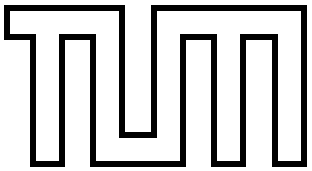


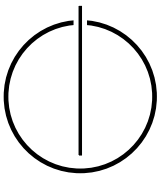
FAKULTÄT FÜR INFORMATIK

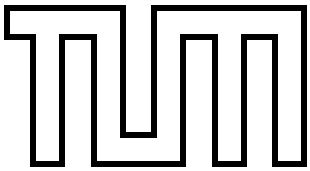
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Detecting Performance Bottlenecks in HPC Applications Using Dynamic Metric Thresholds

Akash Mundra





FAKULTÄT FÜR INFORMATIK

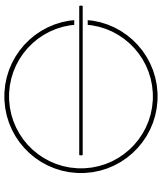
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Detecting Performance Bottlenecks in HPC Applications Using Dynamic Metric Thresholds

Ermittlung von Performance-Engpässen in HPC Anwendungen anhand dynamischer Grenzwerte für Laufzeitmetriken

Author:	Akash Mundra
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Louis Viot, PhD., Nico Tippmann, M.Sc.
Date:	16 January, 2023



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 16 January, 2023 Akash Mundra

Acknowledgements

I would like to express my heartfelt gratitude toward everyone who helped me through this thesis.

First of all, I would like to thank Univ.-Prof. Dr. habil. Philipp Neumann, Louis Viot, and Nico Tippmann for supervising my thesis and for the patience they showed me during the complete duration of this thesis.

I would also like to thank Dr. Axel Auweter and MEGWARE GmbH for the support and resources they provided me for this thesis.

I am extremely grateful to Amartya Das Sharma who helped me during this thesis and my team at QYOBO who were also very accommodating and patient toward me while I was working on this thesis.

Abstract

HPC applications are only able to achieve 15-20% of peak performance as compared to the theoretical peak performance of modern day supercomputers. Traditionally performance analysis of HPC applications has been highly developer dependent. The developer is provided with data containing several metric values and using these metric values the developer needs to both judge the possible bottlenecks present in the application and find possible optimisation techniques to counter these bottlenecks. The connection between analysis and optimization is held together through the ad-hoc knowledge of HPC developers. [Gra19] The aim of this thesis is to create an autonomous application agnostic analysis tool that can analyse runtime data for performance bottlenecks on any system architecture. A blackbox tool, that uses predefined or user defined analysis models to study the performance data generated during an application run to indicate the presence and type of bottlenecks in the application. This thesis explores this idea in five parts: 1) identifying key metrics for analysis models 2) identifying relevant microbenchmarks to aid in calculating thresholds dynamically 3) Analyse different application runtime data for performance bottlenecks 4) Explore different methods to calculate dynamic threshold values for different metrics 5) Propose an idea for the structure of the finished analysis tool and create a prototype of the tool based on the proposed idea.

Zusammenfassung

HPC-Anwendungen können lediglich 15-20% der theoretischen Höchstleistung moderner Supercomputer erreichen. Traditionell ist die Leistungsanalyse von HPC-Anwendungen stark von den Entwicklern abhängig. Dem Entwickler werden Daten mit verschiedenen Metrik-Werten zur Verfügung gestellt und anhand dieser Werte muss der Entwickler sowohl die möglichen Leistungsengpässe in der Anwendung beurteilen als auch mögliche Optimierungstechniken finden, um diesen Engpässen zu begegnen. Die Verbindung zwischen Analyse und Optimierung entsteht durch das Ad-hoc-Wissen der HPC-Entwickler. [Gra19] Ziel dieser Arbeit ist es, ein Konzept für ein anwendungsunabhängiges Analysewerkzeug zu erstellen, das Laufzeitdaten auf beliebigen Systemarchitekturen auf Leistungsengpässe hin analysieren kann. Ein Blackbox-Tool, das vordefinierte oder benutzerdefinierte Analysemodelle verwendet, um die während eines Anwendungslaufs erzeugten Leistungsdaten zu untersuchen, um das Vorhandensein und die Art von Engpässen in der Anwendung anzuzeigen. Diese Arbeit untersucht diese Idee in fünf Teilen: 1) Identifizierung von Schlüsselmetriken für Analysemodelle 2) Identifizierung relevanter Mikrobenchmarks zur Unterstützung der dynamischen Berechnung von Schwellenwerten 3) Analyse verschiedener Anwendungslaufzeitdaten auf Leistungsengpässe 4) Erforschung verschiedener Methoden zur Berechnung dynamischer Schwellenwerte für verschiedene Metriken 5) Vorschlag einer Idee für die Struktur des fertigen Analysewerkzeugs und Erstellung eines Prototyps auf der Grundlage der vorgeschlagenen Idee.

Contents

Acknowledgements vii

Abstract ix

Zusammenfassung xi

I. Introduction, Idea and Objectives 1

1. Introduction 2
1.1. Idea . . . 3
1.2. Objectives of this thesis . . . 4
1.3. Structure of this thesis . . . 4

II. Related Work 5

2. Related work 6
2.1. Intrusive tools . . . 6
2.1.1. ScoreP . . . 6
2.1.2. TAU . . . 6
2.1.3. Periscope . . . 7
2.2. Non-intrusive tools . . . 7
2.2.1. LIKWID . . . 7
2.2.2. HPCToolkit . . . 7
2.2.3. PerfExpert . . . 8
2.2.4. Megware's Continuous Benchmarking framework . . . 8
2.2.5. PerSyst . . . 9

III. Background and Proposed Idea 11

3. Performance Bottlenecks 12
3.1. Memory bound . . . 12
3.1.1. Strided access . . . 13
3.1.2. Locality of reference . . . 13
3.2. CPU bound . . . 14
3.2.1. Vector operations vs Scalar operations . . . 14
3.2.2. Branch misprediction . . . 15

3.2.3. Expensive instructions . . . 16
3.2.4. Single precision vs Double precision . . . 16
3.2.5. Object instantiation . . . 17
3.3. I/O bound . . . 17

4. Microbenchmarks **18**
4.1. STREAM . . . 18
4.2. Interleaved or Random (IOR) . . . 19
4.3. High Performance Linpack (HPL) . . . 19
4.4. Likwid-bench . . . 20
4.5. Intel MPI Benchmarks . . . 20

5. Proposed Idea **21**
5.1. Performance metrics . . . 21
5.2. Analysis models . . . 24
5.3. Thresholds . . . 26
5.4. Analysis of bottlenecks . . . 26
5.5. Severity . . . 27
5.6. Structure of the analysis tool . . . 28

IV. Analysis results and Tool Prototype **29**

6. Analysis results **30**
6.1. Applications studied . . . 30
6.1.1. LULESH . . . 30
6.1.2. KRIPKE . . . 31
6.1.3. 3-D Lid Driven Cavity . . . 31
6.1.4. MaMiCo . . . 32
6.1.5. Switch case . . . 32
6.1.6. Expensive Instructions . . . 32
6.2. Memory Bound Analysis . . . 33
6.3. CPU Bound Analysis . . . 37
6.3.1. Poor Vectorization . . . 38
6.3.2. High Branching Misprediction . . . 39
6.3.3. Expensive Instructions . . . 40
6.3.4. Object Instantiation . . . 41
6.3.5. Proposed analysis model . . . 41
6.4. Tool Prototype . . . 41
6.4.1. Interface . . . 41
6.4.2. Usage . . . 44
6.4.3. Critical analysis . . . 46

V. Conclusion and Future Work **50**

7. Conclusion and Future Work **51**

VI. Appendix **52**

A. Programs used to recreate bottleneck behaviour **53**
A.1. Switch Case 53
A.2. Expensive Instructions 54
A.3. Analysis strategies suggest in PerSyst [GC15] 55

Bibliography **59**

Part I.

Introduction, Idea and Objectives

1. Introduction

In modern day High performance computing (HPC), architectures and applications are developing at a very high rate. Almost every year new processors are brought to market by companies like Intel and AMD promising better processing capabilities [bus][amd]. Processors vary in terms of the number of cores, number of threads, memory hierarchy [amd]. Similarly there are several different parallel input-output (I/O) systems such as Lustre [S+03] and BeeGFS [Hei14]. There has also been a very drastic increase in the Graphic processing unit (GPU) technology in recent times. The CPU-GPU processing combination increases processing speeds exponentially compared to multi-core systems without and GPUs [Bly08]. Performance of both hardware capabilities such as multicore processors, cache, varied architectures, inter and intra node connectivity developments, and software capabilities such as compilers, algorithms, libraries have been improving dramatically; however, it is not necessarily reflected in the increase in application performance. There are several possible reasons for this such as problems in scaling the application with respect to the architecture, problems in the application source code causing inefficient behaviour, limited by system I/O speed, etc [ARR17][HYSW17]. To be able to properly analyse an HPC application for different performance bottlenecks to improve performance on different supercomputers is thus very crucial and very typical.

The process of performance analysis of HPC applications would require someone to have in-depth knowledge about two things: the application source code knowledge and the required architecture dependent knowledge on which the application is running. Burtscher et al. in [BKD+10] have described the general workflow of a code optimisation process with generic workflow tools as follows:

- Selecting relevant performance counters.
- Running multiple measurements.
- Collecting performance data.
- Identifying performance bottlenecks based on collected data.
- Finding different optimisation methods for detected bottlenecks.
- Implementing selected methods of optimisation.

The steps of optimization stated above would traditionally be done manually and the process can be very tedious and time consuming. Developers are expected to have architecture dependent knowledge and also have to have the skillset to use this knowledge to analyse the recorded performance metric data for performance bottlenecks issues. They need to know which metric values are relevant for their purpose and what should be their ideal values as compared to what was recorded.

There are several tools that help developers record and/or analyse performance metrics. These tools can be broadly classified as intrusive and non-intrusive. Tools like ScoreP

[KRB+12], TAU [MMCS10], Valgrind [NS07] are a few examples of tools that use code instrumentation to record performance metrics. Instrumentation can be done either statically by adding instrumentation code at source code level or dynamically by adding instrumentation code at binary level. These methods tend to increase the size of the application binaries which can modify system behaviour and can also affect the flow of instructions. Whereas tools like Likwid [THW10], HPCToolkit [MMCS10] help record performance metrics non-intrusively. The application is not tampered with in any way in this method.

There are several tools that work on automating the process of analysis such as PerfExpert [BKD+10], Periscope [MFG16], PerSyst [GC15], TAU [MMCS10], etc. which help users and developers access the performance of HPC applications at system and application source code level. One common underlying design objective of any automated analysis tool is to decrease the dependence upon the "expert knowledge" required for performance bottleneck analysis of HPC applications at any level. PerSyst does it at system level whereas PerfExpert and Periscope perform it at source code level. This is essential to decrease the entry level barrier faced by different HPC application developers and users.

1.1. Idea

The idea behind this thesis is to create a performance bottleneck analysis tool for HPC applications that takes into account system benchmarks and hardware properties to analyse performance metrics using analysis models and dynamic thresholds. This analysis tool would be an extension of Megware's Continuous Benchmarking (CB) Framework [Tip21], which is used to record runtime performance metrics data at system level.

The proposed idea is that the tool uses system architecture information, like cache structure and size, frequency, etc. and several microbenchmarks such as STREAM, likwid-bench, IOR, etc. to generate important information related to various performance metrics, such as sustainable peak memory and cache bandwidths, floating point operations (FLOPS), etc. are together used to obtain system capability information based on which the analysis is to take place. The tool would consist of different analysis models, a set of heuristics structured in the structure of a tree, that would help us codify rules to locate bottlenecks. The generated system capability information will be used to calculate metric thresholds which are passed to the analysis models along with the runtime performance metric data to detect bottlenecked regions in the provided data and the type of bottleneck detected. The tool would essentially serve as a blackbox, a developer would just deploy their application through the framework and performance results would be generated marking bottlenecked regions and possible reasons for those bottlenecks.

The novelty of our proposed idea lies in the fact that the tool correlates several different performance metric data to possible bottlenecks using analysis models with threshold values of different performance metrics that are dynamic in nature and thus essentially creating a blackbox tool that allows users and developers to analyse any sort of application on any system.

1.2. Objectives of this thesis

This thesis is intended to explore different aspects of the proposed idea of creating such a non-intrusive analysis tool using the data recorded by the CB framework. The idea suggested above is explored in 5 main parts in this thesis to lay a basic groundwork:

1. Identifying key performance metrics. It is important that the metrics we record and use for our analysis are generic. Metrics that would be relevant for analysis regardless of the type of the application and the system architecture.
2. Identifying different microbenchmarks that can help to generate sustainable peak performance values for many different metrics.
3. Analyse different application runtime to find patterns between different metrics pertaining to different bottlenecks.
4. Explore different methods to calculate dynamic threshold values for different metrics.
5. Propose an idea for the structure of the finished analysis tool and create a prototype of the tool based on the proposed idea.

1.3. Structure of this thesis

The rest of the thesis is divided into 4 parts. Part 2 discusses the different tools that are presently used to monitor and/or analyse HPC applications. Part 3 is divided into 3 chapters. The first chapter discusses common performance bottlenecks faced by HPC applications with respect to memory, CPU performance and I/O related bottlenecks. The second chapter explains what microbenchmarks mean and discusses different microbenchmarks. The third chapter outlines the idea of the performance analysis tool. The next part explains the process of analysis, the applications used to gather runtime data for the analysis, the results derived from this analysis and it also discusses the prototype of the suggested analysis tool. This last part concludes this thesis and provides a direction for future work on this project.

Part II.

Related Work

2. Related work

Performance monitoring and analysis tools can be broadly classified into two types - intrusive and non-intrusive.

2.1. Intrusive tools

Intrusive tools use code instrumentation at different levels [GC15]. At source code level, function calls are added to the application that help generate trace information for the application. Other methods such as library wrapping, which use wrapper libraries instead of original libraries. These wrapper libraries have modified routines that contain instrumented code which call the original routines. [MMCS10] Dynamic binary instrumentation is also a common intrusive method to generate performance metrics. [RH01] It updates, modifies or removes instruction from the generated binary code to allow instrumentation. Dynamic code is injected at execution time and the application does not need to be recompiled. This method is especially helpful when the source code is unavailable to the user. [RH01] Users need to be careful of the structural modifications of the application while testing their applications using such intrusive methods. They also add overheads due to the added instructions which can modify application behaviour as well in some cases. [AR01]

A few key intrusive performance monitoring and analysis tools are explained below.

2.1.1. ScoreP

Score-P is a performance measurement framework that generates profile and event trace information using code instrumentation. Several performance analysis tools such as TAU, Vampir, Periscope, etc use ScoreP as their performance measurement framework to generate performance metric data. [KRB+12]

2.1.2. TAU

TAU [MMCS10] is a performance monitoring and analysis tool for HPC applications that uses code instrumentation to generate performance data for any application. Performance events of interest must be determined for the application before any performance experiment is conducted.

TAU comes with a parallel profile analysis tool called ParaProf which allows the user to visualise and generate statistical data for the application run using the performance metrics generated by the tool. There is also a performance data mining tool called PerfExplorer. This tool uses techniques like clustering and dimensionality reduction to manage the large scale performance data generated by the tool to generate relevant data relationships between different metrics using comparative and correlation analysis of these metrics.

2.1.3. Periscope

Periscope is an automated performance tuning framework for HPC applications. [MFG16] It uses intrusive measurement framework ScoreP to generate the metric data. Periscope's performance analysis derives information about an application's execution in the form of performance properties such as load imbalance, communication, cache misses, redundant computations, etc. It uses conditions along with a confidence value (between 0-1) to check for the existence of different performance properties. It also assigns a severity value to all the different performance properties recorded. Higher the severity, more significant the performance property.

A performance property is considered a performance bottleneck by the tool if and only if the severity is over a tool defined threshold value. The tool uses these performance properties in predefined analysis strategies assembled together in a tree-like structure to create its analysis models. System monitoring tools like PerSyst [GC15] have incorporated a similar idea for their work to analyse system wide performance metrics.

2.2. Non-intrusive tools

Non-intrusive performance monitoring methods do not require any sort of code instrumentation and use hardware performance counters to generate performance metrics. This method helps avoid adding more work on the application developer who would have to add instrumentation code to the application to allow any sort of monitoring. This also helps avoid any unwanted side effects that can be caused because of code instrumentation. [GC15]

A few key non-intrusive performance monitoring and analysis tools are explained below.

2.2.1. LIKWID

LIKWID is a performance oriented tool that provides a set of lightweight command-line utilities. [THW10] LIKWID is divided into multiple tools some of which are explained below:

- `likwid-topology` - provides thread, cache and NUMA topology.
- `likwid-perfctr` - provides hardware performance counters.
- `likwid-pin` - enforces thread-core affinity in a multithreaded application without modifying the source code.
- `likwid-bench` - provides micro benchmarks for several CPU architectures.

It uses processor-specific hardware registers, also called model specific registers (MSR) to measure hardware performance counters. This is performed non-intrusively, without touching the source code at any level. [THW10] PerSyst [GC15] and CB framework [Tip21] both use LIKWID to record performance metrics.

2.2.2. HPCToolkit

HPCToolkit is a set of performance observation and analysis tools for HPC applications. It is based on the idea that data collection should be non-intrusive to avoid unnecessary overheads

for which it uses asynchronous sampling to record performance metric data, it should be language independent and thus works directly with application binaries, and should be able to record a varied number of metrics to avoid less informed analysis. [MMCS10]

The application is first run through the framework and data is collected using asynchronous sampling. Then the tool analyses the application binary to understand the structure of the application with regards to information about files, functions, loops and inlined code. The recorded data and application's structure are combined to create a performance database which links different code sections to their corresponding performance metric data. The tool allows the user to interactively view and analyse the performance database generated in the form of a hierarchical top-down graphical user interface.

2.2.3. PerfExpert

PerfExpert is an automated performance analysis tool for HPC applications which identifies, characterises and suggests solutions to tackle core, chip and node level performance bottlenecks present in the application. [BKD+10] As system architecture knowledge plays a crucial role in analysis of parallel applications, the tool consists of embedded expert knowledge of the system architecture that allows it to identify the relevant performance counters to measure during the application run.

PerfExpert is built on top of the HPCToolkit, which uses asynchronous sampling to monitor performance metrics. This is an unobtrusive way to monitor application execution based on recurring sample triggers. [MMCS10] Thus, no sort of instrumentation is required to record performance metrics for analysis.

PerfExpert introduces a local cycle-per-instruction (LCPI) metric to measure the runtime of different code sections. This metric is used to focus optimization efforts of code sections with higher LCPI value. Only if the LCPI value crosses a predefined threshold value, is the code region considered for bottleneck detection.

2.2.4. Megware's Continuous Benchmarking framework

The CB framework presented in [Tip21] allows users to run jobs and non-intrusively collect important performance metrics at system level with a user defined granularity (≥ 5 seconds). The framework provides several metrics related to CPU, I/O, memory, GPU, interconnect and energy consumption. Most of the high level metrics are collected by reading, parsing and calculating them from the proc filesystem. For other low level metrics the framework uses the LIKWID, a command line tool suite that works for Intel, AMD, ARMv8 and POWER9 processors on the Linux operating system. [THW10]

The framework maintains a low measurement overhead which makes sure that the monitoring is done with minimal impact on runtime. The framework also records system data, such as NUMA, cache and thread topology using likwid-topology.

The metrics recorded by the framework are:

- CPU usage metrics - total, system, user, iowait, idle.
- CPU utilisation metrics - Single and double precision FLOPS, vectorisation ratio, branch misprediction rate/ratio, clock speed, CPI, loads to store ratio, operational intensity, AVX single/double precision etc.

- I/O - Utilisation, read/write size, average read/write request size, average read/write wait time, etc.
- Memory - memory bandwidth, memory read/write bandwidth, cache level data - bandwidth, data volume, miss rate/ratio, etc.
- Energy - CPU energy, CPU power, power.
- Interconnect - throughput, errors and dropped packages, transmitted data volume.

Several different performance metrics can be monitored by the tool. The user can configure the metrics it wants to monitor and record and after the application runtime, all of this data is recorded and stored in the database. All the results are displayed on the frontend in the form of several runtime charts displaying the performance metric values over the complete runtime.

This thesis explores the idea of extending the CB framework to add an analysis component to the framework itself.

2.2.5. PerSyst

PerSyst is a system wide on-line analysis tool for all applications running on a supercomputer [GC15]. The tool is used to detect inefficient use of system capabilities by all the different applications running simultaneously. The data is collected non-intrusively using LIKWID.

The LIKWID tool can be used to measure system level performance metrics without any instrumentation. It uses Model-Specific Registers found on Intel and AMD processors. It records data for selected performance groups from a set of predefined groups. The tool uses the properties recorded by its measurement tool, with a set of codified strategies. These strategies are based on expert knowledge and are encoded in a tree-like structure. These strategies are used to filter through data to determine if further analysis is necessary based on the performance metrics encountered while traversing the tree from root to leaf.

The strategy maps mentioned in [GC15] analyse for memory bound, compute bound and I/O bound behaviour. They also look into bottlenecks emerging due to load imbalance and network connectivity.

The decision for further analysis or the identification of a bottleneck is done using thresholds. These thresholds also help evaluate the severity of an identified bottleneck.

Carla, et al. in [GC15] talk about different heuristics, based on which these thresholds can be derived:

- It can be derived based on hardware characteristics and expert knowledge.
- It can be based on a benchmark.
- It can be chosen at a point where changing the property value wouldn't correlate to better performance anymore.
- It can be chosen using statistical data collected over a certain period of time.

I use the ideas presented in [GC15] and [BKD+10] to propose an analysis process and tool in this thesis. Both of these tools function as a blackbox wherein the analysis process is handled completely by the tool without any input from the user. The PerSyst tool utilises likwid to measure performance metrics but is limited to monitoring only at a system level

whereas the PerfExpert tool analyses applications at source code level. The idea suggested in this thesis is based on similar guiding principles but the process of analysis is intended to work on any system architecture and analyse metrics for specific applications rather than a complete system. By using different microbenchmarks the tool should be able to identify threshold values automatically for certain metrics like memory bandwidth and FLOP rate. The proposed analysis models would also be designed to work on all types of applications. The intention is to create a truly application agnostic analysis tool that works like a blackbox to analyse application runtime performance metric data on any system architecture.

Part III.

Background and Proposed Idea

3. Performance Bottlenecks

As per [dic], bottleneck is "a place or stage in a process at which progress is impeded". There can be several reasons for an HPC application to suffer inefficiencies. They can be caused by limited resources, such as limited cache sizes, low FLOP rate of the CPU, latency of the communication hardware [ARR17]. This can also be caused by inefficient use of resources like caching, unconsolidated read write access [HYSW17], strided accesses [BMK+99], expensive instructions [CRON+14], heavy branch misprediction [ESE06], etc.

For the purpose of this thesis I have broadly categorised bottlenecks as: Memory related bottlenecks, CPU related bottlenecks, I/O related bottlenecks.

3.1. Memory bound

An HPC application is said to be memory bound when the majority of the execution time is spent in memory transfers. [PGB14] In this case, processor frequency scaling has little or no impact on performance and increasing core count becomes the most effective way to improve HPC application run-time.

Memory latency can be considered to be a serious bottleneck when the memory bandwidth of the system during the application run crosses 80% of the sustainable memory bandwidth. [ARR17] As shown in figure 3.1, memory latency is almost constant when memory bandwidth lies between 0-40% and increases almost linearly when memory bandwidth lies between 40%-80% of the sustainable memory bandwidth. But there is an exponential increase in memory latency values when memory bandwidth is over 80%.

Application memory bandwidth starts to affect memory latency and overall performance when it rises above 40% of the sustained memory bandwidth. The collisions between

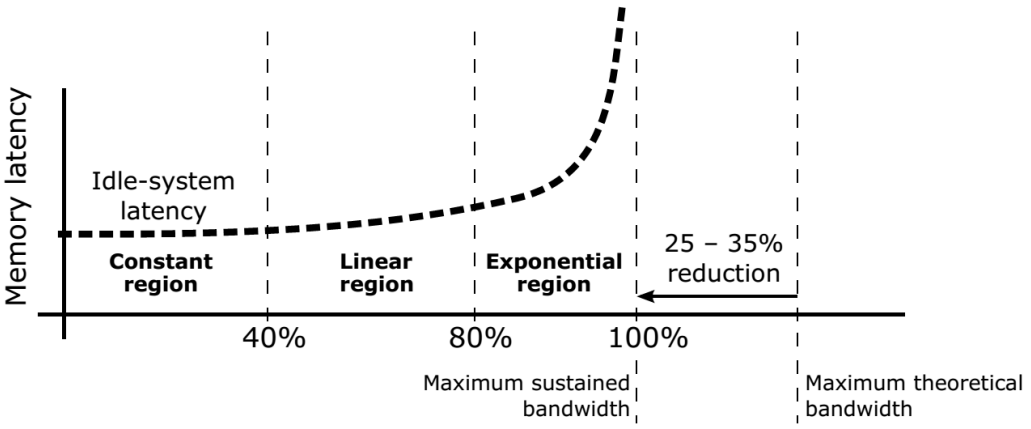


Figure 3.1.: Memory access latency versus used memory bandwidth Source: [ARR17]

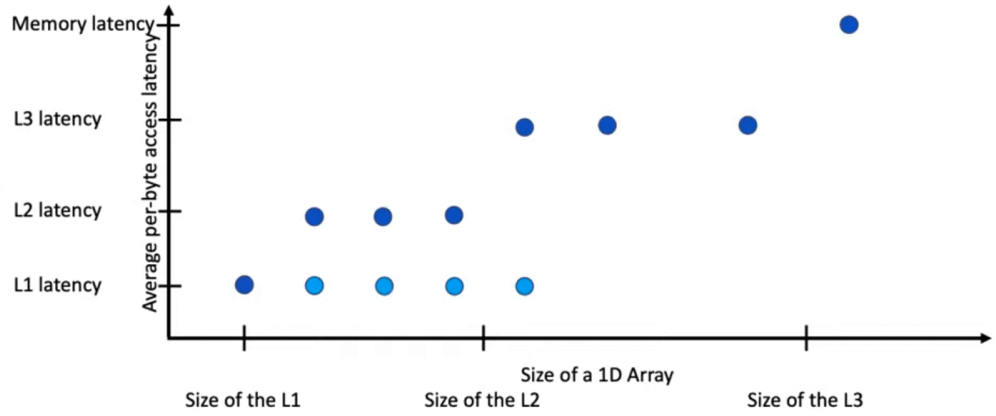


Figure 3.2.: Memory latency vs size of a 1D array Source: [Sel]

concurrent memory requests cause a significant increase in memory delay when the program uses more than 80% of the sustained memory bandwidth.

The theoretical maximum memory bandwidth of a system is calculated by multiplying the memory frequency, the number of bytes of width and number of channels supported for the processor. [the]

In practice, there are many system variables that limit the practically achievable maximum memory bandwidth, such as software workloads and system power states [the]. Although theoretical memory bandwidth gives us an idea of the ideal memory throughput power of the system, peak sustainable memory bandwidth is usually around 65-75% of the theoretical peak memory bandwidth of any system [ARR17]. There are several benchmarks that can help us find the sustainable peak memory bandwidth of any system.

Two of the most common factors that make an application memory bound are strided access and inefficient data locality.

3.1.1. Strided access

As defined in [acc] strided access is when a series of addresses are accessed with a uniform skip between each referenced address. For instance, sequence 1, 11, 21, 31, 41, ... is a strided access pattern with a stride of +10. Strided access can occur due to several reasons such as when structures are packed into an array and only a particular variable is accessed for every element in the array, during matrix computations, etc.

In a scenario where we read every element of an array, the memory latency would be bounded by the L1 latency irrespective of the size of the array. In cases where we have a strided access of N where N is roughly the size of the L1 cache, we would see that if the size of the array is also roughly close to N (<N), memory latency would still be bounded by the L1 latency, but if the size of the array increases (>N), we would see that the memory latency would be bounded by subsequent memory hierarchy latencies (L2, L3, Memory).

3.1.2. Locality of reference

There are two important types of locality of reference: Spatial and Temporal.

Spatial locality refers to the likely hood of an application to reference data present close to a recently referenced data. [JWN10] With this type of locality, access latency can be significantly reduced by retrieving multiple data items in a single fetch cycle. Poor spatial locality would result in multiple fetch cycles to retrieve the relevant data items, which would result in delay due to higher access latency. [Vor13]

Temporal locality refers to the likely hood of an application to refer a recently accessed data again in the near future. This is the founding principle behind caches. [JWN10] Inefficient temporal locality would mean that data items whcih could have been reused are lost from cache memory and must be fetched again. [Vor13]

Poor locality of reference (of either type) would result in longer execution time for the application with higher access latency.

3.2. CPU bound

Hutcheson et al in [HN11], suggest that problems that are not bounded due to memory are often bounded by the compute speed of the processor, which means that performance is constrained by the speed at which the processor can carry out mathematical operations. In such a situation the processor is the reason for the bottleneck as the memory has to wait on the processor.

There can be several different reasons for an application to be compute bound. Low vectorisation, high branch misprediction penalties [ESE06][IOT14][Che00], high number of expensive floating point operations [Fog06], excessive object instantiations[THW12][GC15].

3.2.1. Vector operations vs Scalar operations

Consider the example code given below:

```
1  void add( double* result, const double* a, const double* b, size_t size)
2  {
3      size_t i = 0;
4      // AVX-512 loop
5      for( ; i < (size & ~0x7); i += 8)
6      {
7          const __m512d kA8   = _mm512_load_pd( &a[i] );
8          const __m512d kB8   = _mm512_load_pd( &b[i] );
9          const __m512d kRes = _mm512_add_pd( kA8, kB8 );
10         _mm512_stream_pd( &res[i], kRes );
11     }
12     // Serial loop
13     for( ; i < size; i++ )
14     {
15         result[i] = a[i] + b[i];
16     }
17 }
```

Listing 3.1: Example add function that adds two double precision arrays and saves it to the result array in 2 ways - using scalar and vector operation. [For14]

The above example code performs addition of 2 arrays a and b and stores them in the result. There are 2 for loops each performing the same computation using:

- AVX-512 instruction set - 512-bit Advance vector extensions (AVX) instruction set for x86 architectures proposed by Intel. [cpp] The double precision arrays a and b are loaded onto 2 __m512d kA8 and kB8 respectively, which can hold a vector of 8 64-bit double precision floating-point values each. These two vectors are then added using the _mm512_add_pd intrinsic add operation which takes in kA8 and kB8 and returns a single __m512d vector stored in kRes. At every iteration 8 elements of the array a are added with 8 elements of array b simultaneously. [cpp] AVX and SSE2 are two more instruction sets for x86 architectures that enable 256-bit and 128-bit operations respectively. The stride value, in our case the value of i, would be 4 in the case of AVX and 2 in the case of SSE2 instruction sets.
- Serial addition - In this loop each element of array a and array b are individually added to each other in every iteration of the loop.

The first case is an example of a vectorised loop. In this case two registers of size 512-bit are performing the same operation of each element of one vector with the corresponding element of the other vector. __mm512_add_pd() is one of the Intrinsics for Arithmetic Operations which is used here to perform the add operation. As explained in the Intel developer guide [cpp], "Intrinsics are assembly-coded functions that let you use C++ function calls and variables in place of assembly instructions." In a situation where the size of the arrays is not a multiple of the array stride i (= 8,4,2) the remaining operations are conducted serially. This means that performance increases by almost 8, 4 and 2 times respectively in each case with respect to the performance achieved when each element is added sequentially.

If an application is performing more scalar operations than vector operations then it would not be able to use the full potential of the processing unit which could result in a bottleneck. As suggested in [GC15], in such a situation the applications data level parallelism should be revisited and architecture specific vector operations should be utilised.

3.2.2. Branch misprediction

Deep pipelining was developed to increase compute capabilities of processors. [Che00] In the process of instruction pipelining, the pipeline is divided into multiple stages to speed up the clock. Each stage contains lesser logic and thus runs faster. [HH15] It has been one of the most effective ways to improve processing speed but at higher levels of instruction parallelism there are times when processing speed stalls. One of the reasons for these stalls are unresolved branches. The target address cannot be fetched until the target address has been computed and the branch is resolved. If this process takes multiple clock cycles, it tends to create bubbles in the pipeline. [Che00]

Branch predictors were introduced to deal with this issue. There are static branch prediction schemes where the system predicts whether all branches are taken or not taken. There are also dynamic branch prediction schemes where the the runtime history of the branch is taken into account to predict its branching behaviour. [Che00] These methods have increased performance by decreasing the amount of pipeline bubbles caused due to branching, but as neither of the predictive models are accurate, every wrong branch prediction, branch misprediction, results in lost clock cycles. The hardware typically loses more than ten clock

cycles to remove the instructions from the predicted branch and resume the execution for the actual branch. [IOT14]

Eyerman et al in [ESE06] conduct a study to characterise branch misprediction penalty where they have identified and quantified 5 major contributors to the branch misprediction penalty: the frontend pipeline length, number of instructions since a miss event, such as branch misprediction, instruction cache miss or long data cache miss, instruction level parallelism of the application, functional unit latencies and the number of short data cache (L1D) misses. In this study it was found that programs with more non-branch miss events resulted in lesser misprediction penalties as compared to programs with more branch miss events. Results also showed that programs with lower level of instruction level parallelism showed higher branch misprediction penalties as compared to programs with higher instruction level parallelism. They also show that programs with a higher fraction of L1 D-cache misses showed higher branch misprediction penalties.

Inoue et al in [IOT14], show a simple yet elegant way of speeding up set intersection by decreasing algorithms branch misprediction penalty. They utilise the processor's SIMD, single instruction multiple data, vectorisation capabilities which help in decreasing the number of branches to be computed significantly. Such methods are an intelligent way to deal with bottlenecks created due to branch misprediction.

3.2.3. Expensive instructions

Different assembly level instructions have different ops and latency. Based on these values one can judge which instruction is expensive and which isn't. In this case ops mean the number of macro-operations issues from instruction decoder to the schedulers and latency is the delay caused to process the instruction. For example, on the Intel Skylake processors the ops value ranges from 1-3 and the latency value ranges from 3-4 for an `IMUL` (signed multiplication) instruction. Whereas for an `IDIV` (signed division) instruction the ops value ranges from 10-57 and the latency value ranges from 23-95. These values vary depending upon the size of the registers used. [Fog06]

If your application uses a lot of expensive instructions, it will result in higher latency and more ops that need to be executed. It would increase computation load on the processor while executing lesser instructions per cycle. Applications with a higher number of expensive instructions thus face a compute bound bottleneck.

3.2.4. Single precision vs Double precision

In computer architectures, 32-bit floating point numbers are called single precision and 64-bit floating point numbers are called double precision. Double precision values provide more than 2 as much precision as compared to single precision values, but computations on these double precision numbers also take longer time comparatively. [HH15] Many architectures allow two single precision floating point operations in place of one double precision floating point operation.

In compute bound applications, using more single precision floating point operations could help speed up the processing time at the risk of losing higher precision.

3.2.5. Object instantiation

In object oriented programming, object instantiation can add more latency not observed with in-built types. [GC15] Excessive object instantiation can cause a bottleneck because of the added latency. Low FLOPS rate with low clocks per instruction rate (CPI) can be observed in such situations. [THW12][GC15] In such a situation a developer should try to decrease the number of objects in their application and check if it increases performance.

3.3. I/O bound

Historical trends of HPC systems show that processor performance and memory management has been improving at a much higher rate as compared to I/O devices [HP11]. Although modern day solid state drives help mitigate this gap in performance to quite an extent, but, it is still not enough to keep up with present day processor technology, with the increasing and more efficient usage of GPUs in modern day HPC Systems. This gap between better memory and much better processing power is the main reason for applications to display I/O bound behaviour. Because of this it becomes very crucial for users to manage the I/O behaviour of their applications.

Paul et al. [PFM+20] have shown that write intensive applications tend to decrease in efficiency when they write a lot of data but the bytes of data written per call is very low. They state that inefficiency created due to I/O operations in several applications (running in the same system) showed a common trend - total amount of data written by an application is more than the mean of amount of data written by all applications while the bytes written per call is less than the mean of the bytes written per call for all applications. Whereas, efficient write intensive applications tend to have both the total amount of bytes written and the bytes written per call are both more than their respective mean values across multiple applications.

They also observe a positive correlation between the number of metadata operations and the amount of data written. Which means that more the amount of data is written, more is the stress on the metadata servers.

I/O contention can also limit your application from running faster as the processor has to wait longer to access the disk. This problem cannot be mitigated at the application level as multiple I/O intensive applications can be running at the same time on the system increasing I/O latency per application.

4. Microbenchmarks

According to HPCWiki [mic], "Microbenchmarking is about measuring the time or performance of small to very small building blocks of real programs. This can be a common data access pattern, a sequence of operations or even a single instruction."

A microbenchmark is a program or routine used to measure and test the performance of a single component such as memory bandwidth, I/O speed, latency or FLOP rate. [SZ19] Generally, microbenchmarks are small kernels that mimic real world applications and measure performance counters against these kernels. Kernels that perform arithmetic operations like matrix-matrix multiplication [GFG12], dense linear systems in double precision [Pet04]. Microbenchmarking helps developers understand system capabilities and develop or modify applications to work more efficiently.

There are several microbenchmarks available that deal with different aspects of a system's capabilities. A few important microbenchmarks are covered below.

4.1. STREAM

STREAM is an industry standard, simple microbenchmark that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. [M+95] To ensure that the results are (supposedly) more representative of the performance of very big, vector-style programs, the microbenchmark was specifically created to work with datasets substantially larger than the available cache on any given system. This microbenchmark is intended to measure the bandwidth from main memory so the general rule while running the STREAM microbenchmark is that each array must have at least 1 million elements or 4 times the total size of all last-level caches used in the run, whichever is larger. This makes sure that the data is not cacheable and memory is accessed.

STREAM consists of 4 kernels that are run in a loop and the best results obtained out of multiple runs (usually around 10 runs) are chosen. These 4 kernels are:

- `Copy` $< a(i) = b(i) >$ - this kernel executes a simple copy expression. The uses of `16 bytes/iteration` and `0 FLOPS/iteration`.
- `Scale` $< a(i) = q * b(i) >$ - this kernel multiplies a scalar to every array element and saves the value to the corresponding index of another array. The uses of `16 bytes/iteration` and `1 FLOPS/iteration`.
- `Sum` $< a(i) = b(i) + c(i) >$ - this kernel adds elements of 2 arrays and saves the value to another array. The uses of `24 bytes/iteration` and `1 FLOPS/iteration`.
- `Triad` $< a(i) = b(i) + q * c(i) >$ - this kernel scales the value of each element of one array by a constant factor and adds it to the corresponding element of another array. The result is then stored into the corresponding index of another array. The uses of `24 bytes/iteration` and `2 FLOPS/iteration`.

As it can be seen from the list above, the Triad kernel utilises the most number of system resources per iteration, because of which the values recorded by the Triad kernel are usually considered to benchmark any machine with respect to the sustainable peak memory bandwidth of that.

4.2. Interleaved or Random (IOR)

As explained in [iorb], IOR is a parallel I/O benchmark that measures the performance of parallel storage systems using various interfaces and access patterns. It is a generic parallel I/O microbenchmark that can be run on any POSIX-compliant file system [iora], but it does require a fully installed and configured file system implementation in order to run. It uses MPI to manage the process synchronisation as generally there are many IOR processes running in parallel across several nodes in an HPC system.

IOR benchmark runs 2 tests, one each for write and read operations and it measures the following values for each test:

- Read/write bandwidth measured in `MB/sec`
- Input/output Operations Per Second (`IOPS`)
- Latency

The repository also provides the `mdtest` benchmark which helps in measuring the peak metadata rates of storage systems under varied directory structures. This microbenchmark also uses MPI to manage its parallel synchronisation.

4.3. High Performance Linpack (HPL)

As described by Petitet in [Pet04], "HPL is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers." It is a portable and freely available implementation of the High Performance Computing Linpack Benchmark. It relies upon an efficient implementation of the Basic Linear Algebra Subprograms (BLAS). It generates a linear system of equations in the order of n and solves this system of linear equations using the lower-upper (LU) decomposition with partial row pivoting. It requires installed implementations of MPI and either BLAS or VSIPL to run. [Don87] It is a highly scalable and efficient benchmarking tool that has been used for decades as a metric for compiling the Top500 list which maintains a list of the top 500 supercomputers around the world.

HPL reports the following metrics for a system:

- R_{max} - the performance (in `GFLOPS`) of the largest problem run on the system.
- N_{max} - the size of the largest problem that ran on the system.
- $N_{1/2}$ - the size where half the R_{max} execution rate is achieved.
- R_{peak} - the theoretical peak performance (in `GFLOPS`) for that system.

4.4. Likwid-bench

Likwid-bench [RHa] is an assembly microkernel benchmark suite that contains several kernels that can be used to benchmark any computer system on the basis of memory bandwidths and instruction throughput for specific instruction code on x86 systems. This benchmark suite provides a list of kernels such as triad, copy, stream, loads, stores, update, etc. Each kernel considers an exhaustive list of different combinations of precision type (single precision and double precision) and different instruction sets such as AVX, Fused multiply-add (FMA) and Stream SIMD extensions (SSE).

Likwid-bench can be used to measure these performance metrics for different thread affinity domains, meaning it can be used to measure metric values when running a kernel on all threads simultaneously or specific thread domains.

4.5. Intel MPI Benchmarks

The Intel MPI (message passing interface) Benchmark is used to measure MPI performance measurements for point-to-point and global communication operations for message sizes or varied ranges. The generated benchmark data characterises the performance of a cluster system, including node performance, network latency, and throughput and the efficiency of the implemented MPI protocols. [mpi]

5. Proposed Idea

As explained in section 1, this thesis explores the idea of extending the functionality of the CB Framework mentioned above. Adding an analysis component on top of the measurement capabilities of the framework. The proposed idea of the analysis tool is based on a few key guiding principles:

- The tool should essentially work as a blackbox. The traditional optimisation efforts need a lot of expert knowledge to be able to find inefficient behaviour of HPC applications. To be able to circumvent this, it is essential that the tool works as a blackbox wherein the developer just runs the HPC application through the framework and they would get an analysis report on the existing inefficiencies in the application.
- The tool should be system and application agnostic. Since there are many different types of processors, file systems, compilers, etc available and different supercomputers are built to cater to specific needs. There is a need for a tool that allows the user to be able to analyse any kind of applications on any system.
- Since the process of measuring performance metrics is non-intrusive, the analysis would be in the form of a general recommendation since application level behaviour such as execution times of different loops or functions, etc would be unavailable to us.
- The analysis models should be extendable, i.e the developer can add custom analysis models to the preexisting knowledge base to analyse for certain specific behaviours.

The tool can be broken down into 5 key aspects - performance metrics, analysis models, dynamic thresholds, bottlenecks the tool can identify and the severity of those bottlenecks.

5.1. Performance metrics

The CB framework provides us with an extensive list of metrics. For our purpose it is really important that the tool can identify a bottlenecked region just by parsing through the recorded runtime data. For this, we need to identify a few important metrics that are indicative of a general type of bottleneck as explained in section 2 above. These metrics are referred to as “key metrics” in the context of this thesis.

For instance, as explained above, when an application’s recorded memory bandwidth is above 80% of the system’s sustainable memory bandwidth, it is indicative of a memory related bottleneck as the amount of latency caused due to memory increases exponentially. Thus, memory bandwidth becomes a “key metric” for the analysis tool. All the regions in the recorded runtime data that show memory bandwidth values above this limit would be facing a bottleneck due to memory operations but what would be a specific reason for this bottleneck would still be unclear. The key metric would point the tool in the direction in which further analysis should be conducted.

The intention here is to break down the process of analysis in such a way that these key metrics become indicative of a performance bottleneck in the recorded runtime data and other metrics help us refine our analysis to a specific reason for the bottlenecked behaviour.

The CB framework monitors a wide variety of performance metrics as mentioned in section 2. These metrics have been split into 7 types for the framework frontend, i.e., CPU, memory, I/O, roofline, cycle, interconnect and energy related metrics. The framework records almost 150 metrics, below is a list of some of the metrics under each type and what they mean: (the list is not exhaustive of all the metrics that can be recorded by the framework)

CPU related metrics:

- Branching related metrics:
  - Branch rate - ratio of the number of retired branch instructions to number of all retired instructions.
  - Branch misprediction rate and ratio - branch misprediction ratio is the ratio of the number of retired branch mispredicted instructions to the number of all retired instructions and branch misprediction rate is the ratio of the number of retired branch mispredicted instructions to the total number of retired branch instructions.
  - Instructions per branch - ratio of the number of all retired instructions to the number of all branch instructions. It is essentially 1/(branch rate).
- CPI - the number of cycles required to execute an average instruction. [HH15]
- FLOPS for single precision and double precision - number of floating point instructions executed per second with single or double precision floating point numbers respectively.
- Load to store ratio - ratio of the total number of retired load operations to the total number of retired store operations.
- Utilisation - This metric records the amount of CPU time spent on various tasks such as system operations, user operations, etc. It also records the amount of CPU time used up while waiting for outstanding disk I/O operations to finish. [Can]
- Vectorisation ratio - ratio of all vector operations to the total number vector and scalar operations.

Memory related metrics:

- Bandwidth - amount of data transferred (read from or stored to) per second at memory and cache levels L2 and L3.
- Data volume - total amount of data transferred to or from memory and cache levels L2 and L3.
- Cache miss rate and ratio - miss ratio is the ratio of the number of times the data was not present in the cache memory when requested to the total number of times the data was searched for in the cache memory. Miss rate is calculated as the percentage of the same ratio.

- Cache request rate - Cache level request rates tell us how data intensive your code is or how many data accesses you have on average per instruction. [RHb] It is calculated as the ratio of the number of data access operations requested at a particular cache level to that of the total number of instructions.
- L2D evict bandwidth and data volume - the evict data volume is the amount of modified cache lines evicted from the L1D cache and the bandwidth is the amount of evicted data per second. Caches can be split into 2 types of caches, for example L1I and L1D respectively are a level 1 instruction and data caches respectively. This type of cache is called a split cache. It is a common practice to split the lowest level of cache into 2 physically different caches. The instruction cache only stores instructions whereas the data cache stores only data. A unified cache, usually higher level caches, do not differentiate between instruction and data and stores everything on the same cache. [Cha20]
- L2D load bandwidth and data volume - the load data volume is the amount of data loaded from the L1-D cache and the bandwidth is the amount of data loaded per second.
- Memory read/write bandwidth and data volume - similarly this metric tells us about the total amount of data read from or written to memory and the speed at which these operations were conducted.
- Memory total, used, free, usage, cached - the framework provides us with information regarding the total amount of memory present in the system and also provides information regarding the amount of memory used, unused, cached, etc. during the runtime. It also provides us a usage metric which is basically the percent of memory used during runtime.

I/O related metrics:

- Average read/write request size - mean value of the size of all read or write requests respectively.
- Average read/write wait time - mean value of the amount of time taken to process every read or write command respectively.
- Read/write requests per second and size - this metric tells us about the number of read or write requests made to the disk per second and the size of the request.
- Average request queue length - The average queue length of the requests that were issued to the device. [Can]
- Merged read requests - The number of read requests merged that were queued to the device. [Can]
- Utilisation - Percentage of elapsed time during which I/O requests were issued to the device. [Can]

Device saturation occurs when this value is close to 100% for devices serving requests serially. The CB framework uses `iostat` to record these metrics. [Tip21] For devices serving requests in parallel, such as `RAID` arrays and modern `SSDs`, this number does not reflect

their performance limits as specified in the `iostat` documentation. [Can]

Roofline related metrics:

- Operations intensity - or arithmetic intensity is the ratio of the amount of arithmetic operations executed to the total amount of data transferred. Measured as FLOPS/byte.
- FLOPS SP/DP (AVX) - number vectorized FLOPS executed on single or double precision integer values respectively. This metric only considers operations executed using the AVX instructions.

Cycle related metrics:

- Cycles without execution - number of clock cycles spent without executing any instructions.
- Cycles without execution due to L1D/L2/memory loads - number of clock cycles spent without executing any instructions due to an outstanding data load operation at various levels of the memory hierarchy.
- Execution stall rate - percentage of the total number of execution stalls to that of the number of cycles when the core was not in HALT state.
- Stalls caused by L1D/L2/memory loads miss and miss rate - number and percentage of execution stalls while a miss demand load is outstanding at different levels of memory hierarchy respectively.
- Total execution stalls - total number of execution stalls.

5.2. Analysis models

The idea of our analysis models is derived from the strategies used in PerSyst [GC15]. The structure of our analysis models is also in the form of a tree, much like the strategies used in PerSyst, where the root node of an analysis model is the "key metric" mentioned above. These analysis models are the codified knowledge base that the tool would use for the process of analysis.

The purpose of the analysis model is to provide a refined analysis of all the bottlenecked regions in the recorded runtime data by filtering such regions using a key metric and then look into several other metrics to refine the analysis of these bottlenecked regions.

An analysis model would start by analysing a key metric at the root node and then further analysing different metrics in the bottlenecked regions, leading to the leaf nodes. A leaf node would either provide a possible reason for the bottleneck, or indicate that it isn't actually a bottleneck after all. Since we are able to record only system level metrics during the applications execution and cannot correlate the runtime metrics to source code level data, the type of analysis can only be speculative in nature.

An analysis model is built using a deeper understanding of how different supercomputer architectures behave when facing a bottleneck. Which is basically the "expert knowledge" that is required to optimise HPC applications for different supercomputers. Below I explain one possible method of how an analysis model can be derived:

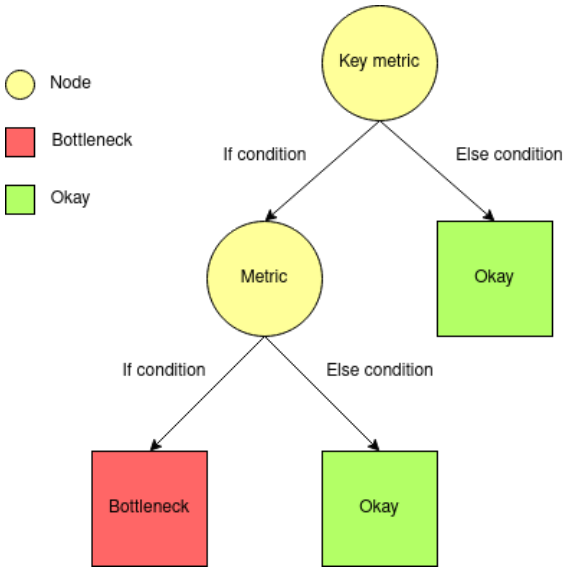


Figure 5.1.: Example of an analysis tree

Let's say we want to analyse an application for memory boundedness. As explained above, memory bandwidth is a very critical metric that can help us identify memory boundedness during an application's runtime. This "heuristic" was obtained from literature [ARR17] in our case, but it can also be derived from general practices in this field, or be derived from different tests conducted to analyse for specific inefficiencies. Now that we have defined a key metric, we use that as the root node to our analysis model and then add more layers of "heuristics" to refine our analysis.

We accumulate all the possible reasons an application can face memory boundedness and figure out how they can be identified from the metrics available to us from the framework. So for instance, as we explained above, strided access can be a major reason for an application to be memory bounded. Our first step here would be to understand what this means and then finding metrics that can help us find this sort of behaviour in the application's recorded runtime data. Since strided access would mean that the array element being accessed is usually present out of the cache, it would correlate to a lot of cache level misses and high amount of load operations from memory. This can be identified using cache level miss rates. So this "heuristic" is added to the analysis model.

In simple english this would mean: "If memory bandwidth and cache level misses are high, the application could be facing a strided access issue making the application memory bounded." [GC15] Similarly other aspects of memory boundedness need to be studied and metrics need to be identified that would help correlate a problem to a metric or set of metrics.

Analysis models in this tool would be built using the same ideas where a bottleneck is identified from a much wider perspective and then more rules or "heuristics" are added onto it to refine the analysis. This is also one of the reasons why a tree-like structure is chosen for our analysis models.

5.3. Thresholds

The process of analysis proposed here uses different metrics structured in a tree-like form that we call analysis models as explained above. Every path from the root node to the leaf node can be regarded as a chain of "heuristics" that would end at either identifying the reason for a bottleneck or the lack of it. In layman terms it would be a list of conditional statements clubbed together to form one path in the model. But the main engine that would support these analysis models is the threshold values for different metrics that would be indicative of the presence of a bottleneck or a possible reason for it.

For the sake of simplicity, I use the same example of memory bandwidth where memory bandwidth of more than 80% of the sustainable peak memory bandwidth of a system is indicative of memory boundness in an application. In this situation 0.8*(sustainable peak memory bandwidth) becomes the threshold value for memory bandwidth, above which the values are considered to be high and below this threshold value it is considered to be low. This is what is meant by a threshold in the context of our tool.

To make the process of analysis truly application and system agnostic, we need to identify metrics and their threshold values in such a way that the analysis process can be generalised throughout different systems for any application. Therefore not all threshold values for different metrics can be static in nature. They need to be calculated based upon the system capabilities and/or can be derived using mathematical or statistical models developed using historical data of different runtime performance metrics.

Metrics which deal with ratios or rates need not be dynamic in nature, for instance, if load to store ratio recorded for an HPC application is above 4 for on system A, there is no reason for it to show a different value on system B. So if it has been identified that 4 load operations for every store operation is high for an application on system A, then it would also be high on system B. Similarly vectorisation ratio, branch misprediction rate/ratio, etc need not necessarily be dynamic in nature.

Whereas threshold values for metrics like memory bandwidth, I/O read/write bandwidth, AVX FLOPS, etc which would vary from system to system need to be dynamic in nature.

5.4. Analysis of bottlenecks

As mentioned above, this tool would be limited in its capabilities with respect to the granularity of the level of analysis. The CB framework [Tip21] allows us to measure performance metrics at a minimum interval of 5 seconds. From the perspective of a processor, this is a long time interval but from the perspective of a standard HPC application which might have a runtime of more than a few hours or sometimes even days, this time interval provides us with enough information to be able to conduct analysis. The tradeoff between loop level or functional level analysis and system level monitoring is exhibited in the kind of analysis that can be conducted using this analysis tool.

The tool would never be able to pinpoint the loops or function calls that are causing inefficiencies in the application, but it would be able to parse the runtime data and filter the timesteps during which a bottleneck was identified and provide the user with possible reasons for the identified bottleneck. For example, the tool would ideally be able to identify strided access during certain time steps based on multiple metrics, but it would not provide

source code level information as to where exactly in the source code are strided accesses occurring.

The types of bottlenecks that the tool can address can be broken down into 5 main categories:

- Memory bound
- CPU bound
- I/O bound
- Load imbalance
- Bottlenecks caused due to the system topology, i.e., data transfer delays between processors placed physically far away.

Bottlenecks of other kinds such as memory leakage [CBG+][SMMC07], parallel inefficiency, etc. might be present and could possibly be identified using the metrics recorded by the framework.

It is important here to identify which different inefficiencies can be identified using only system level information and nothing else to go on. The analysis models would be derived based on trying to identify these inefficiencies and help the user identify possible reasons for said inefficiencies.

5.5. Severity

Since there can be multiple possible reasons for an HPC application to suffer through a bottleneck over it's runtime, the user should ideally be provided with the severity metric which informs them about the seriousness of one bottleneck as compared to the other.

As the tool analyses the runtime data of an application and informs about the timesteps during which these bottlenecks occur, the severity would ideally be calculated with respect to the amount of time the said bottleneck was present during the application runtime.

For instance, if an application performs a lot of I/O operations in its initialisation phase which takes about a minute to process. During this phase a lot of data is read but the read requests are small. It then moves onto its processing phase during which a lot of expensive instructions and inefficient scalar operations are encountered. This phase lasts about 10 minutes and about 8 minutes of this phase is CPU bound. In such a situation, both phases are bottlenecked due to different reasons, but the user should ideally invest it's time in fixing the bottleneck that occurs during the processing phase first and focusing their attention to the initialisation phase of the application as this would ideally solve a bottleneck that is present for a longer duration in the applications runtime.

This is a very rudimentary method to measure the severity of different bottlenecks that are identified during an application's runtime. There can be other more intelligent ways to arrive at such a metric but this is out of the scope of this thesis.

The need for such a metric comes from a common tendency to first solve a bigger problem and then trickle down to minor inefficiencies.

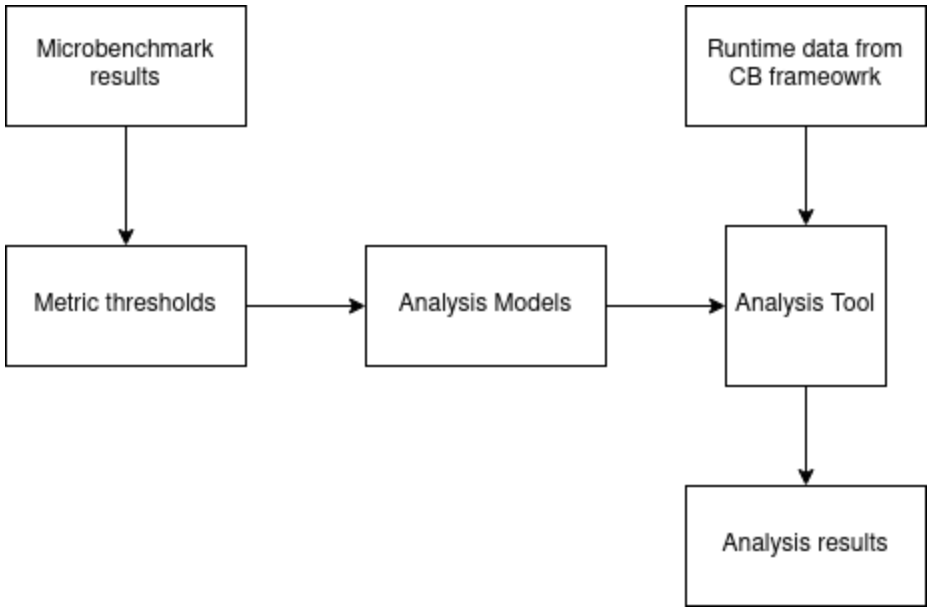


Figure 5.2.: Flowchart to show the proposed analysis process

5.6. Structure of the analysis tool

Based on these points I propose a general flow of the analysis process shown in fig. 5.2

The microbenchmarks would provide relevant results based upon which the certain metric thresholds would be generated, i.e. dynamic in nature, while the threshold for other metrics would be hardcoded, i.e. static in nature. These threshold values would be fed to the analysis models. The tool would take the runtime data generated by the CB framework and the analysis models as input and generate reports based on the two.

As explained in section [microbenchmark section] Microbenchmarks provide peak values for a wide array of performance metrics. It makes it an ideal choice for the tool to generate dynamic thresholds for these metrics.

The idea for the analysis models is closely based on the "strategies" presented in [GC15]. Since the type of data that is being analysed, runtime performance metrics over the course of job execution, is similar for both PerSyst and the tool suggested in this thesis, it made it a good starting point to explore this idea.

Part IV.

Analysis results and Tool Prototype

6. Analysis results

The workflow of this tool (as shown in fig. 5.2) shows that the tool firstly needs to benchmark the system based upon which the thresholds for a few important metrics would be derived. These threshold values are fed to the analysis model which the tool would use to perform analysis.

A few microbenchmarks were tested for these important metrics against runtime data of multiple different applications to find the best fit for the analysis tool. Due to a lack of relevant literature that states how specific metrics behave for different performance bottlenecks, PerSyst [GC15] was used as a baseline to create specific heuristics. These heuristics were tested against experiments that were chosen to mimic specific performance bottlenecks. Based on these results, the heuristics were further refined and the threshold values for these metrics were derived for the suggested analysis model.

All experiments presented in this thesis were executed on Intel Xeon Gold 6252 CPU @ 2.10GHz processor on two different nodes: Node 1 (simultaneous multithreading (SMT) disabled) - 48 threads (2 sockets * 24 cores/socket * 1 thread/core) and Node 2 (simultaneous multithreading (SMT) enabled) - 96 threads (2 threads/core). The CPU last level cache size as per product specifications is 35.75 MB, with a base frequency of 2.1 GHz and a max frequency of 3.7 GHz [xeo]. The processor data reported by the *lscpu* command reports cache size of approximately 38 MB (32 kB * 2 (L1-I and L1-D cache) + 1024 kB L2 cache + 36608 kB L3 cache).

This section highlights the applications used to generate relevant runtime data, and the process of analysis of these experiments with respect to an initial assumption on the heuristic and thresholds for two key performance metrics - Memory Bandwidth and FLOP rate. Memory bandwidth helps in dividing the analysis process into memory bound and compute bound behaviour, whereas FLOP rate is used to differentiate between good and bad performance of applications over their runtime with respect to the processor architecture. The threshold values suggested here for the analysis model are either computed as a percentage of peak values obtained from a microbenchmark for the system (dynamic - based on system architecture) or set values for metrics which report a percentage value or a ratio of some sort.

6.1. Applications studied

6.1.1. LULESH

LULESH is a proxy application created by the Lawrence Livemore National Laboratory (LLNL). It represents a typical hydrodynamics code, like ALE3D. It approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. A node on the mesh is a point where mesh lines intersect. It is built on the concept of an unstructured hex mesh. [lul]

Applications dealing with Hydronomics using Arbitrary Langrangian Eularian (ALE) formulation, perform non-unit-stride access within a region. This makes it an ideal selection to recreate a performance bottleneck caused due to strided access.

The application was tested on Node 2. To run LULESH in parallel, the number of MPI processes must be a perfect cube. Since Node 2 has 96 threads in total and 125 is the closest possible cube (5^3), 125 MPI processes were assigned to perform the experiment.

$$total\_elements = \#MPI\_processes * (per\_node\_problem\_size)^3 \tag{6.1}$$

Two test cases of per node problem size 30 (LULESH30) and 100 (LULESH100) were executed which equates to 3375000 and 125000000 total number of elements respectively according to the eq. 6.1.

6.1.2. KRIPKE

KRIPKE is another proxy application presented by LLNL. It is a 3D Sn deterministic particle transport code. It was built to research how data layout, programming paradigms and architectures affect the performance of Sn transport. [KBB15] KRIPKE supports storage of angular fluxes (Psi) using all six striding orders (or "nestings") of Directions (D), Groups (G), and Zones (Z), and provides computational kernels specifically written for each of these nestings. Most Sn transport codes are designed around one of these nestings, which is an inflexibility that leads to software engineering compromises when porting to new architectures and programming paradigms. [KB22] KRIPKE was tested on Node 1. Much like LULESH, the number of mpi processes needed to be in the power of three so 64 mpi processes were assigned to perform the experiments. Results from experiments with 2 different group sizes - 160 (KRIPKE160) and 320 (KRIPKE320) were used with scattering legendre expansion order of 6. These values were used to analyse runtime behaviour of the application at a high problem complexity with different data sizes.

6.1.3. 3-D Lid Driven Cavity

The HPC Benchmark Project [SSBA$^+$] presents us with a 3-D version of the Lid driven cavity flow tutorial provided by OpenFOAM. [ope] This set of simulations were built to show how memory bandwidth can become a limiting factor when scaling the size of the simulation. The simulation is built using simple geometry and boundary conditions, involving transient, isothermal, incompressible laminar flow in a three-dimensional box domain and uses the icoFoam solver for its computations. [SSBA$^+$] It uses the Pressure Implicit with Splitting of Operators algorithm to solve for the continuity and momentum equations [ope]:

$$\nabla . u = 0 \tag{6.2}$$

$$\frac{\partial(u)}{\partial t} + \nabla.(u \otimes u) - \nabla(\nu \nabla u) = -\nabla p \tag{6.3}$$

Where u is velocity, ν is constant for Newtonian flow [Nil23], and p is the kinematic pressure.

The test-case contains a problem set of 4 different sizes - Small (S), Medium (M) and Extra-Large (XL), Extra-extra-Large (XXL). Simulation results for problem size S, M and XL are explored here.

Test case	S	M	XL
#Cells (million)	1	8	64
Δx	0.001	0.0005	0.00025
Δt	0.001	0.00025	0.0000625

Table 6.1.: Test cases for the 3-D Lid Driven Cavity

6.1.4. MaMiCo

MaMiCo is a macro-micro coupling tool developed to ease the development of and modularize molecular-continuum simulations, retaining sequential and parallel performance. It couples the spatially adaptive Lattice Boltzmann framework waLBerla and continuous fluid dynamics (CFD) software with four molecular dynamics (MD) codes: the light-weight Lennard-Jones-based implementation SimpleMD, the node-level optimized software ls1 mardyn, and the community codes ESPResSo and LAMMPS. [NFA+16] User can set the size of the Experiments were conducted on simulations with different data sizes using the SimpleMD solver. SimpleMD stores 3D position, velocity and two forces for each molecule. 3 test cases with 28 (MAMICO28), 72 (MAMICO72) and 224 (MAMICO224) molecules per direction were experimented with to see if the solver reproduced a memory bound behaviour.

6.1.5. Switch case

This program consists of a simple switch case that takes in a value ranging from 1-100. The first 10 cases take in a DP floating point value and performs any one of the following 10 operations on it: divide input by a random number between 1-57, power of the input for a random integer number between 1-5, log of the input, power of the input by a random number ranging from 0.33-1, sin, cos, tan and arctan on the input, sqrt of the input or floating point mod (fmod) of the input with a random number between 1-123. The switch case is as shown in A.1.

This program was used to try and recreate a performance bottleneck caused due to branch misprediction. It was parallelized using OpenMP and few components used to create the dataset and measure performance time were borrowed from [Bog].

6.1.6. Expensive Instructions

This program was adopted from [Bog]. It calculates the average of an array containing random values. The program was modified to run on multiple threads using OpenMP and a few more heavy operations were added at every loop iteration to increase operation overhead. The loop kernel is as shown in A.2.

This program was used to create a performance bottleneck caused when multiple expensive instructions are executed in a single loop iteration.

6.2. Memory Bound Analysis

Memory bounded applications, as explained in section 3.1, tend to work at a very high memory bandwidth. The heuristic translates to - "if the memory bandwidth is high, then it is bounded by memory" where the threshold value for the metric is 80% of sustainable peak memory bandwidth as suggested in [ARR17].

The 3-D Lid Driven Cavity test cases were used to record runtime data for memory bounded application. Results show that mean memory bandwidth increases very sharply from almost 50000 MB/s for openfoamS to almost 200000 MB/s for openfoamM. Whereas the mean memory bandwidth for the openfoamXL experiment was close to 160000 MB/s but 22% of the runtime is spent with almost 0 MB/s on the decomposePar operation. This operation decomposes the mesh and fields of a case for parallel execution. [ope] If we omit the data collected during this interval, the mean memory bandwidth jumps to almost 210000 MB/s. The bandwidth does not increase much with respect to openfoamM ($\approx$ 220000 MB/s) even though the total number of cells is 8 times more for openfoamXL. This confirms the fact that both openfoamM and openfoamXL are bounded by memory. All of these test cases were performed on Node 1.

Tests on LULESH showed that by increasing the per node problem size from 30 to 100, the mean memory bandwidth increased from almost 50000 MB/s to 166000 MB/s.

Three different microbenchmarks were used to benchmark the peak sustainable memory bandwidth of the system:

- STREAM - as explained in section 4.1, the general rule for this microbenchmark is that each array must be at least 4x the size of the sum of all the last-level caches used in the run, or 1 Million elements, whichever is larger. [M+95] An array size of 20 million, equivalent to 152.6 MB memory per array (38 MB * 4) and total memory of 457.8 MB, to benchmark the system. The value considered here is reported by the Triad kernel as explained above. Node 1 generated a peak sustainable memory bandwidth of approximately 150000 MB/s and Node 2 generated almost 130000 MB/s. The benchmark was compiled using the original source code and with OpenMP directives enabled to run the benchmarks on multiple threads. [McC]

- Likwid-bench stream - the Likwid-bench microbenchmark suite also provides an implementation of the STREAM microbenchmark. The results reported by this benchmark are also based on the same Triad kernel. Node 1 and Node 2 both reported a value of approximately 180000 MB/s with an array of size 457MB.

- Likwid-bench load - In [RHa] the developers of the likwid-bench tool suggest using the load benchmark to assess the performance of memory for the system as it generally produces the highest bandwidth value. The results obtained for this benchmark is close to 250000 MB/s on Node 1 and close to 242000 MB/s on Node 2.

Node	STREAM	Likwid-bench stream	Likwid-bench load
Node 1	120000	144000	200000
Node 2	104000	144000	193600

Table 6.2.: Peak sustainable memory bandwidth values obtained from different benchmarks (MB/s)

Table 6.2 lists the threshold values for memory bandwidth on Node 1 and Node 2 as per the heuristic stated above for with respect to each microbenchmark. Fig. 6.1 shows 2 graphs of the memory bandwidth data of LULESH100 on Node 2 and openfoamM on Node 1. The lines mark the threshold values provided in 6.2 over which the application is considered to be memory bound.

As we can see from the two test cases, we were able to identify memory bound behaviour for both the applications using the threshold calculated using both versions of the stream benchmark but were unable to identify such a behaviour for LULESH when we consider the load microbenchmark provided by Likwid. The threshold value set using the stream benchmark provided by likwid-bench seems to work the best with our 80% threshold hypothesis to provide us a threshold for the memory bandwidth metric.

The runtime data provided by these two applications were further analysed to find a common pattern between the test cases which could be related to specific performance bottlenecks.

Both of the memory bounded applications show a common trend of high L3 miss ratio, L3 bandwidth, high stalls caused due to memory loads and very high number of total stalls. Fig. 6.2 shows the peak sustainable bandwidth obtained by the likwid-bench stream benchmark over different data sizes. At 120 MB, a little more than 3 times the total cache size, the peak sustainable memory bandwidth was close to 480000 MB/s on both Node 1 and Node 2. The data size was adjusted so that the array is small enough to completely fit into the cache memory.

The maximum L3 bandwidth recorded while testing the memory bound test case of LULESH100 was close to 350000 MB/s whereas for openfoamM and openfoamXL this value was close to 390000 and 350000 MB/s respectively. While the L3 miss ratio was close to almost 1 in the case of both openfoamM and openfoamXL and close to 0.9 for the LULESH test case.

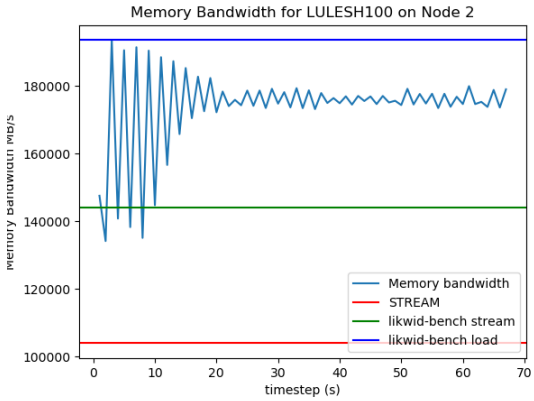
The L3 bandwidth recorded by the LULESH test case with per node problem size 30 shows a maximum value of almost 117000 MB/s. Whereas in the case of 3-D Lid Driven Cavity test case S the maximum value recorded was almost 535000 MB/s. The L3 miss ratio showed a maximum value of 0.6 and 0.36 for both of these cases respectively.

This sort of behaviour is expected in situations where the data has to be loaded from memory quite often as it is not present in the last level caches. This could be indicative of poor data level parallelism. Either the data is accessed poorly (irregular or strided access) or there is inefficient data locality. [GC15]

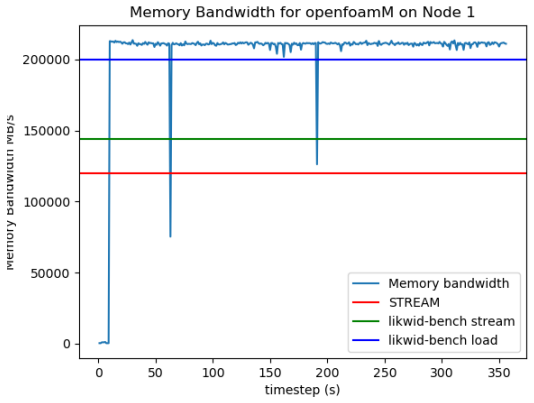
According to these results, it can be said that a memory bound application shows a common trend of high L3 miss ratio and high L3 bandwidth for both the applications. Whereas only high L3 bandwidth is not solely indicative of a performance bottleneck as shown by the test case S. Since the L3 miss ratio metric is independent of the system architecture it should

behave evenly for different applications across different system architectures that face a similar bottleneck as shown by these 2 applications. Which means that the threshold for this metric should be static in nature. The threshold was set to 0.8 for this metric based on the values reported by the memory bound and non-memory bound test cases. Whereas the L3 bandwidth would depend upon the system architecture and its threshold value should be derived using microbenchmarks that would help to measure cache level bandwidth. The threshold value for L3 bandwidth was set to 70% of the measured peak bandwidth. This value was chosen based on the analysis results. For the presented system architecture this value was close to 336000 MB/s for both Node1 and Node2 for the system architecture used for experimentation.

I propose the analysis model shown in fig. 6.3 based on these results.



(a) Memory Bandwidth recorded for LULESH100



(b) Memory Bandwidth recorded for openfoamM

Figure 6.1.: Runtime Memory Bandwidth for LULESH100 and openfoamM with three different threshold values provided in table 6.2

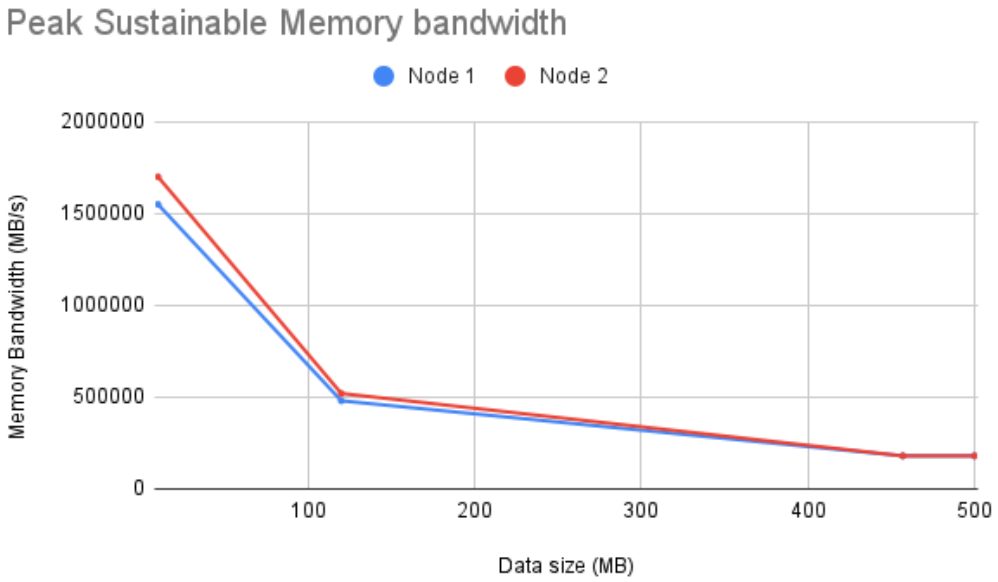


Figure 6.2.: Peak sustainable memory bandwidth values measured using the likwid-bench stream benchmark for different data sizes

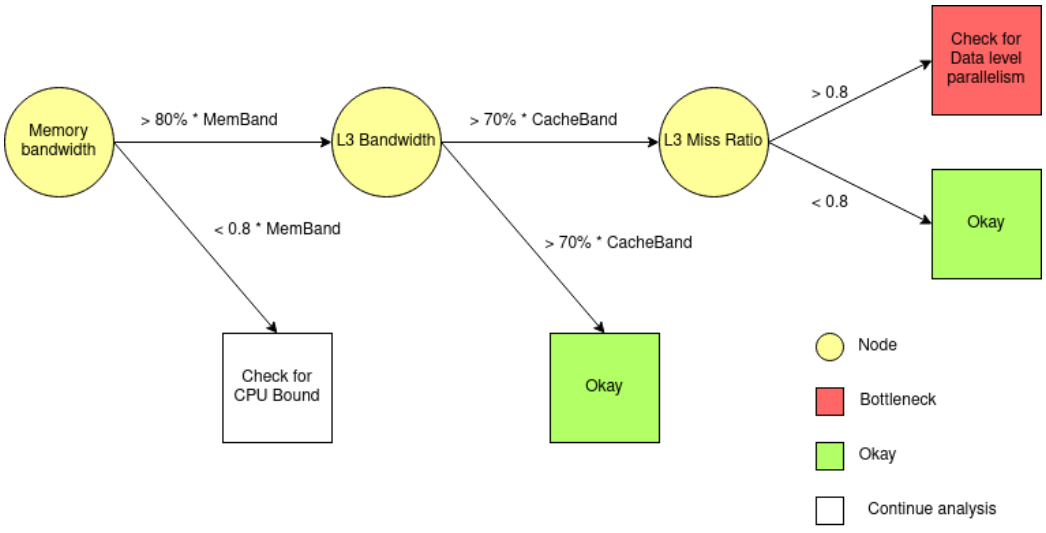


Figure 6.3.: Proposed analysis model for memory bound analysis

6.3. CPU Bound Analysis

The model is set to only analyse CPU bound behaviour for timesteps where the memory bandwidth value is below the set threshold. In such a situation the model checks if the CPU Usage is above 90%. If an application is unable to utilise at least 90% of the CPU, it is possible that the application is not utilising all available cores. Either it is running on fewer threads than available or it is not configured well enough to utilise all of them. This is a static value set as a starting point for the other half of the analysis model that would analyse for compute related performance bottlenecks. The heuristics suggested here are built to analyse situations where the CPU performs inefficiently. Just like in memory bound analysis it is important to benchmark the processor's peak performance. Processors are benchmarked based on their FLOP rate. It tells us how close the processor can perform with respect to its theoretical peak FLOP rate. Table 6.3 lists the theoretical peak FLOPS for DP and SP floating point operations for Node 1. It was computed using the eq. 6.4 provided in likwid-bench peakflops documentation. [RHa]

$$FLOPS = \#cores * op\_width * \#ops\_per\_cycle * \#FMA * cpu\_frequency \tag{6.4}$$

Precision	Scalar	SSE	AVX	AVX-512
DP	201.6 - 355.2	403.2 - 710.4	806.4 - 1420.8	1612.8 - 2841.6
SP	201.6 - 355.2	806.4 - 1420.8	1612.8 - 2841.6	3225.6 - 6451.2

Table 6.3.: Theoretical peak FLOPS (GFLOPS) for DP and SP floating point operations

The likwid-bench peakflops microbenchmark and HPL were used to benchmark the system's peak FLOP rate:

- Likwid-bench peakflops - The peakflops microbenchmark was tested to measure the peak performance of the CPU in FLOPS. The CPU frequency value ranges between 2.1 GHz to 3.7 GHz (max turbo frequency [xeo]) in this case. Table 6.4 lists the results achieved using the likwid-bench peakflops microbenchmark.

Precision	Scalar	SSE	AVX	AVX-512
DP	266.2	533.1	906.2	1500
SP	266.7	1062.5	1817.2	3000.1

Table 6.4.: Peak FLOPS (GFLOPS) measured using microbenchmark

These measurements were made using 48 threads on Node 1 and 96 threads on Node 2 using vectors of size and 1536 kB and 3072 kB respectively (number of threads * L1-D cache size). SMT showed no effect on the FLOP rate. Data suggests that the processor is able to achieve a 32% higher peak sustainable FLOP rate for scalar and SSE operations than the calculated minimum theoretical peak, approximately 12% more for AVX FLOP rate. These values were 25% and 36% lower with respect to the calculated maximum theoretical peak FLOP rate. Interestingly the microbenchmark

for AVX-512 FLOP rate could only achieve 1500 GLFOPS which is approximately 7% less than the calculated minimum theoretical peak FLOP rate and approximately 47% less than the maximum possible value. The FMA measurements for the AVX and AVX-512 were both approximately 2 times the measured FLOP rate for non-FMA test cases. Which is to be expected as the processor has 2 FMA units. [xeo]

- HPL - As explained in section 4.3, it is a computationally intensive algorithm that has been used as a metric of performance to rate the Top500 supercomputers. Multiple tests were performed using different problem sizes and block sizes. The system was able to achieve a maximum performance of 1.5 TFLOPS (1500 GFLOPS) for a problem size of 30000 and block size of 160 on Node 1 with an operational intensity of 19. This value correlates to the maximum GFLOPS measured using the likwid-bench peakflops benchmark used to measure peak AVX-512 FLOPS. This value was also verified by the AVX-512 FLOPS DP metric when the HPL microbenchmark was run through the same CB framework. One important note here is that the benchmark ran using mostly just vector operations. The Packed DP metric recorded a maximum value of 192000 MUOPS in this case while there were negligible scalar operations conducted.

The maximum FLOP rate that was recorded with respect to all experiments was 118 GFLOPS. This was achieved while experimenting with the expensive instructions program. Where the ratio of vectorised operations (Packed) and scalar operations (Scalar) was almost 0.5 for the entire run. Since the program's vectorisation was completely controlled by the C++ compiler itself, it is very difficult to quantify the significance of the Packed and Scalar metrics, on the FLOP rate performance of the processor. This makes it a little tricky to assign a threshold value to the FLOPS metric, statically or dynamically.

6.3.1. Poor Vectorization

s stated above the maximum recorded FLOP rate across different experiments was close to 118 GFLOPS produced by the expensive instructions kernel. The program was compiled using the O3 gcc compiler optimization flag. This flag enables the compiler to find loops that can be vectorised and optimises the assembly code accordingly. The Packed metric, which measures all the retired vectorised micro operations, reported a value of 30000 MUOPS and the Scalar metric reported a value of 60000 MUOPS. When the same program was compiled using the O1 gcc compiler optimization flag, no vector microoperations were executed and the scalar metric recorded close to 60000 MUOPS at a FLOP rate of close to 65 GFLOPS.

APP	Expensive Instructions	LULESH 30	LULESH 100	MAMICO 28	MAMICO 224	openfoam S	openfoam M
FLOPS (GFLOPS)	118	60	50	40	50	60	25
Packed (MUOPS)	30000	5000	700	7200	10000	3300	1500
Scalar (MUOPS)	60000	48000	20000	25000	30000	400000	20000
Ratio	0.5	0.1	0.035	0.288	0.33	0.0825	0.075

Table 6.5.: FLOP rate, Packed, Scalar and their ratio for a few experiments.

The 2 cases of the kripke proxy app tested on the framework showed very erratic values for all three metrics Packed, Scalar and FLOPS. FLOPS peaked at close to a 100 GFLOPS and a minimum of 21 GFLOPS throughout the run, whereas there were several timesteps without any vectorised instructions. The scalar metric kept oscillating between a peak close to 80 GFLOPS and a trough close to 5 GFLOPS.

This shows that a poorly vectorised code can be a reason for limited FLOP rate of the processor. It is a little difficult to set a threshold value for this ratio as even real world applications cannot be completely vectorised as well as in the case of the HPL benchmark. Considering the available data, I suggest a static threshold of 0.25 for the Packed to Scalar ratio.

If an application is properly vectorised and still the FLOP rate and CPI values are still low, the recommendation would be to properly analyse the application for inefficiencies.

Based on these results my recommended heuristic to analyse for poor vectorisation would then be "if low FLOPS and if low Packed to Scalar ratio, then poor vectorisation, else perform in depth analysis."

6.3.2. High Branching Misprediction

The switch case program was used to analyse the effect of a heavily branched kernel on branching related metrics.

PerSyst [GC15] suggests that if the branch misprediction to instructions ratio is high and if the branch mispredicted to branches ratio is also high it relates to performance bottleneck caused due to branch misprediction.

According to the likwid documentation, branch misprediction rate is the ratio of the number of retired mispredicted branches to total number of retired instructions and branch misprediction ratio is the ratio of the number of retired mispredicted branches to that of the total number of retired branches. This means that if the branch misprediction rate is high and if the branch misprediction ratio is also high, it should mean that the program is bottlenecked due to the extra performance penalties incurred because of abnormally high branching in the code. The branch predictor's performance is limiting the processor's computing capacity. Using the switch case I was only able to record a branch misprediction rate of 0.02 and a branch misprediction ratio of 0.1. The branch rate of this program was recorded at almost 0.22. This metric measures the ratio of the total number of branch instructions retired to that total number of instructions retired. These values seem to be small but when compared to all the other applications and programs experimented with, this was the highest value that I was able to record in all of my experiments. The branch rate metric for other experiments range from 0.04-0.18 and both the branch misprediction rate and ratio were all very close to 0. These values are very close to help in making any estimates on their threshold values. But knowing that the program is heavily branched and the branch rate is 4% higher than the highest recorded branch rate for other applications it does make sense to analyse other metrics for clues that could be relevant to analyse for branch misprediction.

As explained in section 3.2.2 branch mispredictions can waste up to ten clock cycles due to the wasted clock cycles used on the operations in the mispredicted branch, cycles wasted to flush out the instructions for the mispredicted branch and then a few cycles to load the instructions for the correct branch. According to [nac], CPI of 1 is acceptable for HPC

applications and Evas et al in [EBB+14] even use 1 as a threshold for CPI in their HPC system monitoring tool to identify applications with possible inefficiencies. Causes for high CPI values can be long-latency memory, floating-point or SIMD operations or non-retired instructions due to branch mispredictions. [nac]

The switch case recorded a value of 9.5 CPI, a very high value for this metric as stated above. The FLOP rate recorded for this program was also very low. It can be reasoned that the stalls caused due to flushing out the wrong instructions and loading the new instructions would decrease the FLOP rate of the program. For all the other experiments, the maximum achieved CPI was for the 3-D Lid Driven Cavity test case XL at almost 1.5 and the test case M recorded a value of 1.3. Whereas for the HPCG benchmark and the LULESH100 test case, the value recorded was close to almost 1. As all of these applications mentioned here are memory bound, their CPI values can be expected to be around a threshold of 1. Other applications such as Mamico, Kripke and even the 3-D Lid Driven Cavity test case S averaged a consistent CPI rate of 0.4 during their compute intensive phases. Based on this I suggest the heuristic - "if FLOPS low and if CPI high check for branch misprediction." with a threshold of 1 for CPI. Memory bound operations would be filtered out before they even reach this deep into the analysis by the root node of the suggested analysis model.

6.3.3. Expensive Instructions

As explained in section 3.2.3, heavy operations performed consistently increase the operational intensity of the application. This usually happens when there are multiple heavy operations taking place in a loop. [GC15] To analyse for such a behaviour the initial heuristic assumed was "if low FLOPS and if high CPI and if high Operational Intensity, then check for expensive instructions."

The logical basis for this assumption was that if the processor is performing heavy operations, it would comparatively perform fewer (but heavier - higher ops value 3.2.3) floating point operations per second and need more cycles per operation.

APP	Expensive Instructions	KRIPKE 320	MAMICO 28	MAMICO 72	MAMICO 224	LULESH 30	openfoam S
FLOPS (GFLOPS)	118	30-90	40	55	50	60	55
CPI	1	0.35	0.66	0.42	0.4	<1	0.4
Op. Intensity	50	1-25	>10	<5	2-5	2	2

Table 6.6.: FLOP rate, CPI and Operational Intensity for a few experiments

Table 6.6 lists the FLOP rate, CPI and operational intensity for a few experiments. The expensive instructions program recorded an operational intensity of 50 while the FLOP rate remained fairly low as compared to the peak value measured by our benchmarks. This behaviour was also recorded while experimenting with the proxy application KRIPKE. The operational intensity value kept alternating between 25-1 between. The FLOP rate for this application was also very low. The runtime operational intensity recorded during the MAMICO28 experiment was close to around 10 throughout the coupling cycle with a FLOP

rate of close to only 60 GFLOPS. Further experiments on Mamico with different numbers of molecules in each direction (72, 224, 448) show a maximum FLOP rate close to 70 GFLOPS for the test-case with 448 molecules per direction. Whereas the lowest operational intensity was recorded at slightly above 5 for the experiment with 72 molecules in each direction. All other experiments recorded fairly low values for operational intensity. Operational intensity for both LULESH30 and openfoamS were limited to 2.

None of the recorded experiment data fits with our assumed heuristic in this case. If the heuristic would be modified to fit for our use cases it would suggest that low FLOP rate with low CPI and high operational intensity would be indicative of operationally heavy kernels. But these results are not conclusive enough to add this heuristic to the analysis model.

6.3.4. Object Instantiation

PerSyst [GC15] suggests that when both the FLOP rate and the CPI of a program are low, it is indicative of performance being limited due to excessive object instantiation. Hidden functions are added to the compiled code that increase overheads and decrease the processors compute efficiency.

Here it is important to note that low FLOP rate and CPI values were also recorded for programs with high operational intensity values as shown above. This observation works counter intuitively for our assumption. It cannot be conclusively said the low FLOP rate with low CPI is indicative of excessive object instantiation.

All memory bound experiments showed low values of operational intensity (<1) which would be expected due to high memory bandwidth. The smaller test cases LULESH30 and openfoamS recorded similar values for operational intensity (<2). None of the results from the experiments were conclusive enough to observe a performance bottleneck caused due to object instantiation.

Based on the FLOP rate measured for all the different experiments I suggest that the threshold be set at about 80% of the system's peak FLOP rate when performing only scalar operations. Below which it would be considered as low. In this case the value is close to 212 GFLOPS. This value is suggested with respect to the data collected across all experiments and considering how volatile a processor's performance is to vectorisation, operational intensity and CPI. This value would be dynamically calculated using a microbenchmark used to measure a system's peak FLOP rate for scalar operations, in our case the likwid-bench peakflops microbenchmark.

6.3.5. Proposed analysis model

Considering all of the suggested heuristics I propose the analysis model shown in fig. 6.4. This would be an extension of the analysis model shown in fig. 6.3.

6.4. Tool Prototype

6.4.1. Interface

The prototype of the tool is built as a web application on the Vue.js framework (Vue2). As shown in fig. 5.2, the microbenchmark results which are used to generate dynamic thresholds

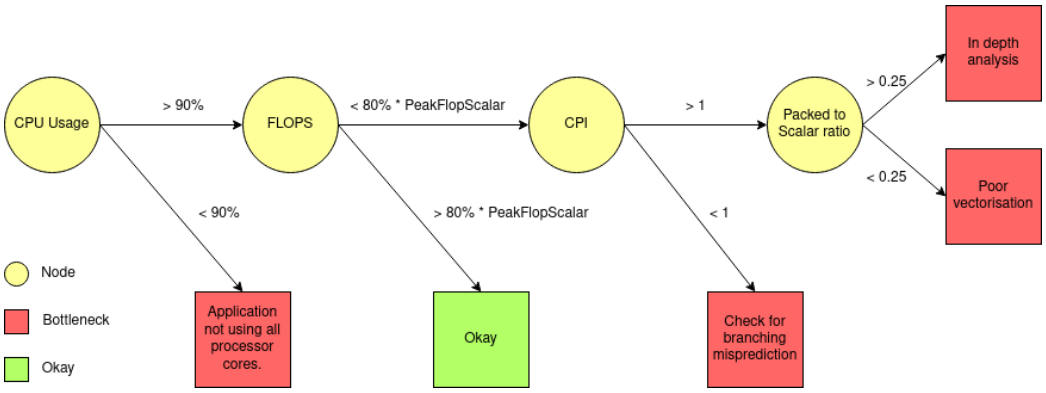


Figure 6.4.: Proposed analysis model for CPU bound analysis

for the key metrics, the threshold for these metrics are passed onto the analysis model. The tool takes this analysis model and the runtime data generated by the CB framework as input to generate relevant charts that mark the bottlenecked regions across the applications runtime data.

In the prototype version, the analysis model is defined using a JSON file. The frontend parses through this file and creates a list of objects that contains all the information required to calculate if a certain metric is above or below its threshold value.

The JSON file contains an array of analysis models. Each model is structured to define a tree. There can be two types of objects in an analysis model:

- Node - A node is a collection of metric, condition ('>'/'<'), threshold and branches of the tree - "yes" and "no" wherein, if the condition is met or not with respect to the metric and the threshold the analysis moves onto the yes branch or the no branch respectively. "yes" and "no" are arrays that either contain more node objects or one leaf object. It is important to note that a model can test for multiple conditions at every level of the tree except the root node but will always have only one leaf object.
- Leaf - A leaf object just contains the output value that can be either a bottleneck or not depending upon the analysis.

All branches of the analysis models terminate on either a possible bottleneck or an 'OKAY'.

Both of these objects also contain a "level" key. This is used to track the position of the node or leaf in the tree. The root node is indicated using (0). Every step further into the model adds onto this level. The level of the yes branch from the root node would be (0)-Y(0) which means that this is the first yes condition of the heuristic at level (0). If there are 3 conditions that need to be checked in the yes branch, then their levels would be (0)-Y(1) and (0)-Y(2). A no branch arising from the root node would similarly be (0)-N(0). The further you go down the tree, the condition initial - Y/N and the array position of the condition in the corresponding branch is concatenated to the level string of its predecessor node. Therefore an analysis model from (0) to (0)-Y(1)-N(1) in the form of a tree would look like fig. 6.5.

It is possible that the analysis model is such that at a single node a combination of multiple metrics need to be analysed. For example if you want to measure the FLOPS SP

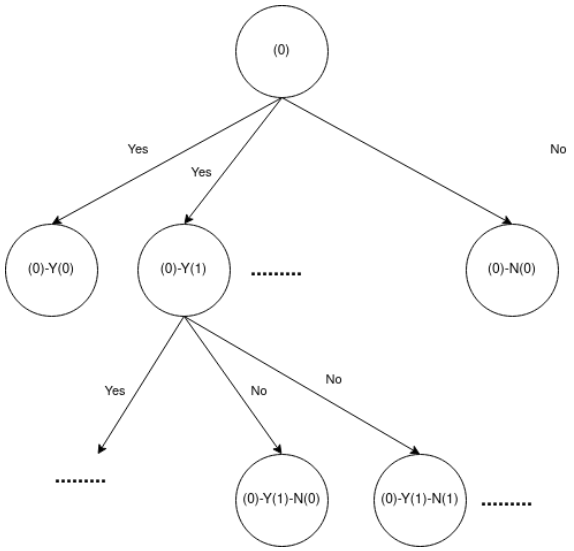


Figure 6.5.: This figure shows how the levels are defined for the analysis model

and FLOPS DP ratio then the metric would be an object which contains the keys 'metric1', 'metric2' and 'op' which is the operation you want to conduct on the two metrics. The prototype presently supports only two metrics and they can either be added, subtracted, multiplied or divided.

The analysis model shown in fig. 6.3 would look like this as a JSON object:

```
{
    "type": "node",
    "level": "(0)",
    "metric": "mem.Memory bandwidth",
    "condition": ">",
    "threshold": 144000,
    "yes": [{
        "type": "node",
        "level": "(0)-Y(0)",
        "metric": "mem.L3 Bandwidth",
        "condition": ">",
        "threshold": 336000,
        "yes": [{
            "type": "node",
            "level": "(0)-Y(0)-Y(0)",
            "metric":"mem.L3 Miss Ratio",
            "condition": ">",
            "threshold": 0.85,
            "yes": [{
                "type":"leaf",
                "level":"(0)-Y(0)-Y(0)-Y(0)",
                "output": "Check data level parallelism"
            }],
            "no": [{
                "type":"leaf",
                "level":"(0)-Y(0)-Y(0)-N(0)",
                "output": "OKAY"
```

```
28                  }]
29              }],
30              "no": [{
31                  "type": "leaf",
32                  "level": "(0)-Y(0)-N(0)",
33                  "output": "OKAY"
34              }]
35          }],
36          "no": [{
37              "type": "leaf",
38              "level": "(0)-N(0)",
39              "output": "OKAY"
40          }]
41      }
```

Listing 6.1: JSON object for the memory bound analysis model

The benefit of using such a JSON object is to allow the user to create their own analysis models by following very simple rules. As mentioned earlier, this JSON object is converted to a list of objects by the frontend. This list is used by the charting library to determine which charts to display, the threshold line for each chart and the regions that need to be marked as bottlenecked based on the condition and runtime data. Once a bottleneck region is marked at one node of the analysis model, only those points of the runtime data are analysed by the following nodes of the analysis model.

6.4.2. Usage

The proposed analysis model is as shown in fig. 6.6. But the model has been divided in two parts - memory bound analysis model (root node metric - Memory bandwidth) and compute bound analysis model (root node metric - CPU Usage) for ease of usage.

Example use case

I present the results generated by the prototype tool to analyse the results from the KRIPKE160 experiment. The interface is divided into 2 tabs. First tab displays charts for all the relevant metrics for the selected dataset. The corresponding threshold values are marked in a green line across each chart. The user can choose the dataset they want to analyse and the key metric they want to analyse for. The tool updates the charts that should be displayed depending upon the selected key metric. Second tab lists the analysis models in the form of a table (fig. 6.7).

Fig. 6.8 shows the memory bandwidth of the KRIPKE160 experiment over its runtime. The memory bandwidth value never crosses the threshold mark set in the analysis model.

Since there is no memory bound behaviour found in this dataset, the next step is to look for compute related performance issues. Fig. 6.9 (a) marks the region in the runtime data that is beyond the set CPU Usage threshold (>90%). This is the root node of the compute bound analysis model. In the next steps the analysis model would only be used on the timesteps marked in this stage. At every level these regions are updated based upon their runtime data and the condition set for that node. The model next checks for the FLOP rate of the application. Fig. 6.9 (b) shows the results generated at level (0)-Y(0) of the analysis model. The model checks for the CPI value. The timesteps for which the CPI value is below

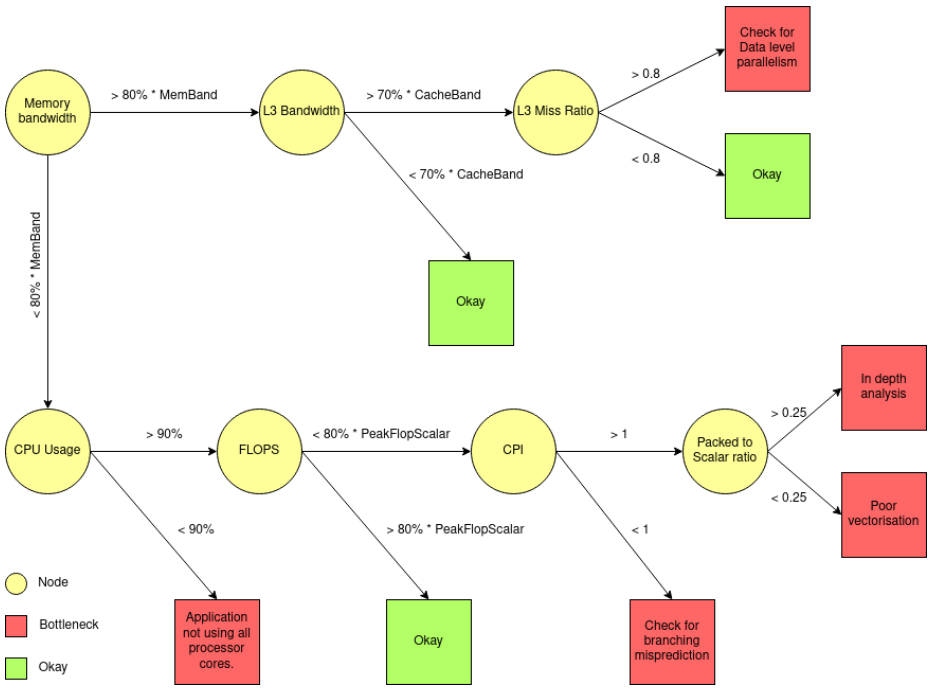


Figure 6.6.: The proposed analysis model used by the prototype. Terms used: MemBand - peak memory bandwidth measured by the likwid-bench stream benchmark with the appropriate data size, CacheBound - peak cache bandwidth measured by the likwid-bench stream benchmark with the appropriate data size, PeakFlopsScalar - peak DP FLOP rate for scalar operations measured by the likwid-bench peakflops microbenchmark.

Aalysis Models

Level	Metric	Condition	Threshold	Yes	No
index: 1					
(0)	mem.Memory bandwidth	>	144000	mem.L3 Bandwidth > 336000	Check for CPU related performance bottlenecks
(0)-Y(0)	mem.L3 Bandwidth	>	336000	mem.L3 Miss Ratio > 0.85	OKAY
(0)-Y(0)-Y(0)	mem.L3 Miss Ratio	>	0.85	Check data level parallelism	OKAY
index: 2					
(0)	cpu.Usage User	>	80	cpu.FLOPS DP < 150000	Check for I/O operations or increase number of compute cores
(0)-Y(0)	cpu.FLOPS DP	<	150000	cpu.CPI < 1	OKAY
(0)-Y(0)-Y(0)	cpu.CPI	<	1	[object Object] < 0.25	Check for Branch Misprediction
(0)-Y(0)-Y(0)-Y(0)	[object Object]	<	0.25	Low vectorisation	In depth analysis

Figure 6.7.: The analysis model tab shows a list of all the analysis models passed to the tool. Index lists the total number of models present. It shows the level of the node, the metric, condition and threshold against which the values are checked. Based on the result, the column yes and no suggest the next steps to be taken in the analysis process. Suggestions can be - a list of metric, condition and threshold to analyse next or an OKAY or a possible bottleneck identified by the model.

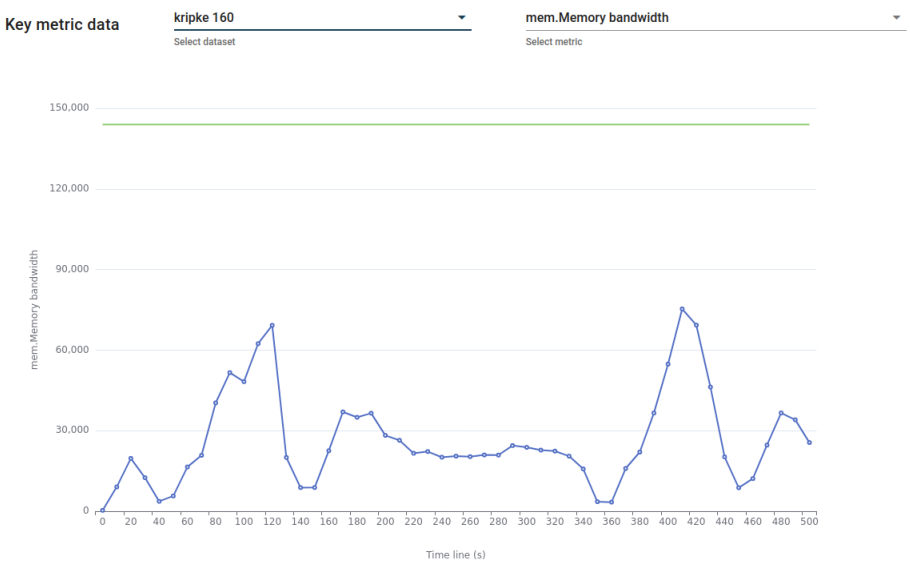


Figure 6.8.: Memory Bandwidth for experiment KRIPKE160

1, the packed to scalar ratio is checked for the runtime data. Fig. 6.9 (c) and fig. 6.9 (d) show these steps of the analysis for the experiment.

As we can see from fig. 6.9 (d) the marked regions in the chart are all a subset of the bottlenecked regions identified from the prior steps. There are a few regions that were marked as bottlenecked by the prior steps but the packed to scalar operations ratio is above the set threshold and thus they are considered to be alright by the analysis model. Due to the scale of the chart, some of the timesteps where the value is above the threshold are indistinguishable.

6.4.3. Critical analysis

The roofline model is a performance model proposed by Williams et al in [WWP09]. It is a popular method to benchmark an application's performance based on the system capabilities. This model is built using the peak FLOP rate of the system and the sustainable peak memory bandwidth using the eq. 6.5.

$$y = \begin{cases} x * MaxBand & \text{if } x * MaxBand < MaxFLOPS \\ MaxFLOPS & \text{if } x * MaxBand > MaxFLOPS \end{cases} \tag{6.5}$$

Applications that lie underneath the slope (as shown in fig. 6.10) are considered to be limited by the memory bandwidth whereas applications that are present on the right side of the slope in this model are considered to be limited by the FLOP rate of the processor.

The proposed analysis model (with respect to the roofline model shown in fig. 6.10) rightly suggests that the HPL benchmark is limited by the peak FLOP rate of the system, but there is no real performance issue - "if FLOPS high, it is OKAY" as shown in the analysis model 6.6. But if we consider the MAMICO224 experiment, the roofline model would suggest that

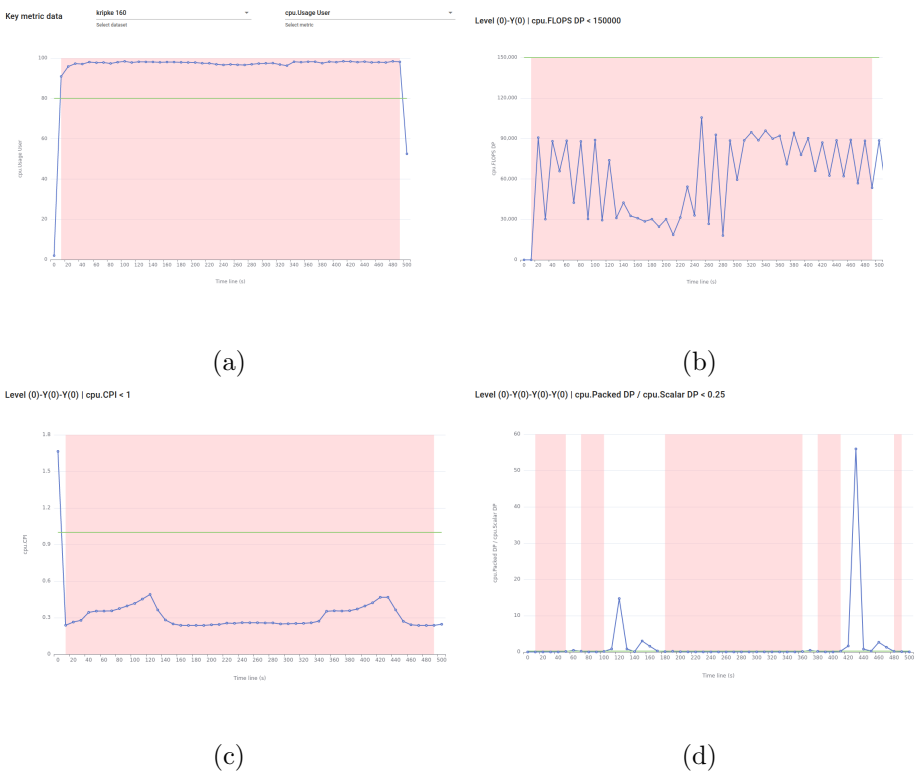


Figure 6.9.: The first graphs shows the CPU Usage plot, the second graph shows the L3 Bandwidth and the third graphs shows the L3 Miss ratio for the 3-D Lid Driven Cavity test case M

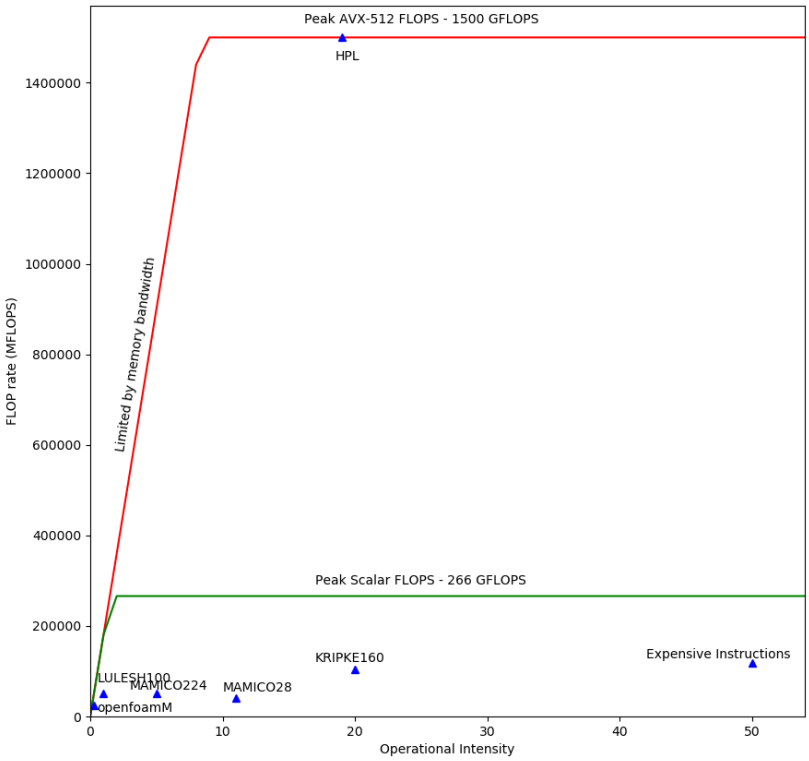


Figure 6.10.: The roofline model of the system architecture used in this thesis. The peak FLOP rate vs the peak memory bandwidth of a few experiments are marked in blue triangles.

the application is a memory bound with respect to the peak AVX-512 FLOP rate. But the analysis model fails to find any region in the runtime data to suggest so.

The LULESH100 and openfoamM experiments which lie very close to the slope (memory bound) are rightly identified by the analysis tool as well, although the only possible recommendation of the present analysis model is with respect to strided access. As shown above the tool marks different regions in the runtime data for the KRIPKE160 experiment where the packed to scalar ratio is lower than suggested. For MAMICO28 the tool would recommend performing an in depth analysis to the user for all the timesteps where the CPI is low but the packed to scalar ratio is high.

Part V.

Conclusion and Future Work

7. Conclusion and Future Work

An analysis model, presented in 6.6, was built using initial assumptions based on ideas and suggestions collected from literature and modified for our tool with respect to the obtained experiment results. Applications with known performance bottlenecks were chosen to generate runtime data. This data was used to analyse metric behaviour during various performance bottlenecks. The threshold values presented in the analysis model were also derived based on different sources and verified using the experiment results. The nature of the threshold value, dynamic or static, for different metrics were suggested along with different microbenchmarks that can help in computing the dynamic threshold values for different metrics.

The proposed idea of the tool is based on the work presented in [GC15] and [BKD+10]. The suggested analysis model isn't currently big enough to work as well compared to other performance evaluation tools. Only a few important metrics have been incorporated into the model and it analyses for a limited number of possible inefficiencies right now. A complete analysis model would ideally consider many other different performance metrics and be able to identify several other performance inefficiencies in any application.

Expert knowledge does play a crucial role in performance analysis of HPC applications and it was made evident during the course of this study. There is limited literature that talks about performance metrics with respect to different performance bottlenecks. It becomes really difficult to understand if a program is facing a performance bottleneck or not just by looking at runtime data without understanding how it should ideally perform. If a bottleneck is identified, the next step of identifying the type of bottleneck is even more difficult.

In this master's thesis, I was able to identify metric patterns for a few different bottlenecked situations but the study needs to be widened to be able to create a complete analysis model that encapsulates several different kinds of performance inefficiencies. To achieve this, either programs that can help benchmark the system while facing a performance bottleneck need to be created, or more experiments need to be conducted on the CB framework with applications that are known to be bounded because of specific reasons, like the 3-D Lid Cavity Driven OpenFOAM simulation test cases used in this thesis. Runtime data from various other HPC fields need to be studied to find similar performance patterns. This would help refine the analysis model and threshold values for different metrics.

Part VI.

Appendix

A. Programs used to recreate bottleneck behaviour

A.1. Switch Case

```
1   switch (x) {
2       case 1:
3           d = rand() % 57 + 1;
4           res = a/d;
5           break;
6       case 2:
7           d = rand() % 5 + 1;
8           res = pow(a, d);
9           break;
10      case 3:
11          res = log(a);
12          break;
13      case 4:
14          d = 1/(rand() % 3 + 1);
15          res = pow(a, d);
16          break;
17      case 5:
18          res = sin(a*PI/180.0);
19          break;
20      case 6:
21          res = cos(a*PI/180.0);
22          break;
23      case 7:
24          res = tan(a*PI/180.0);
25          break;
26      case 8:
27          res = atan(a*PI/180.0);
28          break;
29      case 9:
30          res = sqrt(a);
31          break;
32      case 10:
33          d = (double)(rand() % 123 + 1);
34          res = fmod(a, d);
35          break;
36      ...
37
38      default:
39          std::cout << "default\n";
40          break;
41      }
```

Listing A.1: Switch statement from the switch case program

A.2. Expensive Instructions

```
1     for (int i = 0; i < loop_count; i++) {
2         std::swap(v_double[0], v_double[1]);
3         sum += calculate_average(v_double);
4         double sq = pow(sum, 2);
5         double cube = pow(sum, 3);
6         double log = log10(sum);
7     }
```

Listing A.2: For loop in the program where several expensive instructions are executed at every iteration

A.3. Analysis strategies suggest in PerSyst [GC15]

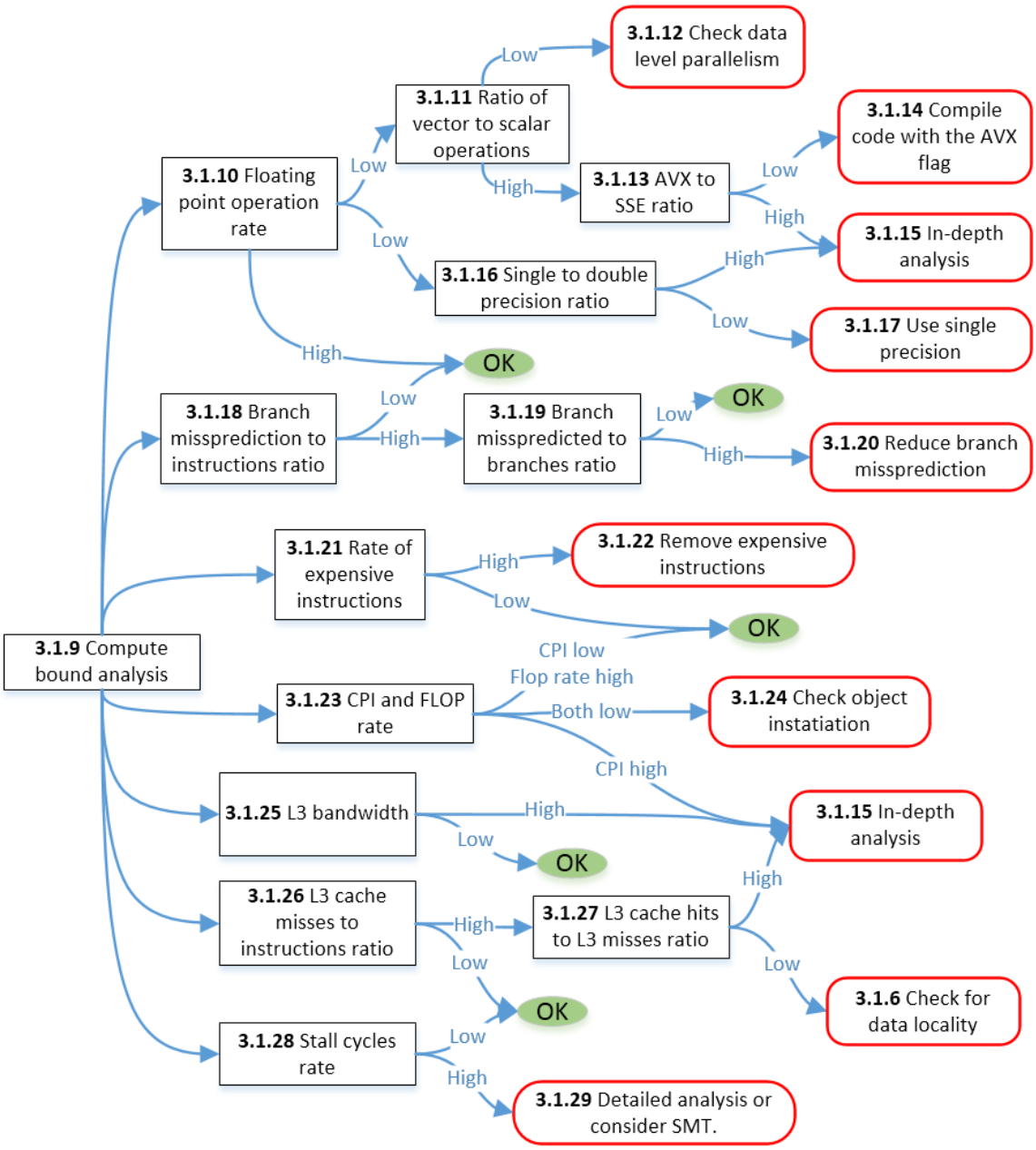


Figure A.1.: Strategies used to differentiate between different performance bottlenecks

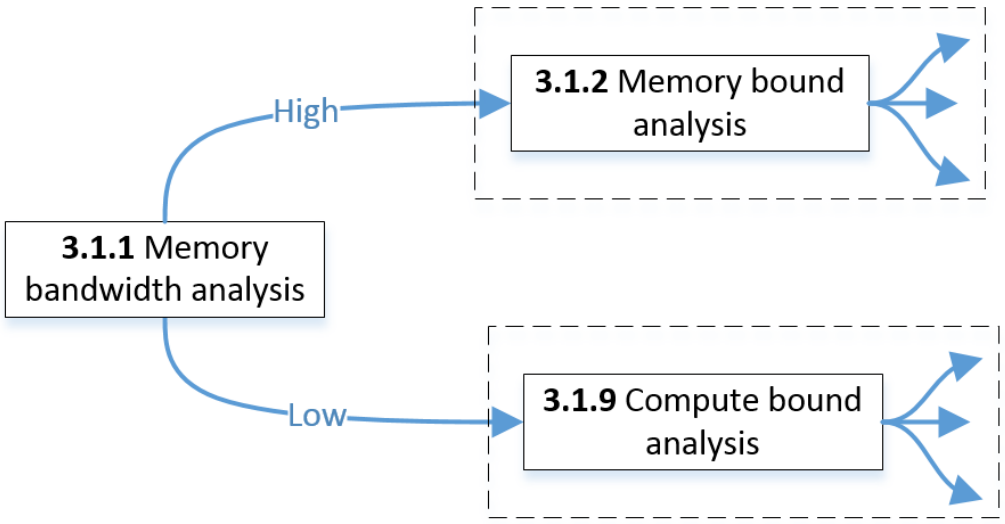


Figure A.2.: Memory bandwidth is used to differentiate between memory bound and compute bound behaviour

List of Figures

3.1. Memory access latency versus used memory bandwidth Source: [ARR17] 12
3.2. Memory latency vs size of a 1D array Source: [Sel] 13

5.1. Example of an analysis tree 25
5.2. Flowchart to show the proposed analysis process 28

6.1. Runtime Memory Bandwidth for LULESH100 and openfoamM with three different threshold values provided in table 6.2 35
6.2. Peak sustainable memory bandwidth values measured using the likwid-bench stream benchmark for different data sizes 36
6.3. Proposed analysis model for memory bound analysis 36
6.4. Proposed analysis model for CPU bound analysis 42
6.5. This figure shows how the levels are defined for the analysis model 43
6.6. The proposed analysis model used by the prototype. Terms used: MemBand - peak memory bandwidth measured by the likwid-bench stream benchmark with the appropriate data size, CacheBound - peak cache bandwidth measured by the likwid-bench stream benchmark with the appropriate data size, PeakFlopsScalar - peak DP FLOP rate for scalar operations measured by the likwid-bench peakflops microbenchmark. 45
6.7. The analysis model tab shows a list of all the analysis models passed to the tool. Index lists the total number of models present. It shows the level of the node, the metric, condition and threshold against which the values are checked. Based on the result, the column yes and no suggest the next steps to be taken in the analysis process. Suggestions can be - a list of metric, condition and threshold to analyse next or an OKAY or a possible bottleneck identified by the model. 45
6.8. Memory Bandwidth for experiment KRIPKE160 46
6.9. The first graphs shows the CPU Usage plot, the second graph shows the L3 Bandwidth and the third graphs shows the L3 Miss ratio for the 3-D Lid Driven Cavity test case M 47
6.10. The roofline model of the system architecture used in this thesis. The peak FLOP rate vs the peak memory bandwidth of a few experiments are marked in blue triangles. 48

A.1. Strategies used to differentiate between different performance bottlenecks . 55
A.2. Memory bandwidth is used to differentiate between memory bound and compute bound behaviour 56

List of Tables

6.1. Test cases for the 3-D Lid Driven Cavity 32
6.2. Peak sustainable memory bandwidth values obtained from different benchmarks (MB/s) 34
6.3. Theoretical peak FLOPS (GFLOPS) for DP and SP floating point operations 37
6.4. Peak FLOPS (GFLOPS) measured using microbenchmark 37
6.5. FLOP rate, Packed, Scalar and their ratio for a few experiments. 38
6.6. FLOP rate, CPI and Operational Intensity for a few experiments 40

Bibliography

[acc] Section 5: Access patterns. URL: https://cs61.seas.harvard.edu/site/2019/Section5/.

[amd] Processor specifications - amd. URL: https://www.amd.com/en/products/specifications/processors.

[AR01] Matthew Arnold and Barbara G Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 168–179, 2001.

[ARR17] Kazi Asifuzzaman, Milan Radulovic, and Petar Radojkovic. 2017. URL: https://exanode.eu/wp-content/uploads/2017/04/D2.5.pdf.

[BKD+10] Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[Bly08] David Blythe. Rise of the graphics processor. *Proceedings of the IEEE*, 96(5):761–778, 2008.

[BMK+99] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[Bog] Ivica Bogosavljevic. Performance benchmarks. URL: https://github.com/ibogosavljevic/johnysswlab.

[bus] The history of intel processors. URL: https://www.businessnewsdaily.com/10817-slideshow-intel-processors-over-the-years.html.

[Can] Canonical. iostat. URL: https://manpages.ubuntu.com/manpages/bionic/man1/iostat.1.html.

[CBG+] Alexey Cheptsov, Steffen Brinkmann, José Gracia, Michael M Resch, and Wolfgang E Nagel. Tools for high performance computing.

[Cha20] Pranabananda Chakraborty. *Computer Organisation and Architecture: Evolutionary Concepts, Principles, and Designs*. Chapman and Hall/CRC, 2020.

[Che00] Chih-Cheng Cheng. The schemes and performances of dynamic branch predictors. *Berkeley Wireless Research Center, Tech. Rep*, 2000.

[cpp] Intel® c++ compiler 19.1 developer guide and reference. URL: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top.html.

[CRON+14] Andres S Charif-Rubial, Emmanuel Oseret, José Noudohouenou, William Jalby, and Ghislain Lartigue. Cqa: A code quality analyzer tool at binary level. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.

[dic] Bottleneck. URL: https://www.dictionary.com/browse/bottleneck.

[Don87] Jack J Dongarra. The linpack benchmark: An explanation. In *International Conference on Supercomputing*, pages 456–474. Springer, 1987.

[EBB+14] Todd Evans, William L. Barth, James C. Browne, Robert L. DeLeon, Thomas R. Furlani, Steven M. Gallo, Matthew D. Jones, and Abani K. Patra. Comprehensive resource use monitoring for hpc systems with tacc stats. In *2014 First International Workshop on HPC User Support Tools*, pages 13–21, 2014. doi:10.1109/HUST.2014.7.

[ESE06] Stijn Eyerman, James E Smith, and Lieven Eeckhout. Characterizing the branch misprediction penalty. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 48–58. IEEE, 2006.

[Fog06] Agner Fog. Optimizing software in c++. *URL: http://www. agner. org/optimize/optimizing_cpp. pdf*, 2006.

[For14] O Forth. Entrywise addition of two double arrays using avx, 2014. URL: https://stackoverflow.com/a/27204877.

[GC15] Carla Beatriz Guillén Carías. *Knowledge-based Performance Monitoring for Large Scale HPC Architectures*. PhD thesis, Technische Universität München, 2015.

[GFG12] Pawel Gepner, David L Fraser, and Victor Gamayunov. Evaluation of the 3rd generation intel core processor focusing on hpc applications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2012.

[Gra19] Brian Gravelle. Understanding the performance of hpc applications. 2019. URL: https://www.cs.uoregon.edu/Reports/AREA-201903-Gravelle.pdf.

[Hei14] Jan Heichler. An introduction to beegfs, 2014. URL: http://www.beegfs.de/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf.

[HH15] Sarah L Harris and David Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2015.

[HN11] Alex Hutcheson and Vincent Natoli. Memory bound vs. compute bound: A quantitative study of cache and memory bandwidth in high performance applications. *Stone Ridge Technology, Internal White Paper*, 2011.

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[HYSW17] Peter Harrington, Wucherl Yoo, Alexander Sim, and Kesheng Wu. Diagnosing parallel i/o bottlenecks in hpc applications. In *International Conference for High Performance Computing Networking Storage and Analysis (SCI7) ACM Student Research Competition (SRC)*, 2017.

[iora] Ior lustre wiki. URL: `https://wiki.lustre.org/IOR`.

[iorb] Ior official documentation. URL: `https://ior.readthedocs.io/en/latest/`.

[IOT14] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proceedings of the VLDB Endowment*, 8(3):293–304, 2014.

[JWN10] Bruce Jacob, David Wang, and Spencer Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[KB22] Adam Kunen and David Beckingsale. Llnl/kripke: Kripke is a simple, scalable, 3d sn deterministic particle transport code, May 2022. URL: `https://github.com/LLNL/Kripke`.

[KBB15] Adam J Kunen, Teresa S Bailey, and Peter N Brown. Kripke-a massively parallel transport mini-app. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2015. URL: `https://www.osti.gov/servlets/purl/1229802`.

[KRB+12] Andreas Knüpfer, Christian Rössel, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E Nagel, et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

[lul] Lulesh. URL: `https://asc.llnl.gov/codes/proxy-apps/lulesh`.

[M+95] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19-25), 1995.

[McC] John McCalpin. Stream benchmark. URL: `https://www.cs.virginia.edu/stream/ref.html#size`.

[MFG16] Robert Mijaković, Michael Firbach, and Michael Gerndt. An architecture for flexible auto-tuning: the periscope tuning framework 2.0. In *2016 2nd International Conference on Green High Performance Computing (ICGHPC)*, pages 1–9. IEEE, 2016.

[mic] Micro benchmarking. URL: https://hpc-wiki.info/hpc/Micro_benchmarking.

[MMCS10] Allen D Malony, John Mellor-Crummey, and Sameer S Shende. Measurement and analysis of parallel program performance using tau and hpctoolkit. *Performance Tuning of Scientific Applications. CRC Press, New York*, 2010.

[mpi] Introducing intel® mpi benchmarks. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-mpi-benchmarks.html.

[nac] Cpi rate. URL: http://portal.nacad.ufrj.br/online/intel/vtune2017/help/GUID-83F8DBA1-8657-11E6-9991-28D24465DA3C.html.

[NFA+16] Philipp Neumann, Hanno Flohr, Rahul Arora, Piet Jarmatz, Nikola Tchipev, and Hans-Joachim Bungartz. Mamico: Software design for parallel molecular-continuum flow simulations. *Computer Physics Communications*, 200:324–335, 2016. URL: https://www.sciencedirect.com/science/article/pii/S0010465515004129, doi:https://doi.org/10.1016/j.cpc.2015.10.029.

[Nil23] Håkan Nilsson. Piso in icofoam, Jan 2023. URL: http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2019/lectureNotes/thePISOalgorithmInIcoFoam.pdf.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[ope] 2.1 lid-driven cavity flow. URL: https://www.openfoam.com/documentation/tutorial-guide/2-incompressible-flow/2.1-lid-driven-cavity-flow.

[Pet04] Antoine Petitet. Hpl-a portable implementation of the high-performance linpack benchmark for distributed-memory computers. *http://www. netlib. org/benchmark/hpl/*, 2004.

[PFM+20] Arnab K Paul, Olaf Faaland, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Ali R Butt. Understanding hpc application i/o behavior using system level statistics. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 202–211. IEEE, 2020.

[PGB14] Bertrand Putigny, Brice Goglin, and Denis Barthou. A benchmark-based performance model for memory-bound hpc applications. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 943–950. IEEE, 2014.

[RHa] Rrze-Hpc. Likwid bench · rrze-hpc/likwid wiki. URL: https://github.com/RRZE-HPC/likwid/wiki/Likwid-Bench.

[RHb] Rrze-Hpc. Likwid/groups/clx at master · rrze-hpc/likwid. URL: https://github.com/RRZE-HPC/likwid/blob/master/groups/CLX.

[RH01] Vinodha Ramasamy and Robert Hundt. Dynamic binary instrumentation on ia-64. In *Proceedings of the First EPIC Workshop*, 2001.

[S+03] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[Sel] Margo Seltzer. Strided access patterns. URL: `https://www.youtube.com/watch?v=SVqKqHHSv58`.

[SMMC07] S. Shende, A. Malony, S. Moore, and D. Cronk. Memory leak detection in fortran applications using tau. In *2007 DoD High Performance Computing Modernization Program Users Group Conference*, pages 387–393, 2007. `doi:10.1109/HPCMP-UGC.2007.47`.

[SSBA+] Ivan Spisso, Simone Simone Bna, Giorgio Amati, Giacomo Rossi, and Fabrizio Magugliani. Hpc benchmark project. URL: `https://develop.openfoam.com/committees/hpc`.

[SZ19] Sherif Sakr and Albert Y Zomaya. *Encyclopedia of big data technologies*. Springer International Publishing, 2019.

[the] Theoretical maximum memory bandwidth for intel® core™. URL: `https://www.intel.com/content/www/us/en/support/articles/000056722/processors/intel-core-processors.html`.

[THW10] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th international conference on parallel processing workshops*, pages 207–216. IEEE, 2010.

[THW12] Jan Treibig, Georg Hager, and Gerhard Wellein. Best practices for hpm-assisted performance engineering on modern multicore processors. *arXiv preprint arXiv:1206.3738*, 2012.

[Tip21] Nico Tippmann. Automation of continuous benchmarking for high-performance computing, 2021.

[Vor13] Ivan Voroshilin. What you should know about locality of reference, Nov 2013. URL: `https://ivoroshilin.wordpress.com/2013/02/06/know-your-locality-of-reference-some-techniques-for-keeping-data-in-the-cpu-cache/`.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[xeo] Intel® xeon® gold 6252 processor (35.75m cache, 2.10 ghz) - product specifications. URL: `https://www.intel.com/content/www/us/en/products/sku/192447/intel-xeon-gold-6252-processor-35-75m-cache-2-10-ghz/specifications.html`.