Hochschule Rhein-Waal

Rhine-Waal University of Applied Sciences

Faculty of Communication and Environment

Prof. Dr. Frank Zimmer

M.Sc. Florian Krebs

Exploration of Intel Resource Director Technology on PikeOS

Bachelor Thesis

by

Naveen Narayanan

Abstract

Systems with mixed-criticality suffer from issues like noisy neighbor caused by a misbehaving application hogging shared resources, thereby, affecting the latency of applications with a higher level of criticality. Such issues are of substantial importance in mission-critical systems such as the field of avionics where failure to meet hard deadlines can be fatal. Intel introduced a framework called Intel Resource Directory Technology (IRDT) which provides mechanisms to mitigate such issues by enabling allocation and monitoring of shared resources such as cache and memory. PikeOS, being a real-time system, will benefit from having support for IRDT, hence, a driver named CAT Kernel driver (ICAT) has been implemented, which drives IRDT and exposes an API that can be used by applications to monitor usage of and allocate shared resources. Using the API, applications can make intelligent decisions about how they would like to access Last Level Cache (LLC) in order to improve performance. Additionally, avionics standards impose stricter regulations on the usage of cache in systems with additional logical cores. ICAT helps with improving the Worst Case Execution Time (WCET) analysis by providing deterministic access to the desired portions of LLC.

Monitoring of shared resources is explored for the explicit purpose of profiling applications. Runtime analysis using such features helps with the analysis of the program in terms of both determinism and performance.

Additionally, the overall design of IRDT, and Performance Monitoring Unit (PMU), from the perspective of ICAT, is described along with the rationale behind certain choices which influenced the internal design of the driver itself. The different use-cases and the results of instrumentation and analysis of ICAT on Intel x86_64 platforms which support IRDT are detailed.

keywords: **Real-Time OS**, **Cache Allocation Technology**, **Intel® x86_64**, **PikeOS**, **LLC**, **Intel® RDT**

Contents

LIST OF ABBREVIATIONS

API Application Programming Interface.
ASP Architecture Support Package.
CAT Cache Allocation Technology.
CBM Capability Bit Mask.
CLOS Class of Service.
CMT Cache Monitoring Technology.
FSB Front Side Bus.
GP General protection fault.
IA-32 Intel Architecture 32-bit.
ICAT CAT Kernel driver.
IPC Inter-Process Communication.
IRDT Intel Resource Directory Technology.
KDEV Kernel driver Framework.
LLC Last Level Cache.
LRU Least Recently Used.
MBA Memory Bandwidth Allocation.
MBM Memory Bandwidth Monitoring.
MCP Maximum Controlled Priority.
MMU Memory Management Unit.
MSR Model-Specific Register.
PMU Performance Monitoring Unit.
PSP Platform Support Package.
PSSW PikeOS System Software.
RMID Resource Monitoring ID.
RP PikeOS Resource Partition.
SMP Symmetric Multi-Processing.
TMAM Intel Top-Down Microarchitecture Analysis Method.
TP PikeOS Time Partition.
UID User ID.
WA Write-Allocate policy.
WB Write-Back policy.
WCET Worst Case Execution Time.
WT Write-Through policy.

List of Figures

List of Tables

Chapter 1

Introduction

On mixed-criticality systems such as automobiles, the possibility of multiple levels of critical software running, especially in this day and age, has risen owing to the multitude of sub-components oriented towards convenience as well as safety. For instance, consider the case of a media center running at a significantly lower level of criticality than other safety features. If the software that drives the media center were to misbehave, although process isolation prevents one process from bringing down the entire system, it can definitely exhaust shared resources such as cache, memory bandwidth etc., thereby, affecting the access times of other cores assigned to safety critical applications which are running at a higher level of criticality. Such disturbance is of significance in Symmetric Multi-Processing (SMP) systems which have more than one CPU trying to access shared resources leading to contention and other side-effects. Analysis and study of such disturbances is not a straight-forward process especially when performed on platforms with no hardware support. Hence, these topics are explored with the help of IRDT on Intel x86_64 based platforms.

Allocation of shared resources in mixed-criticality systems for the purpose of avoiding interference, contention and other side-effects rising from multiple processors trying to access a shared resource is the topic of concern. Such systems suffer from issues like noisy-neighbor which inadvertently affects the behavior of applications running at a higher level of criticality. Such problems are more common in cloud data centers due to the extensive use of shared resources (Lorido-Botran et al., 2017, p.188). Lorido-Botran et al. (2017) argue that noisy-neighbor effect is basically an anomaly which restricts other partitions from using the shared resources and pursues an approach which measures different lags in the system. The approach that is pursued in this research is one that utilizes hardware frameworks such as IRDT.

IRDT is a framework provided by Intel for the purpose of allocation and monitoring of shared resources. It has the following subcomponents: CAT, CMT, MBA etc. which are used for the allocation and monitoring of resources such as last level cache, memory bandwidth etc. Additionally, the implementation of ICAT which is a kernel driver written on PikeOS for the explicit purpose of driving IRDT is discussed. Ultimately, it does some analysis of cache statistics and possible use-cases in avionics environments which warrant deterministic access to cache.

1.1 Motivation

Jean et al. (2012, chap. 9) discuss the requirements for embedded systems on aircraft which include specifications about SMP environments that involves caches. Jean et al.

(2012, p.86) argues that the use of shared caches in embedded aircraft systems warrants solutions to the following problems:

- Prediction of cache content which deals with accesses to cache being deterministic (Jean et al., 2012, p.86).
- Integrity of cache content (Jean et al., 2012, p.87).
- Impact of concurrent access in SMP environments which deals with cache coherency (Jean et al., 2012, p.87).

Prediction of cache content can be accomplished in multiple ways. It can be approached from the perspective of the programmer if there is full visibility into what the program is doing at any moment in time. However, this is not feasible in SMP environments. Another approach, according to Jean et al. (2012, p.88), would be use cache content prediction algorithms, however, such algorithms have not been used in the industrial world yet. The other classical approaches to this problem include Partitioning of cache and Configuration of Cache as SRAM (Jean et al., 2012, p.89). One of these approaches, namely, cache partitioning is supported by IRDT on Intel x86_64 platforms.

1.2 Problem Statement

The approach of using cache partitioning in order to improve deterministic access to shared caches and prediction of cache content is explored using IRDT on PikeOS. Cache partitioning is defined as the ability to allocate specific areas of cache to one core by Jean et al. (2012, p.89). As a result, the respective CPU is allowed to allocate data and instructions in the respective portion of the shared cache. This can be further extended to allocating a specific portion to multiple CPUs which is what is implemented by Intel using IRDT. Additionally, IRDT allows for different configurations of shared cache such as isolated and shared access.

Apart from that, it allows for monitoring of shared resources which can be used to make runtime decisions about the allocation of cache associated with a specific CPU or set of CPUs. The implementation of such a driver that handles IRDT on PikeOS is called ICAT; the exploration of IRDT for the purpose of cache partitioning in order to improve WCET and reduce contention on mission-critical systems such as airborne systems is the approach taken to address said problem. Hence, the real question is whether an approach exists, at least on Intel based platforms, to attack the problem of cache partitioning.

1.3 Methodology

As discussed earlier, the problem of cache partitioning can be solved by utilizing hardware frameworks such as IRDT on Intel platforms. Although Cache Allocation Technology (CAT) in IRDT has been used to achieve fairness in other OSes like Linux as discussed by Selfa et al. (2017, p. 194), such techniques have not been used for the purpose of cache prediction on PikeOS yet.

The first step of this approach is to implement a device driver on PikeOS which drives IRDT. The driver should be capable of handling the partitioning of last level cache such as L3/L2. Once this has been accomplished, the next step is to be able to verify whether the allocation is successful.

Since Intel provides Cache Monitoring Technology (CMT) as part of IRDT, it can be used to monitor the cache used by a CPU or set of CPUs in a fine granular manner. Having

both of these functionalities should be enough to partition the cache and monitor usage. However, the possibility of being able to alter the cache allocated to a specific CPU or set of CPUs during runtime can be explored as well.

1.4 Thesis structure

Chapter 2 discusses related work in the field giving a glimpse into what could be done using CAT on PikeOS. Since the prime motive is the implementation of a driver, ICAT, that makes use of architectural components like IRDT, PMU, and embedded environments like PikeOS, it would help the prospective reader alleviate the coarse terrain of systems programming if the programming environment were to be introduced first. Hence, Chapter 3 deals with the intricacies of PikeOS as a programming environment and how it contrasts with other OSes.

Secondly, a primer on Cache is provided in Chapter 4 owing to its importance in SMP environments and for a bit of history. It introduces concepts such as invalidation, flushing and cache-coherency which are important in understanding how the hardware is able to achieve better performance using caches and how OSes are able to leverage this piece of hardware to work efficiently in a multi-threaded environment.

Once these primary topics are covered, IRDT, its architecture and usage is discussed in detail in Chapter 5. Chapter 6 covers the architecture of PMU and its usage.

Ultimately, this leads to Chapter 7 which discusses the implementation of ICAT, that makes use of all the other components that were discussed prior, in meticulous detail along with its use-cases and some empirical information of the same in certain microarchitectures and platforms is discussed in Chapter 8. Chapter 9 talks about the overall result of the research and how it can be improved so as to address other issues related to shared resources.

Chapter 2

Related Work

Intel (2015) introduced CAT as a solution to the problem of having to invalidate and evict cache lines on systems with additional cores, thereby, negatively impacting the performance of latency sensitive workloads. The solution allowed system administrators to allocate certain portions of cache to a CPU or set of CPUs. This problem came into being due to the emergence of additional logical cores on processors which accelerated the rise in the number of threads running in parallel (Intel, 2015, p.5). Since LLC was shared among CPUs, invalidating and flushing the former inadvertently caused significant portions of L1 and L2 caches to be evicted too. Systems which handled a higher frequency of interrupts were affected since the interrupt service routines had to be fetched again post cache eviction (Intel, 2015, p.5).

Kim et al. (2019) provides evidence for the usage of CAT and IRDT, indirectly, for the purpose of predicting application performance by training a model using the profiling data obtained using Intel Top-Down Microarchitecture Analysis Method (TMAM). This research argues that the predictive management technique which utilized CAT in runtime outperforms static cache configuration by computing the optimal cache size required for applications. Our implementation of ICAT exposes the API that can be used to implement such mechanisms on PikeOS. PikeOS, being a real-time OS, relies heavily on hard deadlines which have to be met. Hence, ICAT enhances it by providing an option to mitigate issues such as those described above especially when handling latency sensitive workloads.

Farshin et al. (2019) talks about utilizing LLC using a memory management scheme called slice-aware memory management for the purpose of keeping frequently used data in LLC for faster access. This research takes a stab at improving network I/O by using a CacheDirector which places the header of a network packet in LLC that is closest to the CPU which is processing said packet. Cache-aware memory management presented in this research is a likely candidate for future work which can be accomplished by utilizing ICAT on PikeOS. Fast packet processing, as described in this research, is another viable use case for CAT which can help with mitigating noisy neighbor effects and improve performance.

Chapter 3

PikeOS

PikeOS is a Real-time operating system(RTOS) based on L4 microkernel authored by Jochen Liedtke. Liedtke (1995) argues that a microkernel differs from a monolithic kernel since it tries to minimize the kernel proper, which is the common part required for all programs to run. Additionally, he states that the microkernel approach provides for a much more modular design which is way more flexible and tailorable. The programming environment merely comprises of a kernel and a virtualization platform, PikeOS, coupled with an Eclipse-based IDE known as CODEO. PikeOS is currently being developed by SYSGO GmbH. The IDE is optional as the build environment is almost entirely composed of makefiles and a toolchain which are available in binary form. Since modern applications are mostly built on RTOS, there are other open source variants such as RTLinux and RTEMS which provide similar environments (Straumann, 2001, p.1). RTOS is usually used in situations which warrant rigid timing requirements and deterministic behavior (Guan et al., 2016, p.19). Guan et al. (2016) argue that the latencies of thread creation, deletion, switching etc. should be predictable, regardless of the number of threads in RTOS which is why it is different from general purpose OS.

The reason for this chapter to be inordinately longer is one that is related to the Modern Age, for those who know Unix™ would find PikeOS atypical owing to it primarily being a hypervisor. So, this chapter shall attempt to abridge the gap between a monolithic kernel like Unix™ and a microkernel such as PikeOS.

Unlike Unix™, PikeOS is not a general purpose operating system by design, although, it can be used as one. Owing to its innate support for virtualization, PikeOS enforces the concept of partitions on which processes are run. Processes can be of different kinds:

- PikeOS Native (SYSGO, 2025d, p.191)
- POSIX (SYSGO, 2025d, p.22)
- APEX (SYSGO, 2025d, p.22)
- P4-Linux (SYSGO, 2025d, p.22)

The partitions are called Virtual Machines (VM) as the processes which run on them can be full-fledged operating systems (SYSGO, 2025d, p.12).

The PikeOS microkernel consists of three parts: a kernel proper which consists of the hardware independent portion, an architecture dependent portion (ASP), and a platform dependent portion (PSP) (SYSGO, 2025d, p.19). The kernel proper provides services such as process abstraction, memory management, driver frameworks (kernel and user-level), IPC, interrupt management, etc., while utilizing the ASP and PSP for specific functionality, that is provided by the architecture and platform, respectively (SYSGO, 2025d, p.20).

The figure shown below depicts a partitioned system running on PikeOS:

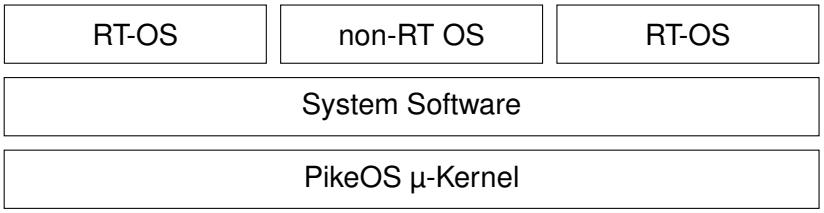

Figure 3.1: Partitioned System (Kaiser & Wagner, 2007, p.51)

As illustrated, OSes with varying levels of criticality can be run simultaneously using partitions. It is also possible to run simple applications instead of OSes in the partitions. Kaiser & Wagner (2007) confirms that although the original implementation of PikeOS was based on Liedtke (1995)'s design, it was written entirely in C and the programming interface was modified owing to several issues.

Time partitioning is another mechanism provided by PikeOS for the purpose of enforcing hard deadlines (SYSGO, 2025d, p.28). It can be used to avoid starvation as well as to ensure that an application runs for a predefined period of time.

The simplest of PikeOS Time Partition (TP) configurations allows for a certain period of time to be consumed by each of the partitions. For instance, if there are 3 PikeOS Resource Partition (RP)s, RP1 runs first, RP2 next, and, ultimately, RP3 (SYSGO, 2025d, p.28). However, the versatility of TP in PikeOS does not end there. There is no need to have a one to one relationship between resource partitions and time partitions (SYSGO, 2025d, p.28). It can be inferred that RP can be mixed and matched with TP (SYSGO, 2025d, p.28). Secondly, the period of time for which a TP runs is not a property of TP. This property, although non-intuitive, calls for some interesting configurations.

As a consequence of this design, a time partition can have multiple RPs running under different windows (SYSGO, 2025d, p.28). A window is an abstraction for the period of time for which the RP is run (SYSGO, 2025d, p.28). As one can imagine, such configurations impose stricter regulations on access latency which is why cache allocation technologies like ICAT is beneficial to RTOSes like PikeOS.

Another important aspect of PikeOS is the different kinds of environment it provides which is referred to as PikeOS Personalities (SYSGO, 2025d, p.22). They basically represent environments with different Application Programming Interface (API) and runtime environments which is expected by certain pieces of software (SYSGO, 2025d, p.22). Supported personalities include POSIX (IEEE, 2004), ARINC 653 (ARINC, 2010) used by software that is conformant to POSIX standard, ARINC standard, respectively, and P4Linux for running Linux on PikeOS (SYSGO, 2025d, p.22).

3.1 PikeOS Internals

This section shall describe the internals of PikeOS pertaining to bootstrapping & kernel initialization, memory management, process creation, and device drivers. It talks about the following: ASP, PSP, PSSW, Process Management, Threads, Scheduling and KDEV framework. In brief, ASP takes care of the architecture specific component, PSP takes care of the platform specific component, and, PSSW is the root task. All of the components are detailed in the following sections.

3.1.1 ASP

The Architecture Support Package (ASP) consists of the architecture dependent part of the kernel (SYSGO, 2025d, p.20). ASP is responsible for handling Exception Management, data transfer between address spaces and architecture dependent initialization of sub-components (SYSGO, 2025d, p.20).

Secondly, handling of exceptions on any architecture is an extremely involved aspect of virtualization, especially so, in PikeOS (SYSGO, 2025b, pp.114-115).

Address space management during bootstrapping is handled by the ASP. On top of that, context management which comes into play during preemption of a process is dependent on the architecture and, hence, falls on the ASP as well (SYSGO, 2025b, pp.114-115).

Additionally, there are several security related quirks and hacks which are implemented in the ASP due to its hardware dependent nature (SYSGO, 2025b, p.114).

3.1.2 PSP

The Platform Support Package (PSP), with the help of ASP, is responsible for bootstrapping the OS (SYSGO, 2025d, p.21). It handles the initialization of various subsystems before finally handing off control to the kernel (SYSGO, 2025c, p.21). From the programmer's perspective, it isolates components of PikeOS that are dependent on the board at both object and source code levels. Some of the subcomponents it initializes includes: interrupts, hardware timers, console, memory, etc. Additionally, caches are setup by the PSP (SYSGO, 2025d, p.21).

3.1.3 PSSW

PikeOS System Software (PSSW) is analogous to *init* in Unix™ world in the sense that it is the first process which starts to run in usermode (Neville-Neil, McKusick & Watson, 2014, p.51). Its responsibilities include reading VMIT, initializing the partitions, Inter-Process Communication (IPC) and health monitoring (SYSGO, 2025d, p.21). PikeOS has no notion of dynamic process creation. This aspect is handled statically using xml-based configuration. It provides several services related to the aforementioned setup during runtime.

3.1.4 Process Management

Processes are an important abstraction in Operating systems; PikeOS relates every process in userland to a task in kernelspace. Userspace programs spawned by PSSW are called processes (SYSGO, 2025d, p.46). The process of spawning processes by PSSW involves the following:

- Allocation of task from partition's allocated task pool (SYSGO, 2025d, p.46)
- Mappings configured in VMIT are realized (SYSGO, 2025d, p.46)
- Configured number of child tasks are donated to the respective process (SYSGO, 2025d, p.46)

A task is simply an abstraction for an address space and a set of schedulable entities (SYSGO, 2025d, p.49). PSSW is the first task which is spawned and it creates the rest of the tasks based on the configuration provided in VMIT.

A task can enter several states during its lifetime such as: passive, active, activated, started, and zombie (SYSGO, 2025d, p.51). It has a Maximum Controlled Priority (MCP) which determines when it gets scheduled in and out. It has an attribute which dictates the maximum number of threads it can spawn. And another attribute, CPU mask, which dictates the set of CPUs that will schedule the said task (SYSGO, 2025d, p.49-50). Apart from that, there are other attributes which dictate other abilities of the task and, lastly, it has a task ID (SYSGO, 2025d, p.49).

The root task, PSSW, which has a task ID of 1, is responsible for running other tasks (SYSGO, 2025d, p.50). Not only does it run other tasks, it donates the required resources too (SYSGO, 2025d, p.50).

3.1.4.1 Threads

Thread is an abstraction for the component of the task which gets scheduled (SYSGO, 2025d, p.59). As discussed earlier, a task can have more than one thread. In the same vein as a task, a thread has a number which identifies itself, a User ID (UID) which is a combination of task id and thread number, a state such as: current, waiting, ready, stopped, and inactive, a priority which can not be higher than that of the task's MCP, an affinity mask which determines the set of CPUs that might run said thread, and an assigned processor which represents the current CPU the thread is scheduled on (SYSGO, 2025d, p.59-61).

Thread affinity plays an important role in how PikeOS manages computational resources. So does it play an important role in the subject of this paper. In SMP systems, a set of CPUs can be assigned to a partition statically using *CpuMask* attribute which is available for each partition (SYSGO, 2025d, p.67). This entitles the threads running on each partition to move freely among the available CPUs (SYSGO, 2025d, p.67).
During runtime, threads can inquire and change their own affinity using the following syscalls (SYSGO, 2025a, p.242-245):

```
p4_thread_get_affinity - Get current CPU affinity
p4_thread_set_affinity - Set current CPU affinity
p4_thread_ex_affinity  - Get and Set CPU affinity
```

Once the change in affinity is made, the respective thread gets migrated over to the desired CPU.

3.1.4.2 Scheduling

The scheduler dictates which process gets scheduled in and out. In PikeOS, the priority, the probability of getting scheduled, is directly proportional to the value assigned to a thread's priority i.e. the higher the thread priority the more likely it is for the thread to get scheduled (SYSGO, 2025d, p.79).

The scheduler is awakened when a state change occurs. A state change can occur due to a couple of reasons such as a time-partition expired, a thread went to sleep, an interrupt occurred etc (SYSGO, 2025d, p.79).

The awakened scheduler is asked to make a decision on the thread to be run. This decision is made with the help of a prioritized FIFO dispatcher (SYSGO, 2025d, p.79). When a thread changes state and becomes runnable, it is placed at the end of a ready list that belongs to the respective priority (SYSGO, 2025d, p.79). Every priority has its

own ready list. The scheduler then picks the first thread from the ready list of the highest priority (SYSGO, 2025d, p.79). This implies that the thread with the highest priority will run indefinitely if it does not yield. In other words, it can easily starve off other threads if care is not taken.

The other two elusive cases include the case of a thread changing its own priority and the case of a thread's priority altered by another. In the first case, the thread will be placed at the beginning of a ready list for the new priority (SYSGO, 2025d, p.83). And when a thread changes the priority of another, the thread will be placed at the end of a ready list for the new priority (SYSGO, 2025d, p.83). This implies that a thread changing its own priority is more likely to get scheduled than one that had its priority changed by another (SYSGO, 2025d, p.83).

The scheduler used in PikeOS warrants the user to be aware of the scheduling policy and configure the system accordingly in order to avoid starvation as the scheduler does not inherently care about fairness unless time partition is used.

3.1.5 Memory Management

Like any modern operating system, PikeOS utilizes a Memory Management Unit (MMU) to make use of a virtual address space on which user processes are run (SYSGO, 2025d, p.88). Invalid accesses to memory, due to access permissions or invalid translation, will be met with page faults which is an exception thrown by the hardware itself (SYSGO, 2025d, p.88).

Every task has its own virtual address space (SYSGO, 2025d, p.88). Virtual to physical memory mapping can be made anywhere within the user addressable range and the translation is done by the MMU (SYSGO, 2025d, p.88).

Mappings can only be manipulated at a granularity level of P4_PAGESIZE which is 4096 bytes on x86-64 (SYSGO, 2025b, p.96). The unit of mapping is called a page.

3.1.5.1 Mapping Attributes

The following attributes dictate various aspects of a mapping:

Task ID represents the task to which the mapping belongs (SYSGO, 2025d, p.88)

Phy Page represents the physical page to which the mapping refers (SYSGO, 2025d, p.88)

Access Permissions represents 3 different permissions (SYSGO, 2025d, p.88): P4_M_READ for read,
P4_M_WRITE for write, P4_M_EXEC for execute.

Cache Attributes represents the caching scheme used for the mapping.
P4_M_C_ENABLE enables caching; P4_M_C_WRITEBACK is used to select between write back and write through caching; P4_M_C_PREFETCH is used to enable prefetches and speculative loads; P4_M_C_COHERENCY is used to enforce memory coherency (SYSGO, 2025d, p.89).

3.1.6 Kernel Driver Framework

Kernel driver Framework (KDEV) allows for I/O drivers, which are linked against the PikeOS kernel proper, to be able to run with supervisor access (SYSGO, 2025d, p.178). PikeOS kernel drivers are generally faster since they run in kernelspace and hence don't require a syscall instruction to make use of some functionality provided by the kernel. The downside is that they should be implemented with care owing to it being executed in kernelspace which has fewer restrictions.

The interface provided by kernel drivers to a prospective user is called a gate. The functionality provided by the gate depends on the implementation of the driver and the services provided (SYSGO, 2025d, p.178). The bare minimum functionality provided by a generic driver would include the following:

- **vm_open** which would open the respective gate; allocate a gate descriptor (SYSGO, 2025d, p.178)
- **vm_close** which would close the respective gate; free the respective descriptor (SYSGO, 2025d, p.178)
- **vm_read** which would allow the user to read data from the device (SYSGO, 2025d, p.178)
- **vm_write** which would allow the user to write data to the device (SYSGO, 2025d, p.178)
- **vm_ioctl** which can be used by the driver to provide some specific functionality (SYSGO, 2025d, p.178)

Gate providers, as stated above, should be linked against the kernel and PikeOS toolchain achieves this using a kernel fusion project which fuses the driver with the kernel (SYSGO, 2025d, p.180). The resultant kernel is then used along with the userspace application, using an integration project, which makes use of the of kernel driver.

This concludes the discussion about the programming environment that PikeOS provides. The next chapter deals with the characteristics of Cache and the rationale for its existence.

Chapter 4

Caches in SMP Systems

This chapter discusses the importance of cache on SMP systems, however, one needs to understand the rationale behind its implementation in order to fathom its usage.

The implementation of cache stems from the idea that code is usually executed sequentially and most of the time only a subset of the program's code is used repeatedly. In other words, cache is able to improve system performance by exploiting *locality of reference* (Schimmel, 1994, p.24). This property of programs can be exploited to speed up system performance by storing the program's current locality in a kind of memory which is faster than main memory - cache (Schimmel, 1994, p.24).

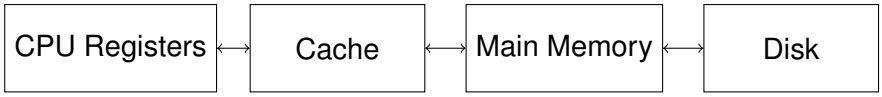

Figure 4.1: Location of Cache (Schimmel, 1994, p.24)

Being only a component in the chain of memories, cache lies in the middle of the spectrum where at one end lies the fastest (registers) and the other end lies the slowest (disk). The cost of access increases as shown above from disk to registers (Schimmel, 1994, p.24).

There are two types of localities: *Temporal* and *Spatial*. Temporal locality is the property that programs are likely to reuse recently referenced items (Schimmel, 1994, p.25). Consider the case of an operating system like PikeOS, ultimately what runs is a loop which calls the scheduler. The scheduler chooses a program to run and when it yields it chooses another one based on some specification and runs it. From a very abstract point of view, most gui applications also seem to do the same thing; they wait till they are asked to do something specific and then performs the desired operation. In such cases, a small subset of components are reused and, hence, referenced repeatedly.

Spatial locality is the property that programs are likely to reference items that are near previously referenced items (Schimmel, 1994, p.25). Sequential execution of programs causes items which are closer to previously used items to be referenced and reused. Consider the case of arrays in C, items referenced are adjacent to those previously used. And, on subsequent iterations, the next adjacent element is referenced (Schimmel, 1994, p.25).

The difference in speed between the CPU and the main memory can cause the CPU to be limited to the speed of main memory. If the speed of main memory is considerably low, this would indeed reflect in degraded performance and increasing the CPU speed

would not have any significant impact at all (Schimmel, 1994, p.25). This is the rationale for having a relatively smaller but faster memory, in the memory hierarchy, which would offset the imbalance in speed between the CPU and the main memory (Schimmel, 1994, p.25).

4.1 Cache Fundamentals

Caches are found on all kinds of systems ranging from PCs to supercomputers. They are usually contained within the CPU or MMU as the proximity to the CPU reduces the access time between the CPU and the cache (Schimmel, 1994, p.26). Cache can be found in many sizes ranging from a few KB to MB. The larger the cache the better the performance owing to the fact that a larger locality of reference is available for faster access (Schimmel, 1994, p.26). Although, this is the true when comparing systems with cache to those with no cache, performance is also dependent on other factors such as the behavior of the program itself. This is illustrated in Chapter 8.

Caches on Intel Architecture 32-bit (IA-32) were implemented due to the frequent core stalls caused due to the CPU trying to access the prefetch queue at a fairly high frequency and the main memory being slow, thereby, causing the prefetch queue to get exhausted (Shanley, 2005, p.386). Secondly, in case of a branch instruction such as an interrupt or exception, the CPU would have to flush the prefetch queue and stall until the Front Side Bus (FSB) completes code fetch (Shanley, 2005, p.386). It would have to do the same thing in case of a memory load since the data would have to be fetched from main memory and placed in the target register. In a similar vein, it would have to stall until a write to main memory finished (Shanley, 2005, p.386).

Caches are usually designed in such a way that they are transparent to both the systems programmer and the user. This helps with moving programs from one system to another with a different cache hierarchy without any considerable changes, thereby, making it portable (Schimmel, 1994, p.26). However, on certain architectures such as ARM this is not the case. ARM uses a weak memory model which requires the developer to use certain instructions to flush or invalidate the cache so as to maintain cache coherency (Sloss, 2004, p.423).

The portions of memory which reside in cache are called *cached* and this is accomplished by *tagging* the data in the cache with its main memory address. This makes it possible for the hardware to check if said data is in memory by checking the tags (Schimmel, 1994, p.27).

When the CPU issues a main memory address it wishes to fetch, it is sent to the cache and the search for the said address is performed by the hardware (Schimmel, 1994, p.27). If the search succeeds, it is called a *hit* and when it fails, it is called a *miss*. The frequency of hits to misses is called the hit ratio and it is represented as a percentage of the hits to the total number of references made (Schimmel, 1994, p.27). If a hit occurs, the data is returned to CPU. But the architecturally intriguing, although computationally taxing, case is when a miss occurs. In this case, the address is passed to the main memory and the data from the location is returned back to the CPU through the cache. This is done so that future references will result in a hit (Schimmel, 1994, p.27).

On IA-32 processors, when the CPU requests for data or code and the cache lookup results in a miss, the CPU has to arbitrate for ownership of FSB and, then, initiate a memory transaction to access the data from main memory (Shanley, 2005, p.387).

The next section discusses the implementation of cache line which is the most fundamental unit of cache, like second being the unit of time.

4.1.1 Cache Line

A cache line is a group of one or more contiguous words of main memory that has an associated tag which distinguishes itself. Hence, a cache consists of *data* and a *tag* (Schimmel, 1994, p.28). Another parameter of a cache line is its *line size* which refers to the number of bytes it holds. Since the data portion of a cache line contains contiguous memory, it need only hold the first address as the addresses of the rest can be inferred from the position of the respective data in the line (Schimmel, 1994, p.28).

Apart from this, the tag portion needs some mechanism to represent whether the data which it holds is valid, as in if it actually reflects the data in memory. This is accomplished with the help of a *valid* bit that tells whether or not the associated line is in use and contains valid data (Schimmel, 1994, p.28).

"For a match to occur, the valid bit must be on and the tag must match" (Schimmel, 1994, p.28)**.**

On IA-32 processors, as discussed earlier, the cache is not only used to fetch data from memory when requested but also to fetch data in case of a cache miss. The address of the first byte stored in the data portion of the cache line always starts at an address boundary that is evenly divisible by the cache line size (Shanley, 2005, p.388). When a load miss happens, the requested data is routed to the processor's execution unit, once it is fetched and stored in the cache (Shanley, 2005, p.388).

Another flag that is used to represent whether or not the data in the cache line has been modified is called a *modified* bit. This bit is used when the cache is configured as write-back cache (Schimmel, 1994, p.28). This will be explained in detail later on.

When a cache miss occurs, data from the memory is used to replenish the entire cache instead of the mere byte or word that the cpu needs (Schimmel, 1994, p.28). This is done so as to exploit spatial locality of programs and modern systems are designed to read or write multiple words at a time (Schimmel, 1994, p.28).

Some implementations of cache use very long cache lines such as 64B, 128B etc. so as to reduce the memory required for the storage of tags since one tag now covers more bytes in the data portion of memory (Schimmel, 1994, p.28). The disadvantage of this approach is that now more bytes need to be transferred from memory to fill the respective cache line. To mitigate the performance loss, some implementations divide the cache lines further into multiple sublines each with its own valid bit (Schimmel, 1994, p.28).

Sublines, with their own valid bit, can be handled as if they were cache lines except that there is only a single address tag covering all sublines in a line. And its position in the cache line can be used to figure out the address of each subline (Schimmel, 1994, p.29). However, this is an implementation detail which is almost always transparent to the operating system (Schimmel, 1994, p.29).

Armed with the information related to implementation, operation of cache and write and replacement policies are discussed next.

4.1.2 Cache in Operation

When the cpu requests data from the cache by passing the address over the address lines, the speed at which the cache is searched is paramount, since, the whole purpose of cache is to be faster than the main memory. Linear search algorithms are too slow so they are only beneficial in very small caches (Schimmel, 1994, p.29).

Most caches use hash tables for searching. The search can be broken down to the following: the address from the CPU is hashed to produce an index, this index is used to access the hash bucket where the data is stored (Schimmel, 1994, p.29). As with any hashing algorithm, different addresses will produce the same index so the time complexity is not strictly O(1) all the time (Schimmel, 1994, p.29).

The tags at these locations have to compared with the address given by the CPU to find the match (Schimmel, 1994, p.29). A hit occurs when the tag matches and otherwise it is a miss. Since hashing limits the search to a small set of one ore more locations it is much faster than linear search (Schimmel, 1994, p.29). All of this is done without any software intervention.

The next section deals with what happens when the cache holds invalid data.

4.1.3 Replacement Policies

The data in the cache has to be discarded when the data in memory changes, during a cache miss operation (Schimmel, 1994, p.29). The data to be discarded has to chosen based on the replacement policy of the cache, which is implementation defined. Once the data is selected, the respective cache line is replenished with new data and the address in the tag is also updated (Schimmel, 1994, p.30).

Cache replacement policies have to be stateless since maintaining state inorder to update cache is computationally taxing and not feasible in most systems (Schimmel, 1994, p.29). Typical cache replacement policies include: Least Recently Used (LRU), pseudo-LRU and random replacement (Schimmel, 1994, p.30).

The next section discusses the policies used by the cache when a store happens.

4.1.4 Write Policies

Write policies come into play when a store operation takes place. Most caches store the data, during a store operation, into the cache directly so as to exploit temporal locality since most of the time the data that is written is re-read (Schimmel, 1994, p.30). The second reason is that cache is much faster than main memory so writing data to cache is way more efficient (Schimmel, 1994, p.24). The cache's write policy dictates when the data is written back to main memory.

Initially, the cache is searched to see if the respective data is present. If a hit occurs, the data in the cache is replaced with new data. As stated above, whether the new data is written back depends on the write-policy (Schimmel, 1994, p.30). The two possible write-policies are write-through and write-back.

4.1.4.1 Write-Through

If Write-Through policy (WT) is used, the data from the CPU is written to both cache and main memory. It takes its name from the fact that writes have to go through the cache

into main memory (Schimmel, 1994, p.30). The advantage of this policy is that the cache and main memory is always in sync as in data in the cache is identical to that in main memory. The disadvantage is that a main memory cycle is used for every store operation and, hence, is computationally taxing (Schimmel, 1994, p.31).

4.1.4.2 Write-Back

If Write-Back policy (WB) is used, data is written to cache but not to main memory until it is forced out during line replacement or explicitly written to memory by the operating system (Schimmel, 1994, p.31). Schimmel (1994) states that this avoids having to use a main memory cycle after each store operation as the data can change any number of times in the cache without any significant performance penalty. However, the disadvantage is that the contents of main memory can become stale or inconsistent with respect to the cache. Hence, operating system intervention is required to maintain coherency (Schimmel, 1994, p.31).

Cache coherency is of significance especially in systems which have multiple processors due to the presence of multiple observers of cache and memory. Although, in the case of load operation, the stale value in memory will not be read since the cache will be searched first and the latest value will be found (Schimmel, 1994, p.31).

The cache lines in WB caches can be replaced anytime due to a subsequent miss, hence, the updated value in the line should be written to main memory before being discarded. This is done transparently by the cache hardware without any OS interaction. This warrants another bit called *modified* bit which is set when the cache line gets updated (Schimmel, 1994, p.32). Schimmel (1994) states that it is cleared when the data is written back to main memory. This makes it possible for the cache to write back only those lines whose data portions have been updated. So the advantages of having WB caches include: fewer main-memory operations, fewer bus operations and overall better performance (Schimmel, 1994, p.32). The disadvantage is that from time to time the OS will need to flush the cache inorder to maintain consistency (Schimmel, 1994, p.32).

When a cache miss occurs, the following write policies come into play.

4.1.4.3 Write-Allocate

Write-Allocate policy (WA) of cache comes into play when a store operation happens and a cache miss occurs. In this case, if WA is used, the data is always written to the cache by evicting data off a cache-line (Schimmel, 1994, p.32). In other words, the mechanism to process a load/read miss is used to first bring data from main memory to cache.

Schimmel (1994) argues that the replacement policy searches for a cache line that can be evicted to make room for new data. If the said cache line has the modified bit on, it is written to main memory, in case of WB. Then the full cache line is read from main memory into the cache and the data written by CPU is inserted into the line. However, if the data written by the CPU is equal to the line size, the read from main memory can be skipped since the entire line is going to be replaced by the data anyway. Otherwise the modified bit is set, in case of WB, after the data is inserted into the cache line (Schimmel, 1994, p.33). An alternative to this is to just write the data directly to main memory. In most cases, WB cache uses WA and WT cache does not owing to hardware cost.

4.1.5 Associative Cache

Associative caches are the most common kind found in IA32 processors (Shanley, 2005, p.399). A direct mapped cache maintains a 1:1 relationship between the address of the data in main memory and the location in cache (Schimmel, 1994, p.42). A two-way associative cache produces a hash that resolves to a set of two lines in cache where the data might be stored (Schimmel, 1994, p.42). A fully associative cache increases the size of this set to include all the cache lines in cache (Schimmel, 1994, p.45). Hence, only a single set exists in a fully associative cache and it includes all the lines in the cache (Schimmel, 1994, p.45).

Such a design minimizes cache thrashing since the program will be able to achieve 100% hit ratio if its locality of reference is less than or equal to the cache size (Schimmel, 1994, p.45).

4.1.6 Cache Flushing

The OS usually takes the ability to remove the data from the cache for granted. And the process of doing this is called *cache flushing* (Schimmel, 1994, p.46). This could be necessary in case WB is used in order to maintain cache coherency/consistency. Flushing can be accomplished in two ways: through *validation of main memory* and *invalidation of cache*.

Schimmel (1994) argues that validation of main memory involves writing the modified data in the write-back cache to main memory. This is done by the cache automatically when a line is evicted and can be explicitly done by the OS so as to avoid other components of the system that don't use the cache from using stale data in main memory. An explicit validation will write the data to main memory and turn off the modified bit in WB caches. WT caches don't need to worry about this case since the main memory is always in sync with the cache (Schimmel, 1994, pp.46-47).

Invalidation of the cache involves discarding the data in the cache without writing it back to main memory (Schimmel, 1994, pp.46-47). It can be used by WB and WT caches and is used primarily in cases where the hardware fetches some data say from a network chip, thereby, turning the data in cache stale. Invalidating it causes a cache miss forcing the updated data from main memory to be read into the cache (Schimmel, 1994, pp.46-47).

Techniques used to improve Quality of Service, by providing cache prediction, isolation and controlled sharing, were needed to avoid contention and this has lead to the emergence of technologies such as CAT and CMT (Hendrich et al., 2016, p.1).

Chapter 5

Intel Resource Director Technology

IRDT provides a framework which helps with monitoring of shared resources such as memory bandwidth, cache allocation as well as allocation of resources like L3/L2 cache(Intel, 2025a). Intel (2025a) argues that IRDT might help with performance interference especially in configurations where multiple OSes are running concurrently. This chapter talks about the mechanisms provided by IRDT for the purpose of allocating and monitoring of shared resources. It delves into the mechanisms used to program IRDT and talks about the details pertaining to its sub-components such as CAT, CMT etc.

The following sections will focus on CAT, CMT, Memory Bandwidth Allocation (MBA) and Memory Bandwidth Monitoring (MBM).

5.1 Cache Allocation Technology

CAT provides OSes or Hypervisors with a mechanism using which the amount of cache used can be configured (Intel, 2025b, p.19-52). Depending on the microarchitecture of the CPU, the last level cache can be of type: L2 or L3.

CAT enables applications with more stringent requirements for accessing the cache to have privileged access to a predefined portion of last level cache. It also allows for dynamic resource management during runtime making it even more flexible (Intel, 2025b, p.19-53). Additionally, this enables the user, be it OS/Hypervisor, of the system to rebalance usage of resources to improve throughput (Intel, 2025b, p.19-53).

5.1.1 CAT Architecture

The basic functionality that CAT provides is the ability to join a Class of Service (CLOS) which is a priority level (Intel, 2025b, p.19-53). CAT uses CLOS to give certain applications dedicated access to resources. There can be several different CLOS levels which are exposed by the CPU so as to allow processes to join them (Intel, 2025b, p.19-53). The allocated cache is determined by the specific CLOS used.

Each CLOS is configured using a Capability Bit Mask (CBM) that dictates the capacity of cache which can be utilized and whether there is an overlap or split between the different CLOS levels. Each logical processor has a set of registers which can be programmed so as to join a specific CLOS (Intel, 2025b, p.19-53). CLOS usage is consistent across resources and having multiple resource control attributes reduces software overhead during context switch time (Intel, 2025b, p.19-53).

In short, CAT provides the ability:

- to discover if cache allocation is architecturally supported by the target cpu (Intel, 2025b, p.19-53).
- to learn about the kind of resources which can be made use of (Intel, 2025b, p.19-53).
- to learn about CLOS levels supported and the length of CBM (Intel, 2025b, p.19-53).
- to configure said CLOS using CBM (Intel, 2025b, p.19-53).
- to join a CLOS and check if it the join operation was successful (Intel, 2025b, p.19-53).

5.1.2 Capability Bit Mask

CBM basically describes how much of the cache can be used for the specific CLOS (Intel, 2025b, p.19-54). The bits allocated in CBM, as stated by Intel (2025b), represent the cache that can be used by said CLOS. It also provides a hint to the hardware regarding overlap and isolation in cache. The length of CBM can be obtained by using the CPUID instruction and will be explained in detail later.

CLOS	B6	B5	B4	B3	B2	B1	B0
CLOS0	1	1	1	1	1	1	1

Table 5.1: Default(Intel, 2025b, p.19-54)

The default bitmask contains all the bits implying the availability of the entire cache. By default, all CPUs, which support CAT, will belong to CLOS0, thereby, sharing the entirety of LLC. The next configuration that is discussed can be considered as a variant of the default configuration, in the sense that multiple CPUs share the LLC.

CLOS	B6	B5	B4	B3	B2	B1	B0
CLOS1	0	0	1	1	1	1	1
CLOS2	0	0	0	0	1	1	1

Table 5.2: Overlap(Intel, 2025b, p.19-54)

The aforementioned set of CLOSes represents an overlapping configuration. CLOS1 allows access to a much more significant amount of cache than CLOS2. However, they share some amount of cache. This allows for a nicer transition from CLOS1 to CLOS2 as the entire cache doesn't get flushed owing to the overlap.

The aforementioned set of CLOSes represents an isolated configuration since the CLOSes are completely disjoint from each other. Transition from CLOS1 to CLOS2, in this case, will be computationally taxing since the cache used by the client will have to be flushed. In all the cases described above the length of CBM is 7.

Contiguous '1' combinations is the only kind of value that is allowed to be used as CBM, unless support for it is specified in CPUID enumeration (Intel, 2025b, p.19-54). A General

CLOS	B6	B5	B4	B3	B2	B1	B0
CLOS1	0	0	0	1	1	0	0
CLOS2	0	0	0	0	0	1	0
CLOS3	0	0	0	0	0	0	1

Table 5.3: Isolated(Intel, 2025b, p.19-54)

protection fault (GP) is issued when an attempt is made to program a value that has non-contiguous '1's. The mapping between the bit in CBM and the respective amount of cache is implementation-dependent (Intel, 2025b, p.19-54). A mask bit of '1' means that the respective CLOS can allocate into the cache subset and a value of '0' means it can't.

The following mechanisms are provided by CAT for the purpose of discovery, configuration and usage:

5.1.3 Discovery

The discover phase includes enumeration of IRDT using CPUID instruction (Intel, 2025b, p.19-58). Intel (2025b) states that support of CAT can be queried by software using CPUID instruction with EAX register set to 07H and ECX register set to 0H. The dynamics of CPUID instruction will be discussed in detail later on. The CPUID leaf 10H can be used to enumerate additional details of available resources.

CAT provides the following interface:

- Available resource types can be probed using the leaf function 10H of CPUID (Intel, 2025b, p.19-58).
- IA32_L3_MASK_n represents the MSRs provided for storing CBM for each of the CLOSes. The number of CLOSes supported can be obtained using CPUID instruction with EAX set to 10H and ECX set to ResID which will depend on the last level cache used (Intel, 2025b, p.19-58).
- IA32_L2_MASK_n represents the MSRs provided for storing CBM for each of the CLOSes, thereby, providing access to L2 cache (Intel, 2025b, p.19-58).
- IA32_PQR_ASSOC.CLOS field of the respective MSR can be used to assign the processor to the respective CLOS (Intel, 2025b, p.19-58).

The following sub-functions are provided by CPUID leaf 10H (Intel, 2025b, p.19-58):

The sub-function 0, which means EAX is set to 10H and ECX is set to 0H, enumerates the following resource types:

- Bit 1 - if set implies L3 Support
- Bit 2 - if set implies L2 Support
- Bit 3 - if set implies MBA Support

The resource ID obtained from this setup is used to get more information. Any sub-function other than 0, that is sub-function 1 for L3 support, 2 for L2 support and 3 for MBA support can be used to get the length of CBM (Intel, 2025b, p.19-59). Intel (2025b) states that the Least significant 5 bits (0-4) of EAX incremented by one will give us the

length of CBM and that this mask is associated with the respective resource be it L3, L2 or MBA.

Sub-functions of CPUID 10H for the respective resource will provide more details such as the length of CBM, the maximum number of CLOSes supported etc (Intel, 2025b, p.19-59). Hence, software is responsible for querying the capabilities of each resource ID found.

CPUID 10H sub-function 1 is used to enumerate L3 CAT capability (Intel, 2025b, p.19-59). The value in EAX[4:0] is used to calculate the maximum length of CBM that is supported. Using a length greater than what is supported will eventuate in GP being thrown by the hardware. The value in EBX[31:0] represents the corresponding bitmask (Intel, 2025b, p.19-59). As described earlier, each bit set in CBM will enable a certain amount of cache to be allocated to the specific CPU. And each cleared bit in CBM represents a portion of cache that is unavailable to the said CPU.

The bit in ECX[1] represents if L3 CAT support is available for non-CPU agents (Intel, 2025b, p.19-59).

The bit in ECX[2] represents support for Code and Data Prioritization technology (Intel, 2025b, p.19-60).

The bit in ECX[3] represents support for non-contiguous capacity bitmask (Intel, 2025b, p.19-60). If this bit is set, use of non-contiguous CBM would not be illegal.

The value in EDX[15:0] represents the maximum number of CLOSes supported for the specific resource, however, the former has to be incremented by 1 to get the actual value (Intel, 2025b, p.19-60).

Extended properties of L2 can be enumerated in a similar fashion using sub-function 2 (Intel, 2025b, p.19-60).

Last but not least, migration of CLOS across logical processors will result in reduction in performance of CAT as excessive time could be used for warming up the processor caches after a migration (Intel, 2025b, p.19-61).

5.1.4 Configuration

Once the length of CBM and the maximum number of CLOSes are determined, the next step is to program each CLOS with the respective CBM (Intel, 2025b, p.19-61). This is done by writing the CBM to the respective the IA32_L2/3_MASK_n register, where 'n' corresponds to a number in the range of [0, max_clos - 1] (Intel, 2025b, p.19-61).

The next step involves setting up the CPU to use a specific CLOS. This is done by setting the CLOS field of IA32_PQR_ASSOC register. Each logical processor has an instance of IA32_PQR_ASSOC register at location C8FH (Intel, 2025b, p.19-62). Bits 63:32 represents the CLOS field which is to be programmed with the right CLOS.

If IA32_PQR_ASSOC is not set using the desired CLOS, then CLOS0 is used by default ; CLOS0 has all 1s set so the entirety of last level cache is used (Intel, 2025b, p.19-62).

5.1.5 Usage

As discussed above, the CPU uses the respective CLOS written to its IA32_PQR_ASSOC register and is allowed access to the respective portion of last level cache, as prescribed by the CBM in the said CLOS. Apart from this, it is also possible to access and update Intel Resource Director Technology Resources such as CAT in real-time.

Updating the registers is done using RDMSR and WRMSR instructions which are discussed in section 7.1.1.4. Writing to these MSRs will trigger a GP if one of the following conditions occur:

- If one or more of the reserved bits are modified (Intel, 2025b, p.19-64).
- If a QOS mask outside the supported CLOS is accessed (Intel, 2025b, p.19-65).
- If a value greater than the max CLOS supported is written to IA32_PQR_ASSOC register (Intel, 2025b, p.19-65).

Reading the IA32_PQR_ASSOC register will return the previously programmed CLOS (Intel, 2025b, p.19-65). As stated by Intel (2025b), reading the IA32_L3/2_MASK_n register will also return the previously programmed CBM.

5.2 Memory Bandwidth Allocation

MBA provides the user with the ability to compute and control the memory bandwidth available per logical core (Intel, 2025b, p.19-66). Xeon Scalable Processor family was the first to receive support for MBA (Intel, 2025b, p.19-66). It allows for control of applications that could be over-utilizing memory bandwidth.

5.2.1 Discovery

The discovery phase includes IRDT enumeration which was described in the previous chapter. Post execution of CPUID instruction with EAX set to 7H and ECX set to 0H, ECX.PQE (bit 15) is checked to see if the CPU supports software control over shared resources (Intel, 2025b, p.19-66). CPUID sub-leaf 10H has to be further enumerated to see if MBA is supported (Intel, 2025b, p.19-66).

Bit 3 of EBX register which is returned post enumeration of CPUID sub-leaf 10H will determine if MBA is supported by the processor (Intel, 2025b, p.19-66). If MBA is supported, CPUID sub-leaf 10H can be further enumerated with EAX set to 10H and ECX set to 3H which is the resource ID (Intel, 2025b, p.19-66).

EAX (bits 11:0) returned after CPUID(EAX=10H, ECX=3H), as described earlier, will provide the maximum MBA throttling value minus one (Intel, 2025b, p.19-66).

ECX (bit 2) provides whether the response of delay values is linear (Intel, 2025b, p.19-66).

EDX (bits 15:0) provides the number of CLOS supported for the feature minus one (Intel, 2025b, p.19-66).

5.2.2 Configuration and Usage

Association of threads to CLOS is accomplished the same way as CAT. Configuration of per-CLOS delay values are accomplished via the IA32_L2_QoS_Ext_BW_Thrtl_n registers (Intel, 2025b, p.19-67).

The delay values shall be in the range [0, MAX_MBA] i.e a delay of 0 implies no delay or full bandwidth and MAX_MBA implies maximum delay (Intel, 2025b, p.19-67). In a similar vein as CAT, software is responsible for writing a specific delay value to IA32_L2_QoS_Ext_BW_Thrtl_n, thereby, updating the value applied to a specific CLOS (Intel, 2025b, p.19-67).

5.3 Cache Monitoring Technology

Shared Resource Monitoring is another facility provided by IRDT. This is accomplished with the help of Resource Monitoring ID (RMID) that is specific to a logical processor (Intel, 2025b, p.19-45). RMID can be assigned to a logical processor or to a set of logical processors which is something you would need in the case of monitoring an application across multiple processors (Intel, 2025b, p.19-45). Only one RMID is active for each logical processor and this is enforced using the MSR IA32_PQR_ASSOC that was used in CAT configuration (Intel, 2025b, p.19-45).

Support for monitoring shared resources, provided by the platform, enables tracking cache metrics such as cache utilization and cache misses (Intel, 2025b, p.19-45). The specific event types supported can be enumerated using CPUID instruction (Intel, 2025b, p.19-45).

In short, CMT provides the following mechanisms:

- a mechanism to check if the platform supports CMT (Intel, 2025b, p.19-45).
- CMT-specific event codes to read cache metrics (Intel, 2025b, p.19-45).

5.3.1 Discovery

Software is responsible for checking if the platform/CPU supports CMT by executing CPUID instruction with EAX set to 07H and ECX set to 0H (Intel, 2025b, p.19-46). If the resultant EBX.PQM (bit 1) is set, the processor does provide support for shared resource monitoring like CMT (Intel, 2025b, p.19-46).

CPUID leaf function 0FH provides details about the different resources types available and which of them can be monitored (Intel, 2025b, p.19-46).

5.3.1.1 CMT

The following subsection is based on Intel (2025b, p.19-46). The following procedure is to be performed for the discovery of CMT:

- So the first step is to execute CPUID instruction with EAX set to 0H to discover cpuid_maxleaf. This will help with further enumeration of CMT support.
- If cpuid_maxleaf ≥ 7, then CPUID instruction with EAX set to 7H and ECX set to 0H is executed and the resultant EBX is checked for PQM (bit 12).
- If PQM is set, then CPUID instruction with EAX set to FH and ECX set to 0H is executed to query the shared resources available for monitoring.
- If L3 (bit 1) in resultant ECX is set, then CPUID instruction with EAX set to 0FH and ECX set to 1H is executed to query the L3 CAT/MBM capabilities available.
- Generally executing CPUID instruction with EAX set to 0FH and ECX set to the respective resource ID will enumerate the respective shared resource capabilities.

5.3.2 Configuration

The resultant EDX from executing CPUID instruction with EAX set to FH and ECX set to 0H can be used to determine if L3 CMT is supported (Intel, 2025b, p.19-47). Bit 1 in EDX defines support for L3 monitoring support. Apart from this, the resultant EBX register provides the highest RMID supported by the CPU (Intel, 2025b, p.19-47).

Executing CPUID instruction with EAX set to FH and ECX set to 1 (ResID for L3 CAT) will eventuate in EBX providing the upscaling factor and ECX providing max RMID (Intel, 2025b, p.19-47). The resultant EDX will provide the different event types supported i.e. bit 0 determines support for L3 Occupancy, bit 1 for L3 Total Bandwidth, bit 2 for L3 Local Bandwidth.

Event Type	Event ID	Context
L3 Cache Occupancy	01H	CMT
L3 Total External Bandwidth	02H	MBM
L3 Local External Bandwidth	03H	MBM

Table 5.4: CMT Event Types(Intel, 2025b, p.19-49)

On CPUs which support Memory Bandwidth Monitoring, L3 Local and Total External Bandwidth monitoring events are supported (Intel, 2025b, p.19-48). L3 total external bandwidth monitoring event monitors L3 external bandwidth to the next level of cache i.e. it helps with demand and prefetch misses from L3 to L2. This represents memory bandwidth in most systems (Intel, 2025b, p.19-48). L3 local external bandwidth monitoring event monitors L3 requests in systems with support for non-uniform memory architecture (Intel, 2025b, p.19-48). It should be noted that local and total bandwidth cannot be measured atomically; hence, it is possible for counters to change between reads (Intel, 2025b, p.19-48).

5.3.3 Usage

Firstly the following registers are used for the process of monitoring events:

- IA32_PQR_ASSOC.RMID is used to assign an RMID to a logical processor (Intel, 2025b, p.19-49). As discussed earlier, this basically distinguishes which thread or set of threads uses the specific logical processor.
- IA32_QM_EVTSEL is used to select a specific event type and the respective RMID (Intel, 2025b, p.19-50).
- IA32_QM_CTR is used to obtain the monitored data and also check for error conditions (Intel, 2025b, p.19-50).

Once the discovery or enumeration of capabilities is done, the first step in the process of using monitoring capabilities is to associate a given thread with an RMID (Intel, 2025b, p.19-49).

The following paragraph is based on Intel (2025b, p.19-49). RMID, as the name suggests, is an ID used by hardware to keep track of what the software wants it to monitor. And the process of associating an RMID with a thread is the same for all shared resources. As mentioned earlier, the MSR IA32_PQR_ASSOC is programmed with the RMID that denotes the respective thread or set of threads. The hardware will use this

id to tag internal operations such as L3 cache requests. The width of the RMID field in IA32_PQR_ASSOC can vary from one implementation to another and is derived from *ceil(log_2(1 + CPUID.(EAX=FH, ECX=0H):EBX[31:0]))*. The value of IA32_PQR_ASSOC is always 0 after power-on.

As with CAT, the mechanism used to program CMT is exposed as an MSR pair. Reporting of data is done on a per-RMID basis. This MSR pair is not shared with architectural Perfmon counters (Intel, 2025b, p.19-50).

In order to successfully use CMT, the MSR pair, IA32_QM_EVTSEL and IA32_QM_CTR, is used to initially program CMT and, then, retrieve the recorded data (Intel, 2025b, p.19-50). Firstly, IA32_QM_EVT_SEL is used to select a specific event by writing an event id to it. EvtID (bits 7:0), in IA32_QM_EVTSEL, is used to specify the event (Intel, 2025b, p.19-50). Once this is done, the next step is to configure RMID. RMID (bits 41:32) is used to specify the respective RMID for which monitoring has to be done and the width of RMID in IA32_QM_EVTSEL should match that of IA32_PQR_ASSOC (Intel, 2025b, p.19-50).

The following paragraph is based on Intel (2025b, p.19-50). IA32_QM_CTR is used to report the monitored data. It has 3 bit fields which are used to signal various errors which occur in case wrong event id is used in IA32_QM_EVTSEL and in other unfavorable conditions. ERROR field (bit 63) is set when an invalid event id is used. If UNAVAILABLE field (bit 62) is set, it indicates that data is not available and the value in DATA field (bit 61:0) should be ignored. The OVERFLOW field (bit 61) is present if CPUID with EAX set to FH and ECX set to 1H yields a resultant EAX with bit 8 set. In case this bit is set, it indicates an overflow of MBM counters. The value from DATA field is converted to bytes by multiplying it with the conversion factor which obtained from the resultant EBX of the previous CPUID execution.

Chapter 6

Performance Monitoring Unit

Intel provides performance monitoring with the help of a set of performance-monitoring Model-Specific Register (MSR) (Intel, 2025b, p.21-2). The PMU uses mechanisms for monitoring events which are not available on all microarchitectures (Intel, 2025b, p.21-2). However, newer Intel processors support enhanced architectural performance events which are discussed in this paper.

Architectural performance events act consistently across microarchitectures (Intel, 2025b, p.21-2). Like CAT and CMT, support for PMU is determined by exploring CPUID leaf 0AH (Intel, 2025b, p.21-3). There are different versions of architectural support which are available across processor implementations, for instance, Intel Core Solo and Intel Core Duo processors support base level functionality identified by version ID of 1 (Intel, 2025b, p.21-2). Intel Core and Xeon processors support version ID of 3 (Intel, 2025b, p.21-2). Intel processors based on Skylake through Coffee Lake support version ID 4 and Ice Lake supports version ID 5 (Intel, 2025b, p.21-2).

Although not a subcomponent of IRDT, PMU is explored for the sole reason of evaluating cache hit ratio and, thereby, being able to provide some insight in cases which involve target CPUs with no support for CMT.

6.1 Configuration

PMU is configured using event ids which correspond to architectural and non-architectural events (Intel, 2025b, p.21-3). The number of performance event select MSRs (IA32_PERFEVTSELx) is finite and dependent on the platform; the result of a monitoring event is reported in a performance monitoring counter MSR (IA32_PMCx) (Intel, 2025b, p.21-3). As in the case of CMT, performance monitoring counters are paired with monitoring select registers to monitor resource usage.

Architectural performance monitoring select registers and counters have the following properties (Intel, 2025b, p.21-3):

- The bit field layout of IA32_PERFEVTSELx is consistent across microarchitectures and a non-zero write of an unsupported field introduced in a newer architectural version results in GP (Intel, 2025b, p.21-3).
- Addresses of IA32_PMCx and IA32_PERFEVTSELx remain the same across microarchitectures (Intel, 2025b, p.21-3).
- Each logical processor has its own set of IA32_PERFEVTSELx and IA32_PMCx registers (Intel, 2025b, p.21-3). This simplifies the locking mechanism used in ICAT which will be discussed in detail later.

6.2 Discovery

Architectural performance monitoring can be enumerated using CPUID mechanism (Intel, 2025b, p.21-3). Details such as the number of performance monitoring counters available in a logical processor, number of bits supported in each performance monitoring counter and number of architectural performance events supported can be obtained by this method (Intel, 2025b, p.21-3). CPUID leaf AH is used to enumerate the aforementioned features (Intel, 2025b, p.21-3).

6.2.1 Bit layout of IA32_PERFEVTSEL MSR

The bit layout of IA32_PERFEVTSEL MSR can be summarized as shown below (Intel, 2025b, p.21-4):

- Event Select field (bits 0:7) is the most important field owing to the fact that it is used to select the desired event logic unit (Intel, 2025b, p.21-4). The set of values is defined architecturally and will be discussed in the next section.
- Unit Mask field (bits 8: 15) is used to set conditions that qualify the respective event for detection (Intel, 2025b, p.21-4). Valid UMASK values for each event logic are specific to the unit and described in the next section.
- USR flag (bit 16) when set enables detection of the selected event when the logical processor is operating in privilege levels 1-3 (Intel, 2025b, p.21-4). From the perspective of PikeOS, this means that the selected event happening in userland is taken into account. It can be used with OS flag (Intel, 2025b, p.21-4).
- OS flag (bit 17) when set enables detection of the selected event when the logical processor is operating in privilege levels 0 (Intel, 2025b, p.21-4). It can be used with USR flag (Intel, 2025b, p.21-4).
- E flag (edge detect) (bit 18) when set enables edge detection of the selected microarchitectural condition (Intel, 2025b, p.21-4). It basically counts the number of deasserted to asserted transitions (Intel, 2025b, p.21-4).
- PC flag (pin control) (bit 19) On Sandy Bridge microarchitecture, this bit is reserved. On previous processors, the bit being set caused the processor to toggle PMi pins and increment the counter when performance-monitoring events occurred and when clear, toggled the PMi pins in case of counter overflow (Intel, 2025b, p.21-4).
- INT (APIC interrupt enable) (bit 20) when set causes the logical processor to generate an exception throught its local APIC on counter overflow (Intel, 2025b, p.21-5).
- EN (Enable Counters) (bit 22) when set enables performance counting. The event logic unit for UMASK must be disabled before writing to IA32_PMCx (Intel, 2025b, p.21-5).
- INV (Invert) flag (bit 23) when set inverts the counter-mask comparison (Intel, 2025b, p.21-5).
- CMASK (Counter Mask) field (bits 24:31) - when this field is set, the CPU compares this mask to the events counted and increments the counter if the count is greater than/equal to this mask and ignores it otherwise (Intel, 2025b, p.21-5).

6.2.2 Pre-defined Architectural Performance Events

The following table lists the pre-defined architectural events which are used in ICAT (Intel, 2025b, p.21-19).

Event Name	UMask	Event Select
Unhalted Core Cycles	0x00H	0x3CH
Instruction Retired	0x00H	0xC0H
Unhalted Reference Cycles	0x01H	0x3CH
LLC Reference	0x4FH	0x2EH
LLC Misses	0x41H	0x2EH

Table 6.1: UMask and Event Select Encodings(Intel, 2025b, p.21-18)

- Unhalted Core Cycles - This event represents the cycles executed when a specific core is running (Intel, 2025b, p.21-18). The counter is not incremented in the following cases: ACPI C-state other than C0, HLT instruction gets executed, STPCLK pin is asserted, Throttling due to TM1 and during frequency switching phase of a performance state transition (Intel, 2025b, p.21-18).
- Instructions Retired - This event represents the number of instructions at retirement i.e. those that got executed already (Intel, 2025b, p.21-19).
- Unhalted Reference Cycles - This event represents the reference cycles counted at a fixed frequency while the clock signal on the core is running (Intel, 2025b, p.21-19). This event is not affected by core frequency changes (Intel, 2025b, p.21-19).
- Last Level Cache References - This event represents requests originating from the core that reference last level cache line (Intel, 2025b, p.21-19). The event includes speculation and cache line fills due to first level cache prefetcher (Intel, 2025b, p.21-19).
- Last Level Cache Misses - This event represents last level cache misses (Intel, 2025b, p.21-19). The event may count speculation and cache line fills due to the first level cache prefetcher (Intel, 2025b, p.21-19).

6.3 Usage

Post enumeration of PMU, armed with details about counters, their width and the architectural version, the next step is to configure IA32_PERFEVTSELx MSRs. Once PERFEVTSELx registers are programmed, the next step is to read IA32_PMCx registers to obtain the values reported by the corresponding counters (Intel, 2025b, p.21-3).

Armed with the knowledge of the programming environment, cache implementations and architectural components which help with allocation and monitoring of shared resources such as PMU, it is time to go ahead with the implementation details of ICAT.

Chapter 7

Implementation of ICAT

This section discusses the implementation of the driver ICAT and the details pertaining to its design. It talks about the IOCTL interface which is exposed to userspace and the different ways of using ICAT for the purpose of driving IRDT. Additionally, it talks about the different modules which comprise ICAT and what each module drives. Apart from usage and configuration, it illustrates how shared resources can be monitored and takes a shot at cache statistics.

7.1 ICAT as a KDEV

ICAT is a kernel driver which handles the discovery, configuration and usage of IRDT on PikeOS. It mainly focuses on CAT, and CMT due to the relevance of cache allocation and the importance of determinism in mission-critical embedded systems. The rationale for implementing such a driver is discussed in detail later on.

The availability of callbacks provided by PikeOS KDEV framework is the rationale for writing ICAT as a kernel driver. As Liedtke (1995) argues, the only reason why a component should be moved outside the kernel is if it wouldn't prevent the component from functioning properly otherwise. In this case, callbacks such as **taskswitch** cannot be implemented in userspace since the driver framework doesn't support this.

7.1.1 Modules

The driver consists of the following modules, each of which will be discussed in detail:

- **irdt**
- **cat**
- **cmt**
- **log**
- **msr**
- **pmu**
- **main**

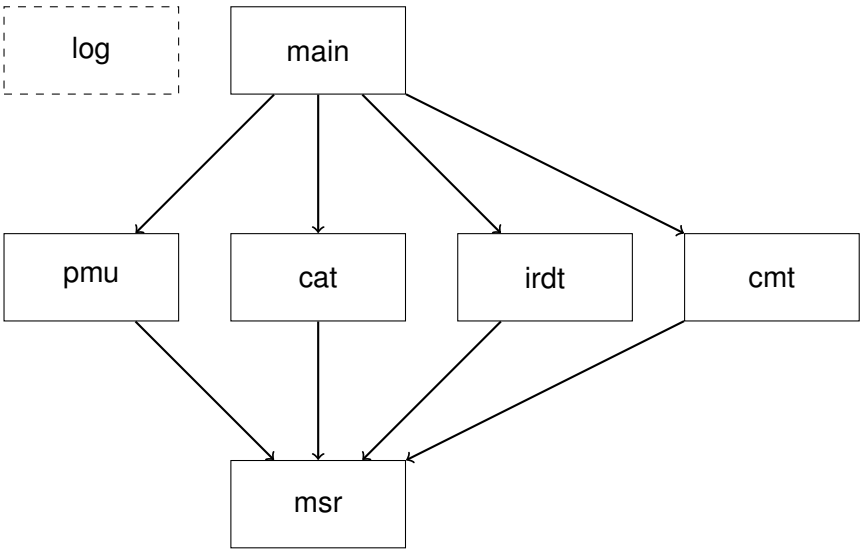

Figure 7.1: Structure of ICAT

The figure shown above depicts the dependency between the different modules of ICAT. The code in ICAT is segregated based on the subcomponent that handles the respective module, and the API it exposes. It also makes for a clean design.

7.1.1.1 Module: irdt

The file *main.c* ties all the modules together, hence, discussion about it will be deferred until the very end. The first piece of code that was implemented was the logic which enumerates the different subcomponents of IRDT, namely CAT, CMT, and MBA. This is handled in the file *irdt.c*. The most important function in *irdt.c* is **icatenumirdt**(). The function **icatenumirdt** utilizes the CPUID instruction to explore the various leafs and sub-leafs pertaining to IRDT.

For instance, the following assembly code is written in C using inline assembly to do a primary check on IRDT support.

```
eax = 0x7;
ecx = 0x0;

__asm__ __volatile__ (
  "mov %4, %%eax\n"
  "mov %5, %%ecx\n"
  "cpuid\n"
  "mov %%eax, %0\n"
  "mov %%ebx, %1\n"
  "mov %%ecx, %2\n"
  "mov %%edx, %3\n"
:   "=r" (eax), "=r" (ebx), "=r" (ecx), "=r" (edx)
:   "r" (eax), "r" (ecx)
:   "%eax", "%ebx", "%ecx", "%edx", "memory");
```

```
if ((ebx & (1u<<15))) {
  LOG("Intel RDT: Software control over"
      "shared res supported\n");
}

if ((ebx & (1u<<12))) {
  LOG("Intel RDT: CMT supported\n");
  cmt = TRUE;
  ic->cmt = TRUE;
}
```

In this case, the CPUID leaf 7H sub-leaf 0H is used to check if IRDT is supported. Once the values 7H and 0H are written to EAX, and ECX registers, the subsequent CPUID call will return values in EAX, EBX, ECX, and EDX. EBX is checked for bit 15 which when set implies that software control over shared resources is supported by the processor (Intel, 2025b, p.3-229). The aforementioned CPUID mechanism is a common programming pattern used through out ICAT for enumeration of various subcomponents. This programming pattern will be, hereinafter, referred to as CPUID mechanism.

In a similar vein, EBX is checked for bit 12 which when set implies support for CMT. Once this is done, the next CPUID leaf FH, and sub-leaf 0H is enumerated to figure out the width of rmid.

```
rmidwidth = ceillog2(ebx+1);
```

The width of rmid is computed by taking the ceil value of $\log_2(ebx+1)$. The implementation of **ceillog2**() can only support values in the range of [0, 255], which in our case is more than enough since rmid width rarely exceeds a value of 5, as observed in Intel Broadwell microarchitecture.

Once, support for IRDT has been confirmed, the CPUID mechanism is used to enumerate leaf FH, sub-leaf 1H. EDX is subsequently checked for bits 0, 1 and 2 that when set implies support for L3 CMT, L3 total bandwidth monitoring, and L3 local bandwidth monitoring respectively.

If support for CMT is reported, then ECX will contain the maximum value of rmid supported, and EBX will contain the upscale factor which is used in the computation of cache usage . The code below reflects this:

```
if (cmt) {
  maxrmid = ecx;
  upscalefactor = ebx;
}
```

Subsequently, CPUID mechanism is used to enumerate leaf 10H, sub-leaf 0H and EBX is checked for bits 1, 2 and 3 that when set implies support for CAT on L3, L2, and MBA respectively. This concludes the initial assessment of IRDT, and by using the information obtained about CAT, CMT, and MBA, the enumeration of L3 CAT, L2 CAT, and MBA is performed.

The functions **enuml3**(), **enuml2**() and **enummba**() handle the enumeration of respective last level cache, and memory bandwidth allocation.

The function **enuml3** uses the CPUID mechanism to enumerate leaf 10H, sub-leaf 1H, and the resultant EAX is used to compute the max length of CBM. The computation is done as shown below:

```
  cbmlen = (eax & 0xF) + 1;
  ic->cbmlen = cbmlen;
  LOG("Intel RDT: L3 CBM Length: %u\n", cbmlen);
```

As shown above, the LSB of EAX is carved out, and incremented by one to obtain the length of CBM. The next parameter that is essential for the configuration of CAT is the maximum number of CLOSes supported. This is computed using EDX as shown below:

```
  nclos = (edx & 0xFF) + 1;
  ic->nclos = nclos;
```

Once, the length of CBM and the number of CLOSes have been obtained, the number of allocatable bits i.e. the number of bits in CBM that can be assigned a new value, is obtained using the following code:

```
  allocbit = numofzero(ebx, cbmlen);
  ic->allocbit = allocbit;
```

As shown above, the function **numofzero**() computes the number of contiguous zeros in EBX which in effect represents the number of allocatable bits.

The function **numofzero**(), as shown below, loops through the bits in *n* starting from bit 0 to the limiting bit provided in *max*.

```
static P4_uint32_t
numofzero(P4_uint32_t n, P4_uint32_t max)
{
  P4_uint32_t r, t;

  r = t = 0;

  while (!(n & 1u<<t)) {
    ++r;
    ++t;
    if (t == max)
      break;
  }
  return r;
}
```

As the name might suggest, the function **enuml2**() is analogous to **enuml3**() as it handles L2 cache instead of L3.

The function **enummba**() uses the CPUID mechanism to enumerate leaf 10H, sub-leaf 3H and EAX is used to compute the maximum throttling value. This sums up the enumeration phase of IRDT. Armed with the information obtained during this phase, ICAT tries to provide a sane interface to the user for the configuration of CAT, CMT, PMU etc.

7.1.1.2 Module: cat

The file *cat.c* contains 5 functions: **icatvalidconf**(), **icatsetupclos**(), **icatjoinclos**(), **icatgetclos**(), **icatsetclos**(). This module handles the configuration and setup of CAT in ICAT. Inorder to describe the functions with clarity, a slight detour through config.h is necessary.

The file *config.h* consists of two arrays: **closcbm** and **closcpu** and a definition ICAT_NCPU. These arrays represent the relationship between CLOSes, and their respective CBM, in the case of **closcbm**, and the relationship between CLOSes, and their respective CPUs, in the case of **closcpu**. This file is used for the static configuration of CAT in ICAT, and the user is expected to configure the arrays, and ICAT_NCPU which represents the maximum number of CPUs in the system. The array **closcbm** uses its index semantically to represent the CBM that belongs to the respective CLOS i.e. closcbm[0] represents the CBM that belongs to CLOS0 and this goes on for 1, 2, etc. Similary, the array **closcpu** uses its index to represent the respective CPU to which the CLOS should be assigned i.e closcpu[0] represents the index of closcbm which indirectly references the CBM that will be assigned to CPU0. So, in effect, these two arrays illustrate the static relationship between CLOSes, CBMs and CPUs.

The function **icatvalidconf**() validates the configuration file *config.h*. This is necessary since invalid configuration should yield a panic during the initialization phase of the driver. This is accomplished with the help of the PikeOS structure P4_kinfopage_t. This data structure, which gets filled-in by PikeOS, can be used to figure out the number of CPUs supported by PikeOS. This, in turn, is used by the function **icatvalidconf**() to check if ICAT_NCPU provided by the user is valid. If it is not, the initialization of ICAT will fail. The number of CPUs is used to validate the setup of closcpu as shown below:

```
if (ICAT_NCPU != kinfo->num_cpu) {
  LOG("ICAT_NCPU is found to be invalid\n");
  LOG("Number of CPU supported: %u"
    "Kindly fix config.h\n",
    kinfo->num_cpu);
  return FALSE;
}

if (ICAT_NCPU != NELEM(closcpu)) {
  LOG("closcpu[] is found to be invalid\n");
  LOG("Kindly provide CLOSes for exactly %u CPUs\n",
    ICAT_NCPU);
  return FALSE;
};
```

Once the validation of number of CPUs, and maximum number of CLOSes is done, the array **closcpu** is validated against the number of CPUs since each CPU should have a valid CLOS assigned to it.

The function **icatsetupclos**() sets up the CLOSes supported by the processor using CBM provided by the user. Firstly, the code uses the information provided by the module **irdt** to determine if the last level cache is L3 or L2. Once this is determined, the respective MSR is used i.e. IA32_L3_MSR_0 in the case of L3 and IA32_L2_MSR_0 in the case of L2.

Then it proceeds with the configuration of CLOSes using the value in allocatable bits which was discovered during the enumeration phase and the CBM provided by the user. It basically carves out the valid portion of CBM using allocatable bits and uses the function **icatwrmsr** to write this value to the respective MSR. As mentioned in earlier chapters, each CLOS has a specific MSR register so a loop is used to assign the CBM to the respective CLOS. The code below reflects this logic:

```
for (P4_uint32_t i = 1; i < ic->nclos; ++i) {
  ecx = msr + i;

  icatrdmsr(ecx, &eax, &edx);

  // calculate CBM
  // Carve out relevant portion of closcbm[i]
  msk = 0xFFFFFFFF;
  cbm = (closcbm[i] & ~(msk << ic->allocbit));

  // Assign new cbm
  eax = cbm;
  edx = 0;

  // writemsr
  icatwrmsr(ecx, eax, edx);

  // Check again
  icatrdmsr(ecx, &eax, &edx);
}
```

The function **icatjoinclos**() helps with associating the CPU with the respective CLOS. In this function, IA32_PQR_ASSOC MSR is used to associate a CPU with the respective CLOS. This MSR exists for each CPU; the function **icatwrmsr**() is used to associate the CLOS provided by the array *closcpu*[] with the respective CPU by writing to the MSR. The code shown below reflects the same:

```
  eax = 0;
  ecx = IA32_PQR_ASSOC;
  edx = closcpu[c];

  icatwrmsr(ecx, eax, edx);
```

The functions **icatgetclos**() and **icatsetclos**() do what they say; they get and set the respective CLOSes in the array *closcpu*[].

7.1.1.3 Module: log

The file *log.c* contains the function **icatlog**() which prints logging information on the console. It uses a spinlock, thereby, making it safe(reentrant) to use even in a multi-threaded environment. It uses the function **psp_vprintf** provided by PikeOS PSP to write to console.

7.1.1.4 Module: msr

Intel provides two instructions RDMSR and WRMSR to read and write from/to MSRs (Intel, 2025b, p.4-542,6-9). The file *msr.c* contains the functions: **icatwrmsr** and **icatrdmsr** which essentially wraps RDMSR and WRMSR instructions. These functions take 3 arguments: ecx, eax, and edx which represent x86 registers of the same name.

RDMSR instruction is used to read the contents of the MSR which is specified in ECX register and the value is returned using EDX and EAX registers. The least significant 32 bits are returned in EAX register and the most significant 32 bits in EDX register. Similarly, WRMSR instruction is used to write a value contained in EAX and EDX registers to the MSR specified in ECX register (Intel, 2025b, p.4-542,6-9).

7.1.1.5 Module: pmu

The file *pmu.c* contains the functions: **icatenumpmu**, **icatinitpmu**, **icatreadpmu** and **setevnt**. The function **icatenupmu** uses the CPUID mechanism, as discussed earlier, to enumerate leaf AH, sub-leaf 0H and the resultant the LSB of EAX is used to figure out the architectural version ID. The second byte of EAX is used to figure out the number of counters supported by PMU in the respective platform. The third byte of EAX is used to figure out the width of the PMC counter and the fourth byte to figure out the number of architectural events. The code shown below reflects the same:

```
vid = eax & 0xFF;
LOG("PMU: Architectural Version ID: %u\n", vid);
pmuconf.vid = vid;

ncounter = (eax & 0xFF00) >> 8;
LOG("PMU: Number of MSRs available: %u\n", ncounter);
pmuconf.nc = ncounter;

counterwid = (eax & 0xFF0000) >> 16;
LOG("PMU: Width of counter: %u\n", counterwid);
pmuconf.cw = counterwid;

nevent = (eax & 0xFF000000) >> 24;
LOG("PMU: Number of architectural events supported: %u\n",
  nevent);
pmuconf.nev = nevent;
```

ICAT supports upto 4 pre-defined architectural events. However, all pre-defined architectural events are defined in the static data structure events, which is an array of struct Icatperfevent whose members include evsel, umask, and name. This structure is defined as shown below:

```
struct Icatperfevent {
  /* Event selector */
  P4_uint64_t evsel;
  P4_uint64_t umask;
  const char *name;
};

/* Pre-defined Architectural Performance events */
static Icatperfevent events[] = {
  {0x3C, 0x0, "unhalted core cycles"},
  {0xC0, 0x0, "inst retired"},
  {0x3C, 0x1, "unhalted ref cycles"},
  {0x2E, 0x4F, "llc ref"},
  {0x2E, 0x41, "llc misses"},
  {0xC4, 0x0, "br inst retired"},
  {0xC5, 0x0, "br miss retired"},
  {0xA4, 0x1, "topdown slots"},
  {0xA4, 0x2, "topdown backend bound"},
  {0x73, 0x0, "topdown bad speculation"},
  {0x9C, 0x1, "topdown frontend bound"},
  {0xC2, 0x2, "topdown retiring"},
  {0xE4, 0x1, "lbr inserts"}
};
```

The function **setevnt** is used to setup the layout of IA32_PERFEVTSELx MSR. This function takes an argument which selects an event from *events* and use it to craft the layout of IA32_PERFEVTSELx MSR. It handles USR flag, OS flag and EN flag which decides whether events that happen userspace or kernelspace get recorded. By default, the kernelspace events don't cause the counters to increment. The code shown below reflects the same:

```
static P4_uint32_t
setevnt(P4_uint32_t i)
{
  P4_uint32_t eax;

  eax = events[i].evsel;
  eax |= events[i].umask << 8;
  eax |= IA32PERFES_USR;
  eax |= IA32PERFES_EN;

  return eax;
}
```

The function **icatinitpmu**() is used to initialize PMU using the information that was obtained using the function **icatenumpmu**. This function sets up PMU by first calling **setevnt**() function to craft a value suitable for IA32_PERFEVTSELx MSR and then it used **icatwrmsr**() to write the value to the respective MSR. This has to be done for each event supported by ICAT and, hence, a loop is used to the same. The code shown below reflects this logic:

```
  /* Handle LLC refs and LLC misses */
  for (i = 0; i < 2; i++) {
    ecx = IA32PERFEVTSEL + i;
    icatrdmsr(ecx, &eax, &edx);

    eax = setevnt(3+i);
    edx = 0;

    icatwrmsr(ecx, eax, edx);
    icatrdmsr(ecx, &eax, &edx);
  }
```

The aforementioned code is repeated to support events like *Unhalted core cycles* and *Instructions retired*.

The function **icatreadpmc**() handles returning the value reported by IA32_PMCx counters. This function depends on **icatinitpmu**() since the initialization of IA32_PERFEVTSELx MSRs influence the IA32_PMCx counters which are read by **icatreadpmc**. As with IA32_PERFEVTSELx, there is a counter for each event which should be read and this warrants a loop. The following code reflects this logic:

```
  for (i = 0; i < 2; i++) {
    ecx = IA32_PMC + i;
    icatrdmsr(ecx, &eax, &edx);
    if (i == 1) {
      *llmiss = edx;
      *llmiss <<= 32;
      *llmiss |= eax;
    }
    if (i == 0) {
      *llref = edx;
      *llref <<= 32;
      *llref |= eax;
    }
  }
```

The counter width decides if there is a need to use two registers EAX and EDX to read the reported value. The least significant 4 bytes are returned in EAX and the most significant 4 bytes in EDX. Hence, these two values are merged to find the actual reported value in the respective counter. This is repeated for every event that was configured.

7.1.1.6 Module: cmt

The portion of code which enumerates CMT and its associated data structure is described in section 7.1.1.7. Its configuration and usage is detailed in section 7.1.2.2.

7.1.1.7 Module: main

The file *main.c* ties all the modules described earlier to provide a programming interface to the user for IRDT on PikeOS. Since ICAT is implemented as a kernel driver, it has two ends with their own respective callbacks which have to be setup for functionality. As discussed in Chapter 2, the provider callbacks are used by the PikeOS kernel to setup the kernel driver and gate callbacks are used by the userspace to ask for services.

The provider callbacks used in ICAT include **initprov**(), **initcpu**(), **initgate**(), and **taskswitch**(). Each of these callbacks serves a purpose during the configuration phase. All of these callbacks are assigned using an object of type *drv_prov_ops_t* and declared to the kernel, using DRV_DECLARE_DRV.

```
static drv_prov_ops_t pops = {
        .init_prov = initprov,
        .init_gate = initgate,
        .init_cpu = initcpu,
        .alert_task_switch = taskswitch,
};
DRV_DECLARE_DRV(icat, &pops);
```

The callback function **initprov** is used for the initialization of various data structures in ICAT. Data structures defined in ICAT include the following:

```
struct Icatconf {
        /* Number of CLOS */
        P4_uint32_t nclos;
        /* CBM length */
        P4_uint32_t cbmlen;
        /* Allocatable bits in CBM */
        P4_uint32_t allocbit;
        /* LLC type */
        P4_uint32_t llctype;
        /* L3/L2 support */
        P4_bool_t llc;
        /* L3/L2 CDP support */
        P4_bool_t llccdp;
        /* CMT Support */
        P4_bool_t cmt;
};
```

The aforementioned structure is used to store the configuration of CAT.

```
struct Icatcmtconf {
        /* Max rmid */
        P4_uint32_t maxrmid;
        /* Rmid width */
        P4_uint32_t rmidwid;
        /* Upscale factor */
        P4_uint32_t upscalef;
};
```

The aforementioned structure is used to store the configuration of CMT.

```
struct Gatepriv {
        P4_glock_t gl;
        /* array for llc usage stats */
        P4_uint64_t llcusage[16];
        /* points to the next free element of llcusage */
        P4_uint64_t *llcu;
        /* enough samples to use for avg/sd */
        P4_bool_t llces;
        /* Resource Monitoring ID */
        P4_uint32_t rmid;
        /* CPUID */
        P4_uint32_t cpuid;
        P4_uint32_t padding;
};
```

The aforementioned structure is used to store the state of certain fields which belong to the respective gate.

The function **initprov** initializes the spin lock used for the purpose of logging called *llock* and one for accessing a shared data structure called *tlock*. The design of ICAT forbids a gate descriptor from being used by another CPU other than the one that opened it. Apart from that, it allocates some memory for an object of type *lcatconf* which holds the state of CAT internally. This is done using **drv_calloc**() function. Since **drv_calloc** will panic if enough kernel memory is not available, the return value is not checked. The code below reflects the same:

```
  conf = drv_calloc(1, sizeof(*conf), DRV_PART_GLOBAL);
```

Once this is done, **icatenumirdt**() is called to enumerate IRDT and the details obtained is stored in *conf*. Subsequently, a check is done to ensure that some kind of last level cache, be it L3 or L2, is supported before validating the configuration provided using the file *config.h*. As discussed earlier, an invalid configuration is considered fatal and a valid configuration will eventuate in the code calling **icatsetupclos** to setup the respective CLOSes. This is done strictly owing to the hardware raising GP fault in case of invalid configuration from which recovery is not possible. The code shown below reflects this logic:

```
        if (conf->llc) {
                if (!icatvalidconf(conf)) {
                        return P4_E_CONFIG;
                }
                icatsetupclos(conf);
        }
```

Support for CMT is then checked and, if found, the function **icatcmtinfo** is used to retrieve it and store the state of CMT in an object of type *lcatcmtconf*. Another data structure of

importance is the array *tasks* which maps the task id with RMID. The size of *tasks* is determined by the maximum rmid that is supported by the CPU. This data structure is allocated using **drv_calloc** and is assigned an invalid value of 256. Valid PikeOS task IDs that range from 0 to 255 is the rationale for choosing the former value. The code shown below reflects this logic:

```
#define INVALTASK 256

  if (conf->cmt) {
          icatcmtinfo(&cmtconf);
          tasks = drv_calloc(cmtconf.maxrmid,
                                  sizeof(unsigned int),
                                  DRV_PART_GLOBAL);
          for (unsigned int i = 0; i < cmtconf.maxrmid; i++)
                  tasks[i] = INVALTASK;
  }
```

Lastly, the function **icatenumpmu** is called to enumerate PMU. Once this is done, the function returns an error code of P4_E_OK, which represents success. Since **initprov** is serialized by the KDEV framework, no other locking is necessary.

The function **initgate**() initializes the data structure *Gatepriv* which is stored in the private area of the per-gate data structure *drv_gate_t*. Initialization of the former includes allocating memory using **drv_calloc**, initializing the lock using **p4_glock_init** and initializing its members. An object of type *Gatepriv* has the following members:

- *gl* which is a lock used for the purpose of concurrency.
- *llcusage* which is an array that is used to store cache statistics.
- *llcu* which is a pointer to the next free element of llcusage.
- *llces* which is a flag that ensures that a minimum number of samples of cache usage is available for the purpose of computation of average and standard deviation.
- *rmid* which is the rmid assigned for the respective task.
- *cpuid* which is the cpuid of the task.

ICAT is designed in such a way that the cache statistics is associated with a specific task which in effect is associated with a RMID. Once the gate is opened, the gate effectively is married to the specific CPUID. It is illegal to use a gate descriptor from another CPU. This decision was made by fiat owing to the adverse repercussions such as the cache statistics becoming stale in its entirety for a different CPU.

Once the initialization of data structure is done, the function **getgatepriv**() is used to assign the object to the private area of the gate. The code shown below reflects this logic:

```
  gp = drv_calloc(1, sizeof(Gatepriv), DRV_PART_GLOBAL);
  p4_glock_init(&gp->gl);
  gp->llcu = gp->llcusage;
  gp->llces = FALSE;
  *getgatepriv(g) = gp;
```

The callback function **initcpu**() is used to setup the logical processors using the CLOSes that are statically configured in the file *config.h*. This callback is called by the KDEV framework on every CPU. In fact, this is the last callback in the init phase that is serialized by default by the KDEV framework. In this function, a check is made to ensure that CAT is supported and this is done by checking *llc* which is set during the enumeration phase. Once it is validated to be set, the function **icatjoinclos** is called on the CPU to join the respective CLOS. The code shown below depicts the same:

```
P4_cpuid_t cpuid;

cpuid = p4_my_cpuid();
if (conf->llc)
        icatjoinclos(cpuid);

return P4_E_OK;
```

The callback function **taskswitch**() is used to improve the precision with which cache measurement is done. This is primarily done because a logical processor can execute more than one task. When a task is preempted and the scheduler starts running another task, the RMID, stored in IA32_PQR_ASSOC MSR, needs to be updated using that of the new task. Failing to do so will result in incorrect cache measurements reported by CMT. This update mechanism is abstracted away in the function **assocrmid**. The latter will be discussed in detail later on. In **taskswitch**(), a check is made to ensure that CMT is supported. If it is not, the function returns. A lock is used subsequently to protect accesses to the data structure *tasks* which can be updated concurrently. The function **taskid2rmid**() is called to obtain the RMID of the respective task and if a valid RMID is found, the CPU is updated with it using **assocrmid**. The code shown below reflects this logic:

```
int rmid;

if (!conf->cmt)
        return;
p4_spin_lock(&tlock);
rmid = taskid2rmid(t);
p4_spin_unlock(&tlock);
if (rmid == -1)
        return;
assocrmid((P4_uint32_t)rmid);
```

Thus ends the init provider callback section. The next section deals with 3 important callbacks required for almost every driver: **open**(), **close**(), and **ioctl**().

The function **open**() is called when the user uses **vm_open**() to open the device ICAT. This function basically sets up the *cpuid* member of the gate's private data structure. As discussed earlier, this is done so that the gate descriptor is not used by any other CPU. The code below reflects this logic:

```
  Gatepriv *gp;

  gp = *getgatepriv(gd->gate);
  gp->cpuid = p4_my_cpuid();
```

The function **close**() is a placeholder since there isn't much to be done.

7.1.2 IOCTL interface

Last but not least, the function **ioctl**() handles the lion's share of ICAT's API. This function provides 4 commands: ICAT_ASSOC, ICAT_MON, ICAT_STAT and ICAT_JOIN; some of these commands require a certain order of execution to behave as intended and provide useful information.

7.1.2.1 ICAT_ASSOC

This command is most likely the first to be used since the IOCTL interface is only used when the user would like to configure ICAT during runtime or for the explicit purpose of instrumentation and analysis. For the latter, the first step involves associating the RMID with the specific task and this is handled by ICAT_ASSOC. ICAT_ASSOC expects the user to provide the TASK ID which is passed through by the KDEV framework in the argument *indata*. This is copied from userspace to kernelspace using the function **drv_memcpy_in** and provided to the respective switch case. The code below reflects the same:

```
if (indata != NULL
&&  (e = drv_memcpy_in(&ics, indata, sizeof(ics))) != P4_E_OK) {
        LOG("drv_memcpy_in failed: %s\n", p4_strerror(e));
        return P4_E_PAGEFAULT;
}
```

The data structure used by the user for purpose of making IOCTL calls to ICAT is shown below:

```
struct icatstat {
        /* Cache usage in bytes */
        P4_uint64_t cu;
        /* Average/Mean cache usage */
        P4_uint64_t cm;
        /* Standard deviation */
        P4_sint64_t sd;
        /* perfevents
         * [0] = unhalted core cycles
         * [1] = inst retired
         * [3] = llc refs
         * [4] = llc misses
         */
```

```
        P4_uint64_t events[13];
        P4_uint32_t rmid;
        P4_uint32_t clos;
        P4_task_t tid;
        P4_uint32_t padding;
};
```

As shown above, it consists of members that are used by ICAT to provide cache statistics such cache usage, average cache usage, standard deviation, and properties associated with the configuration such as RMID, TASK ID and CLOS to which the respective CPU belongs. However, not all of these members are used in the case of most IOCTL commands.

ICAT_ASSOC basically checks if CMT is available, if it is not, the entire setup of RMID is skipped and PMU is configured. However, if this is not the case, a glock is used to enter a critical region where the next available RMID is checked against the maximum RMID supported. This is a static variable and is incremented post use. If the check is found to be positive then a return value of P4_E_LIMIT is returned only after glock is given back. The code shown below reflects this logic:

```
if (!conf->cmt) {
        LOGVV("CMT not supported\n");
        goto trypmu1;
}

p4_glock_enter(&gp->gl);
if (rmid > cmtconf.maxrmid) {
        LOG("ASSOC failed: rmids have been exhausted\n");
        p4_glock_leave(&gp->gl);
        return P4_E_LIMIT;
}
```

After checking the next available RMID, a spin lock is used to set the respective TASK ID in the array *tasks*. The location at which the TASK ID is stored depends on the value of RMID. This abstraction makes access to TASK ID an O(1) operation. The spin lock is used here since the shared array *tasks* is used in the callback function **taskswitch** where the former is used to retrieve the RMID and associate it with the CPU. There can be upto N CPUs, in a platform that supports a maximum of N CPUs, trying to access this array at anytime. Once this is done, the lock is given away and the function **assocrmid** is used to write the respective RMID to IA32_PQR_ASSOC MSR of the respective CPU. The code shown below reflects the same:

```
          p4_spin_lock(&tlock);
          tasks[rmid++] = ics.tid;
          p4_spin_unlock(&tlock);
          gp->rmid = rmid - 1;
          assocrmid(gp->rmid);
```

This concludes the setup of CMT. Once this is done, PMU is initialized using the function **icatinitpmu**() as discussed earlier. If PMU is found to be not supported along with CMT, a return code of P4_E_NOTIMPL is returned to the user. The code shown below reflects the same:

```
trypmu1:
          if (icatinitpmu()) {
                  LOGVV("PMU not supported\n");
                  if (!conf->cmt) {
                          p4_glock_leave(&gp->gl);
                          return P4_E_NOTIMPL;
                  }
          }
```

7.1.2.2 ICAT_MON

The command ICAT_MON is used to monitor the cache usage using CMT and PMU. As with ICAT_ASSOC, the first thing that is done is check if CMT is supported. If it is not, control is handed off to a section which uses PMU for cache measurement. Otherwise, the cache usage provided reported by CMT is obtained using the IA32_QM_EVTSEL MSR. The event EVENT_L3_CMT is used along with the respective RMID to obtain the cache usage in bytes. The reported usage is also stored in the array *llcu* for further analysis. Once this is done, the function **icatreadpmc** is used to read the last level cache misses and references and all these data points are returned to the user using the structure *icatstat*. The code shown below reflects the same:

```
          if (!conf->cmt) {
                  goto trypmu2;
          }

          /* Check L3 cache usage using CMT */
          ecx = IA32_QM_EVTSEL;
          icatrdmsr(ecx, &eax, &edx);
          eax = P4_BIC(eax, (1 << 8)-1) | EVENT_L3_CMT;
          edx = P4_BIC(edx, (1 << 10)-1) | gp->rmid;
          icatwrmsr(ecx, eax, edx);
          icatrdmsr(ecx, &eax, &edx);

          ecx = IA32_QM_CTR;
          icatrdmsr(ecx, &eax, &edx);
          if (!(edx & ((P4_uint32_t)1 << 31))
          && !(edx & ((P4_uint32_t)1 << 30))) {
                  cu = cacheusage(eax, edx);
                          gp->cpuid, cu/KB);
          }
          *gp->llcu++ = cu;
```

```
        /* Works like a ring buffer */
        if (gp->llcu == gp->llcusage + NELEM(gp->llcusage)) {
                gp->llcu = gp->llcusage;
                gp->llces = TRUE;
        }

trypmu2:
        if (icatreadpmc(&llcref, &llcmiss)) {
                if (!conf->cmt)
                        return P4_E_NOTIMPL;
        }
```

As with ICAT_ASSOC, if CMT and PMU are not supported, a return code of P4_E_-NOTIMPL is returned. PMU being supported architecturally on most CPUs and, hence, being more likely to be present is the rationale behind this logic. This command shall only be used after ICAT_ASSOC, since in order for CMT to report a measurement, the CPU has to be associated with the respective RMID. However, if CMT is not supported or not of any significance to the user, one can use ICAT_ASSOC to read PMC counters for misses and references.

7.1.2.3 ICAT_STAT

This command is used to provide some statistical analysis of the measurements obtained. It uses two functions to accomplish this: *mean*() to compute average mean and *sd* to compute standard deviation. Both of these attributes are computed using the cache usage entries reported by CMT. So, if CMT is not supported, this ioctl command will return P4_E_NOTIMPL. It shall also provide other details like the current RMID, CLOS etc to the user. The code shown below reflects the same:

```
        if (!conf->cmt) {
                return P4_E_NOTIMPL;
        }

        a = 0;
        sd = 0;

        if (!gp->llces) {
                return P4_E_STATE;
        }
        a = mean(gp->llcusage, NELEM(gp->llcusage));
        sd = stddev(gp->llcusage, NELEM(gp->llcusage), a/KB);

        ics.rmid = gp->rmid;
        ics.cm = a;
        ics.sd = sd;
        ics.clos = icatgetclos(gp->cpuid);
```

As shown above, the entries obtained from *llcu* are used for obtaining the averge mean and standard deviation. The flag *llces* is used to check if there are enough samples to perform this computation. If there is not, an error message is printed on console, and an error code of P4_E_STATE is returned to the user. This is owing to the fact that the algorithm implemented relies on the existence at least N samples. The code below shows the algorithm which computes standard deviation:

```
        s = 0;
        for (unsigned int i = 0; i < n; i++) {
                d = a[i]/KB - mean;
                s += d * d;
        }
        s = s/n;
        LOGVV("Variance(KB): %u\n", s);
        return isqrt(s);
```

As with ICAT_MON, this command shall only be used after setting up the RMID using ICAT_ASSOC. It cannot be used in any other context, since it depends predominantly on the existence of CMT, and the cache usage entries reported by the same. Additionally, the user is expected to collect at least N usage entries, usually 32, by using ICAT_MON before invoking ICAT_STAT for the purpose of obtaining mean, standard deviation etc. The array *llcusage* along with the pointer *llcu* basically acts like a ring buffer, since it wraps around once the last element is encountered. This is done due to the fact the algorithm only cares about newer data and stale data can be overwritten, thereby, re-using the storage.

7.1.2.4 ICAT_JOIN

Last but not least is the command ICAT_JOIN and it is used to coerce the CPU into joining a specific CLOS at runtime. This can definitely impact performance as observed empirically. So, unless it is absolutely warranted, usage of this command is advised against. Joining a CLOS in runtime is done by writing the desired CLOS to IA32_PQR_ASSOC. The code shown below reflects the same:

```
                if (ics.clos >= conf->nclos) {
                        return P4_E_CONFIG;
                }
                ecx = IA32_PQR_ASSOC;
                icatrdmsr(ecx, &eax, &edx);

                eax = 0;
                edx = ics.clos;
                icatwrmsr(ecx, eax, edx);

                icatrdmsr(ecx, &eax, &edx);
                icatsetclos(gp->cpuid, ics.clos);
```

As shown above, if an invalid CLOS is provided by the user, an error code of P4_E_CONFIG is returned.

The next section describes proposed usage of ICAT and caveats of certain use-cases.

7.1.3 Use Cases

As with any driver, there is some initial setup required with respect to configuration of ICAT. This includes providing the process with access to the file ”icat:”, which denotes the kernel driver itself, in VMIT. Once this is done, ICAT can be opened, one or more ioctl commands can be performed and, eventually, closed.

7.1.3.1 Initial discovery

In this specific use-case, ICAT is used to discover the components of IRDT supported by the platform. In order, to accomplish this, all that needs to be done is build the driver with verbose flag set, fuse it with the kernel and run the executable obtained on the platform. This should yield a report akin to the following:

```
icat: Initializing provider
icat: Intel RDT: Software control over shared res supported
icat: Intel RDT: L2 CAT supported
icat: Intel RDT: L2 CAT/CDP supported
icat: Intel RDT: L2 CBM Length: 4
icat: Intel RDT: L2 CLOS Length: 8
icat: Intel RDT: L2 CBM: 0
icat: CLOS setup done
icat: PMU: Architectural Version ID:LOS setup done
icat: PMU: Architectural Version ID: 5
icat: PMU: Number of MSRs available: 8
icat: PMU: Width of counter: 48
icat: PMU: Number of architectural events supported: 8
```

Figure 7.2: Discovery of ICAT

7.1.3.2 Static Configuration

In this specific use-case, ICAT is configured statically using the file *config.h*. As described earlier, the arrays **closcpu** and **closcbm** are configured with the respective CBMs and CLOSes which will be used by ICAT to setup the CPUs with respective access to LLC. This is the most straightforward of use-cases and requires no programming at all.

7.1.3.3 Analysis of cache usage

In this specific use-case, although *config.h* is configured so that the CPUs join their respective CLOSes, a userlevel application is supposed to be analyzed for its cache usage. This is done by first opening ICAT using the following call:

```
vm_open("icat0:0", VM_O_RD|VM_O_WR, &fd1);
```

Once this call succeeds, then an ioctl call is made using the command ICAT_ASSOC to associate the RMID with the CPU. And this is done as shown below:

```
ics.tid = p4_my_task();
vm_ioctl(&fd1, ICAT_ASSOC, &ics);
```

Care should be taken to set tid using p4_my_task() since ICAT expects this to be set by the user. Then, ICAT_MON can be used to calculate the cache usage as shown below:

```
vm_ioctl(&fd1, ICAT_MON, &ics);
```

This will, ultimately, return the respective RMID, cache usage provided by CMT and PMU events such as LLC references and misses. This data point will serve as an initial read. After this, the specific piece of code which should be analyzed can be run followed by another call to ICAT_MON which will provide another data point. Using these data points, one can analyze the cache used and the cache hit ratio.

Another possibility of usage exists if the average cache usage is required along with the standard deviation. This can be done by using ICAT_STAT call as shown below:

```
vm_ioctl(&fd1, ICAT_STAT, &ics);
```

As discussed earlier, this call will return the average cache used and the standard deviation of the set of samples that ICAT has obtained. Care should be taken to ensure that enough samples are taken using ICAT_MON before invoking ICAT_STAT.

7.1.3.4 Dynamic configuration

In this specific use-case, a static configuration was used initially to setup the respective CLOSes and CPUs. Eventually, need has arisen to join a different CLOS which might be inferior or superior to the current CLOS. In this case, the CLOS has to be first setup and then ICAT_JOIN ioctl is used to join the respective CLOS as shown below:

```
ics.clos = 10;
vm_ioctl(&fd1, ICAT_JOIN, &ics);
```

As with all functions provided by the system, the return value of vm_open, vm_close and vm_ioctl should be checked for errors and the file *icat.h* provides the respective data structure that is to be used with ioctl calls.

7.1.3.5 Caveats

There is a use-case which IRDT allows for, however, is not implemented in ICAT that this section talks about. This is the case of a uniprocessor PikeOS installation which runs more than one task, each of which requires different CLOS setup. This configuration was ignored not because of compatibility reasons but because of the fact that each time a context switch happens, a new CLOS must be joined for the respective task, and this by default invalidates the data in the cache and, thus, gets flushed. This leads to a significant performance drop since each context switch requires further cache warming before moderate performance can be attained. And this usually happens so frequently that the net effect would be of a machine that has none or significantly smaller amount of cache. Hence, this use-case was deliberately discarded.

Chapter 8

Evaluation of ICAT

This chapter deals with the empirical analysis of LLC usage and behavior of ICAT on x86_64 microarchitectures, namely, Broadwell and Tiger Lake. It elaborates on the results of the proposed use-cases which were detailed in the previous chapter and draws some conclusion based on the data obtained.

Allocation of last level cache, L3 and L2, brought about some interesting results. Firstly, L3 cache on Intel Broadwell microarchitecture was tested to see if increasing CBM actually increased the amount of the LLC that was allocated for use to the specific CPU. The graph shown below illustrates this:

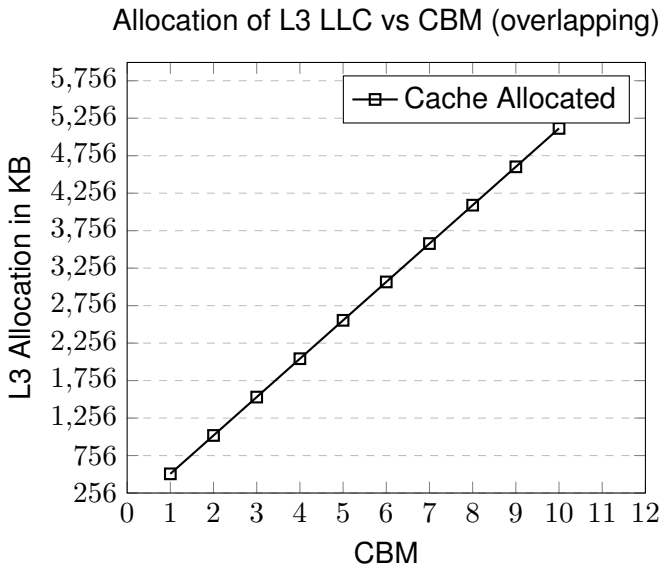

Figure 8.1: Allocation of L3 LLC vs CBM (overlapping)

CBM	Cache Usage in KB
0x1	512
0x3	1024
0x7	1536
0xF	2048
0x1F	2560
0x3F	3072
0x7F	3584
0xFF	4096
0x1FF	4608
0x3FF	5120

Figure 8.2: CBM (overlapping) and L3 LLC usage

In this experiment, a range of CBMs starting with 0x1 to 0x3FF were supplied using a static configuration. In order to exhaust the cache, 30MB of data was copied from uncached memory which was used as heap. This experiment was then conducted for each of the CBMs and the cache usage was recorded using CMT. Using the most minimal CBM, 0x1, a cache usage of 512KB was recorded and the maximum, 0x3FF, yielded 5120KB of allocatable cache.

As depicted by the plot, using a higher number of bits in CBM effectively increases the amount of cache available to the respective CPU. The resultant usage of cache was obtained using CMT by utilizing the ioctl command ICAT_MON. This substantiates that CAT/CMT can be used to allocate and monitor shared resources like LLC on PikeOS using ICAT.

The case of using isolated bits in CBM is more interesting. One might be inclined to think that one bit, set anywhere in CBM, would allocate the same amount of cache. This would, however, be a mistake since the position of the bit in CBM is just as important as the number of bits. The following table provides us with the relationship between the position of a single bit in CBM and the allocated LLC, on broadwell microarchitecture.

CBM	Cache Usage in KB
0x1	512
0x2	1024
0x4	1536
0x8	2048
0x10	2560
0x20	3072
0x40	3584
0x80	4096
0x100	4608
0x200	5120

Figure 8.3: CBM (isolated) and L3 LLC usage

In this experiment, L2 CAT support of ICAT was exercised. However, the target used for this test, which belongs to Tiger Lake microarchitecture, did not support CMT. Hence, a different approach was taken using PMU to draw a conclusion based on cache hit ratio. The plot below illustrates this:

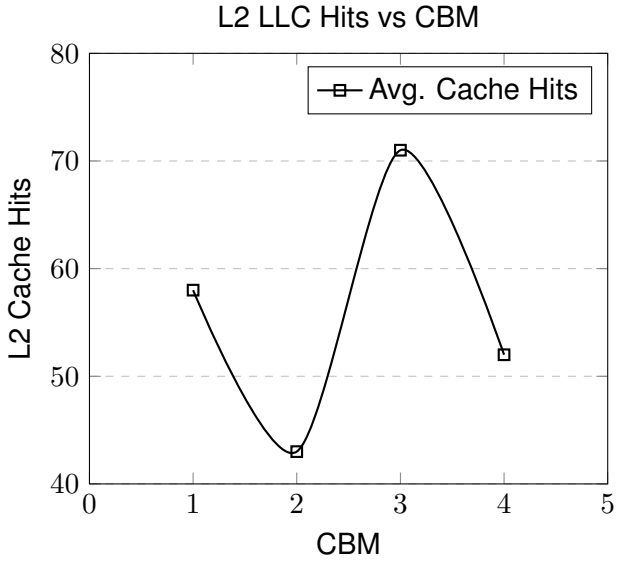

Figure 8.4: L2 LLC Hits vs CBM

Iteration	CBM	Avg. L2 LLC Hits
1	0x01	58
2	0x03	43
3	0x07	71
4	0x0F	52

Figure 8.5: CBM and L2 LLC Hits

In this experiment, the cache was exhausted by copying data to/from uncached memory used as heap. However, CMT could not be used to measure the amount of cache available for allocation.

This plot, however, didn't end up as telling as the one above. As illustrated by the plot, improvement of cache hit ratio does not only depend on the amount of cache available but also on the code used. In this specific case, it increases when using 3 bits of CBM, seems to be moderate when using 1 or 4 bits and degrades when using 2 bits of CBM. This result should not be considered to be reflective of the performance of the application in any definitive degree, since PMU is not considered to be entirely reliable for the purpose of statistical analysis, as is done here.

This experiment exercises changing of CLOS in run-time to check if it actually yields the intended effect. The plot below illustrates this:

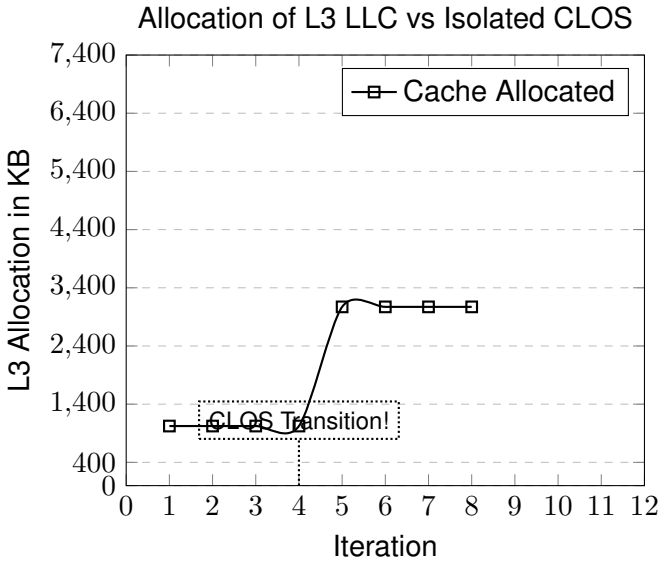

Figure 8.6: Allocation of L3 LLC vs Isolated CLOS

Iteration	CBM	Cache Usage in KB
1	0x30	1024
2	0x30	1024
3	0x30	1024
4	0x0F	1024
5	0x0F	3584
6	0x0F	3584
7	0x0F	3584
8	0x0F	3584

Figure 8.7: CBM (Isolated) and L3 LLC usage

As you can see, changing CLOS in runtime affects the performance by momentarily invalidating and flushing the cache, although this is done transparently by the hardware. In this case, two isolated CBMs were used and iterations 1-3 were conducted using CLOS1, CBM: 0x30, which had less bits set in CBM than CLOS2, CBM: 0x0F. The IOCTL cmd ICAT_JOIN was used to switch from CLOS1 to CLOS2 after iteration 3 which should have lead to improved performance. However, this change in CLOS does not come in effect right away instead the performance suffers as shown in iteration 4. Subsequently, the cache gets warmed up and further iterations, 5-8, show expected usage of LLC. However, the case of overlapping CLOSes is a bit different and is illustrated below:

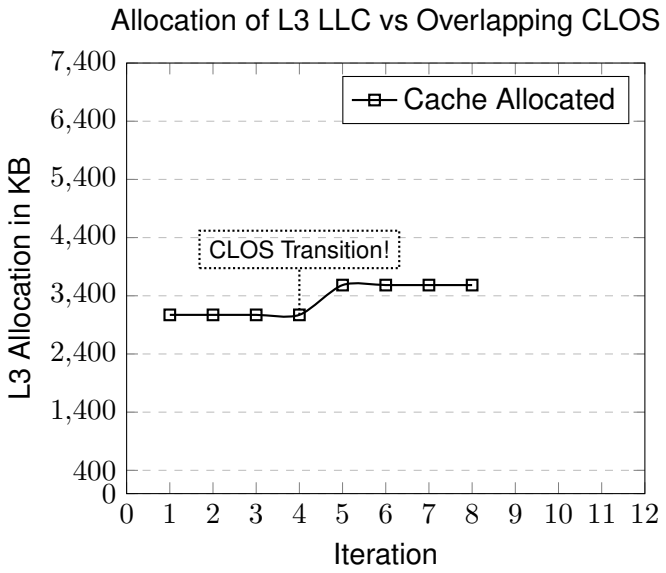

Figure 8.8: Allocation of L3 LLC vs Overlapping CLOS

Iteration	CBM	Cache Usage in KB
1	0x3F	3072
2	0x3F	3072
3	0x3F	3072
4	0x7F	3072
5	0x7F	3584
6	0x7F	3584
7	0x7F	3584
8	0x7F	3584

Figure 8.9: CBM (Overlapping) and L3 LLC usage

As you can see in this case, since the CLOSes were overlapping, the cache usage merely stays the same after the CLOS transition that happens after iteration 3. Hence, iteration 4 shows the same cache usage as iteration 3, and, the amount of allocatable cache increases in iteration 5 as expected.

The next experiment demonstrates the noisy-neighbor scenario and deals with 2 applications trying to access LLC run on CPU1 and CPU2, respectively, both utilizing the default CBM of 0xFFF.

Iteration	CPU	CBM	Cache Usage in KB
1	1	0xFFF	2800
1	2	0xFFF	0
2	1	0xFFF	2928
2	2	0xFFF	2928
3	1	0xFFF	2928
3	2	0xFFF	3024
4	1	0xFFF	2944
4	2	0xFFF	3024
5	1	0xFFF	3024
5	2	0xFFF	3040
6	1	0xFFF	3040
6	2	0xFFF	3040
7	1	0xFFF	3072
7	2	0xFFF	3088
8	1	0xFFF	3088
8	2	0xFFF	3104
9	1	0xFFF	3088
9	2	0xFFF	3216
10	1	0xFFF	3104
10	1	0xFFF	3360

Figure 8.10: Iteration vs L3 LLC usage (without CAT)

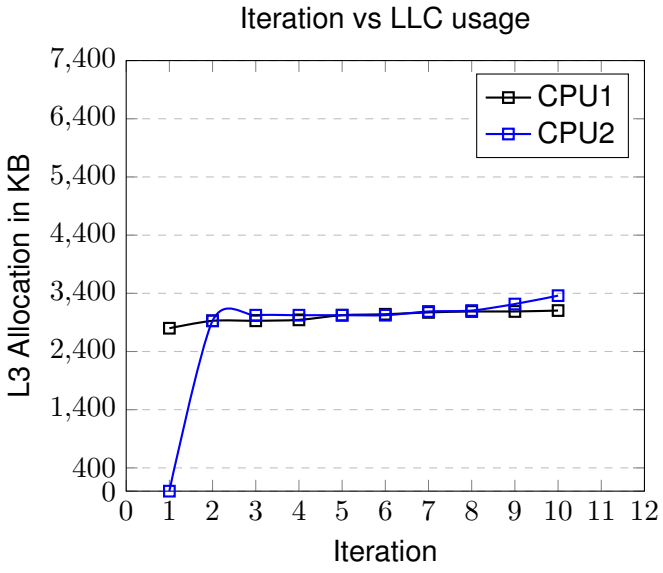

Figure 8.11: Noisy-neighbor scenario - problem

As you can see in this case, both CPUs are trying to allocate as much as possible causing contention. This can be solved using ICAT by choosing different CBMs. In the case shown below, CPU 2 utilizes a CBM of 0x3F and CPU 1 utilizes the default CBM of 0xFFF, thereby, giving CPU1 access to more cache.

Iteration	CPU	CBM	Cache Usage in KB
1	1	0xFFF	112
1	2	0x3F	0
2	1	0xFFF	3424
2	2	0x3F	1440
3	1	0xFFF	4032
3	2	0x3F	1440
4	1	0xFFF	4256
4	2	0x3F	1520
5	1	0xFFF	4480
5	2	0x3F	1520
6	1	0xFFF	4528
6	2	0x3F	1520
7	1	0xFFF	4560
7	2	0x3F	1520
8	1	0xFFF	4560
8	2	0x3F	1600
9	1	0xFFF	4576
9	2	0x3F	1616
10	1	0xFFF	4592
10	1	0x3F	3072

Figure 8.12: Iteration vs L3 LLC usage (with CAT)

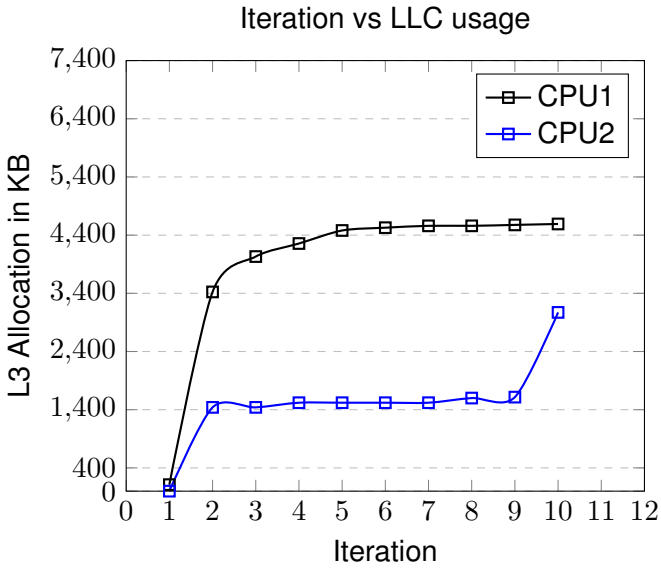

Figure 8.13: Noisy-neighbor scenario - solution

Another important observation was that CLOS0 has access to even more cache than that can be allocated using CAT. As it was observed in the discovery phase, on our target, the length of CBM supported is 12 bits, however, the number of allocatable bits is 10. There are 2 bits which were filled, by default, usually assigned to the graphics engine or other

hardware units (Intel, 2025b, p. 19-59). However, this extra cache can be utilized by CPUs which use the default CLOS0. In our experiments, it was possible to allocate upto 6144KB of LLC on broadwell microarchitecture using CLOS0.

Chapter 9

Conclusion

The research done has resulted in the implementation of ICAT on PikeOS. As described in previous chapters, the driver ICAT was able to expose an API which makes use of various subcomponents of IRDT such as CAT, CMT etc. as well as generic architectural components such as PMU. It was possible to analyse use-cases of ICAT and caveats arising from certain design choices. The implemented prototype is available upon request.

Implementation of ICAT has opened the doors to mechanisms which can make use of IRDT for various purposes on PikeOS such as optimization of application, optimization of stacks such as network stack, analysis of software behavior from the perspective of shared resources such as cache and memory, improving WCET analysis by being able to predict cache content, adherence to avionics standards etc.

In short, it was possible to show that IRDT can be utilized in real-time operating systems like PikeOS for the purpose of allocating and monitoring shared resources. The driver helped with being able to ensure that PikeOS can comply with the strict regulations of embedded systems imposed by avionic standards as described by Jean et al. (2012), ultimately, contributing to the field of Real-Time OS and Systems Research.

9.0.1 Future Work

There exists a possibility of allocating another shared resource, namely, memory bandwidth, which is of importance in a SMP environment. Memory bandwidth, as described by Jean et al. (2012, p.56), is the maximum amount of data that the memory bus can handle at a point in time. Shanley (2005, p.839) states that the FSB is what connects the processor with other devices on x86 platforms; embedded systems which run critical software should be wary of applications which over-utilize the bandwidth of this bus. As stated by Jean et al. (2012, p.57), one of the approaches that can be pursued is to schedule the applications accordingly so as to not exhaust the bandwidth. However, another approach is to utilize MBA, which is a sub-component of IRDT, to allocate memory bandwidth to specific CPUs which run applications with higher levels of criticality. Although ICAT handles the discovery of MBA, the logic for allocation and monitoring is not implemented yet. Extending ICAT to handle MBA/MBM would be beneficial to PikeOS for deterministic analysis of software which is important in real-time environments as well as for reducing interference from neighboring partitions.

Bibliography

Schimmel, C. (1994) Unix Systems for Modern Architectures, Boston, Addison–Wesley

Shanley, T. (2005) The Unabridged Pentium 4: IA32 Processor Genealogy, Boston, Addison–Wesley

SYSGO GmbH (2025a), PikeOS Kernel Reference Manual Version: D5.1-312, SYSGO GmbH, Mainz

SYSGO GmbH (2025b), PikeOS Platform Manual for x86-amd64 Boards Version: D5.1-618, SYSGO GmbH, Mainz

SYSGO GmbH (2025c), PikeOS PSP and KDEV Developer's Guide Version: D5.1-318, SYSGO GmbH, Mainz

SYSGO GmbH (2025d), PikeOS User Manual Version: D5.1-1153, SYSGO GmbH, Mainz

Neville-Neil, G., McKusick, M., and Watson, R.N.M, (2004) The Design and Implementation of the FreeBSD Operating System (2nd Edition), Boston, Addison–Wesley

Sloss, A.N. (2004) ARM system developer's guide, Massachusetts, Morgan Kaufmann Publishers

Intel Corp. (2015), Improving Real-Time Performance by Utilizing Cache Allocation Technology, Doc Version: 331843-001US, Available at: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf

Intel Corp. (2025a), Intel® Resource Director Technology Framework, Available at: https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html [Accessed 29 April 2025]

Intel Corp. (2025b), Intel® 64 and IA-32 Architectures Software Developer's Manual, Doc Version: 325462-085US, Available at: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Liedtke, J. (1995) On µ-Kernel Construction, In Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP '95). Association for Computing Machinery, New York, NY, USA, pp.237-250, Available at: https://doi.org/10.1145/224056.224075

Jean, X., Gatti, M., Berthon, G., Fumey, M. (2012) MULCORS Project: The Use of Multicore Processors in Airborne Systems, Rev. 07, France: Thales Avionics

Selfa, V., Sahuquillo, J., Eeckhout, L., Petit, S., Gómez, M. E. (2017), Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), Portland, OR, USA, pp. 194-205, Available at: https://doi.org/10.1109/PACT.2017.19

Kim, Y., More, A., Shriver, E., and Rosing, T. (2019), Application Performance Prediction and Optimization Under Cache Allocation Technology, 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, pp. 1285-1288, Available at: https://doi.org/10.23919/DATE.2019.8715259

Farshin, A., Roozbeh, A., Maguire, G.Q., and Kostić, D. (2019), Make the Most out of Last Level Cache in Intel Processors, In Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19), Association for Computing Machinery, New York, NY, USA, Article 8, pp.1–17, Available at: https://doi.org/10.1145/3302424.3303977

Kaiser, R., and Wagner, S. (2007), Evolution of the PikeOS Microkernel, MIKES 2007 First International Workshop on Microkernels for Embedded Systems, pp. 50-56, Available at: https://www.researchgate.net/publication/285592141_Evolution_of_the_PikeOS_Microkernel

Institute of Electrical and Electronics Engineers (2004), IEEE Std 1003.13-2003 (Revision of IEEE Std 1003.13-1998): Institute of Electrical and Electronics Engineers

ARINC (2010), Avionics Application Software Standard Interface: ARINC Specification 653 Part 1, Required Services

Lorido-Botran, T., Huerta, S., Tomas, L., Tordsson, J., & Sanz, B. (2017), An unsupervised approach to online noisy-neighbor detection in cloud data centers, Expert Systems with Applications, pp. 188-204, Available at: https://doi.org/10.1016/j.eswa.2017.07.038

Straumann, T. (2001), Open Source Real Time Operating Systems Overview, SSRL, Menlo Park, USA, Available at: https://doi.org/10.48550/arXiv.cs/0111035

Guan, F., Peng, L., Perneel, L., & Timmerman, M. (2016), Open Source FreeRTOS as a case study in real-time operating system evolution, The Journal of Systems and Software, pp.19-35, Available at: https://doi.org/10.1016/j.eswa.2017.07.038

Hendrich, A., Verplanke, E., Autee, P., Illikkal, R., Gianos, C., Singhal, R. & Iyer, R. (2016), Cache QoS: From Concept to Reality in the Intel® Xeon® Processor E5- 2600 v3 Product Family, 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Available at: https://doi.org/10.1109/HPCA.2016.7446102

Declaration of Authenticity

I, Naveen Narayanan, hereby declare that the work presented herein is my own work completed without the use of any aids other than those listed. Any material from other sources or works done by others has been given due acknowledgement and listed in the reference section. Sentences or parts of sentences quoted literally are marked as quotations; identification of other references with regard to the statement and scope of the work is quoted. The work presented herein has not been published or submitted elsewhere for assessment in the same or a similar form. I will retain a copy of this assignment until after the Board of Examiners has published the results, which I will make available on request.