel tracer with the help of Valgrind under various load configurations generated by load program (system load) and tbench (process and network load)
- **Experiment 8** – Running the load and tbench application instrumented with 10 markers under UST (Userspace Tracing) with the help of Valgrind

The findings from these experiments have enabled us to conclude on the following points:

- The impact of LTTng kernel tracer on kernel operations against vanilla kernel is 1.6%
- There is almost negligible difference between the performances of LTTng kernel tracer in Non Overwrite mode and in Flight Recorder mode
- The LTTng userspace tracer and the compiled markers both have an effect of around 0.50% on the performance of the userspace application against the original copy of the applications without markers
- The impact of UST on userspace applications marginally increase with the increase in the number of instrumentations compiled in, though the pattern of increase for all load configurations are not similar

- LTT Control and Trace Daemon have minimal cache miss and Branch Misprediction rate in order of 10^{-4} percent
- Branch Mispredictions of both LTTng Kernel Tracer and UST decreases significantly with increase in load. Memory handling thus becomes more efficient with load increase
- LTT kernel Tracing Daemon is much more memory efficient than UST Daemon
- Memory loss though is of insignificant number but is more for UST tracing. UST also has problem of not freeing a chunk of memory after completion of execution
- The impact of LTTng kernel tracer and UST together is quite similar to LTTng kernel tracer and there is no additional impact on the percentage of CPU cycles needs to perform kernel operations

Contents

Index of Tables

Index of Graphs

Index of Figures

1. Introduction

Ericsson as a company is rapidly growing in telecom sector with deployment of advanced technologies and increase in its user base. Slowly due to the pressure of the industry and hunger for more performance, Ericsson has moved into multicore processors and PowerPC architectures. Multicore architectures help to reduce footprint through virtualization, replacing many small processor boards and packing it into one slot and thus giving better and higher performance per slot and much more value for money. In a huge multicore system, it's often difficult to track problems, issues and performance degradations. Many problems occur only once and do not repeat its behavior, and it's a pain for the developers to look for it in large multiprocessor and multicore systems. LTTng provides a highly efficient set of tracing tools for Linux which is used for resolving performance issues and troubleshooting problems. Ericsson is in need for such a tool which can help it's developers to backtrack and debug the problems and errors in the system. The research question catered in our thesis is to test the efficiency of LTTng as a kernel and userspace tracer in a multicore environment. As even nanoseconds of delay can cause performance degradations for telecommunication systems, we need to gauge the footprint of LTTng over a multicore system and in case the tool has pretty low overhead, Ericsson can deploy it on the system for helping its developers to effectively backtrack the performance loopholes.

1.1 Organization of Thesis

The thesis document is organized in several chapters:

The chapter *Problem Formulation* introduces the problem statement of our thesis work. Then it divides the problem into several sub-problems and finally tries to offer a solution to the bigger problem by solving the smaller problems.

The chapter *Background* provides the background knowledge on the technologies on which our thesis work is based. It explains the basics of tracing and then provides an insight of embedded systems and multicore systems. Then it provides detailed information about LTTng kernel and userspace tracer architecture and functionality. Finally this chapter wraps up with the description of the lab environment on which our experiments have been executed.

The chapter *Related Work* starts off with an overview of our goal. Then it explains our search methodology. Finally, it concentrates on citation of the previous work done that was useful to us to proceed in the correct direction and helped us making the correct decisions throughout our thesis work. It also describes similar work done in past.

The chapter *Experiment Setup* describes in detail the use of technologies in our experiments. These technologies include the hardware and software configurations and tools, utilities and scripts used to perform the experiments.

The chapter *Experiment Methodology* describes the experiment methods in detail that are to be performed on LTTng kernel tracer and userspace tracer.

The chapter *Results* presents the analysis of results that are obtained by performing the experiments mentioned in the experiment methodology chapter.

The chapter *Discussion* mainly focuses upon the constraints of experiments executed and the issues faced during the research period. The issues discussed concentrates upon the unavailability of tools and time limitation of the thesis standing as the main barriers. Last part of the Discussion aims to evaluate the benefits of this research to the community and the industry.

The chapter *Conclusion* focuses on the important findings from the experiments performed in course of this research work and tries to draw a conclusion from the findings.

The chapter *Future Work* throws light on the possibilities of continuing our research work. These also include extending our research by overcoming the limitations we faced.

The references are organized in the final chapter called *References*.

2. Problem Formulation

This chapter introduces the problem statement of our thesis work. Then it divides the problem into several sub-problems and finally tries to offer a solution to the bigger problem by solving the smaller problems.

List of technical terms

LTTng	Linux Trace Toolkit Next Generation
TCF	Target Communication Framework

2.1 Problem Statement

Modern day systems are becoming more complex which invites the need of an effective and high performance trace mechanism. LTTng being developed as a next generation tracing utility for Linux supports both kernel space and user space tracing and claims to perform high performance tracing with a very low overhead on the system. LTTng has the capability to dump the trace data either to the disk or to the network.

The primary question that our research is going to address is:

How efficient is LTTng as a kernel as well as userspace tracer on a multicore environment?

2.2 Problem Analysis

To quantify the efficiency of tracing utility it's very necessary to size down the fingerprints of the tool on the system or on the other applications running in it. To measure the fingerprint on the system various data and control flow analysis on LTTng modules should be carried out. This will help us to get a broader picture for the fingerprint involving details of how it affected the system or the user programs.

Thus the first and preliminary refinement of our research question stands as:

1. *How does LTTng affect the control flow and data flow in kernelspace as well as userspace on a multicore environment?*

LTTng has a trace viewer called LTTV which helps to view the trace generated by LTTng in a GUI environment thus helping the end user to view the system trace effectively with control flow charts and resource viewers. For multicore AMP systems the efficiency of LTTng can be increased if the tracing can be controlled remotely from another system and also if it can stream the trace over network effectively.

Thus our research question can be refined to the below question:

2. *How to efficiently stream the trace data for multicore systems to remote host (Eclipse)?*

Eclipse team is in process to develop LTTng integration tool.

The above mentioned sub problems are elaborated below which ultimately leads us to the final aim of the thesis.

- *How does LTTng affect the control flow and data flow in kernelspace as well as userspace on a multicore environment?*

Control Flow analysis involves the use of a profiler to scan through the function and system calls of the events in the multicore environment. Data Flow study on the events focus on the Cache Behavior and Data Path analysis. All together these both help us to reach to the conclusion regarding the effect of LTTng on a multicore system.

- *How to efficiently stream the trace data for multicore systems to remote host (Eclipse)?*

The general approach will be to use a network protocol called Target Communication Framework (TCF) for streaming the trace data to Eclipse. We will measure the efficiency of LTTng in streaming huge amount of trace data, gathered from different cores, over the network to Eclipse. The results will enable us to narrow down on an optimal architecture to stream the LTTng trace on a multicore system.

The results of the sub-problems will enable us to comment on the efficiency of LTTng as a kernel as well as userspace tracer on multicore environment.

All the above discussion on the problem statement and analysis marks the need of a brief background study on tracing, embedded systems, multicore environment, LTTng, Control Flow and Data Flow analysis. We also discuss in the forthcoming Related Work chapter about similar work and experiments carried out by other researchers and also some other useful work which helps us to get a correct direction to proceed forward with our study and analysis.

3. Background

This chapter provides the background knowledge on the technologies on which our thesis work is based. It explains the basics of tracing and then provides an insight of embedded systems and multicore systems. Then it provides detailed information about LTTng kernel and userspace tracer architecture and functionality. Finally this chapter wraps up with the description of the lab environment on which our experiments have been executed.

List of technical terms

PC	Personal Computer
ASIP	Application Specific Instruction Set
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
I/O	Input / Output
AC	Alternating Current
DMA	Direct Memory Access
GPP	General Purpose Processor
AMP	Asymmetric Multiprocessing
SMP	Symmetric Multiprocessing
LTTng	Linux Trace Toolkit Next Generation
UST	Userspace Tracer
LTTV	LTT Viewer
RCU	Read Copy Update
OMAP	Open Multimedia Application Platform
MIPS	Microprocessor without Interlocked Pipeline Stages
NMI	Non-Maskable Interrupt
PID	Process ID
GUI	Graphical User Interface
IBM	International Business Machine

3.1 Tracing

Tracing is a mechanism to identify and analyze the behavior of a system. Tracing is a technique of recording low level events that frequently occur in a system along with the timestamps and attributes of the events [SHE99]. A tool that performs tracing on a system is known as *tracer*. A tracer records a huge number of events that occur in a system in a period of time and generates large amounts of data known as *traces*. The size of a trace may vary from a few megabytes to several gigabytes [LTT10].

A tracer generally records operating system kernel events that include [LTT10]:

- Interrupt Requests
- System Calls
- Scheduling Activities
- Network Activities

A tracer may also be capable of recording events that are generated by an application.

Equally important is to present the trace data in a meaningful way to the user. A *trace analyzer* or *trace viewer* is an application that produces graphs and statistics from the trace data generated by the tracer [LTT10].

Tracing helps in the following activities [LTT10][SHE99]:

- **Debugging:** A tracer helps to identify performance bugs and bottlenecks in complex parallel systems and real time systems.

- **Monitoring:** A tracer helps to maintain and analyze statics of events and their timestamps, attributes and flow of control from one event to another. These data may be utilized in a lot of different activities.

Tracing a system involves the following steps [SHE99]:

- **Instrumentation:** Instrumentation is the modification of source code of an application where instructions are added to the program that helps to generate trace data.

- **Measurement:** Recording different aspects of execution of an application such as resources consumed, mapping of these resources to the application routines and statements.

- **Analysis:** Analysis of the performance data that is generated in the subsequent phases of *instrumentation* and *measurement*.

3.2 Embedded Systems

As technology is climbing new heights we need more and more systems which are standalone and can work without human intervention. An embedded system is a microprocessor-base system that is built to control a function or range of functions and is not programmed by the end user in the same way that a PC [Sér02]. Often embedded systems also handle time critical applications which require utmost time precision. It can respond, monitor and control the external environment using sensors and actuators and is based upon application level processors. One of the major considerations when designing an embedded system is the consumption of power, which should always be less whether it is battery driven or wall powered. Manufacturing cost is an important aspect to be maintained during design of Embedded Systems.

3.2.1 Classes of Embedded Systems

Embedded Systems can be typically categorized into two different subclasses [NR98]. They are:

- **Embedded Controllers:** *Embedded Controllers* are those which are dedicated to control particular functions and are thus reactive to external environmental events. Control systems react to external stimuli by changing its internal state and producing desired result. Home appliances can be cited as a example for Embedded Controllers.

- **Embedded Data Processing Systems:** *Embedded data processing systems* are also called *transformational systems* as they are dedicated to communication and data processing. They are data flow dominated real time systems that execute a special function within a predefined time window. These systems require much higher performance than the *embedded controllers* and thus require powerful microprocessors like ASIP (Application Specific Instruction Set) and circuits like ASIC (Application Specific Integrated Circuit). Audio/Video Application and Wireless Communicators can be cited a example for this

3.2.2 Challenges in Embedded Systems Design

In case of design requirements embedded systems face several challenges [Sér02][Tam05]. They are:

1) **Physical size and weight restrictions**: It varies greatly with the application. The high performance processing systems tends to be much larger and heavier than the slower systems. At system level design higher cache memory needs bigger circuit boards and at CPU level the board size increases if there is increase in number of pines.

2) **Performance**: The main performance metrics are instructions per second, interrupt response characteristics, context switching overhead and I/O performance.

3) **Power Consumption**: It should always be low in case of battery driven equipments. For direct AC powered systems the power consumption should be kept minimal to reduce heat generation or increase of cooling requirements.

4) **Cost of the embedded system:** Manufacturing cost includes cost of components and assembly. Non-recurring engineering cost which includes personnel and other cost of system designing.

5) **Reliability**: Embedded systems can be subjected to extreme operative conditions like in military or automotive sectors. The embedded system should be properly functional at extreme conditions and deliver results within its time boundaries.

3.2.3 Real Time Architecture Constraints

Embedded systems have two important performance rules to be maintained, *predictability* and *determinacy*. In many occasions embedded systems work in real time environments in which it must finish operation by certain deadlines failing to which can cause major disasters or in some cases degradation of performance. There are many architectural features which are considered as inappropriate for hard real time embedded systems [KP90][Sér02] and are discussed below -

Cache Memory provides the biggest source of unpredictability and non-determinacy. The problem is with scheduling the instruction cache misses because the time required to process a cache miss is a magnitude slower than processing a cache hit. If smaller percentage of cache miss is present during the execution, it dedicatedly reduces the speed of operation. Thus hard real time embedded system hardware is designed with fast static memory chip that renders cache management hardware superfluous. There is also dependability on the variable execution time of the instructions, as depending upon the input data, different instruction sets take variable number of clock cycles for the execution.

With **write buffers** CPU can perform write operation to the memory without waiting for the memory cycle to occur. Processor must be stalled in case the write buffer overflows and there are no subsequent free bus cycles. There should also be additional stalls in case the memory read corresponds to a memory location that is yet to be updated by the write buffer. The interaction which takes place between cache misses, write buffer and data fetching causes loss of both predictability and determinacy of the Embedded System.

Interrupt Response Latency increases with deep instruction **pipelines**. An instruction takes several clock cycles to pass through pipelines and perform its job. The pipeline also needs some handling of delays in memory access or data dependencies which results in either software generated instructions rearrangement or hardware generated pipeline halts which results in unpredictability and non-deterministic behavior. Multiple instructions can be issued in a single clock cycle by the microprocessor. The number of instructions that can be issued together depends on the type of instructions, the

available hardware resources and execution history of the program. Thus these all factors make it very difficult to calculate single instruction executed time.

Branch Target Buffers is a mechanism in which the program execution history is used for caching instructions at branch targets for faster access. Branch target buffers are used with the context of branch prediction strategies in which compiler guesses which branch is to be taken by the instruction causing to fetch the next instruction or branch goal before the outcome of the ongoing instruction. The challenge occurs to calculate the branch completion time as it depends on the matching of branch target buffer value and the compiler guess.

Prefetch Queues affects the predictability of an operation because the time required for completion of the instruction cycle is solely determined by the fact that whether the preceding instructions were slow enough to allow the Prefetch queue to accumulate new instructions. Thus to determine execution time of one instruction cycle, it is required to determine the clock (depends upon data dependent path, cache misses etc) for several preceding instruction cycles so that there are free memory cycles or not for the Prefetch queue to fill.

3.3 Multicore Systems

In the computer industry, the customers always expect faster and more powerful systems. There is a persistent need of increase in performance in the computer industry, be it embedded systems or desktop computing.

Multicore processing comes into the picture as a key to continuous improvement of performance according to the consumers' expectations. The cope up with the consumers' expectations is a real challenge not only for the semiconductor industry but also for the software industry.

3.3.1 Migration from Single Core to Multicore

The multicore systems can increase the performance of multi-threaded applications significantly by increasing the processing power but with a relatively low latency. The migration from single core systems to multicore systems requires considerable changes to the system as well as to software. Therefore, the factors that have driven the semiconductor industry to migrate from single core to multicore systems should be worth the additional work that is required to be done.

The most prominent driving factors [SLES09] for the migration from single core to multicore are:

- **Performance**

  A way to increase the performance of a single core system is to increase the core frequency. But, pushing up the core frequency does not always increase the performance of the system in the same proportion. The techniques like parallelism and pipelining does not always scale with the frequency. It is not always easy for a single core processer to handle Parallel Processing timeline issues. If the frequency of the core does not match with frequency of off-chip memory and I/O subsystems the core may have to wait for the off-chip busses to transfer data. This particular phenomenon is called *memory wall*.

- **Power Consumption**

  The power consumption for a core to operate is proportional to the frequency of the core. Therefore, doubling the frequency of a core to gain performance increases the power consumption by four times. The equation presented below shows the relation between power and frequency.

$$power = capacitance \times voltage^2 \times frequency$$

To overcome the processor and off-chip memory and I/O subsystems frequency lag, large fast on-chip caches have been implemented which increases power consumption. An efficient cooling system will consume power, whereas if the generated heat is substantially less the core can reside even without a cooling mechanism.

- **Simplicity in Design**

  Multicore architecture enables less complicated or no cooling mechanisms and better performance with smaller caches. These contribute to simpler board design rather than increasing the frequency of a single core.

- **Cost**

  Multicore architecture enables less complicated or no cooling mechanisms and better performance with smaller caches. These contribute to comparatively lesser costs rather than increasing the frequency of a single core.

3.3.2 Parallelism in Multicore Processing

Parallelism is an important feature for modern day computing. Most of the modern systems are equipped with parallelism. The different types of parallelism [SLES09] that are implemented in multicore processing are:

- Bit Level Parallelism
- Instruction Level Parallelism
- Data Parallelism
- Task Parallelism

Bit Level Parallelism enables the hardware to operate on larger data. For example, if an 8-bit core is performing computation on a 16-bit data, it will need two instruction cycles to complete the computation. Therefore by increasing the word length from 8 to 16 will enable the processor to do the computation in one instruction cycle. Currently we have 64-bit word length to perform computation on large data in a single instruction cycle.

Instruction Level Parallelism is the method of identifying the instructions that does not depend on one another and processing them simultaneously.

Data Parallelism is the technique of processing multiple data in a single instruction cycle. In multicore architecture, performance improvement depends on different cores being able to work on the data at the same time.

Task Parallelism is the method to distribute the applications, processes and threads to different units for processing.

3.3.3 Types of Multicore

The multicore systems can be categorized into two distinct types based on the core topology [SLES09]:

Homogeneous Multicore System

A *homogeneous* multicore system consists of identical cores that execute the same instruction set.

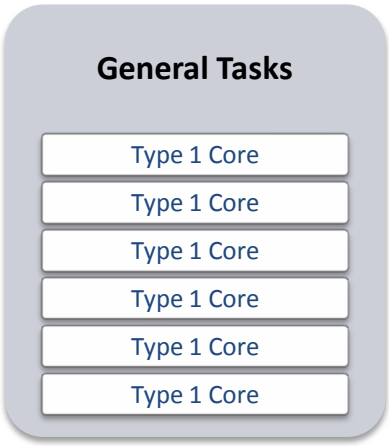


Figure 4.1: Homogeneous Multicore Environment

Heterogeneous Multicore System

A *heterogeneous* system consists of cores that are not identical. Here, different types of cores execute different instruction sets.

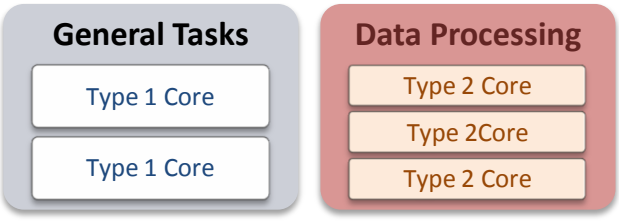


Figure 4.2: Heterogeneous Multicore Environment

The multicore systems can be categorized into the following types based on the memory topology [SLES09]:

Distributed Memory Multicore System

In *distributed memory* multicore systems each core has a private memory. The communication between the cores takes place over a high speed network.

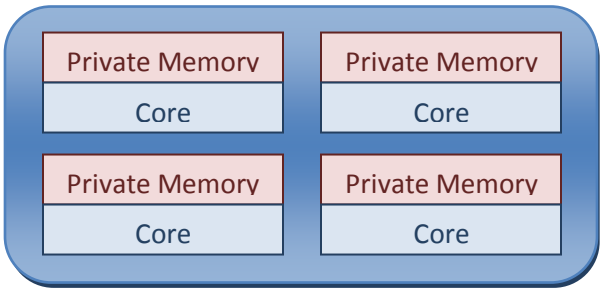


Figure 4.3: Distributed Memory Multicore Environment

Shared Memory Multicore System

In *shared memory* multicore systems there is a common memory which is shared by all cores in the system.

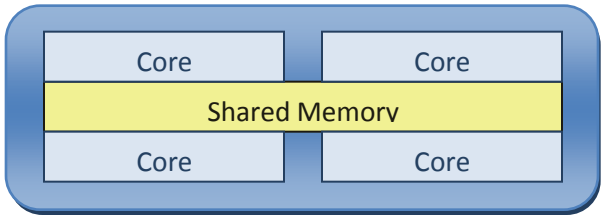


Figure 4.4: Shared Memory Multicore Environment

Hybrid Memory Multicore System

In *hybrid memory* multicore systems there is a common memory which is shared across all cores in the system. Each core also has its own private memory as well.

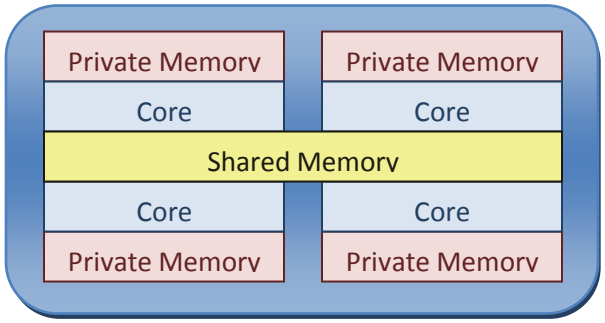


Figure 4.5: Hybrid Memory Multicore Environment

3.3.4 Inter-core Communication

In a multicore system it is very important for the individual cores to communicate within themselves. In most multicore systems the cores can be connected to each other with the help of *high speed buses* or *coherent communication fabric* [SLES09].

The common network topologies in which the cores can be interconnected are bus, mesh, ring or crossbar. The cores may also share caches or memory as a part of inter-core communication.

3.3.5 Multicore Design Approaches

The multicore system architecture focuses mostly on data and task parallelism. Multicore design approaches vary depending on the data management and handling of tasks [SLES09]. They are:

- Asymmetric Multiprocessing (AMP)
- Symmetric Multiprocessing (SMP)

In *asymmetric multiprocessing* design each core operates independently and performs dedicated tasks. Each core has its own logically or physically separated memory and can run operating systems independent of the other cores. The operating system running on different cores communicate with a help of *hypervisor*. The cores can either be *homogeneous* or *heterogeneous* in type.

In *symmetric multiprocessing* design all the cores share the same memory, operating system and other resources. The operating system takes care of the distribution and tasks and resources across the cores. The cores should be *homogenous* in type in order to support symmetric multiprocessing.

3.3.6 Problems in Multicore Systems

However, a few problems still exist in multicore systems [MUC09]. The memory performance does not match the core performance thus creating a bottleneck, which results in starvation of cores. It is not easy to create algorithms having independent tasks to execute on different cores simultaneously.

3.4 LTTng

3.4.1 Overview

LTTng is an effective tracing platform that has been developed to take over its previous version, the Linux Trace Toolkit [LTT00]. The LTTng Project provides effective kernel space and user space tracing solutions for Linux platforms for performance monitoring and debugging. The LTTng Project comprises of the following tracing tools [LTT10]:

- LTTng Kernel Tracer
- LTTng Userspace Tracer (UST)
- LTT Viewer (LTTV)

3.4.2 Features of LTTng

LTTng was developed keeping in mind the requirements that a new generation tracing application should provide [DD06]. The most distinctive features [LTT10] sported by the latest release of LTTng are:

- **Integrated tracing of kernel space and user space:** LTTng provides a way of tracing the kernel as well as the applications that are present in the user space simultaneously. The LTTng userspace tracer can be used along with the kernel tracer for effective debugging and performance monitoring.
- **High performance yet Low Impact:** LTTng provides effective tracing probes without any system calls and a good instrumentation coverage for kernel tracing that helps to get a detailed analysis of the performance of the system. LTTng has very low observer effect on the traced system. This is essentially done using *userspace RCU*, *atomic data structures* to have really *lockless algorithms* and *cache optimization*. Inactive instrumentation has almost negligible performance impact on the system. Active instrumentation points have a very low performance impact.
- **Timestamp precision:** LTTng provides effective clock synchronization technique for maintaining timestamp precision for events.
- **Security:** LTTng has been designed keeping in mind that it has to be deployed in Linux production systems where security is an issue. The flow of data between kernel and userspace might not be acceptable in production environment. Therefore, use of *per-CPU buffers* for reading and writing by kernel or a user process keeps it fit for use in production environment.
- **Portable:** LTTng is portable to various system architectures. The latest release of LTTng kernel tracer supports x86-32, x86-64, PowerPC 32/64, ARMv7 OMAP3, MIPS, sh, sparc64, s390. The latest release of LTTng userspace tracer supports x86-32 and x86-64.

- **Scalable:** The LTTng tracer supports multiple CPU cores and a wide range of CPU frequencies with very little impact in performance.
- **Flexible and extensible:** LTTng provides the flexibility to add custom instrumentation. It also provides an easy to use interface for trace analysis which is also extensible for adding new functionalities for trace analysis.
- **Reentrancy:** LTTng provides complete NMI-reentrancy to ensure that NMI nesting does not cause deadlocks in the system.

3.4.3 LTTng Tracer Architecture

To perform extensive analysis of traces the trace data is extracted from the kernel. The tracing process has been divided into two phases, *tracing phase* and *post processing phase* [DES09]. Between the tracing phase and the post processing phase, *Input/output* represents the extraction of trace data to disk or network. Figure 4.6 presents the LTTng architecture with the two phases.

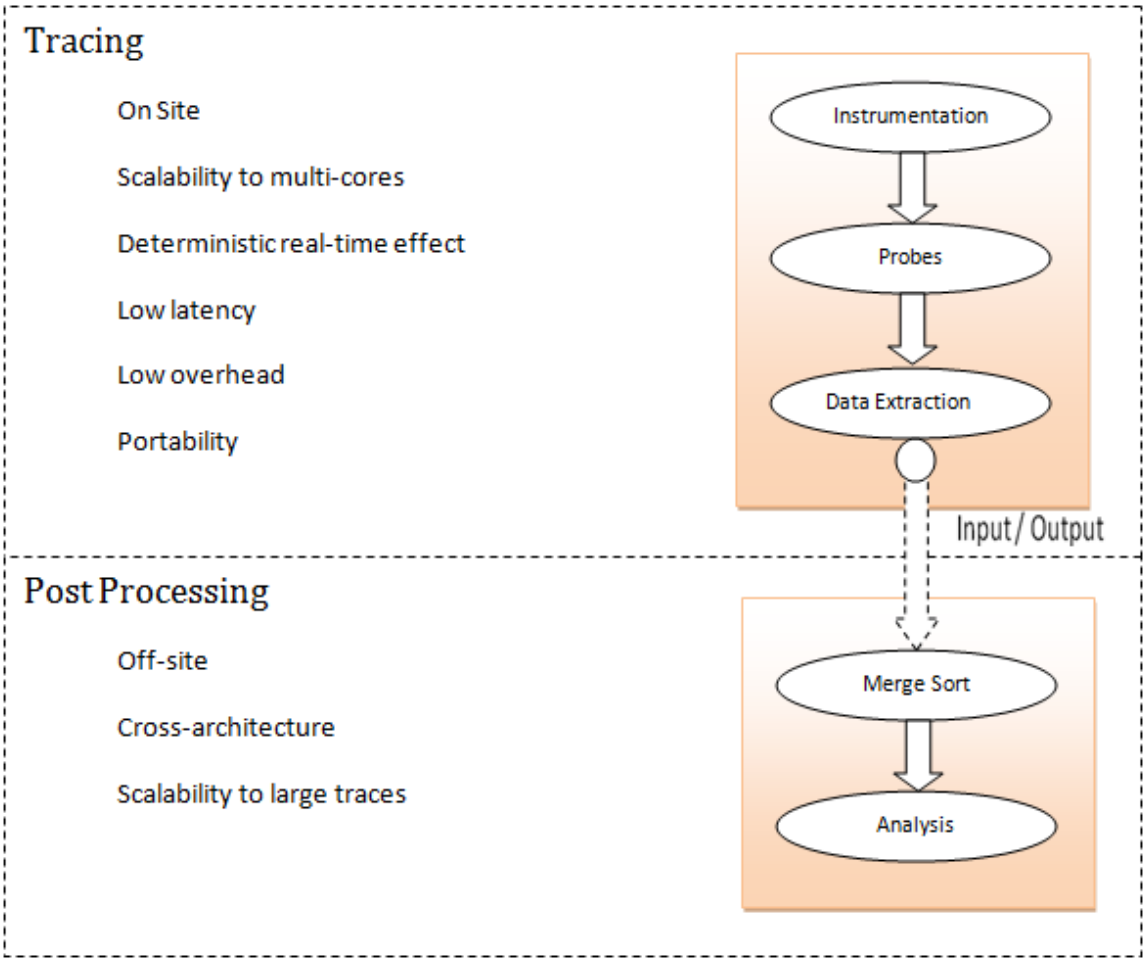


Figure 4.6: LTTng Tracer Architecture [DES09]

The *tracing phase* is carried out in the target system, which uses processor, memory and I/O resources. Initially the kernel is patched by inserting the *instrumentation* in the kernel. When the kernel reaches an instrumentation point, it verifies if the instrumentation point is activated, it calls the tracing probes attached to that instrumentation site. The probes write the trace event data into circular buffers in a synchronized manner. Trace data can be extracted in two different modes [DES09]:

- **Flight Recorder Mode:** Trace data is not extracted when the circular buffers are full. Eventually, when the trace is stopped that latest data in the buffers is extracted. This mode of tracing in called *flight recorder mode*.
- **Non-Overwrite Mode:** Trace data is extracted whenever the circular buffers are full. Therefore, trace data is not lost. This mode of tracing is called *non-overwrite mode*.

I/O operations required to write the trace data to the disk or network are costly, therefore not done by the probes. There are specialized threads for performing the I/O operations. It can be done while the tracing is being done as well as when the trace session is over.

To minimize the effect on the system's performance while extracting large amount of trace data a *zero-copy* approach has been taken at LTTng design level while data extraction. A *zero-copy* approach ensures that no trace data is copied between memory locations in the tracing phases. This also ensures an effective use of memory bandwidth.

The recording of the events in the *post processing phase* is done by collecting a timestamp value from the traced processor and then the probe writes the timestamp value to the event header. The timestamp is a time-source that is synchronized between all the traced processors.

The tracing phase and post processing phase may be performed in the same environment or it might be in different environments. Therefore, the trace output is a self described binary file for easy extraction and portability.

3.4.4 LTTng Design

The kernel code can be instrumented in two ways:

- Static Instrumentation at source code level using Linux kernel markers and tracepoints
- Dynamic instrumentation using Kprobes

When an active instrumented code is reached during the execution of the kernel, the *LTTng probe* is called. The probe reads the *trace session* and writes the events into *channels*. Figure 4.7 portrays the different components of LTTng kernel tracer and their interactions [DES09].

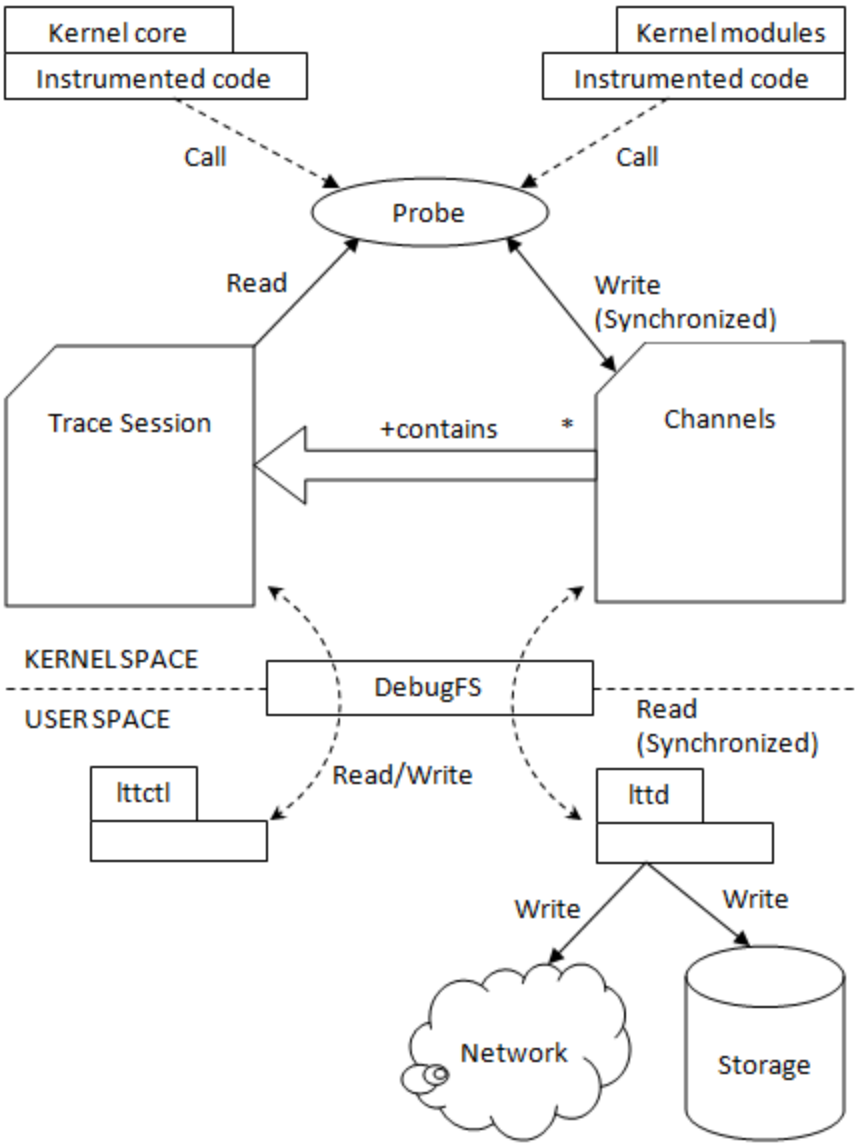


Figure 4.7: LTTng Tracer Components [DES09]

Trace Session: A trace session consists of the trace configuration and a set of channels that are to be used by the trace session. A trace session can have several channels attached to it. The trace configuration consists of the following data:

- Trace session is active or not
- The event filters to be applied

Channel: A channel is an information pipe between the writer and the reader. It acts a buffer to transport data efficiently. A channel consists of one buffer per CPU eliminate *false sharing* and at the same time having a *cache locality*. A few properties can be configured at the time of trace session creation. The configurable properties of channels are:

- Tracing mode
- Buffer size
- Buffer flush period

A channel in turn is composed of several sub-buffers and in each sub-buffer *slots* are reserved by the LTTng probes to write the event data into them. The *lttd* extracts each sub-buffer separately to the disk or network. The components of a channel are displayed in Figure 4.8.

Slot: A *slot* is a part of sub-buffer reserved for exclusive write access by the LTTng probe. The data written by the probe to each slot is the *sub-buffer header*, *event header* or *payload*.

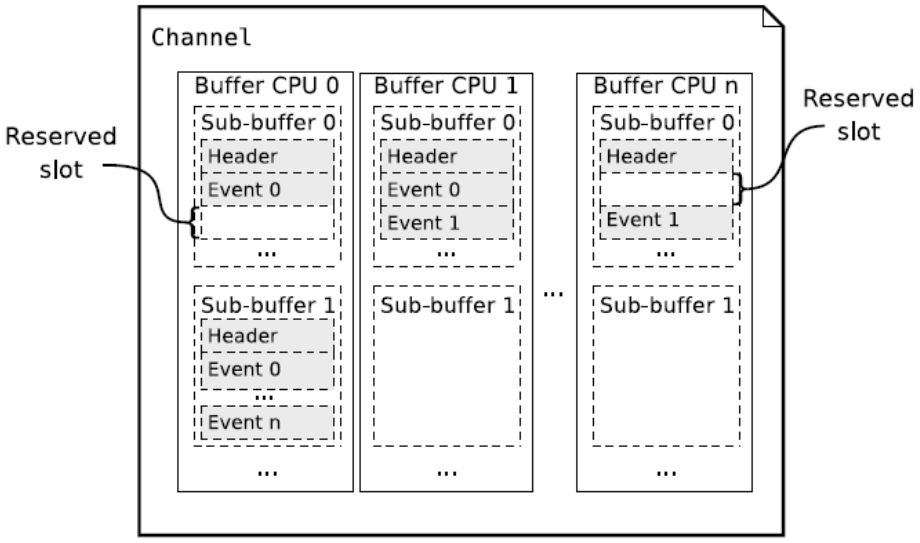


Figure 4.8: LTTng Channel Components [DES09]

DebugFS: *DebugFS* is a virtual file system which provides an interface to extract data from kernelspace to userspace. The trace session and channel data structures are represented as DebugFS virtual files so that *lttctl* and *lttd* can interact with them.

lttctl: The command-line application *lttctl* is an interface that interacts with the DebugFS in order to control the kernel tracing. The *lttctl* is responsible for the following:

- Configuration of the trace session before tracing is started
- Start tracing
- Stop tracing

lttd: The userspace daemon *lttd* is responsible to interact with the DebugFS an extract the channels data to disk or network. The *lttd* does not have in interaction with the trace session directly.

3.4.5 LTTng Userspace Tracer

LTTng provides a highly efficient kernel tracer but lacks a userspace tracer with similar performance. The LTTng userspace tracer (UST) is basically ported from the LTTng static kernel tracer to userspace, and is a work in progress.

3.4.5.1 UST Architecture

The userspace tracer has the following design level goals that should reflect in its architecture [FDD09]:

- UST is completely independent of the kernel tracer during trace time. The UST trace and the kernel trace can be correlated during the analysis time.
- UST is completely reentrant
- UST supports tracing of event handlers and multithreaded applications in userspace.
- To achieve better performance and low impact UST does not have system calls.
- UST employs *zero-copy*, therefore data is never copied.
- UST is able to trace code in executables as well as shared libraries
- The instrumentation point whether it is a marker or a tracepoint, should support unlimited number of arguments.
- UST does not require any compiler or linker support to generate trace data.
- UST produces a compact and coherent trace format for analysis.

Figure 4.9 shows the architecture of UST:

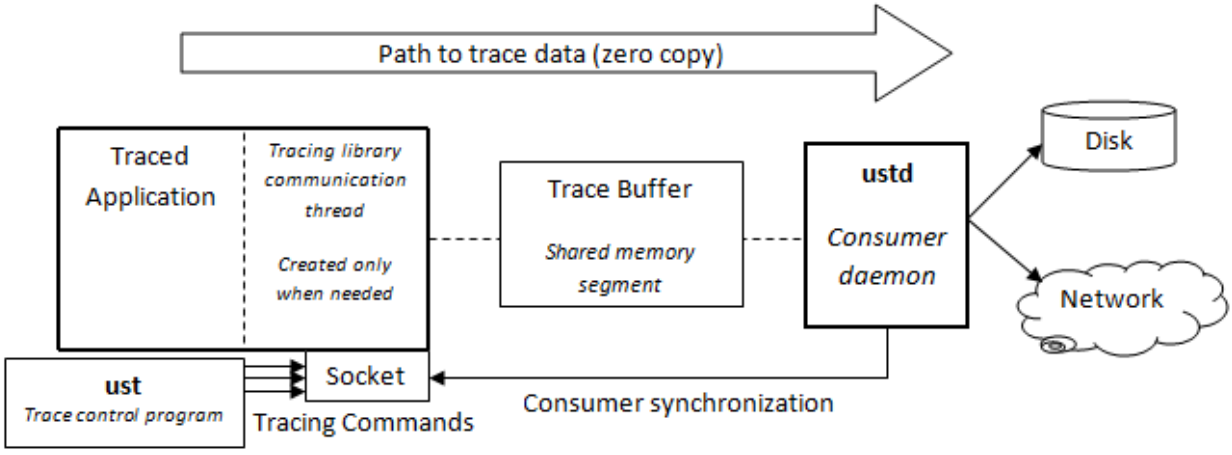


Figure 4.9: UST Architecture [FDD09]

3.4.5.2 UST Libraries

The programs must be linked with two libraries in order to get traced. The libraries are [FDD09]:

- Userspace tracing library – *libust*
- Userspace RCU library – *liburcu*

3.4.5.3 Time Synchronization

The LTTng userspace tracer does on have any dependency on the LTTng kernel tracer or vice versa [FDD09]. However, in order to do a combined analysis of the userspace and kernel traces it is necessary that the event timestamps of both the traces should be from the same time-source. The UST currently runs only on x86_32, x86_64 and ppc32 architectures [FDD09].

3.4.5.4 UST Data Collection

A userspace process called *ustd* collects data for all the processes that are being traced. The *ustd* opens a socket named *ustd* in the same directory as the traced application and wait for the command to collect the traced data from a certain buffer for a PID. On command ustd creates a consumer thread that eventually writes the trace data into the trace file [FDD09].

3.4.6 LTT Viewer (LTTV)

The LTT Viewer is a common GUI based trace analysis tool for kernel tracer as well as userspace tracer. LTTV is a trace viewer and is independent of LTTng tracer. It can open and filter traces based on different plugins.

As LTTV is easily extensible, developers can extend the functionality of LTTV by developing plugins. To get better performance results LTTV is written in C and uses glib and GTK graphical library.

3.4.7 Use of LTTng

LTTng has been used by some organizations for debugging and performance monitoring. *IBM* used LTTng successfully to solve some of their distributed filesystem related issue. *Autodesk* used LTTng to solve some real time issues in their application development. *Siemens* used LTTng to do some internal debugging and performance monitoring. The Linux distributions *Montavista*, *Wind River*, *STLinux* and *Suse* have included LTTng in their package [LTT10].

3.4.8 How to use LTTng?

LTTng kernel tracer has good instrumentation coverage, which are basically code changes to insert LTTng probes into a kernel. Therefore, the LTTng instrumentation set is distributed as a kernel patch series. LTTng has the flexibility to build the kernel instrumentation inside the kernel as well as build them as modules. Another package called *ltt-control* contains the *lttctl* and the *lttd* userspace applications needed for tracing.

LTTng userspace tracer comes as a package called *ust* that installs userspace tracer in a system. The ust package depends on the *userspace-rcu* library, which has to be installed before the *ust* package.

LTTV acts as a trace viewer and analyzer for both LTTng kernel and userspace trace files. LTTV comes as a separate *lttv* package and does not have any dependency on LTTng kernel or userspace tracer. But the lttv depends on the trace format the LTTng tracer produces. Therefore, the *lttv* package should be compatible with the LTTng kernel and userspace tracer trace format to be able to view trace files.

3.5 Control and Data Flow Analysis

3.5.1 Control Flow Analysis

In the context of this thesis report we define control flow as the order or pattern in which the application program calls or executes various other applications or snippets of code (like functions and procedures). Control Flow analysis helps to optimize the work flow execution of application programs and to determine functions and system calls which takes more amount of time [LPGF08]. If the execution of any program is getting more delayed than usual, the control flow analysis can easily help to find out the reason for the delay. For the whole system and the running application programs the control flow analysis can be done by a system profiler who takes time based samples of all the applications depending upon its footprint on the system and displays it at the end of sampling. From such sampling we can generate call graph which diagrammatically represents the functions and the system calls that the application made during its time of execution.

An example of a call graph which gets generated during the program execution is shown in Figure 4.10.

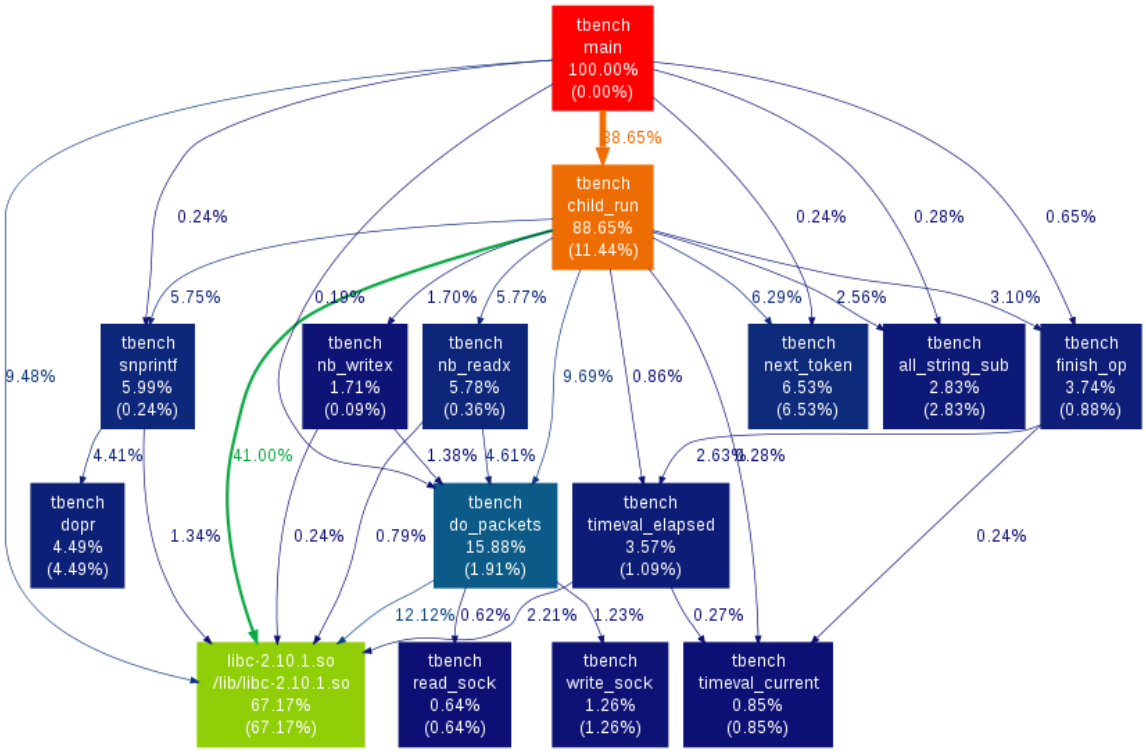


Figure 4.10: Example of Call Graph

The call graph in Figure 4.10 shows how the application program tbench makes calls to functions and programs during its sample run. We can also see how much time is spent in each of the functions and

system calls. We can either optimize the program checking in which function it spends more time and refractor it or otherwise can trace the delay of execution of the particular application program.

3.5.2 Data Flow Analysis

Data Flow analysis also helps to optimize the program to run better and faster than usual with effective use of system and memory. By annotating the source code of the application program we can get the lines of code which are using more number of CPU cycles and thus can be modified. One of the biggest performance measurement criteria in a multicore system is effective usage of memory and the CPU resources [ACU09]. The reason behind a line of code taking more CPU cycles can be improper memory management, irresponsible cache usage or improper data structures being used. Thus with certain tools we can gauge the usage of cache, rectify the temporal and spatial locality problem, hide the latencies involved in memory access and thus adding more Prefetch instructions. Increasing the cache line utilization of an application program decreases its execution timings and thus optimizes it further. We know that different ways of data representation and data access pattern can affect the performance of an application program. Thus Data and Control Flow analysis detects issues in application source code and also helps in optimizing the source program.

An example of annotated source code of an application program is shown in Figure 4.11.

```
               :      while (1) {
               :          kill(child, SIGSTOP);
     1 2.8e-05 :          usleep((100 - load_percent) * 1000);
     4 1.1e-04 :          kill(child, SIGCONT);
     7 2.0e-04 :          usleep(load_percent * 1000);
    10 2.8e-04 :          end = (long) time(NULL);
               :          sec = (end - start);
               :          //printf("%d\n",sec);
     1 2.8e-05 :          if(sec >= duration)
               :              break;
               :          counter++;
               :      }
               :      /* never here at this moment */
               :      printf("sending SIGTERM to the child\n");
               :      kill(child, SIGTERM);
               :      printf("loops executed = %d\n", counter);
               :
               :    } else if (child == 0) {
               :      /*
               :       * child process
               :       */
               :      while (1) {
    66  0.0019 :          if (syscall) {
               :              getpid();
               :          }
3560000 99.9958 :      }
```

Figure 4.11: Example of Annotated Source

The annotated source code provides the details of how many samples were taken for the particular line of code and its percentage of the total samples for that line. If the sample count for a particular line/block of code is more then it shows that it spends more CPU cycles than other lines of code.

4. Related Work

This chapter starts off with an overview of our goal. Then it explains our search methodology. Finally, it concentrates on citation of the previous work done that was useful to us to proceed in the correct direction and helped us making the correct decisions throughout our thesis work. It also describes similar work done in past.

List of technical terms

LTTng	Linux Trace Toolkit Next Generation
I/O	Input / Output
CPU	Central Processing Unit
RCU	Read Copy Update
ASCI	Accelerated Strategic Computing Initiative
RAM	Random Access Memory
SATA	Serial Advanced Technology Attachment
RPM	Revolutions Per Minute
GB	Gigabyte

4.1 Overview

LTTng comes with a set of efficient tracing tools for Linux which helps in solving performance and debugging issues involving multi threads and processes. LTTng aims to provide low disturbance and architecture neutral tracing tools which helps to track the pains in the system without involving much overhead.

Our goal mainly focuses on effectively gauging the fingerprint of LTTng as a tracing tool in a multicore Environment. There have been quite similar researches undergone before in getting either the effectiveness of tracing or the efficiency of LTTng in different architectures and environments which are described and explained in this chapter.

4.2 Search Methodology

LTTng was developed by Mathieu Desnoyers and was presented in his PhD thesis [DES09], so his thesis was the base for searching all the initial papers relating to LTTng and various other tracers which are there at present in the Linux Systems. The [LTT10] carries lots of invaluable papers from various conferences and journals which are somehow related to LTTng. As the thesis goal involved us to determine the performance measures for LTTng in a multicore environment, so the "multicore system" keyword search in Databases like IEEE and ACM gave us a lot of results and references to scientific journals. We got the information for the QorIQ™ P4080 Multicore processor board from the manufacturer's website which contained documents explaining the whole structure and features of P4080. Our next objective was to do Control Flow and Data Flow Analysis of the System and LTTng respectively and thus we went through the details of what those terms actually meant and what are the details that can be found out from that. Keyword "Control Flow Analysis" and "Data Flow Analysis" when searched in Google Scholar ™ lead to multiple papers leading to either IEEE or ACM or different university Lecture sessions. The Cited papers in those papers also contained the tools for doing Control and Data Flow Analysis and from there we got profiling tools like OProfile, TAU, Gprof, Valgrind and many others in which we chose OProfile for Control Flow based upon the results and its effectiveness of working in different architectures. The other part of our thesis goal focuses in Streaming of the LTTng Trace to Eclipse over the TCF Framework. We got research material regarding TCF Framework from Google Scholar ™ which pointed us to Eclipse website and LTT Tools plugin page. All the scientific research papers and books referred to here are from Google Scholar ™, IEEE and ACM Databases.

4.3 State of the Art

LTTng was developed by Mathieu Desnoyers as his PhD project and his PhD dissertation [DES09] shows how he tested LTTng performance for different load conditions on different type of architectures and compared it with the existing tracer tools. He took the load simulator tools like *dbench* (Disk load) and *tbench* (Network Load) to get the scalability of the tracer in multicore environments and benchmarking tool *lmbench* to measure the tracing effect on important system components like system calls and traps. Running 8 *tbench* clients with warm up of 120 seconds and execution time of 600 seconds revealed that tracing had very low impact on the overall performance with the network load on a 100Mb/s network card. During the test for scalability it was noticed that *tbench* linearly increases its workload in absence of tracing and LTTng tracing overhead was linearly maintaining same line with increase in processors thus showing that the overhead being totally independent on the number of processors. The *dbench* tests showed that disk throughput gets affected in heavy I/O workloads in tracer non-overwrite mode. In non-overwrite mode the tracer suffers from a lot of event loss than normal. *lmbench* tests showed that how the performance of the system get affected by a tracer running in the background. Results from lmbench proved that the instrumented code portions and paths suffered from more overhead than normal. All the existing tracers are compared with the performance results of LTTng and it shows it has quite low overhead and affect on the system performance than the other tracing tools.

Before that in 2006 performance of LTTng was determined by Mathieu Desnoyers and Michel Dagenais with micro and macro benchmarks [DD06]. The test was conducted on a 3 GHz Intel Pentium 4 without hyper threading and CPU clock calibrated to 3,000.607 MHz. For micro benchmarks kernel probe tests are done, without enabling interrupts. Results suggested that LTTng probe points do not increase the latency measure as they work without disabling the interrupts. The LTTng scheduler time gets affected due to the instrumentation as it needs the disabling of preemption on RCU list which is used for control. With macro benchmarks the time spent in the lttd and in the probe site was measured on application of variable loads on the system. Under kernel tracing it was found out that during high and medium load scenarios CPU time utilized by the tracing varies from 1.54 % to 2.28 % [DD06]. In user space tracing gcc application was instrumented and it showed an execution time variation. It execution time was 1.73 % more than the normal runtime. 2 % of the CPU time is taken by LTTng in case of a high workload to the system.

During December 2006 there was a study conducted by Kathryn Mohror and Karen L. Karavanic to find out the tracing overhead on the High Performance Linux clusters. The experiment setup was designed with three contexts; firstly execution times of execution of applications that contained instrumentation and wrote file to the disk. Secondly, executions having trace instrumentations but the trace file was not written to the disk and the Third condition involved no instrumentation and normal execution of a program [MLK06]. The tracing overhead was also measured due to scaling the number of processors. Execution time was compared by ASCI Purple Benchmark SMG2000. TAU was the main tracing tool and PerfTrack was the software used for collecting results. In the results of the experiment it

was found that the overhead of writing the trace data to disk was nearly 27 % of the normal execution time. Also the execution time of the application depended on the trace buffer size, as if it was larger the memory used by the buffer and the amount of time required to flush that off, largely varied. When the number of processors were increased it was seen that the overhead due to the was quite interrelated with the number of events generated in the whole trace session, though the overhead of writing trace buffer to the disk didn't had much relation with the increase in number of events.

During 2008 there was another study conducted by Parisa Heidari, Mathieu Desnoyers and Michel Dagenais to measure the overhead caused due to tracing and virtualization in a system [HDD08]. The experiment was setup considering 3 scenarios in which the one related to tracing was the impact caused by LTTng observed on a Domain 0 (Linux running over Xen), Domain U (one or more virtual system) and a normal system in 4 different scenarios – LTTng not compiled in the kernel, compiled in kernel but disabled the markers, flight recorder mode active, fully active with trace data being written to the disk. The tests were grouped into two parts one consisting of original application creating system stress (compiling, archiving, compression) and other part was standard benchmarks which simulates the load (*dbench*). The whole experiment was carried out in a machine having Intel Pentium 4, 3GHz hyper threaded processor, 2 GB RAM and a single 320 GB 7200 RPM SATA. The results showed that the cost associated with tracing is less than 3 % which when compared with the correctness, compactness and completeness of the information collected was a very small amount of disturbance. In the scenario when LTTng was compiled in but the probe was disabled, it caused a very less impact. LTTng without loading the probes fast completes the test but the difference is smaller and lesser than standard deviation. There is a effect of less than 2 % in the performance scale when probes are not loaded and less than 5 % deviation in performance when the trace is written by LTTng [HDD08]. There was no impact on scheduling or real time response as LTTng uses atomic operations.

During late 2009 Pierre-Marc Fournier, Mathieu Desnoyers and Michel Dagenais gauged the performance of UST and also compared it with the performance of DTrace on an equivalent tracing task [FDD09]. The tests were conducted on a cache hot dual quad-core Xeon 2 GHz with a RAM of 8GB. DTrace was run under Solaris environment. The command,

$$find\ /usr\ -regex\ '.*a'. \qquad \text{[FDD09]}$$

was run 60 times. This regular expression was chosen to prove malloc / free activity and they were also instrumented. The UST performance was measured with and without instrumentation compiled in. The difference in the two measures came to much less significant value. When the probes were connected but there was no tracing then there was a slight increase in the execution timing. With Tracing on the cost/event was found to be approximately 698 ns [FDD09]. The LTTng and UST together have a cost per events 7 times lower than that of DTrace.

5. Experiment Setup

This chapter describes in detail the use of technologies in our experiments. These technologies include the hardware and software configurations and tools, utilities and scripts used to perform the experiments.

List of technical terms

LTTng	Linux Trace Toolkit Next Generation
AMD	Advanced Micro Devices, Inc.
DDR	Double Data Rate
SDRAM	Synchronous Dynamic Random Access Memory
I/O	Input / Output
CPU	Central Processing Unit
SMP	Symmetric Multiprocessing
L2 cache	Level 2 cache

5.1 System Configuration

The first step is to configure the system on which we should run our experiments. System configuration consists of two parts:

- Hardware Configuration
- Software Configuration

5.1.1 Hardware Configuration

The hardware on which we performed our experiments is an x86 based Intel® Core™ 2 Quad processor desktop. The specifications of the system are:

- Intel® Core™ 2 Quad with four 64 bit Q9550 SMP cores operating at frequency 2.83 GHz
- 3 GB of DDR2 SDRAM operating at frequency 667 MHz
- 100 Mbps Ethernet

5.1.2 Software Configuration

The based Intel® Core™ 2 Quad Desktop has been running *openSUSE 11.2* Desktop Linux operating system with dual kernels. One of the kernels are kernel version 2.6.33.2 patched with LTTng 0.211 instrumentation set built as loadable modules. Therefore, unless the LTTng modules are loaded using *modprobe* the LTTng instrumentation set will remain dormant. The other kernel is kernel version 2.6.31.5 without LTTng instrumentation in it.

Apart from the LTTng kernel patch series, the application package *ltt-control* version 0.84 has been installed in the system to control the tracing activity. Userspace tracer has also been installed in the system by installing the packages *userspace-rcu* and *ust* version 0.4.3

LTT Viewer has been installed in the system to view the trace files, by installing the *lttv* package version 0.12.31, according to the compatible trace format.

5.2 Tools and Utilities

5.2.1 Load Generation Tools

The tools described under this section generate various types of load on the system and in varying amount. The purpose of using these tools in our experiment is to apply varying amount of load in the test system.

5.2.1.1 load

The load is a command line program written in C, which can generate specified amount of load on a single CPU core for a specific period of time.

Usage:

```
romik@linux-2t0w:~> gcc –g –o load load.c
romik@linux-2t0w:~> ./load –l 50 –t 180
```

load is executed with 20% CPU load for 180 seconds.

Output:

```
generating CPU load : 20 %
running for 180 seconds
sending SIGTERM to the child
loops executed = 1789
```

5.2.1.2 tbench

The tbench is a command line utility that can generate network and process load by simulating similar socket calls as done by the Samba daemon during a *Net Bench* run in real environment. The tbench utility has two components:

- **tbench_srv:** The *tbench_srv* is the server utility that listens to *tbench* client connections
- **tbench:** The *tbench* utility which has the capability of spawning multiple clients to connect to the *tbench_srv*

Usage:

```
romik@linux-2t0w:~> tbench_srv
waiting for connections
```

tbench_srv is executed and waiting for tbench clients to run.

```
romik@linux-2t0w:~> tbench -t 5 5 localhost
dbench version 4.00 - Copyright Andrew Tridgell 1999-2004

Running for 5 seconds with load '/usr/local/share/client.txt' and minimum warmup 1 secs
0 of 5 processes prepared for launch   0 sec
5 of 5 processes prepared for launch   0 sec
releasing clients
   5     14603   226.10 MB/sec  execute   1 sec  latency 11.225 ms
   5     23155   227.33 MB/sec  execute   2 sec  latency 7.511 ms
   5     31630   227.18 MB/sec  execute   3 sec  latency 11.902 ms
   5     40197   228.05 MB/sec  execute   4 sec  latency 8.037 ms
   5  cleanup   5 sec
   0  cleanup   5 sec
```

tbench is executed for a time span of 5 seconds with 5 clients for tbench_srv running in *localhost*.

Output:

```
Operation      Count    AvgLat    MaxLat
 ----------------------------------------
 NTCreateX      36535     0.097     8.020
 Close          26932     0.084    11.663
 Rename          1546     0.107     2.934
 Unlink          7291     0.096     8.132
 Qpathinfo      33225     0.096     7.894
 Qfileinfo       5823     0.085     2.981
 Qfsinfo         6042     0.085     2.905
 Sfileinfo       2958     0.086     2.988
 Find           12839     0.097    10.371
 WriteX         18078     0.236    11.887
 ReadX          57510     0.108    11.761
 LockX            120     0.085     0.206
 UnlockX          120     0.089     0.209
 Flush           2540     0.085     0.344

Throughput 228.048 MB/sec  5 clients  5 procs  max_latency=11.902 ms
```

5.2.2 System Activity Measurement Tools

The tools described under this section are used to record the activity of the system related to CPU, memory, I/O and other parameters. The purpose of using these tools in our experiment is to record and analyze the activity and performance of the system.

5.2.2.1 Sysstat

Sysstat is a set of utilities that can monitor and record system activities that can be used to measure system performance. The tools present in the Sysstat package are *sar*, *sadf*, *iostat*, *mpstat*, *pidstat*, *sa1* and *sa2*.

The utilities we used to capture various system activities are:

- **sar:** The *sar* utility collects and saves the system activity information. It includes information about CPU, memory, I/O, interrupts, disk and other parameters.

Usage:

```
linux-2t0w:~ # sar -u -o sysdata 2 3
```

sar is executed to run 3 times at an interval of 2 seconds, output statistics to file *sysdata* and display CPU activity information.

Output:

```
Linux 2.6.33.2-0.1-desktop (linux-2t0w)         04/21/10        _i686_  (2 CPU)

20:46:29        CPU     %user     %nice   %system   %iowait    %steal     %idle
20:46:31        all      4.27      0.00      1.07      0.00      0.00     94.67
20:46:33        all      5.91      0.00      2.06      0.77      0.00     91.26
20:46:35        all      6.22      0.00      2.24      0.50      0.00     91.04
Average:        all      5.49      0.00      1.80      0.43      0.00     92.28
```

- **sadf:** The *sadf* utility is used to export the data collected by *sar* in multiple human readable formats such as CSV, XML etc.

Usage:

```
linux-2t0w:~ # sadf -d sysdata -- -u
```

sadf is executed to read information from file *sysdata* (created by *sar*) and display CPU activity information.

Output:

```
# hostname;interval;timestamp;CPU;%user;%nice;%system;%iowait;%steal;%idle
linux-2t0w;2;2010-04-21 18:46:31 UTC;-1;4.27;0.00;1.07;0.00;0.00;94.67
linux-2t0w;2;2010-04-21 18:46:33 UTC;-1;5.91;0.00;2.06;0.77;0.00;91.26
linux-2t0w;2;2010-04-21 18:46:35 UTC;-1;6.22;0.00;2.24;0.50;0.00;91.04
```

5.2.3 Control Flow and Data Flow Analysis Tools

The tools described under this section are useful for control flow analysis of a system or an application. These tools are basically profilers and utilities to generate call graphs. The purpose of using these tools in our experiment is to generate profiling data, annotations and call graphs that will help us in control flow analysis of the system or a binary compiled with debug information.

5.2.3.1 OProfile

OProfile is the most commonly used system-wide profiler for Linux based systems. It is capable of profiling all running code in the system with very little overhead. The *OProfile* package consists of a kernel driver, a daemon and several profile analysis tools. OProfile supports collection of data from various hardware performance counters. Therefore, applications, shared libraries, kernel modules, kernel as well as software and hardware interrupt handlers can be profiled using *OProfile*. OProfile supports a wide range of architecture from 32 and 64 bit x86 to PowerPC, MIPS, ARM etc. In a work, *OProfile* is a useful utility to determine performance bottlenecks within a system. *opcontrol* is used to control the profiler and *opreport* is used to extract the profiled data.

Usage:

```
linux-2t0w:~ # opcontrol --reset
linux-2t0w:~ # opcontrol --vmlinux=/usr/src/linux-2.6.33.2/vmlinux --separate=lib --
callgraph=32
linux-2t0w:~ # opcontrol --start
Using default event: CPU_CLK_UNHALTED:100000:0:1:1
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.
```

Start *OProfile*.

```
linux-2t0w:~ # opcontrol --dump
linux-2t0w:~ # opcontrol --shutdown
Stopping profiling.
Killing daemon.
```

Dump profile data or stop *OProfile*.

Output:

```
CPU: AMD64 processors, speed 800 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Cycles outside of halt state) with a unit mask of 0x00 (No
unit mask) count 100000
CPU_CLK_UNHALT...|
  samples|      %|
------------------
    17675 38.9463 vmlinux
     8649 19.0578 kdeinit4
        CPU_CLK_UNHALT...|
          samples|      %|
        ------------------
             4050 46.8262 libQtGui.so.4.5.3
             1220 14.1057 libQtCore.so.4.5.3
             1155 13.3541 libc-2.10.1.so
              503  5.8157 libkonsoleprivate.so
              392  4.5323 libX11.so.6.2.0
              196  2.2662 libglib-2.0.so.0.2200.1
              195  2.2546 libpthread-2.10.1.so
              160  1.8499 libplasma.so.3.0.0
              133  1.5378 libkdeui.so.5.3.0
              113  1.3065 libxcb.so.1.1.0
```

opreport output (clipped).

In our experiments for Control Flow analysis we have used *OProfile* with the performance counter event CPU_CLK_UNHALT and with libraries separated. The CPU_CLK_UNHALT event gives the number of CPU clock cycles outside the halt state of CPU which implies the amount of time spent by a binary image while execution. We have also generated call graphs of binary images using *OProfile* [PZWSS07].

In our experiments for Data Flow Analysis we have used *OProfile* with two performance counter events LII_MISSES and INST_RETIRED_ANY_P and with the libraries separated [PZWSS07]. Caches are high speed memories placed closest to the CPU. It takes less number of CPU cycles to fetch data stored in cache memory rather than the main memory. Therefore, performance will increase if the cache misses decreases. The LII_MISSES event gives the number of L2 cache misses for a particular binary image while execution. Branch prediction is an important technique to achieve parallelism in multicore systems. Branch prediction is a technique to predict and process instructions for a particular branch even before the decision is made. In case of a branch misprediction the processed instructions have to be retired. The event INST_RETIRED_ANY_P helps us determine the number of times branch mispredictions have happened for a particular binary image while execution [PRA03].

5.2.3.2 Valgrind

Valgrind is a tool suite consisting of debugging and profiling tools. It consists of utility Memcheck (Memory leak Checker), Cachegrind (Cache Profiler), Callgrind (Cachegrind with Callgraphs), Massif (Heap Profiler) and Helgrind (Thread debugger) [VAL10]. In our experiments we use Memcheck, which detects the memory errors in programs during runtime. Memcheck mainly has 4 different kinds of Memory checking [SN05]:

- It tracks addressability of each byte of memory getting updated with the information of whether the memory is free or allocated.
- It keeps a note of all heaps which gets allocated with malloc () and new, so that it can detect leaking of memory at program termination time.
- It checks that strcpy () and memcpy () doesn't have same memory blocks as arguments.
- It performs *definedness checking* which ensures the definedness of every data bit in memory and registers.

Usage:

```
linux-2t0w:~ # valgrind --tool=memcheck --leak-check=full --trace-children=yes --show-
reachable=yes -v lttctl -C -w /tmp/trace trace
```

Output:

```
==3531== Memcheck, a memory error detector
==3531== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==3531== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==3531== Command: lttctl -C -w /tmp/trace -o channel.all.overwrite=1 trace
==3531==
--3531-- Valgrind options:
--3531--    --tool=memcheck
--3531--    --leak-check=full
--3531--    --trace-children=yes
--3531--    --show-reachable=yes
--3531--    -v
--3531-- Contents of /proc/version:
--3531--   Linux version 2.6.33.2-Lttng0.203 (root@linux-ambr) (gcc version 4.4.1 [gcc-4_4-
branch revision 150839] (SUSE Linux) ) #3 SMP Wed May 5 17:15:59 CEST 2010
--3531-- REDIR: 0x40cad00 (rindex) redirected to 0x4027840 (rindex)
--3531-- REDIR: 0x40ca260 (index) redirected to 0x40278d0 (index)
--3531-- REDIR: 0x40ca970 (strlen) redirected to 0x4027c00 (strlen)
--3531-- REDIR: 0x40cca20 (memcpy) redirected to 0x4028080 (memcpy)
--3531-- REDIR: 0x40ca440 (strcpy) redirected to 0x4027c60 (strcpy)
--3531-- REDIR: 0x40c7050 (malloc) redirected to 0x4026c07 (malloc)
--3531-- REDIR: 0x40ca3d0 (strcmp) redirected to 0x4027f20 (strcmp)
--3531-- REDIR: 0x40ca0b0 (strcat) redirected to 0x40279c0 (strcat)
--3531-- REDIR: 0x40cc550 (mempcpy) redirected to 0x4028d10 (mempcpy)
--3531-- REDIR: 0x40cf510 (strchrnul) redirected to 0x4028cc0 (strchrnul)
--3531-- REDIR: 0x40c6f70 (free) redirected to 0x4026821 (free)
==3531== 152 bytes in 17 blocks are definitely lost in loss record 1 of 5
==3531==    at 0x4026C8C: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==3531==    by 0x40CA6C0: strdup (in /lib/libc-2.10.1.so)
==3531==    by 0x4051297: lttctl_set_channel_enable (liblttctl.c:472)
==3531==    by 0x8049E1F: main (lttctl.c:631)
==3531==
==3531== 152 bytes in 17 blocks are definitely lost in loss record 2 of 5
==3531==    at 0x4026C8C: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==3531==    by 0x40CA6C0: strdup (in /lib/libc-2.10.1.so)
==3531==    by 0x4051117: lttctl_set_channel_overwrite (liblttctl.c:536)
==3531==    by 0x8049E49: main (lttctl.c:637)
==3531==
==3531== 284 bytes in 1 blocks are still reachable in loss record 3 of 5
==3531==    at 0x4026C8C: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==3531==    by 0x8049346: parst_opt (lttctl.c:238)
==3531==    by 0x80496DA: main (lttctl.c:425)
==3533==
==3533== HEAP SUMMARY:
==3533==     in use at exit: 0 bytes in 0 blocks
==3533==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==3533==
==3533== All heap blocks were freed -- no leaks are possible
==3533==
==3533== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

In our experiments of Data Flow, Memcheck is used here to detect memory leaks in LTTng Kernel and Userspace tracer. We are using the option "`--leak-check=full`" to get a full report of any types of memory leaks from the program. The argument "`--trace-children=yes`" is used to track any forked program from the main program, so that the Memcheck utility can even show memory leaks of forked child programs. The other two arguments "`--show-reachable=yes`" and "`-v`" are used to get more detailed report about memory mismanagement of the application program.

5.2.3.3 gprof2dot.py

The *gprof2dot.py* is a Python script that can convert the call graph output from different profilers into a dot graph. It supports different profilers such as gprof, OProfile, callgrind, sysprof and others. The script has dependencies on *Python* and *Graphviz* to run.

Usage:

```
linux-2t0w:~ #  opreport image:/usr/local/bin/dbench -cgf | gprof2dot.py -f oprofile | dot -
Tpng -o output.png
```

Output:

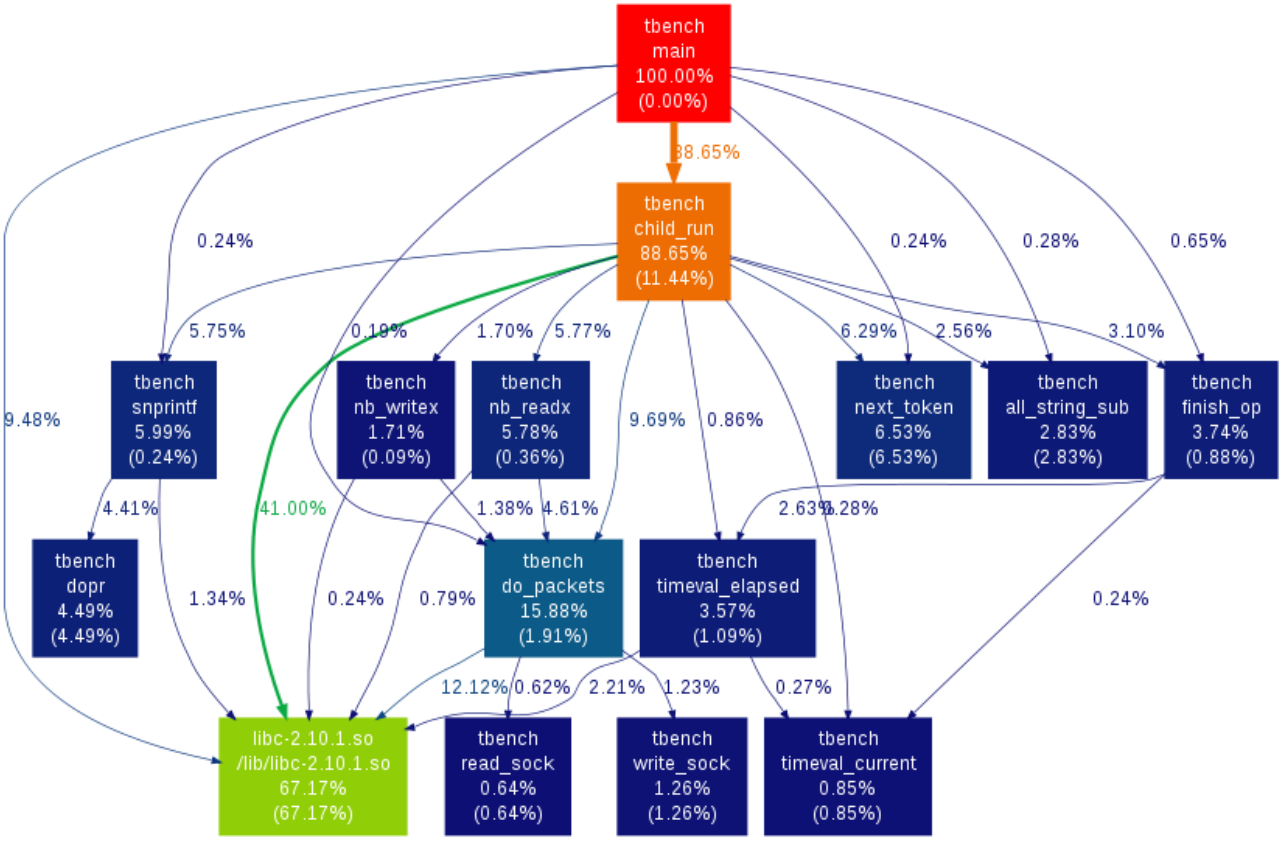


Figure 5.1: tbench call graph output

We have used gprof2dot.py to convert *OProfile* generated call graphs into dot graphs for better understanding and comparison.

5.3 Test System Setup

5.3.1 Tools Setup

The different tools mentioned in the previous section are used to set up the test system. Figure 5.2 describes how the different tools are used in combination in Test System Setup.

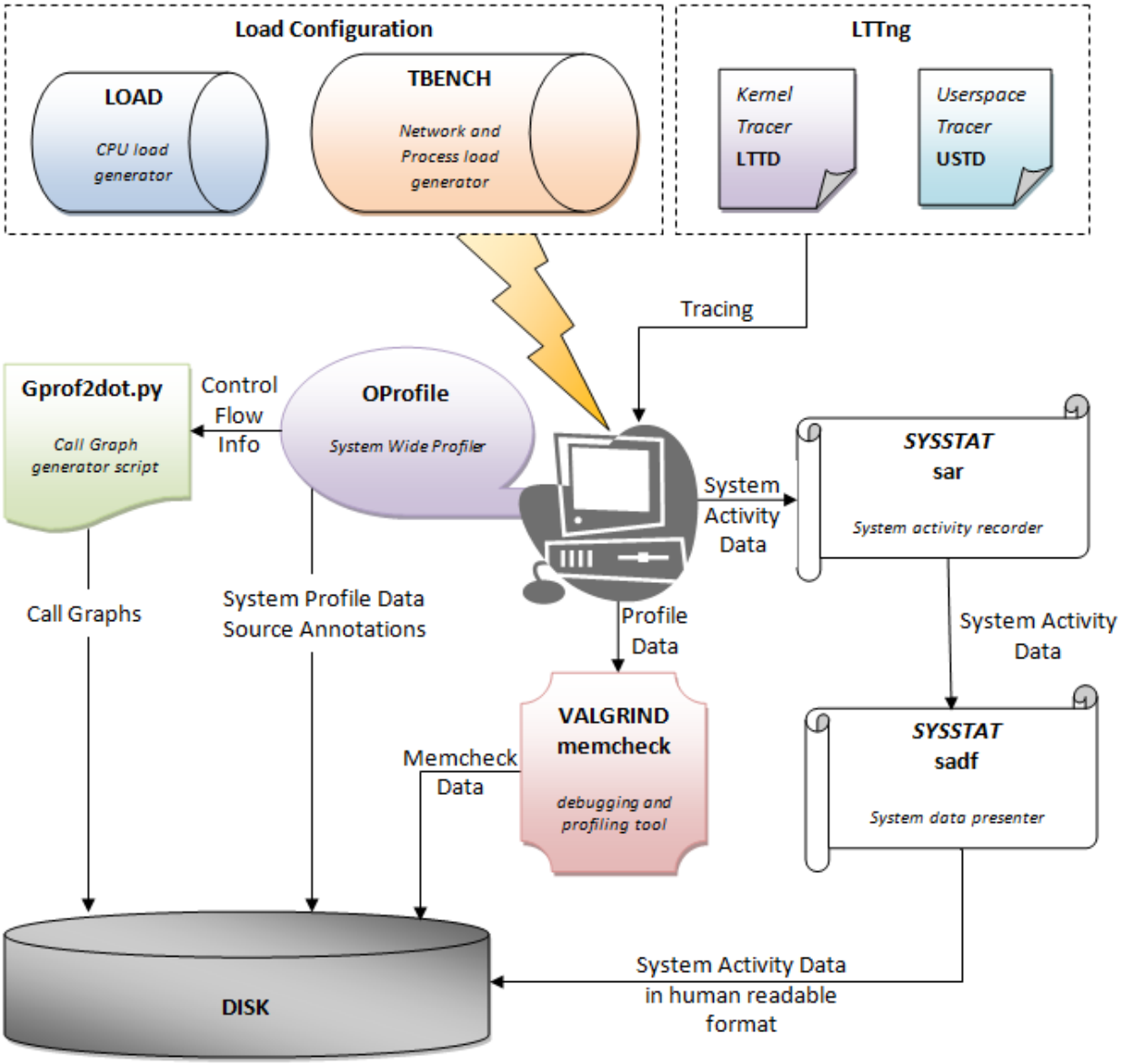


Figure 5.2: Test System Setup

The system runs *LTTng* tracer which comprises of the *kernel tracer* and *the userspace tracer*. The tools *load* and *tbench* are used to generate load on the system in different configurations while the

tracing in on or off. The system activity is recorded by *sar* utility of the *SYSSTAT* tools bundle, and the *sadf* utility from the same bundle converts the *sar* generated data into human readable form and stores the data in the disk. The system is also profiled using a system wide profiler called *OProfile*. It stores the system profile data and the source annotations of binary files in the disk. These profile data and source annotations are used for *Control Flow Analysis* and *Data Flow Analysis* of the traced system. *OProfile* passes the control flow information to a call graph generation script named *Gprof2dot.py* that generates visual call graphs from the control flow information. It saves the call graphs to the disk.

5.3.2 Load Configuration

Two different types of load generators have been used to generate load on the test system.

The *load* program generates specified amount of CPU load on a single core. Therefore for a four–core processor, four instances of the load program have to be executed. The load program is a program written in C language and generates CPU load by forking child processes continuously. The source code of the *load* program has been provided in Appendix B.

The *tbench* utility produces process and network load on the test system. In our experiments tbench is run on the loopback interface adapter with a standard of 10 clients. The amount of load on the system is varied by varying the throughput data rate of tbench clients.

Table 5.1 describes the load configuration for the experiments to be performed:

Configuration	Load Generator	Load Level	Load %
CNF-01	Load	Low	20
CNF-02	Load	Medium	50
CNF-03	Load	High	90
CNF-04	Tbench	Low	30
CNF-05	Tbench	Medium	50
CNF-06	Tbench	High	80

Table 5.1: Load Configuration

5.3.3 Test Automation

The experiments are automated with the help of shell scripts. These shell scripts are responsible for running the tools and utilities in proper order and recording all the test data in the disk for analysis at a later period.

5.3.4 Performance Measurement

Various criteria have been measured in order to judge the performance of LTTng kernel trace and userspace tracer in a 4-core SMP system under various load configurations. Performance measurement has been done in system level, program level and function level. Following are the different criteria for performance measurement.

System Level Performance Measurement

- CPU usage by a user program
- CPU usage due to system activities
- CPU usage due to I/O waits

Program Level Performance Measurement

- Percentage of CPU cycles needed for an image execution
- Percentage of L2 cache misses
- Percentage of retired instructions
- Percentage of memory leaks

Function Level Performance Measurement

- Call Graph analysis
- Source annotation evaluation

5.3.5 Result Analysis

The results obtained from the experiments are stored in the disk in form of system activity data, OProfile data, VALGRIND memory report, call graphs and source annotations. These result data provide a valuable input for system level, program level and function level performance measurement.

The *system level performance measurement* includes analyzing and comparing percent CPU utilization for user programs, system and I/O waits. The *program level performance measurement* includes CPU cycles needed for execution, L2 cache misses, memory leaks while execution and the retired instructions while branch mispredictions for a binary image. The *function level performance measurement* includes call graph analysis and source annotation evaluation for percent CPU cycles at instruction level. The program level and function level performance measurement comprise the Control Flow and Data Flow Analysis.

The results analysis will help us to zero down on the efficiency of the LTTng kernel tracer and userspace trace on a multicore SMP system. Figure 5.3 presents the different phases of result analysis.

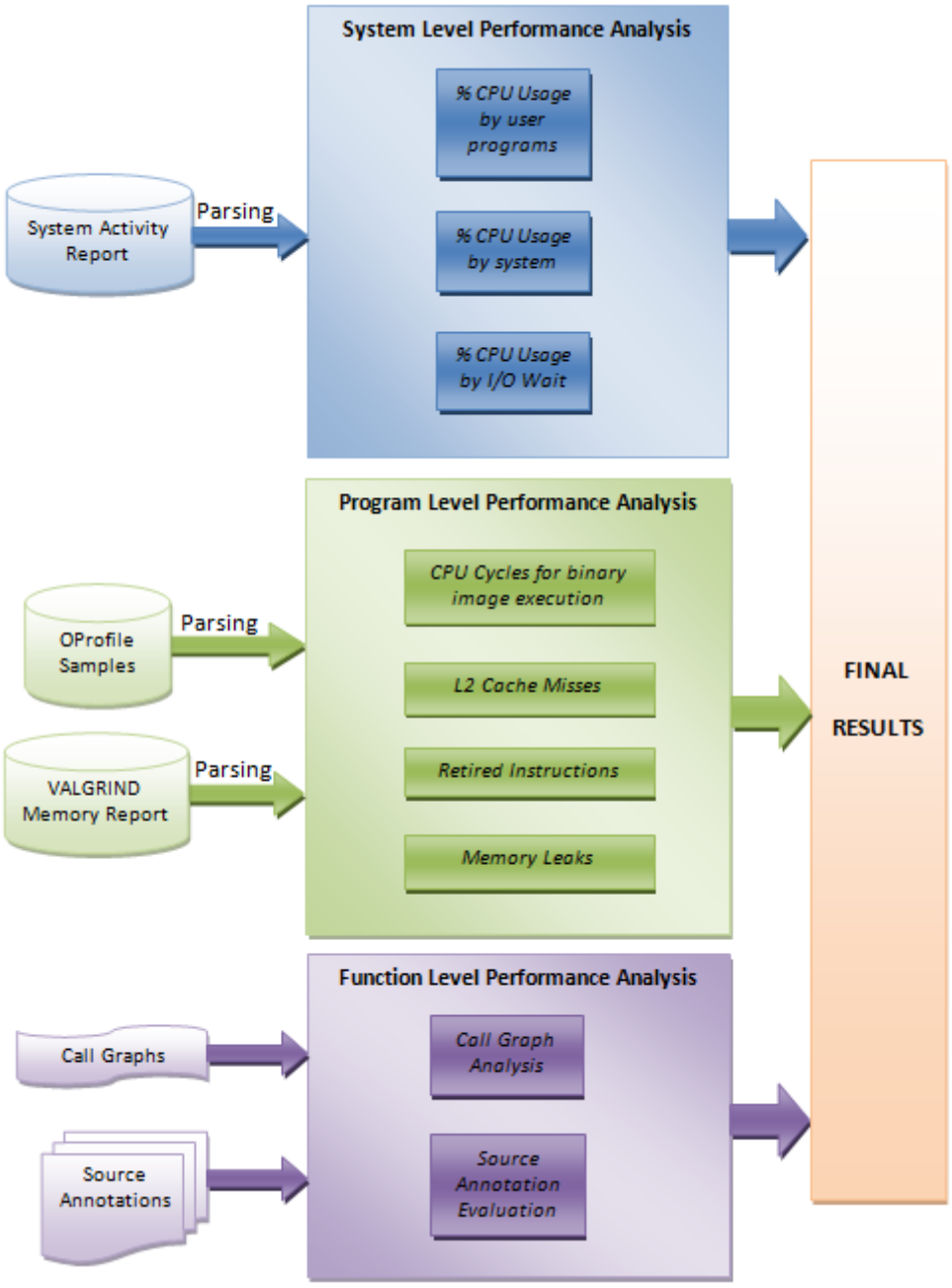


Figure 5.3: Result Analysis

6. Experiment Methodology

This chapter describes the experiment methods in detail that are to be performed on LTTng kernel tracer and userspace tracer.

List of technical terms

LTTng	Linux Trace Toolkit Next Generation
UST	Userspace Tracer
SUT	System Under Test
CFG	Control Flow Graph
CPU	Central Processing Unit

6.1 Load Configuration

6.1.1 Experiment 1 – Determination of load configuration parameters for System Under Test (SUT)

Objective – Setting up the parameters for load configurations as described in Table 6.1 in chapter Experiment Setup. These load configurations are to be used in forthcoming experiments.

Explanation – For the Control Flow and the Data Flow analysis the idea is to generate a substantial load in the whole system. For the determination of low, medium and high load of CPU, the idle time of the CPU will be collected by the *sar* utility of *SYSTAT* bundle. Table 6.1 shows the desired percentage of idle time for each low, medium and high load configurations of the CPU.

Configuration of LOAD	CPU Idle Time (%)
Low	67-100
Medium	33-66
High	0-32

Table 6.1: Load Configuration

The load program generates specified percentage of load on a single CPU core. As the system under test has four SMP cores, 4 instances of load program is executed simultaneously in vanilla kernel to generate the specified percentage of CPU load on the system. The total number of clients that will be required for *tbench* (network load) is kept fixed at 10 and by running the tbench (in loopback) in a fresh vanilla kernel with controlled throughput. Table 6.2 provides the details of six different load configurations to be determined.

Utility	Instances/Clients	Target CPU Idle Time (%)	Load Configuration
load	4	80	CNF-01
load	4	50	CNF-02
load	4	10	CNF-03
tbench	10	70	CNF-04
tbench	10	50	CNF-05
tbench	10	20	CNF-06

Table 6.2: Load Configurations to be determined

After the determination of the load configuration parameters, Control and Data Flow analysis are done on empty vanilla kernel and kernel compiled with LTTng with variation of different parameters thus differentiating between their usages of CPU cycles.

6.2 Control Flow Analysis

6.2.1 Experiment 2 – Measuring the efficiency of LTTng Kernel Tracer

Objective – Measuring the efficiency of LTTng kernel tracer for different load configurations. Detailed program level and function level performance analysis is to be performed on the gathered results.

Explanation – The *OProfile* tool is used to get the Control Flow parameters in the whole system with different load configurations. The *opcontrol* command is run with kernel image and separate libraries as the argument (to get the control flow of any process inside the libraries) to get the appropriate *opreport* depicting the CPU cycles spent by each of the functions and binaries. The CPU usage is also collected during the load generation with the help of *sar* tool. This experiment helps us to determine the CPU activity of the system having upon different load configurations along with a generalized sample report of individual CPU cycles used by separate functions of binaries under varied load. *OProfile* was run with the CPU_CLK_UNHALTED hardware performance counter to get the actual CPU time spend by the binaries and source annotations. *OProfile* output was fed into a python script to generate control flow graphs for *LTT Daemon*. Table 6.3 describes all the test cases to be executed. To get better results all test cases are run 3 times.

Test Case	Kernel	Markers	LTTng Armed	Tracing	Tracing Mode	Load Configuration
T1	Vanilla	Off	No	Off	NA	CNF-01, CNF-04
T2	Vanilla	Off	No	Off	NA	CNF-02, CNF-05
T3	Vanilla	Off	No	Off	NA	CNF-03, CNF-06
T4	Instrumented	Off	No	Off	NA	CNF-01, CNF-04
T5	Instrumented	Off	No	Off	NA	CNF-02, CNF-05
T6	Instrumented	Off	No	Off	NA	CNF-03, CNF-06
T7	Instrumented	On	No	Off	NA	CNF-01, CNF-04
T8	Instrumented	On	No	Off	NA	CNF-02, CNF-05
T9	Instrumented	On	No	Off	NA	CNF-03, CNF-06
T10	Instrumented	On	Yes	Off	NA	CNF-01, CNF-04
T11	Instrumented	On	Yes	Off	NA	CNF-02, CNF-05
T12	Instrumented	On	Yes	Off	NA	CNF-03, CNF-06
T13	Instrumented	On	Yes	On	Non Overwrite	CNF-01, CNF-04
T14	Instrumented	On	Yes	On	Non Overwrite	CNF-02, CNF-05
T15	Instrumented	On	Yes	On	Non Overwrite	CNF-03, CNF-06
T16	Instrumented	On	Yes	On	Flight Recorder	CNF-01, CNF-04
T17	Instrumented	On	Yes	On	Flight Recorder	CNF-02, CNF-05
T18	Instrumented	On	Yes	On	Flight Recorder	CNF-03, CNF-06

Table 6.3: Test Cases for Experiment 2

6.2.2 Experiment 3 – Measuring the efficiency of LTTng Userspace Tracer

Objective – Measuring the efficiency of LTTng userspace tracer for different load configurations. Detailed program level and function level performance analysis is to be performed on the gathered results. The effect of the number of instrumentations is also measured.

Explanation – The Profiling tools are having same configurations for all the experiments. For this experiment the UST is installed on vanilla kernel and load and tbench are freshly compiled with UST instrumentation inside. The profiler is started after running the load generating programs with the instrumentation ON/OFF. The profiler (OProfile) gathers sample generating *opreport* which shows the CPU cycles usage at program level, the annotations of the source files of UST and load generation programs with the time spend in the CPU and the call graph which helps into digging deep regarding the reason of the overhead (if any). Overall system overhead is also measured. The experiment helps in determining that either or not User Space Tracing Instrumentation has an overhead on the execution time of the binary and also on the system performance. Table 6.4 shows the test cases executed for this experiment. All test cases are repeated 3 times to get better results. To determine the effect of the number of instrumentations all the test cases are repeated with 1, 5 and 10 instrumentation(s) compiled in the source code of both load and tbench.

Test Case	Kernel	UST Markers	Tracing	Load Configuration
T1	Vanilla	Off	Off	CNF-01, CNF-04
T2	Vanilla	Off	Off	CNF-02, CNF-05
T3	Vanilla	Off	Off	CNF-03, CNF-06
T4	Vanilla	On	Off	CNF-01, CNF-04
T5	Vanilla	On	Off	CNF-02, CNF-05
T6	Vanilla	On	Off	CNF-03, CNF-06
T7	Vanilla	On	On	CNF-01, CNF-04
T8	Vanilla	On	On	CNF-02, CNF-05
T9	Vanilla	On	On	CNF-03, CNF-06

Table 6.4: Test Cases for Experiment 3

6.2.3 Experiment 4 – Measuring the impact on System as well as Traced Application when LTTng Kernel Tracer and Userspace Tracer are executed together

Objective – This experiment aims to find out the overhead in the system and the instrumented application in case LTTng kernel tracing and userspace Tracing is both running together and the application binaries are instrumented with UST markers.

Explanation – In an LTTng installed kernel we already have tested all variations to test the overhead of LTTng kernel tracing in Experiment 2. In this case we also install UST (Userspace Tracing) in the LTTng patched kernel and instrument the UST markers in the binaries. With all combination of LTTng tracing along with UST instrumentation we take a set of test cases for which we generate an annotated source code for LTT Daemon and UST Daemon. We get overall sample report showing the CPU cycles taken in all the possible conditions by LTTng binaries and UST binaries. There is also control flow graph of the binaries helping us to find the effect of LTTng kernel tracer on UST and vice versa, and also the effect of both on the system and the traced application. Table 6.5 provides the test cases to be executed for this experiment. All test cases are executed 3 times to get better results. As we are running userspace tracer we have repeated all test cases with 1, 5 and 10 UST markers compiled in with the userspace applications to find out the impact of increasing number of instrumentations as well.

Test Case	Kernel	Kernel Tracing	Tracing Mode	UST	Load Configuration
T1	Instrumented	On	Non Overwrite	On	CNF-01, CNF-04
T2	Instrumented	On	Non Overwrite	On	CNF-02, CNF-05
T3	Instrumented	On	Non Overwrite	On	CNF-03, CNF-06
T4	Instrumented	On	Flight Recorder	On	CNF-01, CNF-04
T5	Instrumented	On	Flight Recorder	On	CNF-02, CNF-05
T6	Instrumented	On	Flight Recorder	On	CNF-03, CNF-06

Table 6.5: Test Cases for Experiment 4

6.3 Data Flow Analysis

6.3.1 Experiment 5 – Running load program and tbench on LTTng Kernel with Non Overwrite and Flight Recorder tracing modes

Objective – The objective of this experiment is to find out any LII cache misses or branch misprediction of LTT Control module or the LTT daemon during tracing when there is a low, medium and high load generated on the system by a child forking program **load** and process and network load generated by the benchmarking utility **tbench.** This is an effort to make the tracer run through different type of loads and check the internal memory allocation issues for the LTT Control and Daemon modules.

Explanation – In LTTng kernel we use load program and tbench to generate low, medium and high load according to the load configuration matrix of Experiment 1. We start the LTTng kernel tracer in Non overwrite and Flight recorder tracing modes one after the other and use OProfile Hardware Events LII_MISSES and INST_RETIRED_ANY_P to sample LTT Control and LTT Daemon to find out respective Cache Misses and Branch Mispredictions in it. All the experiments are controlled by automated script which initially triggers the load program under different load configuration and then starts OProfile sampling and LTTng Kernel Tracer parallely. The trace gets destroyed after 180 sec when the load program ends. For tbench also the same process is followed where the trace gets destroyed after the completion of tbench. Table 6.6 provides all test cases for this experiment.

Test Case	Kernel	Tracing Mode	Load Configuration
T1	LTTng	Non Overwrite	CNF-01, CNF-04
T2	LTTng	Non Overwrite	CNF-02, CNF-05
T3	LTTng	Non Overwrite	CNF-03, CNF-06
T4	LTTng	Flight Recorder	CNF-01, CNF-04
T5	LTTng	Flight Recorder	CNF-02, CNF-05
T6	LTTng	Flight Recorder	CNF-03, CNF-06

Table 6.6: Test Cases for Experiment 5

6.3.2 Experiment 6 – Running UST tracing on load and tbench program each instrumented with 10 markers under different load configurations

Objective – The aim of this experiment is to find out any L2 cache misses or branch misprediction of UST daemon and UST Libraries during userspace tracing when there is a low, medium and high load generated on the process and network by the benchmarking utility **tbench** and on the system by the **load** program. The tests are done under various load circumstances to gauge the memory management efficiency of the UST Daemon and the Libraries during Userspace tracing.

Explanation – In plain vanilla kernel the program **load** and **tbench** are instrumented with 10 markers and are recompiled for current experiment use.UST Tracer triggers Load program and tbench are again triggered to generate the respective loads in the system according to the load generation matrix. An automated script is fired which triggers *OProfile* sampling with hardware counters same as Experiment 5. After 180 sec UST tracer dumps the trace file for both load and tbench program and the OProfile sampling also ends. Table 6.7 represents the test cases for this experiment.

Test Case	Kernel	Tracing	Load Configuration
T1	Vanilla	UST	CNF-01, CNF-04
T2	Vanilla	UST	CNF-02, CNF-05
T3	Vanilla	UST	CNF-03, CNF-06

Table 6.7: Test Cases for Experiment 6

6.3.3 Experiment 7 – Running the Kernel tracer with the help of Valgrind under various load configurations generated by load program (system load) and tbench (process and network load)

Objective – The experiment aims to find out the memory leaks with the help of Valgrind tool within LTT Control module during its run under different load configurations generated by load and tbench.

Explanation – In LTTng kernel we use load and tbench program one after the other to generate necessary load configurations in the system. We use the *Memcheck* utility of the Valgrind tool under the arguments of complete memory leak check and tracing of forked programs turned on. The usage detail of the tool is explained in the tools section of the Experiment Setup chapter. The Valgrind tool starts the Kernel Tracer in both Non Overwrite and in Flight Recorder tracing modes. After the load and tbench program completes the execution the trace is destroyed to get the Valgrind memory report. The report generated by Valgrind helps to get down to the functional level of instruction which is responsible for memory leaks in the LTT Control application (lttctl). Table 6.8 shows the test cases involved in the above experiment.

Test Case	Kernel	Tracing Mode	Valgrind Tool	Load Configuration
1	LTTng	Non Overwrite	Memcheck	CNF-01, CNF-04
2	LTTng	Non Overwrite	Memcheck	CNF-02, CNF-05
3	LTTng	Non Overwrite	Memcheck	CNF-03, CNF-06
4	LTTng	Flight Recorder	Memcheck	CNF-01, CNF-04
5	LTTng	Flight Recorder	Memcheck	CNF-02, CNF-05
6	LTTng	Flight Recorder	Memcheck	CNF-03, CNF-06

Table 6.8: Test Cases for Experiment 7

6.3.4 Experiment 8 – Running the load and tbench application instrumented with 10 markers under UST (Userspace Tracing) with the help of Valgrind

Objective – The objective of this experiment is to run UST Tracing on the load and tbench applications instrumented with 10 markers with the help of Valgrind *Memcheck* tool. This experiment will give us the detailed report of any memory leak issues faced by UST tracer under different load configurations.

Explanation – In fresh vanilla kernel we instrument the tbench and load with 10 markers and recompile it. We run the UST tracing on the instrumented load and tbench programs with *Memcheck* utility of Valgrind tool. The argument of tracing of forked programs are disabled for this experiment as it is not able to recognize the system call made by the forked child programs inside *usttrace* (Userspace Tracer) utility. After the load and tbench ends after its stipulated duration the Valgrind report gets generated. The reports generated by Valgrind are more detailed and drills down to functional level of the application program, i.e. we will get the line numbers of the application program which is responsible for the memory leaks. Table 6.9 shows the set of test cases designed for the experiment.

Test Case	Kernel	Tracing	Valgrind Tool	Load Configuration
T1	Vanilla	UST	Memcheck	CNF-01, CNF-04
T2	Vanilla	UST	Memcheck	CNF-02, CNF-05
T3	Vanilla	UST	Memcheck	CNF-03, CNF-06

Table 6.9: Test Cases for Experiment 8

7. Results

This chapter presents the analysis of results that are obtained by performing the experiments mentioned in the experiment methodology chapter. Detailed Results have been presented in Appendix A.

List of technical terms

LTTng	Linux Trace Toolkit Next Generation
UST	Userspace Tracer
CPU	Central Processing Unit
LTTD	Linux Trace Toolkit Daemon
USTD	Userspace Tracer Daemon

7.1 Load Configuration

7.1.1 Load Configuration parameters for System Under Test (SUT)

Experiment 1 enabled us to determine the load configuration parameters for the load generation utilities. Table 7.1 provides the parameters for different load configurations for *load* utility.

Load Configuration	Instances	Specified Load (%)	Execution Time (s)	Average CPU Usage (%)
CNF-01	4	20	180	21.16
CNF-02	4	50	180	50.53
CNF-03	4	90	180	89.52

Table 7.1: Results for Load Configuration of *load* utility

Table 7.2 provides the parameters for different load configurations for *tbench* utility.

Load Configuration	Clients	Specified Data Rate	Execution Time (s)	Average CPU Usage (%)
CNF-03	10	12	180	29.05
CNF-04	10	31	180	53.45
CNF-05	10	60.55	180	81.81

Table 7.2: Results for Load Configuration of *tbench* utility

From Table 7.1 and Table 7.2 we can observe that the average CPU usages for all load configurations are close to the target load described in Table 5.1. Therefore, we can proceed with these load configurations for the other experiments.

7.2 Control Flow Analysis

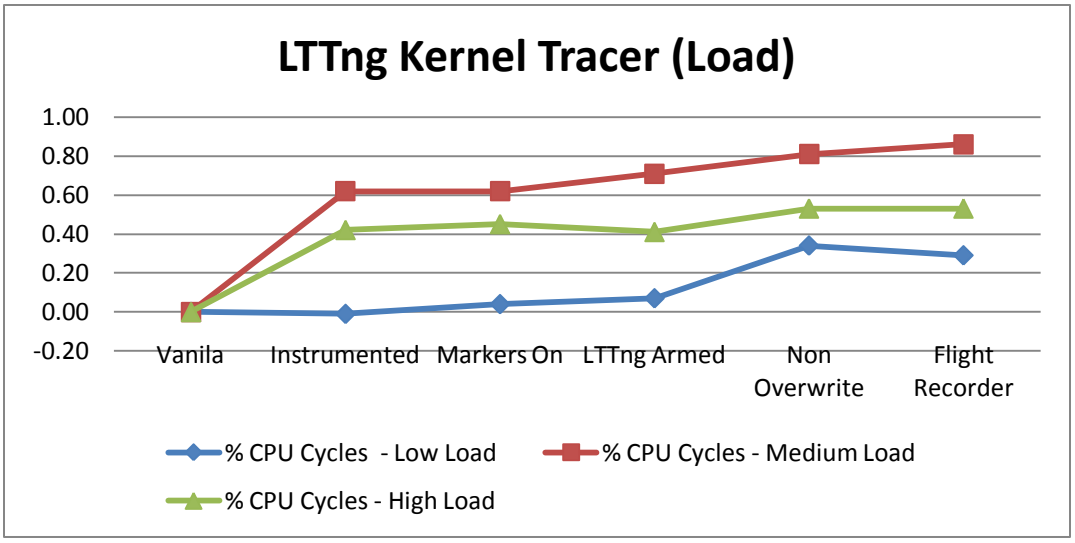
7.2.1 Efficiency of LTTng Kernel Tracer with Load utility

We have identified the performance impact of LTTng Kernel Tracer on the kernel operations of the system while the system is under varied amount of stress exerted by the Load utility. The percentage of impact has been calculated in terms of percentage of CPU cycles needed for performing kernel operations in different scenarios against the percentage of CPU cycles needed for performing kernel operations on a vanilla kernel. Table 7.3 provides the results.

KERNEL OPERATIONS - LOAD				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Vanilla	0.00	0.00	0.00	0.00
Instrumented	-0.01	0.62	0.42	0.34
Markers On	0.04	0.62	0.45	0.37
LTTng Armed	0.07	0.71	0.41	0.40
Non Overwrite	0.34	0.81	0.53	0.56
Flight Recorder	0.29	0.86	0.53	0.56

Table 7.3: Impact of LTTng kernel tracer on kernel operations (Load)

Graph 7.1 presents the impact of LTTng kernel tracer on kernel operations with Load utility executing in various configurations.



Graph 7.1: Impact of LTTng kernel tracer on kernel operations (Load)

From Graph 7.1 we can identify the effect of instrumented kernel, markers on, LTTng armed and tracing in non overwrite and flight recorder modes. Here, the impact varies with the amount of load exerted by

the load utility, low load having the least impact, high in between and medium load having the most impact. The percentage impact on kernel operations, calculated by measuring the percentage of CPU cycles needed, ranges between 0.25% to 0.85% with tracing on with almost negligible difference between non overwrite and flight recorder modes and average impact near 0.56%.

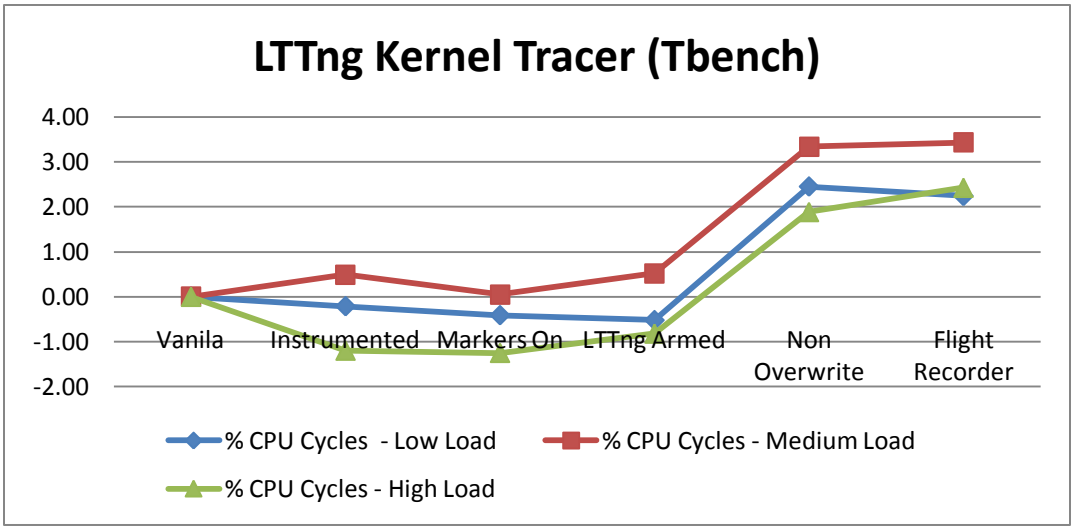
7.2.2 Efficiency of LTTng Kernel Tracer with Tbench utility

We have identified the performance impact of LTTng Kernel Tracer on the kernel operations of the system while the system is under varied amount of process and network load exerted by the Tbench utility. The percentage of impact has been calculated in terms of percentage of CPU cycles needed for performing kernel operations in different scenarios against the percentage of CPU cycles needed for performing kernel operations on a vanilla kernel. Table 7.4 provides the results.

KERNEL OPERATIONS - TBENCH				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Vanilla	0.00	0.00	0.00	0.00
Instrumented	-0.21	0.49	-1.20	-0.31
Markers On	-0.41	0.05	-1.25	-0.54
LTTng Armed	-0.52	0.52	-0.82	-0.27
Non Overwrite	2.45	3.34	1.89	2.56
Flight Recorder	2.24	3.43	2.43	2.70

Table 7.4: Impact of LTTng kernel tracer on kernel operations (Tbench)

Graph 7.2 presents the impact of LTTng kernel tracer on kernel operations with Tbench utility executing in various configurations.



Graph 7.2: Impact of LTTng kernel tracer on kernel operations (Tbench)

From Graph 7.2 we can identify the effect of instrumented kernel, markers on, LTTng armed and tracing in non overwrite and flight recorder modes. Here, the impact varies with the amount of load exerted by the tbench utility, high load having the least impact, low in between and medium load having the most impact. The percentage impact on kernel operations ranges between 1.85% to 3.45% with tracing on with almost negligible difference between non overwrite and flight recorder modes, the average impact with tracing on being approximately near 2.6%.

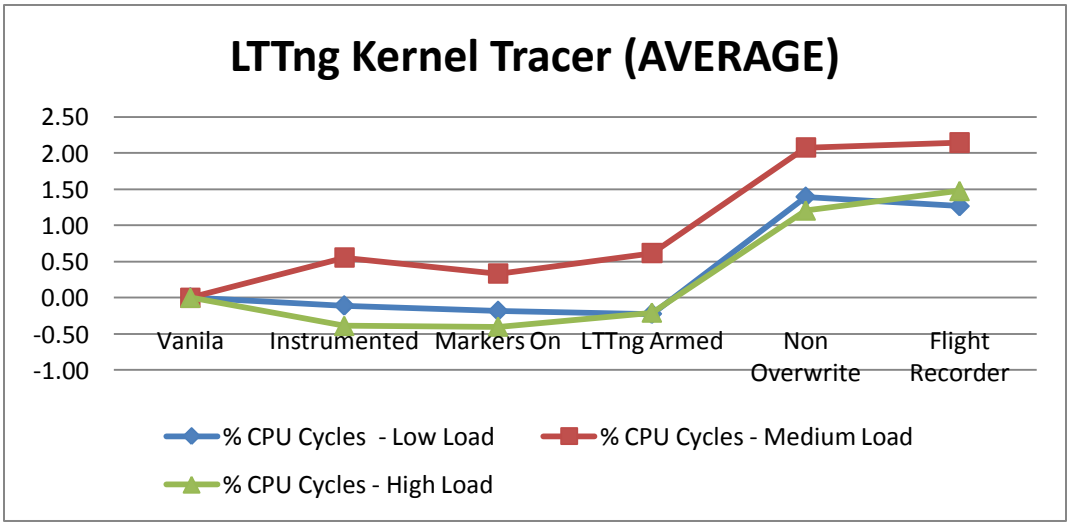
7.2.3 Efficiency of LTTng Kernel Tracer

The efficiency of LTTng kernel tracer is determined by calculating the average percentage CPU cycles needed for kernel operations between the load generators load and tbench. The averaged results are displayed in Table 7.5.

KERNEL OPERATIONS - AVERAGE				
Load Configuration	% CPU Cycles			AVERAGE %IMPACT
	Low Load	Medium Load	High Load	
Vanilla	0.00	0.00	0.00	0.00
Instrumented	-0.11	0.56	-0.39	0.02
Markers On	-0.19	0.34	-0.40	-0.08
LTTng Armed	-0.23	0.62	-0.21	0.06
Non Overwrite	1.40	2.08	1.21	1.56
Flight Recorder	1.27	2.15	1.48	1.63

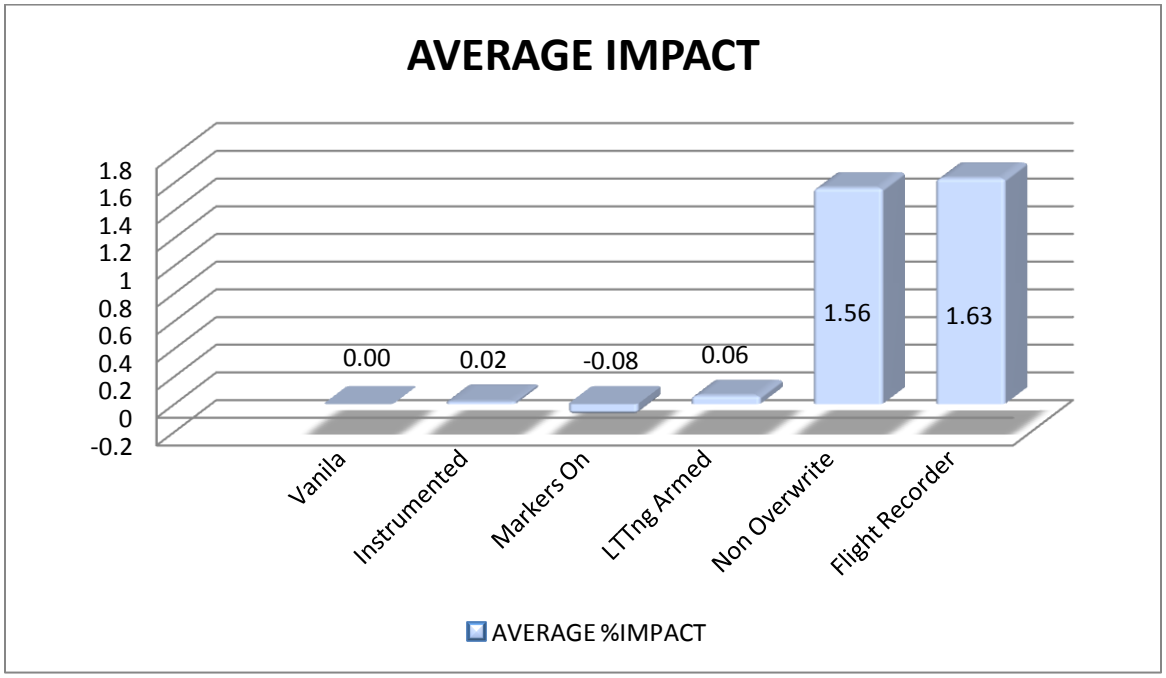
Table 7.5: Impact of LTTng kernel tracer on kernel operations (Average)

Graph 7.3 presents the impact of LTTng kernel tracer on kernel operations in low, medium and high load configurations.



Graph 7.3: Average Impact of LTTng kernel tracer on kernel operations for different load configurations

From Graph 7.3 we can observe the effect of instrumented kernel, markers on, LTTng armed and tracing in non overwrite and flight recorder modes. Here, the impact varies with the amount of load, high and low load performing similarly and medium load having the most impact. The percentage impact on kernel operations ranges between 1.2% and 2.15%, the average impact being near 1.5%. Graph 7.4 provides the average impact on kernel operations for all scenarios of LTTng kernel tracer.



Graph 7.4: Average Impact of LTTng Kernel Tracer

Therefore, from Graph 7.4, we can see that the impact of LTTng on kernel operations in terms of percentage of CPU cycles against vanilla kernel is (1.56 + 1.63)/2 = 1.6%. We can also observe that there is marginal difference between the performances of LTTng kernel tracer in Non Overwrite mode and in Flight Recorder mode.

7.2.4 Footprint of LTTng Kernel Tracer Daemon (LTTD)

Table 7.6 and Table 7.7 records the footprint of LTTng Kernel Tracer Daemon (LTTD) in terms of percentage of CPU cycles utilized by LTTD to operate through the trace sessions for load utility and tbench utility respectively.

LTTD - LOAD				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Non Overwrite	0.0003	0.0000	0.0000	0.0001
Flight Recorder	0.0005	0.0001	0.0001	0.0002

Table 7.6: Footprint of LTTD (Load)

LTTD - TBENCH				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Non Overwrite	0.0002	0.0001	0.0002	0.0002
Flight Recorder	0.0002	0.0004	0.0003	0.0003

Table 7.7: Footprint of LTTD (Tbench)

From the data in Table 7.6 and Table 7.7 we can identify that LTTD has very less footprint within the system and does not affect the system's performance by any means. Table 7.8 provides the average results for the footprint of LTTD in terms of percentage of CPU cycles needed for execution of LTTD.

LTTD - AVERAGE				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Non Overwrite	0.0003	0.0001	0.0001	0.0001
Flight Recorder	0.0004	0.0003	0.0002	0.0003

Table 7.8: Footprint of LTTD (Average)

From Table 7.8 we can see that LTTD has almost negligible footprint on both Non Overwrite mode and Flight Recorder mode. We have already seen that both the modes have almost similar amount of impact on the system's performance, but still LTTD takes more CPU cycles in flight recorder mode than in Non Overwrite mode.

7.2.5 Call Graph Analysis for LTTng Kernel Tracer

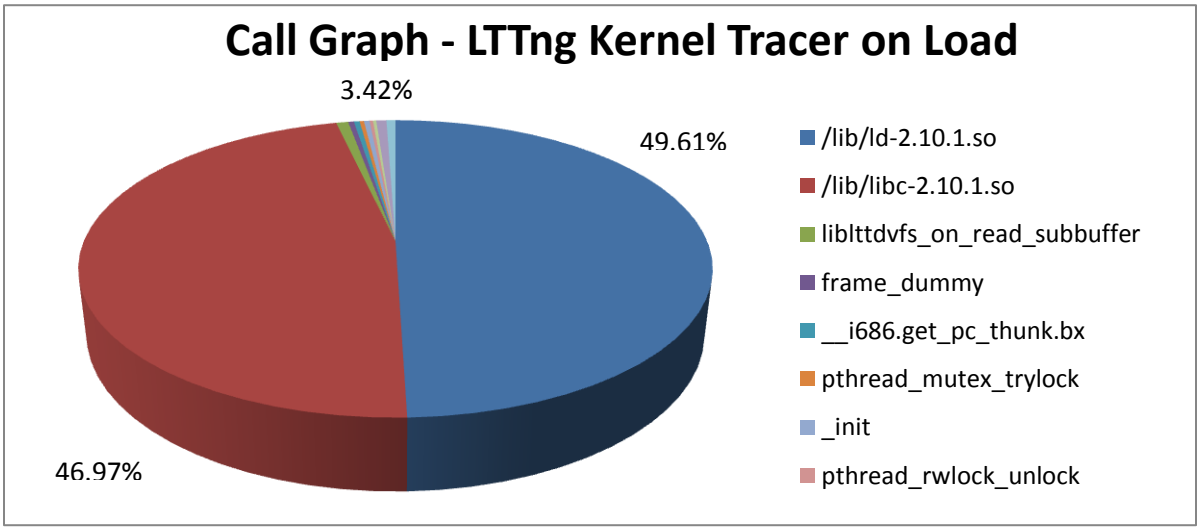
In our experiment we have generated call graphs for LTTng kernel tracer those help us to know in which libraries and functions are explored by the LTTng kernel tracer in course of its execution. We have identified the libraries and the respective functions where the LTTng kernel tracer spends time.

Table 7.9 provides the list of libraries and functions called during the execution of LTTng kernel tracer with the load utility and the average percentage of CPU time spent within the functions.

Library	Function(s)	Average
ld-2.10.1.so	/lib/ld-2.10.1.so	49.61
libc-2.10.1.so	/lib/libc-2.10.1.so	46.97
liblttd.so.0.0.0	liblttdvfs_on_read_subbuffer	0.73
	frame_dummy	0.35
	__i686.get_pc_thunk.bx	0.35
libpthread-2.10.1.so	pthread_mutex_trylock	0.26
	_init	0.35
	pthread_rwlock_unlock	0.21
	__pthread_diaable_asynccancel	0.21
	__close_nocancel	0.62
	__pthread_initialize_minimal	0.55

Table 7.9: Libraries and functions for LTTng Kernel Tracer (Load)

Graph 7.5 displays the average percentage of CPU time spent by LTTng on each function that in turn belongs to a library, with load utility.



Graph 7.5: Call Graph Analysis of LTTng Kernel Tracer on Load

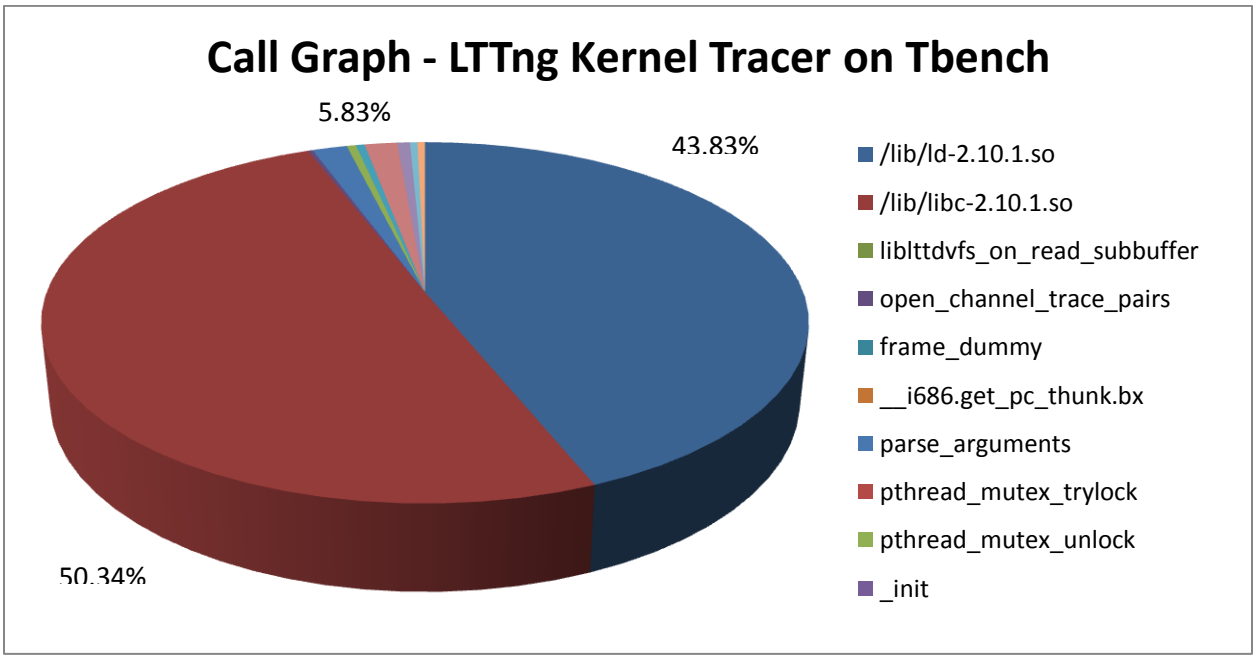
From Graph 7.5 we can observe that LTTng kernel tracer spends most of its time in *libc* and *ld* standard C libraries. It spends only 3.42% of the time in other functions and libraries which includes *liblttd*.

Table 7.10 provides the list of libraries and functions called during the execution of LTTng kernel tracer with the tbench utility and the average percentage of CPU time spent within the functions.

Library	Function(s)	Average
ld-2.10.1.so	/lib/ld-2.10.1.so	43.83
libc-2.10.1.so	/lib/libc-2.10.1.so	50.34
liblttd.so.0.0.0	liblttdvfs_on_read_subbuffer	0.00
	open_channel_trace_pairs	0.23
	frame_dummy	0.00
	__i686.get_pc_thunk.bx	0.00
lttd	parse_arguments	1.72
libpthread-2.10.1.so	pthread_mutex_trylock	0.00
	pthread_mutex_unlock	0.46
	_init	0.00
	sigaction	0.43
	pthread_rwlock_unlock	0.00
	__pthread_disable_asynccancel	0.00
	__reclaim_stacks	1.62
	__close_nocancel	0.00
	__do_global_dtors_aux	0.63
	pthread_create@@GLIBC_2.1	0.37
	__errno_location	0.37
	__pthread_initialize_minimal	0.00

Table 7.10: Libraries and functions for LTTng Kernel Tracer (Tbench)

Graph 7.6 displays the average percentage of CPU time spent by LTTng on each function that in turn belongs to a library, with tbench utility.



Graph 7.6: Call Graph Analysis of LTTng Kernel Tracer on Tbench

From Graph 7.6 we can observe that LTTng kernel tracer spends most of its time in *libc* and *ld* standard C libraries. It spends only 5.83% of the time in other functions and libraries which includes *liblttd*.

Therefore, we can say that as LTTng kernel tracer spends so less time in executing its own functions it has so little impact in the systems performance.

7.2.6 Efficiency of LTTng Userspace Tracer with Load utility

We have identified the performance impact of LTTng Userspace Tracer or UST on the Load utility while the system is under varied amount of stress exerted by the Load utility itself. The percentage of impact has been calculated in terms of percentage of CPU cycles needed for executing the instrumented Load binary in different scenarios against the percentage of CPU cycles needed for executing an original copy of the Load binary image. The instrumentation compiled within the Load utility has also been varied as 1, 5 and 10 instrumentations in order to observe the effect of varying markers as well. Table 7.11, Table 7.12 and Table 7.13 provides the results for load program compiled with 1, 5 and 10 markers respectively.

LOAD - 1 Marker				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Original	0.00	0.00	0.00	0.00
Markers On	0.33	0.30	0.33	0.32
UST On	0.02	0.26	0.23	0.17

Table 7.11: Impact of UST on Load with 1 marker

LOAD - 5 Markers				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Original	0.00	0.00	0.00	0.00
Markers On	0.57	0.56	0.18	0.44
UST On	0.53	0.45	0.33	0.44

Table 7.12: Impact of UST on Load with 5 markers

LOAD - 10 Markers				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Original	0.00	0.00	0.00	0.00
Markers On	0.47	0.46	0.48	0.47
UST On	0.50	0.54	0.56	0.53

Table 7.13: Impact of UST on Load with 10 markers

Table 7.14 provides the average data for the impact of UST on varying markers on Load utility executed with markers compiled without UST and with UST running, against an original run of the load utility without the markers.

LOAD – AVERAGE				
Load Configuration	% CPU Cycles			AVERAGE %IMPACT
	Low Load	Medium Load	High Load	
Original	0.00	0.00	0.00	0.00
Markers On	0.46	0.44	0.33	0.41
UST On	0.35	0.42	0.37	0.38

Table 7.14: Impact of UST on Load (Average)

From Table 7.14 we can identify that the userspace trace similar to the kernel tracer has very less impact on the userspace application. We know, when markers are compiled in, even if UST is not running, the control goes to the marker site and returns back. Therefore the impact of the compiled markers on UST can be justified. It is seen that UST has an impact of 0.38% on the load application.

7.2.7 Efficiency of LTTng Userspace Tracer with Tbench utility

We have identified the performance impact of LTTng Userspace Tracer or UST on the Tbench utility while the system is under varied amount of stress exerted by the Tbench utility itself. The percentage of impact has been calculated in terms of percentage of CPU cycles needed for executing the instrumented Tbench binary in different scenarios against the percentage of CPU cycles needed for executing an original copy of the Tbench binary image. The instrumentation compiled within the Tbench utility has also been varied as 1, 5 and 10 instrumentations in order to observe the effect of varying markers as well. Table 7.15, Table 7.16 and Table 7.17 provides the results for Tbench utility compiled with 1, 5 and 10 markers respectively.

TBENCH - 1 Marker				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Original	0.00	0.00	0.00	0.00
Markers On	0.81	0.57	0.50	0.63
UST On	0.89	0.49	0.57	0.65

Table 7.15: Impact of UST on Tbench with 1 marker

TBENCH - 5 Markers				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Original	0.00	0.00	0.00	0.00
Markers On	0.49	0.56	0.63	0.56
UST On	0.55	0.54	0.54	0.54

Table 7.16: Impact of UST on Tbench with 5 markers

TBENCH - 10 Markers				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Original	0.00	0.00	0.00	0.00
Markers On	0.72	0.47	0.58	0.59
UST On	0.57	0.49	0.63	0.56

Table 7.17: Impact of UST on Tbench with 10 markers

Table 7.18 provides the average data for the impact of UST on varying markers on Tbench utility executed with markers compiled without UST and with UST running, against an original run of the Tbench utility without the markers.

TBENCH - AVERAGE				
Load Configuration	% CPU Cycles			AVERAGE %IMPACT
	Low Load	Medium Load	High Load	
Original	0.00	0.00	0.00	0.00
Markers On	0.67	0.53	0.57	0.59
UST On	0.67	0.51	0.58	0.59

Table 7.18: Impact of UST on Tbench (Average)

From Table 7.18 it can be seen that both markers and UST has an impact of 0.59% on the Tbench application.

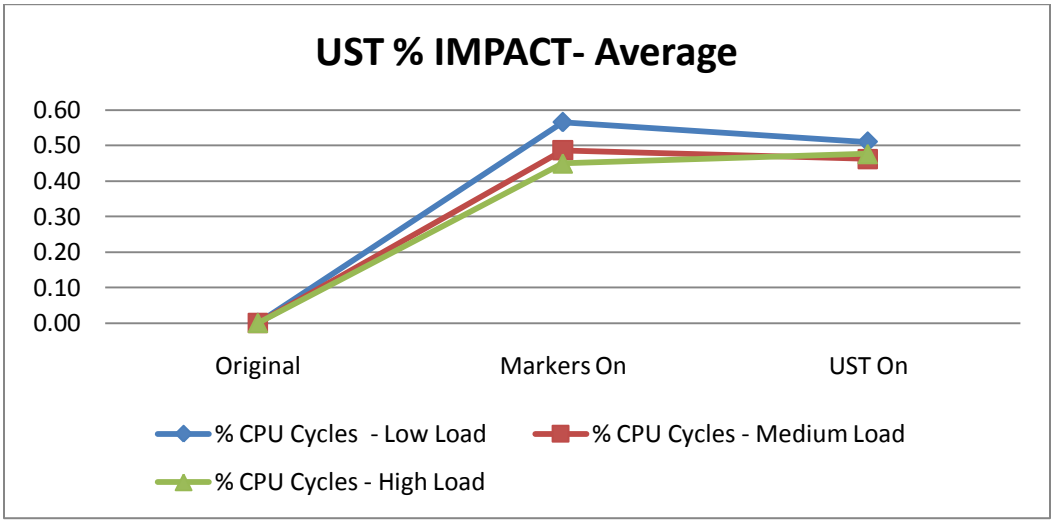
7.2.8 Efficiency of LTTng Userspace Tracer

The efficiency of LTTng userspace tracer is determined by calculating the average percentage impact on CPU cycles needed for the execution of the load generators load and tbench. The average is calculated from the already averaged data for load and tbench in Table 7.14 and Table 7.18 respectively. The final averaged results are displayed in Table 7.19.

AVERAGE				
Load Configuration	% CPU Cycles			AVERAGE %IMPACT
	Low Load	Medium Load	High Load	
Original	0.00	0.00	0.00	0.00
Markers On	0.57	0.49	0.45	0.50
UST On	0.51	0.46	0.48	0.48

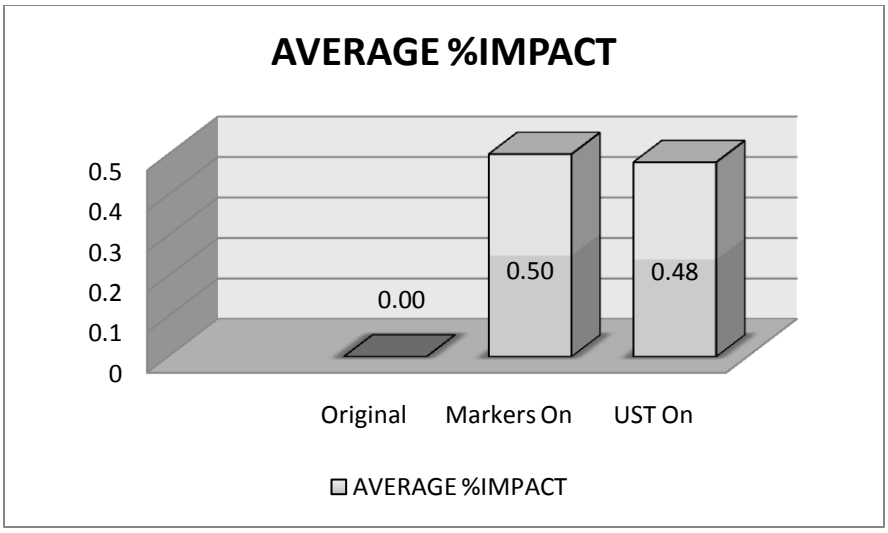
Table 7.19: Impact of UST on userspace applications (Average)

Graph 7.7 presents the impact of LTTng userspace tracer on userspace applications in low, medium and high load configurations.



Graph 7.7: Impact of UST on userspace applications for different load configurations

From Graph 7.7 we can observe that there is an impact of both compiled instrumentation as well as the userspace tracer on the traced application, but the impact is as low as between 0.45% and 0.51% depending on the load configurations. Graph 7.8 shows the average impact of UST on userspace applications.



Graph 7.8: Average Impact of UST on userspace applications

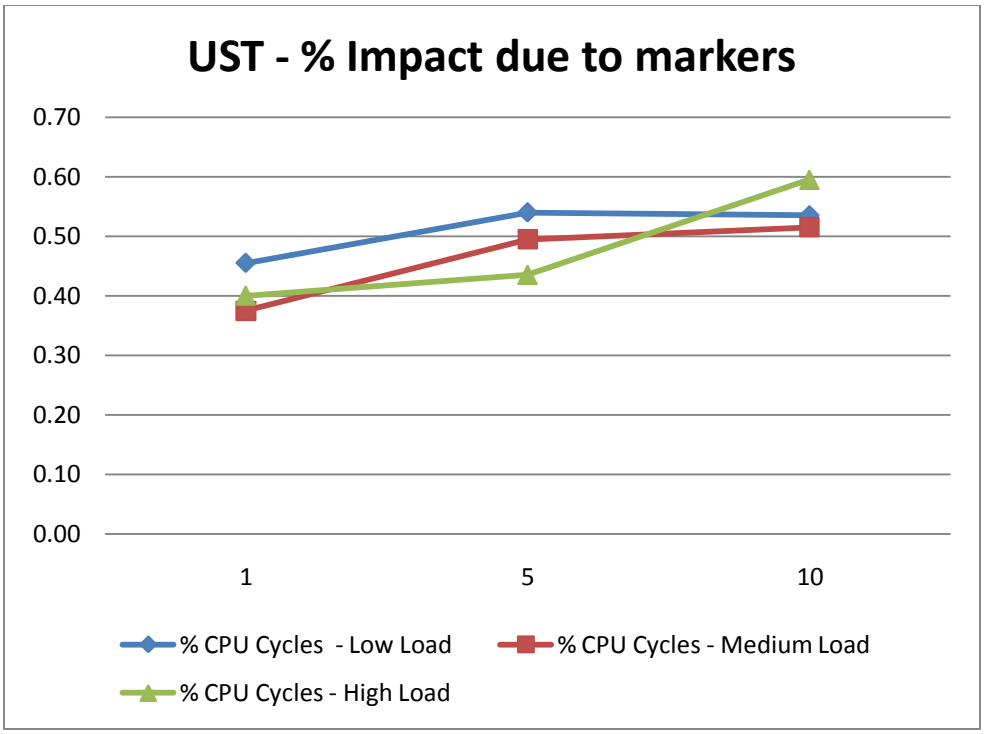
From Graph 7.8 we can identify that the LTTng userspace tracer and the compiled markers both have an effect of around 0.50% on the performance of the userspace application.

Table 7.20 provides the average impact of UST on userspace applications based on the number of markers for low, medium and high load configurations.

AVERAGE				
Markers	% CPU Cycles			AVERAGE %IMPACT
	Low Load	Medium Load	High Load	
1	0.46	0.38	0.40	0.41
5	0.54	0.50	0.44	0.49
10	0.54	0.52	0.60	0.55

Table 7.20: Impact of UST based on number of markers

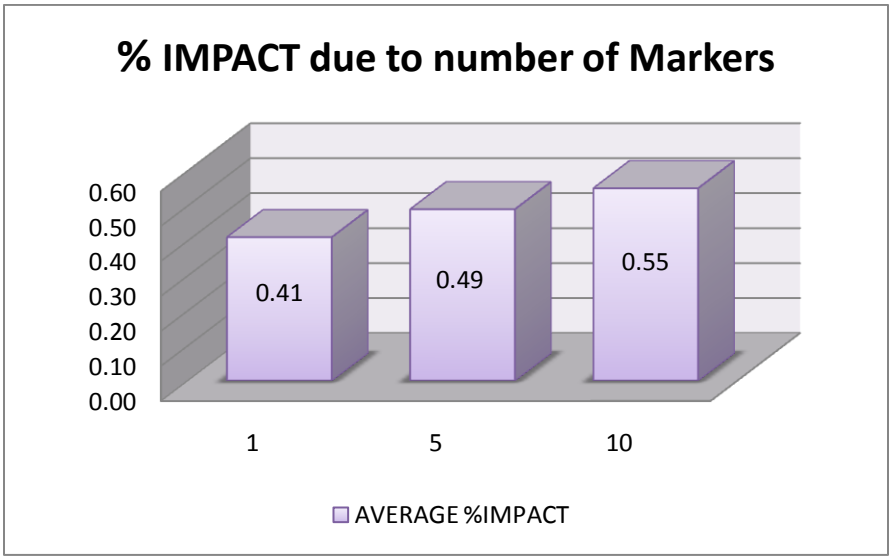
Graph 7.9 represents the impact of the number of markers compiled into the userspace application for low, medium and high load configurations.



Graph 7.9: Impact of UST based on number of markers for different load configurations

From Graph 7.9 we can observe that the impact of UST on userspace applications marginally increase with the increase in the number of instrumentations compiled in, though the pattern of increase for all load configurations are not similar.

Graph 7.10 provides the average impact of UST based on the number of markers.



Graph 7.10: Average Impact of UST based on number of markers

From Graph 7.10 we can observe that the percentage impact of UST on the userspace application increases with the increase of number of markers.

7.2.9 Footprint of LTTng Userspace Tracer Daemon (USTD)

Table 7.21 and Table 7.22 records the footprint of LTTng Userspace Tracer Daemon (USTD) in terms of percentage of CPU cycles utilized by USTD to operate through the trace sessions for load utility and tbench utility respectively for 1, 5 and 10 number of instrumentations in userspace application.

USTD - LOAD				
Markers	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
1	0.0017	0.0007	0.0005	0.0010
5	0.0018	0.0008	0.0003	0.0010
10	0.0017	0.0007	0.0005	0.0010

Table 7.21: Footprint of USTD (Load)

USTD - TBENCH				
Markers	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
1	0.0003	0.0002	0.0002	0.0002
5	0.0003	0.0002	0.0002	0.0002
10	0.0003	0.0003	0.0002	0.0003

Table 7.22: Footprint of USTD (Tbench)

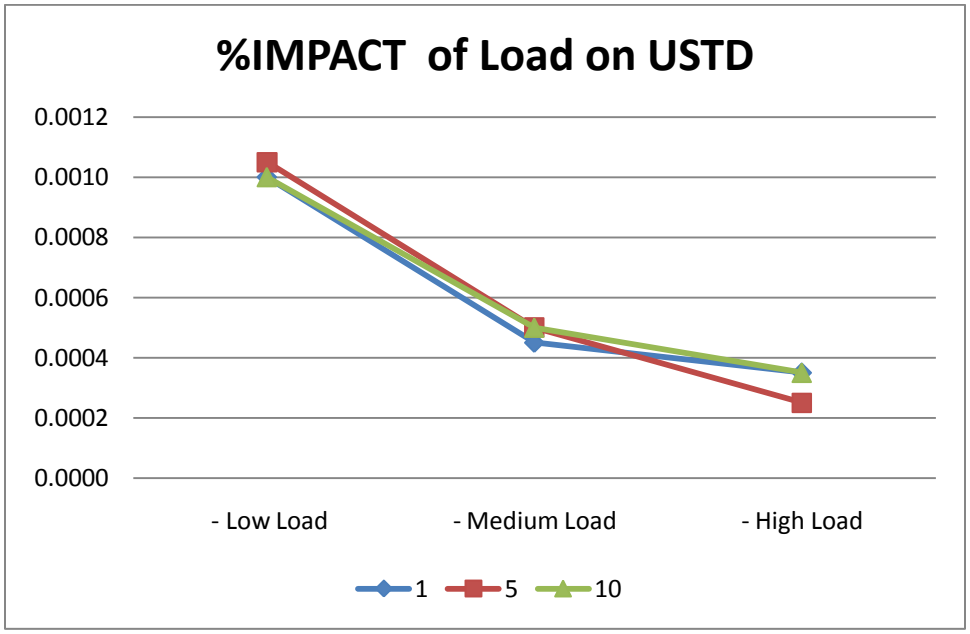
From the data in Table 7.21 and Table 7.22 we can identify that USTD has very less footprint within the system and does not affect the system's performance by any means. Table 7.23 provides the average results for the footprint of USTD in terms of percentage of CPU cycles needed for execution of USTD.

USTD – AVERAGE				
Markers	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
1	0.0010	0.0005	0.0004	0.0006
5	0.0011	0.0005	0.0003	0.0006
10	0.0010	0.0005	0.0004	0.0006

Table 7.23: Footprint of USTD (Average)

From Table 7.23 we can see that USTD has almost negligible footprint on the system for different load configurations or different number of markers. But it is noticeable that the footprint of USTD is not as good as compared to the footprint of LTTD. USTD has got a footprint a little higher than LTTD but still is almost negligible to affect the system's performance.

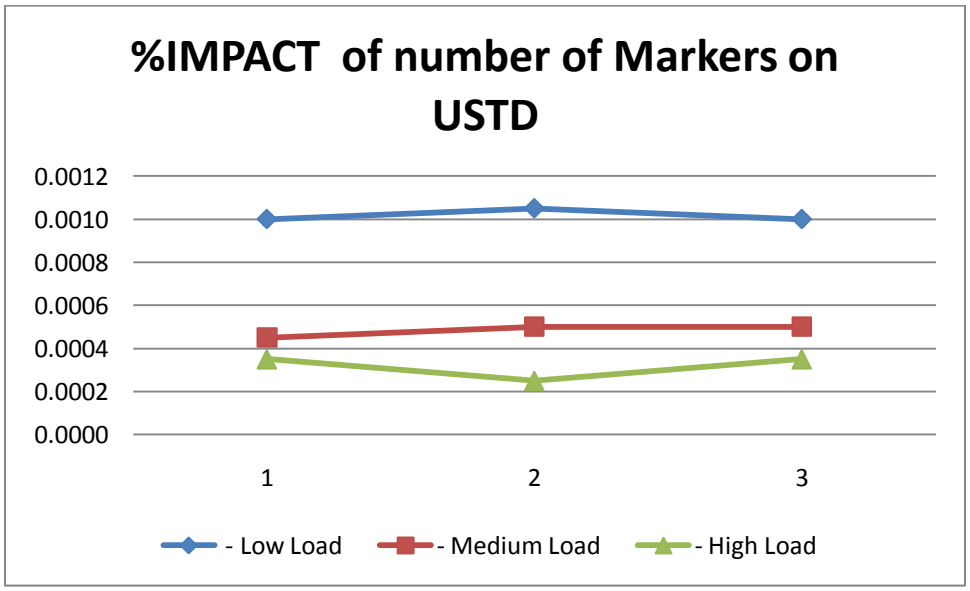
Graph 7.11 represents the impact of load on the footprint of USTD.



Graph 7.11: Impact of load on USTD

From Graph 7.11 we can observe that the footprint of USTD decreases as the load increases in the system. Therefore, the performance of USTD gets better with increasing amount of load.

Graph 7.12 presents the impact of the number of markers on USTD.



Graph 7.12: Impact of the number of markers on USTD

From Graph 7.12 we can observe that the footprint of UST is liner to the increasing number of markers. Therefore, the number of markers compiled in does not have any effect on the footprint of USTD.

7.2.10 Call Graph Analysis of LTTng Userspace Tracer

In our experiment we have generated call graphs for LTTng userspace tracer as well those help us to know in which libraries and functions are explored by the LTTng userspace tracer in course of its execution. We have identified the libraries and the respective functions where the LTTng userspace tracer spends time.

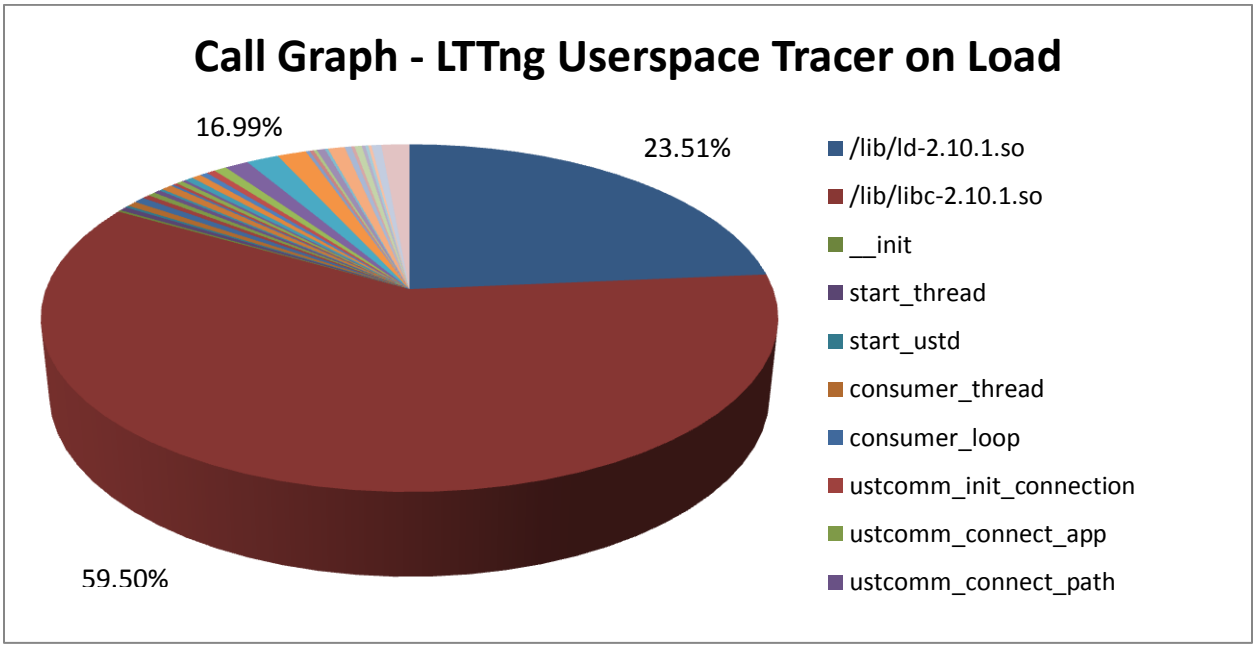
Table 7.24 provides the list of libraries and functions called during the execution of LTTng userspace tracer with the load utility and the average percentage of CPU time spent within the functions.

Library	Function(s)	Average
ld-2.10.1.so	/lib/ld-2.10.1.so	23.51
libc-2.10.1.so	/lib/libc-2.10.1.so	59.50
ustd	__init	0.17
	start_thread	0.38
	start_ustd	0.20
	consumer_thread	0.51
	consumer_loop	0.55
	ustcomm_init_connection	0.38
	ustcomm_connect_app	0.38
	ustcomm_connect_path	0.33
	ustcomm_send_requrst	0.19
	ustcomm_recv_message	0.56
	ustcomm_close_app	0.16
	get_subbuffer	0.21
	put_subbuffer	0.34
	unwrite_last_subbuffer	0.20
	connect_buffer	0.41
	__i686.get_pc_thunk.bx	0.51
	finish_consuming_dead_subbuffer	0.38
	__do_global_dtors_aux	0.44
	send_message_fd	0.66
	recv_message_fd	1.21
libpthread-2.10.1.so	__pthread_enable_asynccancel	1.65
	__pthread_disable_asynccancel	1.50
	__init	0.20
	connect	0.21
	send	0.17
	recv	0.42
	write	0.16
	pthread_create@@GLIBC_2.1	0.86

	__deallocate_stack	0.34
	__free_stacks	0.17
	pthread_mutex_lock	0.37
	pthread_mutex_unlock_usercnt	0.17
	__pthread_unregister_cancel	0.16
	__pthread_initialize_minimal	0.16
	__pthread_cleanup_push_defer	0.51
[heap]	[heap]	1.44

Table 7.24: Libraries and functions for LTTng Userspace Tracer (Load)

Graph 7.13 displays the average percentage of CPU time spent by LTTng on each function that in turn belongs to a library, with load utility.



Graph 7.13: Libraries and functions for LTTng Userspace Tracer (Load)

From Graph 7.13 we can observe that LTTng userspace tracer spends a bulk of time in *libc* and *ld* standard C libraries. It spends only 16.99% of the time in other functions and libraries which includes *liblttd*.

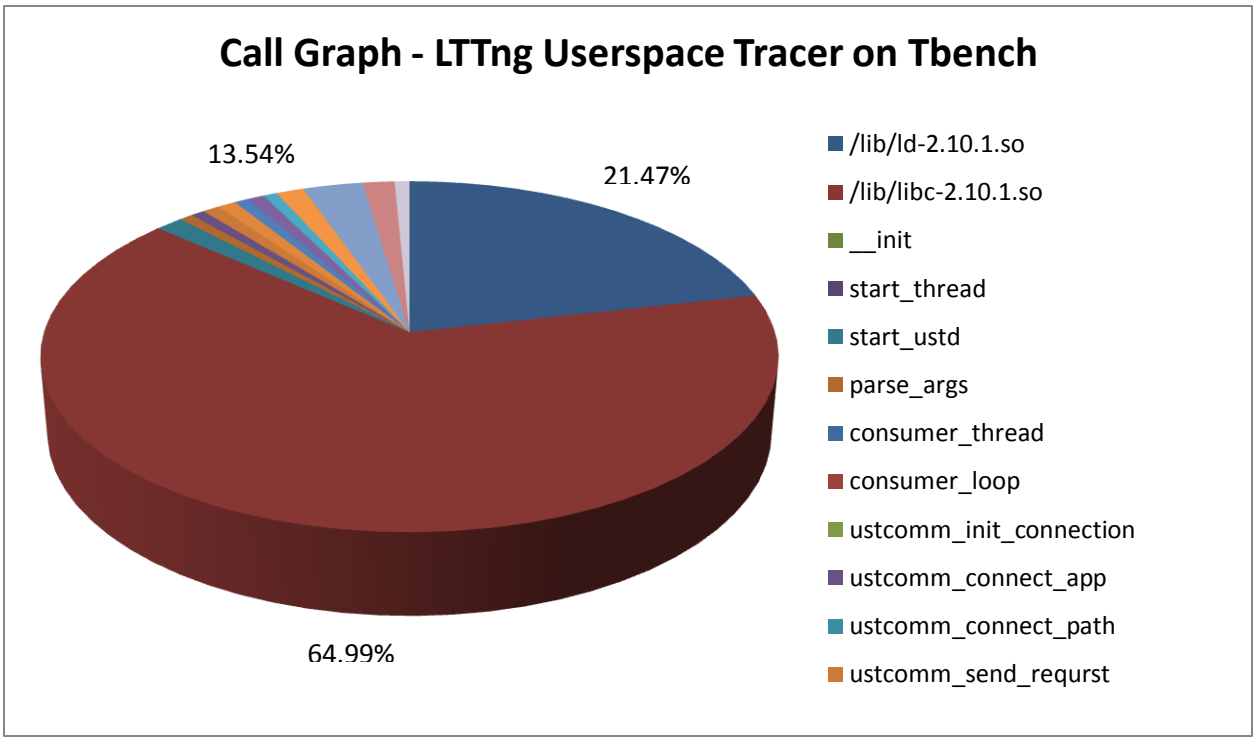
Table 7.25 provides the list of libraries and functions called during the execution of LTTng userspace tracer with the tbench utility and the average percentage of CPU time spent within the functions.

Library	Function(s)	Average
ld-2.10.1.so	/lib/ld-2.10.1.so	21.47

libc-2.10.1.so	/lib/libc-2.10.1.so	64.99
ustd	__init	0.00
	start_thread	0.00
	start_ustd	1.39
	parse_args	0.69
	consumer_thread	0.00
	consumer_loop	0.00
	ustcomm_init_connection	0.00
	ustcomm_connect_app	0.69
	ustcomm_connect_path	0.00
	ustcomm_send_requrst	0.85
	ustcomm_recv_message	0.00
	ustcomm_close_app	0.00
	get_subbuffer	0.00
	put_subbuffer	0.00
	unwrite_last_subbuffer	0.00
	connect_buffer	0.93
	__i686.get_pc_thunk.bx	0.69
	finish_consuming_dead_subbuffer	0.00
	__do_global_dtors_aux	0.00
	send_message_fd	0.85
	recv_message_fd	0.69
liburcu-bp.so.0.0.0	rcu_bp_register	1.39
libpthread-2.10.1.so	__pthread_enable_asynccancel	3.01
	__pthread_disable_asynccancel	1.60
	__init	0.00
	connect	0.00
	send	0.00
	recv	0.00
	write	0.00
	pthread_create@@GLIBC_2.1	0.00
	__deallocate_stack	0.00
	__free_stacks	0.00
	pthread_mutex_lock	0.00
	pthread_mutex_unlock_usercnt	0.00
	__pthread_unregister_cancel	0.00
	__pthread_initialize_minimal	0.00
	__pthread_cleanup_push_defer	0.00
[heap]	[heap]	0.74

Table 7.25: Libraries and functions for LTTng Userspace Tracer (Tbench)

Graph 7.14 displays the average percentage of CPU time spent by LTTng on each function that in turn belongs to a library, with tbench utility.



Graph 7.14: Libraries and functions for LTTng Userspace Tracer (Tbench)

From Graph 7.14 we can observe that LTTng userspace tracer spends a lot of its time in *libc* and *ld* standard C libraries. It spends only 13.54% of the time in other functions and libraries which includes *liblttd*.

We can observe that unlike LTTng kernel tracer, LTTng userspace tracer spends greater amount of its execution time in the C libraries, still it spends a lot of time (approximately 13% to 17%) in executing its own functions. Therefore, we can say that the LTTng userspace tracer is not as efficient as the LTTng kernel tracer and there is a scope of improving its performance.

7.2.11 Combined Impact of LTTng Kernel and Userspace Tracer

We have already evaluated the performance of LTTng kernel tracer and the userspace tracer separately. We also wanted to know if there is any additional impact on the system if LTTng kernel tracer and userspace tracer are executed together. LTTng kernel tracer was executed with instrumented load utility together with the LTTng userspace tracer. The impact can be identified against the percentage of CPU cycles required for the kernel operations for a vanilla kernel and a load program compiled without the markers. Table 7.26, Table 7.27 and Table 7.28 shows the results for 1, 5 and 10 instrumentations respectively.

KERNEL OPERATIONS - LOAD - 1 UST Marker				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Vanilla	0.00	0.00	0.00	0.00
Non Overwrite + UST	0.67	0.95	0.64	0.75
Flight Recorder + UST	0.84	0.88	0.61	0.78

Table 7.26: Impact of LTTng kernel tracer and UST on kernel operations for 1 marker in load

KERNEL OPERATIONS - LOAD - 5 UST Markers				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Vanilla	0.00	0.00	0.00	0.00
Non Overwrite + UST	0.61	0.88	0.60	0.70
Flight Recorder + UST	0.53	0.85	0.57	0.65

Table 7.27: Impact of LTTng kernel tracer and UST on kernel operations for 5 markers in load

KERNEL OPERATIONS - LOAD - 10 UST Markers				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Vanilla	0.00	0.00	0.00	0.00
Non Overwrite + UST	0.68	0.96	0.70	0.78
Flight Recorder + UST	0.67	1.01	0.52	0.73

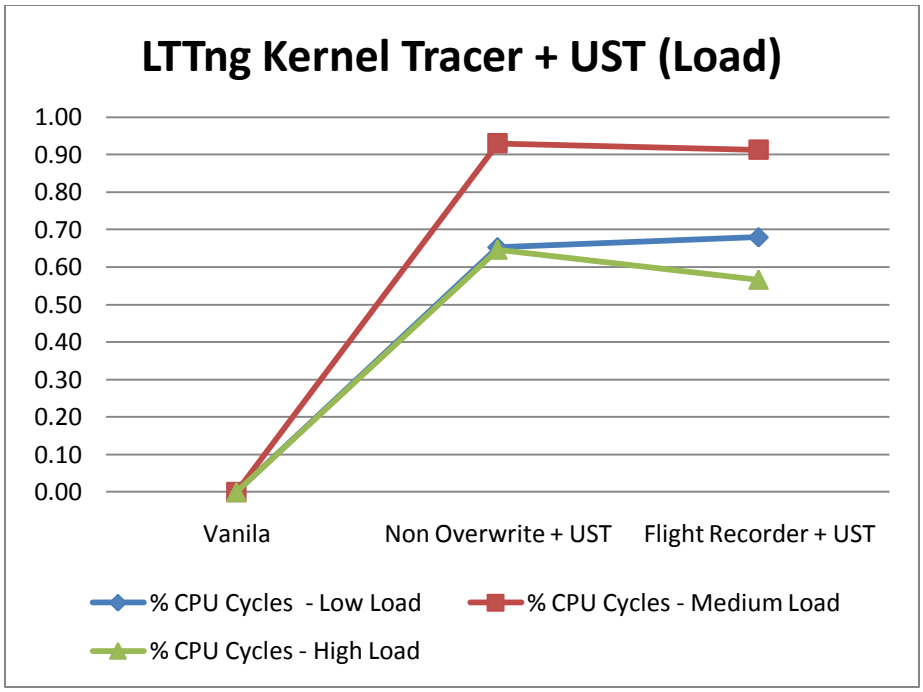
Table 7.28: Impact of LTTng kernel tracer and UST on kernel operations for 10 markers in load

Table 7.29 provides the average data for the combined impact of LTTng kernel tracer and UST on varying markers on load utility.

KERNEL OPERATIONS - LOAD - AVERAGE				
Load Configuration	% CPU Cycles			AVERAGE %IMPACT
	Low Load	Medium Load	High Load	
Vanilla	0.00	0.00	0.00	0.00
Non Overwrite + UST	0.65	0.93	0.65	0.74
Flight Recorder + UST	0.68	0.91	0.57	0.72

Table 7.29: Average Impact of LTTng kernel tracer and UST on kernel operations (Load)

Graph 7.15 shows the average combined impact of LTTng kernel tracer and userspace tracer on load utility for varying load configurations.



Graph 7.15: Impact of LTTng kernel tracer and UST on kernel operations (Load)

From Graph 7.15 we can observe that the impact is quite similar to LTTng kernel tracer with load utility where the average impact ranges between 0.5% and 1%.

LTTng kernel tracer was executed with instrumented tbench utility as well together with the LTTng userspace tracer. The impact can be identified against the percentage of CPU cycles required for the kernel operations for a vanilla kernel and a tbench utility compiled without the markers. Table 7.30, Table 7.31 and Table 7.32 shows the results for 1, 5 and 10 instrumentations respectively.

KERNEL OPERATIONS - TBENCH - 1 UST Marker				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Vanilla	0.00	0.00	0.00	0.00
Non Overwrite + UST	0.93	1.80	1.19	1.31
Flight Recorder + UST	1.10	2.31	1.86	1.76

Table 7.30: Impact of LTTng kernel tracer and UST on kernel operations for 1 marker in tbench

KERNEL OPERATIONS - TBENCH - 5 UST Markers				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Vanilla	0.00	0.00	0.00	0.00
Non Overwrite + UST	1.24	2.36	1.27	1.62
Flight Recorder + UST	1.65	2.81	1.82	2.09

Table 7.31: Impact of LTTng kernel tracer and UST on kernel operations for 5 markers in tbench

KERNEL OPERATIONS - TBENCH - 10 UST Markers				
Load Configuration	% CPU Cycles			AVERAGE
	Low Load	Medium Load	High Load	
Vanilla	0.00	0.00	0.00	0.00
Non Overwrite + UST	1.12	2.27	1.07	1.49
Flight Recorder + UST	1.59	2.95	1.86	2.13

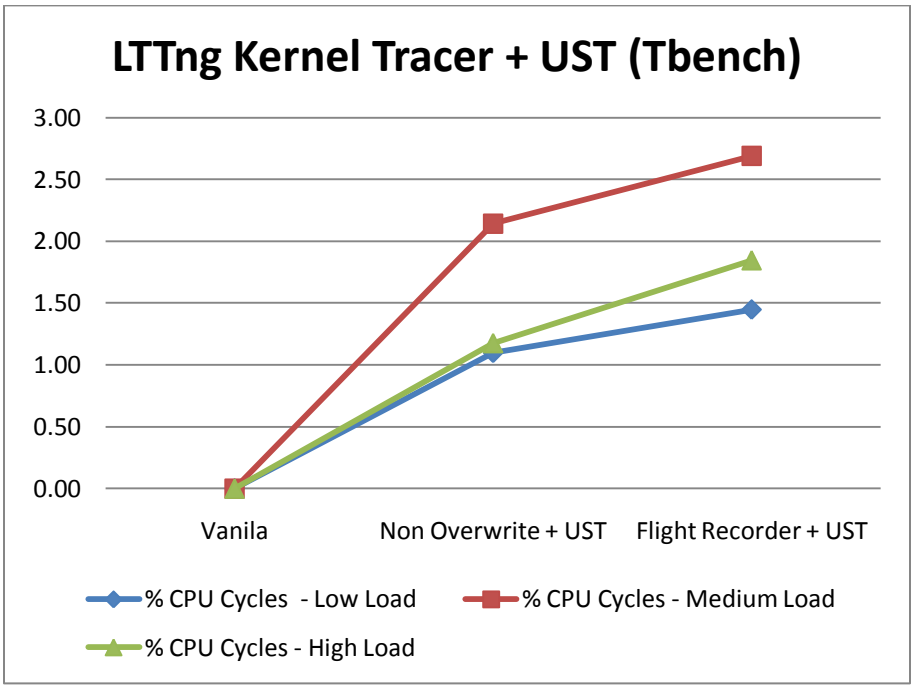
Table 7.32: Impact of LTTng kernel tracer and UST on kernel operations for 10 markers in tbench

Table 7.33 provides the average data for the combined impact of LTTng kernel tracer and UST on varying markers on tbench utility.

KERNEL OPERATIONS - TBENCH - AVERAGE				
Load Configuration	% CPU Cycles			AVERAGE %IMPACT
	Low Load	Medium Load	High Load	
Vanilla	0.00	0.00	0.00	0.00
Non Overwrite + UST	1.10	2.14	1.18	1.47
Flight Recorder + UST	1.45	2.69	1.85	1.99

Table 7.33: Average Impact of LTTng kernel tracer and UST on kernel operations (Tbench)

Graph 7.16 shows the average combined impact of LTTng kernel tracer and userspace tracer on load utility for varying load configurations.



Graph 7.16: Impact of LTTng kernel tracer and UST on kernel operations (Tbench)

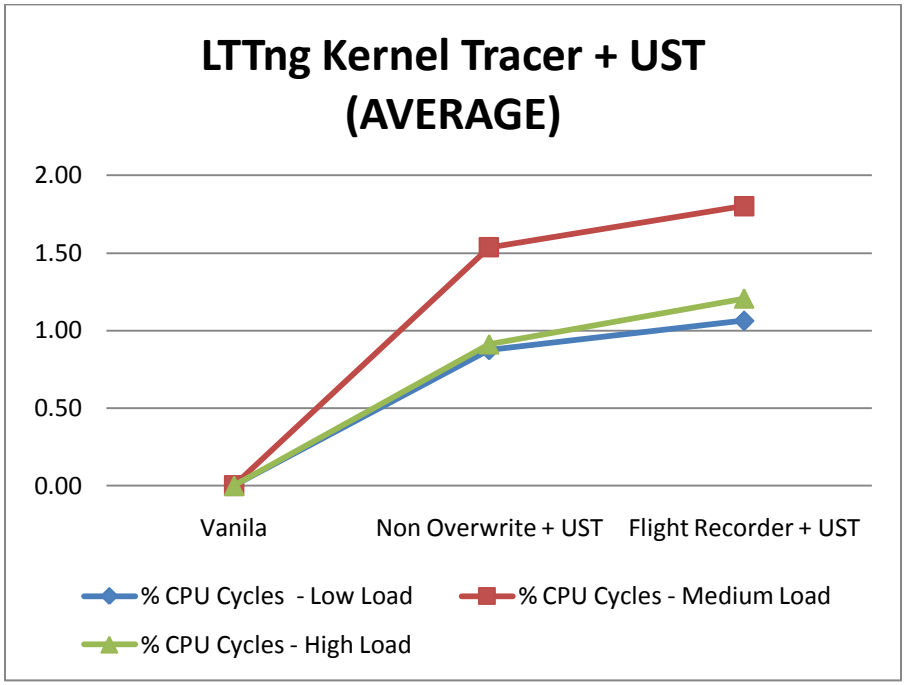
From Graph 7.16 we can observe that the impact is quite similar to LTTng kernel tracer with tbench utility where the average impact ranges around 1.5% to 2%.

Table 7.34 shows the average combined impact of LTTng kernel tracer and userspace tracer on varying load configurations.

KERNEL OPERATIONS - AVERAGE				
Load Configuration	% CPU Cycles			AVERAGE %IMPACT
	Low Load	Medium Load	High Load	
Vanilla	0.00	0.00	0.00	0.00
Non Overwrite + UST	0.88	1.54	0.91	1.11
Flight Recorder + UST	1.06	1.80	1.21	1.36

Table 7.34: Average Combined Impact of LTTng kernel tracer and UST on kernel operations

Graph 7.17 displays the average impact of LTTng kernel tracer and UST on kernel operations based on the percentage of CPU cycles for kernel operations against the kernel operation of a vanilla kernel with the load generator running without the markers compiled in.



Graph 7.17: Average Combined Impact of LTTng kernel tracer and UST on kernel operations

From Graph 7.17 we can observe that the impact is quite similar to LTTng kernel tracer and there is no additional impact on the percentage of CPU cycles needs to perform kernel operations. We can also observe that the LTTng Kernel tracer Non Overwrite mode has performed a bit better then the Flight Recorder mode while executed with Userspace Tracer in terms of CPU cycles needed for kernel operations.

7.3 Data Flow Analysis

7.3.1 L2 Caches Misses during execution of LTT Control Module with respect to various load configurations generated by load program and tbench

We sampled the whole system with OProfile Hardware event LII_MISSES to evaluate the cache misses of LTT Control Application (lttctl) under various load parameters generated by the load program and Tbench Application separately. The results are presented in Table 7.35 and Table 7.36 shows the L2 Cache Misses for LTT Control Module during Non Overwrite and Flight Recorder Tracing Modes for load program and tbench separately.

Load	Non Overwrite (lttctl)	Flight Recorder (lttctl)
Low	0.003333	0.002933
Medium	0.003100	0.000000
High	0.014500	0.018633

Table 7.35: L2 Cache Miss (lttctl) for load program

Tbench	Non Overwrite (lttctl)	Flight Recorder (lttctl)
Low	0.000029	0.000056
Medium	0.000023	0.000022
High	0.000014	0.000022

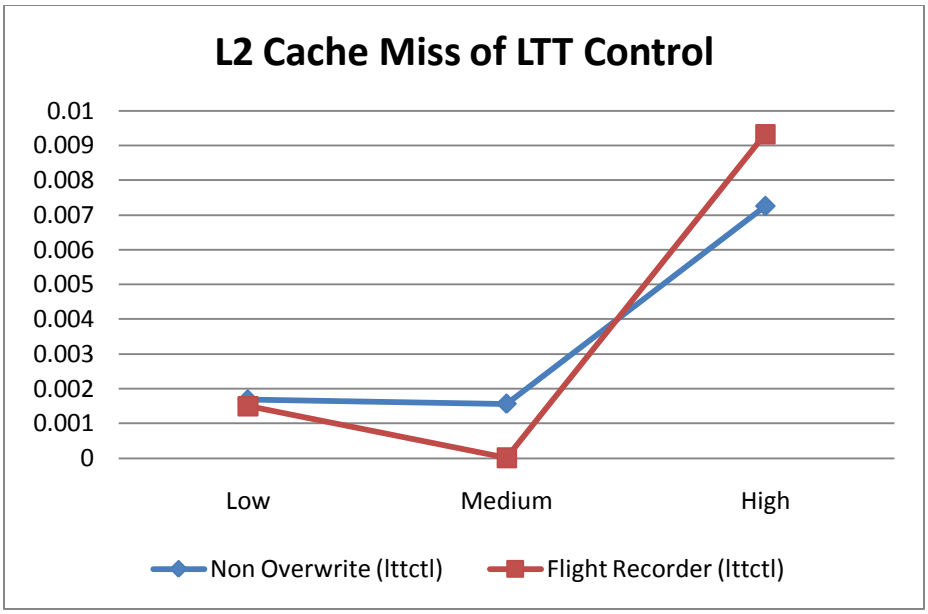
Table 7.36: L2 Cache Miss (lttctl) for tbench application

From Table 7.35 and Table 7.36 we can see that the Cache Miss of LTT Control (lttctl) is very less and in order of 10^{-4} percentage of samples collected by OProfile. For both the application and the different kind of load generated the cache miss trend seems to vary. Table 7.37 shows average L2 cache miss for lttctl.

Load/Tbench	Non Overwrite (lttctl)	Flight Recorder (lttctl)
Low	0.001681	0.0014945
Medium	0.0015615	0.000011
High	0.007257	0.0093275

Table 7.37: Average L2 Cache Miss (lttctl)

We now draw Graph 7.18, an average graph to conclude the overall Cache Miss rate for LTT Control Application with the effect of both load and tbench taken together.



Graph 7.18: Overall L2 Cache Miss Rate for LTT Control

From Graph 7.18 we can see that Non Overwrite Tracing and Flight Recorder tracing have similar L2 Cache misses for Low load configuration but maintains a steady difference in medium and high load for lttctl. Also the Cache misses dips for medium load and shoots up for high load configuration for both tracing modes.

7.3.2 L2 Cache Misses of LTT Daemon with respect to various load configurations generated by load program and tbench

When lttd (LTT Daemon) was sampled with OProfile for the same hardware counters as section 7.1.1 for L2 Cache Misses against different load configurations for load and tbench program we got the results as shown in Table 7.38 and Table 7.39.

Load	Non Overwrite (lttd)	Flight Recorder (lttd)
Low	0.003500	0.002933
Medium	0.000000	0.000000
High	0.000000	0.000000

Table 7.38: Cache Miss (lttd) for load program

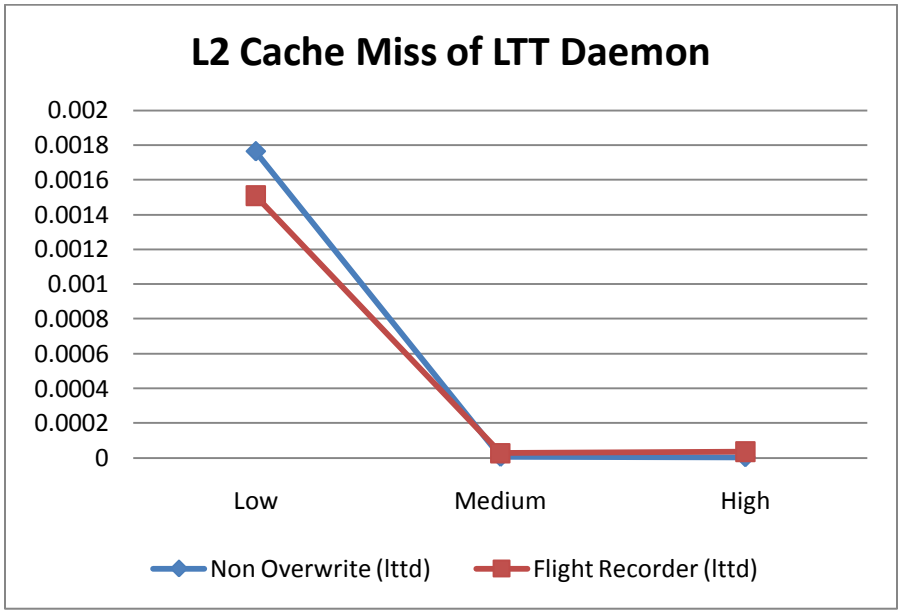
Tbench	Non Overwrite (lttd)	Flight Recorder (lttd)
Low	0.000029	0.000085
Medium	0.000011	0.000055
High	0.000007	0.000070

Table 7.39: Cache Miss (lttd) for tbench program

Overall cache misses are very low for LTT Daemon as seen previously for LTT Control. For lttd the cache miss is often not there means it's highly memory efficient. We take an average of the cache miss result for lttd in Table 7.40 and represent in the Graph 7.19.

Load/Tbench	Non Overwrite (lttd)	Flight Recorder (lttd)
Low	0.0017645	0.001509
Medium	0.0000055	0.0000275
High	0.0000035	0.000035

Table 7.40: Cache Miss (lttd)



Graph 7.19: Overall L2 Cache Miss Rate for LTT Daemon

Graph 7.19 shows that Cache Miss is highest for LTT Daemon in case the load configuration is low and reduces largely to be almost NULL when the load increases in system, process or network by load program and tbench respectively. The difference in Non Overwrite mode and Flight Recorder mode is almost negligible in any of load configurations.

7.3.3 Branch Mispredictions exhibited by LTT Control module with respect to various load configurations generated by load program and tbench

The whole system was sampled with OProfile Hardware event **INST_RETIRED_ANY_P** to evaluate the branch mispredictions of LTT Control Application (lttctl) under various load parameters generated by the load program and Tbench Application separately and in different tracing modes. The results are presented in Table 7.41 and Table 7.42.

Load	Non Overwrite (lttctl)	Flight Recorder (lttctl)
Low	0.000049	0.000052
Medium	0.000035	0.000033
High	0.000027	0.000011

Table 7.41: Branch Mispredictions (lttctl) for load program

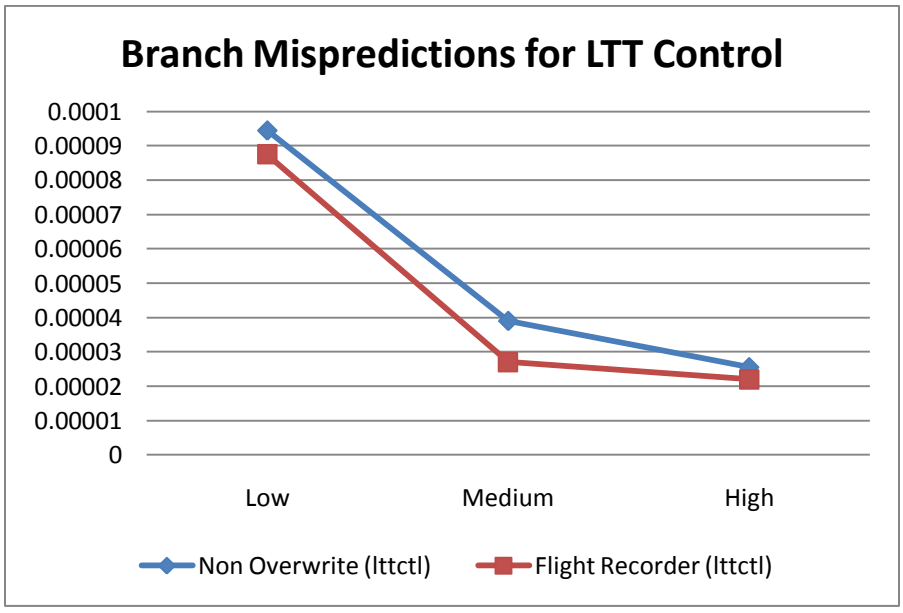
Tbench	Non Overwrite (lttctl)	Flight Recorder (lttctl)
Low	0.000140	0.000123
Medium	0.000043	0.000021
High	0.000024	0.000033

Table 7.42: Branch Mispredictions (lttctl) for tbench

Branch Mispredictions are also much low at 10^{-4} samples of OProfile. For Non Overwrite tracing mode for both the load and tbench program the Branch Misprediction rate of **lttctl** decreases upon the increase of load on the system. We calculate the average effect for both the programs in Table 7.43 and form Graph 7.20 from the result to determine the average Branch Mispredictions rate for LTT Control.

Load/Tbench	Non Overwrite (lttctl)	Flight Recorder (lttctl)
Low	0.0000945	0.0000875
Medium	0.000039	0.000027
High	0.0000255	0.000022

Table 7.43: Branch Mispredictions (lttctl)



Graph 7.20: Overall Branch Misprediction for LTT Control

In Graph 7.20 the trend shows that the Branch Mispredictions differ much less between Non Overwrite and Flight Recorder tracing modes on any different load configurations. We can also see that when load increases, Branch Mispredictions decrease.

7.3.4 Branch Mispredictions of LTT Daemon with respect to various load configurations generated by load program and tbench

When LTT Daemon was sampled for Branch Mispredictions under various load configurations generated by load program and tbench, we got results which are tabulated in the Table 7.44 and Table 7.45.

Load	Non Overwrite (lttd)	Flight Recorder (lttd)
Low	0.000040	0.000237
Medium	0.000031	0.000113
High	0.000029	0.000115

Table 7.44: Branch Mispredictions (lttd) for load program

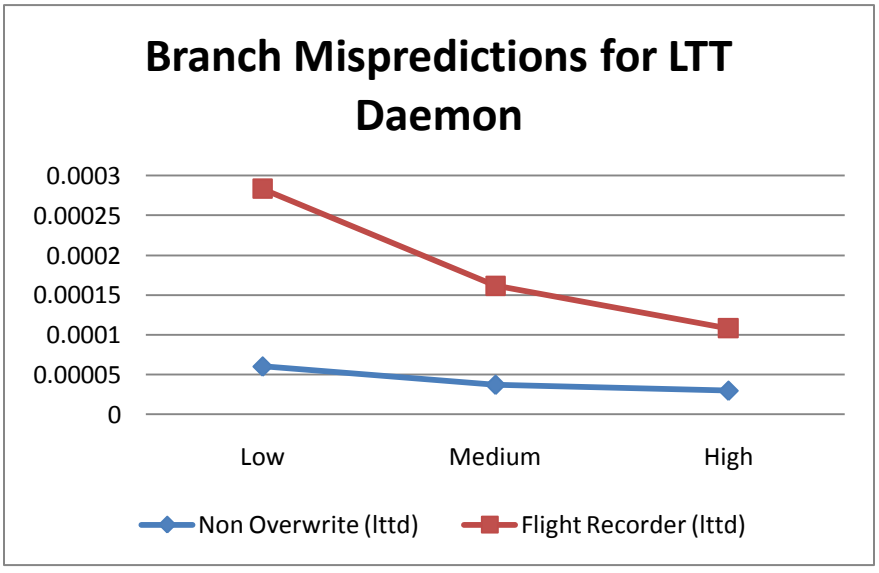
Tbench	Non Overwrite (lttd)	Flight Recorder (lttd)
Low	0.000080	0.000330
Medium	0.000043	0.000210
High	0.000030	0.000101

Table 7.45: Branch Mispredictions (lttd) for tbench

The LTT Daemon was also sampled with same Hardware counter by OProfile as the earlier 7.3.3 Section. The consolidated average result for the Branch Mispredictions for LTT Daemon is represented in Table 7.46 and Graph 7.21.

Load/Tbench	Non Overwrite (lttd)	Flight Recorder (lttd)
Low	0.00006	0.0002835
Medium	0.000037	0.0001615
High	0.0000295	0.000108

Table 7.46: Branch Mispredictions (lttd)



Graph 7.21: Overall Branch Misprediction for LTT Daemon

From Graph 7.21, we can see the branch predictions of LTT Daemon also scales down similarly like the LTT Control application with increase in system, process or network load from load program and tbench. But unlike the LTT Control module, the LTT Daemon exhibits a difference in Branch Mispredictions between Non Overwrite and Flight Recorder tracing modes. Branch Mispredictions is more in Flight Recorder mode for LTT Daemon.

7.3.5 Analysis of Memory Leak of LTT Control and LTT Daemon program during execution with respect to various load configurations generated by load program and tbench

When we used Valgrind memory checking tool *Memcheck* to trigger the kernel tracing (firing up lttctl) it was seen that for any types of load configuration the memory leaks for LTT Control is very minimal and constant. LTT Daemon showed a zero memory loss during its execution. The Summarized results are in Table 7.47.

Tracing Modes	Lost Blocks	Lost Memory	Blocks(Not Free)	Memory not Freed
Non Overwrite	17	152 bytes	3	988
Flight Recorder	34	304 bytes	3	988

Table 7.47: Memory Leak for LTT Control (Kernel Tracer)

From Table 7.47 we see that Flight Recorder mode registered a memory loss of 304 bytes double to that of Non Overwrite tracing mode under any load circumstances. Also the there is a equal amount of memory, 988 bytes which are not released or rather freed after completion of execution in both the tracing modes. The lines of code in the LTT Control application responsible for the memory losses are also captured and are given in details in Appendix A.

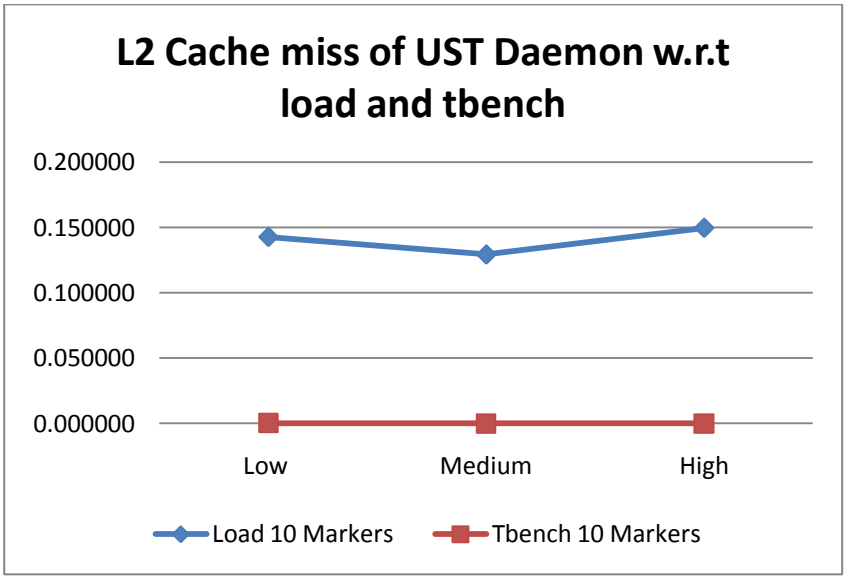
7.3.6 L2 Cache Misses for UST Daemon during tracing of load and tbench program (10 markers) under various load configurations

To find out L2 Cache misses for UST, OProfile was run with **LII_MISSES** hardware event. The UST Tracing was done on the load program and tbench program one after the other instrumented with 10 markers. The result is displayed in Table 7.48.

Load	Load 10 Markers	Tbench 10 Markers
Low	0.142800	0.000293
Medium	0.129433	0.000116
High	0.149700	0.000096

Table 7.48: L2 Cache Miss for UST Daemon

Graph 7.22 displays the L2 cache miss of UST Daemon with respect to load and tbench.



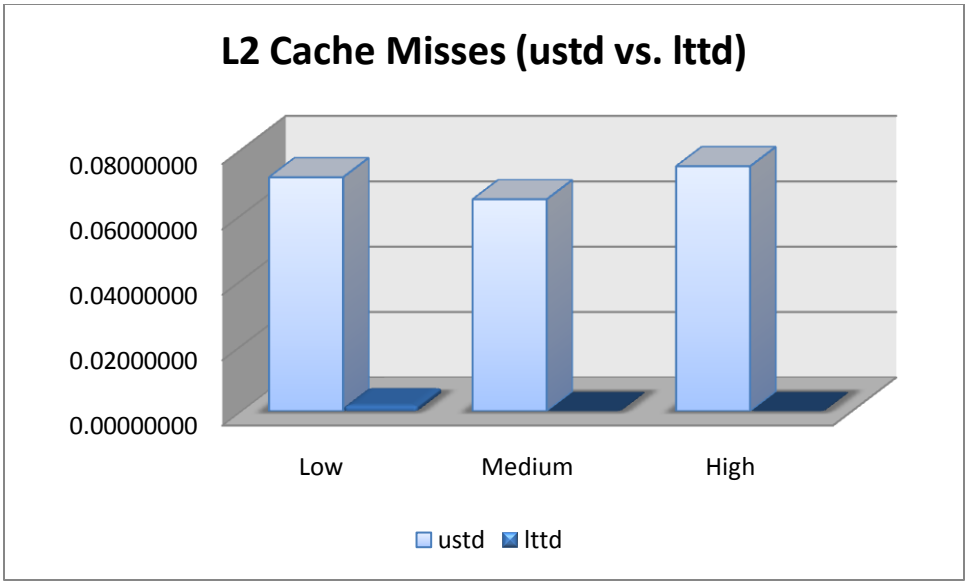
Graph 7.22: L2 Cache Miss for UST Daemon

From Table 7.48 and the Graph 7.22 we can see for load program the L2 Cache miss dips a bit for Medium load and shoots up for high load. For Tbench program the UST Daemon Cache miss shows it is very minimal and decreases with increase in load.

We tried to compare the performances with respect to cache misses for both LTT Daemon (lttd) and UST Daemon (ustd) from the experiments done in Table 7.49 and we came up with Graph 7.23.

Load	ustd	lttd
Low	0.07154650	0.00163675
Medium	0.06477450	0.00001650
High	0.07489800	0.00001930

Table 7.49: L2 Cache Miss (ustd & lttd)



Graph 7.23: L2 Cache Miss (ustd vs. lttd)

From Graph 7.23 we can see that though both **ustd** and **lttd** have very less percentage of cache misses but UST daemon has a big scope to improve in L2 Cache hits with respect to LTT Daemon. For different load configuration UST Daemon has lot more cache misses when compared to LTT Kernel Tracer daemon.

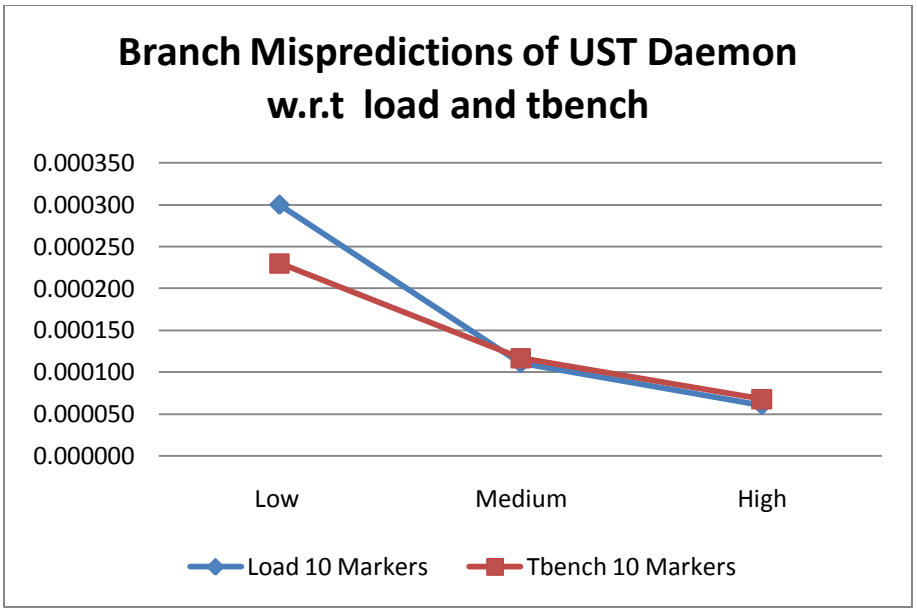
7.3.7 Branch Misprediction for UST Daemon during tracing of load and tbench program (10 markers) under various load configurations

To find out Branch Mispredictions, OProfile was run with **INST_RETIRED_ANY_P** hardware event. UST was tracing under varying load configurations the tbench and load program which was each instrumented with 10 markers and recompiled in the system. Table 7.50 shows the result data.

Load	Load 10 Markers	Tbench 10 Markers
Low	0.000300	0.000230
Medium	0.000111	0.000117
High	0.000061	0.000068

Table 7.50: Branch Mispredictions for UST Daemon

Graph 7.24 represents the branch mispredictions of UST Daemon with respect to load and tbench.



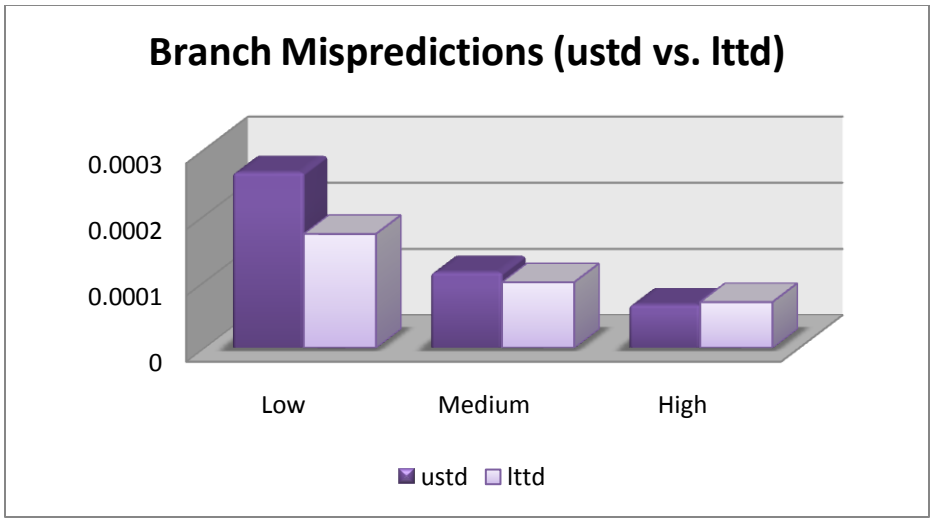
Graph 7.24: Branch Mispredictions for UST Daemon

From the Graph 7.24 and the Table 7.50 we can say that Branch Mispredictions for UST Daemon doesn't depend on the application it is tracing as for both load and tbench it shows similar trend of higher branch mispredictions during low load and lower branch mispredictions during higher load. The branch mispredictions gradually decrease with increase in load.

We also compared the average branch misprediction rate of both **lttd** and **ustd** which is represented by the Table 7.51 and Graph 7.25.

Load	ustd	lttd
Low	0.000265	0.000172
Medium	0.000114	0.000099
High	0.000065	0.000069

Table 7.51: Branch Mispredictions (ustd & lttd)



Graph 7.25: Branch Mispredictions (ustd vs. lttd)

The Branch Mispredictions graph plotted for **ustd** and **lttd** shows that LTT Kernel Tracer Daemon performs better than UST daemon is low load but in High load there is no performance difference between the two.

7.3.8 Analysis of Memory Leak of UST Tracer during execution with respect to various load configurations generated by load program and tbench

Load and tbench which are instrumented with 10 markers and UST tracing was done on it one after the other. During the UST tracing on each load and tbench, it was run with the help of *Memcheck* utility of Valgrind which helped to get the report of any memory leaks during execution. In Table 7.52 is the result from the experiment.

Application	Blocks (Lost)	Memory Lost	Blocks (Not Free)	Memory not Freed
load	1	654 bytes	1267	33599 bytes
tbench	1	654 bytes	1270	33658 bytes

Table 7.52: Memory Leak for UST Tracer (load & tbench)

From the Table 7.51 we can find out that with increase of load in system doesn't affect the memory leak of the User Space Tracer. During tracing of both load program and tbench Userspace Tracer lost 1 Block of data (654 bytes), and that was during saving the trace data to the disk. But the problem with Userspace Tracer seemed to be the number of allocated memory spaces which it doesn't free when it completes its execution. It has approximately 33600 bytes in 1270 blocks of blocked data after completion of execution. The memory blocks not freed doesn't depend too much on the application it traces as we can see from Table 7.52 where the difference is very less among tbench and load program.

8. Discussion

This chapter mainly focuses upon the constraints of experiments executed and the issues faced during the research period. The issues discussed concentrates upon the unavailability of tools and time limitation of the thesis standing as the main barriers. Last part of the Discussion aims to evaluate the benefits of this research to the community and the industry.

List of technical terms

LTTng	Linux Trace Toolkit Next Generation
AMP	Asymmetric Multiprocessing
SMP	Symmetric Multiprocessing
CPU	Central Processing Unit
LTTD	Linux Trace Toolkit Daemon
USTD	Userspace Tracer Daemon

8.1 Limitations of the performed experiments

Initially all the experiments were scheduled to be carried out in a P4080 multiprocessor board with 8 cores having an AMP setup. But due to unavailability of hardware setup, all the experiments were performed on an Intel Quad Core multiprocessor having a SMP setup. Due to SMP setup and Quad Core all the arenas and possibilities of multicore environment couldn't be explored. In SMP the Operating System controls the cores, rather than in AMP where every core has a separate Operating System.

Due to the time constraint all the experiments were run only 3 times, but to get appropriate expected result we needed 20 runs approximately. The consistency of the results could only have been judged by such extensive experiments as the percentage of CPU samples varied within a range of 10^{-4} to 10^{-6}.

We used the profiling and sampling tool called OProfile as we found it quite efficient than other tools for the particular set of information we were looking for. But the main drawback of this tool is it cannot execute properly in a virtual system like Virtutech Simics or Virtual box as it doesn't get access to respective hardware events. When we were using OProfile we faced an issue during experimentation in which there was significant amount of buffer overflows due to high rate of sampling and longer run durations. We minimized the sampling frequency and thus the overflow got down below 1%.

During the execution of the experiments when OProfile continuously was collecting samples, after certain point of time the Opreport failed to gather data from the system to generate a sample report. This was due to memory stack overflow and memory flush errors, by which the system couldn't dump the earlier samples collected. For every case like this the system was rebooted and the test cases were rerun and it completed successfully.

For gauging the efficiency of LTTng and measuring its footprint we only could analyze the binaries which were running in the system, i.e. the LTTD (LTTng daemon for kernel tracing) and USTD (User Space Tracing daemon). The major portions of code and additions of LTTng are inside the kernel as patches. For Data Flow analysis and code coverage especially the patched kernel should have been tested extensively with respect to kernel analysis tools which we left out of the scope of the thesis because of time constraints.

For testing the working of LTT agent with Eclipse LTTng tool we found many hiccups during successfully setting up the system for the test configurations. We reported the errors for many of them. Still there is a absence of proper manual for carrying out the setup and also due to lack of time we decided not carrying on further with the LTT agent and Eclipse LTTng tool experiments.

We used benchmark tool such as tbench for generating load into the system. More benchmark tools could have been used to make the experiment real time with minimal amount of limitations.

8.2 Choice of Control and Data Flow Analysis Tools

Control Flow tools such as OProfile, Gprof and sysprof came in our first set of tools marked to measure the control flow of the system. But from the above three OProfile was chosen best to serve our purpose.

Problems with Gprof

- It couldn't handle multithreaded application and thus application binaries with multi threading were not profiled appropriately.
- The program needs to be compiled with '-pg' parameters thus increasing the overhead of statically linking it before execution. For this reason third part binaries are difficult to be sampled with Gprof [LYN10].

Problems with sysprof

- LTTD and USTD has such low footprints that system wide profiler couldn't collect the sample needed for it. During the LTTng run when the system was profiled there was no samples for either LTTD or USTD. But we had to get results to proof the low footprint and thus this tool was also not used.

For Data Flow analysis we had in mind many tools based on their usage and way of working. The main tools decided upon were Zoom, Valgrind, Acumem and OProfile. Acumem was a very efficient tool and exactly served our purpose for getting the pain points inside the LTTD and USTD application but due to having its evaluation license there was limitations in capturing of samples and thus we had to opt out from this tool. For our experiments we needed a tool which can attach itself with the running program and sample it so that during its run the issues in the code can be gathered, but no tool provided this except Acumem.

Problems with Zoom system profiler

- Again because of very low foot print Zoom cannot capture very low sampled events like LTTD and USTD.
- Zoom cannot attach itself to running programs, though provides all sorts of valuable code refactoring guidelines for a hugely sample application.

Problems with Valgrind

- Though Valgrind cannot attach itself to running programs but it can follow a forked child program from a master program. Trying to check memory errors in LTTD code, Valgrind gave an unhandled *syscall* error as shown in Figure 8.1.

```
WARNING: unhandled syscall: 313
--2089-- You may be able to write your own handler.
--2089-- Read the file README_MISSING_SYSCALL_OR_IOCTL.
--2089-- Nevertheless we consider this a bug.  Please report
--2089-- it at http://valgrind.org/support/bug_reports.html.
```

Figure 8.1: Valgrind Error

- Valgrind eats up a lot of memory and the thus the programs running under it is typically slowed down from 20 to 30 times than the usual run [SN05].

Problems with Acumem

- Acumem was the required and most focused tool for doing Data Flow Analysis, but the only problem with it was the unavailability of the License file. We were using the Evaluation License to test the tool and we found out that Acumem in its evaluation license cannot handle the no. of sample it gets from the running LTTD or USTD program. The error displayed during the test run is shown in Figure 8.2.

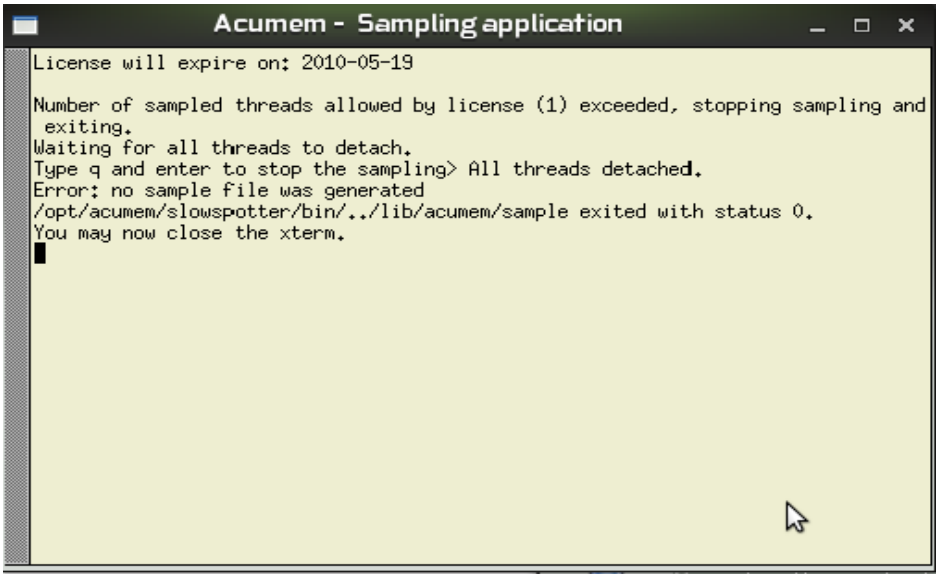


Figure 8.2: Acumem Error

With all the problems faced we decided to do Data Flow Analysis with Valgrind and OProfile. Valgrind was used to check the memory leaks occurring during running of LTT Control during non overwrite and flight recorder tracing. Though it couldn't capture memory leaks of the LTT Daemon spawned by the LTT Control binary but it gave out memory leak errors for the LTT Control binary during execution. Valgrind also could capture the memory leak errors for UST Daemon.

OProfile was used with two events, one of LII_MISSES and another of INSTRUCTION_RETIRED_P to get the details of which part of the code has most no. of L2 Cache misses and branch misprediction.

8.3 Benefits of the Research

Ericsson has the need to deploy a well performing, low overhead tracing tool and thus the results from this above experiments will help Ericsson to think over LTTng as the needed tracing utility across its multicore systems.

During our experimentations we did found few bugs in UST (Userspace Tracer). We reported them to the development team, and they were corrected immediately.

In many a cases we couldn't advance due to various constraints like unavailability of proper tools, licenses or resources. We documented the methodology to do the experiments for the future research community, so that in case all things are proper this thesis report will guide new researchers to further carry on our work with greater precision.

9. Conclusion

The experiments performed in course our research works have produced a variety of results. By analyzing those results we can conclude the following points:

LTTng Kernel Tracer

The impact of LTTng on kernel operations in terms of percentage of CPU cycles against vanilla kernel is 1.6%.

There is almost no difference between the performances of LTTng kernel tracer in Non Overwrite mode and in Flight Recorder mode.

LTTD has almost negligible footprint on both Non Overwrite mode and Flight Recorder mode. We have already seen that both the modes have almost similar amount of impact on the system's performance, but still LTTD takes more CPU cycles in flight recorder mode than in Non Overwrite mode. The difference is almost negligible as it is in the order of 10^{-4} percent.

Memory loss is of very negligible amount but it doubles itself in case of flight recorder mode with respect to Non Overwrite mode during kernel tracing.

LTTng kernel tracer spends most of its time in *libc* and *ld* standard C libraries. It spends only 5.83% of the time in other functions and libraries which includes *liblttd*. Therefore, we can say that as LTTng kernel tracer spends so less time in executing its own functions it has so little impact in the systems performance.

LTT Control and Trace Daemon have minimal Cache miss and Branch Misprediction rate in order of 10^{-4} percent.

Branch Mispredictions of LTTng Kernel Tracer decreases significantly with increase in load. Memory handling thus becomes more efficient with load increase.

Branch Mispredictions in case of different tracing modes vary in case of LTT Control and LTT Daemon. For LTT Daemon branch misprediction rate is high in case of flight recorder mode. LTT Control exhibits no major change with change in tracing modes.

LTTng Userspace Tracer

The LTTng userspace tracer and the compiled markers both have an effect of around 0.50% on the performance of the userspace application in terms of percentage of CPU cycles against the original copy of the applications without markers.

The impact of UST on userspace applications marginally increase with the increase in the number of instrumentations compiled in, though the pattern of increase for all load configurations are not similar.

USTD has almost negligible footprint on the system for different load configurations or different number of markers. But it is noticeable that the footprint of USTD is not as good as compared to the footprint of LTTD. USTD has got a footprint a little higher than LTTD but still is almost negligible to affect the system's performance.

The footprint of USTD decreases as the load increases in the system. Therefore, the performance of USTD gets better with increasing amount of load.

The footprint of UST is liner to the increasing number of markers. Therefore, the number of markers compiled in does not have any effect on the footprint of USTD.

Unlike LTTng kernel tracer, even if LTTng userspace tracer spends greater amount of its execution time in the C libraries, still it spends a lot of time (approximately 13% to 17%) in executing its own functions. Therefore, we can say that the LTTng userspace tracer is not as efficient as the LTTng kernel tracer and there is a scope of improving its performance.

Branch Mispredictions of LTTng Userspace Tracer decreases significantly with increase in load. Memory handling thus becomes more efficient with load increase.

Memory loss though is of insignificant number but is more for UST tracing with respect to kernel tracing. UST also has problem of not freeing a chunk of memory after completion of execution.

A memory leak for UST Daemon happens with a loss of small amount of data during saving trace data to disk.

LTTng Kernel and Userspace Tracer Together

The impact is quite similar to LTTng kernel tracer and there is no additional impact on the percentage of CPU cycles needs to perform kernel operations.

The LTTng Kernel tracer Non Overwrite mode has performed a bit better then the Flight Recorder mode while executed with Userspace Tracer in terms of CPU cycles needed for kernel operations.

LTT kernel Tracing Daemon is much more memory efficient than UST Daemon.

10. Future Work

We were originally set out to do experiment in the P4080 Freescale board with AMP setup for each of the 8 cores. But due to the unavailability of the hardware and setup we ended up doing experiments in a Quad Core SMP Setup. Our experiments methodology can be used to do experiments in the real hardware for evaluating LTTng on a multicore platform.

One part of the research question couldn't be answered in the thesis report which pertained to the use of LTTng agent and Eclipse LTT Tools. This was mainly unaccomplished due to immature build of the LTTng agent and lack of documentation related to its use. There are still few bugs which are reported and yet to be corrected. This all needed much more time and thus was skipped. This work can be carried on after these issues are resolved as it will open new doors to monitor and stream LTTng traces in remote systems.

In some part of the result analysis we found that both the Kernel and Userspace Tracer Daemon performs better with respect to memory handling incase the load increases on the system. While analyzing with Valgrind we also found that the number of time the data gets collected and buffered for high load is very high. So from above two sentences the reason of this may be that the prefetcher already gets to know the branch to be taken, due to large rate of trace data collection, but this requires further analysis which couldn't be done because of lack of time.

Because of lack of proper tools we couldn't dig deep into Data Flow Analysis to check for incorrect data structures or cyclic loop issues. Acumem is suitable tool, provided it's with a full version license. We could only gauge the memory performances of the LTTng and UST tracer with the help of OProfile and Valgrind. Deeper data flow analysis can be taken up as a future work.

We could only limit our studies to the binaries and the running programs of LTTng Kernel Tracer, but the main involvement of LTTng is inside the kernel where its code gets patched. Thus in future, if the Control and Data Flow analysis can be carried out for that part of LTTng then it can give more interesting and useful data to analyze.

GDB (GNU Debugger) has tracepoints to collect trace data which can be analyzed later with help of GDB commands. An interesting future work will be involving this with UST tracepoints and LTTng to see the performance tradeoff.

11. References

[LTT10] **LTTng Project**, http://lttng.org/, Last Updated: 2010-04-20

[SHE99] Sameer Shende, **Profiling and Tracing in Linux**, In Proceedings of the Extreme Linux Workshop 2, Monterey, CA, June 1999 USENIX

[Tam05] Tammy Noergaard, **Embedded Systems Architecture**, Pages (5 – 13), Newnes Publisher, ISBN-13: 978-0750677929, February 2005

[Sér02] Sérgio de Jesus Duarte Dias, **Embedded Systems Architecture,** International Conference on Computer Architecture 2001/02

[NR98] Niemann, Ralf, **Hardware/Software Co-Design for Data Flow Dominated Embedded Systems**, Kluwer Academic Publishers, (1998)

[KP90] Koopman, Philip, **Design Constraints on Embedded Real Time Control Systems**, Systems, Design & Networks Conference, (1990)

[SLES09] Jonas Svennebring, John Logan, Jakob Engblom, Patrik Strömblad, **Embedded Multicore: An Introduction**, Published: 2009-07

[MUC09] Philip Mucci, **Linux Multicore Performance Analysis and Optimization in a Nutshell**, NOTUR 2009

[FRE10] **P4 Series P4080 multicore processor**, Freescale Semiconductor, 2010

[LTP10] Eclipse.org, **Linux Tools Project – LTTng Integration**, Last Updated: 2010-04-20

[LTT00] Karim Yaghmour and Michel R. Dagenais, **The Linux Trace Toolkit**, Linux Journal, Published: May 2000

[DD06] Mathieu Desnoyers and Michel R. Dagenais, **The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux**, Linux Symposium 2006

[DES09] Mathieu Desnoyers, **Low-Impact Operating System Tracing**, Ph.D. dissertation, École Polytechnique de Montréal, Published: December 2009

[FDD09] Pierre-Marc Fournier, Mathieu Desnoyers, Michel R. Dagenais, **Combined Tracing of the Kernel and Applications with LTTng**, Linux Symposium 2009

[MLK06] K. Mohror and K.L. Karavanic, **A Study of Tracing Overhead on a High-Performance Linux Cluster**, Portland State University Computer Science Technical Report number TR-06-06, December 2006

[HDD08] Heidari, P.; Desnoyers, M.; Dagenais, M.; **Performance analysis of virtual machines through tracing**, Electrical and Computer Engineering, 2008, CCECE 2008. Canadian Conference on, vol. no. pp.000261-000266, 4 - 7 May 2008

[FDD09] Fournier, Pierre-Marc; Desnoyers, M.; Dagenais, M.; **Combined Tracing of the Kernel and Applications with LTTng**, In Linux Symposium, Ottawa, Ontario, Canada, July 2010

[DEB10] **Dbench Readme**, http://samba.org/ftp/tridge/dbench/README, Last Visited: 2010-04-20

[BYFS10] **Beyond Linux® From Scratch, Chapter 11. System Utilities**, http://www.linuxfromscratch.org/blfs/view/svn/general/sysstat.html, Last Visited: 2010-04-20

[SYS10] **SYSTAT**, http://pagesperso-orange.fr/sebastien.godard/documentation.html, Last Visited: 2010-04-20

[OPR10] **OProfile**, http://oprofile.sourceforge.net/, Last Visited: 2010-04-20

[PRA03] Prasanna S. Panchamukhi, **Smashing performance with OProfile**, Linux Technology Center, IBM India Software Labs, http://www.ibm.com/developerworks/linux/library/l-oprof.html, Last Updated: 2003-10-16

[PZWSS07] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, Kang G. Shin, **Performance Evaluation of Virtualization Technologies for Server Consolidation**, Enterprise Systems and Software Laboratory, HP Laboratories Palo Alto, Published: 2007-04-11

[GPR10] **gprof2dot**, http://code.google.com/p/jrfonseca/wiki/Gprof2Dot

[LPGF08] Li, Peng; Park, Hyundo; Gao, Debin; Fu, Jianming; **Bridging the Gap between Data-flow and Control-flow Analysis for Anomaly Detection**, 2008 Annual Computer Security Applications Conference, December 8–12, 2008, Anaheim, California, USA

[ACU09] **Acumem SlowSpotter | Optimization Tutorial**, http://www.acumem.com/images/stories/articles/tutorial-slowspotter.pdf, 2009

[VAL10] **Valgrind**, http://valgrind.org/info/tools.html, Last Visited: 2010-05-31

[SN05] Seward, Julian; Nethercote Nicholas; **Using Valgrind to detect undefined value errors with bit-precision**, Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, April 2005

[LYN10] Jaqui Lynch, UNIX and Web PERFORMANCE, Last Visited: 2010-05-31

Appendix A – Experiment Results

11.1 Control Flow Analysis

11.1.1 Experiment 2 – Measuring the efficiency of LTTng Kernel Tracer

			CPU CYCLES			
LOAD	**TEST CASE**	**RUNS**	**LOAD**	**KERNEL OPS**	**OPROFILE**	**LTTD**
LOW	T1	R1	94.140000	4.200900	0.948300	0.000000
		R2	94.089200	4.371600	0.957800	0.000000
		R3	93.765200	4.610700	1.016600	0.000000
LOW	T4	R1	93.896600	4.392600	1.001600	0.000000
		R2	93.730900	4.510600	1.000200	0.000000
		R3	94.081500	4.237300	0.989600	0.000000
LOW	T7	R1	94.062200	4.422400	0.986200	0.000000
		R2	94.170800	4.358800	0.990400	0.000000
		R3	93.957100	4.515100	0.984700	0.000000
LOW	T10	R1	93.851200	4.573400	0.999200	0.000000
		R2	93.965300	4.487700	0.985500	0.000000
		R3	94.131300	4.340200	0.985100	0.000000
LOW	T13	R1	93.514300	4.759500	1.032000	0.000600
		R2	93.786500	4.662500	1.001200	0.000110
		R3	93.353200	4.776100	1.047700	0.000160
LOW	T16	R1	93.962900	4.502200	1.007600	0.000740
		R2	93.741000	4.537100	0.997900	0.000330
		R3	93.142800	5.003500	1.054800	0.000430
MED	T2	R1	96.239700	2.537700	0.865200	0.000000
		R2	96.038300	2.722500	0.870100	0.000000
		R3	96.302400	2.470500	0.865400	0.000000
MED	T5	R1	95.446900	3.227900	0.901400	0.000000
		R2	95.540300	3.139100	0.905100	0.000000
		R3	95.505900	3.217900	0.909800	0.000000

			CPU CYCLES			
LOAD	**TEST CASE**	**RUNS**	**LOAD**	**KERNEL OPS**	**OPROFILE**	**LTTD**
MED	T8	R1	95.499900	3.220500	0.904700	0.000000
		R2	95.604400	3.167000	0.907700	0.000000
		R3	95.598500	3.196500	0.893500	0.000000
MED	T11	R1	95.450500	3.272100	0.917900	0.000000
		R2	95.319200	3.346600	0.937900	0.000000
		R3	95.363200	3.250400	0.936200	0.000000
MED	T14	R1	95.159700	3.448900	0.933900	0.000051
		R2	95.313200	3.342500	0.923900	0.000041
		R3	95.343600	3.381900	0.926800	0.000051
MED	T17	R1	95.137500	3.488300	0.959200	0.000093
		R2	95.234600	3.402000	0.954100	0.000200
		R3	95.098400	3.427700	0.971600	0.000093
HIG	T3	R1	96.712300	2.197100	0.829800	0.000000
		R2	96.697300	2.191400	0.842200	0.000000
		R3	96.754000	2.129200	0.848200	0.000000
HIG	T6	R1	96.270600	2.612600	0.845700	0.000000
		R2	96.395600	2.557300	0.766200	0.000000
		R3	96.328000	2.603800	0.780900	0.000000
HIG	T9	R1	96.329100	2.603300	0.799500	0.000000
		R2	96.244500	2.627800	0.806300	0.000000
		R3	96.246400	2.647300	0.807100	0.000000
HIG	T12	R1	96.157100	2.668800	0.838400	0.000000
		R2	96.368800	2.574900	0.785200	0.000000
		R3	96.399700	2.500500	0.788000	0.000000
HIG	T15	R1	96.136000	2.630100	0.886900	0.000006
		R2	95.957200	2.822500	0.848200	0.000035
		R3	96.120600	2.645900	0.876300	0.000029
HIG	T18	R1	96.112400	2.673400	0.844800	0.000081
		R2	96.230200	2.658000	0.811100	0.000081
		R3	96.032900	2.765600	0.865100	0.000093

Table A1: Experiment 2 Results (Load)

			CPU CYCLES			
LOAD	**TEST CASE**	**RUNS**	**TBENCH**	**KERNEL OPS**	**OPROFILE**	**LTTD**
LOW	T1	R1	11.055100	81.445400	4.339800	0.000000
		R2	11.075800	81.483000	4.332000	0.000000
		R3	11.165000	81.622600	4.246500	0.000000

			CPU CYCLES			
LOAD	**TEST CASE**	**RUNS**	**TBENCH**	**KERNEL OPS**	**OPROFILE**	**LTTD**
LOW	T4	R1	10.123600	81.328400	5.619600	0.000000
		R2	10.013900	81.276600	5.696000	0.000000
		R3	10.059500	81.329100	5.717700	0.000000
LOW	T7	R1	10.015100	81.032100	5.776800	0.000000
		R2	10.108500	81.229200	5.720000	0.000000
		R3	10.143000	81.057800	5.651800	0.000000
LOW	T10	R1	9.143100	80.982700	5.620800	0.000000
		R2	9.141700	81.013800	5.667500	0.000000
		R3	9.155100	80.989700	5.694300	0.000000
LOW	T13	R1	7.206800	84.049300	4.989700	0.000160
		R2	7.304000	83.782900	5.090000	0.000160
		R3	7.041100	84.061200	4.992800	0.000130
LOW	T16	R1	6.496400	83.607200	6.295200	0.000180
		R2	6.548900	84.070500	5.807100	0.000180
		R3	6.577500	83.594300	6.034400	0.000360
MED	T2	R1	12.385100	81.652800	3.551600	0.000000
		R2	12.456400	81.572800	3.507400	0.000000
		R3	12.587100	81.568200	3.481300	0.000000
MED	T5	R1	12.200700	82.388300	3.110700	0.000000
		R2	12.595800	81.838200	3.314900	0.000000
		R3	12.229800	82.035700	3.395100	0.000000
MED	T8	R1	12.232800	81.949600	3.478500	0.000000
		R2	12.700000	81.016800	3.841800	0.000000
		R3	12.164100	81.977400	3.445500	0.000000
MED	T11	R1	10.800500	82.394600	3.448100	0.000000
		R2	10.930800	82.122800	3.534800	0.000000
		R3	10.716500	81.826200	3.969100	0.000000
MED	T14	R1	8.192400	84.960700	3.585200	0.000050
		R2	8.490900	84.943200	3.254200	0.000180
		R3	8.403700	84.916900	3.396500	0.000150
MED	T17	R1	7.558300	85.487200	3.909500	0.000610
		R2	7.920700	84.261100	4.690800	0.000300
		R3	7.554500	85.329700	4.075200	0.000360
HIG	T3	R1	13.017900	82.545100	2.366300	0.000000
		R2	13.013900	82.563900	2.397800	0.000000
		R3	13.032300	82.553000	2.387400	0.000000

			CPU CYCLES			
LOAD	**TEST CASE**	**RUNS**	**TBENCH**	**KERNEL OPS**	**OPROFILE**	**LTTD**
HIG	T6	R1	13.290600	81.447300	3.102900	0.000000
		R2	13.128100	81.315600	3.415900	0.000000
		R3	13.220800	81.296700	3.304900	0.000000
HIG	T9	R1	13.236900	81.380700	3.219900	0.000000
		R2	13.181200	81.178100	3.440700	0.000000
		R3	13.476500	81.341600	3.021500	0.000000
HIG	T12	R1	12.028300	81.636700	3.084700	0.000000
		R2	11.957200	81.845300	2.867600	0.000000
		R3	11.885800	81.710900	3.021800	0.000000
HIG	T15	R1	9.327400	84.284000	3.342000	0.000100
		R2	9.459000	84.448500	3.057700	0.000200
		R3	9.251300	84.606100	3.011500	0.000200
HIG	T18	R1	8.665400	84.515200	3.767800	0.000210
		R2	8.732200	85.074000	3.233600	0.000380
		R3	8.758700	85.348000	2.851100	0.000280

Table A2: Experiment 2 Results (Tbench)

11.1.2 Experiment 3 – Measuring the efficiency of LTTng Userspace Tracer

			CPU CYCLES	
LOAD	**TEST CASE**	**RUNS**	**LOAD**	**USTD**
LOW	T1	R1	93.809000	0.000000
		R2	93.832000	0.000000
		R3	93.362200	0.000000
LOW	T4	R1	94.186900	0.000000
		R2	94.090300	0.000000
		R3	93.715200	0.000000
LOW	T7	R1	94.191900	0.001800
		R2	93.170600	0.001700
		R3	93.713500	0.001500
MED	T2	R1	95.068800	0.000000
		R2	95.962800	0.000000
		R3	95.913000	0.000000

			CPU CYCLES	
LOAD	**TEST CASE**	**RUNS**	**LOAD**	**USTD**
MED	T5	R1	95.910900	0.000000
		R2	95.961100	0.000000
		R3	95.961900	0.000000
MED	T8	R1	95.893200	0.000550
		R2	95.919400	0.001000
		R3	95.921800	0.000680
HIG	T3	R1	96.785000	0.000000
		R2	96.689100	0.000000
		R3	95.919700	0.000000
HIG	T6	R1	96.723500	0.000000
		R2	96.852800	0.000000
		R3	96.806100	0.000000
HIG	T9	R1	96.692300	0.000570
		R2	96.929200	0.000410
		R3	96.768500	0.000590

Table A3: Experiment 3 Results (Load with 1 Marker)

			CPU CYCLES	
LOAD	**TEST CASE**	**RUNS**	**LOAD**	**USTD**
LOW	T1	R1	93.218300	0.000000
		R2	93.625300	0.000000
		R3	93.751800	0.000000
LOW	T4	R1	93.914900	0.000000
		R2	94.200500	0.000000
		R3	94.200500	0.000000
LOW	T7	R1	94.101800	0.002100
		R2	94.286500	0.001900
		R3	93.795000	0.001500
MED	T2	R1	95.216900	0.000000
		R2	95.924900	0.000000
		R3	95.172700	0.000000
MED	T5	R1	96.001100	0.000000
		R2	95.844800	0.000000
		R3	96.139300	0.000000
MED	T8	R1	95.929400	0.000640
		R2	95.861300	0.001100
		R3	95.879300	0.000620

			CPU CYCLES	
LOAD	**TEST CASE**	**RUNS**	**LOAD**	**USTD**
HIG	T3	R1	95.848400	0.000000
		R2	96.688000	0.000000
		R3	96.583000	0.000000
HIG	T6	R1	96.913000	0.000000
		R2	96.867700	0.000000
		R3	95.884900	0.000000
HIG	T9	R1	96.761000	0.000200
		R2	96.933200	0.000300
		R3	96.800300	0.000340

Table A4: Experiment 3 Results (Load with 5 Markers)

			CPU CYCLES	
LOAD	**TEST CASE**	**RUNS**	**LOAD**	**USTD**
LOW	T1	R1	93.325200	0.000000
		R2	93.910800	0.000000
		R3	92.862300	0.000000
LOW	T4	R1	93.871600	0.000000
		R2	93.860200	0.000000
		R3	93.789800	0.000000
LOW	T7	R1	94.220700	0.001900
		R2	93.836500	0.001900
		R3	93.549600	0.001300
MED	T2	R1	95.038800	0.000000
		R2	95.041900	0.000000
		R3	96.068400	0.000000
MED	T5	R1	95.937100	0.000000
		R2	95.767200	0.000000
		R3	95.835500	0.000000
MED	T8	R1	95.885800	0.000880
		R2	95.881800	0.000670
		R3	96.001500	0.000460
HIG	T3	R1	96.611200	0.000000
		R2	96.747000	0.000000
		R3	95.765200	0.000000
HIG	T6	R1	96.777600	0.000000
		R2	96.722200	0.000000
		R3	96.858800	0.000000

			CPU CYCLES	
LOAD	**TEST CASE**	**RUNS**	**LOAD**	**USTD**
HIG	T9	R1	96.981300	0.000390
		R2	96.932700	0.000400
		R3	96.888000	0.000640

Table A5: Experiment 3 Results (Load with 10 Markers)

			CPU CYCLES	
LOAD	**TEST CASE**	**RUNS**	**TBENCH**	**USTD**
LOW	T1	R1	10.971200	0.000000
		R2	10.123200	0.000000
		R3	10.782900	0.000000
LOW	T4	R1	11.317500	0.000000
		R2	11.518100	0.000000
		R3	11.470100	0.000000
LOW	T7	R1	11.498800	0.000270
		R2	11.506100	0.000420
		R3	11.532500	0.000320
MED	T2	R1	12.460300	0.000000
		R2	12.518400	0.000000
		R3	12.383300	0.000000
MED	T5	R1	13.094900	0.000000
		R2	12.929900	0.000000
		R3	13.048000	0.000000
MED	T8	R1	12.976200	0.000210
		R2	12.841100	0.000210
		R3	13.000900	0.000290
HIG	T3	R1	13.080500	0.000000
		R2	13.020700	0.000000
		R3	12.969400	0.000000
HIG	T6	R1	13.538300	0.000000
		R2	13.379300	0.000000
		R3	13.643300	0.000000
HIG	T9	R1	13.714800	0.000140
		R2	13.676500	0.000260
		R3	13.403400	0.000220

Table A6: Experiment 3 Results (Tbench with 1 Marker)

			CPU CYCLES	
LOAD	**TEST CASE**	**RUNS**	**TBENCH**	**USTD**
LOW	T1	R1	10.894900	0.000000
		R2	10.939300	0.000000
		R3	10.982700	0.000000
LOW	T4	R1	11.474700	0.000000
		R2	11.419300	0.000000
		R3	11.406000	0.000000
LOW	T7	R1	11.473900	0.000370
		R2	11.427800	0.000320
		R3	11.562600	0.000290
MED	T2	R1	12.359000	0.000000
		R2	12.394600	0.000000
		R3	12.397200	0.000000
MED	T5	R1	12.898800	0.000000
		R2	12.974100	0.000000
		R3	12.943400	0.000000
MED	T8	R1	12.944600	0.000170
		R2	12.888400	0.000220
		R3	12.935200	0.000160
HIG	T3	R1	13.007500	0.000000
		R2	13.079700	0.000000
		R3	13.053200	0.000000
HIG	T6	R1	13.726400	0.000000
		R2	13.661000	0.000000
		R3	13.630900	0.000000
HIG	T9	R1	13.692300	0.000200
		R2	13.456100	0.000170
		R3	13.597700	0.000190

Table A7: Experiment 3 Results (Tbench with 5 Markers)

			CPU CYCLES	
LOAD	**TEST CASE**	**RUNS**	**TBENCH**	**USTD**
LOW	T1	R1	11.057700	0.000000
		R2	10.965700	0.000000
		R3	10.930500	0.000000
LOW	T4	R1	12.030800	0.000000
		R2	11.536600	0.000000
		R3	11.542100	0.000000

			CPU CYCLES	
LOAD	**TEST CASE**	**RUNS**	**TBENCH**	**USTD**
LOW	T7	R1	11.596600	0.000290
		R2	11.524000	0.000440
		R3	11.552800	0.000290
MED	T2	R1	12.498600	0.000000
		R2	12.499400	0.000000
		R3	12.620600	0.000000
MED	T5	R1	12.945200	0.000000
		R2	13.054700	0.000000
		R3	13.020500	0.000000
MED	T8	R1	12.893100	0.000250
		R2	13.051500	0.000310
		R3	12.944300	0.000280
HIG	T3	R1	12.974100	0.000000
		R2	12.999500	0.000000
		R3	13.004900	0.000000
HIG	T6	R1	13.503800	0.000000
		R2	13.588000	0.000000
		R3	13.636100	0.000000
HIG	T9	R1	13.623300	0.000250
		R2	13.663400	0.000260
		R3	13.573000	0.000180

Table A8: Experiment 3 Results (Tbench with 10 Markers)

11.1.3 Experiment 4 – Measuring the impact on System as well as Traced Application when LTTng Kernel Tracer and Userspace Tracer are executed together

			% CPU CYCLES				
LOAD	**TEST CASE**	**RUNS**	**LOAD**	**KERNEL OPS**	**OPROFILE**	**LTTD**	**USTD**
LOW	T1	R1	93.047600	5.104500	1.043700	0.000082	0.001900
		R2	92.841500	5.202800	1.044500	0.000081	0.001300
		R3	93.249100	4.891700	1.028400	0.000082	0.002000
LOW	T4	R1	93.040900	5.088200	1.060300	0.000410	0.001900
		R2	92.790200	5.260600	1.066600	0.000380	0.001900
		R3	92.655500	5.357900	1.104900	0.000430	0.001800

			% CPU CYCLES				
LOAD	**TEST CASE**	**RUNS**	**LOAD**	**KERNEL OPS**	**OPROFILE**	**LTTD**	**USTD**
MED	T2	R1	95.100200	3.446200	0.951100	0.000072	0.000760
		R2	95.095600	3.498400	0.938000	0.000041	0.000480
		R3	94.920100	3.629600	0.958400	0.000051	0.000600
MED	T5	R1	95.038200	3.511500	0.959700	0.000190	0.000610
		R2	94.996500	3.484600	1.012600	0.000150	0.000830
		R3	95.233800	3.360600	0.945600	0.000082	0.000630
HIG	T3	R1	95.911200	2.829800	0.897400	0.000023	0.000380
		R2	96.018800	2.761500	0.851700	0.000023	0.000350
		R3	95.987000	2.856600	0.780100	0.000035	0.000380
HIG	T6	R1	96.018700	2.776800	0.846700	0.000041	0.000350
		R2	95.878400	2.835800	0.897500	0.000058	0.000380
		R3	96.059000	2.744900	0.799100	0.000087	0.000390

Table A9: Experiment 4 Results (Load with 1 Marker)

			% CPU CYCLES				
LOAD	**TEST CASE**	**RUNS**	**LOAD**	**KERNEL OPS**	**OPROFILE**	**LTTD**	**USTD**
LOW	T1	R1	93.310400	4.990700	1.036800	0.000220	0.002000
		R2	93.121500	5.037600	1.036700	0.000190	0.001700
		R3	93.172900	4.977500	1.044600	0.000110	0.001400
LOW	T4	R1	93.390900	4.844100	0.998400	0.000440	0.001900
		R2	93.089200	5.124800	1.016000	0.000380	0.001800
		R3	93.591000	4.790200	1.005400	0.000380	0.001800
MED	T2	R1	95.141400	3.512200	0.927500	0.000051	0.000650
		R2	95.194400	3.466200	0.962800	0.000051	0.000640
		R3	95.195000	3.402500	0.934700	0.000082	0.000550
MED	T5	R1	95.345900	3.395200	0.906600	0.000130	0.000680
		R2	95.178700	3.391700	0.941000	0.000180	0.000640
		R3	95.140400	3.481500	0.952000	0.000092	0.000560
HIG	T3	R1	96.171600	2.736700	0.749700	0.000040	0.000360
		R2	96.038700	2.780600	0.865000	0.000023	0.000380
		R3	96.118300	2.788300	0.774400	0.000058	0.000390
HIG	T6	R1	96.097100	2.743300	0.839600	0.000098	0.000310
		R2	96.150000	2.750500	0.781500	0.000087	0.000390
		R3	96.081600	2.722500	0.834400	0.000052	0.000360

Table A10: Experiment 4 Results (Load with 5 Markers)

			% CPU CYCLES				
LOAD	**TEST CASE**	**RUNS**	**LOAD**	**KERNEL OPS**	**OPROFILE**	**LTTD**	**USTD**
LOW	T1	R1	93.360400	4.949800	1.079100	0.000140	0.001600
		R2	92.770600	5.323300	1.129500	0.000190	0.001900
		R3	93.278100	4.943400	1.070900	0.000140	0.001600
LOW	T4	R1	92.879900	5.260600	1.111200	0.000330	0.001600
		R2	93.441200	4.923200	1.025100	0.000330	0.002000
		R3	93.276400	5.004100	1.077200	0.000350	0.001800
MED	T2	R1	95.079500	3.473300	1.008800	0.000062	0.000630
		R2	94.992600	3.602800	0.978700	0.000052	0.000650
		R3	95.041100	3.543800	0.971700	0.000072	0.000560
MED	T5	R1	95.072200	3.688700	1.549400	0.000170	0.000720
		R2	95.052200	3.545600	0.975800	0.000160	0.000620
		R3	95.117300	3.529700	0.968400	0.000140	0.000530
HIG	T3	R1	95.882300	2.853200	0.925300	0.000017	0.000360
		R2	95.956100	2.880000	0.821000	0.000029	0.000370
		R3	95.950300	2.871400	0.841200	0.000029	0.000340
HIG	T6	R1	96.103200	2.767800	0.791500	0.000058	0.000430
		R2	96.292900	2.561500	0.823800	0.000087	0.000350
		R3	96.051800	2.737400	0.891300	0.000080	0.000290

Table A11: Experiment 4 Results (Load with 10 Markers)

			% CPU CYCLES				
LOAD	**TEST CASE**	**RUNS**	**TBENCH**	**KERNEL OPS**	**OPROFILE**	**LTTD**	**USTD**
LOW	T1	R1	7.384500	82.585600	6.289600	0.000180	0.000280
		R2	7.390800	82.528900	6.356100	0.000180	0.000230
		R3	7.529100	82.214200	6.251900	0.000130	0.000310
LOW	T4	R1	6.799200	82.772300	6.652300	0.000280	0.000490
		R2	6.813300	82.552700	6.707000	0.000180	0.000360
		R3	6.750300	82.512300	6.736700	0.000390	0.000260
MED	T2	R1	8.799700	83.408500	4.587300	0.000099	0.000270
		R2	8.580600	83.113000	5.090800	0.000099	0.000270
		R3	8.810300	83.664500	4.322400	0.000220	0.000350
MED	T5	R1	7.794800	83.956000	4.773200	0.000200	0.000400
		R2	7.816900	83.840900	4.976500	0.000300	0.000380
		R3	7.910900	83.914400	4.925600	0.000100	0.000330

			% CPU CYCLES				
LOAD	**TEST CASE**	**RUNS**	**TBENCH**	**KERNEL OPS**	**OPROFILE**	**LTTD**	**USTD**
HIG	T3	R1	9.661800	83.688800	3.485400	0.000200	0.000370
		R2	9.765500	83.563300	3.542700	0.000170	0.000250
		R3	9.738200	83.970000	3.033300	0.000099	0.000270
HIG	T6	R1	9.042800	84.644800	3.308800	0.000480	0.000400
		R2	9.129200	84.174200	3.453200	0.000200	0.000450
		R3	8.950600	84.428100	3.492500	0.000250	0.000450

Table A12: Experiment 4 Results (Tbench with 1 Marker)

			% CPU CYCLES				
LOAD	**TEST CASE**	**RUNS**	**TBENCH**	**KERNEL OPS**	**OPROFILE**	**LTTD**	**USTD**
LOW	T1	R1	7.294700	82.827400	6.130100	0.000100	0.000310
		R2	7.434700	82.711500	5.896000	0.000150	0.000330
		R3	7.414700	82.734000	5.641600	0.000210	0.000180
LOW	T4	R1	6.669200	83.335000	6.215800	0.000300	0.000630
		R2	6.781100	83.333200	6.202800	0.000130	0.000290
		R3	6.741900	82.845800	6.076800	0.000260	0.000520
MED	T2	R1	8.648300	84.176300	3.904200	0.000075	0.000170
		R2	8.811200	83.631700	4.235500	0.000220	0.000220
		R3	8.719800	84.076400	4.092600	0.000100	0.000230
MED	T5	R1	7.736000	84.226600	4.734100	0.000380	0.000530
		R2	7.824000	84.612600	4.291000	0.000100	0.000460
		R3	7.781200	84.389800	4.563100	0.000280	0.000560
HIG	T3	R1	9.785600	83.840100	3.316500	0.000130	0.000430
		R2	9.723000	84.042300	3.116700	0.000120	0.000270
		R3	9.823200	83.585400	3.574000	0.000150	0.000250
HIG	T6	R1	8.772400	84.270000	3.471200	0.000280	0.000450
		R2	8.828600	84.141800	3.574400	0.000250	0.000300
		R3	8.848000	84.701400	3.167500	0.000230	0.000350

Table A13: Experiment 4 Results (Tbench with 5 Markers)

			% CPU CYCLES				
LOAD	**TEST CASE**	**RUNS**	**TBENCH**	**KERNEL OPS**	**OPROFILE**	**LTTD**	**USTD**
LOW	T1	R1	7.430400	82.163300	6.122200	0.000130	0.000310
		R2	7.336000	82.694300	5.929100	0.000130	0.000310
		R3	7.471400	83.049900	5.683300	0.000100	0.000130

			% CPU CYCLES				
LOAD	**TEST CASE**	**RUNS**	**TBENCH**	**KERNEL OPS**	**OPROFILE**	**LTTD**	**USTD**
LOW	T4	R1	6.944000	83.030400	6.469500	0.000380	0.000590
		R2	6.878600	83.053000	6.330800	0.000210	0.000390
		R3	6.906000	83.227900	6.196900	0.000360	0.000310
MED	T2	R1	8.698900	83.519800	4.406100	0.000200	0.000420
		R2	8.697800	84.001200	4.056800	0.000120	0.000320
		R3	8.839700	84.081700	3.807900	0.000050	0.000270
MED	T5	R1	8.018900	84.012500	4.547100	0.000510	0.000430
		R2	7.927700	84.585800	4.279900	0.000330	0.000460
		R3	7.974000	85.042800	3.868700	0.000430	0.000540
HIG	T3	R1	9.846100	83.570500	3.469400	0.000170	0.000120
		R2	9.717400	83.543600	3.696100	0.000099	0.000420
		R3	9.830000	83.743900	3.320900	0.000150	0.000400
HIG	T6	R1	8.993500	84.230700	3.553800	0.000150	0.000510
		R2	8.967900	84.515200	3.182700	0.000430	0.000360
		R3	9.116800	84.488600	3.246200	0.000430	0.000450

Table A14: Experiment 4 Results (Tbench with 10 Markers)

11.2 Data Flow Analysis

11.2.1 Experiment 5 – Running load program and tbench on LTTng Kernel with Non overwrite and Flight recorder tracing modes.

Load		LII_MISSES			INST_RETIRED_ANY_P	
		lttctl	lttd		lttctl	lttd
Low	R1	0	0.0105		0.000064	0.000046
	R2	0	0		0.000018	0.000046
	R3	0.01	0		0.000064	0.000028
Medium	R1	0.0093	0		0.000044	0.000038
	R2	0	0		0.000039	0.000033
	R3	0	0		0.000022	0.000022
High	R1	0.0228	0		0.000016	0.000032
	R2	0	0		0.000011	0.000038
	R3	0.0207	0		0.000054	0.000016

Table A15: Experiment 5 Results (Running load program in Non Overwrite Tracing Mode)

Load		LII_MISSES			INST_RETIRED_ANY_P	
		lttctl	lttd		lttctl	lttd
Low	R1	0.000000	0.000000		0.000028	0.000250
	R2	0.008800	0.008800		0.000055	0.000230
	R3	0.000000	0.000000		0.000073	0.000230
Medium	R1	0.000000	0.000000		0.000033	0.000060
	R2	0.000000	0.000000		0.000044	0.000160
	R3	0.000000	0.000000		0.000022	0.000120
High	R1	0.000000	0.000000		0.000011	0.000150
	R2	0.035300	0.000000		0.000006	0.000140
	R3	0.020600	0.000000		0.000016	0.000055

Table A16: Experiment 5 Results (Running load program in Flight Recorder Tracing Mode)

Tbench		LII_MISSES			INST_RETIRED_ANY_P	
		lttctl	lttd		lttctl	lttd
Low	R1	0.000000	0.000000		0.000170	0.000000
	R2	0.000000	0.000088		0.000150	0.000120
	R3	0.000088	0.000000		0.000099	0.000120
Medium	R1	0.000000	0.000000		0.000028	0.000047
	R2	0.000034	0.000000		0.000019	0.000028
	R3	0.000034	0.000034		0.000081	0.000054
High	R1	0.000021	0.000000		0.000040	0.000023
	R2	0.000021	0.000021		0.000011	0.000034
	R3	0.000000	0.000000		0.000022	0.000033

Table A17: Experiment 5 Results (Running tbench program in Non Overwrite Tracing Mode)

Tbench		LII_MISSES			INST_RETIRED_ANY_P	
		lttctl	lttd		lttctl	lttd
Low	R1	0.000000	0.000000		0.000110	0.000440
	R2	0.000084	0.000170		0.000150	0.000420
	R3	0.000085	0.000085		0.000110	0.000130
Medium	R1	0.000033	0.000000		0.000016	0.000230
	R2	0.000000	0.000065		0.000024	0.000200
	R3	0.000033	0.000099		0.000024	0.000200
High	R1	0.000044	0.000000		0.000036	0.000120
	R2	0.000000	0.000000		0.000047	0.000062
	R3	0.000022	0.000022		0.000016	0.000120

Table A18: Experiment 5 Results (Running tbench program in Flight Recorder Tracing Mode)

11.2.2 Experiment 6 – Running UST tracing on load and tbench program each instrumented with 10 markers under different load configurations.

Load with 10 markers		LII_MISSES		INST_RETIRED_ANY_P
		ustd		ustd
Low	R1	0.156300		0.000250
	R2	0.131700		0.000320
	R3	0.140400		0.000330
Medium	R1	0.141900		0.000120
	R2	0.175300		0.000094
	R3	0.071100		0.000120
High	R1	0.127100		0.000054
	R2	0.168600		0.000069
	R3	0.153400		0.000061

Table A19: Experiment 6 Results (Running UST tracing on load program)

Tbench 10 Markers		LII_MISSES		INST_RETIRED_ANY_P
		ustd		ustd
Low	R1	0.000550		0.000180
	R2	0.000110		0.000270
	R3	0.000220		0.000240
Medium	R1	0.000120		0.000130
	R2	0.000058		0.000110
	R3	0.000170		0.000110
High	R1	0.000110		0.000055
	R2	0.000120		0.000055
	R3	0.000057		0.000094

Table A20: Experiment 6 Results (Running UST tracing on tbench program)

11.2.3 Experiment 7 – Running the Kernel tracer with the help of Valgrind under various load configurations generated by load program (system load) and tbench (process and network load).

lttctl							
Tracing Modes	No of Blocks	Bytes of memory	Bytes Lossed due to	Total Memory	Blocks	Memory not freed	Load & Tbench

		Loss		Loss (bytes)		(bytes)	
	17	152	by 0x4051297: lttctl_set_channel _enable (liblttctl.c:472)	152	3	988	Low
			by 0x8049E1F: main (lttctl.c:631)				
Non Overwrite	17	152	by 0x4051297: lttctl_set_channel _enable (liblttctl.c:472)	152	3	988	Medium
			by 0x8049E1F: main (lttctl.c:631)				
	17	152	by 0x4051297: lttctl_set_channel _enable (liblttctl.c:472)	152	3	988	High
			by 0x8049E1F: main (lttctl.c:631)				
	17	152	by 0x4051297: lttctl_set_channel _enable (liblttctl.c:472)				
			by 0x8049E1F: main (lttctl.c:631)	304	3	988	Low
	17	152	by 0x4051117: lttctl_set_channel _overwrite (liblttctl.c:536)				
Flight Recorder			by 0x8049E49: main (lttctl.c:637)				
	17	152	by 0x4051297: lttctl_set_channel _enable (liblttctl.c:472)				
			by 0x8049E1F: main (lttctl.c:631)	304	3	988	Medium
	17	152	by 0x4051117: lttctl_set_channel _overwrite (liblttctl.c:536)				
			by 0x8049E49: main (lttctl.c:637)				

	17	152	by 0x4051297: lttctl_set_channel _enable (liblttctl.c:472)				
			by 0x8049E1F: main (lttctl.c:631)	304	3	988	High
	17	152	by 0x4051117: lttctl_set_channel _overwrite (liblttctl.c:536)				
			by 0x8049E49: main (lttctl.c:637)				

Table A21: Experiment 7 Results (Running Kernel tracer with Valgrind)

11.2.4 Experiment 8 – Running the load and tbench application instrumented with 10 markers under UST (Userspace Tracing) with the help of Valgrind

usttrace							
Low, Medium, High	Clients	No of Blocks	Bytes of Memory loss	Total Memory Loss	No of Blocks	Bytes not freed	Total Bytes not freed
	1	0	0	654	1,146	27,088	
Load					1,334	29,873	
	After load ends	1	654		1,442	31,603	
	ust shutdown - Final				1,267		33,599
	2	0	0	654	1,146	27,088	
					1,334	29,873	
	After load ends	1	654		1,442	31,603	
	ust shutdown - Final				1,267		33,599
	3	0	0	654	1,146	27,088	
					1,334	29,873	
	After load	1	654		1,442	31,603	

	ends						
	ust shutdown - Final				1,267		33,599
	4	0	0	654	1,146	27,088	
					1,334	29,873	
	After load ends	1	654		1,442	31,603	
	ust shutdown - Final				1,267		33,599
Low, Medium, High							
Tbench	10			654	1,149	27,134	
					1,337	29,932	
	Tbench Execution starts						
	Tbench Execution stops						
		1	654		1,445	31,669	
	ust shutdown						
					1,270		33,658

Table A21: Experiment 8 Results (Running Userspace tracer with Valgrind)

Appendix B – Load Program Source

```
/*
 * load.c is a simple CPU load generating program:
 *
 * Copyright (C) 2007  Masanori ITOH <masanori.itoh_at_gmail.com>
 * Modified by (C) 2010  Romik Guha Anjoy <romik.03305@gmail.com>
 * Soumya Kanti Chakraborty <soumyakanti.chakraborty@gmail.com>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * $Id: load.c 114 2007-04-29 11:54:07Z marc $
 */
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

#define PROGNAME "load"
#define VERSION "1.1"

void usage(void)
{
    printf("%s: %s\n", PROGNAME, VERSION);
    printf("    Generate specified CPU load between 1%% to 100%\n");
    printf("    Options:\n");
    printf("      -l LOAD : CPU load 1 to 100 (%) against one physical CPU\n");
    printf("      -s      : Generate CPU load calling getpid(2)\n");
    printf("      -t      : Duration in seconds\n");
}


int main (int argc, char **argv)
{
    pid_t child;
    int i, load_percent = 25, syscall = 0, duration = 10, counter=0;
    char c;
    long start, end, sec;
```

```
start = (long) time(NULL);
while ((c = getopt(argc, argv, "l:t:sh")) != -1) {
   switch(c) {
   case 'l':
       load_percent = atoi(optarg);
       break;

   case 's':
       syscall = 1;
       break;

   case 't':
       duration = atoi(optarg);
       break;

   case 'h':
       usage();
       exit(0);
       break;

   default:
       printf("Unkown option '%c'\n", c);
   }

}

if ((load_percent <= 0) || (load_percent > 100)) {
   printf("Invalid CPU load: %d %\n");
   exit(1);
}

printf("generating CPU load : %d %\n", load_percent);
printf("running for %d seconds:\n", duration);

if ((child = fork()) > 0) {
   /*
    * parent process
    */
   //printf("%d\n",duration);
   while (1) {
       kill(child, SIGSTOP);
       usleep((100 - load_percent) * 1000);
       kill(child, SIGCONT);
       usleep(load_percent * 1000);
       end = (long) time(NULL);
       sec = (end - start);
       counter++;
       if(sec >= duration)
          break;
   }
   /* here at the end */
   printf("sending SIGTERM to the child\n");
   kill(child, SIGTERM);
   printf("loops executed: %d\n", counter);

} else if (child == 0) {
   /*
    * child process
    */
   while (1) {
       if (syscall) {
          getpid();
```

```
            }
        }

    } else {
        printf("fork failed\n");
        exit(0);
    }

    return 0;
}
```