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

About the author

Daniel Ortiz-Yepes was born in March 1985 in Bogotá, Colombia. He holds bachelor degrees in Electronics Engineering and Systems and Computing Engineering, both obtained at the University of los Andes in Bogota, Colombia in year 2010. In October 2013 Daniel began his Masters Studies in Computer Science at Munich's Technical University, for which this work aims as a thesis.

He can be contacted at his email: daorti@egresados.uniandes.edu.co

Contents

1. Introduction

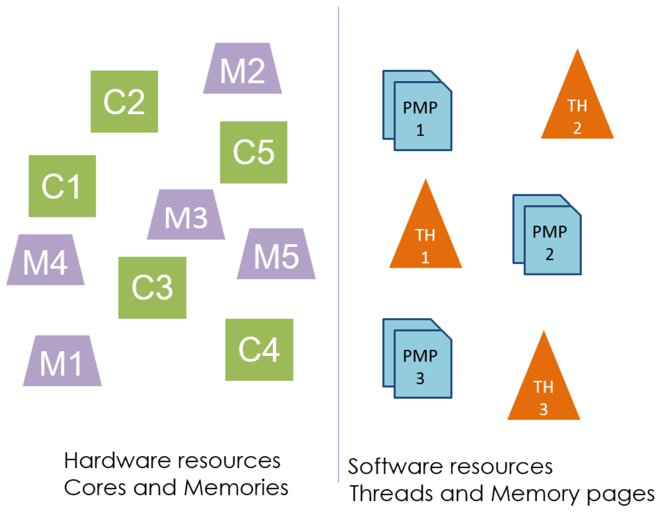

Figure 1.1.: Abstract view of a modern computing environment

An abstract representation of a contemporary computing environment is shown in figure 1.1: The left side consists of computing cores and memories, which are hardware resources and the right hand side shows threads and program memory pages, which are software resources. One of the most challenging goals in computing is to extract the maximum performance possible, which implies the execution of the desired applications with the greatest amount of throughput and the least possible amount of energy usage. For this goal to be achieved, the mapping of the software resources to the hardware resources plays a very important role, this means, which processor executes which threads and which physical memory stores which software memory pages. Another characteristic in today's computing environments is the dynamic makeup of the software resources, which means that the number of resources present in the system change through time and also its nature such as threads being computing bound or memory bound.

To respond to these challenges, the chair of Computer Architecture and Organization of the Technical University of Munich has developed research which aims to address those requirements: The development of the Autopin and Autopin+ tools deal with the mapping of the threads to CPU cores and the relocation of the threads present on the system in the presence of varying loads. With the mapping of the treads to the cores addressed, now the mapping of the program memory pages to the memory modules remains. In NUMA computer systems every processor package has the memory directly attached to it, which helps to increase the access bandwidth and decrease the latency when the accesses originate from the cores present in the chip, but this latency increases when the access originates from one core which belongs to a package different from the one connected to the memory holding the requested page. The increase in the number of remote accesses might be caused for one of the thread reallocations made by the Autopin tools, and the need to optimize the memory allocation is what leads to the development of the present project.

This document aims to give detail of the development of the sample page migrate tool, or SPM. SPM is a software tool which analyzes the memory access behavior of an observed process by fetching data provide by the processors' Performance Measurement Unit and based on it reallocates the distribution of the pages if necessary. SPM has two main requirements: First it must support the processors featuring the Intel Sandy Bridge-EP microarchitecture and second its implementation must be made up only of user space code.

The action of the SPM tool is divided into three phases: First, in the observation phase SPM will collect the memory behavior information from the processor's performance measurement unit, and after a fixed amount of time it will reallocate the pages it seems fit, which happens in the migration phase. Finally in the improved phase it lets the program run until completion and also collects the memory behavior data, but just to facilitate a comparison of the performance with respect to the first phase. In this phase a better performance in comparison to the observed phase is an indicator of the success of the tool.

The SPM tool analyzes the behavior of the observed application in a black box manner, which means that SPM does not have any information about the internals of the observed application and the only information it has is the one provided by the Performance Measurement Units, this means that these units must be tuned to get a fair amount of samples, which is possible by configuring a few parameters that regulate how this units works.

The present document is structured as follows: Chapter 2 presents the concepts that support the development of the SPM tool. Chapter 3 presents a deeper description of the performance sampling process in a Linux system and explains in detail the implementation of the SPM tool. Chapter 4 introduces the characterization of the Intel Sandy Bridge-EP PMU and the benchmarks to be used. Chapter 5 presents the results of running the SPM tools together with the distgen and LAMA algorithms and Chapter 6 presents the conclusions and possibilities of future work.

2. Theoretical Foundations

This chapter gives a brief review of the topics that are needed to have an understanding of the concepts that the present developments deals with. It begins with a description of multicore systems and the interconnection between processor and memory and then introduces a special kind of multiprocessor system: the multiprocessor system and goes further into one very special and famous kind of multiprocessor system that we are interested in: the NUMA system. In the second part an introduction to the topic of computing performance measurement is given, giving some examples of current alternatives for performance measurement programs.

2.1. Multicore Systems

Until the beginning of the decade of year 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 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.

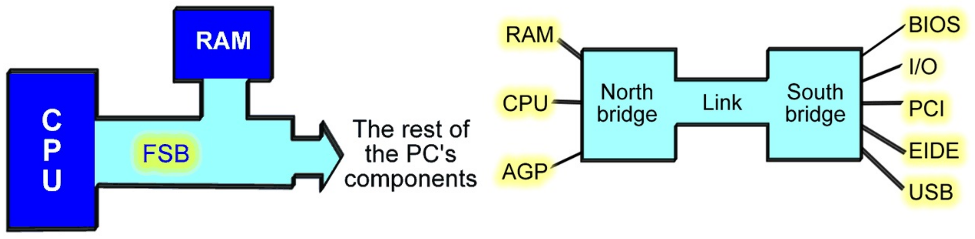

Figure 2.1.: Abstraction of the data traffic between the CPU and the external entities. The left hand side abstracts the RAM as being closer to the cpu than the other components, and the right side shows a more accurate description with the Northbridge and Southbridge chips redirecting the traffic. Image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

Given 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 over a memory location the other processors -even the ones located in other chip-

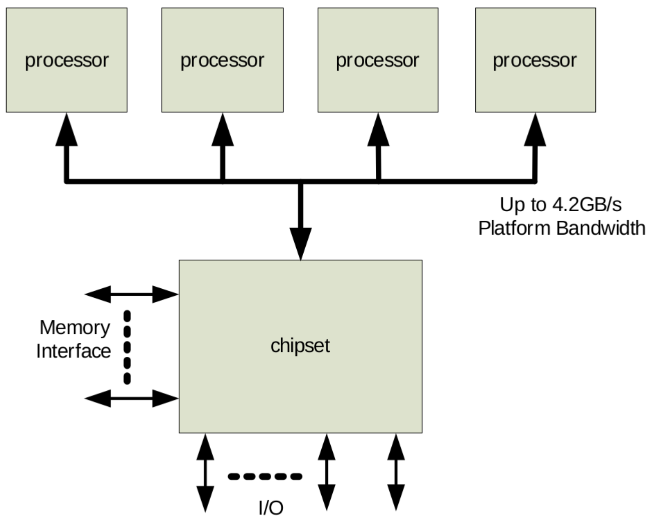

Figure 2.2.: Basic UMA System with the Frontside Bus as an unifying element. Image Intel Corporation [Int09].

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 single path. This bottleneck, together with the need to give the processor more memory

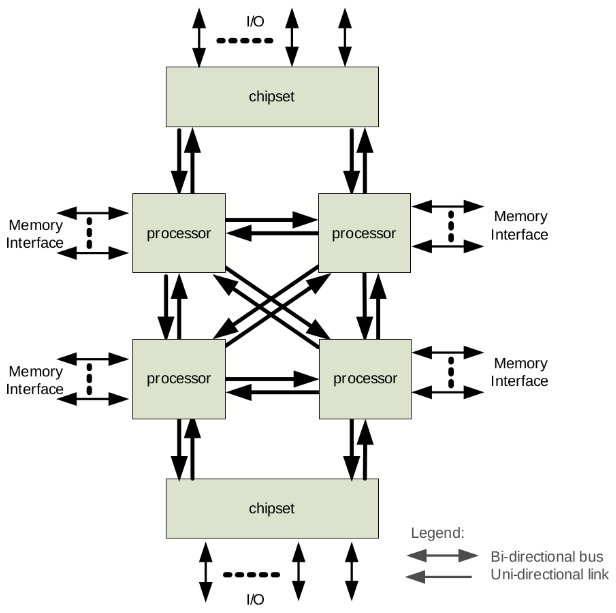

Figure 2.3.: Numa system with four nodes with a QPI interconnection. Image Intel Corporation [Int09].

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 3.3.

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 [Int15], 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 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

Event Num	CDH
Umask Value	01H
Event Name	MEM_TRANS_RETIRED.LOAD _LATENCY
Definition Description and comment	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

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

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 Autopin+

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 3.3.

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

This chapter aims to describe the technical details of the developed software, namely the sample page migrate, also referred to SPM throughout this document. The main questions to discuss in this chapter are:

- What are the working fundamentals of the SPM tool?
- How does the sampling process work?
- How is the SPM structured?

This chapter develops three parts: Sections 3.1 and 3.2 describe from a very high level point of view the desired functionality and inspiration for the SPM tool, then sections 3.3, 3.4 and 3.4 describe the sampling acquisition process from a PMU in a Linux environment and finally section 3.5 discussese the design and implementation details of the SPM tool.

3.1. Overview of the SPM tool

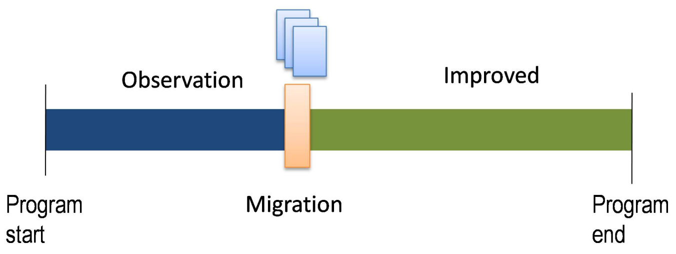

Figure 3.1.: The developed tool consists of three main phases: the observation phase, the page migration phase and the improved phase.

Figure 3.1 shows the high level operation of the SPM tool through time, which has three phases of execution: the **observation phase**, the **page migration phase** and the

improved phase. During the measuring phase, 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 observation 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. 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 to make possible an assessment over the benefits of the page migration process.

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 *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 library calls. A short description of both alternatives will be given in this chapter.

Figure 3.2 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 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

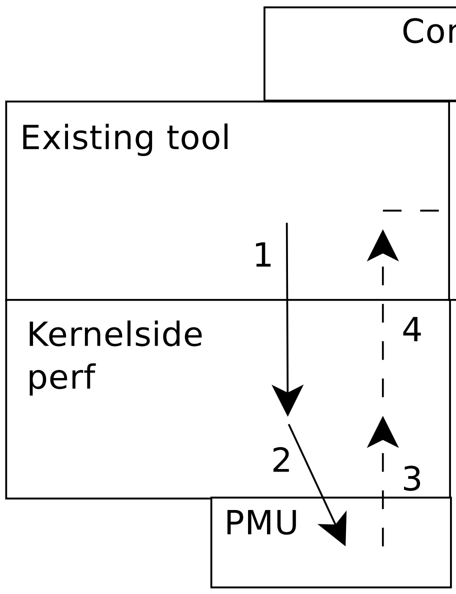

Figure 3.2.: Interaction between the parts of the performance measurement. The continuous arrows show the flow of control calls and the dashed line shows the flow of the performance samples.

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.

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 better understand how the sample acquisition process better 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

Even though both provide a suitable alternative to the functionality needed, the implementation based on numaTop was chosen mainly because it provides a smaller code base that compiles faster and is easier to understand.

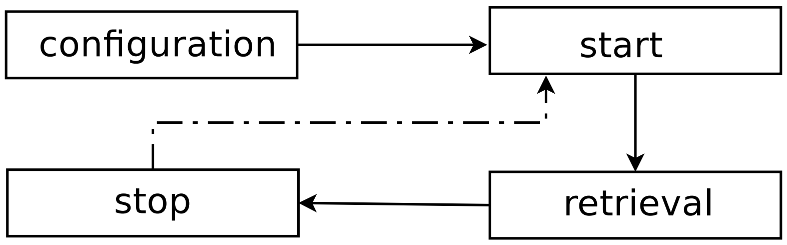

Figure 3.3.: Steps involved in the performance measurement process, the dashed line means an optional transition.

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 mentions the system calls used. Figure 3.3 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 is to be watched, 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 3.4.
- Pid: Takes the value -1, in order to be able to monitor all the executing PIDs.
- CPU: Takes the identifier of the core whose PMU is to be configured.
- 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. The most important fields of this struct are explained in section 3.4.1.

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 desired action.

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.

3.3.4. The numatop core

The SPM tool is based on an adaptation of the numatop functions 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 described in section 3.3 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.

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 3.3. In its fields the configuration parameters of the PMU are set. As it has a relatively high 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 tree. 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 only 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 2.4.1).
- 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 a specific event requires.
- Sample_type: This field consists of the logic OR of the chosen 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 or starts immediately after the configuration call.

3.4.2. A perf sample

The samples are retrieved from a sequential read to a file descriptor, 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 the payload of 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 functions *profiling_sample_read* and *ll_sample_read* provide the sample reads that are used in the development.

- 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. Internals of the spm tool

3.5.1. Layout of the spm tool

Figure 3.4 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.

3.5.2. Structure of the spm tool

The SPM tool consists of the following files:

- spm.h: Contains struct declarations and function prototypes.

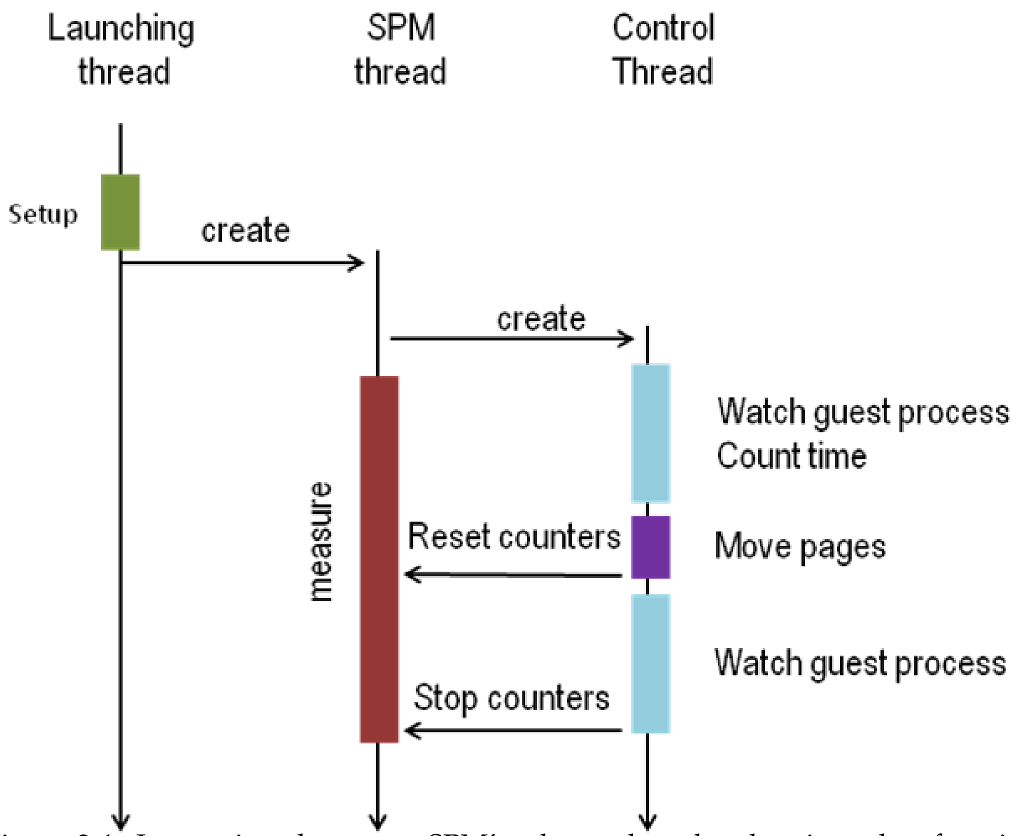

Figure 3.4.: Interaction between SPM's three threads showing the functionality through time

- sampling-core.c: Contains the core functions from numatop that access the PMU.
- perf_helpers.c: Contains a few functions borrowed from perf's user side tools.
- sample-processing.c: Contains the functions related to the processing of a sample.
- perf_helpers.c: Contains a few functions borrowed from perf's user side tools.
- sample-processing.c: Contains the functions related to the processing of a sample.

- control.c: Contains the logic that controls the whole execution lifetime of the SPM tool.

3.5.3. Information collected by the SPM tool

The information shown by SPM at the end of a measurement round is as follows: Total number of samples taken in the phase and number of samples collected belonging to the observed process.

- Number of remote candidate and total samples taken for every core.
- Breakdown of samples taken by access type: How many samples are L1 hits, L2 hits, remote access and so forth.
- Histogram of frequency of page accesses, that is a detail of how many pages were accessed a given time
- Histogram of the number of pages with a latency value that belong to a certain interval. During this document the access weights are placed on intervals of 50.

3.5.4. Overview of the sample processing performed by SPM

The samples are retrieved from the numatop functionality in the read_samples function. From there the consume_sample function is invoked, which collects all the needed information from a sample:

- The counter for the total number of samples is increased, as well as the number of samples per core.
- Only the samples that belong to the PID we are interested in are further processed.
- From the data_source field of a sample the type of access is derived. L1, L2 and L3 hits are not further processed because they are local accesses. This type of access is added to the type of access histogram.
- From the latency value an entry is added to its corresponding interval in the latency histogram.

- If the address of the page contained in the sample belongs to a candidate remote access, that page address will be added to the hash table of page accesses, which are used as candidates for a page migration. If the entry already exists it will be retrieved and an access belonging to the originating core will be registered, otherwise it will be created and incorporated into the hash table.

3.5.5. Access to the SPM functionality

The SPM functionality is engaged by simply calling the function:

```
int init_spm(struct sampling_settings *ss)
```

where it takes as a parameter a properly filled struct of type sampling_settings.

3.5.6. Configuration of the SPM settings

The configuration of SPM is passed as start time to the *init_spm* function as a pointer to a struct of type *sampling_settings*, whose fields have the following meaning:

- int pid_uo: Indicates the process id of the observed process. If its value is equal or less than 0, SPM will launch the command indicated in command2launch and fill this field automatically.
- int *core_to_cpu: This field contains an array whose value at position n represents the NUMA node to which core n belongs, it is filled automatically at configuration time.
- const char** command2_launch: Specifies a command to launch by SPM together with its arguments. If a pid is specified it is not necessary to specify it.
- char* output_label: Used to prepend a label to the statistics print by the tool, useful when using automatic processing of the results.
- Int argv_size: Size of the command2_launch array.
- int measure_time: Time in seconds that the measure phase will last.
- boolean_t only_sample: Used to indicate SPM to run only a observation phase, without page migration.

- int ll_sampling_period: Contains the sampling period of the precise registers.
- int ll_weight_threshold: Samples below this value are ignored, must be greater than 3.

The following fields are meant only for internal use of the tool:

- int n_cpus: This field contains the number of NUMA nodes in the system.
- int n_cores: This field contains the number of cores in the system.
- perf_cpu_t *cpus_ll: File descriptor to access the load latency registers.
- perf_cpu_t *cpus_pf: File descriptor to access the non-precise registers.
- boolean_t end_recording: Internal flag to signal the sampling system that sampling should be ended.
- struct l3_addr *pages_2move: Contains a linked list with all the pages candidates to be moved.
- int number_pages2move: Contains the size of the pages_2move list.
- int moved_pages: Contains how many pages were moved.
- struct sampling_metrics: Contains many structures used for handling the statistics.

3.5.7. Page migration procedure

Once the control invokes the do_great_migration function, the following procedure will be followed:

1. The hash table with the possible remote accesses is used to fill an array with all the candidate addresses. This must be done because the candidates of type L3 miss might be accesses to local memory or accesses to the memory of a remote node, so it is necessary to retrieve further information on the nature of the access. This is accomplished by calling the move_pages function in query mode, where it returns an array with the NUMA node where the page is currently located.

2. With the owner node and page access information, the function will decide whether the page should be moved or not. In the affirmative case its address will be placed in an array, and a subsequent call in move mode will be made.

3. The new location of the pages that were moved will be checked in order to make sure that the page was moved.

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

1. A new descendant of the *PerformanceMonitor* class 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* method 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.

4. Benchmarking, environment setup and characterization of the Sandy Bridge Performance Measurement Unit

This chapters aims to introduce the environment used in the execution of the developed solution and the benchmarks that will be used as supervised programs to test the functioning of such solution. The second part of this chapter aims to present the first experimental results that seek to explain how the setting of the sampling configuration influences the obtained samples.

4.1. Testing environment

The testing unit consists of a host with two NUMA nodes joined by a QPI link. Every node is an Intel Xeon E5 compliant to the Intel Sandy Bridge-EP microarchitecture. Every node contains 8 physical cores, where every core has a clock speed of 2.6 GHz. However, because of the Hyper threading available in this processor, the operating system sees 16 cores for every NUMA node, making it 32 cores in total. The host altogether has a RAM capacity of 128 GB. All the tests were performed under the 3.16 version of the Kernel.

4.2. Employed Benchmarks

4.2.1. Distgen

The first algorithm to test aims to explore the behavior of the page movement functionality in a simple application which accesses a memory location. The simplest scenario would be to implement a loop that traverses a consecutive memory space sequentially and use the SPM tool to evaluate the performance of the developed algorithm. The objection to this idea is that because of the hardware prefetcher present in most of the modern processors, this application will execute very quickly. The first modification would be to access a random memory location in the memory space for each iteration, instead of the sequential access proposed first. The random location can be chosen by making use of the **random** call, present in the stock C library. The traversal executes slower than the sequential access as predicted, but the problem that was found is that the C random function has locks in its implementation that become a considerable important bottleneck.

Because of the implementation with locks of the C function, what is needed is a random number generator faster than the stock random function. It does not matter if the random generation is less strict, because this generation is only needed to avoid engaging the hardware prefetcher and for this purpose **Distgen** comes into play. Distgen, coded by Josef Weidendorfer, whose availability is presented in code listing *original distgen*, is an algorithm that transverses a memory area in a sequential or pseudo random manner where the latter mode is executed with low overhead. The execution command for distgen is as follows:

```
distgen [-p] [n-iterations] [size]
```

Where the presence of the -p indicates that the traversal will use the pseudo random sequence, when not present it will transverse the space in a streaming fashion. The optional parameter n-iterations determines how many times the space will be traversed and the size parameters determines the size of the domain to iterate through. If any of these parameters is not specified the default will be applied.

4.2.2. LAMA

4.3. Characterization of an Intel Sandy Brige Performance Measurement Unit

4.3.1. Latency weight distribution

The sample latency distribution show the distribution of the latency values when the sample is taken under a certain period and minimum weight threshold, where every bar corresponds to a specific latency weight threshold and all the bars on the same picture correspond to the same sampling period. The weight of every sample is placed in its corresponding interval and every color corresponds to the percentages of samples belonging to that interval. The results are shown in figures X and Y, which correspond to the periods of 20 and 200. The other figures are omitted because of an almost identical distribution to the ones shown

It can be observed how the working of the minimum weight threshold works: for all the distributions with a weight under 250, it can be seen how the samples with weights between 200 and 150, 250 and 300 and 300 and 350 dominate the distribution but with filtering values above that value the samples in those intervals no longer appear. 2. The distribution of the samples is very similar throughout all the period, which shows that it is independent of the sampling period.3: In this algorithm memory latency is an important problem, taking into account that local memory request are served with latencies of tens of clock cycles, at most.

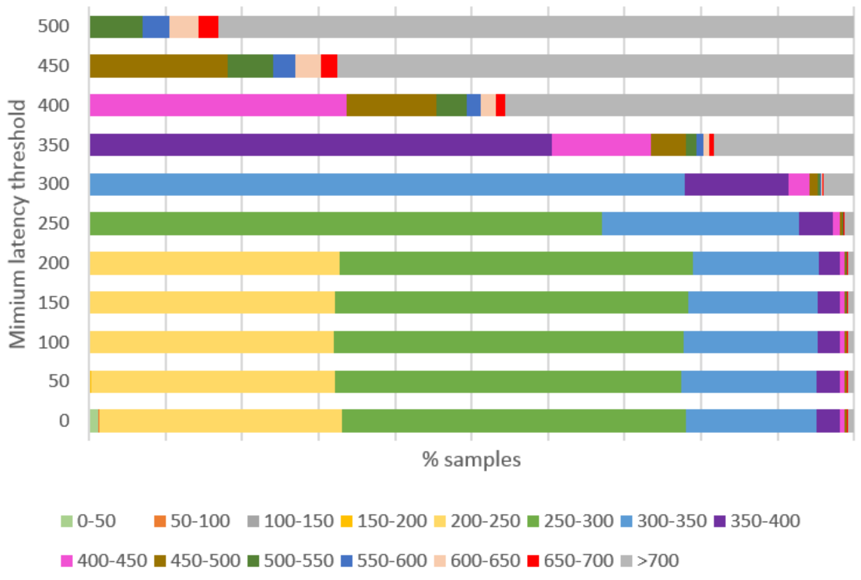

Figure 4.1.: Sample weight distribution with a period of 20 instructions per sample with different minimum weight thresholds.

4.3.2. Number of obtained samples

The sample distributions ommit a very important type of information, which is the number of samples which are acquired for every period and weight threshold combination. Knowing how many samples is important, because a high number of samples helps the SPM tool form a good idea of the memory access behavior of the observed

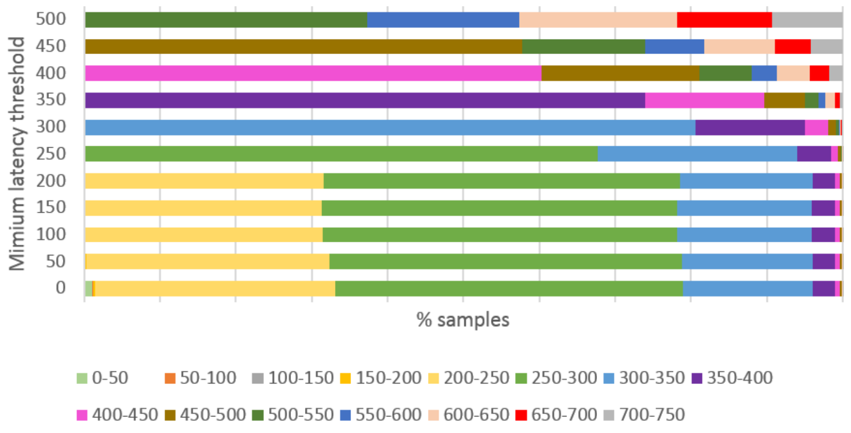

Figure 4.2.: Sample weight distribution with a period of 20 instructions per sample with different minimum weight thresholds.

process, but on the other hand every sample received must be processed which in turn implies CPU consumption which could be taken away from the supervised process.

Figure X shows how the number of samples taken reacts to changes in the sampling period and minimum weight threshold. Because a higher weight threshold implies discarding a bigger number of samples under a certain value this, relationship decreases. This experiments also confirms that a higher period is directly related to an increasing number of samples.

4.3.3. Frequency of page access

Figure X shows how the number of page accesses are distributed in a distgen run, this means that for an execution run of the algorithm a certain percentage of the pages were accessed within the given interval, this information is highly dependent on the nature of the algorithm observed. It can be seen that most of the pages were accessed between 3 and 10 times. This information is of very essential importance because it is necessary to decide after how many accesses it is possible to consider a page a certain remote or local access.

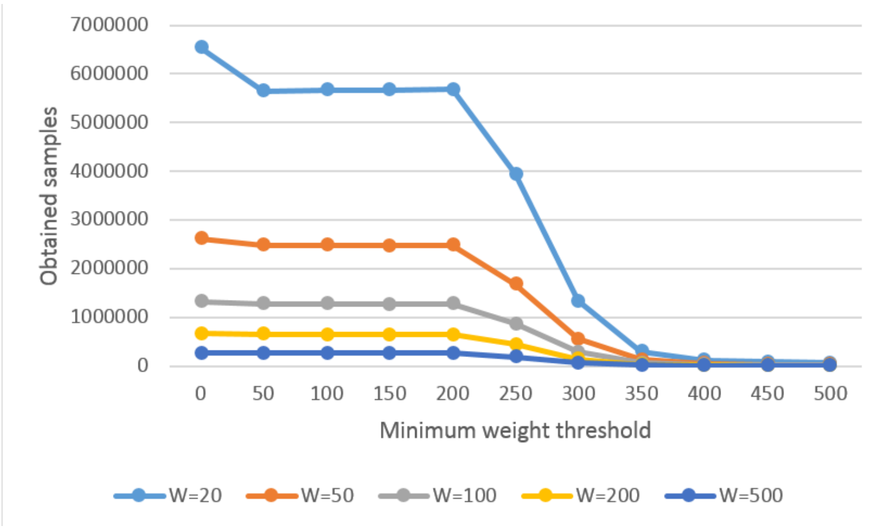

Figure 4.3.: Number of samples obtained for different weight settings

4.3.4. Completion time of the move pages call

The move_pages directive plays a very important role in the SPM tool because it provides the means to reallocate one page from one node to another, as well as querying in which node a given page, or pages reside in Figure XYZ shows the behavior of such call as function of the number of pages to move. Two scenarios are measured: the first one is the relocation of the pages in conditions of such process running exclusively on the system and the other is the same relocation together with a distgen process using all the cores, which represents a congestion scenarios. It can be seen that for a small number of pages this behavior is almost lineal but it meets an inflection point where the function completion time tends to grow smaller. The effects of the congestion can be seen since for most of the measurements the function takes longer to complete. In more elaborate scenarios as the ones shown here the function could take longer to complete, figures in the order of four seconds were measured with scenarios of redirections in multiple directions with sizes of 200 thousand samples.

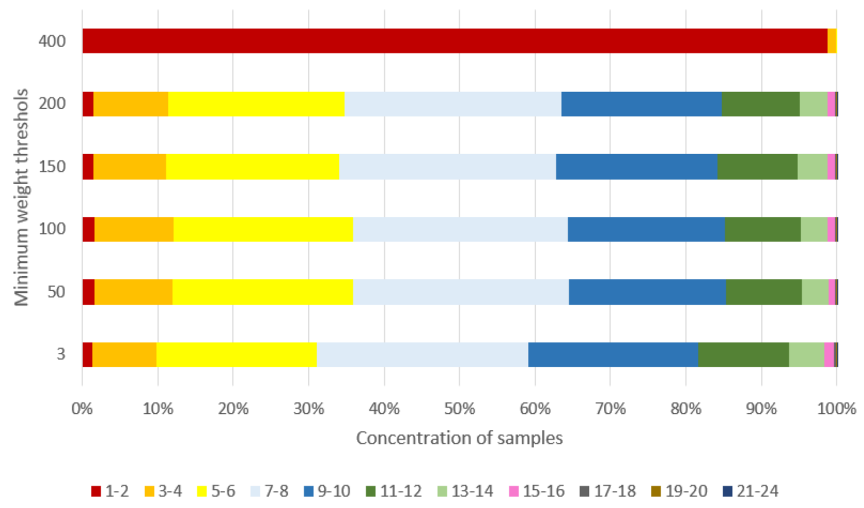

Figure 4.4.: Percentage of pages that were accessed a given number of times

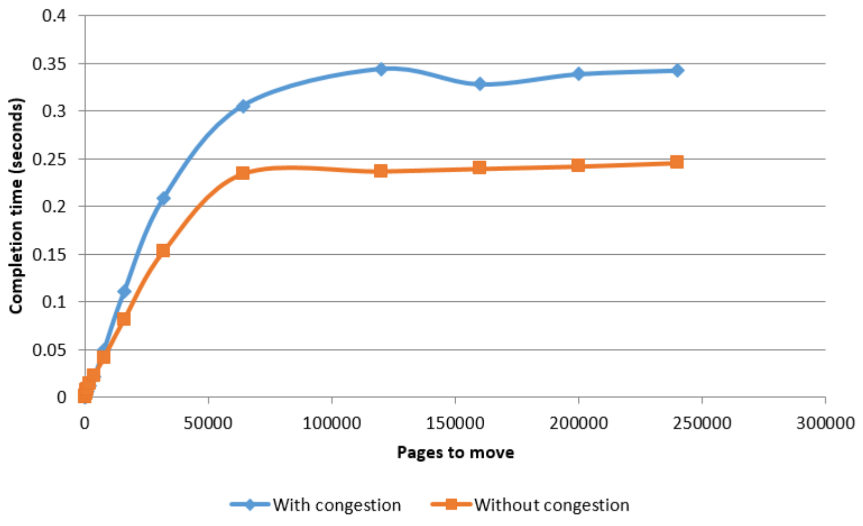

Figure 4.5.: Completion times for the move pages call for different page count under contention and no load scenarios

5. Results and analysis

5.1. SPM and distgen

This section will discussed the different scenarios that will be used for the testing of distgen as observed process. For an overview of Distgen, please refer to section 4.2.1.

5.1.1. SPM and distgen with two threads

Scenarios under consideration

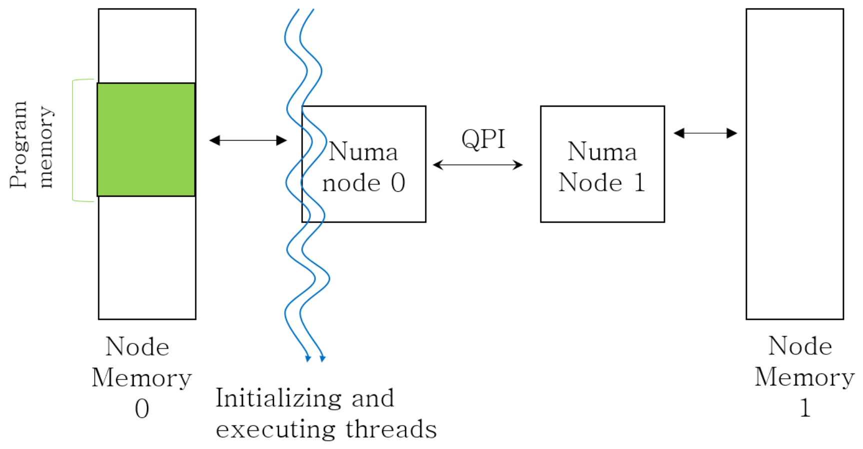

Figure 5.1.: Depiction of the distgen vanilla local with two running threads. Both threads are in the same NUMA node, therefore only local accesses will be performed.

- The **vanilla local scenario** is the code in its original form, two threads are run in the same NUMA node.

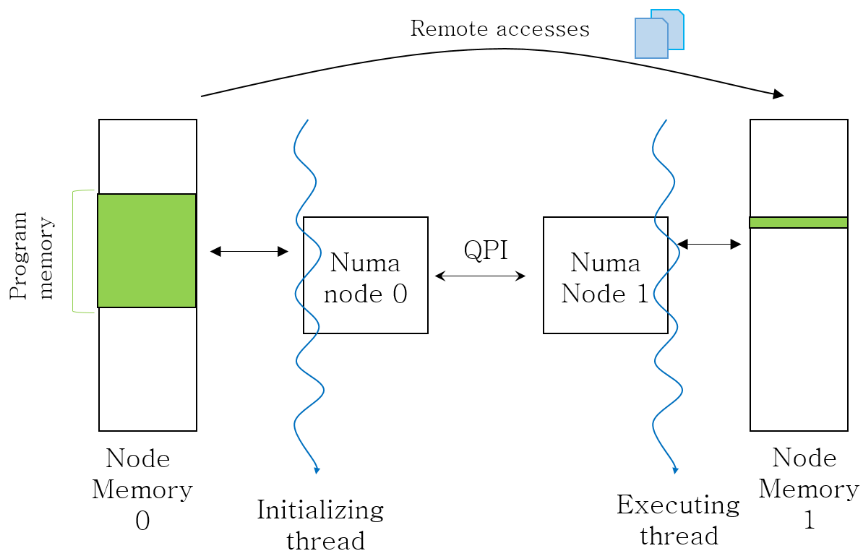

Figure 5.2.: Depiction of the distgen vanilla remote with two running threads. Each thread is pinned to a different NUMA node, which causes a lot of remote accesses to be done by the running thread.

- The **vanilla remote scenario**: Each thread runs on a separate NUMA node. The first thread initializes the memory space and the second thread accesses it.
- The **SPM scenario** is the same as the vanilla remote scenario but this time the program execution will be overseen by the SPM tool and the pages observed to be accessed more from a remote node are moved to the node that originates the greater number of memory accesses.
- The **moveall** scenario is a distgen modification based on vanilla remote that after a determinate time of execution will move all the memory pages from a remote node to a local node.

Every one of this scenarios has a specific significance for the measurement of the performance: The vanilla local scenario shows the maximum speed possible for the given algorithm, Vanilla remote will show how much performance deteriorates when the algorithm is forced to perform many remote accesses in order to fetch the required data, moveall is a reference scenario that tells how much it is possible to fix the performance degradation by taking the pages placed in a remote node and bringing them close to the executing node and SPM also tries to fix the slowdown caused by the remote placement of the pages, but since the SPM does not know about the internal implementation of the supervised process, it will only move the pages that the PMU reports as remote accesses. The closer SPM's results are to the moveall scenario, the best performing the SPM tool is. Figures 5.1 and 5.2 present depictions of both local and remote vanilla scenarios.

In order to be able to be able to place the pages remotely a modification has to be made to the original distgen code where the initialization is done by the master thread and the access is done only by the other thread, which is not the master. To determine whether the two threads are placed in a local node the environment variable *GOMP_CPU_AFFINITY* is manipulated in order to determine the core in which the running threads will be placed. The availability of this modified two threads version is described in code listing 5 in Appendix A.

Performance for the random mode

Tables Tabelle 1 and Tabelle 2 Show the results of the execution of distgen random. Figures Abbildung 7: Normalized execution times for distgen random with two threads Abbildung 7 and Abbildung 8 show the execution times and throughputs respectively as a normalized quantity in respect to the distgen vanilla result which means that a value closer to 1 resembles a performance closer to that obtained in the local algorithm. For this random version of the distgen did a very good job with a performance very similar to the moveall situation, which is the theorethical maximum that can be archived. Another important result that appears in most of the measurements is that the performance of the moveall strategy after moving the pages never gets to be the same as that obtained in the local version, effect that has to be researched further. As for the performance of the SPM it can be seen from Tabelle 3 the percentage of caught pages remain above 40 percent with decreasing value as the domain gets bigger. This happens because with a bigger domain size the observation time occupies a smaller relative portion of the total execution time and therefore less samples can be caught.

Size	Vanilla local	Vanilla Remote	Moveall after 45s	SPM
400G	61.470	102.573	85.375	80.221
600G	91.382	154.340	124.067	134.695
800G	125.757	215.269	200.078	187.817
1T	151.291	151.291	198.249	202.729

Table 5.1.: Execution time given in seconds of the different distgen scenarios in random mode.

Size	Vanilla local	Vanilla Remote	Moveall after 45s	SPM
400G	1.12973E+08	6.77034E+07	1.00708E+08	9.68249E+07
600G	1.13992E+08	6.74924E+07	8.24585E+07	9.35968E+07
800G	1.10442E+08	6.45197E+07	7.36434E+07	7.10097E+07
1T	1.14753E+08	6.66698E+07	8.85143E+07	9.39763E+07

Table 5.2.: Execution throughput given in reads per second of the different distgen scenarios in random mode.

Size	Moved pages	Moved pages %	
400G	422971	108.280576	
600G	372289	63.53732267	
800G	413297	52.902016	
1T	423478	43.3641472	

Table 5.3.: Number of pages moved in the SPM scenario

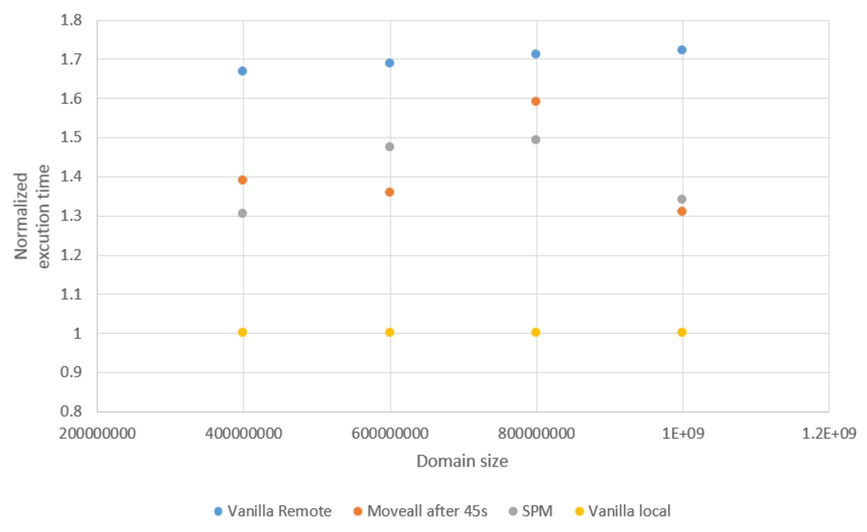

Figure 5.3.: Normalized execution times for distgen random with two threads

Performance for the sequential mode

In this opportunity the serial version of distgen is run. Because of the faster running time of this algorithm, the number of iterations is doubled to 2000 to obtain a long enough execution time. Tabelle 4 and Tabelle 5 show the execution time and performance results and the Abbildung 9 and Abbildung 10 show the normalized values with respect to distgen vanilla. In comparison to the serial version the performance of the SPM tool is not as good and only matches that of the moveall scenario in with the smallest size. Part of this slowdown can be blamed on the smaller portion of pages that the tool is getting.

Size	Vanilla local	Vanilla Remote	Moveall after 45s	SPM
400G	55.924	92.534	75.146	75.969
600G	83.894	138.851	104.407	117.433
800G	111.823	185.144	133.699	159.905
1T	139.810	231.453	163.018	199.576

Table 5.4.: Execution time for the different scenarios of the distgen sequential algorithm with units given in seconds.

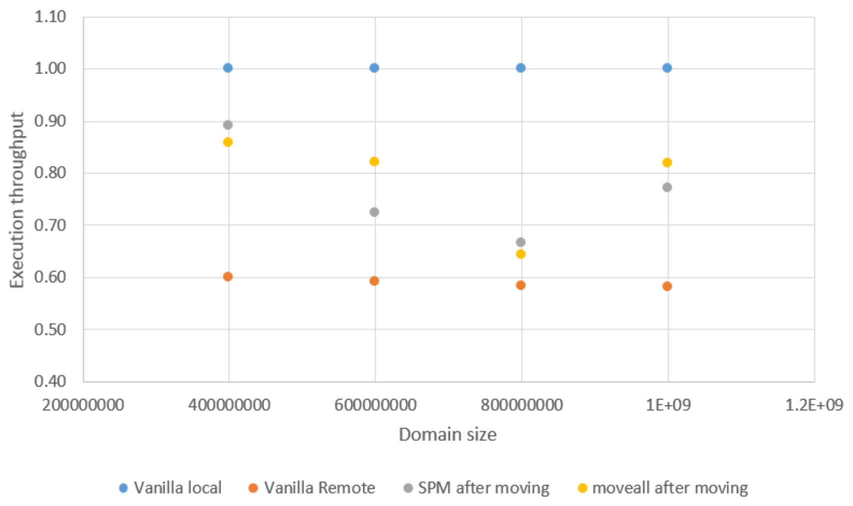

Figure 5.4.: Normalized execution throughput for distgen random with two threads

Size	Vanilla local	Vanilla Remote	Moveall after 45s	SPM
SZ1	2.48368E+08	1.50106E+08	2.37768E+08	2.37636E+08
SZ2	2.48350E+08	1.50048E+08	1.96037E+08	2.37708E+08
SZ3	2.48409E+08	1.50039E+08	1.86950E+08	1.48624E+08
SZ3	2.48352E+08	1.50023E+08	1.48624E+08	2.37542E+08

Table 5.5.: Execution throughput for the different scenarios of the distgen sequential algorithm with units given in reads per second.

5.1.2. SPM and distgen with all threads employed

The scenarios used here follow the same logic as what was shown in the previous section, but now we want to go further and try a contention scenario where more cores are running the algorithm. For this test all the cores except two were used, the two saved are used for running SPM. In the remote cases, every core is mapped to an opposite core in the opposed numa node and the data will be allocated in this remote node.

Size	Moved pages	Moved pages %
400G	232777	59.590912
600G	234551	40.03003733
800G	226748	29.023744
1T	233191	23.8787584

Table 5.6.: Number of pages moved in the distgen sequential scenario.

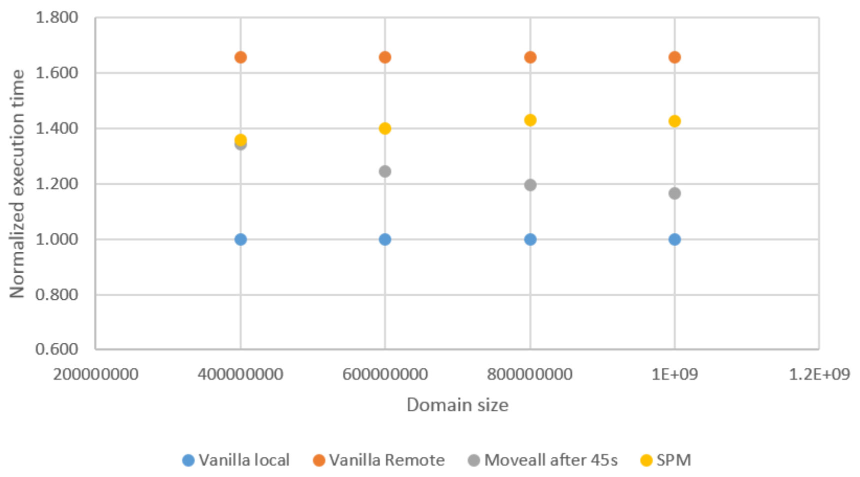

Figure 5.5.: Normalized execution speed for the different scenarios of the distgen sequential with two threads with respect to the vanilla local scenario

Performance for the random mode

Performance for the sequential mode

Measuring the effect of SPM's action

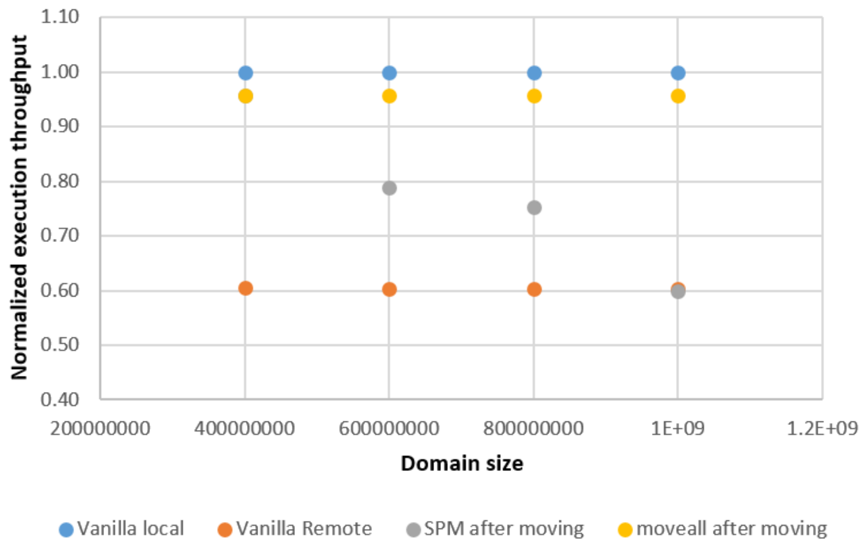

Figure 5.6.: Normalized execution throughput for the different scenarios of the distgen sequential with two threads with respect to the vanilla local scenario.

Size	Vanilla local	Vanilla Remote	Moveall after 45s	SPM
90G	83.51	196.688	152.959	155.405
110G	103.617	242.471	184.673	191.259
130G	129.617	282.619	194.029	210.215
150G	141.588	329.403	260.219	254.283

Table 5.7.: Completion time in seconds for the random distgen with 30 threads employed

Size	Vanilla local	Vanilla Remote	Moveall after 45s	SPM
90G	127.763	341.35	152.959	262.535
110G	156.145	417.683	184.673	331.794
130G	184.482	493.134	194.029	395.134
150G	212.992	569.438	260.219	465.617

Table 5.8.: Latencies and timing table

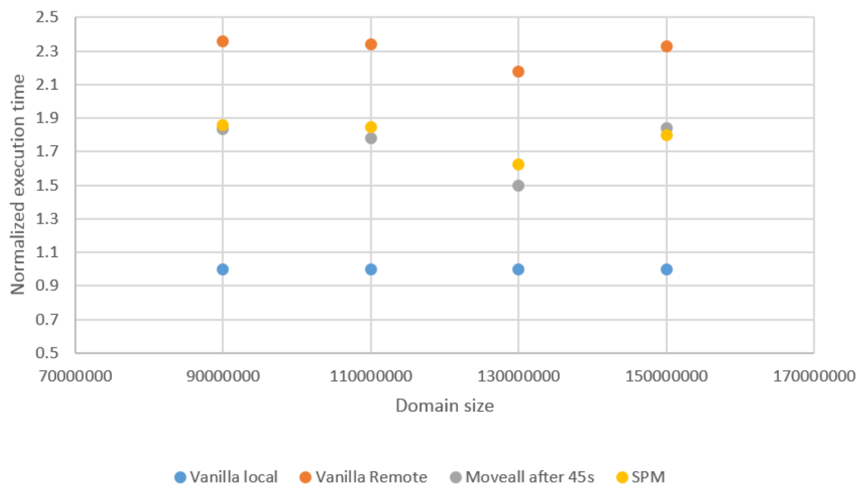

Figure 5.7.: Normalized execution time for the random distgen cases with 30 threads employed.

Size	Moved pages	Moved pages %
90G	4438270	84.16274963
110G	4556914	54.38554704
130G	3526065	46.29090462
150	4647763	52.88121458

Table 5.9.: Number of moved pages by the SPM tool for the distgen random case with all threads engaged

Size	Moved pages	Moved pages %
90G	770653	13.70049778
110G	809007	11.76737455
130G	809901	9.96801230
150	853033	9.099018667

Table 5.10.: Number of moved pages by the SPM tool for the distgen random case with all threads engaged.

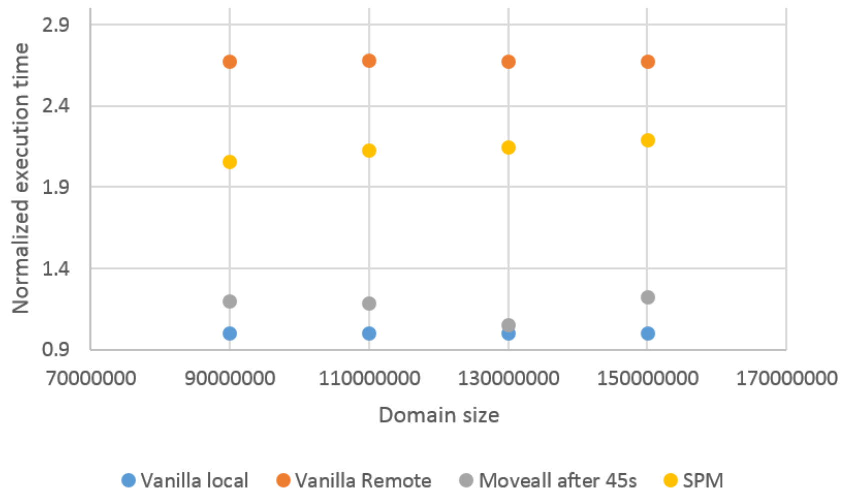

Figure 5.8.: Normalized execution time for the sequential distgen cases with 30 threads employed.

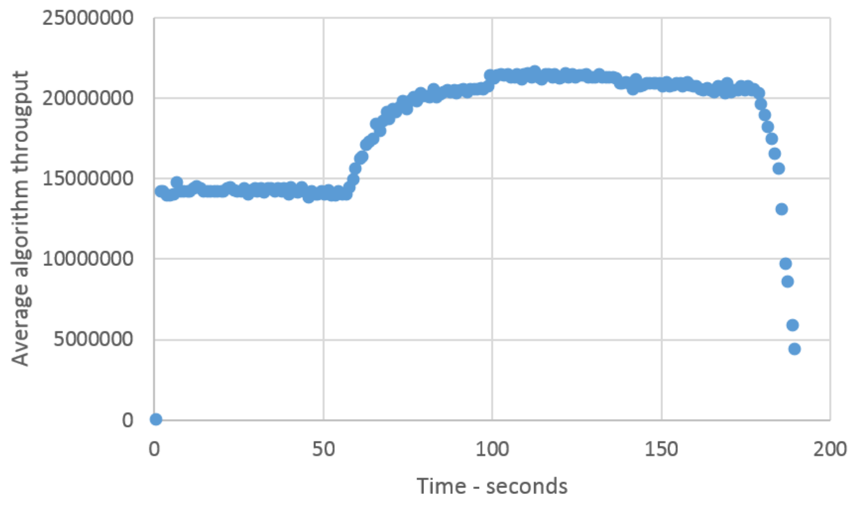

Figure 5.9.: View of the average throughput during the execution of distgen under SPM supervision. The page migration is done at time 45.

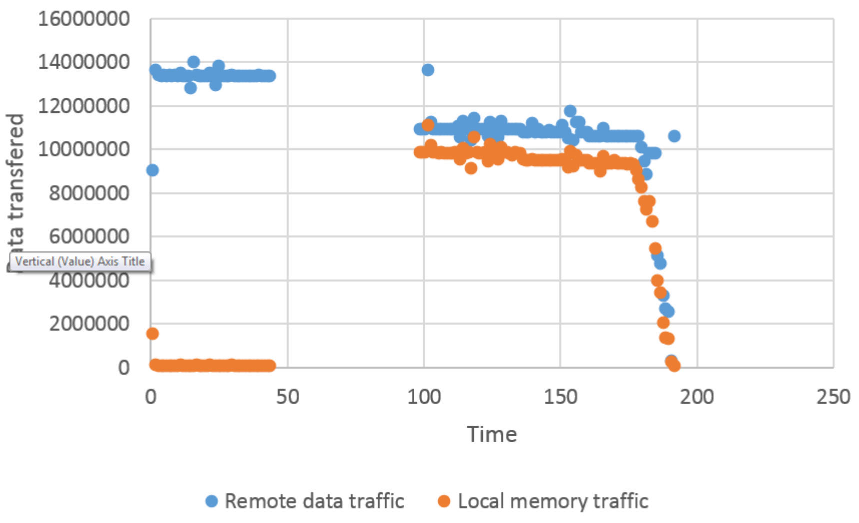

Figure 5.10.: View of the average and local and remote transfer rates during the execution of distgen under SPM supervision. The page migration is done at time 45.

6. Conclusions and possibilities of future work

A. Code resources

These chapter lists all the code resources used during the development of this project.

1. **Perf source code**: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git The kernel sources contain the implementation for both the kernel and user sides of perf. The subdirectory tools/perf contains the userside sources which have their separate make file.

2. **Numatop**

3. **Autopin+ with the SPM tool integrated**

4. **Original distgen**: https://github.com/lrr-tum/reuse.git. The original code for the distgen benchmark.

5. **Distgen based benchmarks**: https://github.com/daniel-ortiz/spm-benchmarks The repository that contains the modified distgen versions used for the measurements: Subfolder *bench-oneremote* has the code for running the benchmark with two threads, subfolder bench-allremote contains the code for running the benchmark in all the cores of the system and subfolder *movepages_time* has the code for characterizing the completion time of the move_pages function.

List of Figures

List of Tables

Bibliography

[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*. Intel Corporation, Jan. 2009. URL: `http://www.intel.de/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf`.

[Int15] Intel 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`.

[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." In: *Transactions on High-Performance Embedded Architectures and Compilers III*. Vol. 6590. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 219–235.