HAL
open science

numap: A Portable Library For Low-Level Memory Profiling

Manuel Selva, Lionel Morel, Kevin Marquet

To cite this version:

Manuel Selva, Lionel Morel, Kevin Marquet. numap: A Portable Library For Low-Level Memory Profiling. Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), Jul 2016, Samos, Greece. 10.1109/SAMOS.2016.7818331 . hal-01408979

HAL Id: hal-01408979

https://inria.hal.science/hal-01408979v1

Submitted on 5 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

numap: A Portable Library For Low-Level Memory Profiling

Manuel Selva, Lionel Morel, Kevin Marquet
Univ Lyon, Insa Lyon, Inria, CITI
F-69621 Villeurbanne
France
Email: firstname.lastname@insa-lyon.fr

***Abstract*—The memory subsystem of modern multi-core architectures is becoming more and more complex with the increasing number of cores integrated in a single computer system. This complexity leads to profiling needs to let software developers understand how programs use the memory subsystem. Modern processors come with hardware profiling features to help building tools for these profiling needs. Regarding memory profiling, many processors provide means to monitor memory traffic and to sample read and write memory accesses. Unfortunately, these hardware profiling mechanisms are often very complex to use and are specific to each micro-architecture. In this work, we present `numap`, a library dedicated to the profiling of the memory subsytem of modern multi-core architectures. `numap` is portable across many micro-architectures and comes with a clean application programming interface allowing to easily build profiling tools on top of it.**

I. Introduction

The increasing number of cores sharing memory in a single computer system leads to the so-called *memory wall*. The memory subsystem can not satisfy the simultaneous requests of all the cores and thus becomes a serious performance bottleneck. To alleviate this issue, computer architects have proposed many changes to memory architectures [1]. In particular, new cache levels have been added to processors and new distributed shared memory architectures, also known as Non Uniform Memory Access (NUMA) architectures, have been designed. However, these innovations have increased the burden on the software for getting the best possible performance out of the hardware [2], [3]. In other words, the software must be aware of the underlying memory organization to efficiently exploit these complex memory architectures. Nevertheless, statically knowing the memory architecture details is not enough to exploit it efficiently. In particular, the dynamism of modern operating systems and applications leads to a form of non-determinism regarding the memory hierarchy usage. As a consequence, software developers need runtime profiling mechanisms to deeply understand how the software interacts with the memory and to identify memory bottlenecks. To allow low-level performance measurement, modern processors now provide hardware profiling mechanisms often referred to as Performance Monitoring Unit (PMU). Intel, AMD and ARM all include a PMU in their processors.

This work was partly supported by the Auverge-Rhone-Alpes Region, through the COOPERA Program, under the project number 16768.

As far as memory is concerned, PMUs usually allow for two modes of operation. A first mode permits to count the number of memory requests that reach the memory controllers. From this count, one can monitor the memory bandwidth. A second mode, named sampling, periodically records samples that can contain more information about the initiator of the memory request, typically the address of the instruction that initiates the request, the level into the memory hierarchy where the data was found and information about the latency of the access.

Unfortunately, using these profiling mechanisms is a daunting task because of low-level concerns. In particular, it requires writing micro-architecture dependent code to assign the correct model specific registers (MSR) with the correct configuration values. This code can be different even for different micro-architectures from the same processor vendor.

From the programming point of view, the increasing complexity of hardware platforms has led to the emergence of numerous concurrent programming models. Custom profiling tools are then needed, that ought to be aware of the way the application is written. In order to provide useful performance information to the application developer, low-level profiling events should be linked to the programming model. An abstraction of the memory profiling capabilities of the hardware is thus required as a basic building block of these high-level profiling tools. This will allow to isolate and alleviate the difficult task of using the PMU from the other difficult task of correlating performance samples with programming models. Several existing libraries provide programming abstractions on top of the PMUs, but none of them handle the sampling of memory requests. Moreover, none of these approaches propose a portable way of monitoring memory bandwidth using the PMU in counting mode.

The main contribution of this work is the `numap` low-level memory profiling library. `numap` stands for Non Uniform Memory Access architectures Profiling. `numap` abstracts the common memory profiling mechanisms exposed to software by processors from different vendors by providing a powerful yet simple interface. It is an open source library [1] that already supports many micro-architectures from Intel. `numap` has been initially thought and designed for profiling memory usage

[1] https://github.com/numap-library/numap

on NUMA architectures but it can definitely be used for memory profiling in centralized shared memory systems.

The remainder of the paper is organized as follows. We describe in Section II PMUs and how to use them. In Section III we present related work. We then motivate the proposal of the `numap` library in Section IV. Section V gives the details of the library interface and implementation while Section VI presents an example of using the library for profiling programs written in a high-level dataflow programming language. Finally, we conclude and describe future works in Section VII.

II. Background

In this section we first describe the different features provided by PMUs. Then we present how to access these features.

A. Performance Monitoring Unit

A PMU provides means to characterize hardware usage through hardware performance counters. These counters can be configured by software to count some specific hardware events among a huge number of possibilities. As shown in Figure 1, counters are either located at the level of cores or at the level of memory controllers. At core level, examples of hardware events that can be profiled are clock cycles, number of floating point instructions, number of level 1 cache misses or number of branch mispredictions. At memory level, counters can be configured to count the exact number of memory read or memory write requests.

In addition to this *counting* mode, most PMUs provide a *sampling* mode for events at core level. Intel's technique for sampling is called Precise Event-Based Sampling (PEBS) [4] and AMD's is called Instruction-Based Sampling (IBS) [5]. On Intel platforms, when sampling, instead of counting the occurrences of a specific event, the PMU is configured to generate a sample with detailed information every time the event occurred a specified number of times. On AMD platforms, IBS allows sampling all instructions independently of a particular event. On both Intel and AMD platforms, information that can be recorded in samples includes the address of the instruction that has generated the event and, in the case of memory-related events, the targeted memory address, the memory level that served the memory access and latency of the memory access. From this information, one can associate low-level information about memory accesses to source code.

B. Performance Monitoring Unit Usage

Using hardware monitoring counters requires a deep understanding of the processor's architecture and requires to write very low-level code. On Intel and AMD architectures, the PMU is accessed through Model Specific Registers (MSR) that can be written in processor's supervisor mode only. Figure 2 illustrates various ways to access these registers on top of Linux.

First, a kernel module can be used, such as the `/dev/cpu/msr` standard module. This module only allows to access the PMU configuration registers in supervisor mode, it does not provide any abstraction.

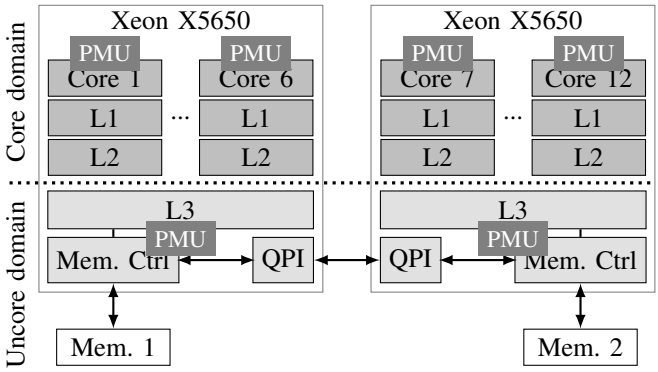

Fig. 1. PMU on an Intel dual processors NUMA architecture. Some performance counters are located at Core level, and others at Uncore level.

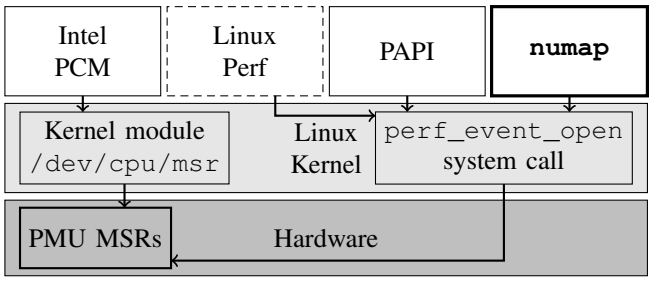

Fig. 2. Different ways of accessing the PMU from code. On top of Linux, the PMU MSRs can be accessed either through the `perf_event_open` system call or through a kernel module. Compare to Intel PCM, PAPI and `numap`, Linux perf is not a library but a command line tool.

Second, the `perf_event_open` system call can be used. It was introduced to support the Linux perf command line profiler [6], and provides a first level of abstraction to access hardware performance counters. This system call abstracts the need to manually write bits to MSRs to start and stop hardware profiling. Nevertheless, much work is still required from the developer willing to profile his/her applications.

For both solutions, the first difficulty consists in finding which events corresponding to memory accesses to count or to sample. The PMU events to be used are obviously different across processor vendors, but they are also different between micro-architectures from the same vendor. Finding them requires to step into the specific documentation for the targeted micro-architecture and to understand the low-level details of the underlying hardware. This task is daunting and time consuming.

In the case of `perf_event_open`, another important difficulty lies in understanding how to set-up all the parameters to be passed and which other system calls need to be invoked. To support this claim it is worth mentioning that the corresponding man page[2] is 1583 lines long. Figure 3 shows the signature of `perf_event_open`. Quoting the man page, "*the `pid` and `cpu` arguments allow specifying which process and CPU to monitor. The `group_fd` argument allows event*

[2] http://man7.org/linux/man-pages/man2/perf_event_open.2.html

```
int perf_event_open(struct perf_event_attr *attr,
                    pid_t pid, int cpu,
                    int group_fd,
                    unsigned long flags);
```

Fig. 3. `perf_event_open` system call signature. Parameters specify what and when to count or to sample. The system call returns a file descriptor to be used for reading count value or access recorded samples.

groups to be created". Note that the `pid` argument is not a process id but a thread id. The `flags` argument is mainly used to associate a new event to an existing group. Finally, and quoting the man page again, the `attr` structure argument provides "*detailed configuration information for the event being created*". This structure contains all the hardware-dependent information, in the form of raw numbers, the meaning of which depends on the micro-architecture. The system call returns a file descriptor to be used for reading count values or access recorded samples. When sampling is required, the user code that calls `perf_event_open` must use the returned file descriptor to allocate memory where the kernel will record samples. When this allocated memory is full, the kernel can not record samples anymore until user code indicates which samples it has already read. Section IV shows in detail the complexity of using `perf_event_open` to sample memory reads on a particular Intel micro-architecture.

III. Related Work

This section first focuses on work regarding how to use the PMU. We then review existing profiling tools that use memory sampling capabilities.

A. Libraries For Accessing PMUs

To hide either the complexity of `perf_event_open` or the one of using directly the `/dev/cpu/msr` kernel module, two libraries are widely used today as shown in Figure 2. Their main goal, is to provide abstractions for common profiling mechanisms based on the PMU. The first library is Intel PCM [7]. It is available for both Linux and Windows. Nevertheless, it does not provide access to memory sampling features and as the name suggests is designed specifically for Intel processors. The second widely used library aiming at simplifying the use of PMUs of different architectures is PAPI [8]. It provides an abstraction for the common counters found on all the targeted architectures in such a way that the programmer only needs to say, *e.g.,* "*I want to count the number of instructions retired*". Unfortunately, as Intel PCM, the current PAPI version[3](5.4.3) does not provide any way to use the memory sampling capabilities of modern PMUs. Moreover, counting memory requests using PAPI requires to know which hardware event to use for every targeted micro-architecture. A work proposing the addition of sampling capabilities in PAPI has been published very recently [9]. The main difference between this work and our proposed `numap` library concerns the interface exposed to user code. The authors of this work advocate for a low-level interface - "*PAPI_sample_init() is intended as a low-level interface, we have left setting up the buffer, handling the interrupt, and parsing the buffer to the user*" - while we advocate for a higher level of abstraction to let developers using `numap` focus only on analyzing the data and not on how to get the memory samples. Moreover, to the best of our knowledge, this PAPI extension is not open source and its official integration into PAPI is not planned yet.

[3]http://icl.cs.utk.edu/papi

B. Profilers And Runtimes Using Memory Sampling

We now review existing profiling and runtime software approaches that are based on memory sampling capabilities. These tools are either non portable or require very expensive work to support different micro- architectures. All of them, except the Linux perf profiler, could benefit from `numap` to ease their development and to increase their portability.

The Linux perf [6] command line profiler provides a memory sampling feature. It is portable on all the processors supported by Linux. perf is based on the `perf_event_open` system call and hardware dependent code is developed by Linux kernel developers for supporting various models. It is a very complex tool designed to be used by end users. The only provided interface is the command line, no Application Programming Interface (API) is provided. As a consequence, compared to `numap`, it is impossible to build programming model aware profilers on top of perf.

Memphis [10] and MemProf [11] are memory profilers dedicated to identify remote memory accesses in parallel applications running on NUMA architectures. These tools work only for AMD processors and required the development of a specific Linux Kernel module dedicated to memory sampling on AMD processors based on IBS [5]. Carrefour [12] is a runtime extension of MemProf integrated into the Linux kernel memory management subsystem. Its main goal is to limit remote memory accesses on NUMA architectures and to alleviate memory load imbalance. As MemProf, Carrefour required expensive developments only for running on AMD processors. HPCToolkit is a profiler that has been recently extended to support memory profiling [13]. These extensions were required to identify memory bottlenecks in parallel applications. This profiler is portable on different micro-architectures. It relies on PAPI for the profiling aspects not related to memory and relies on micro-architecture dependent code for memory sampling. ScaAnalyzer [14] is another extension to HPCToolkit allowing to detect memory scalability bottlenecks. It also relies heavily on memory sampling, and required a lot of development to support different micro-architectures. The development of all these tools could be easier using the `numap` library.

IV. Motivation

As stated in the previous section, none of the existing PMU libraries provide high-level abstractions for memory sampling. More specifically, they do not provide any abstraction for techniques like Intel PEBS and AMD IBS. Developers willing

```
// Set system call parameters
struct perf_event_attr attr;
attr.config = 0x100b;
attr.config1 = 3;
attr.sample_period = 20000;
attr.sample_type = PERF_SAMPLE_IP
    | PERF_SAMPLE_ADDR | PERF_SAMPLE_WEIGHT
    | PERF_SAMPLE_DATA_SRC;
attr.precise_ip = 2;
attr.mmap = 1;
attr.task = 1;
attr.exclude_kernel = 1;
attr.exclude_hv = 1;
attr.disabled = 1;

// Make the call
int fd = perf_event_open(&attr, TID, -1, -1, 0);
```

Fig. 4. `perf_event_open` system call complexity. This example shows how to sample memory loads for a single thread with id TID.

to analyze memory usage and performance thus have to use the `perf_event_open` system call or MSRs to setup memory profiling. Again, using any of these two solutions requires to find which event to profile in the documentation of the targeted processor and to write a lot of setup code. We now describe step by step how to do memory sampling using `perf_event_open`.

Once a developer knows which event he/she needs to use for memory sampling using `perf_event_open`, the first thing to do is to setup the parameters and then to effectively make the call using these parameters. Figure 4 illustrates this task. Line 3 specifies the memory load event that must be recorded for the Intel Westmere micro-architecture in this case. This event is part of the several hundreds of events documented in processors' datasheets (Intel's Software Developer Manual [4] chapter 18 for Intel processors). Line 4 specifies that a sample must be recorded only if the latency of the read access is above 3 cycles (which is the minimal latency for a memory load on the considered Westmere platform). Then, line 5 indicates the rate at which sampling must occur. This rate is in number of events. In this example, we record a sample every 20000 memory loads. In practice, the sampling rate will depend on the hardware cost of sampling memory accesses and on the acceptable overhead. The acceptable overhead for profiling tools dedicated to offline analyzes should be greater than the one for runtime decisions based on memory sampling. Lines 6 to 8 indicate which information must be recorded in each sample. Lines from 9 to 14 configure other details about the memory sampling. In these lines we indicate that we want precise recording of the instruction address, that we want to record `mmap` information for offline symbol decoding and that we exclude kernel and hypervisor code from sampling. Line 17 finally performs the system call and we get a file descriptor as a result. In this example, the kernel is told to record samples only for the Linux thread with the given `TID` identifier.

Once the `perf_event_open` call has been issued, user code must map the memory containing the records in its address space. User code must also effectively start the sampling using another system call, namely `ioctl`. Figure 5 shows how to do these two tasks. The `mmap` system call must be used with the file descriptor previously returned by `perf_event_open` with a particular size and with the appropriate protection arguments. As shown on line 6, the size of the mapped memory area must be in the form $1+2^n$ *pages* where *pages* is the system default page size. Finally, the `ioctl` call should be made twice with two specific perf flags to first reset and then start the sampling (lines 10 and 11).

```
1  // Map result, size should be 1+2^n pages
2  size_t mmap_pages_count = 2048;
3  size_t p_size = sysconf(_SC_PAGESIZE);
4  size_t mmap_len = p_size*(1 + mmap_pages_count);
5  struct perf_event_mmap_page *metadata
6      = mmap(NULL, mmap_len, PROT_WRITE,
7             MAP_SHARED, fd, 0);
8
9  // Start sampling
10 ioctl(fd, PERF_EVENT_IOC_RESET, 0);
11 ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);
```

Fig. 5. Map in user space the result of `perf_event_open` system call and start sampling. There are constraints on the size of the mapped memory.

```
1  #define rmb() asm volatile("lfence":::"memory")
2
3  // Stop sampling
4  ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
5
6  // Analyze the samples
7  uint64_t last_samp = metadata -> data_head;
8  rmb();
9  struct perf_event_header *header =
10         metadata + p_size;
11 uint64_t consumed = 0;
12 while (consumed < last_samp) {
13   if (header->type == PERF_RECORD_SAMPLE){
14     // Do something with the sample which
15     // fields start at address header + 8
16   }
17   consumed += header->size;
18   header = (struct perf_event_header *)
19            ((char *)header + header -> size);
20 }
```

Fig. 6. Stop sampling and analyze the results. The iteration over the samples recorded using `perf_event_open` requires dealing with low-level concerns.

When user code wants to stop sampling, an `ioctl` call must be performed (symmetric to the one to start sampling) as shown in line 4 of Figure 6.

To analyze the samples, extra precautions should also be taken as shown by lines 7 to 19 in Figure 6. First (line 7) we must get the position of the last sample written by the kernel to know when we have to stop reading the samples. Then, a memory barrier instruction should be issued (line 8) as specified in the `perf_event_open` man page ("*On SMP-capable platforms, after reading the data_head value, user space should issue an rmb()*"). Lines 9 and 10 get the address of the header of the first sample. From then on, we can iterate over samples whose body is located at address `header + 8`

as shown by lines 12 to 19.

V. The Library

We propose the numap library to alleviate developers from knowing which exact hardware events need to be used for monitoring memory bandwidth and sampling memory requests. numap also remove most of the burden described in the previous section when using perf_event_open. By automatically choosing the correct event for the underlying hardware, numap provides a portable support for low-level memory profiling.

A. Application Programming Interface

The main functions of the numap API are shown in Figure 7. These functions count memory requests, generate memory samples and provide access to them for analyzing memory behavior of applications.

Because numap relies on specific hardware components and operating system configurations, it can sometimes be complex just to know whether or not memory sampling is available on particular machine. To alleviate this problem, all the numap functions return an integer for error management. From this error code, user code can get a human readable message using the error_message function. The message may indicate that the micro-architecture does not provide memory sampling, that numap does not support yet the micro-architecture or that the system policy does not allow memory sampling along with information about how to enable it.

All functions except the error management one, take a custom structure as a parameter that is used both to provide results to the caller and to internally keep track of the events that have been activated. There is one structure for functions related to sampling and another one for functions related to counting. The definitions of these structures are shown in Figure 8.

Regarding sampling, this structure is called samp_session. It contains a list of thread ids. User code must fill this list to specify the threads to be profiled before passing the structure to functions starting profiling. The samp_session structure is also used to specify the sampling rate and the number of pages to allocate for saving the samples (this number is used to compute the size in the correct form which is then passed to the mmap call described in the previous section). The current version of numap only allows to save samples until the specified size is filled. As stated in Section VII, we plan to allow the profiling of very long lived applications. The init_samp_session function is provided to setup all the profiling parameters at once.

The samp_read_start and samp_write_start functions internally performs the mmap system call described in the previous section. There may be several calls in case the user code specifies several threads to be profiled. The result of each of them is stored in the metadatas list of the samp_session object passed as parameter.

```
// Init sampling parameters
int init_samp_session(
    struct samp_session *ses, int nb_threads,
    int sampling_rate, int mmap_pages_count);

// Start and stop memory sampling
int samp_read_start (struct samp_session *s);
int samp_read_stop  (struct samp_session *s);
int samp_write_start(struct samp_session *s);
int samp_write_stop (struct samp_session *s);

// Analyze the samples
int print_rd(File *f, struct samp_session *s);
int print_wr(File *f, struct samp_session *s);
int cpy_samples(struct samp_session *s,
             struct samp **dest, int *nb_sp);

// Init counting parameters
int init_count_session(
    struct count_session *ses, int nb_nodes,
    int *nodes);

// Start and stop counting memory traffic
int count_read_start (struct count_session *s);
int count_read_stop  (struct count_session *s);
int count_write_start(struct count_session *s);
int count_write_stop (struct count_session *s);

// Get count value
long get_count(struct count_session *s);

// Get human readable error message
const char *error_message(int error);
```

Fig. 7. The main functions of numap API

Finally, the samp_session structure contains internal fields that should only be read and written by numap itself. These fields are needed to handle all the details described in Section IV.

For counting, the structure is called count_session. It contains the list of NUMA nodes to be profiled. As for thread ids in sampling mode, user code must fill this list to specify the nodes to be profiled. The structure also contains a file descriptor for each profiled NUMA node. When counting has been stopped, user code gets the actual count value by calling the get_count function that use this file descriptor.

As shown in Figure 7, numap provides functions to start and stop profiling both for sampling and counting. The starting functions configure and start profiling according to the parameters contained in the structure object passed as parameter. In both modes, numap differentiates memory read and memory write profiling. The reason for that is the hardware, PMUs provide different events for read and write profiling.

numap also provides functions to let user code easily access the samples generated by the library. All these functions take as input the samp_session structure to avoid the user code having to deal with low-level details. The library provides functions allowing to print the samples generated by numap either in a file or on the console. To perform custom work on the generated samples, numap provides the cpy_samples function. This function produces a ready-to-use list of simple

```
// Structure for a sampling session
struct samp_session {

  // Fields to be written by user code
  // before using numap functions
  unsigned int nb_threads;
  pid_t *tids;
  unsigned int sampling_rate;
  unsigned int mmap_pages_count;

  // Field to be read by user code
  // where to find the samples
  struct perf_event_mmap_page **metadatas;

  // Fields to be written and/or
  // read internally by numap only
  size_t page_size;
  size_t mmap_len;
  long *fd_per_tid;
};

// Sample structure
struct samp {
  uint64_t ip;
  uint64_t addr;
  uint64_t weight;
  union perf_mem_data_src data_src;
}

// Structure for a counting session
struct count_session {

  // Fields to be written by user code
  // before using numap functions
  unsigned int nb_nodes;
  int *nodes;

  // Fields to be written and/or
  // read internally by numap only
  long *fds;
};
```

Fig. 8. The numap API data structures. The structure samp_session is used by user code to specify to numap what should be profiled and to get the result. The structure samp represents a single sample.

sample objects. This list is then accessible through the dest pointer. This list is dynamically allocated on the heap by numap, and its size is returned to user code through the nb_sp parameter. Each element of this list is of type struct samp. As shown in Figure 8, this structure contains the address of the instruction that generated the sample, the targeted address, the latency of the access (weight) and the memory level that served it. These values are the fundamental information needed to build appropriate profiling or runtime mechanisms such as the one described in Sections III-B and VI. This cpy_samples function alleviates the user code from the details described in the previous section required to read the memory filled by the kernel.

Figure 9 summarizes the usage of numap for sampling read memory accesses. Using numap, user code only issues 3 function calls with simple parameters. It must first call the init_samp_session function (line 3) and specify the identifiers of the threads to be profiled (line 8). In this example we are profiling only the current thread whose

```
1  // Init read sampling
2  struct samp_session s;
3  res = init_samp_session(&s, 1, 10000, 64);
4  if(res < 0) {
5    printf(``err:%s\n``, error_message(res));
6    return -1;
7  }
8  sm.tids[0] = syscall(SYS_gettid);
9
10 // Start memory read access sampling
11 res = samp_read_start(&s);
12 if(res < 0) {
13   printf(``err:%s\n``, error_message(res));
14   return -1;
15 }
16
17 // Code to be profiled here
18
19 // Stop memory read access sampling
20 res = samp_read_stop(&s);
21 if(res < 0) {
22   printf(``err:%s\n``, error_message(res));
23   return -1;
24 }
25
26 // Print memory read sampling results
27 printf(``Memory read sampling results\n``);
28 print_rd(stdout, &s);
```

Fig. 9. Basic usage of numap. User code must use the sequence init_samp_session, samp_read_start and samp_read_stop. Then user code exploits the resulting samples. In this example, they are only printed on standard output using the numap built-in print_rd function.

identifier is retrieved through the syscall(SYS_gettid) call. Then, sampling is started just before the code to be profiled with samp_read_start (line 11) and stopped with samp_read_stop when needed (line 20). At this point the samples are available for use. Figure 9 only prints them on standard output using the numap built-in print_rd function (line 28). A real life usage of numap will deeply analyze the samples either using cpy_samples, or directly accessing the s.metadatas list. This example also shows the usage of the error_message function (lines 5, 13 and 22).

B. Implementation

The implementation of numap relies on the classic perf_event_open usage such as described in Section IV. Indeed, one of the goal of numap is to provide portability for memory profiling and perf_event_open already supports many micro-architectures.

Regarding portability, numap implementation automatically selects the correct hardware event to be used for memory read or write sampling and for memory requests counting. The set of supported micro-architectures is coded as a list of structures tagged by processor and family identifiers. For each micro-architecture, the structure contains the identifier of the events to be used with perf_event_open. This identifiers are not directly the integer representation of the events to be passed to perf_event_open but a string representation. This string is feed to the pfm library [15] which purpose is only to convert a string event to a number usable by

Name	Read Sampling	Write Sampling
Nehalem - Lynfield decline	Yes	No
Westmere - EP decline	Yes	No
Sandy Bridge	Yes	Yes
Sandy Bridge - EP decline	Yes	Yes
Ivy Bridge	Yes	Yes
Ivy Bridge - E decline	Yes	Yes
Haswell - E decline	Yes	Yes
Haswell - DT decline	Yes	Yes

TABLE I
MICRO-ARCHITECTURES SUPPORTED BY NUMAP

`perf_event_open`. To find the string identifiers of the events to be used for each supported Intel micro-architecture, we used the software developer manual [4], chapters 18 and 19. Navigating through this document is really difficult for newcomers. It requires both having a clear overview of the history of Intel micro-architectures and a lot of time to find the particular event among all the events provided. The micro-architectures currently supported are shown in Table I. As shown in this table, write accesses sampling has been introduced by Intel with the Sandy Bridge family. All these micro-architectures support counting of memory requests.

The implementation of `numap` hides as much as possible the details described in Section IV. Calls to `perf_event_open`, `mmap`, `ioctl` and memory barriers are used internally by the implementation of the `numap` functions. `perf_event_open` calls are done by the `samp_xxx_start` and `count_xxx_start` (xxx is either read or write) functions. For sampling, the starting functions also perform the `mmap` call. The `ioctl` calls needed to start and stop profiling are also done internally by `numap`. The `cpy_samples` function manages the call to the memory barrier and the conversion from the raw results provided by the kernel to the list of `struct samp` elements.

VI. LIBRARY USAGE EXAMPLE

`numap` was initiated as a mean to profile dataflow applications executed on NUMA architectures [16]. It started as an ad-hoc tool to evolve to a clean API to be used outside its initial scope. This section shows how we use `numap` to build a memory profiler dedicated to dataflow applications.

A. Dataflow Programming

In the dataflow programming model, an application is described as a graph. Each node of the graph, called an actor, represents an independent computing unit operating on input data to produce output data. Edges between actors specify the data dependencies between them and are the only mean for actors to communicate. More precisely, these edges behave as First-In-First-Out (FIFO) channels. The independence between actors and the blocking-read semantics associated to channels allow the dataflow compiler and/or runtime to automatically distribute the computing work of actors among the available cores when targeting parallel hardware.

We use `numap` to profile dataflow applications written using the RVC-CAL language and compiled with the Open RVC-CAL Compiler (Orcc) [17]. This compiler is a source-to-source compiler producing C multi-threaded code to be compiled by a standard C compiler to produce the binary, e.g., GCC.

B. Profiling RVC-CAL Applications With `numap`

Understanding the memory performances of dataflow applications is crucial to map them on the hardware in an efficient way. `numap` allows to easily get memory samples for the dataflow application in a portable way.

Figure 10 shows the profiling toolchain we implemented inside Orcc. As presented in this figure, the Orcc compiler has been modified to include memory sampling mechanisms within the generated C code. Mainly, calls to `numap` start and stop sampling functions are inserted each time a thread is created. The generated C source code is then linked against `numap`. In this instrumented C source code, when a thread is stopped, the samples are recorded to a file for offline analysis.

This offline analysis uses the information reported in each sample provided by `numap` in order to associate samples to channels and actors. This allows, typically, to determine the cost (in terms of latency) of an actor producing data tokens to a particular FIFO channel. This association is required to provide hints to the application programmer and the designer of the dataflow compiler. They can then modify and hopefully optimize the application and/or the compiler accordingly.

Using `numap`, the instrumentation of the Orcc compiler to perform low-level memory profiling was straightforward. Only few calls to the library are required without the need to understand at all how the Linux `perf_event_open` system call behaves and how the PMU gets the information. The `numap` user code (generated by the Orcc compiler in this case) just tells `numap` to perform memory read and/or memory write sampling for its own threads. Having a library such as `numap` also allows to perform very fine grain sampling, by allowing to start and stop profiling at any time.

As a result of our profiler, we get a memory cost information for all the dataflow objects involved in the execution of the program. Table II shows the profiling results of a video decoder executed on a NUMA machine made of two Intel Westmere processors. In this case, the most expensive objects regarding memory are scheduling objects and FIFO channels with identifiers 18 and 2. The scheduling objects are not visible to the programmer but generated by the compiler and are used to ensure dataflow dependencies are satisfied before executing actors. From this information, we can take better decisions regarding the mapping of the FIFO channels 18 and 2, the actors writing and reading to and from these channels and the scheduling objects.

VII. CONCLUSION AND FUTURE WORKS

This work proposes the `numap` library to ease the usage of memory features proposed by the PMU of many modern processors. `numap` is both portable and easy to use. Portability

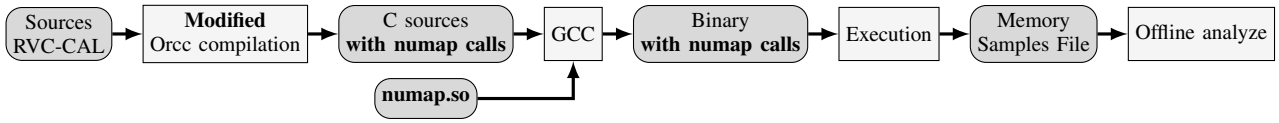

Fig. 10. RVC-CAL memory profiler based on numap. The Orcc compiler is modified to generate code including `numap` function calls. During the execution of the application, memory samples provided by `numap` are recorded for offline analysis.

Source	Cost(%)
schedWait	7.88
sched	3.92
FIFO-18	3.86
FIFO-2	3.58

TABLE II
DATAFLOW OBJECTS WITH THE HIGHEST TOTAL MEMORY LATENCY

is ensured by hiding micro-architectural concerns to user-code, while usability is provided by a clean interface. We are successfully using `numap` as a basis of a memory profiler for dataflow applications. We believe that such libraries are required to help the software community build portable profiling tools and runtime mechanisms in an efficient way.

Nevertheless, to increase the adoption of `numap` we are working on several directions. First, we are currently porting `numap` on AMD processors. This requires to understand the details of IBS, the PMU memory sampling capabilities for AMD. This task will be nevertheless straightforward because `perf_event_open` already supports IBS[18], and because we gained deep experience of this system call while developing `numap` for Intel platforms. Independently of the complexity of porting `numap` on AMD micro-architectures, it is worth mentioning that our API will stay the same, because the IBS mechanism provides all the sampling attributes of the `numap samp` structure and because it also requires a frequency input parameter.

Second, we are planning to add better support for the management of long lived applications. Indeed, `numap` currently only allows to save a maximum number of events specified at initialization time. To support applications willing to process samples while running, we will extend the API with functions to indicate to the library the part of the samples already read that can be erased. This information will then be propagated to the Linux kernel to let him recycle corresponding memory pages. To support applications only requiring offline analysis of the samples, the library will provide a mechanism to automatically dump the samples into a file. This mechanism will be toggled with a simple additional parameter to the initialization function of `numap`.

Last, we think that the features provided by `numap` can and should be integrated into other profiling tools. Among those, a preferred target would be the widely used PAPI library. In a near future, the best of both `numap` and the memory sampling abstraction API proposed recently for PAPI should be integrated into the main version of PAPI.

REFERENCES

[1] D. Molka, D. Hackenberg, R. Schone, and W. Nagel, "Cache coherence protocol and memory performance of the intel haswell-ep architecture," in *Parallel Processing (ICPP), 2015 44th International Conference on*, Sept 2015, pp. 739–748.

[2] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for numa-aware contention management on multicore systems," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 1–1.

[3] B. Lepers, V. Quema, and A. Fedorova, "Thread and memory placement on numa systems: Asymmetry matters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 277–289.

[4] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, January 2015.

[5] P. J. Drongowski. (2007) Instruction-based sampling: A new performance analysis technique for amd family 10h processors.

[6] https://perf.wiki.kernel.org/index.php/, (Retrieved 2016-02-19).

[7] https://software.intel.com/en-us/articles/intel-performance-counter-monitor, (Retrieved 2016-02-19).

[8] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra, "Using papi for hardware performance monitoring on linux systems," in *In Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute*, 2001.

[9] I. Lopez, S. Moore, and V. Weaver, "A prototype sampling interface for papi," in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, ser. XSEDE '15. New York, NY, USA: ACM, 2015, pp. 27:1–27:4.

[10] C. McCurdy and J. Vetter, "Memphis: Finding and fixing numa-related performance problems on multi-core platforms," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, March 2010, pp. 87–96.

[11] R. Lachaize, B. Lepers, and V. Quéma, "Memprof: A memory profiler for numa multicore systems," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 5–5.

[12] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placement on numa systems," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 381–394.

[13] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on numa architectures," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 259–272.

[14] X. Liu and B. Wu, "Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 47:1–47:12.

[15] http://perfmon2.sourceforge.net, (Retrieved 2016-02-19).

[16] M. Selva, "Performance Monitoring of Throughput Constrained Dataflow Programs Executed On Shared-Memory Multi-core Architectures," Theses, Institut National des Sciences Appliquées de Lyon, Jul. 2015.

[17] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: Multimedia development made easy," in *Proceedings of the 21st ACM International Conference on Multimedia*, ser. MM '13. ACM, 2013, pp. 863–866.

[18] https://lwn.net/Articles/490418, (Retrieved 2016-02-19).