SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis and Implementation of AMD IBS Sample Collection for Performance Analysis Purposes

Maximilian Geitner

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis and Implementation of AMD IBS Sample Collection for Performance Analysis Purposes

Evaluierung und Nutzung von AMD‘s IBS Sample Collection für Leistungsanalyse

Author:	Maximilian Geitner
Supervisor:	Prof. Dr. rer. nat. Martin Schulz
Advisor:	Stepan Vanecek, M.Sc.
Date:	17.07.2023

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

Munich, 17.07.2023　　　　　　Maximilian Geitner

Acknowledgements

For the completion of this thesis, I appreciate the support I have received for working on this thesis. First, I want to thank the Chair of Computer Architecture and Parallel Systems for offering me the opportunity to work on this thesis and access to the CAPS cloud infrastructure. I appreciate the guidance I have received from Stepan very much, and I have learned a lot in half a year. Also, I want to thank Martin for his great help in finding mistakes in my thesis drafts. I also want to thank my family for their continuous support in the past few years during my studies and, of course, for this thesis.

Abstract

Instruction-based sampling (IBS) is a statistical profiling method developed by AMD and has been available in AMD processors since 2007. This technique features the data collection of the two event types, IBS Fetch and IBS Op. These two classes of events provide various metrics related to the front- and back-end of the pipeline. In contrast to the Intel PEBS solution with configurable programmable counters, IBS cannot choose specific metrics usually available with hardware performance counters.

IBS is implemented in the profiling interface perf_event. Perf_event is part of the Linux kernel since version 2.6 and developed in cooperation with hardware vendors in order to use functionalities related to the performance monitoring unit (PMU) of the processor. Perf_event provides the system call `perf_event_open()` with the purpose of making performance metrics accessible to other profiling tools. One such tool is Mitos, which has been under development at the Lawrence Livermore National Laboratory and is now modified with new features at the Chair of Computer Architecture and Parallel Systems.

This work implements the data collection functionality utilizing IBS in Mitos. It takes a closer look at the capabilities of perf_event with IBS, various implementation aspects, and how IBS is usable in MPI and OpenMP applications. Currently, IBS requires system-wide data collection privileges and does not support per-process monitoring. Due to the missing per-process monitoring option, monitoring all processor cores individually and filtering event samples on a higher level might be a suitable workaround method. The other workaround idea only observes processor cores executing the target application and automatically moves with the target process on change detection. However, the detection implementation has performance-related issues and requires further modifications.

Contents

1. Motivation

In HPC systems, looking at various aspects to improve efficiency and performance has always been important. The popular TOP500 list contains the most powerful systems in the world, the current leader in the June 2023 ranking is the Frontier supercomputer with 8,699,904 cores and power consumption of 22,703kW [Tea23]. This system uses AMD 3rd generation EPYC 64C 2GHz processors and AMD Instinct MI250X accelerators.

However, the hardware is only one aspect that needs to be considered. The software also plays a massive role in choosing a new computing platform. Compatibility and software support are often major reasons for advocating for certain systems. Although new ARM-based microarchitectures have arrived on the market and have advantageous characteristics such as high performance at relatively low power consumption, x86 processors still exist. The reason is the related software. This might be modern games, old commercial software, or software for HPC clusters using certain functionalities only available on a specific hardware and software stack.

Sometimes, the hardware alone is not the main reason to buy a certain system. It could be the toolchain that works together with the hardware. One such example is the toolkit CUDA and Nvidia accelerators in the artificial intelligence sector. The toolchain could also be a framework of profilers, compilers and other utility programs binding users to a certain vendor. The reasons might be the hype from other people, existing knowledge about the system or in the end the better software support.

Software support is also relevant in HPC and necessary for designing resource efficient applications. To analyze potential bottlenecks, finding a suitable profiling tool that can discover potential problems is necessary.

With Intel PEBS, a profiling solution is already available since the NetBurst architecture in 2000. [Int23a] The other vendor, AMD, has a similar solution with instruction-based sampling (IBS). This solution has been available since 2007 and is a response to the request for precise sampling. It is interesting to know about their data collection capabilities and their usability in practice. To answer the question, knowledge about processor design, performance monitoring, the functionality of profiling tool interfaces and their usage is necessary.

2. Theoretical Background

This chapter is about important basics to better understand profiling tools. It starts with processor design decisions and how to implement an efficient pipeline and memory architectures. This information is relevant for the next part about the PMU, its interaction with the pipeline, and what kind of monitoring techniques exist. The remaining parts cover Intel PEBS and AMD IBS, two different solutions for precise sampling, and profiling interfaces for two common application types in HPC and its interface for profiling tools, MPI and OpenMP.

2.1. Processor Design Decisions

2.1.1. Pipeline-Stages in Superscalar Processors

The concept of an instruction pipeline is essential for modern processor architectures to improve their instruction-level-parallelism (ILP) cost-effectively. A pipeline divides the workload of an instruction into multiple smaller subtasks and processes these in the corresponding number of stages. Although this approach increases the system's complecity and may increase the duration to process a single instruction, there is the benefit of performing work on multiple instructions simultaneously. [Gon11]

It is desirable to work on multiple instructions at once, but there are certain aspects pipeline design needs to consider for an efficient implementation. One primary goal is to have stages that complete their subtask in roughly the same amount of time. Slower subtasks could slow down successive stages by forcing them to wait for their next assignment and, as a result, reduce pipeline efficiency. These issues can be masked by employing multiple units to solve subtasks at potential bottleneck stages, reorder subtasks for a more efficient workflow or implement buffers between stages.

Buffers temporarily store information related to an instruction until the next stage has completed a previous assignment. Buffers might fill up if the successive stage cannot complete subtasks, such as memory accesses, as fast as previous stages in the pipeline and empty in cases where the stage is faster than its predecessor stage. Thus, buffers provide options to keep pipeline stages running and prevent them from falling into an idling state due to unavailable subtasks or by being unable to hand over subtask results to their successor stage and resume working. Technical reasons still prevent buffers from resolving problems such as cache misses or more extended periods of difficult computational tasks. Implementing larger buffers can be costly by taking away space on the processor and not improving efficiency. Also, implementing too many stages in a pipeline does not necessarily improve the pipeline's efficiency. At some point, too small subtasks make processor design increasingly complex. There is a particular trade-off between the number of stages and the increase in performance. [She13]

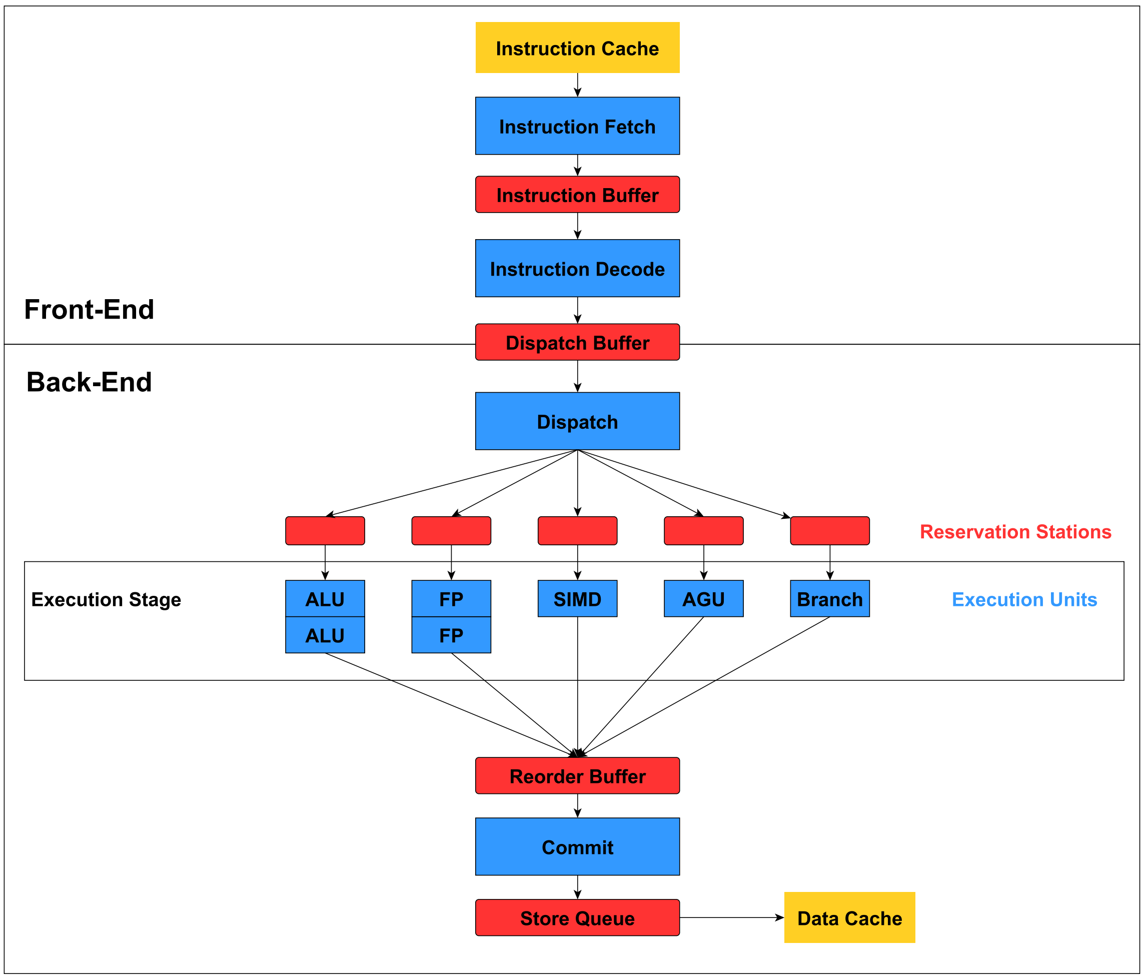


Figure 2.1.: Pipeline Overview.

The following example in Figure 2.1 shows how a small instruction pipeline could be separated into different stages. An instruction pipeline needs to handle many different kinds of instructions. Instructions are usually classified into one of the following categories:

- load/store instructions
- conditional instructions
- arithmethic and logical operations

The first stage is the instruction fetch unit, which uses the instruction pointer to determine the next instruction. This step requires an address translation involving the TLB and access to the instruction cache. In unfortunate cases, an expensive main memory access might also be necessary. Another concern is the occurrence of branches in a program. The unit's branch predictor predicts whether the branch is taken or not to continue the instruction execution. Predicting the correct fetch address involves the branch target buffer (BTB) and a branch predictor. An incorrect prediction involves a costly pipeline flushing resulting in a performance loss. [Gon11]

The next stage decodes the instruction, and this step transforms the instruction into one or multiple micro-operations depending on the complexity of the task. Micro-operations are small and simple instructions that are processed in one cycle at the following execution stage. The first two stages belong to the front-end of a pipeline, and instructions are processed in-order.

Unlike the first two stages, out-of-order execution allows the reordering of operations in order to increase performance. It is usually used on modern x86 processors and ARM-based high-performance processors. The out-of-order execution is implemented in the following stages of the pipeline and is part of the back-end of the pipeline. After the decoding stage, the micro-operation is dispatched into the pipeline, issued, and the execution starts. The related dispatch stage assigns register operands and the associated resources to the micro-operation. Consecutive micro-operations might use these register operands and could cause data dependencies. Some are artificial name dependencies, and the technique register renaming removes them before the execution stage. Between the dispatch stage and the execution stage are special buffers, the so-called reservation stations, storing micro-operation-related information until the execution unit is ready. The micro-operations wait until all operands are ready and then issued to execution units.

In the execution stage, the actual micro-operation performs its specified workload. Common execution units are:

- Arithmetic Logic Unit (ALU)
- Branch Unit
- Floating-Point Unit
- SIMD unit
- Address Generation Unit (AGU)

Quite special are memory operations, they either load data from memory or store register values in the main memory. Memory operations specify the memory address with several source operands. There is a dedicated address generation unit (AGU) involved in generating the effective physical address by using a base address, an offset, and potentially other parameters. The cache is between the main memory and the processor, which stores data temporally for faster memory access. It is desirable to put often accessed data in the data cache and have cache hits in the case of future data requests to the same location, as main memory access is relatively slow. A cache miss generates a time penalty for accessing the slower main memory. All the other execution units working on non-memory-access-related tasks use register operands to compute the result. Operation results move to the reorder buffer (ROB). Its location is between the execution and the commit stage. The reorder buffer is necessary for out-of-order processors. It is part of the process to revert the micro-operation order to its original state as specified in the application.

Modern processors can decode multiple instructions per cycle. Therefore, later stages should be able to also process multiple micro-operations per cycle. The execution stage can have more than one execution unit in order to process multiple pending micro-operations. These execution units are generally divided into different types and can only handle specific operations. For example, there could be two integer units and two floating point units in the execution stage which could execute four different independent operations per clock cycle in the best-case scenario. Out-of-order execution improves the possibility of utilizing all execution units in the pipeline by finding an operation order that can be executed simultaneously. Additionally, it might reduce stalls in the pipeline due to cache misses.

The last stage, the commit stage, writes back the result to the specified register and commits the changes. After the write-back, the operation changes to the “completed”-state, and is often also declared as “retired”. Memory operations need to finalize memory access changes and have a delay of a few clock cycles until they retire. A store queue or store buffer helps in reducing the time to retirement, and it stores values until they can move to the main memory.

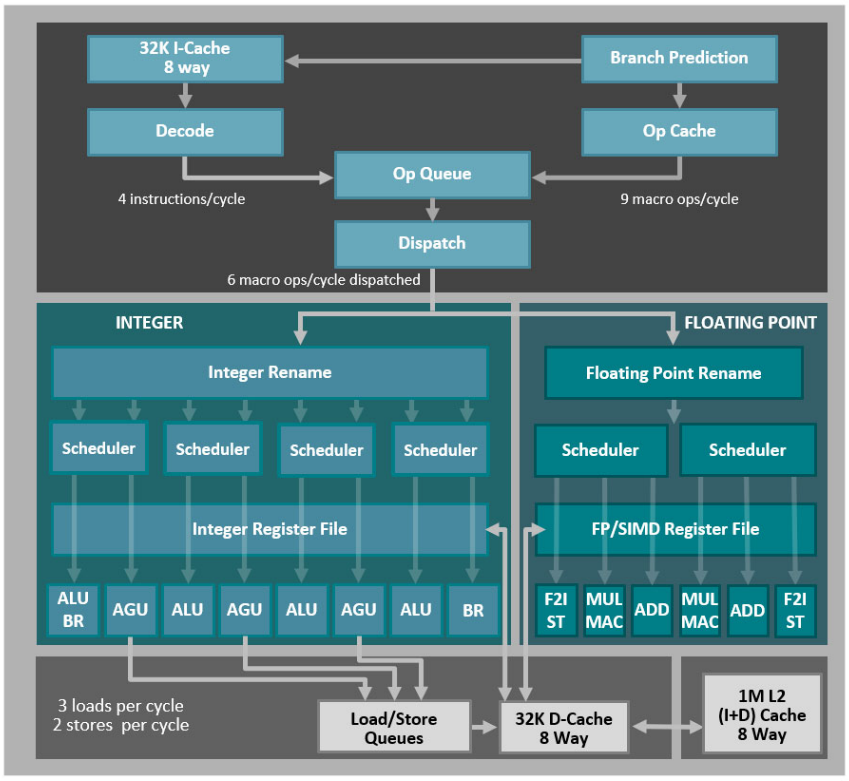


Figure 2.2.: Instruction Pipeline in Zen 4. [AMD23a]

Figure 2.2 shows an instruction pipeline implemented in the AMD Zen 4 microarchitecture. AMD defines the three following types of operations varying in complexity:

- instructions
- macro-operations
- micro-operations

One Instruction decodes to one or multiple macro-operations. The dispatch of a macro-operation is the transfer from the front-end to the back-end of the pipeline. A macro-operation may be issued as multiple micro-operations which perform even simpler operations, such as arithmetic or load/store operations in the pipeline. [AMD23a, AMD23c]

2.1.2. Caches

For a processor, it is desirable to have storage as fast as registers. However, registers are relatively costly and require a lot of space on the chip. Thus, using caches is a trade-off between speed and cost efficiency. Modern processors also employ several tiers of cache, the L1, L2, and L3 cache. The fastest cache is the L1 cache, which is usually a small SRAM with a size of a few KB and exclusively used by a single processor core. The L2 cache is another private cache with slower memory access timings than the L1 cache. However, it has a larger cache size in the range of a few MB. The largest cache is the L3 cache. It is typically implemented as a cache with even more available memory shared between multiple cores. The exact specification depends on the vendor and architecture. Besides the different cache tiers, it is common to separate between instruction and data caches on the fastest L1 cache. Necessary for cache locality, this separation exists on x86 processors from both vendors, Intel and AMD.

Another relevant cache is the translation lookaside buffer, also called TLB, which contains a small number of entries. Its purpose is address translation and it is designed in a fully-associative way, and allows the processor to do a fast lookup on recent entries. [She13]

2.2. Performance Monitoring on the Hardware-Level

This section focuses on analyzing programs by using different techniques modern processors provide. In order to collect performance-related data, such as the number of cycles or cache misses in a program run, the processor has a dedicated hardware component called a performance monitoring unit (PMU). These low-level metrics are made available as events by the PMU and can then be analyzed by profiling tools such as PERF_Events, Intel VTune, or AMD µProf. [Kuk15] The following subsections focus on different evaluation methods utilizing the PMU.

2.2.1. Hardware Performance Counters

Performance monitoring counters (PMCs) are special registers in the PMU that increment their value when certain hardware events occur in the processor pipeline or other parts of the computer. They can also measure the duration of certain events. [AMD10] The CPUID instruction register provides information about supported features by each CPU. Other sources for finding the available events are Intel® 64 and IA-32 Architectures Software Developer's Manuals [Int23a], AMD's BIOS and Kernel Developer's Guide (BKDG) [AMD10, AMD11, AMD16], AMD's Processor Programming Reference (PPR) [AMD20, AMD21, AMD23c] and Intel's online documentation for performance monitoring events [Int23b].

Performance counters vary in the degree of software configuration and their event sources. Common are programmable counters that can select an event in software. This approach is ideal for letting the user choose a small subset of events. Modern processors have thousands of available metrics. Programmable counters are usually distributed across the pipeline to be closer to the source of the performance metrics. The locality of the counters and the complexity of wiring the counters to the information sources set limitations in the event selection of the programmable counters, some counters might only be able to record cache events, and others are limited to branch speculation events. Configuring one programmable counter requires a certain event code that activates the necessary logical unit and an umask that allows filtering events. Modern AMD and Intel x86 processors usually offer four to eight programmable counters per core. [AMD23b, Int23a]

The opposite of a programmable counter is the fixed-function counter. It continuously monitors one specific event. Suitable events for fixed-function counters are metrics regularly used in monitoring programs. Implementing these metrics in dedicated hardware, it allows the user to use the programmable counters for other purposes. Fixed-function counters on Intel systems record important metrics such as instructions retired, the number of unhalted core cycles, or the number of unhalted reference cycles. [Kuk15]

Depending on the microarchitecture, there might also be additional counters collecting information not related to a single processor core. Uncore refers to shared resources outside of the processor core. One example is the shared L3 cache. An uncore PMU is available on Intel and AMD processors. [AMD23c, Int23a, Kuk15] AMD also uses northbridge event as another name for uncore event. [Wea17]

The Intel Core 13th generation Intel Core processors are hybrid processors using a combination of performance cores and more efficiency-oriented cores, called P-core and E-core, respectively. The P-cores are based on the Golden Cove microarchitecture and offer three fixed-function counters and eight programmable counters per thread for the P-cores. There are some differences in the E-cores based on the Gracemont microarchitecture compared to the larger P-cores. They have no Hyperthreading support and cannot run two threads on one core. Additionally, the amount of programmable counters has been reduced to six.

The microarchitecture specifies which events are supported in the PMU. Intel separates events into two classes: Architectural events define metrics available across multiple microarchitectures and have the same or at least a similar specification. These types of events only relate to a small subset of events. Non-architectural events with metrics exclusive to a specific microarchitecture are the more common class. The disadvantage of non-architectural events is the missing guarantee of existence in future processor generations. [Int23a]

Compared to Intel, AMD has a similar configuration to offer for its PMC implementation. According to the AMD64 Architecture Programmer's Manual, all implementations have at least four core performance counters. The processor has special registers that indicate additional core performance counters, cache event counters, and support for other event types.. [AMD23b] The current Zen 4 microarchitecture has six core performance event counters per thread, six performance events counters per L3 complex, and sixteen Data Fabric performance events counters related to DRAM access metrics. [AMD23c] Detailed event codes are described either in the Processor Programming Reference (PPR) [AMD20, AMD21, AMD23c] or in the BIOS and Kernel Developer's Guide (BKDG) for older microarchitectures. [AMD10, AMD11, AMD16]

2.2.2. Event-Based-Sampling

Event-based-Sampling is a statistical profiling technique that takes performance metrics in regular intervals of a program run. The sampling interval is decided by a dedicated counter that increases after each cycle. Suppose the counter reaches a certain threshold or a counter overflow occurs. In that case, the PMU interrupts the program execution and collects information about the current program state from the related hardware counters and related MSRs. The CPU creates samples containing data from PMUs and usually stores the data in the file system for analysis. The interval between the two samples depends on the implementation. It might be defined by the number of cycles or given as a period in milliseconds. This statistical profiling technique allows more fine-grained information gathering than simple hardware counters due to the increased amount of data. Furthermore, certain implementations make it possible to link the event sample to a specific part of the program. By aggregating the samples for each part of the program, the overview helps to find potential bottlenecks indicated by relatively many samples.

However, in non-precise interrupt routines, there is a problem in the data collection process. Until the interrupt arrives at the event source and initiates the data collection for a specific instruction, the pipeline might have moved forward, and the PMU receives information about an instruction executed a few cycles later. This delay between the request for the information and receiving an answer is called skid. [Int23a, W$^+$16] Large skid values lead to a disconnect between the event samples and the instruction that caused this program state. Therefore, having either no or a very low skid is desirable while sampling applications. Section 2.3 and Section 2.4 cover vendor-specific solutions for reducing skid.

2.3. Intel Processor event-based sampling (PEBS)

The Intel Pentium processors introduced model-specific registers (MSRs) as performance monitoring counters supporting non-precise data collection. Only later, the Intel NetBurst processors implemented the more precise monitoring feature, Processor event-based sampling (PEBS), to reduce skid. [Int23a] Sometimes, PEBS is incorrectly written as Precise event-based sampling due to its purpose to remove the latency between issuing the data collection for a specific instruction and retrieving the related information from the processor pipeline. [Kuk15, W$^+$16] Significant delays could lead to incorrectly fetched values related to instructions issued a few cycles later. These special counters can be configured to select specific processor performance parameters, which are then monitored to tune the system and improve compiler performance. [W$^+$16]

Since the Intel NetBurst microarchitecture, the processor has a debug store (DS) save area that collects information in a memory-resident buffer for further processing by debugging and profiling tools. It stores PEBS records and branch trace store records (BTS). BTS records are another type of information separate from PEBS, and they record traces about taken branches, occurring interrupts, and exceptions to determine how a certain code location has been reached. [Int23a]

Across several microarchitecture generations, Intel improved PEBS by adding more selectable events and introducing architectural events in the Intel Core Solo and Intel Core Duo generation. Architectural events, such as unhalted core cycles or instructions retired, refer to metrics available across several microarchitectures. Unfortunately, most available events belong to the group of non-architectural events, processor generation-specific metrics. A disadvantage is the missing portability across several generations and the increased workload to configure events in profiling tools. Besides introducing architectural events, several extensions of the PEBS have been implemented to improve other aspects of the data collection. To reduce the skid problem, the processor can signal an imminent counter overflow and improve trapping the correct value upon the moment of the actual data collection across all pipeline stages (Precise Distribution of Instructions Retired). Allowing the selection of non-precise events with the feature extended PEBS since Icelake makes it possible to choose from a broader range and combinations of events utilizing PEBS. [Int23a]

2.4. AMD Instruction-Based Sampling (IBS)

Instruction-based sampling (IBS) is a solution by AMD and has been implemented in their processors since 2007, introduced with the AMD Family10h processors "Barcelona" (AMD Opteron Quad-Core processor). It is a statistical method and an implementation reducing skid, providing precise program performance information similar to Intel PEBS. [Dro07]

For IBS, the data collection can collect information about the front- or back-end of the instruction pipeline. These events are called IBS Fetch and IBS Op, respectively. Each IBS event type has its own model-specific registers (MSRs). These can be control registers containing the counter triggering an interrupt upon an overflow, configuration parameters, or data registers with the values collected from the pipeline. Compared to PEBS, the provided data is bound to the IBS event type and the processor specification and cannot be customized like programmable hardware counters with thousands of selectable events.

IBS has been officially supported by perf_event since kernel version 3.2 [Gre21], but some implementation details make it more difficult to use than Intel's PEBS solution. Currently, monitoring in a per-process mode is impossible and instead requires the selection on a per-CPU basis. Besides the Linux kernel requirement, AMD systems could utilize perf_events correctly if the event source is configured correctly. Existing folders at `/sys/bus/event_source/devices/ibs_fetch` and `/sys/bus/event_source/devices/ibs_op` indicates sampling support.

2.4.1. IBS Fetch

This event type provides data related to the pipeline's instruction decoding and instruction fetch phases. The control register Core::X86::Msr::IBS_FETCH_CTL contains 16 bits of the internal 20-bit counter, the maximum count value of the counter, values, and flags related to the tagged fetch operation. Values such as instruction fetch latency and instruction cache L1TLB page size require multiple bits. Flags need one bit and define events describing metrics from the pipeline like L2 cache miss, instruction cache misses, or providing configuration settings about IBS Fetch. One such setting in IBS Fetch is the flag IbsRandEn which defines whether the last four bits of the fetch counter are randomized or not when starting the fetch counter. This behavior is desirable for cases where periodic instruction fetches might result in repeatedly sampling the same instruction in a computational loop. With the randomizing of the fetch counter, this pattern might be avoided, and the following sample could be related to a different instruction.

Not only the 64-bit control register belongs to the IBS Fetch MSR-registers, there are two additional 64-bit registers called `Core::X86::Msr::IBS_FETCH_LINADDR` and `Core::X86::Msr::IBS_FETCH_PHYSADDR` containing the virtual 64-bit address and the 48-bit physical address. Sometimes, the control register indicates an invalid physical address with the `IbsPhyAddrValid` field set to zero.

The last register belonging to IBS Fetch is the extended control register with one single 16-bit field. This metric measures the number of cycles when the fetch engine is stalled for an ITLB reload for the sampled instruction fetch. If there is no reload, then the value is zero. Microarchitectures before the AMD Zen generation (AMD Family 17h) do not have this register [AMD10, AMD11, AMD16] and therefore it is recommended to check the feature set of the processor with the CPUID command before sampling with IBS Fetch.

2.4.2. IBS Op

IBS Op provides information about the backend of the instruction pipeline. Therefore event samples are generated for tagged macro operations instead of instructions as sampled by the IBS Fetch sample type. Like IBS Fetch, the event type IBS Op has its own control register `Core::X86::Msr::IBS_OP_CTL` with a 27-bit internal counter and fields for the sampling configuration. The following register `Core::X86::Msr::IBS_OP_RIP` contains only the linear address of the tagged macro-operation. Four additional registers `Core::X86::Msr::IBS_OP_DATA` to `Core::X86::Msr::IBS_OP_DATA3` capture detailed information about the marked macro operation. These metrics describe the operation type, TLB metrics, and data cache latencies related to this event. Some fields have conditions for their validity. Examples are the fields `taken branch op` and `mispredicted branch op`, which contain relevant data when the condition `branch op retired` is set to one. Another case is the register `Core::X86::Msr::IBS_OP_DATA2`, which includes data source fields and is only valid for load operations without any hits in the L1 data cache or L2 cache.

The next registers `Core::X86::Msr::IBS_DC_LINADDR` and `Core::X86::Msr::IBS_DC_PHYSADDR` might contain the linear and physical address for the tagged load or store operation. The validity depends on the flags `IbsDcLinAddrValid` and `IbsDcPhyAddrValid` given by the IBS Op Data 3 register. Besides these two addresses, there is an additional field `IbsBrTarget` for branch operations specifying the logical address for the branch target. This register has been implemented since AMD Family 12h [AMD11], and the CPUID instruction register indicates the feature support.

Most fields have existed since the first microarchitecture implementing IBS, but there have also been modifications to IBS Op since its existence. Minor changes have been implemented over several microarchitectures adding, renaming, or removing fields in the register. A few bits in the registers are marked as reserved for future usage. Therefore it is possible to add new metrics to the event type without needing a new register for IBS. The latest Zen 4 microarchitecture uses two previously reserved bits in the OP Data 2 register to extend the DataSrc field by two additional bits, increasing the total size to four bits. Increasing the existing entry size allows a more precise evaluation of load operation samples. [AMD23c]

In the AMD Family 15h generation, IBS has introduced the OP Data 4 register to store a single bit of information in the field `IbsOpLdResync`. However, AMD has decided to revert the change in the following generation by removing the register and not storing this particular information in any other register. [AMD18]

2.5. OpenMP and OMPT

OpenMP is suitable for implementing parallel applications using a shared-memory model. Due to its portability and scalability, it is possible to execute applications on platforms such as embedded systems, multicore systems, and even hardware accelerators. The standard is controlled by the architecture review board (ARB). Members are CPUs and GPU vendors, research laboratories, universities, OEMs, and other companies supporting the standard. [Ope23]

To use first-party tools with OpenMP-applications, OpenMP provides the OMPT-interface with mechanisms to initialize the tool and examine the OpenMP state of OpenMP threads. The tool can receive notifications of OpenMP events from the OpenMP application by implementing callback functions. It is also possible to monitor the activity on OpenMP target devices or to evaluate implementation-specific details of an OpenMP application.

OpenMP version 5.0 introduced the new interface OMPT support in November 2018. [Ope18] The LLVM/clang compiler version 16.0 supports most OpenMP 5.0 features, and the OMPT interfaces are considered “mostly done” according to the documentation. [Cla23] However, OMPT is still not supported on all compilers. The GCC 13 compiler still lacks the OMPT interface support and a few other OpenMP 5.0 features. [GCC23]

A tool implementing the OMPT-interface must declare and implement the initialization function `ompt_start_tool()`. This function defines a struct of the type `ompt_start_tool_result_t` and requires two functions usually called `ompt_initialize()` and `ompt_finalize`. The type definition are specified in the header file omp-tools.h and requires the availability of the OMPT-interface. These functions are called before the OpenMP-application starts and after the program has terminated. The defined struct is then returned by the `ompt_start_tool()` function. Linking the tool to an OpenMP-application works by compiling the tool and either linking it statically to the OpenMP-program or linking it to the application by specifying the environment variable `OMP_TOOL_LIBRARIES` with the path of the library. [Ope18]

2.6. MPI and PMPI

The Message Passing Interface (MPI) is a message-passing library interface specification first published in 1994 and designed for parallel and distributed programming purposes. The goal of MPI is the development of a standard for message-passing applications with a focus on portability, efficiency, and the establishment of a flexible standard for message passing. All MPI operations are defined as functions, subroutines, or methods with the appropriate programming language bindings. The bindings for C and Fortran are part of the MPI specification. [Mes21] The application runs on MPI nodes and might be distributed across multiple machines. Therefore, the communication between nodes is more complex than OpenMP and requires communication via messages. MPI provides functions for sending and receiving data. Examples are `MPI_Send`, `MPI_Recv`, and `MPI_Bcast`.

MPI has a profiling interface called PMPI for profiling tools. The MPI functions can be called with either the `MPI_` or the `PMPI_` prefix. For example, the profiling tool could implement its own `MPI_Init()` function in which it calls `PMPI_Init()`. This approach allows the tool to keep the original MPI behavior as intended and additionally perform its own tasks. The original MPI application then uses the re-implementation from the tool.

3. Description of Tools

This work is about the profiling tool Mitos and its implementation of IBS. The data collection of IBS samples uses the perf_event profiling interface of the Linux kernel. Related to Mitos is the Visualization Tool MemAxes, which uses the post-processed data from Mitos as input data. Currently, it is designed to display information about PEBS. Both tools were originally developed by Alfredo Gimenez at the Lawrence Livermore National Laboratory and are distributed under the Apache-2.0 license with the LLVM exception. [Gam15] The following work covers modified versions of both tools from the Chair of Computer Architecture and Parallel Systems at TUM. The last part of this chapter covers the systems used in this work.

3.1. The Tool Perf_Event

The tool perf began as part of the Linux subsystem for utilizing CPU performance counters. In later versions, it also supported tracepoint events, kprobes, and uprobes. This tool is also known under the names perf_events, Performance Counters for Linux (PCL), or Linux perf events (LPE). As part of the Linux kernel, perf_event is included under tools/perf. Perf_events is a lightweight profiling tool and provides many configuration options as a command line tool and via the perf_event_open() system call. Despite the deep integration into the Linux system and the importance of its functionality, the official documentation [Lin23] does not provide much information about implementation details. Some additional sources related to perf_events are Brendan Gregg's perf website [Gre23] and Vince Weaver's unofficial perf_events website [Wea23].

3.1.1. The CLI of Perf_Event

Table 3.1 shows common options that the perf_event command line tool provides. Perf_event is quite helpful for performing a Top-Down Analysis [per22, Yas14], an approach to discover software performance bottlenecks. For this purpose, the command `perf stat` collects hardware event counters and tries to identify the bottleneck type. After identifying the general bottleneck type with hardware counters, the `perf record` command takes event samples containing data related to the bottleneck type during the program run, and `perf report` analyzes which specific functions are the related bottlenecks.

perf_event command	Description
perf stat	Collect performance counter statistics
perf record	Record events for reporting and write to file perf.data
perf report	Read file perf.data and display event samples by process, function, etc.
perf annotate	Read file perf.data and annotate assembly or source code with event counts
perf top	Display live event counts
perf bench	Run different kernel microbenchmarks

Table 3.1.: Overview of common perf_event commands. Data mainly taken from: https://perf.wiki.kernel.org/index.php/Tutorial

To configure perf_event to collect the correct data, the command `perf record` has various parameters for customizing the profiling routine. The flag `-e` specifies the PMU event, and some common options are available with the command `perf list` and not vendor-specific metrics. Specifying raw events from processors or features such as IBS Fetch or IBS Op is also possible. Besides the event selection, parameters might specify recording in a per-thread, per-process, per-CPU, or system-wide monitoring mode. Also relevant for the data collection routine is the correct choice for the event sample period that is adjustable with the `-c` flag. Shorter event periods are helpful for collecting enough data in shorter program runs, but there are also disadvantages. Lower sample periods in more extended program runs lead to large record files and may not be necessary for a sufficient analysis. Additionally, the overhead from the profiling tool increases with the higher frequency of interrupt routines caused by the data collection. An overview of `perf record` parameters is available in Table 3.2. [per23a]

`perf record` Flag	Description
-a	System-wide monitoring mode
-C	Per-CPU monitoring mode and requires a list of core ids
-c	Event sample period
-e	Event selection
-p	Per-process monitoring mode with pid value
-t	Per-thread monitoring mode with tid value
-v, -vv	Verbose command line output providing errors and configuration of internal states and the `perf_event_open()` system call

Table 3.2.: Overview of perf record parameters. [per23a]

3.1.2. The perf_event_open() System Call

Before the `perf_event_open()` system call happens, it needs to know which event must be monitored, the selected sampling frequency, and many other settings. The system calls requires a parameter with type `perf_event_attr`, and the type definition is available in the header file `perf_events.h` from the perf_event tool and part of the Linux kernel. One important field in the `perf_event_attr` struct is the attribute `type` that specifies which kind of event is selected. The documentation defines following constants:

- PERF_TYPE_HARDWARE
- PERF_TYPE_SOFTWARE
- PERF_TYPE_TRACEPOINT
- PERF_TYPE_HW_CACHE
- PERF_TYPE_RAW
- PERF_TYPE_BREAKPOINT
- *Dynamic PMU*

For example, `PERF_TYPE_HARDWARE` and `PERF_TYPE_SOFTWARE` are constants from perf_event for common events available on systems regardless of the vendor. Events from the *Dynamic PMU* type are available depending on the system, and a hint about supported events gives the subdirectory `/sys/bus/event_source/devices`. This directory contains folders specifying PMU instances. [per23b]

Two other relevant parameters besides the `type` field are the attributes `sample_type` and `precise_ip`. The `sample_type` defines which information is present in the event sample. Each bit in the `sample_type` field corresponds to a specific field, and setting the bit to 1 enables the collection. Table 3.3 shows most relevant fields for the perf_event configuration in Mitos. The other attribute `precise_ip` is relevant for enabling precise sampling with PEBS or IBS. The attribute accepts values between 0 and 3. 0 is for non-precise data collection, and higher values require more precise sampling conditions.

`sample_type` Constant	Description
PERF_SAMPLE_IP	Records the instruction pointer
PERF_SAMPLE_TID	Records the pid and tid
PERF_SAMPLE_TIME	Records a timestamp
PERF_SAMPLE_ADDR	Records an address, if applicable
PERF_SAMPLE_CPU	Records CPU number
PERF_SAMPLE_RAW	Records raw data (such as IBS register fields) if applicable

Table 3.3.: Overview `sample_type` field. [per23a]

The system call also needs additional parameters for the fields `pid`, `cpu`, `group_fd`, and `flags`. Combinations of the fields `pid` and `cpu` determine the monitoring mode. The system call supports running in the per-CPU mode on a specific processor core or in a per-process mode returning event samples for a given pid. The per-CPU mode refers to the monitoring of one logical core as seen by the operating system, and in perf_event, the numbering of the core ids starts at zero. The system call returns on success an integer specifying the file descriptor, which is used for setting up mmap buffers and collecting perf_event data from the PMU of the processor. Not only memory needs to be configured, but also signal handlers that write the data from the mmap buffer to disk when a buffer overflow occurs. After everything has been configured, an `ioctl()` command starts the sampling routine, and another `ioctl()` command is used to stop the perf_event routine.

3.2. Mitos

The tool Mitos allows profiling of applications utilizing Intel PEBS. Sampling with PEBS enables perf_event to monitor applications by specifying the pid. There are two entry points to Mitos: The first one is Mitosrun and works as a command line tool that requires the executable path, configures the sampling routine, interacts profiling with Mitos, take event samples and write them to disk while the application runs and postprocesses the sample data after the monitored program has terminated. The second entry point is Mitoshooks and is specially designed for MPI-applications running on several different nodes. In this case, each node runs its own Mitos instance with preprocessing, sampling and postprocessing. Due to its strict separation between nodes, each process creates its own result folder containing hardware specifications and event sample data. This means the results from one MPI application run still require a manual merging of the data and is a feature desirable to be implemented within Mitoshooks.

Mitos has a few requirements such as CMake, OpenMP, MPI, Dyninst and hwloc. OpenMP and MPI dependencies are necessary for profiling applications requiring using one of these interfaces. Dyninst is used for matching perf_event samples and their related instruction in the source code. Finally, the hwloc dependency is used in the initial phase of Mitos for collecting information about the hardware configuration of the system.

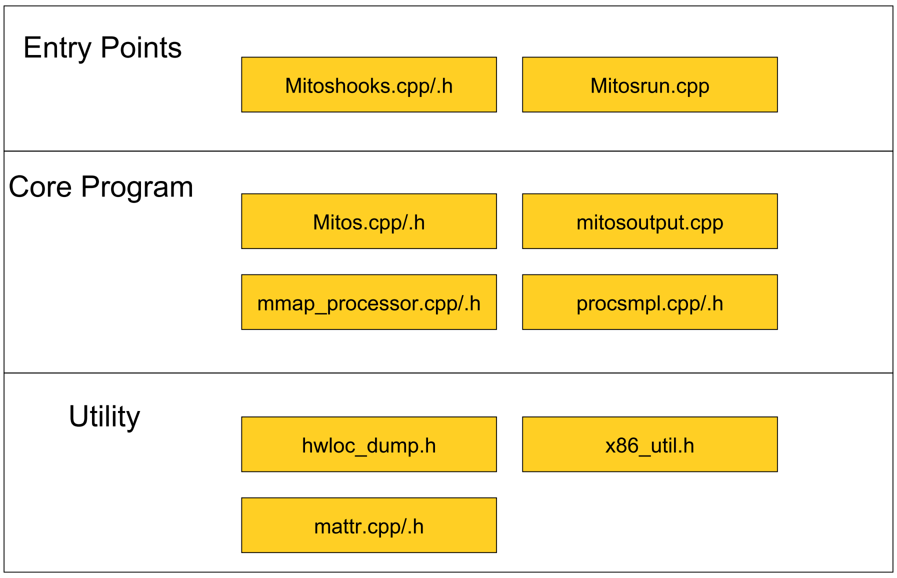


Figure 3.1.: Mitos Program Structure Overview.

Figure 3.1 shows the Mitos program structure. These files belong to either the utility functions, core program, or are entry points using Mitos. Utility files define helper functions used for interacting with the dependency hwloc in `hwloc_dump.h` or in the post-processing routine on event samples such as `mattr.cpp/.h` and `x86_util.h`.

Essential to Mitos are files from the core program. These files contain struct definitions used in the profiling tool's lifecycle, accessing the file system to read and write event sample data and hardware-related information (`mitosoutput.cpp`), functions to process raw event samples from perf_event (`mmap_processor.cpp/.h`) or provide an interface interacting with entry points (`Mitos.cpp/.h`). The core program and entry points interact in the pre- and post-process phases. The source file `Mitos.cpp` contains the implementation of most Mitos functions interacting with entry points and always start with the prefix `Mitos_`, some of these functions are I/O related Mitos functions and are implemented in `mitosoutput.cpp`. Entry points need to specify values such as the event sample period, the pid of the profiled application, how to process event samples during the profiling phase, and other settings before the profiling begins. The last pre-processing step is starting the sampling process. After the profiled application terminates, the entry point signals Mitos to stop the profiling, initiate the post-process routine evaluating the collected raw samples, and terminate the profiling tool.

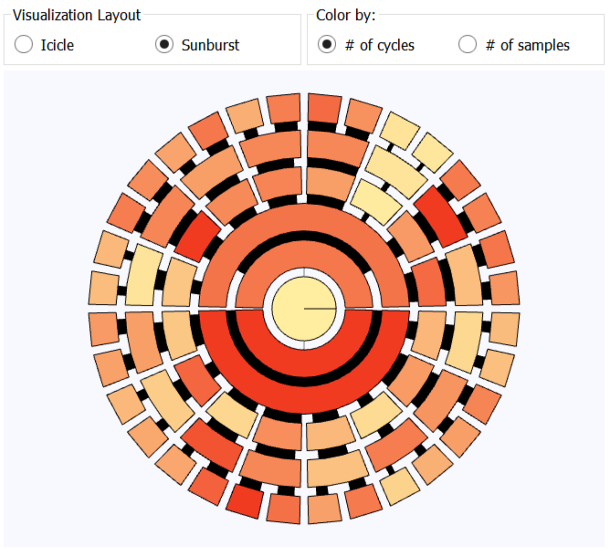
As mentioned, Mitos defines its own structures to handle event samples and its internal state. The struct definitions for event samples and data related to I/O operations are specified in `Mitos.h`. The struct `perf_event_sample` contains raw values returned by perf_event and a few other processed values that are later written to disk for usage with MemAxes. The struct `mitos_output` contains folder paths and file streams. It is first initialized in the pre-processing phase, collects hardware specification and topology with the help of hwloc, and initializes the `raw_samples.csv` file for collecting perf_event samples. During the profiling phase, mmap buffers are filled with perf_event samples until they are full, then an interrupt signals Mitos to process the data and write it to disk. After the profiled application has terminated, Mitos transforms the raw data to its final state. Each line in the raw data is copied to the actual samples output file called `samples.csv`. Additionally, the instruction pointer value is used to find the source code's actual position and instruction with the Dyninst package's help and added to the result data of a perf_event sample. So far, Mitos can only collect perf_event samples with PEBS, and one goal of this work is to implement similar routines for AMD IBS.

The essential part of Mitos is configuring the perf_event tool with the `perf_event_open()` system call, collecting and writing the samples to an output file, and finally stopping the profiling activity. The related data is stored in several different structs defined in `procsmpl.h`, the first one is called `perf_event_container` and stores information for one `perf_event_open()` system call. For Intel PEBS, there exists exactly one such struct at runtime, which collects performance data for one process on all threads of the system. The other structs `procsmpl` and `threadsmpl` store information for one process and a single thread, respectively. In the thread-related data, there is an array of `perf_event_container` which is initialized by the method call of `Mitos_init()` and configures all requirements for `perf_event_open()`.

3.3. MemAxes

The visualization tool MemAxes uses output data from Mitos and displays data from event samples, primarily information about memory accesses. This tool requires CMake, Qt5, and VTK. The latest MemAxes version from the Chair of Computer Architecture and Parallel Systems at TUM also requires the sys-sage companion tool. [GV22] Sys-sage is a library currently under development at TUM, and its purpose is to record and manipulate hardware topology information of compute systems. The library is licensed under LGPL-2.1 and available on spack or accessible via Github. [Van23]

Besides the system requirements, MemAxes also requires sampling data. The current version works with Mitos results with samples using Intel PEBS. The plan is to extend the feature set with the support of AMD IBS event samples. Figure 3.2 shows a screenshot from MemAxes and a visualization of the hardware topology in a hierarchical structure. This hierarchy structure represents existing NUMA nodes and caches, and its coloring indicates the resource usage. [GV22]



Visualization Layout: Icicle [✓ Sunburst]

Color by: [✓ # of cycles] # of samples

Figure 3.2.: Hardware Topology Vizualisation in MemAxes. [GV22]

3.4. Computation Platforms

Mitos has two entry points, Mitosrun, and Mitoshooks. On the one hand, Mitosrun is a profiling tool for general applications running on one or multiple threads. Thus, it is generally suitable for single-node applications. Mitoshooks, however, is designed to work with MPI programs and OpenMP applications. The support for MPI makes this suitable for profiling on MPI nodes and, therefore, compute clusters.

In this work, the focus lies on systems with an AMD processor. The first system is an AMD Zen 3 system and runs the evaluations in Section 4.3. It is a consumer system from the Ryzen 7-series with a modest amount of memory and eight cores with 16 threads. The second system is the `time-x` system, a single node in the CAPS cloud. CAPS cloud is a collection of servers, workstations, and other machines with highly configurable settings. Compared to the first system, it has double the amount of cores and threads but is from the previous Zen 2 generation. [CAP23] This system is relevant for testing the deployment in environments comparable to HPC systems with only limited administrative privileges. The CAPS cloud is a collection of servers, workstations, and other machines with highly configurable settings. Table 3.4 shows the hardware specifications of both systems.

Feature	Zen 3	`time-x`
Number of Nodes	1	1
Processor	AMD Ryzen 7 5800X	AMD Ryzen Threadripper PRO 3955WX 16-Cores (“Zen 2”)
Cores per Node	8	16
Threads per Core	2	2
Base Frequency [GHz]	3.8	3.9
L1-cache [KB]	512	1024
L2-cache [MB]	4	8
L3-cache [MB]	32	64
RAM per Node [GB]	32	32

Table 3.4.: Overview of the technical details of the used platforms. [CAP23]

4. IBS in Mitos

Implementing IBS requires a few considerations: The solution must run on various AMD systems, including future processors. A practical solution provides an extraction mechanism for IBS event samples, has official AMD development support, and accessible documentation. A suitable candidate is the profiling interface perf_event that is available in the Linux kernel. This work focuses on how perf_event and the system call `perf_event_open()` is used for collecting IBS data and analyzes the potentials and pitfalls of the current implementation. Furthermore, a dedicated part shows new Mitos features such as OpenMP in Mitoshooks, merging results from several folders, and writing virtual offsets to disk for the Dyninst analysis in the post-processing phase.

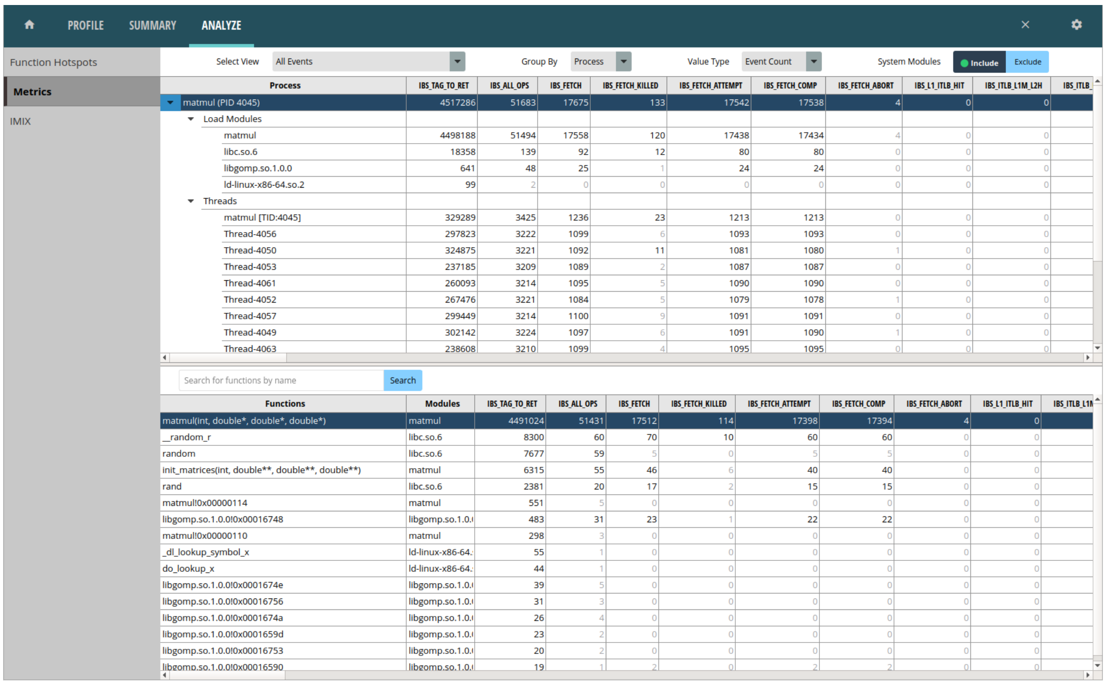
4.1. IBS Event Sample Data Collection

There are several ways to collect IBS samples, suitable approaches seem to be software profiling analysis tool AMD µProf, directly extracting data from IBS-MSRs and the profiling interface perf_event integrated into the Linux kernel since version 2.6. Essential requirements are collecting event sample data, code portability, and low code change requirements for later IBS revisions. Aggregated data is insufficient as the tool MemAxes works on the actual event sample data.

The probably most user-friendly option to experience IBS is the tool AMD µProf ("MICRO-prof") for x86 applications. There are versions available for Windows, Linux, and FreeBSD operating systems. Not only is the tool usable in a GUI without much difficulty, but it also offers CLI support and a variety of parameters to configure data collection and reporting. [AMD22] Despite its rich feature set, such as profiling analysis, system analysis, and energy analysis, it is unsuitable for Mitos and its purpose to supply event samples to the companion tool MemAxes. AMD µProf can collect raw information and save them in binary form. Still, the data is only used for generating reports aggregating samples by their respective processes, threads, load modules, functions, and instructions. There is no documentation available on how to parse µProf binary data to a Mitos-compatible samples output file.

The next option is to access the IBS-MSRs and extract the data directly. This concept is implemented in the AMD Research Instruction Based Sampling Toolkit Github repository by installing a special driver to extract IBS samples. [Gre21] The toolkit provides example programs collecting and processing IBS event samples. Also, some header files contain structure definitions specifying the correct order of the IBS attributes in the relevant registers. However, the author states it is only an experimental solution and does not recommend this approach in the long term. The author refers to the official AMD implementation in perf_event should as a long-term solution.

The probably best option is to collect IBS samples with perf_event due to its official support by AMD and the availability on modern linux systems. It allows to select either IBS Fetch or IBS Op



Select View: All Events | Group By: Process | Value Type: Event Count | System Modules: ● Include / Exclude

Process	IBS_TAG_TO_RET	IBS_ALL_OPS	IBS_FETCH	IBS_FETCH_KILLED	IBS_FETCH_ATTEMPT	IBS_FETCH_COMP	IBS_FETCH_ABORT	IBS_L1_ITLB_HIT	IBS_ITLB_L1M_L2H	IBS_ITLB_
matmul (PID 4045)	4517286	51683	17675	133	17542	17538	4	0	0	
Load Modules										
matmul	4498188	51494	17558	120	17438	17434	4	0	0	
libc.so.6	18358	139	92	12	80	80	0	0	0	
libgomp.so.1.0.0	641	48	25	1	24	24	0	0	0	
ld-linux-x86-64.so.2	99	2	0	0	0	0	0	0	0	
Threads										
matmul [TID:4045]	329289	3425	1236	23	1213	1213	0	0	0	
Thread-4056	297823	3222	1099	6	1093	1093	0	0	0	
Thread-4050	324875	3221	1092	11	1081	1080	1	0	0	
Thread-4053	237185	3209	1089	2	1087	1087	0	0	0	
Thread-4061	260093	3214	1095	5	1090	1090	0	0	0	
Thread-4052	267476	3221	1084	5	1079	1078	1	0	0	
Thread-4057	299449	3214	1100	9	1091	1091	0	0	0	
Thread-4049	302142	3224	1097	6	1091	1090	1	0	0	
Thread-4063	238608	3210	1099	4	1095	1095	0	0	0	

Functions	Modules	IBS_TAG_TO_RET	IBS_ALL_OPS	IBS_FETCH	IBS_FETCH_KILLED	IBS_FETCH_ATTEMPT	IBS_FETCH_COMP	IBS_FETCH_ABORT	IBS_L1_ITLB_HIT	IBS_ITLB_L1M
matmul(int, double*, double*, double*)	matmul	4491024	51431	17512	114	17398	17394	4	0	
__random_r	libc.so.6	8300	60	70	10	60	60	0	0	
random	libc.so.6	7677	59	5	0	5	5	0	0	
init_matrices(int, double**, double**, double**)	matmul	6315	55	46	6	40	40	0	0	
rand	libc.so.6	2381	20	17	2	15	15	0	0	
matmul!0x00000114	matmul	551	5	0	0	0	0	0	0	
libgomp.so.1.0.0!0x00016748	libgomp.so.1.0.(	483	31	23	1	22	22	0	0	
matmul!0x00000110	matmul	298	3	0	0	0	0	0	0	
_dl_lookup_symbol_x	ld-linux-x86-64.:	55	1	0	0	0	0	0	0	
do_lookup_x	ld-linux-x86-64.:	44	1	0	0	0	0	0	0	
libgomp.so.1.0.0!0x0001674e	libgomp.so.1.0.(	39	5	0	0	0	0	0	0	
libgomp.so.1.0.0!0x00016756	libgomp.so.1.0.(	31	3	0	0	0	0	0	0	
libgomp.so.1.0.0!0x0001674a	libgomp.so.1.0.(	26	4	0	0	0	0	0	0	
libgomp.so.1.0.0!0x0001659d	libgomp.so.1.0.(	23	2	0	0	0	0	0	0	
libgomp.so.1.0.0!0x00016753	libgomp.so.1.0.(	20	2	0	0	0	0	0	0	
libgomp.so.1.0.0!0x00016590	libgomp.so.1.0.(	19	1	2	0	2	2	0	0	

Figure 4.1.: AMD µProf Analysis User Interface.

in the configuration, execute the `perf_event_open()` system call, configure buffers and interrupt handlers, collect all event samples and in the end save them in a similar format as the PEBS samples in Mitos. Perf_event provides IBS data in a raw sample field, which means IBS attributes need to be correctly extracted from the byte array and require knowledge about the structure of a raw sample.

The AMD IBS Toolkit, AMD Processor Programming Reference (PPR), and files in the Linux kernel under `/arch/x86/include/asm/amd-ibs.h` provide more information about the raw sample structure. [Gre21, AMD23c] Additional information about the register order is not documented by perf_event, but the related information is defined in the Linux kernel. Two important files are the MSR type definitions `/arch/x86/include/asm/msr-index.h` and the `perf_ibs_handle_irq()` method for the actual event sample generation under `/arch/x86/events/amd/ibs.c`. The IBS collection might change in future Linux kernel versions and could require the adjustment of the data extraction, e.g., changing the attribute and register order, or adding entirely new IBS registers in future microarchitectures might require additional event sample parsing logic.

There are some downsides related to perf_event. First, collecting by selecting a process is not possible with IBS. The only other option is to specify which logical core the monitoring should happen and, thus, use per-CPU monitoring. Second, configuring monitoring on all cores is possible, but it increases the overhead of listening to all events. Also, this approach is unsuitable for monitoring single-thread applications due to its overhead and takes away resources from potential other applications on the same system running on different cores. Third, The perf_event data collection provides all event samples on the configured core. This behavior is practical for cases where the complete system needs to be monitored, but in most cases, only event samples related to the profiled application are relevant. Implementing event sample filtering on a higher level increases the overhead because these samples are first stored in the mmap buffer, triggering interrupts more often and using additional resources when processing the sample on a layer above perf_event.

4.1.1. Perf_Event Implementation with IBS

Resources about `perf_event_open()` and its usage with IBS are scarce. One of the few detailed guides is available in the Github Repository of the AMD IBS Research Toolkit in the file `ibs_with_perf _events.txt`. [Gre21] This document shows how to use and configure IBS data collection with `perf_event_open()`. Before perf_event is ready to use IBS, there are multiple steps necessary involving the system capabilities, system call `perf_event_open()`, mmap buffer, signal handler, and the system call `ioctl()` to start the monitoring. Regarding system capabilities, IBS is only available on AMD processors since family 10h and in a per-CPU or system-wide monitoring mode in perf_event. These accesses require special permissions and can be configured in the file `/proc/sys/kernel/perf_event_paranoid` with a value of 0 or lower for non-root users.

The first step is the initialization of the `perf_event_attr`, which is included in the header file `linux/perf_event.h`. The `type` field contains an entry of the *Dynamic PMU* category, and available PMU instances have an entry in the subdirectory `/sys/bus/event_source/devices`. [per23b] Entries for IBS Fetch and IBS Op are required to use IBS with perf_event. The correct values for the `type` field are specified in the files at path `/sys/bus/event_source/devices/ibs_fetch/type` and `/sys/bus/event_source/devices/ibs_op/type`. Experimental results on an AMD Zen 3 system returned the value 8 for IBS Fetch and 9 for IBS Op. This result does not correspond with the information provided by the AMD IBS Research Toolkit [Gre21], the `type` value for both IBS events are swapped in the documentation about perf_event with IBS and very likely incorrect. Another relevant attribute is the `precise_ip` field of `perf_event_attr`, and it specifies a perf_event setting regarding the skid problem. The AMD IBS Research Toolkit specifies a skid of 1 [Gre21] and requests for a constant skid according to the Linux manual. [per23b] The Mitos implementation uses the second strictest setting of 2 and requests a skid of 0.

In the second step, the `perf_event_open()` initializes a file descriptor later used for configuring the mmap buffer and the data collection. Besides the `perf_event_attr` struct, the two parameters `pid` and `cpu` require suitable values. Unfortunately, the IBS implementation in perf_event only supports per-CPU or system-wide monitoring. Therefore, Mitos chooses the values `pid = -1` and for `cpu` a value greater equal to 0 to configure per-CPU monitoring. A successful system call returns a file descriptor with a positive value and -1 in cases with problems with the configuration.

The third step continues after the successful system call and configures the mmap buffer. This buffer will fill up with perf_event samples while profiling until an interrupt triggers the data collection. Therefore, the program must configure an interrupt handler routine to clear the buffer. In the Mitos tool, the interrupt handler routine writes event samples from the buffer to the user space and transfers these raw samples regularly to the file `raw_samples.csv` in the filesystem.

The final step is activating the file descriptor to monitor a specific core. Two system calls with `ioctl()` are necessary, the first is a reset command, and the second enables the data collection.

4.2. Mitos Implementation Changes

Implementing IBS results in significant changes to the core parts of Mitos, it especially alters the pre-processing and post-processing implementation. It also leads to some minor changes in the entry points Mitosrun and Mitoshooks and adjustments to the event samples' struct definitions and I/O-related operations. Besides the IBS changes, Mitoshooks also has new routines related to OMP and its OMPT interface. Also relevant is the new file `virtual_address_writer.h` which writes an virtual address offset from a dynamically loaded object to disk. This functionality is relevant for the Dyninst and event sample post-processing.

Figure 4.2 shows file changes in red and yellow depending on the degree of change, the green highlighted files do not have any modifications. The two files in blue are newly created files related to implementation changes mentioned in this section.

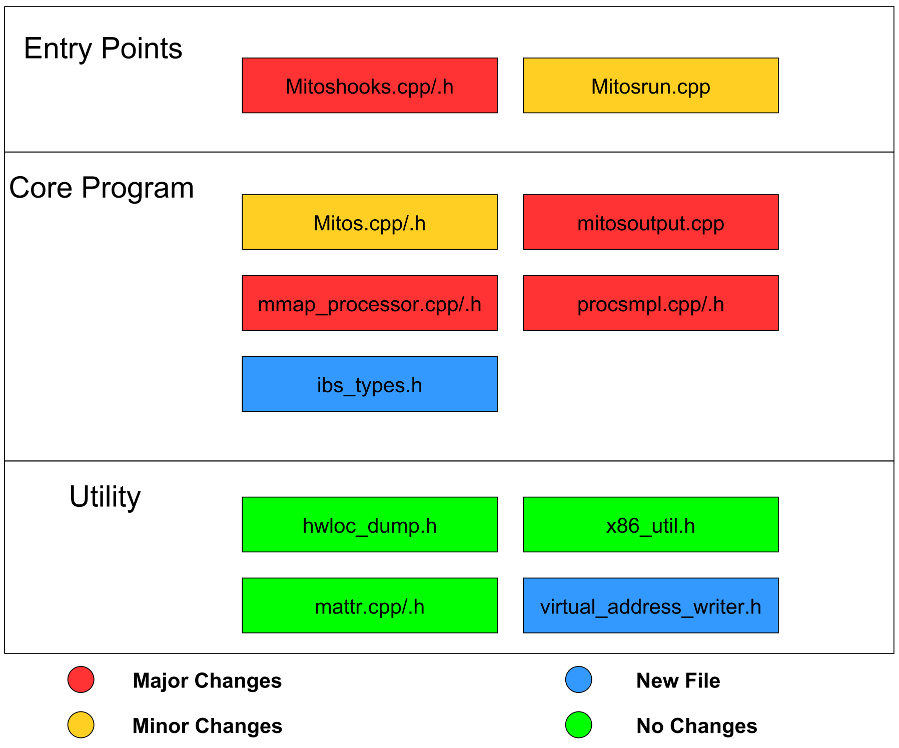


Figure 4.2.: Mitos Changes Overview.

4.2.1. IBS Implementation

The IBS implementation has a few key aspects it needs to handle. First, the IBS values need to be extracted from perf_event samples. This step is not trivial due to the necessary action to read the IBS register information from the `raw_data` field and the required knowledge about the correct register order in the raw data. Another problem is the lacking capability of IBS to monitor per-process. There are two potential solutions implemented in this work. The first solution is to monitor all processor cores. The second solution involves a tracking routine that knows on which logical core the program runs and changes the monitoring if necessary. Also relevant for the IBS implementation are changes to the data collection routine in `mitosoutput.h` and changes to the build system and the CMake configuration.

IBS Type Definition

The manual AMD PPR and the AMD IBS Reasearch Toolkit repository provide information about the correct order of registers [AMD23c, Gre21], this information is used in `mmap_processor.cpp` to read the correct values from the `raw_data` field of the event samples. The size of the `raw_data` field is specified in the preceding `raw_size` field. On Zen 3 systems and most other AMD microarchitectures with IBS, the raw data has a size of 36 bytes for IBS Fetch samples. IBS Fetch samples provide information about four registers with a length of 8 bytes each. The remaining padding of 4 bytes at the start of the raw data field is necessary to comply with the 8-byte alignment in the `perf_event_sample` struct. The sum of the `raw_size` and `raw_data` is 40 bytes and 72 bytes for IBS Fetch and IBS Op, respectively. The padding also exists for IBS Op samples. IBS Op has eight samples of 8 bytes each. Figure 4.3 and Figure 4.4 show the register order in the raw data field for both IBS types. So far, only the processors from family 15h implemented an additional register in IBS OP, the IBS Op data 4 field, and is therefore an exception to the mapping above. [AMD18]

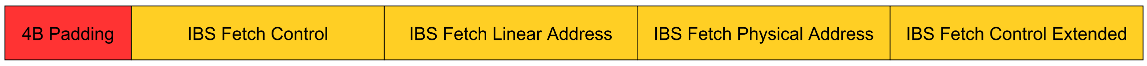


Figure 4.3.: IBS Fetch register overview with a total size of 36 bytes.

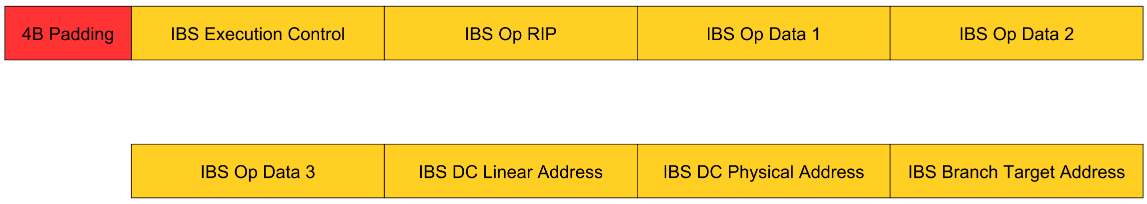


Figure 4.4.: IBS Op register overview with a total size of 68 bytes.

Most of the registers are available since the introduction of IBS, the two exceptions are the IBS branch target address register with its first implementation in family 12h and IBS Fetch Control Extended register in family 15h. [AMD11, AMD12, AMD18] The meaning of individual fields in a register, the number of register, or the order might change depending on future IBS revisions.

There are variables for each register in the modified struct `perf_event_sample` of the Mitos project in `Mitos.h`. The register type definitions from `ibs_types.h` allow simpler access to individual fields. This file originates from the IBS Research Toolkit and reproduces the same structure as denoted in AMD manuals describing the IBS specification. [Gre21]

Monitoring all Cores

This approach replicates the Intel PEBS solution, which allows receiving event samples related to a certain process from any core it runs on. That means that IBS in perf_event requires one system call per logical core and increased complexity of Mitos compared to per-process monitoring. It must create more mmap buffers, run more system calls, initialize signal handlers for interrupts, etc. This approach is only available with Mitosrun, the other tool, Mitoshooks has a slightly different monitoring configuration with multiple Mitos instances for individual MPI nodes and OpenMP threads.

Furthermore, the thread performing the perf_event_open()-system call and enabling the data collection is also responsible for writing the event samples from the kernel buffer to disk. With monitoring of all cores in one single thread, the overhead distribution onto all available cores could be better. An improvement could be distributing the monitoring workload onto multiple threads by creating new threads in Mitosrun. Still, the increased complexity of Mitos may not outweigh the overhead reduction for the main thread of the profiling tool.

The Mitos option to monitor all the cores initializes the `perf_event_open()` system call in the methods `procsmpl::init()` and `threadsmpl::init()` of `procsmpl.cpp`. On success, the configuration initializes the mmap buffer for each core and the corresponding signal handler routines and is ready to enable IBS monitoring. The method `Mitos_begin_sampler()` starts monitoring all logical cores and is called from by Mitosrun or Mitoshooks. During the profiling, kernel buffer overflows lead to interrupts that are handled in the method `threadhandler()`. The method collects IBS samples and writes them to a vector containing events. In regular intervals, the vector entries are written to disk and then deleted from the vector. The termination of the profiling is implemented in the `Mitos_end_sampler()` method, which finally turns off the performance monitoring on all cores.

Migrating Core Monitoring

Another approach with potentially less resource overhead is migrating the monitoring from one core to another. Mitosrun and Mitoshooks support this approach. Mitosrun only monitors a single thread in this case. The Mitoshook program can run one Mitos instance per MPI node or OpenMP thread. Therefore it should be able to monitor all cores with related computations of the profiled application. This idea seems promising for profiling applications with low thread count on hardware with many available cores. Processors with high core counts are not unlikely to be used in HPC systems. In these environments, it is more common to see workstation and server processors with a higher core count and a lower clock frequency than consumer processors.

However, migrating the monitoring core is a costly operation. First, the sampling routine needs to regularly check on which core the profiled application is running and detect core changes. If the core changes, the sample collection for the previous core needs to be stopped, and the initialization routine needs to be performed for the new core.

The migrating core option also initializes the `perf_event_attr` struct and other Mitos control variables but does not prepare any mmap buffers and signal handler routines in advance. These steps are delayed until the Mitos knows which core should be monitored, therefore the related methods `procsmpl::enable_event()` and `procsmpl::disable_event()` are responsible for performing the system call, setting up the buffer, and starting the monitoring or disable the monitoring routine for a specific core.

The core migration option calls the `update_sampling_events()` method to determine where the profiled application runs and then start the perf_event monitoring by using the `pref_event_open()` system call and other related configuration steps. The signal handler routine regularly writes the event samples from the buffer to the user space and checks where the current thread runs.

Suppose the profiled thread has moved to another logical core, the current monitoring needs to be disabled, and a new performance monitoring sampler needs to be configured. Not only is the reconfiguration of the data collection slow, but the routine for determining the core location for a certain pid is also hindered by its current implementation. In Mitoshook routines, the profiled application and the Mitos monitoring routine usually run on the same thread. They can determine with the command `sched_cpu()` the logical core in a reasonable amount of core. Unfortunately, Mitosrun is different and runs the profiled application and the Mitos routines in separate processes. Therefore, it needs to determine the necessary information with an expensive function processing the command `ps -o psr <target pid>` in the command line, reading the result, and returning the core id after parsing the result from an output stream.

Mitosoutput

Writing the event samples back to disk happens in three phases. The first phase is the pre-processing routine creating the result folders, collecting hardware topology information, and creating a file called `raw_samples.csv`. This file fills up in the second phase and contains data from the perf_event data collection. The `Mitos_write_sample()` method in `mitosoutput.cpp` writes all information of one event sample to this file. One line in the file contains the information of one sample. The first half contains collected perf_event attributes except for information from the `raw_data` field. The second half consists of IBS fields, either IBS Fetch or IBS Op information is written to the `raw_samples.csv` file. The third phase is the post-processing phase, which transforms the file `raw_samples.csv` to `samples.csv` by adding a header containing the column information and additional columns with results from Dyninst analyzing instruction pointers and the executable file. These results might contain the position in the source code or the assembly instruction.

4.2.2. CMake Changes

New CMake variables were introduced to implement IBS and its different monitoring options. The first new variable specifies the collected IBS event type, valid options are IBS Fetch, IBS Op, or off. The last option turns off all IBS code routines and switches to the Intel PEBS implementation of Mitos. There is no implementation for collecting both event types in one program run because of the problem of generating samples with partially missing data. IBS Fetch samples do not have valid data in IBS Op fields; the same applies vice-versa. The second variable is a switch between the collection modes. It allows monitoring on all processor cores or only one thread with the ability to change the logical core it monitors. The third variable is a switch for the OMPT interface, which is currently only supported for the LLVM-based Clang and Intel compilers.

Compiling with GCC causes errors due to the missing `omp-tools.h` header file.

4.2.3. Mitosrun

The application here only has smaller changes. Mitosrun uses the same functions with IBS as with PEBS. The only change is the `Mitos_set_pid()` method for enabling the event sample filtering to only collect samples related to the profiled application. This problem is irrelevant in PEBS due to its ability to configure the filtering in the `perf_event_open()` system call itself and provide only relevant samples by itself.

4.2.4. Mitoshooks

Like Mitosrun, the Mitoshooks core functionality is the same for IBS and PEBS and has a new `Mitos_set_pid()` function call to implement IBS event sample filtering. The entry point Mitoshooks has two different ways to be used: The first is the usage together with an MPI application and overrides the original `MPI_init()` and `MPI_finalize()` functions for implementing the start and end routine of Mitos profiling. In these cases, the MPI application includes the header file `mitoshooks.h` and links the libraries in the compilation process. The second use case is using the OMPT interface of OpenMP to implement the callback functions for the start and termination of an OpenMP thread with Mitos code. The usage is a little different with OpenMP and is explained in Section 2.5. There are several ways to link first-party tools with OMPT to the OpenMP application. It is to note that MPI and OpenMP in one application are not supported by Mitos. Further information is covered in Subsection 4.5.5.

Mitoshook data collection might create multiple folders. It depends on the count of MPI processes and OpenMP threads of the profiled application. For these cases, the new `Mitos_merge_files()` function takes arguments for the prefix of folders to merge and a more specific prefix to identify the folder belonging to the MPI process with rank 0 zero or the first created OpenMP thread. The identified folder is treated as the base for the new result folder, which then contains all files, such as sample data, hardware specification, and topology information. At the code section with the merge file function call, it is not known how the complete folder is named due to the way that `Mitos_create_output()` appends a timestamp to every output folder. For all other folders matching with the first prefix, only the sample data is appended to the file in the new result folder. Hardware specifications are currently discarded from other Mitos results in order to avoid duplicate hardware information. However, MPI might run on nodes with different hardware specifications, and it would be helpful to keep all different hardware specifications. This feature is currently not implemented.

4.3. Evaluation

This first part focuses on the correctness of the data Mitos with IBS collects, which means it checks whether the correct values have been extracted from the perf_event sample fields. One possibility to check is by using a synthetic program with code sections with known performance properties, such as the possibility of having cache misses, branch mispredictions, and other metrics to correlate between code and event samples. Another concern is the behavior of the two IBS collection methods in Mitos. The second part compares both implementations and looks at the quality of their data collection and the overhead compared to the program without the use of profiling tools.

4.3.1. Data Collection Validation

The following scenario checks whether perf_event and IBS deliver plausible values in a matrix multiplication program. The first half of the program is a matrix multiplication with the size 32×32 for 100,000 iterations running on logical core 3. The second half switches to logical core 7 and runs only two iterations of matrix multiplication with the dimension 1024×1024. The location separation is due to the simple sample identification and allocation to certain code sections. A suitable Mitos entry point is Mitosrun for this scenario because of the utilization of only one thread in the application. One metric indicating the program performance is the field `IbsTagToRetCtr` available with IBS OP. This metric measures the number of cycles a macro-operation spends since it was tagged until it retires. Cache misses increase the time until the macro-operation can retire.

Working on smaller memory areas produces fewer cache misses than a program working on a much larger dataset. Figure 4.5 displays the time axis on the x-axis and the number of cycles the operation takes to move from the tagged state to the retired state on the y-axis. One dot represent one sample and the green dots belonging to the first execution half clearly show an trend of overall lower values.

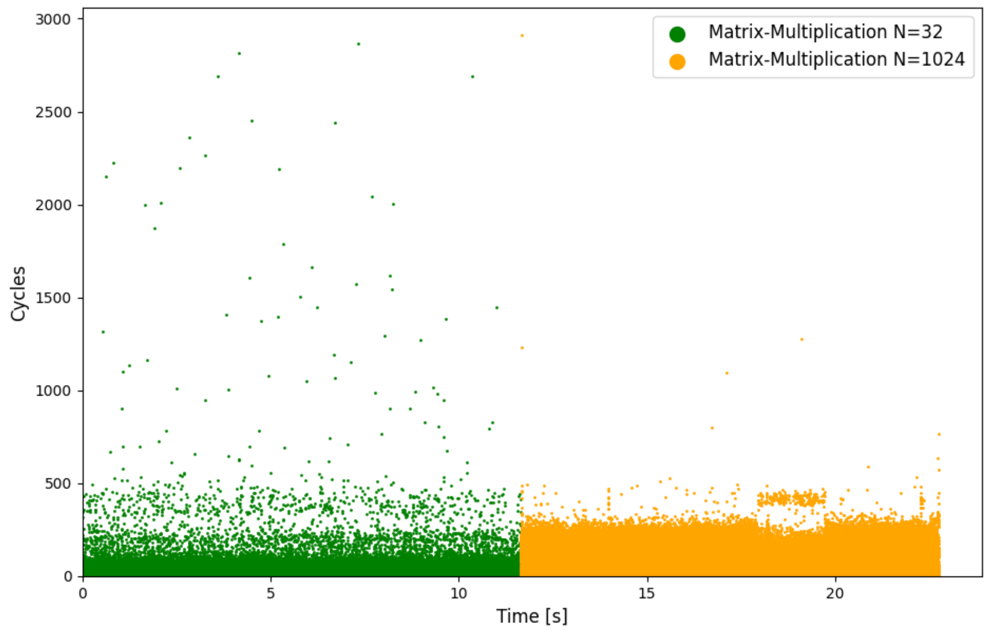


Figure 4.5.: Number of Tagged to Retired Cycles for Event Samples in Matrix Multiplication Program 1.

Besides the `IbsTagToRetCtr` field, other metrics related to cache misses such as flags and latency fields might proof the correlation between the program and the collected IBS samples. Table 4.1 confirms the expectations regarding the cache miss behavior.

IBS Op Metric	Matrix Multiplication N = 32	Matrix Multiplication N = 1024
Tagged To Retired [cycles]	55.28	102.94
Completed To Retired [cycles]	34.63	63.53
Occurence of Data Cache Miss [%]	0.43	3.60
L2 Cache Miss [%]	0.0255	0.95

Table 4.1.: Average Values of IBS Op Fields.

4.3.2. Data Collection Method Evaluation

This work implements two different IBS data collection modes, one with system-wide all-core monitoring and the other with a core migration feature. For this comparison, another matrix multiplication program runs on the Zen 3 system, and the average runtime is calculated across ten runs. The reference application without active sampling routines has an average execution time of 9.12 seconds. The application does one core migration in the program run. This should be a slight advantage to the core migration solution due to the reduction of potential core migrations during the execution and allows for a simple reproduction of these results.

Figure 4.6 shows the average execution times for both solutions, and the core migration solution seems to be faster than the all-core monitoring. Also, comparing the number of collected samples shows differences in the range of under 10 percent. This discrepancy means the second solution is promising for sample applications with relatively few core migrations.

In a different matrix multiplication program with an average execution runtime of around 5 seconds, the program switches processor cores about every 0.5 seconds. Occurring core migrations are no issue for the all-core solution because of the capability to process event samples from any processor core. The second solution using core migration after a change is detected does not work well. In this setup, the first solution collects, on average, 100MB of IBS data in 5 to 6 seconds. The second solution can only collect between 15MB and 25MB of event sample data, a sample loss of at least 75 percent is quite significant. Therefore, the second solution works well for cases in which threads rarely migrate to another processor core and has problems detecting these migrations efficiently. Additional measurements on the method `update_sampling_events()` show that the execution without changing the processor core takes around 200 microseconds. In cases with core migration, the core migration takes up to 250 microseconds. Further analysis reveals that the method `get_psr()`, a function providing the running processor core for a given pid and executing a command line command, uses a significant amount of time. The same code without this method takes, on average, about 1 microsecond. Thus, improving this core migration solution depends on finding a better way to detect core changes in other processes.

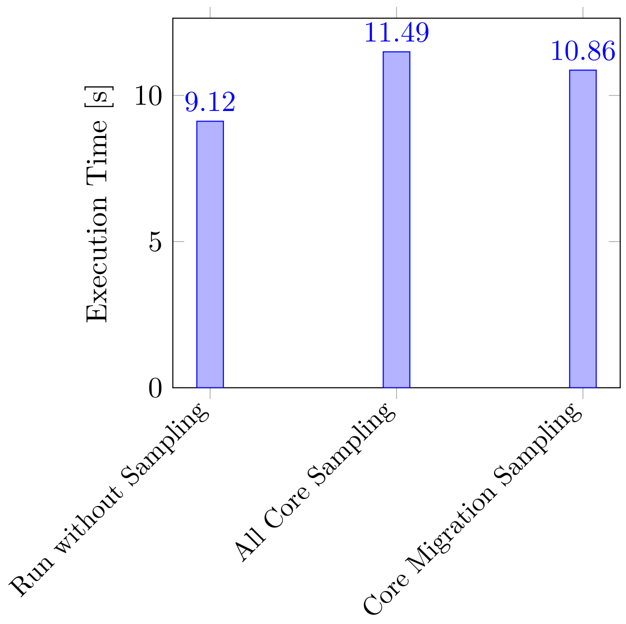


Figure 4.6.: Average Execution Times of Matrix Multiplication Program 2.

4.4. IBS Limitations and Challenges

The current support by perf_event can collect IBS data for a variety of application types. Unfortunately, some aspects of the data collection methods make monitoring with IBS in perf_event slightly more complicated. These aspects may be limited by the implementation of IBS in the Linux kernel or are related to the implementation in Mitos.

4.4.1. System-wide Monitoring

Another concern is the system-wide monitoring with perf_event and IBS because it is not possible to specify a process by its pid as a monitoring target. This leads to the requirement of a parameter change in the `/proc/sys/kernel/perf_event_paranoid` file to -1. This step requires administrative priviliges and could be a problem in a workstation or HPC environment if it is not configured in the system by someone with these permissions. Additionally, filtering event samples on a higher level compared to perf_event increases the program complexity and consumes more resources by processing unrelated samples more often than actually necessary. Again, this problem is an issue caused by limited configuration option in perf_events and could be fixed in a future version of the tool.

4.4.2. Core Migration Overhead

This is a Mitos specific issue that is related to how data collection for multiple threads is handled. It regularly checks where on which core the thread runs and migrates the perf_event monitoring if a change is detected. Migrating between cores is not cheap, it requires to disable the current monitoring and redo the initialization including the perf_event_open() system call. This is a quite complex workaround to make sure only the necessary cores are tracked and not necessary if process-specific monitoring would be available for IBS in perf_event.

4.5. Future Considerations for Mitos

4.5.1. Simultaneous Data Collection of IBS Fetch and IBS Op

The current Mitos implementation is not ready for recording both IBS event types simultaneously. Mitos needs to use double the amount of active performance monitoring routines to record both types. Therefore, this implementation change increases the number of system calls, mmap buffers, and other resource usages at runtime. But even after the profiling ends, the result is one larger result file with mixed IBS event types. Mixed event types lead to missing column entries in each line. Tools evaluating these results must expect missing metrics and process them correctly.

4.5.2. Issues with Dyninst and OpenMP

For Mitosrun and Mitoshooks, event samples contain an instruction pointer field, which can be used to identify the line of code that triggered this instruction pointer. By using precise sampling such as IBS and PEBS, the generated event sample data can be matched to the correct instruction or, in some cases, very closely to the related instruction. This relative offset is unique to each executable and needs to be acquired before the instruction parsing can happen. This functionality is either integrated by the profiled application or already integrated into Mitoshooks and calls the function `write_virtual_offset()` as part of a pre-processing routine in the header file `virtual_offset_writer.h`.

Dyninst and OpenMP in Mitoshooks do not work with the linking of the Mitoshook-library with the environment variable OMP_TOOL_LIBRARIES. In the Mitos post-processing routine, Dyninst opens a given executable file to match instruction pointer values with the correct instruction and additional metadata. The file opening does not work when a valid executable is given as a function argument. In this case, the file opening method never completes and seems stuck in an infinite loop. This behavior was observed with Dyninst version 12.3.0. Therefore, the current Mitoshooks implementation gives a non-existent executable path, an empty string, as input to jump to an error-handling routine, preventing being stuck in the file opening method indefinitely. One cause could be how Mitoshooks is linked to the application, and perhaps other linking mechanisms work better for OMPT.

4.5.3. Additional IBS Zen 4 Extensions

Some features were introduced to IBS in Zen 4, already mentioned in Subsection 2.4.2 is the extension of the `DataSrc`-field denoting values for local or shared caches, DRAM, and other targets. [AMD23c] Not yet mentioned are two filter options for both IBS event types that allow the collection of samples with an occurring L3 cache miss. [Lar22a] This filter option is available since Linux kernel version 5.19 and the `DataSrc`-extension requires at least version 6.0. [Lar22a, Lar22b]

```
$ perf record -a -e ibs_op/l3missonly=1/ --raw-samples sleep 5
```

Listing 4.1: Perf Command with L3Miss Filter and IBS

4.5.4. Result Aggregation in Multi-Thread Applications

This is only an issue related to Mitoshooks, where multiple folders are created belonging to one MPI process or one OpenMP thread. The current implementation copies all event samples to one file and keeps the hardware information about the primary process/thread. This implementation aspect might be an issue for MPI because the hardware in a regular OpenMP application runs on the same system. MPI is designed to run in a distributed system on different nodes, where hardware information is much more interesting due to the potential difference in hardware specifications of the nodes. One solution would be to compare the hardware files and filter duplicate hardware specification files. This approach seems straightforward and requires no additional configuration by the user. Another approach is the introduction of a configuration option either in CMake or as some kind of command line parameter which toggles between collecting only the hardware file of the primary file or all hardware specifications. This approach should work if the user knows about this parameter, and it should be manageable to specify when the application runs on multiple nodes.

4.5.5. MPI + OpenMP Applications

Previous parts of this work covered programs running on a single thread, using OpenMP, or running with MPi. However, in some cases, having more fine-grained control of using multiple threads in an MPI process is desirable. Using OpenMP and MPI in one application is possible, but Mitos is not prepared to handle this case. No special mechanisms are implemented for this scenario, and the compatibility has not been tested so far. There exist Mitos routines for both cases and using the MPI hooks for these scenarios seems insufficient because it would only monitor one thread per MPI process, although OpenMP might spawn more threads. Thus, it would make more sense to collect data with Mitoshooks and its OMPT interface in this specific scenario.

The Mitoshooks routine in its current form would probably correctly merge results from the OpenMP threads belonging to the same MPI process as long as the MPI part of Mitoshooks is disabled. Still, the next step to merge the aggregated MPI process folders is missing and requires some considerations on when to activate this routine and how to identify the MPI_processes after OpenMP has aggregated the data. The identification of this particular edge case isn't easy to implement. Therefore it might make sense to manually identify the folders by hand.

4.5.6. Per-Process Monitoring with Perf_Event

AMD might implement a per-process monitoring routine like Intel PEBS has done in perf_event. In this case, the option with core migration should probably be removed due to feature redundancy. Per-process monitoring also helps to remove the event sample filtering for IBS on a higher level. Perf_event would automatically handle this issue by only providing samples for the requested process. A per-process monitoring solution must follow the same steps as the current Intel PEBS implementation. The only deviation in the pre-processing phase is initializing the struct `perf_event_open()` system call with IBS-specific settings. In the profiling phase, there are still significant differences in how Intel PEBS and IBS provide data in the event samples and therefore need to keep the current implementation in `mmap_processor.cpp` and the related struct `perf_event_sample`. Due to the differences in the event sample output, the post-processing routine must handle different sets of columns in the sample output file.

5. Conclusion

5.1. Summary

Instruction-Based sampling has its advantages in providing various performance metrics in one event sample, solutions such as Intel PEBS cannot compare with the quantity of collected data. With IBS Fetch and IBS OP, there are two different event types focusing on instructions at the front-end or on macro-operations executed in the back-end. This combination of available information allows for extensive analysis of profiled applications.

Implementing IBS in perf_event brings significant usability improvements compared to attempts to directly access the IBS-specific registers. Perf_event is not only a command line tool but also provides the system call `perf_event_open()` with various configuration options ranging from event selection, monitoring modes for tracking per-process or system-wide on a range of cores to the specification of the requested fields in the event samples. However, there is still room for improvement for the profiling interface and its usage with IBS. It is not very clearly documented how IBS raw samples are stored in the `perf_event_sample` structure, there might be better ways to inform about a 4 byte padding at the beginning of the `raw_data` field than to look at the source code of the Linux kernel. Also, the capability to monitor in all existing monitoring modes is necessary for various use cases. System-wide monitoring may be fine in environments where administrative privileges are available and the system only used by few people. However, the lack of per-process monitoring increases the complexity to monitor applications running on systems with many processor cores, which is already reality with the latest workstation and HPC processors. In June, new AMD processors has just announced new server processors with 128 cores. [Bha23] Efficient and information-rich profiling solutions should be considered when choosing a new compute platform , IBS has definitely the potential to fulllfil these goals.

The profiling tool Mitos with its IBS implementation is now usable for Intel and AMD processors despite its difference in data collection mechanism. New functionality in Mitoshooks allows it to use the OMPT interface introduced in OpenMP 5.0 to configure thread-specific monitoring routines similar to its MPI implementation. Another feature in Mitoshoooks is merging Mitos results from multiple folders to a single result folder.

5.2. Outlook

The latest AMD Zen 4 processors still offer new features not implemented in this work, such as L3 cache miss filtering and accessing extended data source information. Another Linux kernel update with per-process monitoring would significantly improve IBS data collection even further.

The development of Mitos might lead to improvements in other areas of the profiling toolchain.

The related visualization tool MemAxes is currently without IBS event sample support. Implementing the compatibility to IBS Fetch and IBS Op samples would make sense. Also, further improvements in the data post-processing could offer new possibilities. The implementation still lacks support for exotic programs such as combined MPI-OpenMP hybrid programs. Also missing is full compatibility with Dyninst and the OpenMP Mitoshooks functions.

Part I.

Appendix

A. Abbreviations

A.1. General Terminology

BTB Branch Target Buffer

HPC High-Performance-Computing

pid Process-ID

ROB Reorder Buffer

PMC Performance Monitoring Counters

PMU Performance Monitoring Unit

tid Thread-ID

TLB Translation Lookaside Buffer

A.2. AMD Terminology

BKDG AMD's BIOS and Kernel Developer's Guide

IBS Instruction-Based sampling

PPR AMD's Processor Programming Reference

A.3. Intel Terminology

BTS Branch Trace Store Records

DS Debug Store

MSR Model-Specific Register

PEBS Processor Event-Based Sampling

List of Figures

List of Tables

Bibliography

[AMD10] AMD. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*, april 2010 edition, 2010. [Online; accessed 20-June-2023].

[AMD11] AMD. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 12h Processors*, october 2011 edition, 2011. [Online; accessed 04-July-2023].

[AMD12] AMD. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 14h Models 00h-0Fh Processors*, february 2012 edition, 2012. [Online; accessed 16-July-2023].

[AMD16] AMD. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 16h Models 30h-3Fh Processors*, march 2016 edition, 2016. [Online; accessed 04-July-2023].

[AMD18] AMD. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 70h-7Fh Processors*, june 2018 edition, 2018. [Online; accessed 04-July-2023].

[AMD20] AMD. *Preliminary Processor Programming Reference (PPR) for AMD Family 17h Model 31h, Revision B0 Processors*, september 2020 edition, 2020. [Online; accessed 04-July-2023].

[AMD21] AMD. *Processor Programming Reference (PPR) for AMD Family 19h Model 51h, Revision A1 Processors*, september 2021 edition, 2021. [Online; accessed 04-July-2023].

[AMD22] AMD. *AMD uProf User Guide*, november 2022 edition, 2022. [Online; accessed 09-July-2023].

[AMD23a] AMD. *Software Optimization Guide for the AMD Zen4 Microarchitecture*, january 2023 edition, 2023. [Online; accessed 20-June-2023].

[AMD23b] AMD. *AMD64 Architecture Programmer's Manual Volumes 1–5*, january 2023 edition, 2023. [Online; accessed 20-June-2023].

[AMD23c] AMD. *Processor Programming Reference (PPR) for AMD Family 19h Model 61h, Revision B1 Processors*, march 2023 edition, 2023. [Online; accessed 20-June-2023].

[Bha23] Suresh Bhaskaran. AMD Expands Leadership Data Center Portfolio with New EPYC CPUs and Shares Details on Next-Generation AMD Instinct Accelerator and Software Enablement for Generative A. `https://ir.amd.com/news-events/press-releases/detail/1136/amd-expands-leadership-data-center-portfolio-with-new-epyc`, 2023. [Online; accessed 17-July-2023].

[CAP23] CAPS Team. CAPS Cloud. `https://www.ce.cit.tum.de/caps/hw/caps-cloud/`, 2023. [Online; accessed 16-July-2023].

[Cla23] Clang Team. Clang 16.0.0 documentation - OpenMP Support. `https://releases.llvm.org/16.0.0/tools/clang/docs/OpenMPSupport.html`, 2023. [Online; accessed 05-July-2023].

[Dro07] Paul J Drongowski. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. *Advanced Micro Devices*, 2007.

[Gam15] George T. Gamblin. Mitos. [Computer Software] `https://doi.org/10.11578/dc.20220805.1`, jun 2015.

[GCC23] GCC Team. GNU Offloading and Multi-Processing Project (GOMP). `https://gcc.gnu.org/projects/gomp/`, 2023. [Online; accessed 05-July-2023].

[Gon11] Antonio González. *Processor Microarchitecture*. Synthesis lectures on computer architecture. Morgan & Claypool Publishers, [San Rafael, Calif.], 2011.

[Gre21] Joseph Greathouse. AMD Research Instruction Based Sampling Toolkit. `https://github.com/jlgreathouse/AMD_IBS_Toolkit`, 2021.

[Gre23] Brendan Gregg. perf Examples. `https://www.brendangregg.com/perf.html`, 2023. [Online; accessed 16-July-2023].

[GV22] Alfredo Gimenez and Stepan Vanecek. MemAxes. `https://github.com/caps-tum/MemAxes`, 2022.

[Int23a] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*, march 2023 edition, 2023. [Online; accessed 20-June-2023].

[Int23b] Intel. Reference for performance monitoring events. `https://perfmon-events.intel.com/`, 2023. [Online; accessed 13-July-2023].

[Kuk15] Jim Kukunas. *Power and Performance*. Morgan Kaufmann, Amsterdam, [2015].

[Lar22a] Michael Larabel. AMD Zen 4 IBS Extensions Under Review For Linux. `https://www.phoronix.com/news/AMD-Zen-4-IBS-Linux`, 2022. [Online; accessed 04-July-2023].

[Lar22b] Michael Larabel. Linux 6.0's Perf Tooling Ready For AMD Zen 4 IBS. `https://www.phoronix.com/news/Linux-6.0-Zen4-IBS-Perf-Tools`, 2022. [Online; accessed 12-July-2023].

[Lin23] Linux Kernel Team. perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`, 2023. [Online; accessed 16-July-2023].

[Mes21] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[Ope18] OpenMP. *OpenMP Application Programming Interface Version 5.0*, november 2018 edition, 2018. [Online; accessed 05-July-2023].

[Ope23] OpenMP. The OpenMP API specification for parallel programming - About Us. `https://www.openmp.org/about/about-us/`, 2023. [Online; accessed 04-July-2023].

[per22] Top-down analysis with the perf tool. `https://perf.wiki.kernel.org/index.php/Top-Down_Analysis`, 2022. [Online; accessed 07-July-2023].

[per23a] *perf-record(1) — Linux User's Manual*, March 2023.

[per23b] *perf_event_open(2) — Linux User's Manual*, 6.04 edition, April 2023.

[She13] John Paul Shen. *Modern Processor Design.* Waveland Press, Long Grove, reissued edition, 2013.

[Tea23] TOP500 Team. TOP500 June 2023. `https://www.top500.org/lists/top500/2023/06/`, 2023. [Online; accessed 17-July-2023].

[Van23] Stepan Vanecek. sys-sage. `https://github.com/caps-tum/sys-sage`, 2023.

[W^+16] Vincent M Weaver et al. Advanced hardware profiling and sampling (PEBS, IBS, etc.): creating a new PAPI sampling interface. *Technical Report UMAINE-VMWTR-PEBS-IBS-SAMPLING-2016-08. University of Maine, Tech. Rep.*, 2016.

[Wea17] Vincent Weaver. System-wide performance counter measurements: Offcore, uncore, and northbridge performance events in modern processors. Technical report, Tech. Rep., 6 2017, 2017.

[Wea23] Vince Weaver. The Unofficial Linux Perf Events Web-Page. `https://web.eece.maine.edu/~vweaver/projects/perf_events/`, 2023. [Online; accessed 16-July-2023].

[Yas14] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.