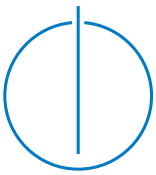
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

Page migration strategies on NUMA systems based on Sampling

Daniel Alberto Ortiz-Yepes



DEPARTMENT OF INFORMATICS

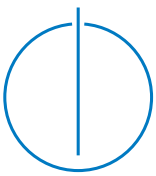
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

Page migration strategies on NUMA systems based on Sampling

Strategien zur Seitenmigration auf NUMA-Systemen

Author:	Daniel Alberto Ortiz-Yepes
Supervisor:	Dr. rer. nat. Josef Weidendorfer
Advisor:	Dr. rer. nat. Jens Breitbart
Submission Date:	November 15th, 2015



I confirm that this Master's Thesis in Computer Science is my own work and I have documented all sources and material used.

Daniel Alberto Ortiz-Yepes
Munich, Federal Republic of Germany, November 15th, 2015

Abstract

Contents

1 Introduction

2 Theorethical Foundations

2.1 Multicore Systems

Until the beginning of the decade of zear 2000 the advances in processor technology were made through improvements in the single CPU performance. These advances allowed the increase in the clock speed of the processors. Given the fact that the heat generated by a transistor is proportional to its operating frequency, the high clock frequencies reached led to a great amount of dissipated heat that could not be easily handled. In order to decrease the consumed power and take advantage of the increasing transistor count in every chip, the industry decided to incorporate multiple CPUs with a moderate performance instead of a single one with a high frequency and therefore power consumption. This new design trend is known as **multicore**.

In a single CPU the running code can be executed in a more efficient way by the CPU by using mechanisms such as the reordering of instructions. With the appearance of multicores there is a greater demand on the programmer to write code that can exploit all the available capacity offered by the processor. Part of this increasing programming effort consists of allowing the code to execute in units called **threads**, where a n core CPU has the capacity to execute n threads simultaneously. This expression of code in threads can be explicitly done with the help of frameworks such as *pthreads* or can be done implicitly by annotating regions of code that can be parallelized using tools such as *OpenMP*.

2.1.1 Thread pinning

Given a code that executes in n threads using a processor with m cores, we would like to have the ability to control which executing core will be used to execute a given thread. The mapping of an executing thread to a hardware core is known as **thread pinning** or **thread affinity**.

There are many mechanisms available to set the pinning of a program, such as:

- In OpenMP the environment variable GOMP_CPU_AFFINITY=a,b,c,... sets the affinity of a program as a list where a,b and c represent the identifiers of the cores that will execute the thread with the id in the position in which the identifier was

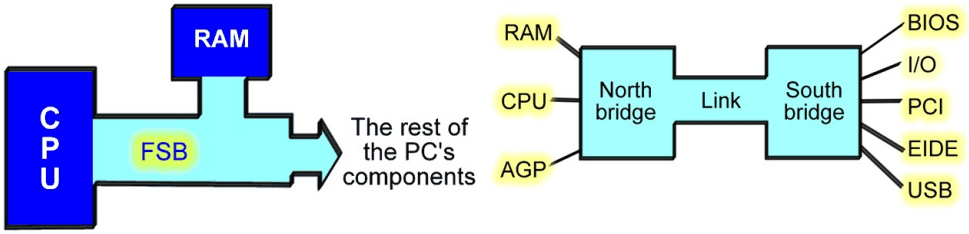


Figure 2.1: Abstraction of the data traffic between the CPU and the external entities. The left hand side abstracts the RAM as being closer to the cpu than the other components, and the right side shows a more accurate description with the Northbridge and Southbridge chips redirecting the traffic. Image Karbosguide [KA].

given. In this example thread 0 will be executed in core a, thread 1 in core b and so on.

- In numactl the parameter *physcpubind* can specify in a similar manner as is done in OpenMP, as a listing of the desired cores.
- In Autopin+ the user specifies many possible pinnings, and the tools will explore these combinations of pinnings for the most optimal in terms of execution speed.

Pinning can play a very important role in the performance of a program. In many cases distributing a thread to a free core might yield the best performance, but in other scenarios -especially in NUMA systems- this might not be the case depending of the locations of memory accesses by the thread, these factors will be explored later in more depth.

2.2 The bus interface and multicore systems

In earlier computer systems, the processor chip had a communication with the memory through a high speed path called the Front Side Bus (FSB), which connects the last level of cache of a processor chip with the memory controller. This Front Side Bus also allows the communication between the processor chip and other peripherals through the Northbridge chip. This layout means that all the traffic between the processor and the exterior world was centralized through the Front Side Bus, but in many cases a great share of the traffic is occupied by the memory-processor data. The visualization of this layout is given in Figure 2.1.

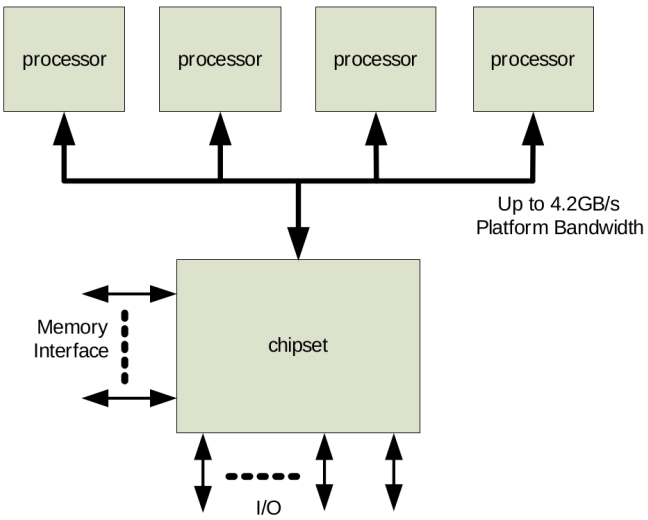


Figure 2.2: Basic UMA System with the Frontside Bus as an unifying element. Image Intel Corporation [Int09].

Given the always present need to optimize the supply of memory data to the CPU, this front side bus scheme has been changed and a quicker way for the CPU to access memory has been devised, these improvements can be seen in the form of higher bandwidth and low latency. Nowadays the path between the processor and the memory controller has been taken in-chip, which means that the processor has a straight path to the memory controller inside the same silicon die. In many modern multicore processors, specifically those from Intel and AMD, every core has its own L1 and L2 caches and all the cores share the L3 or last level cache. With the introduction of the dedicated memory access interface every processor chip has a part not specific to any core which handles the access to memory, in Intel technology this in-chip intermediation area is known as the *uncore*.

2.3 Multiprocessor Systems

In situations where the power of a single processor computer is not enough -even if this processor is a multicore- it might be possible to incorporate in a computer system multiple processor chips. Usually in such multichip systems all the memory can be accessed by any of the participating processors. As the cores see a global memory space, it is possible that two processes access the same memory location at the same time. In this simultaneous access case it is advisable that when one processor writes

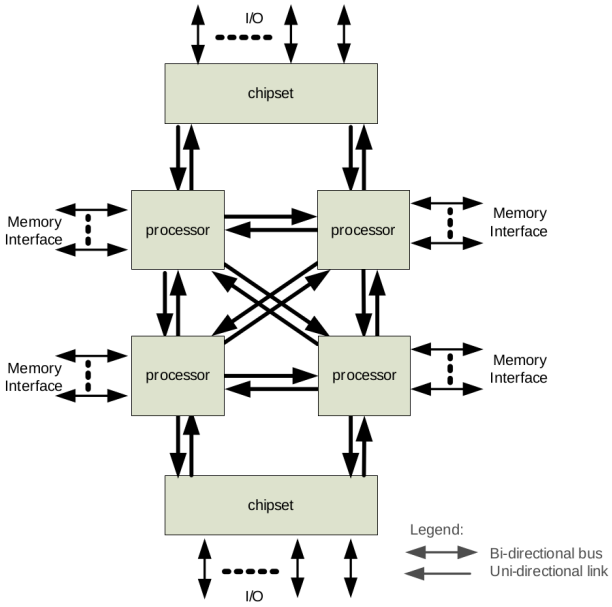


Figure 2.3: Numa system with four nodes with a QPI interconnection. Image Intel Corporation [Int09].

over a memory location the other processors -even the ones located in other chip- see this change as well. This concept of keeping a synchronized version of a shared memory location is known as *cache coherency*.

In earlier multiprocessor systems the sharing of data between processors was done through a connection to the Front Side Bus, where memory can be accessed through the chipset (Northbridge) as shown in the Figure 2.2. This shared bus topology meant that all memory accesses are treated equally no matter which processor accesses it, thus guaranteeing properties such as the latency of an access. This equality of accesses leads that this type of interconnection is known as an **Uniform Memory Access**. Later some enhancements were made to this architecture such as the segmentation of the shared buses and joining of these separate buses through cache coherency enforcement hardware [Int09].

2.3.1 Numa systems

Although the interconnection through the Front Side Bus provides very firm guarantees about the access time it has a very obvious downside, which is the bottleneck that could be produced when all the memory accesses are conducted through a sin-

gle path. This bottleneck, together with the need to give the processor more memory access bandwidth with less latency has led to the development of point to point connection between processors. Instead of being just a bus protocol, these processor-memory links have taken the form of a layered protocol with packetized units of communication specific to every manufacturer. The most common forms of this new interface are Intel's Quick Path Interconnect and AMD's HyperTransport technologies. Figure 2.3 shows the representation of an Intel NUMA system interconected using QPI.

With the introduction of the point to point interconnects a processor faces two options when given the task of making a memory access: either the desired location resides in its attached memory or it must fetch the location from a remote processor. A remote memory access has more latency than a local one due to the interprocessor transport involved in the transaction, which leads to remove the guaranteed access latency given in the UMA system. This latency guarantee cannot be held anymore because of the placement of the threads and memory pages in a multiprocessor system can be easily changed thus introducing the NUMA architecture, which stands for: **Non uniform memory access**.

2.4 Computer performance measurement

Modern computers are complex systems and there are multiple factors that might contribute to a degradation in system performance. The addition of performance counters, as they are usually called, allows to help track occurrences that happen inside a processor and reading them facilitates the analysis of where a bottleneck could be taking place. These occurrences can be for example cache or TLB misses, memory loads or instructions retired from the pipeline, among others. The performance tracking facilities are also referred to as the *Performance Management Unit (PMU)*.

In modern CPUs the number of available counters is fixed and every counter is programed by assigning it an event such as the ones mentioned above. The set of available events for every processor varies depending on the microarchitecture.

2.4.1 Instruction sampling

In the performance analysis of an algorithm sometimes it is desirable to associate the occurrence of an event with the instruction that caused it. Due to the great number of instructions executed by a processor it is not possible to keep track of every one of them. The strategy to apply here is to register information every certain number of instructions and extrapolate the results to the execution of the whole program. This follows the principle of *Statistical Sampling* and thus it is called the **instruction sampling**.

The first sampling versions of instruction sampling that appeared in Intel and AMD processors sometimes had trouble with associating the exact event measured with the instruction in a precise manner, which caused a deviation in the association between the instruction and the event. Later, a revised version of the technology eliminated this shortcoming and it is present in current AMD and Intel processors with the proprietary names of *Instruction Based Sampling* [Dro17] and *Precise Execution Based Sampling* respectively.

2.4.2 Accessing the performance counters from code

There is no direct API provided to access the information contained in the access counters and due to the information that some counters expose in many cases, this access can only be performed just by code that executes in system mode.

In Linux, which is the operating system chosen for the implementation of the solution, the access can be done through the perf facilities implemented in the kernel. There is a system call that allows to configure the counters to be used and the resulting readings can be retrieved by reading to a memory location. For a more detailed description of the access to the counters through perf, refer to the section **??**.

2.4.3 Documentation of the performance measurement facilities: The Intel case

Every processor equipped with performance counters has the description of these facilities in the operation manuals. In the case of Intel, the relevant information can be found in the chapters 18 and 19 of the System Programming Guide [Cor15], although there are separate documents which extend this information, such as the uncore performance monitoring guides. The availability and layout of performance information is not architecturally uniform, this means that every microarchitecture, even from the same architecture family, has very specific performance measurement features that might differ from other microarchitectures.

One of the examples of these difference between microarchitectures is the case of the registers, those who are present in all the architectures such as the EAX registers are named Architectural registers and those that change through the changes in architectures are called **Model Specific Registers**, or MSRs. Chapter 18 gives an account of the performance measurement features present in every processor family and chapter 19 gives the concrete event information for all the events available.

Following this separation of architectures, these chapters present an overview of the performance measurement in Intel processors and then all the specific information for every microarchitecture family is presented separately. Usually, in order to make use

Event Num	CDH
Umask Value	01H
Event Name	MEM_TRANS_RETIRED.LOAD _LATENCY
Definition	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization. PMC3 only
Description and comment	

Table 2.1: Description of the load latency record event as it appears on the Intel manual

of a counter we need to program it with the identifier of the event we are interested on, together with some additional information, if necessary.

All the information about bit fields in this project is ignored, because such low level is not needed and the low level access to the registers will be performed by the kernel side of perf. For us it will be necessary only to fill a struct with some fields that specify the events we want to read. For example, in the case of the Sandy Bridge microarchitecture, we will be using an event called the **load latency record**, which is defined in table 19.9 of such guide. Table 2.1 gives an example of the information contained in the event reference. The identifier of the event to be measured is given by the event number together with the umask value, the definition provides a definition of what the event measures and the description and comment field indicates that for this case additional information can be specified, such as the minimum weight that a sample must have in order to be passed to the software .

2.5 Related software

2.5.1 Libnuma

Libnuma [Kle05] is one of the most important libraries for the control of NUMA systems. Among its most important features it incorporates the ability to control the processor and memory mappings of a process within a NUMA context.

Of particular importance to this project is the `move_pages` call, which allows to determine in which node a given memory page is and commands the move of the desired memory pages from one node to another. Many of the functionalities offered by libnuma are available as console invocable commands by using the *numactl* application. Numactl also provides a quick way to discover the NUMA layout of a processor by using the `-hardware` argument.

2.5.2 Autopin and Autpin+

Autopin is a tool developed by the chair of computer technology and organization of the Technical University of Munich [Klu+11]. As explained in the introductory section the pinning or mapping of the running threads to the available cores can be crucial to obtaining good algorithm runtime performance. When the optimal pinning of an application for a given system is not known, a good approach is testing many alternative pinnings to find out which one helps the application perform better, which is precisely the way Autopin works. For Autopin to work, the process to run and the alternative pinnings to be tested are given as an input. On execution Autopin will run the requested process or attach to an existing one, try the different pinnings for a certain amount of time and when all the pinnings are tried stick to the best alternative in terms of performance. For performance measurement Autopin will calculate the performance rate as a ratio between the difference of two PMU readings and the time the program was allowed to run under a certain pinning. The PMU readings are related to the number of instructions executed and are taken at the beginning and end of the measurement period. Before the measurement of the performance rate and the set up of an specific pinning there is a period of time that the program will be allowed to run without making any measurement, this interval will be referred to as the warm-up period.

2.5.3 Software for performance measurement

Currently there are many alternatives for accessing the performance information and even for analyzing it in order to formulate diagnostics about the performance of the program. This section aims to briefly indicate some ready to use alternatives present in the market:

- Intel VTune: Closed source licensed tool which features a rich graphical interface and advanced diagnostics capabilities. Only available for Intel processors.
- Perf: Performance measurement implementation for the Linux kernel. It features the ability to access performance information for multiple processor architectures. Provides an interface to create user space programs that process the measurement information.
- Intel Performance Counter Monitor: Open source tool which provides facilities to access the CPU performance counters facilities as a C++ API. Only available for Intel processors and for many features requires root access.

- Intel NumaTop: Open source tool that follows the principle of the Linux Top Tool. Instead of providing information of the CPU utilization provides information of how much remote and local bandwidth each process is making use of, together with other information such as information of the allocation of memory and measurement of the average memory latency. Only available for Intel processors.

2.5.4 An introduction to perf

Perf has become one of the most important tools for accessing the CPU performance information, its code has been incorporated into the Linux source three. Unfortunately the code is scarcely documented and the best way to learn its workings is through the analysis of the implementation and the reading of the repository commits. Although perf belongs to the kernel source tree it is not necessary to compile a kernel to generate a working version of the user side of perf because it contains its own Makefile.

Perf is divided in two parts, the Kernel side and the user tools: The kernel side takes care of the direct communication with the Performance monitoring unit and providing a representation of the performance data independent of the underlying architecture and even different microarchitectures of the same processor family. The user side contains a set of tools which allow to gather the performance information and perform different analyses. There is also the opportunity to create a custom user side that relies on the kernel side, which is the approach that will be taken during the development of this project. The specific aspects of invoking the kernel side will be discussed in section **??**.

Perf not only allows to access the performance information given by the CPU, but also allows to access performance information that is stored in the kernel, for more information type `perf list`. The system administrator must tune the `perf_event_paranoid` setting depending on how much access he wants the regular user to give.

3 Solution design

3.1 Overview of the SPM tool

Figure [X] shows the high level operation of the SPM tool through time, which has three parts: the measuring part, the page migration and the improved part. The measuring part SPM measures the memory behavior of the guest process, where the length of this process is determined by the user and set in the overall SPM configuration. The page migration will reallocate the home of the pages according to the information obtained in the measurement phase and a pre-established policy. The length of the page migration phase depends solely on how long it takes to migrate the pages marked for this process. At the end, the improved phase lasts until the end of program execution, where it will again analyze the memory behavior of the program and print information at the end so that an analysis can be derived to find out whether there was an improvement on the performance after making the page migration.

3.2 Strategy and alternatives of development

In order to implement the desired functionality the idea is to adapt the code from one of the already working tools that access the PMU to the needs of the project. Two options were considered: perf and Intel numaTop, with the latter being chosen. It is worth noting that for the communication with the Performance Measurement Unit both tools make use of the same set of calls. A short description of both alternatives will be given in this chapter.

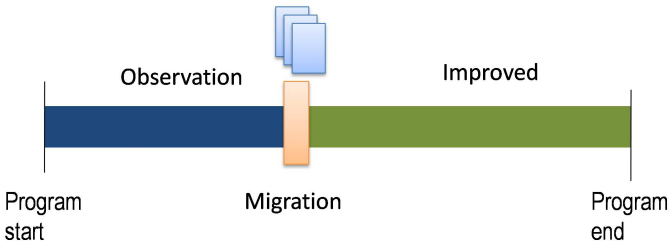


Figure 3.1: TODO add caption

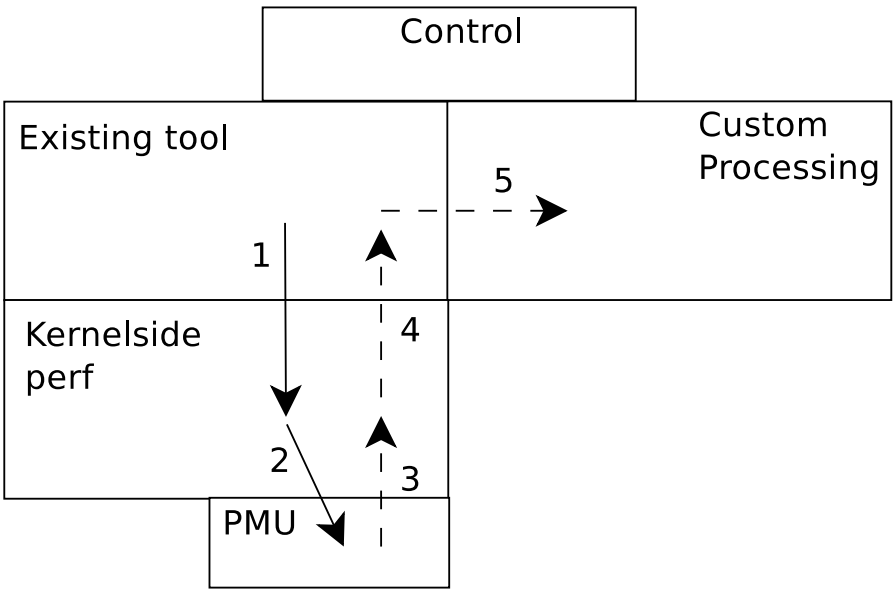


Figure 3.2: TODO add caption

Figure [x] shows the overall working principle of the developed tool. The developed tool invokes the configuration capabilities from the kernel side, which then configures the physical PMU. Once configured and started the PMU starts passing samples to the kernel side, which passes them on to the measurement tool once the buffers are read. Upon a read in user space the sample will be passed to the custom tool for processing the samples. The control part will be in charge of coordinating the execution of all the parts of the program.

3.2.1 Perf user tools

This is the most comprehensive in terms of supporting events and covering a diverse range of architectures and performance events. Unfortunately this complexity together with the lack of documentation make the adaptation process a time consuming task and the end solution comprises a great number of source files, which cannot be shrunk without undertaking a large effort.

The perf user tool `perf top` was adapted in order to have a constant and never ending access to the samples. In the function perf_event process samples in the file builtin_top.c a call is made to intercept the produced sample and analyzed according to our needs. The GUI of the executable is disabled.

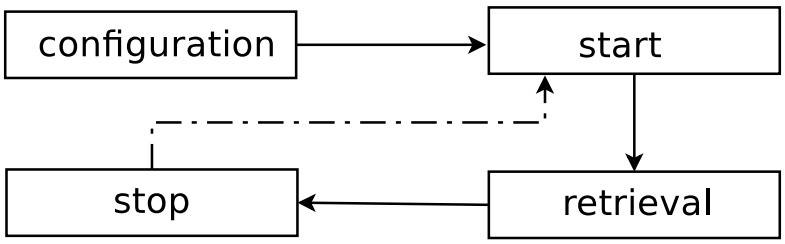


Figure 3.3: TODO add caption

3.2.2 NumaTop

NumaTop provides a reduced functionality in comparison with the user side of perf, but it still provides all the functionality that is needed for this project. Its smaller code size helps to understand how the sample acquisition process works. The needed methods were grouped into a smaller file in order to make the number of source files smaller and around this the SPM code was built

3.2.3 Evaluation

NumaTop provides a reduced functionality in comparison with the user side of perf, but it still provides all the functionality that is needed for this project. Its smaller code size helps to understand how the sample acquisition process works. The needed methods were grouped into a smaller file in order to make the number of source files smaller and around this the SPM code was built.

3.3 Communicating with perf's kernel side

In order to be able to retrieve samples from the kernel side, a small amount of system calls have to be made, this section describes the flow for using the measuring system and lists the system calls used. Figure [x] shows an overview of the sampling process with its four stages: configuration, start, retrieval and stop.

3.3.1 Configuration

The configuration of the PMU is made for every event and for every CPU that we want to watch. By making the system call with identifier __NR_perf_event_open. An actual call would take the form:

```
syscall(__NR_perf_event_open, attr, pid, cpu, group_fd, flags)
```

- Attr: A struct of type perf_event_attr which specifies the sampling parameters, its fields are better explained in section [INSERT REF].
- Pid: Takes the value -1, in order to be able to monitor all the executing PID.
- CPU: Takes the identifier of the core whose PMU to configure.
- Group_fd: For this implementation takes the value -1.
- Flags: For this implementation it takes the value 0.

After setting up the PMU the function mmap is called in order to create a file descriptor which will be used to read the samples. In order to create the descriptor, the first parameter of the mmap function call is NULL. The most important parameter in the PMU set up is the attr field because it specifies what we want to measure and how we want it.

3.3.2 Start and stop of the sampling

To start and stop the sample unit the following system call is made:

```
ioctl(descriptor, constant, 0))
```

Where descriptor is the file descriptor assigned to every combination of CPU and event and constant takes the value PERF_EVENT_IOC_DISABLE or PERF_EVENT_IOC_ENABLE depending on the objective.

3.3.3 Retrieval and the reading the samples

The file descriptor must be read periodically in order to collect samples. Given that the acquisition of the samples is made through low level I/O, the read is made by reading a chunk of certain number of bytes and it is up to the programmer to turn this stream into a suitable representation.

The read of the descriptor is made using the mmap function, which takes as parameter the descriptor, the length of the read and the buffer where the read information will be kept.

3.4 Important data structures

3.4.1 The perf_event_attr struct

The perf_event_attr is the struct used to configure the PMU as shown in section [missing section]. In its fields the configuration parameters of a PMU are set. As it have a relative number of fields only the most important will be explained here. For the complete description of this struct it is advisable to check the man pages of the sys_perf_event_open call and the document design.txt located on the root of the perf user side source three. This struct is defined in the kernel header perf_event.h

- record_type: For the use in our program will always take the value 4 (PERF_TYPE_RAW) since we are retrieving data from the hardware PMU. It is possible to collect data from other sources, such as the kernel itself, for instance.
- sample_period: Specifies the number of instructions between consecutive samples. It is also possible to specify a frequency instead of a period by setting the use_freq field.
- Precise_ip: It is used to specify that the event to be read uses precise sampling(see section xx)
- Config and config1: Config specifies the event to read as it is formed by concatenating the umask and the id. Config1 is used to specify any additional information that the event measurement requires.
- Sample_type: This field consists of the logic of the fields specified in the perf_event_sample_format enumeration and determines which information is returned in every sample , for example PERF_SAMPLE_IP | PERF_SAMPLE_TID | and so forth.
- Disabled: Determines whether the sampling requires further additional activation.

3.4.2 A perf sample

The samples are retrieved from a sequential read to a file, so it is up to the programmer to give sense to this incoming information. The first information coming from a sample is the header, which follows the layout of the struct perf_event_header. From this header we obtain two important pieces of information: the first one is the size of the sample, and the other is the type of the sample. For the type of the sample and our purposes it is only important the samples of type PERF_RECORD_SAMPLE. Once read the header, the next size bytes will belong to a sample.

The information coming in a sample is determined by what was configured in the Sample_type field when configuring the sampling. What is still unknown is the order in which the fields are returned. In order to figure this out it is better to refer to an existing implementation. For the consumption of the sample body the idea is to read each field and displace the reading position the number of bytes that a field occupies. The methods profiling_sample_read and ll_sample_read provide the sample reads to be used in the development.

3.4.3 The numatop core

The SPM tool is based on an adaptation of the numaTop methods used for communicating with the perf kernel side. This functions are located in the sampling-core.c file of the SPM sources, or on the numaTop sources themselves.

The functions follow the logic stated in section (Coming with perfs kernel side) and they exist in two varieties, the methods with "profiling" in their name refer to the access of the registers that refer to a sampling value, such as the access to the offcore registers and the methods with "ll" refer to the load latency records, which are precise methods. Their functionality is very similar.

- The functions responsible for setting up the PMU are pf_profiling_setup and cpu_ll_setup
- The functions responsible for starting the PMU are pf_profiling_start and pf_ll_start
- The functions responsible for stopping the PMU are pf_profiling_stop and pf_ll_stop

3.5 Main layout of the spm tool

Figure [x] shows the overall principle of implementation of the SPM tool. The launching thread sets up the system and creates the SPM thread, which is in charge of collecting the samples from the numaTop core. The SPM thread then will launch the control thread, which controls the overall execution of the three phases, prints the statistics, invokes the migration and ensures that the observed process is executing on the background.

3.6 Integration with Autopin+

The SPM functionality is integrated into the code base of autopin+. For this purpose SPM's sources are added to the ccmake configuration script (CMakeLists.txt). For the

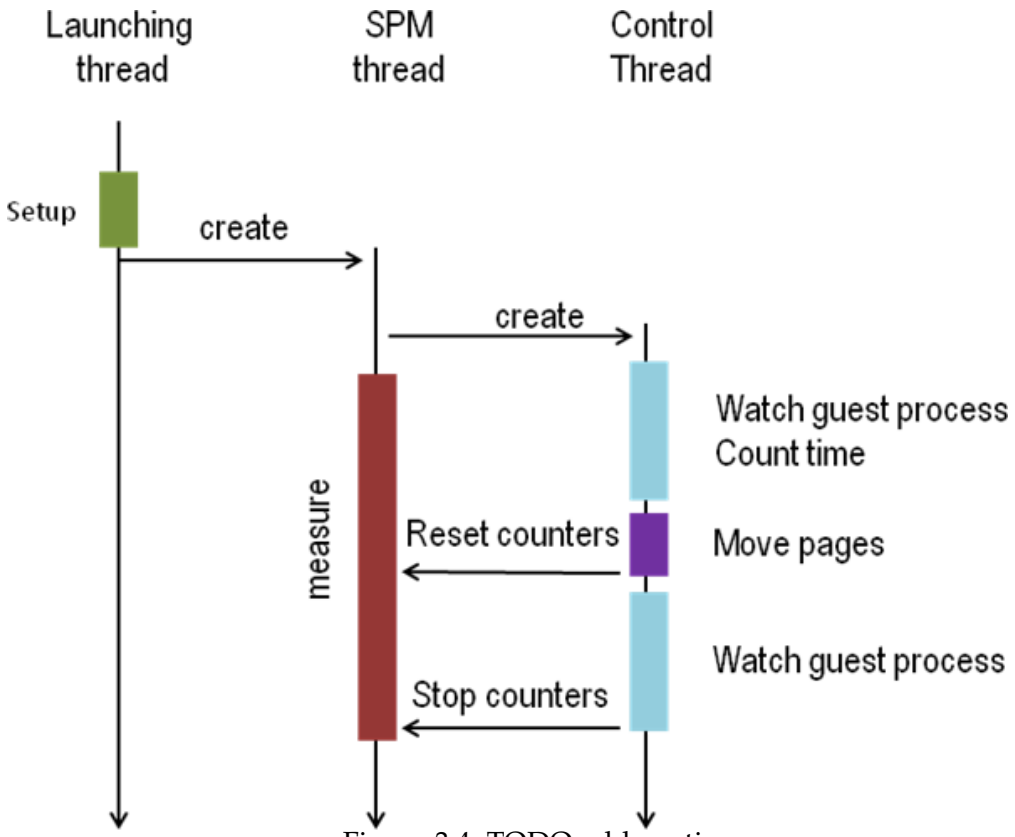


Figure 3.4: TODO add caption

calling of the SPM functionality within autopin+, a few additions to the autopin+ code are done:

1. A new descendant of the PerformanceMonitor is created, which is then named PageMigrate.
2. The recognition of the new class is enabled by adding the corresponding logic in the selection if present in the createPerformanceMonitors located in the Watchdog class.
3. A new field is added to the PerformanceMonitor class together with the setter method, which allows it to retain the identifier of the process to watch. This is necessary because until now a performance monitor is enabled using the identifier of a thread and in the SPM tool the pid is employed to enable the tool.
4. The start method is modified in order to invoke the init_spm function. Before the invocation the configuration struct is created and the necessary fields are filled.

List of Figures

List of Tables

Bibliography

[Cor15] ". Corporation". *Intel 64 and IA-32 Architechtures Software Developer's Manual, Volume 3: System Programming Guide*. Jan. 2015. URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[Dro17] P. J. Drongowski. *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*. AMD. Nov. 2017. URL: http://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf.

[Int09] Intel Corporation. *An introduction to the Intel QuickPath Interconnect*. http://www.intel.de/content/dam/doc/white-paper/quick-path-interconnect-introduction-pa pdf. Intel Corporation, Jan. 2009.

[KA] M. Karbo and E. Aps. *PC Architecture*. URL: http://www.karbosguide.com/books/pcarchitecture/chapter00.htm.

[Kle05] A. Kleen. *A NUMA API for Linux*. AMD. Apr. 2005. URL: http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf.

[Klu+11] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis. "autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems." English. In: *Transactions on High-Performance Embedded Architectures and Compilers III*. Ed. by P. Stenström. Vol. 6590. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 219–235. ISBN: 978-3-642-19447-4. DOI: 10.1007/978-3-642-19448-1_12. URL: http://dx.doi.org/10.1007/978-3-642-19448-1_12.