Tools for workload characterization

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Ashish Kumar Gupta and Parul Sangwan
(111801052 and 111801053)

COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

*This is to certify that the work contained in the project entitled "**Tools for workload characterization** " is a bonafide work of **Ashish Kumar Gupta and Parul Sangwan** (**Roll No. 111801052 and 111801053**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

Dr. Sandeep Chandran

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Acknowledgements

A project is bridge between theoretical and practical working. With a will to learn more and work more, we took this project. Firstly, we would like to take a moment to thank our mentor Dr. Sandeep Chandran for his sustained efforts in guiding us on every step of the project. From providing the motivation behind the problem statement to explaining the necessity of each sub-problem and work-around, he has always been there for us. We are highly obliged in taking the opportunity to thank the department of Computer Science, IIT Palakkad and Dr. Unnikrishnan(BTP coordinator) for giving us an opportunity to work on this project. We have no valuable words to express our gratefulness, but our heart is still full of the favours received from every person.

We would also like to thank our parents and friends who supported us during these tough Covid Times, and helped us to keep up with the curriculum.

Contents

‘

List of Figures

Chapter 1

Introduction

There are many softwares that run on our system, for instance the Zoom Client, Text Editors, Web Browsers etc. All of these applications are called workloads which are the inputs received by the system at a given time. These workloads put various demands on the resources of the system they are being executed on and utilizes them to accomplish their tasks. These demands put by the workloads on system resources are called workload characteristics. Gathering and examining these workload characteristics help us to understand the workload and hence design better systems in terms of compilers, Operating Systems, Malware Detection etc. These characteristics help in designing modern computer systems by influencing the changes required at various system levels - from hardware to operating systems. The advancement in processing power has introduced new kinds of workloads and hence new demands. The way a workload executes illuminates how well resource utilization is done which can in turn be used to do performance optimizations at both hardware and software level. Similarly, knowing runtime characteristics of an application let the developers know how well that application is using the modern standard processing power and they can figure out the reason of lower performance.

Different characterization tools like the Linux Perf Utility [3], Pintool etc. are built to mon-

itor these applications statistics like the cpu usage, instructions per cycles (IPC) executed, memory usage etc. and emit instruction traces. Modern workloads that run on CPUs are considerably different as compared to previous times. Now-a-days, CPUs are able to support cloud infrastructures, run no-sql databases, perform big-data processing and run web-servers. Hence, the type of workloads today are considerably different than earlier workloads, which used to operate in user-mode most of the times, and hence characterizing such applications is comparatively a difficult task because many of these modern workloads are predominantly system-call intensive, that is, they operate on kernel-level most of the time. These characterisation tools like Pintool operate at the user-level and hence are not able to provide valuable insights when the application is operating in kernel mode.

While it is helpful to know the complete application execution patterns, it might sometimes take very long time. Hence, determining shorter application execution phases which are representative of the whole application run, helps to analyze the application and its characteristics better and is time-efficient. Tools like Pintool emit the instruction traces but these traces are at the application level and yield no information about what happens after a system call and thus is not suitable for system-intensive I/O bound modern applications which mostly operate in kernel space. Moreover, using the pre-existing tools requires manual intervention and hence are tedious to use. This whole analysis of identifying the workload phases, getting the workload's instruction traces and then analyzing this huge dump of assembly is done manually in industry. In some places where they use automated tools, their way of generating instruction traces takes way too long and this makes the process tiresome and lengthy.

Workload Characterization even though it seems a deterministic problem because of the involvement of a lot of statistical techniques is a non-deterministic one. The statistical techniques involve large amount of data and large no. of parameters interacting with each other which lead to probabilistic approximations. Hence, workload characterization isn't a easy task.

We aim to build an end to end workflow, which would find out the application phases automatically. After phase identification is done, all the user needs to do is input the interval required to trace in Qemu, and he gets very precise instruction traces to work with.

1.1 Problem Description

The project aims to build tool which is able to collect valuable insights about the program. The tool will not only characterize the application code like already available tools but will also monitor the system activity by being able to trace system level instructions as well with minimal or zero user intervention.

1.2 Methodology

1.2.1 Phase Detection

This journey into workload characterization started by learning about various Linux Utilities. This was step 0 for the project, though it is not explicitly mentioned in the Fig. 1.1. Initial exploration and tool development is done using the cpu2006 [1] Performance-Evaluation Benchmark, because it is well-studied as well as well-documented. The Linux Perf utility was used for plotting statistics like the IPC, cycles per second, instruction count vs time plots. The tool developed, would then automatic do plotting and data generation required for the clustering algorithm. This data would then be fed to the clustering algorithm - DBSCAN. This algorithm would then cluster the phases to determine various application behavioural phases. These tools created to perform perf analysis and Clustering, would then be used on the Guest Operating system running on Qemu-hypervisor in order to emit traces corresponding to both the Kernel Level code and the user-level code. Pintool can't perform the same because it is incapable of gathering information when the system is running instructions in kernel mode. Qemu, which is a hypervisor,

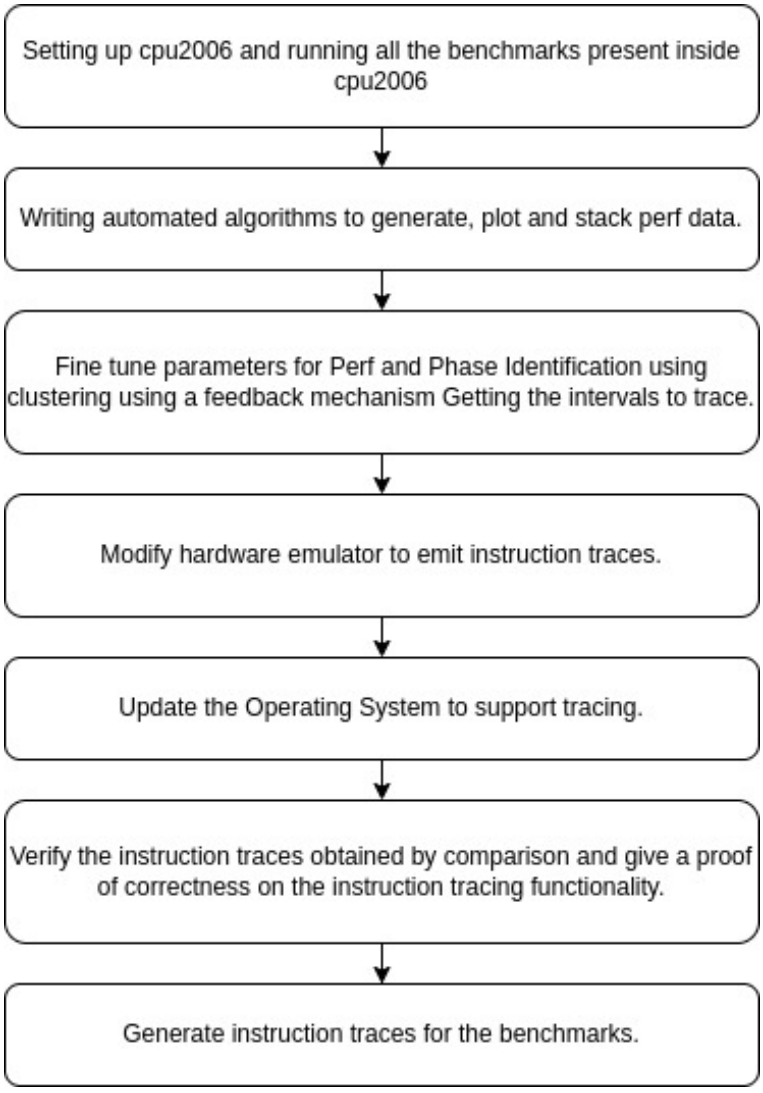


Fig. 1.1: Workflow of the Project.

sitting on top of the actual hardware is capable of emitting instruction traces by changes to its open-source code.

1.2.2 Instruction Tracing

Instruction traces should be found out just before the instructions are being sent for translation. The dynamically obtained set of instructions that are going to be fed to the vCPU(virtual CPU for the guest OS), are the ones the are required.

The overall approach was that the user would give `-XOXO` argument while running

workload/program. The Operating System would identify this input argument, and would add one of `int 100` or `nops`, based on the plugin they are using, dynamically in the assembly. In case it is `int 100`, qemu source code would call a shell script which in turn would send qemu a signal to start tracing. The plugin would then start recording the instruction count from the start of the application. The user would have already run Qemu with the interval as an command line argument. The plugin would skip logging the instructions outside the interval and would log the ones that lie in the interval input by the user. This would give very precise instruction traces to the user. While in the case of nops, the plugin would keep counting the number of consecutive `nop` instructions that appearing in the assembly of the application and start the tracing mechanism as soon as 997 'nops' are encountered.

The first step to achieving this was to write a plugin, that would be able to read and disassemble the set of instructions that are to be executed on the vCPU. This dump of instructions would yield both user and kernel level instructions. This plugin should take input the interval to trace as input. Hence logic was developed to start and end tracing based on input interval to trace. Once the plugin is built, its verification was a major task involving the count of instructions obtained, the order of instructions, loop unrolling, checking for compiler optimizations, etc.

Firstly, we did the changes in xv6 to get an idea of how the operating system interrupt handling and system call work. We modified `exec` system call to look for the argument `-XOXO` and the later remove this argument from the process's stack to avoid it crashing at any later point in time. It also adds the assembly instruction whose only task is to initiate the tracing in the plugin that we built. We then verified that the count of the instructions being skipped is as expected along with re-investigation of the qemu traces obtained. This way we made the tracing application-independent.

The aim in phase II of BTP was prepare an end to end workflow starting from phase identification to emitting traces in that particular representative interval. We wanted to

build a mechanism where there is very less intervention from user and he can get the application characterization in just 2 simple steps with no intermediate hassle, that is, get interval of workload to trace and get traces directly in the interval. We aim to obtain this complete workflow for behavioral phase detection and instrument phases of an application. Since the project is research based, we had to learn and experiment a lot of tracing mechanism, three large open-source softwares. We were able to develop this end to end workflow, and an hassle free workload characterization tool.

1.3 Organization of The Report

Chapter 1 provides a background and a gentle introduction to the topics covered in this report. We have discussed the motivation, need for this project and the methodology followed in the project. It builds up for the detailed approaches explained in the upcoming chapters.

Chapter 2 tries to describe the efforts and work done in this field and how our goals align/differ from previous efforts.

Chapter 3 describes in brief the work done in BTP Phase-I. It provides background about the Perf Utility, CPU2006 benchmark. It discusses perf data generation, automation algorithm for perf data generation and storage of this generated data. Later, it talks about noise reduction in the data.

Chapter 4, talks about identifying phases from the data that is generated in Chapter 3. It discusses the significance of obtaining phases from the application along the method used for phase detection. Furthermore, it discuss the smoothing of these obtained phases and extracting intervals to trace from these phases. Chapter 3 and 4 summarize the work done in BTP Phase 1, while latter chapters present BTP Phase 2 work.

Chapter 5 answers the what, why and how to use Qemu for our project. It elaborates on the plugin that we developed and two approaches that we used to build that plugin. Later, it demonstrates the verification process for the instruction traces obtained from the plugin.

Later part of it builds up for the next chapter on how we aim to modify the Operating System to inject the markers for tracing initiation.

Chapter 6 discusses our approach of modification of the OS. It discusses this modification first on a small Operating System xv6 and then to the widely used Linux Kernel. It gives the proof of correctness of the approach and discusses the extra instruction problem that we encountered while performing the change. Later, it dives into Linux source code and making changes in it along with verification of the traces obtained.

Finally in chapter 7, we conclude by giving an insight on what we currently are working on and what will be our next steps moving forward towards building a full-fledged Workload Characterization Tool.

Chapter 2

Review of Prior Works

Our approach to develop a tool for Workload Characterization can be divided into two major parts - Characterization of application behaviour by runtime phase analysis and instruction tracing. Most of the Workload Characterization tools which are used in industry only characterize the user-code in application, and not the system-level activity. Also, most of such tools require a lot of manual intervention and are generally tiresome to use.

2.1 Previous BTP

One of the previous BTPs done by our senior was to identify phases in the application using clustering algorithm. The process, however, was carried out manually for each benchmark. The issue with that was, in order to determine phases in the program's run, they considered too many hardware events like the cycles, l1d miss.pending, uops executed.thread, cpu clk unhalted.thread, etc. Considering multiple events, in turn, led to amplification of the noise generated and produced results which were not reliable.

2.2 Qemu Trace

There is also some work done for getting instruction traces from the application running on the Guest OS using Qemu. One of them is QemuTrace, research done by Dr. Rajshekhar, IIT Delhi. We referred to his research. He used two approaches to generate instruction traces - an interrupt to start tracing, while the other inserted a particular assembly instruction in application's assembly code to mark the beginning of tracing. The former approach requires an interrupt to be send from the host machine to Qemu while the latter requires the source code access of the application. He worked with an older version of Qemu, after which Qemu source code went through some major changes. The problem with the above mentioned approaches is that in the first one, the user won't be able to exactly start tracing when the application starts to execute. While in the second one, user needs access to source code of the application which he doesn't always have access to. In contrast to the approaches stated above, we would modify the OS itself, so that every time the target application is starts to execute, user doesn't have to make changes to the application's assembly code. Moreover, we are using plugins for tracing instead of doing lot of modifications on Qemu's source code which makes our approach version independent.

2.3 Other works

There are some other closely related works in Workload Characterization and its usage. Some of them are:

1. A structured approach to the simulation, analysis and characterization of smartphone applications [4] had a similar motivation like us. It involved characterizing mobile application phases. Their approach is similar to the methodology we stated, however their target was mobile-based applications and their generation of SimPoints for phase detection is different from ours.

2. Energy-Efficient Cluster Computing via Accurate Workload Characterization [5] uses

accurate Workload Characterization to dynamically adjust processor's frequency and voltage to reduce energy and power consumption without affecting application performance by a significant amount.

2.4 Conclusion

This chapter provides details of some of the prior work in workload characterization and the efforts done as a part of the previous BTP.

It also gives an insight on how QemuTrace can be used to generate user as well as system level instruction traces and how that approach is different from our approach, where we try to eliminate the drawbacks of their approaches.

Chapter 3

CPU Performance Monitoring using Perf

3.1 SPEC CPU-2006 [1]

The SPEC CPU-2006 is compute intensive benchmark suite, which mainly stresses computer processor, memory architecture and compiler. It provides two different types of compute intensive suites namely, CINT2006 and CFP2006 which focuses on integer and floating point performance respectively. SPEC CPU-2006 provides 31 benchmarks - all of which contain multiple tests that can be run per benchmark. One of our primary reasons to use SPEC CPU-2006 in order to start the development of the tool, despite it being CPU-intensive, is that it is a well studied and well documented benchmark. Hence, we development the tool using this benchmark. This tool can later be fine-tuned for other benchmarks and applications as well.

3.2 Perf-events data generation

Perf (Performance Counters for Linux) is a Linux-utility which is used as a performance monitoring tool. It provides a number of different options (sub-commands) and is able to

perform a lightweight statistical profiling of the entire system in both kernel and user level. Here, we use the stat sub-command of perf utility. It provides total event count for a single program or for a system in a given interval.

So, we used perf for performance data generation. Currently for this analysis, we only used IPC, in contrast to techniques used in previous attempts described in chapter 2. The reason for this is that, it conveys the performance of the processor and load on it better than other statistics available. Though the IPC of the system depends on various things like the processors in the system, its memory hierarchy, and the type of workload running on it etc., the pattern/plot obtained of IPC with multiple systems shows similar patterns. Hence IPC may not be enough to convey the bigger picture but was a good starting point indeed. Other statistics, however, were very system dependent.

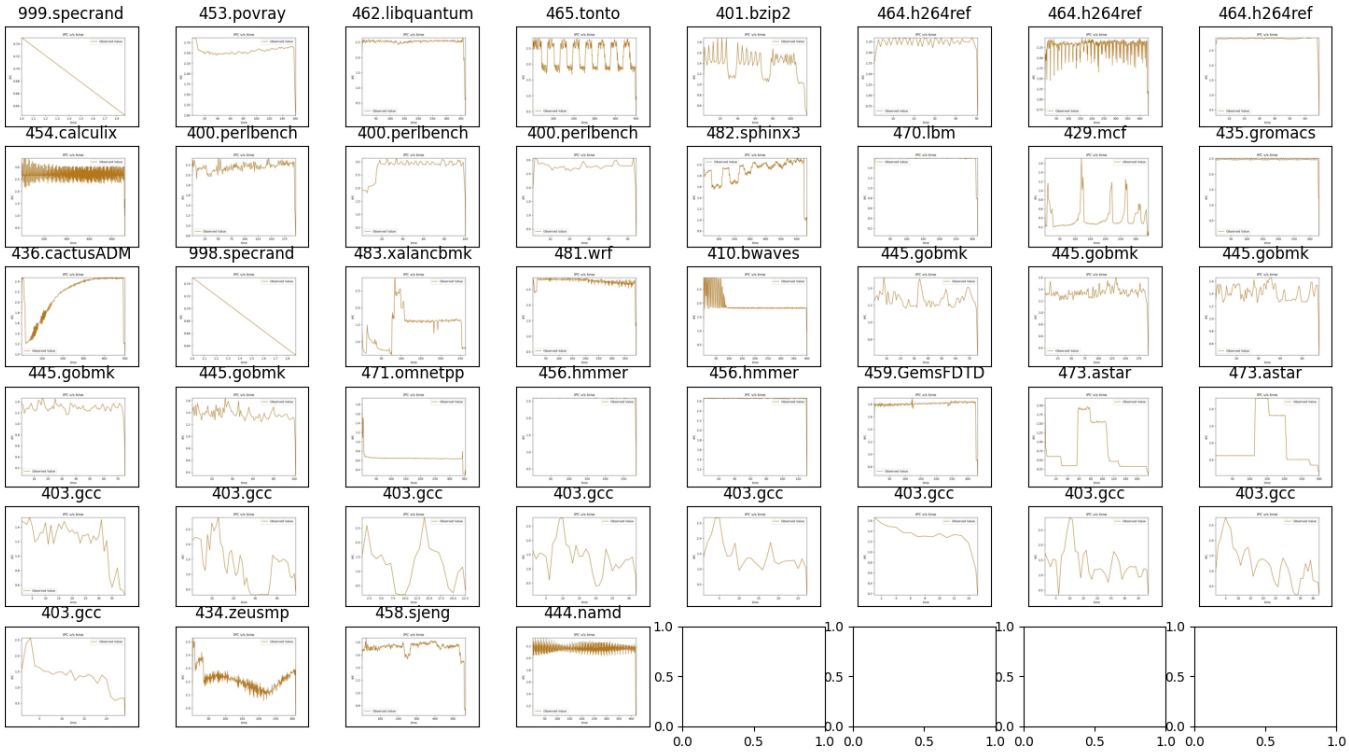


Fig. 3.1: Stacked plots for all the benchmarks. Refer Section 7.2 { Github }

The data yielded by perf was in a raw format, out of which some fields were useful for analysis, while some were not. Moreover, it was also required to run this tool multiple times over all the benchmarks while each benchmark had multiple inputs. This whole process was repeated again and again in the process of tool creation. Hence, we automated the whole process giving user a flexibility to run single or group or all benchmarks under any number of specified events under observation. This automation is done using python scripting (refer to Appendix-A 8.1) and it automatically stores all the data generated in form of csv files and plots a graphical representation of their run. Fig.3.1 shows how the graphs obtained for all of the 44 benchmarks look like.

The system was completely cut off from external disturbances by turning off WiFi, shifting the system to Aeroplane mode and stopping all other user processes running on the system so as to get the best possible graphs. We tried to provide each benchmark an almost isolated system for run. This made sure that there were no spurious values in the performance counter data which could otherwise occur due to system disturbances.

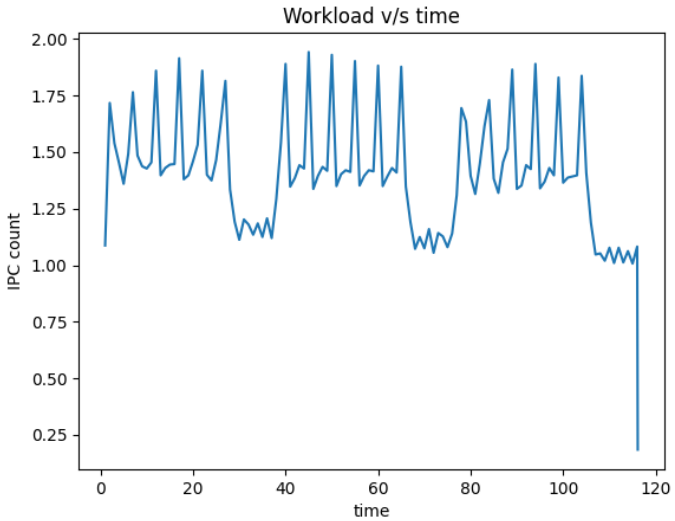


Fig. 3.2: Time varying graph of IPC(Instructions per cycle) for bzip2 reference benchmark. The executable run with input - input.source 280.

3.3 Noise Removal

The data generated by perf command was dependent on multiple factors, for instance other applications running on the system, the hardware being used, context switches in the kernel etc. These factors induced noise in the data generated by perf command. Hence, to counter this noise, average over multiple runs for the same benchmark was taken with respect to time.

The graphs obtained this way were comparatively noise-free and hence more reliable to be given as input to the phase-detection algorithm. In order to make the data even more reliable, the dataset prepared by using perf command was averaged over multiple runs, and the averaged values were taken for dataset creation.

3.4 Conclusion

In this chapter, we analyzed the cpu2006 benchmark and automated the setting up and running the benchmarks, averaging and plotting the graphs for these benchmarks. We illustrated a way of reducing noise in data by averaging over multiple runs.

Chapter 4

Phase Detection and Interval Estimation

4.1 Phases in Application

Phase Detection refers to finding portions of the application that exhibit similar characterization statistics over time. In some applications finding out these repetitive phases is helpful as in [6], to understand the nature of the application without actually having to run and analyze the application for a very long time. Analysis of behaviour of the application in these phases is enough to predict the behaviour of the complete run of application.
IPC vs time graphs were obtained for different benchmarks using perf-events data. From the IPC values obtained during the execution, different phases of the application can be distinguished. To detect the phases, advanced clustering algorithms are required for the following two reasons. Firstly, the clustering algorithm should give phases which are continuous in time. This means that the algorithm should not treat the data as a random set of points with no time-series information associated with them. Secondly, each phase should have some considerable amount of data points to be called a phase, else it should just be treated as noise for instance when the application ends. DBSCAN - Density Based Spacial

Clustering with Noise (refer Appendix-A Section 8.2) fulfills both of these requirements.

4.2 Phase Detection using DBSCAN

DBSCAN based Clustering was applied on IPC values because it serves as a good metric for a starting analysis to describe how the application is performing.

The data generated using perf-events is used in clustering. For the IPC-values obtained, 1-D clustering was performed using the DBSCAN algorithm. The algorithm took as input 2 hyper-parameters, d which is minimum distance to consider a point as a neighbour of another and n which denotes the minimum number of neighbors for a point to be considered as a part of cluster. The value of d chosen is 0.05 and the value of n is 6. These values worked well for cpu-2006 benchmark. Fig. 4.1, 4.2, 4.3 are some of the results obtained on performing clustering with this algorithm.

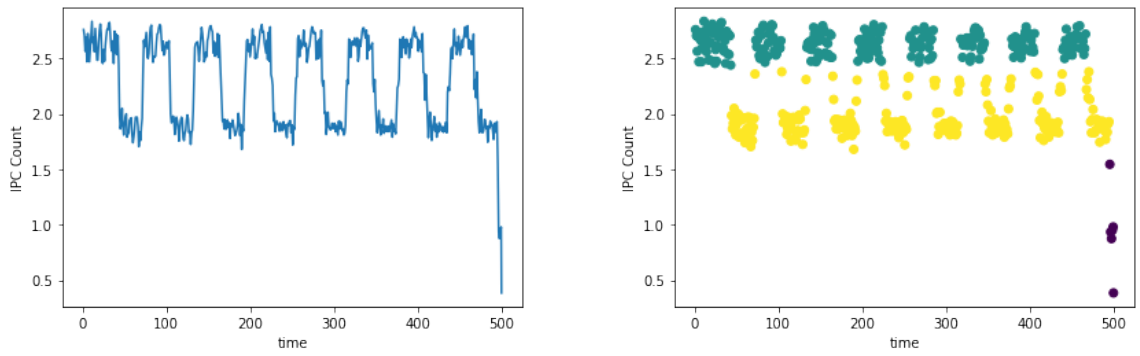


Fig. 4.1: IPC values and obtained Clusters for 465.tonto benchmark

In Fig. 4.1, the benchmark has 2 main phases, the crests and the troughs. The algorithm is able to identify clearly the two phases- sea-green region and yellow region. Along with this, it labels the outliers with purple near the end of the graph, when the benchmark completes execution and exits. These noise(purple points) in this case are labelled as -1 by the algorithm. In Fig. 4.2, the execution of benchmark is divided into 5 main phases represented by different colours, and values at the point of transition from one phase to another are labelled as noise. In Fig. 4.3, there are no clear phases in the application and

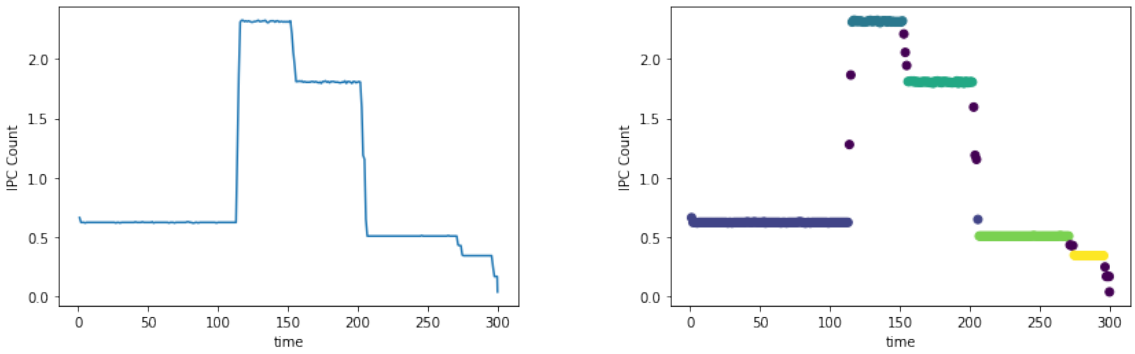


Fig. 4.2: IPC values and obtained Clusters for 473.astar(input:reverse.cfg)

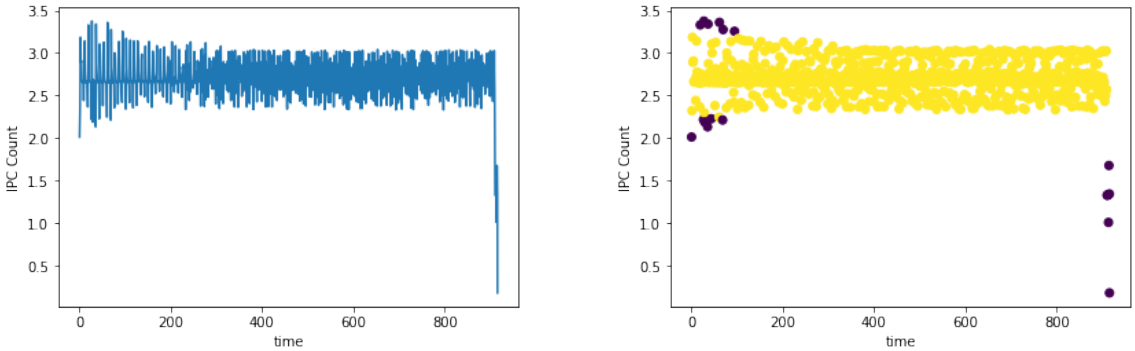


Fig. 4.3: IPC values and obtained Clusters for 454.calculix

so it should be labelled as a single phase of the application. DBSCAN correctly identifies this as a single phase.

Before DBSCAN, other algorithms and methods were tried but none of them worked in the desired way. K-Means Clustering (ignored the continuity in the time series data) and LSTM (Longest Short Term Memory), were also tried out to detect phases.

4.3 Extracting Phase-Representative Data

After phases have been identified, more information about the behaviour of the application can be extracted from them. This includes the number of times a particular phase occurs, the average IPC value of that phase, count of instructions being executed in that phase, onset and retreat of a phase etc. These intrinsic details are required in order to get an instruction interval which represents that phase.

4.3.1 Label Smoothing

The IPC values obtained from perf around the end of execution yielded values which didn't match the trend followed by the rest of the values. This is because of our noise-removal technique. Our noise reduction approach involved taking average of IPC values at each timestamp for multiple runs. However, each run might take slightly different amount of time because of some unavoidable system induced noises in the run and hence slight dips in the values can be seen in the Fig. 4.4 towards the end of execution. These are the values which got averaged.

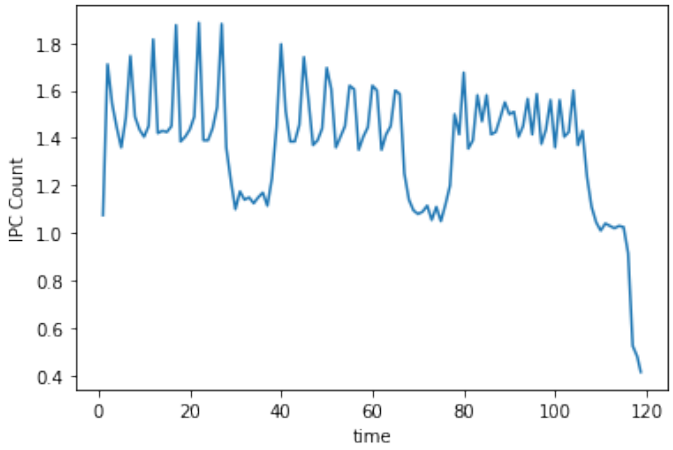


Fig. 4.4: Time varying graph of IPC for bzip2 reference benchmark averaged over 2 runs.

In some benchmarks, there were some spurious values occurring in between the phases which our clustering algorithm labelled as noise. In reality, since these spurious values occurred periodically and in between a phase it can neither be regarded as a phase nor noise, hence smoothing was done for labels to reduce this effect.

Smoothing technique which we used, re-classified such spurious values to their nearby cluster. This was done only when such a outlier was surrounded completely by a single phase, and that outlier was not dense in that particular region. The algorithm takes up a window of IPC values and finds the mode of the cluster identifier, say m. If the no. of occurrences of this m is higher than some threshold value, (in the tool, 90% was used),

then all the IPC values in the window are put in the cluster m.

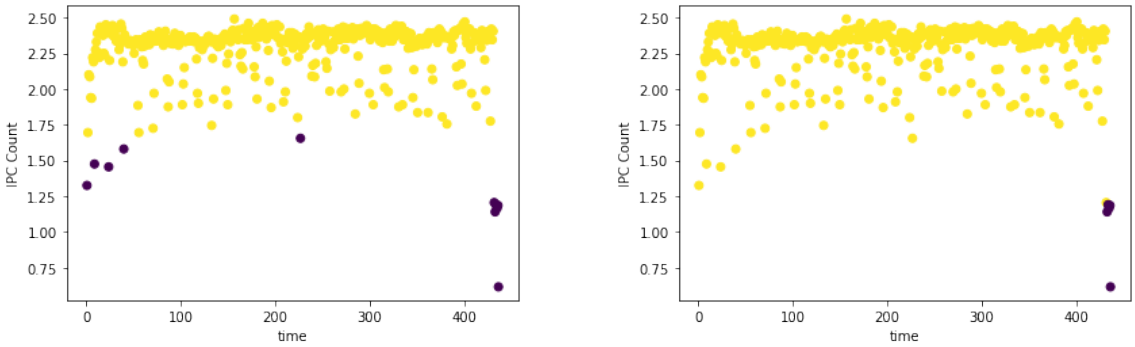


Fig. 4.5: DBSCAN Clusters and corresponding smoothed clusters for 464.h264ref

Benchmarks in Fig. 4.5 and Fig. 4.6, show improvement after smoothing operation. However, this approach might introduce some ambiguities when there is sharp change in phases as in Fig. 4.7, 465.tonto benchmark. These ambiguities do not produce a pronounced effect, since the tool we aim to develop would be relying mostly on average values. But on the other hand, not classifying these periodically occurring spurious values in a cluster and treating them as noise might give improper cluster averages as stated in the next subsection. Hence, smoothing the label values is an important step in our use-case.

4.3.2 Phase Information

To better analyse each phase occurring during the execution of an application, it was important to extract information like the place of occurrence, number of instructions being executed in each phase, number of instructions executed till its occurrence etc. It should also contain some metadata like duration of phase in seconds and the timestamp at which it started. In Fig. 4.8, sample extracted phase information can be seen, The information is corresponding to the yellow phase. Similarly, information for the other phase is stored in form of python dictionary. This process of identification of cluster metadata was also automated with the help of a python script.

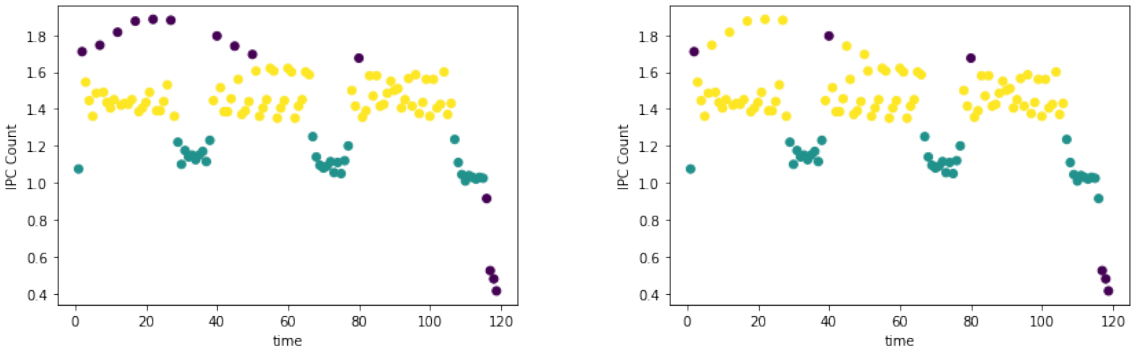


Fig. 4.6: DBSCAN Clusters and corresponding smoothed clusters for 401.bzip2

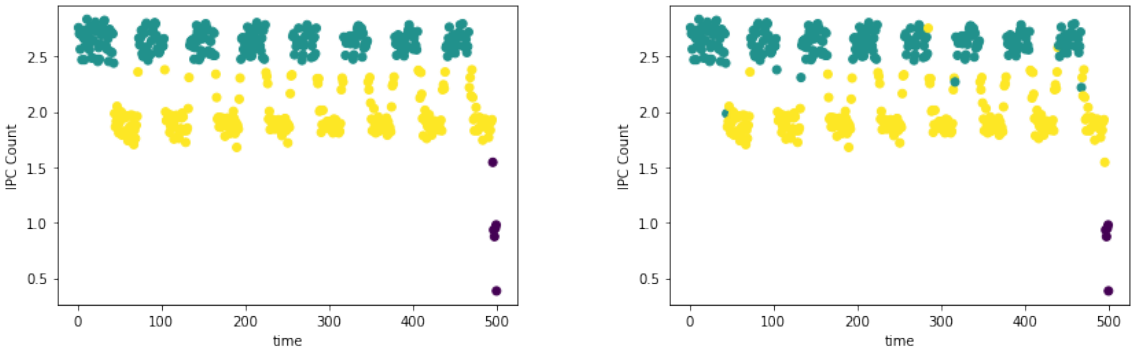
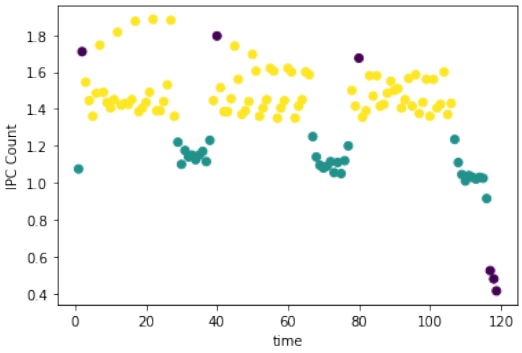


Fig. 4.7: DBSCAN Clusters and corresponding smoothed clusters for 465.tonto




```
1: [{'inst_in_phase': 98014922545.5,
     'inst_till_now': 6885240617.0,
     'phase_duration': 25.003674922000005,
     'start_of_phase': 3.000460453},
    {'inst_in_phase': 104784769402.0,
     'inst_till_now': 133770892089.0,
     'phase_duration': 27.003831815999987,
     'start_of_phase': 39.005740993},
    {'inst_in_phase': 106666487242.0,
     'inst_till_now': 269233717680.5,
     'phase_duration': 28.00407995700003,
     'start_of_phase': 78.01126087299998}]}
```

Fig. 4.8: The extracted data for first phase: Identified by yellow - 401.bzip2

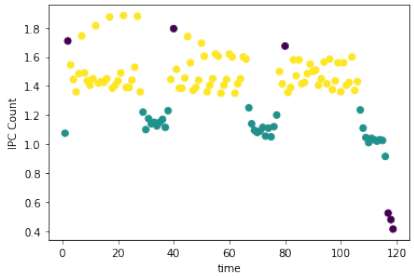
4.3.3 Cluster-Wise and Weighted Average

For each of the clusters obtained (IPC values for a phase), an average IPC value was found. The instructions which are executed around this average are the points that will be given as input to Qemu for emitting traces. A weighted average for each of the clusters was also taken in order to know what was the average IPC for the whole execution of the application. Note that this average is different from the simple average of raw values since here average is taken for points belonging to some cluster. This means the noise is not included in the average calculation. Weights here are the no. of occurrences of the cluster eqn. (4.1). This will be used for comparison of predicted IPC with the help of Qemu.

$$w_1.avg_1 + w_2.avg_2 + w_3.avg_3 + ...w_n.avg_n = avg_{weighted} \tag{4.1}$$

4.3.4 Instructions to Trace

From the average value for each cluster, the point in the data whose IPC value is closest to this average value is found. Using this, one can easily find the timestamps at which this value occurs. For simplicity, the first occurrence (lowest timestamp) of this value is taken. For this timestamp, the no. of instructions executed in that second subsequent to that timestamp and the instructions executed till that timestamp are recorded and saved. This value will be fed to modified Qemu Fig. 4.9 to emit instruction traces.




```
{0: {'inst_this_second': 2774332053.5000005,
     'inst_till_now': 258054322159.0,
     'ipc': 1.1099999999999999,
     'timestamp': 74.010705326},
 1: {'inst_this_second': 3702055858.5,
     'inst_till_now': 25754992466.5,
     'ipc': 1.49,
     'timestamp': 8.001215624}}
```

Fig. 4.9: Cluster-wise Input for Qemu - 401.bzip2

4.4 Conclusion

This chapter picks up the task after perf data generation. It introduces the concept of phases in application execution and a way to identify those phases using density based clustering algorithm, DBSCAN. Then that cluster is fine-tuned to extract out the other useful information like instruction count. The chapter illustrates the methods used for fine tuning and techniques used for information extraction.

Chapter 5

Instruction Trace Emission using Qemu

5.1 Introduction

In previous chapter, using the Clustering Algorithm, we had identified the interval of instructions to analyze. This identified interval gives the instruction number from the start of the application execution to start the tracing from. Using this interval, we need to find the assembly instructions that are executed in this particular interval. Note that the assembly dump that we would get from tracing should contain both the kernel-level and the user-level assembly.

Qemu Plugins are an efficient and developer friendly way to do this. They allow us to extract useful information while Qemu execution without actually altering the source code of Qemu. Hence, we created a plugin namely, Tracelog, to dump the assembly of both kernel level and user level instructions along way its execution.

This former part of chapter develops the basic background required and explains the approach that we followed. The latter part explains the execution logic, working and verification of the Tracelog plugin.

5.2 Background

5.2.1 Qemu

Qemu is an open source hardware emulator and virtualization software. It lets the user run a different Operating System called Guest OS on his own machine, called the host. Through Qemu, we can also emulate the hardware of our choice.

Qemu virtualizes hardware for the user by creating virtual CPUs(vCPU), RAM etc. The instructions from the guest OS are fed to the vCPU created by Qemu. These cannot be directly fed to Host because the host and the guest might be using different instruction sets. Refer Fig. 5.1.

This translation is dynamic, which means conversion happens as and when needed. TCG maintains an internal cache called Translation buffer which helps make the translation process fast by retrieving the recently translated blocks of code, in case the block appears again while its translated code block is still in the cache.

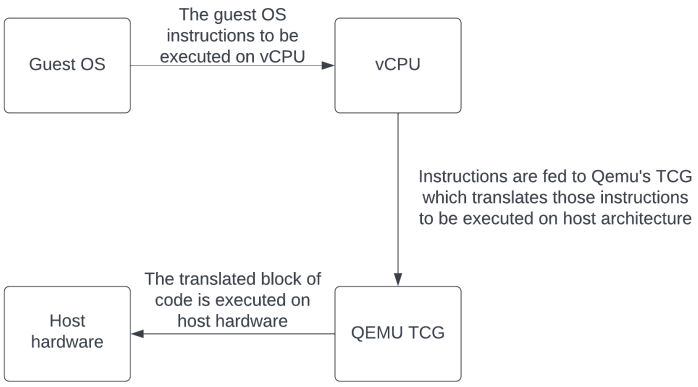


Fig. 5.1: Brief working of Qemu

Why use Qemu?

Primary reasons for choosing QEMU over other emulators are that it is open source, fast and debuggable which gives it an edge over other emulators. Being open source Qemu Refer

GitHub, one can modify the code. Also, one can't get all instruction traces directly from the source since no user-level program have access to kernel level code. Hence, using an emulator like Qemu, we can get a sneak peek into both kernel and application level instructions during the translation process itself. Moreover, Qemu supports external TCG Plugins [7]. Above stated reasons and the vast usage of Qemu across the developer community for such purposes makes it our favourite choice as an emulator.

5.2.2 Tracing in Qemu

Tracing in Qemu is event driven. Qemu provides various inbuilt options for tracing. Some of inbuilt tracing mechanisms that we explored and tried out were Nop, Log, Simpletrace and FTrace etc. These tracing mechanisms trace the functioning of Qemu and not what we want. We want to emit out instruction traces which are supposed to be executed on vCPUs. Qemu also provides debug options `in asm` and `out asm` to get a hold on the assembly code which is processed by TCG. `in asm` shows the target assembly code for each compiled translation block (TB). While they emit traces, they don't give the user explicit control - the user won't be able to start or stop tracing from a particular point. Moreover, the assembly code generated in these cases involves optimization by qemu. So, we shifted to TCG-tracing through TCG Plugins.

TCG Plugins

Qemu is a large open-source project and making changes in the code of Qemu becomes scary. Hence, Qemu supports external TCG plugins [**?**]. Through plugins, we can subscribe to multiple events and whenever the event happens, the local variables and state at that point can be monitored. For working with plugins, Qemu exposes some pre-defined set of functions which the user can use to track the TCG translation process. Most of them register for events and then call callback function when that event occurs. Plugins are kept in `build/contrib/plugins` directory of the qemu source. These can be run by adding

`-plugin ../contrib/plugins/<plugin-lib-name> -d plugin,nochain`

option from the `build` directory in Qemu code.

5.3 Tracelog Plugin

To obtain instruction traces for both kernel and user level code, we developed our own plugin named `Tracelog`. This plugin would start tracing as soon as the application code begins to execute. We explain below the two approaches that we developed, based on how Qemu initiates tracing.

5.3.1 Approach 1: Insertion of Nops

Tracing would begin when qemu encounters a fixed large number of `nops` from the instructions that are to be executed on the vCPU. Since the translation process is dynamic, this point of occurrence of `nops`, can help to get all the instruction executed after tracing is enabled in application execution.

Logic

1. The plugin takes two arguments(relative to the start of application execution) - start(The instruction count to start the execution from) and count(The number of instructions from start for which we want to trace). The arguments to plugin are given as `-plugin ../contrib/plugins/libtracelog.so,start=10000,count=1000`.

2. The `qemu_plugin_install` function registers the signal handler for SIGINT 40, parses the command line arguments as well as subscribes to the event of translation of instructions in qemu.
   `qemu_plugin_register_vcpu_tb_trans_cb`. It takes the callback function `vcpu_tb_trans`. This callback is registered whenever the instruction translation is about to happen.

3. The callback takes the instruction, one at at time, from the translation block using the pre-defined function `qemu_plugin_tb_get_insn`. It disassembles the instruction and registers another callback `vcpu_insn_exec` using `qemu_plugin_register_vcpu_insn_exec_cb`.Here, the plugin also sets a global static variable `start_tracing = true` when it receives consecutive 997 `nop` instructions. This function calls the callback every time instruction is executed.

4. The `vcpu_insn_exec` appends the instruction to the count in case `start_tracing` is enabled. It prints the trace to a file `instTrace.log` given `start_tracing == 1` and we have reached the `start` point of tracing relative to the start of application and have not exceeded tracing `count` number of instructions else does nothing.

   On plugin exit plugin exit prints the instruction count on the console for the user.

Printing to file

When printing the traces directly to the console, the application execution got stuck in the kernel mode in the scheduler code. We anticipated that this was due to some timeouts in kernel since printing to the console while performing translation in Qemu was a slow process. So, to overcome this, we printed the trace directly to a file.

5.3.2 Approach 2: Addition of `int 100` instruction

As soon as qemu encounters `int 100` instruction from the instructions that are to be executed on the vCPU, it would initiate tracing.

Logic

In this approach, the plugin would start recording the trace as soon as it encounters the `int 100` instruction. The `int` instruction causes an interrupt for the OS and `100` is the interrupt number. This interrupt number is not defined in the OS, so we can use this to

start tracing.

Handling the generated interrupt from Qemu

This interrupt should not be forwarded to the Host OS, which would otherwise kill in case this interrupt is not handled. In Qemu source code, function `gen_interrupt` is responsible for generating interrupts to the host. We skip generating an interrupt to the host in case the instruction `int 100` or `int 101` are caught through this change in file `target/i386/tcg/translate.c`.

```
if(intno == 100){
    int retVal = system("../cmd.sh");
    return;
}
```

It also executes the script cmd.sh Fig. 5.2 to raise an interrupt from the host to Qemu process, which is registered in the plugin Tracelog. This initiates the tracing from the plugin.

```
#!/bin/bash

process_id=`/bin/ps -fu $USER| grep "qemu-system-x86_64" | grep -v "grep" | awk '{print $2}'`
echo "PID for qemu instance: $process_id"
kill -n 40 $process_id
echo "Trace-start signal sent"
```

Fig. 5.2: Shell-script to send signals to running qemu-process.

1. Argument Parsing, and registration for events similar to the above approach. Additionally, the plugin registers a signal handler for SIGINT 40. On receiving this signal from the host, the plugin sets `start_tracing = true`.

2. Perform printing and plugin exit as above approach.

5.4 Verification of Plugin

Both the plugins have same output trace generation mechanism. Hence we aim in this section to verify that mechanism.

To verify the instructions being printed, we found out a command-line argument `-icount` pre-built in Qemu, that helps to get the count of instructions from the start of the program. When this option is supplied, the file `icount.c` starts to come in way. This files plays an active role in yielding the count of instructions that are being executed on Qemu. In this file `icount_update_locked` function is responsible for updating the count of instructions atomically. Hence, we put a print statement in this function which yields the icount value every time it is being updated.

The interrupt functionality is stopped for some time and the count obtained from the start of the run is observed from both Qemu and the `-icount` option. These both turned out to be same. This verifies that the count of instructions that we get are indeed correct.

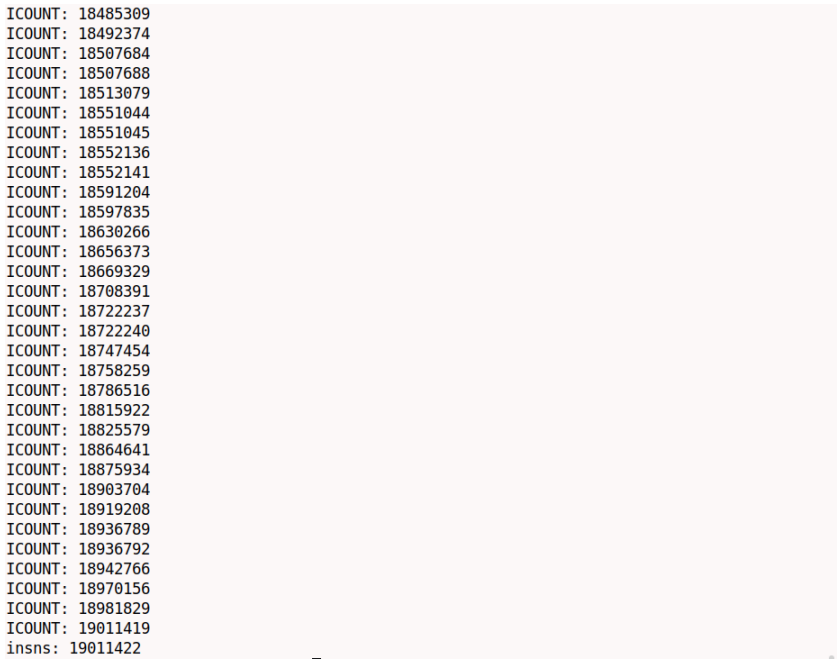


Fig. 5.3: Verification of Number of instructions executed from start of Qemu execution.

Fig. 5.3, shows the output generated by both `-icount` and Tracelog plugin. The last

line is the output of plugin which is the total number of instructions executed from the start of boot of Guest OS on Qemu. The rest are the print statements that we introduced in file `icount.c`, with second last being the number executed as recorded by icount. We can see that both of them turned out to almost same.

5.4.1 Verification of instruction traces

To check that the instruction traces that we obtained are correct, we need to check both the kernel level and the user level instructions and the order in which they are given out by the plugin. Along with this, we also need to verify that the loops are unrolled and there is no optimization/caching in the code done by Qemu itself.

To do this, we disassembled the kernel executable of xv6. From the trace generated, we could figure the value of instruction pointer (IP). We could then choose one IP randomly from the log of instruction traces obtained and figure out where that IP lies in the disassembled kernel. We kept matching the instructions from that point onwards and checked in the corresponding trace that the next instruction is indeed the one which is supposed to executed. Such checks were specially important in case of jump statements and function calls, to know that there was actually a jump in instruction pointer to the other address in the generated log of inst-traces.

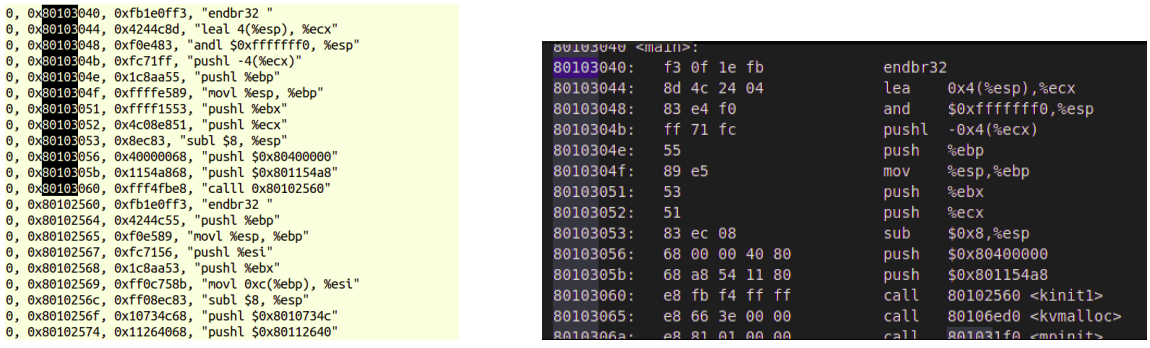


Fig. 5.4: Verifying instruction trace (left) from kernel dump (right) of xv6.

As can be seen from Fig. 5.4, the instructions in the main function of kernel are matching

kernel dump. Also, once the call instruction is encountered, the instruction pointer jumps to function address and it is evident in instruction trace itself.

Then we found out loops in the xv6 source code to verify that they are correctly unrolled to the number of times the loop is actually executed while running. Matching was done based on the instruction pointer values which was same in both the disassembled kernel code and the traces obtained from plugin. We also added our own loops in the source code o xv6 to verify further. The instructions in the body of loop were checked and we made sure that there was no caching or optimization done by qemu.

Same was done for checking user level code as well. We executed a user level program and made sure that the instructions specific to that program were coming out in correct order in the traces. Hence, we made sure that our plugin was generating correct instruction traces.

5.5 Examining Precision in start tracing for general methods

We have mentioned two different approaches used by different versions of our plugin. These two approaches give us means by which we can start and end tracing by either inserting `nops` or `int 100` (marker instructions), but these instructions need to be added just before the application has started to execute. One way is to insert this instruction at the start of the assembly of the application code, but the source code of the application is not always available(it might be a complete black box to us) and even if code is available, modifying it is not a convenient method.

Another way would be to create a wrapper program for the unknown program. This wrapper program will first insert the marker instruction to start tracing, then will execute the target program like a black-box execution and finally will insert another marker instruction to stop tracing. However, this will lead a lot of kernel level instruction involving a new process creation, copying of stack variables, scheduling the process etc. to come into picture. This would take time and delay our tracing. So this is not precise enough to start tracing just at start of application.

Yet another option for starting tracing mechanism is to give manual signal from the host to qemu(instead of sending from Qemu as in Section 5.3.2) as soon as the user runs the application on the guest OS. But as it is evident, this method would not be precise enough, since the user will be manually giving signals to start and stop.
Because of these limitations, using any of the above approach makes tracing either tiresome (due to large amount of time required) or imprecise. In the next chapter, we provide a much more precise and novel way by modifying Operating System itself.

5.6 Conclusion

In this chapter, we build the background required for Qemu and motivate its requirement in the project. Along with this, we discuss TCG Plugins, and how they make extracting information while qemu is executing easier. We describe the two approaches to initiate and obtain the instruction tracing using Qemu Trace. The first approach described is `nop` instructions while the other is using `int 100` instruction. Moreover, we proved that the instruction traces we are getting are indeed correct. The implementation of the approaches described is novel and the contrast can be seen from the previous work 2.

At the end of the Chapter, we motivate the need the novel approach of starting tracing from the Kernel itself instead of application code or manually by the user.

Chapter 6

Modification in Operating Systems

6.1 Introduction

As of now, the disadvantages of starting tracing manually for black-boxed programs are clear. In order to get more accurate traces, we need a more robust and precise way to start tracing at a point which is closer to initial instruction of application. This chapter illustrates modification of operating systems in such a way that the tracing starts just before the execution of actual program along with the proof of correctness of the approach.

6.2 Modifying EXEC System Call

Whenever a program or an application is executed, a new process is created on kernel with the help of exec system call. This is common for every operating system. Exec copies all the necessary information from user stack to kernel's process stack and mark it READY/RUNNABLE to be scheduled on processor. Since we can't have control over scheduler, exec is the closest we can get to before the execution of the application. This makes our approach much more precise as compared to any other tool available as of now. Since exec is used to run many kernel and user processes, blindly inserting a marker instructions in exec will start tracing at boot time itself. To avoid that, we need a mechanism

to let exec know when to insert the marker instruction. This can be done by passing a marker argument to the process while executing. Exec takes the command line arguments from the previous process's stack and copies them onto new process's stack. We hack this process, and check if the marker argument is present in the list of arguments. If so, then that marker argument is not copied to new process's stack and the marker instruction for Qemu to start tracing is inserted. So, now whenever the user wants to trace a program, it just has to execute the program with the marker argument as one of the arguments.
In the next two sections we will dive deeper into the changes done in the operating systems and their proof of correctness.

6.3 XV6

XV6 is an open-source, light-weighted operating systems developed in MIT for educational purposes. It follows UNIX style of operating systems. Although the benchmarks we want to run are not xv6 based, we chose xv6 as our starting point because its light-weighted, easy to modify and debug and it boots fast, in contrast to Linux which is heavy and takes a lot of time to build and boot. Since we wanted to modify Qemu and test our plugin (Section 5.3.1), doing it using xv6 was far more faster and simpler. Also, the extensive educational use of xv6 made it well-versed and simple to use.

6.3.1 Modifying xv6 kernel

In xv6, exec was the only function (in exec.c) where the modification was required. As stated in the previous section, when the arguments are copied to new process's stack, there we intercept the marker argument and do the necessary changes. This can be seen in the updated code snippet of a part of exec function Fig. 6.1. Here we check for the marker argument '-XOXO' and the remaining part of code is modified accordingly.

In xv6, a system call starts with an interrupt to processor and so there might be other

```
int startTracing = 0;     // flag to indicate onset of tracing
int loopVar = 0;
argc = 0;

// Push argument strings, prepare rest of stack in ustack.
for(loopVar = 0; argv[loopVar]; loopVar++) {
  argc = loopVar;

  // compare with marker argument '-XOXO'
  int tempA = strlen(argv[loopVar]), tempB = strncmp(argv[loopVar], "-XOXO", 5);

  if((tempA==5) && (tempB== 0)){
    startTracing = 1;
    cprintf("I am inside. with startTracing = %d\n", startTracing);
    continue;
  }
  if (startTracing){
    argc--;
  }
  if(argc >= MAXARG)
    goto bad;
  sp = (sp - (strlen(argv[loopVar]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[loopVar], strlen(argv[loopVar]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
}

if (startTracing)
  argc--;
ustack[3+argc] = 0;
```

Fig. 6.1: Part of updated exec function in xv6

interrupts happening while a system call is being performed. So, on the onset of any system call, all other interrupts are temporarily disabled for the top half. The lower half is scheduled on processor which is optimal to do, but at that time interrupts can happen. This could be an issue in one of our methods to send marker instruction to Qemu. The way we are using a series of continuous `nops` as marker instruction can be disrupted due to unwanted interrupt in between, even if it is very unlikely. Hence, we disable the interrupts before adding the 997 `nops` and enable it just after addition through the `pushcli` and `popcli` already implemented in xv6 as in Fig. 6.2. This ensures the there is a contiguous sequence of 997 `nops`.

```
if (startTracing)
{
  pushcli();
  addNops();
  popcli();
}
```

Fig. 6.2: Disabling and Enabling CPU-Interrupts

Fig. 6.3: xv6 running forktest with `-XOXO` argument to start the tracing.

6.3.2 Verification and Proof of Correctness

We have already verified that the instruction traces emitted by Qemu, either user-level or kernel-level are correct. That verification was to check the sanity of the traces obtained. However, it didn't ensure that the traces obtained are complete for a program and the tracing functionality isn't breaking due to any system call in the program and there is a smooth transition between kernel and user level instruction.

In order to do this, we created a custom program (`userCodeTest`) which only comprised of assembly instruction in its body. These assembly instructions included a simple `add` and `mov` instruction only with RAW dependency. This code file was be generated using a shell script and the user can specify the no. of instructions it wanted in the `userCodeTest` by command line argument in the script `codeGenerator.sh` that we wrote. The code file generated from the script is then added to xv6 kernel's file system and can be executed in xv6 directly just like `forktest` and `usertests`. However, there is a limit upto which size of an executable is permissible in xv6's file system due to which we couldn't add more than 40K instructions, but those were enough. Refer to Appendix Section.9.3.

In this program, after adding all these instructions, we added a final marker instruction to stop tracing, since we wanted to know the exact no. of instructions executed. However it was found that this number was varying over multiple runs. On careful examination of traces of different runs, it was found that the number of user-level instructions were always same but the number of kernel level instructions fluctuated. Moreover we noticed that order

Fig. 6.4: kdiff3 output for two different runs of `userCodeTest`

of user-level as well as kernel level instructions was correct and transition between the two modes is systematic and non-abrupt. We compared traces of two runs using tools like kdiff3, and found that the cpu interrupts for the two programs were occurring at different times, and sometimes the program spend more time in kernel mode due to scheduler. As can be seen in Fig.6.4, for the second run the cpu timer interrupt occurred before the first run so that's why in the former we can see the user program instructions coming before the latter. By using python scripts, we made sure that there is no out of the box instruction which is executing while the user-instructions are completely executed.

To check for the case when there is a system call present in the code, we created a new system call named `nothing`, which just simply returns an integer and does nothing. This system call was then used in our `userCodeTest`, and the instruction traces obtained were verified to be correct.

6.4 Linux

Linux is an open family of UNIX Operating Systems based on Linux Kernel. Instead of installing a complete Linux-distribution like Ubuntu, we sticked to Linux kernel itself since we need that only. Using only Linux kernel with minimal packages reduces the boot time by a good amount without affecting the functionality that we want.

In Linux-kernel, the exec system call is not as straight-forward as it is in xv6. There are two different system calls which have the functionality similar to exec in xv6. Those are `execve` and `execveat`. Depending on the value of the macro `CONFIG_COMPAT`, these two system calls call four different functions, which in turn calls a common function `do_execveat_common`.

This function creates the binary process page map for the new process to execute. It uses a function called `copy_strings` to copy arguments from the current user stack to new process's stack. Once it has loaded all arguments it calls `bprm_execve` which executes the new process.

6.4.1 Modifying Linux Kernel

The copying of arguments of exec from old process's stack to new process's stack is done in the function `copy_strings`. Similar to xv6, here too the basic approach is to hack this copying process and check for a marker argument. However, in Linux it's not that trivial. The arguments that need to be copied are stored in user stack, so trying to access/modify them from kernel causes General Protection Fault. This fault occurs because kernel doesn't trust user space stack pointers and so creating a variable in kernel and trying to tamper with its value is not allowed. This fault takes the cpu in panic mode and will not even let it boot. Hence, blindly adding checking statements in the code don't work. Moreover, the whole kernel is build with several build-sensitive compiler optimizations which makes checking of this hard.

```
            flush_dcache_page(kmapped_page);
            kunmap(kmapped_page);
            put_arg_page(kmapped_page);
        }
        kmapped_page = page;
        kaddr = kmap(kmapped_page);
        kpos = pos & PAGE_MASK;
        flush_arg_page(bprm, kpos, kmapped_page);
    }
    if (copy_from_user(kaddr+offset, str, bytes_to_copy)) {
        ret = -EFAULT;
        goto out;
    }

    argString = kaddr + offset;

    strncat(fullArg, argString, offset);
}

// reading what is copied to the kernel address spacce
pr_info("The argument character is %s\n", fullArg);

if (lenBackup ==6 && (strcmp(fullArg, "-XOXO") == 0)){
    pr_info("Starting the Tracing Mechanism.\n");
    bprm->p+=lenBackup;
    bprm->argc--;
    startTracing = 1;
}
```

Fig. 6.5: Part of update `copy_strings` in `fs/exec.c`

So instead of directly accessing the incoming arguments from user stack, what we do is let the argument get copied on kernel map-page one by one (by `copy_from_user` utility, it does all the required checks on the user space arguments) and then check if the latest copied argument is the marking argument or not. If yes, then change the page's pointer position back to last value so that the latest value gets overwritten and removed.

However, there is one more case that needs to be accounted. If the size of total arguments become greater than what a page can accommodate, then usually it leads to any one of the argument to be split in multiple pages. The same could happen with our marker (`-XOXO`) argument also. This will lead us to miss the marker argument unknowingly and this can tamper with the functionality of the application. To avoid that, we maintained a local variable to keep a copy of complete argument string even if its broken down in pages and do the check on that local variable. 6.5 shows the snippet of part of `copy_strings` modified code.

```
[  537.164244] Going in
[  537.164356] The argument character is -XOXO
[  537.164889] Starting the Tracing Mechanism.
[  537.165143] Going in
[  537.168900] The argument character is 280
[  537.169649] Going in
[  537.169700] The argument character is input.source
[  537.170007] Going in
[  537.170047] The argument character is ./bzip2_base.amd64-m64-gcc43-nn
```

Fig. 6.6: Kernel log printing on console

6.4.2 Verification

To verify the instruction tracing in Linux, simple codes with `add` and `mov` instruction was written and executed. Just like the verification in xv6, the same conditions were checked to verify that the traces emitted are indeed correct. Moreover, to check the sanity of kernel, we printed out the kernel logs on console so that if any process fails, the user will be able to know about it, which helps in debugging. Fig. 6.6 shows how the kernel logs look like. Refer to Appendix-B Section 9.4 for the instruction to print kernel logs.

6.5 Results

Now we have a linux instruction emitter ready. The input to this emitter was obtained in Interval Estimation in Chapter 4 where we got the point to start tracing from and the number of instructions to trace for. These values are then used as input to our plugin to emit the desired instruction trace.

The benchmarks executables generated in when running the workloads in Chapter 3,

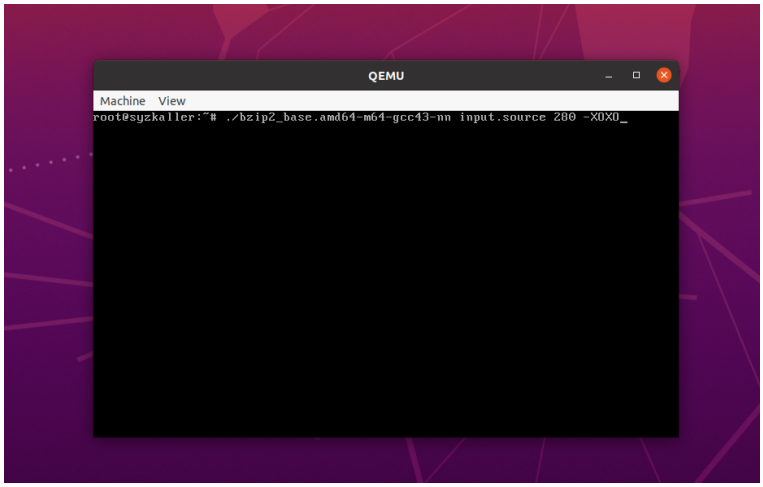


Fig. 6.7: Running bzip2 on Qemu to get instruction traces

were copied to the guest operating system along with the input files they required. These executables were then run on our modified Linux kernel with `-XOXO` argument to indicate tracing. Depending on the benchmark used, corresponding input were given to plugin which were obtained from Interval Estimation.

The benchmark 401.bzip2 was run on qemu with input - `input.source 280` as shown in the Fig. 6.7. The corresponding values used as input to plugin were: 127 billion instruction to skip before starting tracing and 2.78 billion instructions to emit in the trace. Fig. 6.8 shows the sample instruction traces obtained. These instructions were verified to be a part of bzip2 executable.

```
0, 0x55d218768445, 0x25e78944, "movl %r12d, %edi", 0x0
0, 0x55d218768448, 0x87863c2b, "subl (%rsi, %rax, 4), %edi", 0x0
0, 0x55d21876844b, 0x1ac76348, "movslq %edi, %rax", 0x0
0, 0x55d21876844e, 0x1013d, "cmpl $0x101, %eax", 0x0
0, 0x55d218768453, 0x9f6870f, "ja 0x55d218768e4f", 0x0
0, 0x55d218768459, 0x247c8b48, "movq 0x20(%rsp), %rdi", 0x0
0, 0x55d21876845e, 0x2587048b, "movl (%rdi, %rax, 4), %eax", 0x0
0, 0x55d218768461, 0x28244489, "movl %eax, 0x28(%rsp)", 0x0
0, 0x55d218768465, 0x2824448b, "movl 0x28(%rsp), %eax", 0x0
0, 0x55d218768469, 0x40244439, "cmpl %eax, 0x40(%rsp)", 0x0
0, 0x55d21876846d, 0x197e840f, "je 0x55d218769df1", 0x0
0, 0x55d218768473, 0x2401f883, "cmpl $1, %eax", 0x0
0, 0x55d218768476, 0x1b5c860f, "jbe 0x55d218769fd8", 0x0
0, 0x55d21876847c, 0x244c3944, "cmpl %r9d, 0x38(%rsp)", 0x0
0, 0x55d218768481, 0x9c88e0f, "jle 0x55d218768e4f", 0x0
0, 0x55d218768487, 0x28247c8b, "movl 0x28(%rsp), %edi", 0x0
0, 0x55d21876848b, 0x9ff478d, "leal -1(%rdi), %eax", 0x0
0, 0x55d21876848e, 0x2824c289, "movl %eax, %edx", 0x0
0, 0x55d218768490, 0x280ff883, "cmpl $0xf, %eax", 0x0
0, 0x55d218768493, 0x1b59870f, "ja 0x55d218769ff2", 0x0
0, 0x55d218768499, 0x8cb36348, "movslq 0x1e8c(%rbx), %rsi", 0x0
0, 0x55d2187684a0, 0x40060c8d, "leal (%rsi, %rax), %ecx", 0x0
```

Fig. 6.8: Sample instruction traces emitted from qemu on running bzip2

```
[  790.012294] The argument character is -XOXO
[  790.014911] Starting the Tracing Mechanism.
[  790.015971] Going in
[  790.028248] The argument character is 280
[  790.031192] Going in
[  790.043312] The argument character is input.source
[  790.045209] Going in
[  790.049997] The argument character is ./bzip2_base.amd64-m64-gcc43-nn
PID for qemu instance: 5375
Trace-start signal sent
Called Handler
FOUND IT IN TRANSLATE.C: 0
```

Fig. 6.9: Kernel log using printk() tool (Refer to Appendix 9.4) to send signal to Qemu for initializing tracing by the plugin.

6.6 Conclusion

In this chapter, we proposed a novel way to modify exec system call of kernel in order to support our cause. Not only this approach is much more precise but also it takes out the need to insert marker instruction in the source code of an application, as compared to the previous approaches we have stated. We then provided the details of modifying kernels for both - xv6 and linux, along with the verification. Towards the end of the chapter, we have fully developed our instruction-trace emission tool and used it to emit the traces for execution intervals of cpu-2006 benchmarks.

Chapter 7

Conclusion and Future Work

In this project, we completed all the steps mentioned in Section 1.1. We were able to identify application phases during application execution based on application characterizers like the IPC count in our case. This process included setting up of CPU-2006 benchmarks and creating a tool which can run these benchmarks automatically and generate perf data (IPC, instructions vs time information) and plot the same. Along the way, we developed other utilities for instance for building a stack of all the plots. Using the generated perf data, the tool will itself do noise reduction and identify the phases in the execution of these benchmarks. We then performed the smoothening operation on the identified phases and found out the representative intervals for a phase that could be traced in Qemu.

Qemu was explored and plugins were identified to be an effective way to obtain instruction traces. We wrote our plugin Tracelog and verified its correctness along the way. The development of plugin was gradual, from a basic plugin which emits out all the instruction traces to a plugin which starts tracing on receiving signal from host and emits out traces only for a desired interval of instructions ensuring correctness of emitted traces at each point. Later, modification to the Operating System was performed. These changes included managing the addition of trace option and addition of the marker instruction at the start point of execution of the program. We also discuss the extra instruction problem

that we faced along the way.

We developed a novel tracing mechanism and an end to end workflow starting from phase detection to emission of traces in that particular representative interval.

One can extend this project by fine-tuning the end-to-end work flow. Mainly the phase detection part can be extended to take into consideration more system characterization parameters. One can run the obtained instruction traces on a simulator like Tejas and compare the statistics generated by the tool with the Perf statics. Alongside, the tool can even be fine-tuned by running it on other SPEC benchmarks.

Chapter 8

Appendix-A

8.1 Automation of Perf-Data Generation

A python script was written which was able to run all the benchmarks with perf monitoring and store the results in secondary storage of the system in forms of plots and csv files. Also, in case metric needs to be changed for analysis, for instance studying cache hits/misses instead of IPC, this automated script will help us to do it hassle-free. The overview of the features is described below.

- A directory structure was maintained to store perf data in the form of CSV files in an organized fashion.
- The script could run all the benchmarks at once or specific benchmarks as specified by the user through command line arguments. It could be used to execute benchspec with different input-size('ref', 'test') as specified by user. By default, it will run on 'ref' data. Initially, if the benchmark was not setup, it would also setup the benchmark, except for perlbench benchspec which required change in the Makefile (it can be compiled only with specific gnu version), and those changes had to be done manually.
- Multiple runs of the script would generate new CSV files in the directory instead of

overwriting the previously generated ones. This helps to gather data over multiple runs for averaging.

Another python script was written to iterate over the directory structure created by the above script to average out the data in the csv files. The averaged files generated by this script were in the same location as the data of perf utility. This script also provisioned users to plot graphs (for instance Fig. 8.1) for all the averaged data and store them in another folder with same directory structure as was created while running the benchmarks.

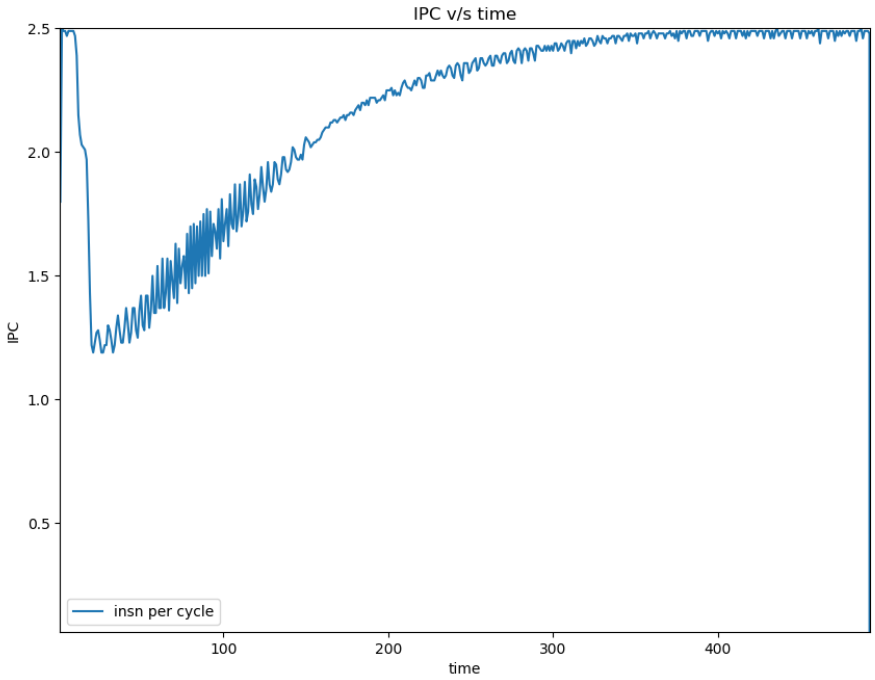


Fig. 8.1: Time varying graph of IPC(Instructions per cycle) for 436.cactusADM reference benchmark. The executable run with input - benchADM.par

8.2 Density-Based Spatial Clustering of Application with Noise

Density-based spatial clustering of application with noise (DBSCAN) is a non-parametric data clustering algorithm which forms clusters for closely lying points and marks the lower dense outliers (points which don't have close neighbours) as noise. DBSCAN is a better

choice than a lot of other clustering algorithms like K-Means clustering or Mean-Shift clustering, as it doesn't need a pre-set number of clusters and can classify outliers as simply noise in the data. Moreover, it takes into account the continuity required for a phase in time series data. The clusters formed can be arbitrarily shaped or sized. Because of these features offered by the algorithm, DBSCAN is ideal for phase detection.

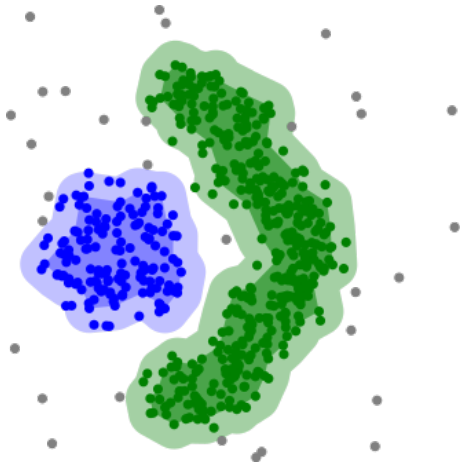


Fig. 8.2: Clustering using DBSCAN. [2]

The working of the algorithm is described below:

1. From the input set of points, a random point is chosen and all points which are close to it, upto a distance d apart, are found. If the number of points is greater than a threshold value n then the clustering process starts otherwise the point is marked as noise.

2. Now for all the points found in step 1, nearby points which are upto distance d are found and this step is repeated until no other point is left.

3. Then the set of points found in step-2 are grouped together in one cluster. If there are no more points left in data, the algorithm terminates, otherwise step 1 - step 3 are repeated, until all points lie in one of the clusters or are labelled as noise.

8.3 Exploring Tracing Options of Qemu

This section describes those options with along with their drawbacks encountered after trying each of those options.

In qemu, tracing is mechanised with the help of a utility present in the source code called trace-events. In brief, this utility allows user to select a particular function and the logger logs out information present in that function like the value of variables. In order to do this tracing, the user has to register this `trace_<function_name>` in a file and include it in the same directory. The user is also supposed to call this `trace_<function_name>` from inside the function. As in Fig. 8.3, whenever the function `acpi_cpu_has_events` is called it will print out corresponding idx, inserting and removing values.

The way these traces will be recorded by qemu can be handled by qemu's trace-backends.

```
cpuhp_acpi_invalid_idx_selected(uint32_t idx) "0x%"PRIx32
cpuhp_acpi_read_flags(uint32_t idx, uint8_t flags) "idx[0x%"PRIx32"] flags: 0x%"PRIx8
cpuhp_acpi_write_idx(uint32_t idx) "set active cpu idx: 0x%"PRIx32
cpuhp_acpi_write_cmd(uint32_t idx, uint8_t cmd) "idx[0x%"PRIx32"] cmd: 0x%"PRIx8
cpuhp_acpi_read_cmd_data(uint32_t idx, uint32_t data) "idx[0x%"PRIx32"] data: 0x%"PRIx32
cpuhp_acpi_read_cmd_data2(uint32_t idx, uint32_t data) "idx[0x%"PRIx32"] data: 0x%"PRIx32
cpuhp_acpi_cpu_has_events(uint32_t idx, bool ins, bool rm) "idx[0x%"PRIx32"] inserting: %d, removing: %d"
```

Fig. 8.3: Sample trace-events for cpus.c (A file in qemu source code)

Some of the trace-events might be suitable for a particular trace-backend only. These trace-backends are

1. **Nop**: It ignores all the trace-events and is equivalent to no tracing.

2. **Log**: It sends all the trace-events to std-error with an option to record timestamps.

3. **Simpletrace**: It writes binary trace-logs to a file which is faster than Log. This binary file is decoded using a python API present in the source code. We used it in order to understand the execution flow of qemu, using generic trace-events which are present in source code.

4. **Others**: The other trace-backends which are present are - Ftrace, LTTng, Syslog

and SystemTap. Out of these we also used SystemTap, which uses dtrace sdt probes, to acquire in-depth knowledge of qemu execution.

The user has to specify the trace-backend at compile time during configure command. While executing qemu user will have an option to either trace one function or a group of functions residing in a file. The example commands to do so can be found here 3.

8.4 howvec and execlog Plugins

- **howvec** - This plugin finds hits of each type of instruction that was run by the vCPU and displays the top n to the user on host. The plugin subscribes to the translation event of the vCPU, disassembles the instruction and adds that to a map datastructure as in Fig. 8.4. This plugin gave us a direction on how to count the instructions.

- **execlog** - This plugin prints the instruction in the order of execution. This gives an indication on how to obtain instructions from the vCPU.

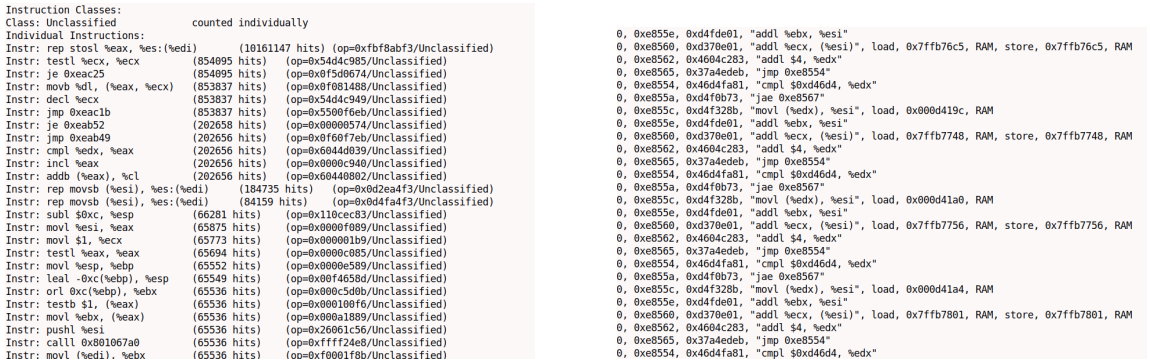


Fig. 8.4: Output for Howvec and Execlog plugins

Chapter 9

Appendix-B

9.1 Qemu Commands

List of some of the sample commands that were used:

1. To configure qemu with Simpletrace as trace-backend and target architecture x86_64:

   ```
   qemu/configure --target-list="x86_64-linux-user,x86_64-softmmu"
   --enable-trace-backends=simple --disable-xen --disable-kvm
   ```

2. To configure qemu with debugging and tcg plugins enabled:

   ```
   qemu/configure --target-list="x86_64-linux-user,x86_64-softmmu"
   --enable-debug-tcg --enable-gtk --enable-debug-info --enable-debug
   --enable-debug-stack-usage --disable-kvm --disable-xen
   ```

3. To run qemu with tracing enabled (Simpletrace), add the following option to Command 4:

   ```
   --trace events=qemu/build/trace/trace-events-all
   ```

4. To run qemu and boot xv6 on it, given the path to img files. The option smp indicates the number of vCPUs to be formed while the option m states the amount of physical memory(in MBs) given to the process:

```
qemu/build/qemu-system-x86_64 -drive file=fs.img,index=1,media=disk,
format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 1
-m 2048
```

5. To run qemu with howvec plugin, nochain option signifies no chaining of translation blocks, add the following option to 4:
   `-plugin qemu/contrib/plugins/libhowvec.so -d plugin,nochain`

6. To run qemu with tracelog plugin, note that the input value for in the instruction interval [10000,20000] from the start of the Application add the option to 4 :
   `-plugin qemu/contrib/plugins/libtracelog.so,start=10000,count=10000 -d plugin,nochain`

9.2 Debugging Qemu

There are two ways to use Qemu with gdb. First is when we want to debug the OS running on Qemu while Second is when we want to debug Qemu itself. The latter case is useful for us, since our work resides in understand and modifying Qem. So the basic way to invoke qemu inside gdb is:
`gdb --args {path_to_qemu_executable} {qemu_arguments}`.
An example for this can be:

```
gdb --args qemu-system-x86_64 -drive file=../../xv6/fs.img,index=1,media
=disk,format=raw -drive file=../../xv6/xv6.img,index=0,media=disk,format
=raw -smp 1 -m 2048
```

which boots xv6 on qemu x86-64 architecture.

However, while debugging Qemu we have to ignore SIGUSR1 signal so that GDB doesn't care about its occurrence at all as well as should not stop if this signal happens. To do so, after opening gdb console the user can type: `handle SIGUSR1 noprint`

9.2.1 Debugging Plugins

While debugging one can't directly get into plugins code since they are dynamically linked and are not directly compiled with qemu's executable. So, to get their symbol table, first we need to compile them with `-g` option by making change to the Makefile of xv6 and then add the symbol file inside gdb console using `add-symbol-file`.

9.3 Adding large user Tests in Linux

When large tests were added, the fs.img image failed to create. This file was required to boot the xv6 kernel on Qemu. Hence, we restricted the verification process till 40k assembly instructions in UserCodeTests that we created.

```
./mkfs fs.img README _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressf
s _wc _zombie _userCodeTest
nmeta 59 (boot, super, log blocks 30 inode blocks 26, bitmap blocks 1) blocks 941 total 100
0
mkfs: mkfs.c:270: iappend: Assertion `fbn < MAXFILE' failed.
make: *** [Makefile:193: fs.img] Aborted (core dumped)
make: *** Deleting file 'fs.img'
```

Fig. 9.1: File Exceed error while performing `make fs.img`. [2]

9.4 Printing the Kernel Execution logs on the console

To debug the kernel, we used the function `printk()` [8]. This tool helps to print the kernel log for debugging purposes. The logs printed by this command can be obtained by using the `dmesg` instruction on the console of the completely booted Guest OS. But in case the kernel fails to boot because of the fault we are trying to debug, we need to alter this functionality to print the logs on the screen using kgdbdoc.

```
./qemu-system-x86_64 -m 10240 -smp 1 -kernel ~/BTP/linux-kernel/
linux-5.17.1/arch/x86_64/boot/bzImage -append "console=ttyS0
root=/dev/sda earlyprintk=serial net.ifnames=0 nokaslr" -drive
file=~/BTP/linux-kernel/image/stretch.img,format=raw -net user,host=
```

```
10.0.2.10,hostfwd=tcp:127.0.0.1:10021-:22 -net nic,model=e1000 -plugin
;../contrib/plugins/libtracelog.so -d plugin,nochain
```

This logging helped us to figure out that General Protection fault was occurring when we tried to access the variables from the user space. Otherwise, it would not have been possible to figure out the error because the kernel simply hung at that point.

9.5 Github Repository

All the code related to this tool and work done can be found in: `https://github.com/Parul664/Workload-Characterisation`

References

[1] J. Henning, "Spec cpu2006 documentation," October 2011. [Online]. Available: https://www.spec.org/cpu2006/Docs/

[2] "Image illustration for dbscan." [Online]. Available: https://en.wikipedia.org/wiki/DBSCAN#/media/File:DBSCAN-density-data.svg

[3] "Linux perf documentation." [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[4] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. D. Emmons, and N. C. Paver, "A structured approach to the simulation, analysis and characterization of smartphone applications," *IEEE*.

[5] W. Feng and S. Huang, "Energy-efficient cluster computing via accurate workload characterization," *IEEE*.

[6] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," 2002. [Online]. Available: https://cseweb.ucsd.edu/~calder/papers/ASPLOS-02-SimPoint.pdf

[7] "Qemu tcg documentation." [Online]. Available: https://qemu.readthedocs.io/en/latest/devel/tcg-plugins.html

[8] "Linux kernel printk() reference." [Online]. Available: https://www.slideshare.net/xen_com_mgr/from-printk-to-qemu-xenlinux-kernel-debugging