M.S. THESIS

SnuMAP: A Modular Trace Profiler Tool for Manycore Systems

SnuMAP: 매니코어 시스템을 위한 어플리케이션 추적 프로파일러

AUGUST 2018

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Camilo Andres Celis Guzman

M.S. THESIS

SnuMAP: A Modular Trace Profiler Tool for Manycore Systems

SnuMAP: 매니코어 시스템을 위한 어플리케이션 추적 프로파일러

AUGUST 2018

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Camilo Andres Celis Guzman

SnuMAP: A Modular Trace Profiler Tool for Manycore Systems

SnuMAP: 매니코어 시스템을 위한 어플리케이션 추적 프로파일러

지도교수 Bernhard Egger

이 논문을 공학석사 학위논문으로 제출함

2018 년 06 월

서울대학교 대학원

컴퓨터 공학부

카밀로 셀리스

카밀로 셀리스의 공학석사 학위논문을 인준함

2018 년 07 월

위 원 장	김 선	(인)
부위원장	Bernhard Egger	(인)
위 원	Srinivasa Rao Satti	(인)

Abstract

In this thesis, we propose SnuMAP, an open-source modular trace profiler for many-core systems. SnuMAP provides per-application and whole-system views of multiple data points of interest: core allocation, power, CPU and memory utilization. Additionally, SnuMAP is light-weight, requires no source-code instrumentation and does not degrade the performance of the target parallel application. It alternatively gathers valuable information and insights for application developers and many-core resource managers. This type of tools continues to gain importance as today's many-core systems co-schedule multiple parallel workloads to increase system utilization. We have put SnuMAP to use in numerous research projects and present in this paper a snapshot of essential findings enabled by the visualization and data SnuMAP can provide. This project started originally as a simple open-source profiler, then it grew and evolved into the complex analysis tool it is today. More information is available at `http://csap.snu.ac.kr/software/snumap`.

Keywords: profiler, many-core systems, NUMA, power, Intel RAPL

Student Number: 2016-22100

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

Motivated by the endless need for additional performance, in recent years there has been a clear trend transforming computer systems from simple single-core architectures into multi-socket many-core systems. Moving from raising the CPU frequency to uniting more and more cores in different arrangements on single/multiple sockets, it is now common for modern data centers and high-performance computing (HPC) centers to operate distributed many-core computer nodes in single or multi-socket arrangements. Not only high-performance systems take advantage of a large number of cores, but also desktop-level users have been exposed to this trend. Flagship smartphones and IoT devices now boast having two, four, and even eight cores in their system. Intel recently released a modified server-level Xeon-processor architecture for amateur consumers as their Intel X-series processors family [40] with up to 18 cores and 36 threads.

This architectural shift has forced software developers to consider parallel systems even more. Not long ago, parallel workloads and application were con-

sidered to be a craft for graphics related programs or large-scale systems. Nowadays, only efficient parallel programs can take full advantage of all resources of a system as more and more devices have moved to this many-core technology. However, modifying a program to be scalable and parallel is not simple and in some cases impossible. Writing efficient parallel software requires a precise understanding of the underlying architecture and the hardware resources, as these can profoundly affect the performance of a parallel application.

This added complexity motivated the creation of novel tools that help developers understand exactly how a program executes on a target architecture and aid the development process by pointing out performance degradation and bottlenecks in the system [76, 33]. Trace visualization is one of the most natural and most powerful ways to interpret the behavior of (parallel) programs. It allows programmers to have a transparent and comprehensive picture of the execution of an application, and the interaction with different resources in a system. A variety of trace visualization and profiler tools have been introduced and are widely used by parallel application developers especially in HPC and performance-critical environments [23, 1, 69, 82].

There are three principal themes of modern large computer systems that are important to account for, as those form the primary motivation for SnuMAP and the performance metrics it profiles: co-located execution, power consumption, and non-uniform memory sub-systems (NUMA). Recently, parallel programs are rarely executed in isolation on modern multi/many-core platforms. For example, Mesos [36], a data-center OS, executes several big-data application frameworks across distributed nodes for manageable execution with varying resource requirements. Besides, general-purpose servers and HPC centers often execute several parallel workloads [13] simultaneously to enhance system utilization and energy efficiency. In such environments, understanding how par-

allel applications are executed on a given many-core platform is gaining on importance for both software developers and data center managers. Nevertheless, prevalent trace profilers [58, 32, 15, 76, 61] are not adequate to provide the before-mentioned information because most of them consider standalone execution and do not collect compelling scheduling information from system schedulers.

Furthermore, power consumption has rapidly become a significant concern. It is well known that with current performance requirements the amount of power required to power the next super-computers will exceed that of standardized limits [14]. Smartphone users and, more importantly, IoT devices require the system to be tailored to low-power consumption. These are examples of scenarios where power consumption can be the driving force, even more than performance. Energy, temperature, and overall power consumption are then essential data points to be studied during the development and execution of parallel applications. SnuMAP, consequently, realizes this and profiles different energy metrics from different system-level resources which allow a correlation and understanding of bottlenecks or optimizations for both better performance and lower power consumption.

Lastly, due to the large number of cores that are being placed on a single socket, chip manufactures apprehended that the performance gap between the CPU and memory was exponentially increasing [63, 24, 64]. Adding more and more cores cannot solve the performance needs as the memory subsystem cannot keep up with these advances. Contention caused by the many memory operations started to become even more important for software developers. As a means to solve this problem, new architectures like NUMA were introduced that divide the memory system into several nodes, and the cores into multiple sockets. This creates a hierarchical structure of computing and memory

nodes which adds complexity as memory requests can now be issued to a local or a remote memory node. Memory accesses might not have been of interest when tailoring performance for software developers, but it now is. Performance can drastically increase if the placement of data considers NUMA characteristics [12, 52, 81, 51].

In this paper, we introduce the SNU Many-core Applications Profiler (SnuMAP), a modular open-source trace profiler framework for multi/many-core systems. SnuMAP is light-weight, requires no source-code instrumentation and does not incur any performance degradation of the target (parallel) application. SnuMAP profiles three types of logically different data points: core allocation, resource utilization, and energy.

First: core allocation. The execution trace of all threads in a (multi-threaded) parallel applications is collected inside the Linux kernel by tracking every context switch event. Since the Linux scheduler collects the scheduling information, SnuMAP can capture performance interference and scheduling conflicts between different (parallel) workloads and provide insights for application developers and platform resource managers. This feature is gaining relevance as today's multi-core systems co-schedule multiple parallel program instances to increase system utilization.

Second: resource utilization. SnuMAP implements a small architecture-optimized performance monitoring unit (PMU). This unit is capable of monitoring CPU utilization and performance plus memory access patterns and the behavior of the requests, allowing an understanding of NUMA-related performance information, performance bottlenecks, and various types of interference.

Third: energy. SnuMAP acts as an interface for Intel's power running average power limit (RAPL) [19] in order to measure energy and power consumption of CPU, memory, and graphical devices on the system and correlates this infor-

mation with resource utilization and core allocations, providing a system-level or application-level view of the effects on energy and power consumption.

The proposed tool is not limited to the programming language. SnuMAP captures the creation and termination of threads during an application's lifetime by intercepting calls to the standard operating system library. If a specific program uses other means of execution, most frameworks provide a means to trace such calls. In this thesis, we limit ourselves to C/C++, Java, and Python.

This thesis is structured as follows. Chapter 2 prepares the reader on different concepts and ideas needed to understand the remaining chapters. Chapter 3 compares and introduces similar approaches used for profiling and related tools. Chapters 4 and 5 go explain in detail SnuMAP's design choices and implementation. We discuss various real-world use-cases on which SnuMAP excelled on chapter 6, and conclude this thesis and present our future work on chapter 7.

Chapter 2

Background

In this chapter, we overview various concepts and ideas necessary for a better understanding of the motivation behind SnuMAP, its implementation, and the later presented use cases.

2.1 Timing Mechanisms

In this section, we briefly explain software-based mechanisms to keep track of time (clocks). We focused on the Linux kernel and x86 architectures. Discussing how time is defined and kept in a system is fundamental as it forms the basis for any profiling tool like SnuMAP.

Many of the kernel most critical parts rely on some timekeeping mechanism. Initially, the default software-based timing system in the kernel was called *timer wheel* [30]. This system is based on a global kernel value (jiffies) which is incremented once on every timer-interrupt. All other timing metrics fall-back to this value. HZ is the compile-time constant that defines the rate at which the

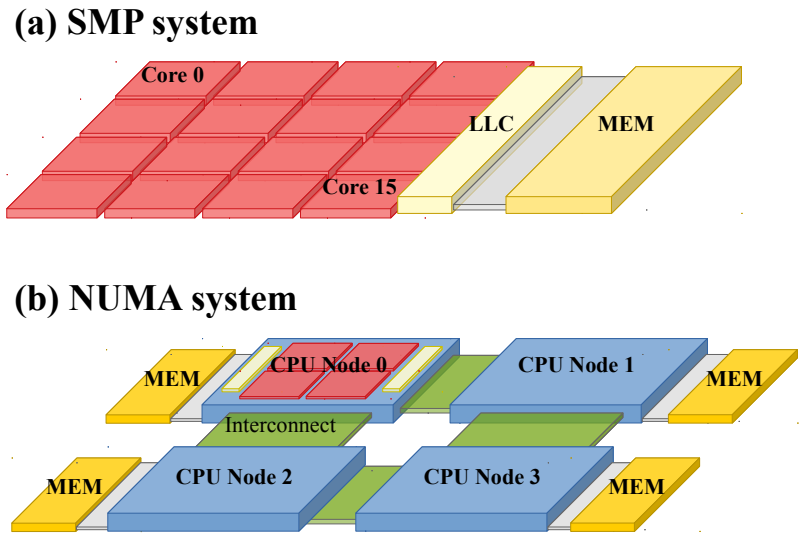


Figure 2.1: SMP and NUMA system architecture.

system performs time-interrupts. Therefore, the values of jiffies and HZ can be used for time conversion[1]:

```
jiffies = no.ticks
HZ = jiffies / second

seconds_to_jiffies(seconds) = seconds * HZ
jiffies_to_seconds(jiffies) = jiffies / HZ
```

2.2 Manycore Processors Characteristics

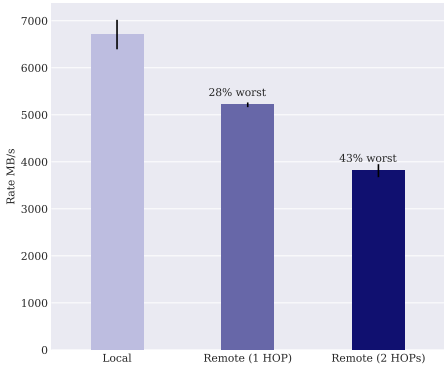
Many/multi-core systems have various definitions in the literature. For this thesis we define a *multiprocessor* or *manycore processor* as a computer system that is confirmed by two or more homogeneous Central Processing Units (CPUs) or cores running independently of each other. Many-core systems can

[1]For example most x86 system kernels default to HZ of a 100 yielding a jiffy interval of 10ms.

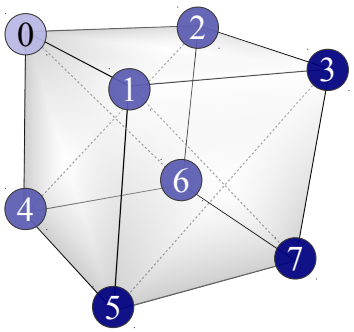
be categorized into two main groups regarding their architectures Symmetric Multiprocessing (SMP), and Non-Uniform Memory Access (NUMA) systems.

Figure 2.1 shows two 16-cores systems for both architectures. The main difference between both architectures is the way cores are connected to each other and to main memory. On SMP systems all cores share main memory via a single bus. On the other hand, on NUMA systems cores are grouped on nodes, and each node is directly connected to its local main memory via local buses. There also exists an interconnection fabric between different nodes, allowing all nodes to communicate with each other. Note that multiple types of interconnection exist, for example in Figure 2.1 node zero has no direct way to communicate to node three without communicating first with node one or two. For this NUMA architecture to allow all its nodes to directly communicate with all other nodes a cross-bar interconnection needs to be added between nodes zero and three, and two and one. The memory request latency will vary depending on if requests go to local memory or not. To showcase the effects this NUMA characteristics have in performance we executed a simple benchmark and report the results on Figure 2.2a for a 8-node 4-way AMD NUMA platform (see Figure 2.2b). This simple benchmark determines the best memory rate in megabytes (MB) per second on three different scenarios: (a) requests to local memory, (b) requests to remote memory one hop[2] away, and (c) requests to remote memory two hops away. It is surprising that there is a reduction in performance at must 43% only by deciding to which memory node the data used by the benchmark is bound to.

[2] *hop* is the relative distance between two NUMA nodes. It takes an integer value in the range $[1, \infty)$ where 1 is the shortest possible distance. For completeness, the hop distance between a node and itself is 0.



(a) Request rate vs. memory distance.



(b) 8-node 4-way NUMA system.

Figure 2.2: Effects of local and remote memory requests on a 8-node AMD Opteron NUMA platform with 64 cores connected with 2 hops or less.

2.3 Intel RAPL

Released by Intel in 2010 as part of their open source technology initiative [42] Intel Running Average Power Limit (RAPL) is a set of utilities and drivers to provide both power measurements and capping power on modern systems that use Intel chips. Power measurement using analytical models is not new; yet, it has been possible for Intel to implement hardware power measurement functions in their new processor architectures starting with SandyBridge. Multiple approaches have been implemented and evaluated in the past [45, 8, 11, 57, 44]. However, RAPL is one of the first approaches to consider not only CPU power modeling but also memory as an additional resource of which power can be measured and capped [19]. In its core, RAPL is an interface for limiting both CPU and memory power. Instead of maintaining instantaneous power limits on both resources, RAPL tries to keep an *average power limit* over a sliding time window. This single feature mitigates performance degradation present on most power capping/budgeting frameworks that consider highly dynamic and tran-

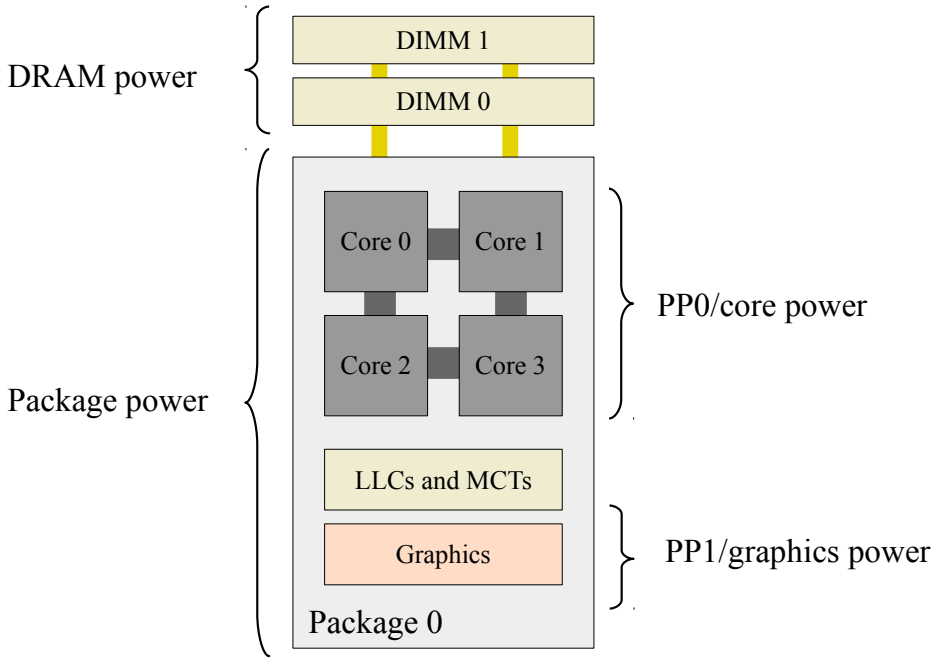


Figure 2.3: Intel RAPL power zones representation on a single socket.

sient workloads. Details regarding this interface are described in Intel Software Developer's Manual [43].

2.3.1 Power Measurement

RAPL allows power and energy measurement via on an online power-estimation model based on hardware performance counters, leakage models, and temperature. The system is logically divided into multiple power zones each of which report independent energy values (see Figure 2.3): package, `PP0`/cores, `PP1`/graphics, and DRAM power. Note that not all Intel processors variations support all zones. This estimation model has been introduced in detail and thoroughly validated by Intel [65].

2.3.2 Power Capping

RAPL's power capping algorithm determines the power budget that it needs to apply independently for CPU and memory resources. For the memory subsystem, power is monitored and controlled by mapping a power budget to different low-power memory states. This, in turn, translates into different reduction ratios of memory bandwidth. For example, as seen in Table 2.1 from the initial publication of RAPL by Howard David *et.al*, the memory subsystem and platform used was composed of 16 different memory power limiting (MPL) states with direct mappings into bandwidth and average power consumption values. The methods used to achieve this low-power states where mainly two: modification of the Column Access Strobe (CAS) timing [3] and idle injection to allow for delays when issuing commands by the memory controller.

Similarly; for CPU resources RAPL maps different power budgets to the different mechanisms to reduce power. However, CPU power management, reduction, and capping have been studied and implemented more broadly than memory. This is why the mechanisms used to reduce power varies much more than in the case of memory. The most commonly used mechanisms by RAPL are:

- **P-states** or performance steps. These are CPU energy steps represented as a voltage and frequency pair. On Intel systems this does not quite compare to dynamic voltage and frequency scaling (DVFS) as the OS can control only part of these levels directly; the hardware (HW) is in charge to go beyond DVFS levels and increase performance even further via turbo states (via Intel Turbo Boost [41]).

[3] Control signal used in asynchronous DRAM modules. DRAM stores data in cells indexed by a column and row identifiers, CAS is the strobe sent by the processor to activate a column address.

Table 2.1: Example of MPL states for Intel RAPL's memory capping algorithm.

MPL State	Best Bandwidth (%)	Worst Power (W)	Method
MPL0	100	11.9	Adjust CAS timing
MPL1	80	10.5	
MPL2	66.67	9.46	
MPL3	57.14	8.70	
MPL4	50.0	8.12	
MPL5	44.44	7.67	
MPL6	40.0	7.30	
MPL7	36.16	6.31	Idle injection
MPL8	33.33	6.01	
MPL9	30.25	5.69	
MPL10	27.07	5.35	
MPL11	23.58	4.99	
MPL12	20.0	4.61	
MPL13	16.34	4.23	
MPL14	12.33	3.80	
MPL15	8.24	3.37	

- **T-states** or throttling states. Represented as software-controlled CPU-clock modulation levels. These are commonly controlled via thermal cooling devices to avoid over-heating and damaging the processors.
- **C-states** or idle injection. This method enforces an idle time for each online CPU using ratio selectable by software.

2.3.3 User Interfaces

RAPL provides different interfaces to allow users to interact with its power capping and energy measuring capabilities. Not all interfaces provide the same level of freedom and flexibility. The two most common interfaces are: (1) a directory structure under `sysfs/powercap/intel-rapl` and (2) via Model Specific Registers (MSR). The latter approach requires the detailed specification of the MSR address and layout, therefore, can be more cumbersome to interact with.

Moreover, it requires supper-user access permissions. On the other hand, `sysfs` files can be read easily by any user in the system. The basic directory structure for RAPL under `sysfs` is as follows:

- `/sys/class/powercap`
  - `intel-rapl:X`
    - `constraint_Z_max_power_uw`
    - `constraint_Z_name`
    - `constraint_Z_power_limit_uw`
    - `constraint_Z_time_window_us`
    - `enabled`
    - `energy_uj`
    - `max_energy_range_uj`
    - `name`
  - `intel-rapl:X:Y`

Where `X` represents the node id, `Y` the memory controller (MCT) id within each node id (so this is usually `0`), and finally `Z` represents the constraint id; some systems for example support two different constraints on the package zone but only one constraint on the DRAM zone. As can also be seen from the `sysfs` directory structure there is no way to measure PP1 or PP0 zones independently. For this, access to the MSRs is required.

2.3.4 Power Measurements in Other Architectures

Intel RAPL is by far the most advanced and readily accessible power meter framework that does not require hardware instrumentation. Nevertheless, other hardware manufacturers also allow energy measurements using different methodologies, accuracy, and sampling rates; we provide a list below:

- Fairly recently AMD 15h processors family can report power via the *Processor Power in TDP* and *Average CPU Power* MSRs [2].

- NVIDIA GPUs can report power usage with an error of approximately 5 Watts at a 60Hz sampling rate via the NVIDIA Management Library (NVML) [62].

- Manufactures like ARM and IBM support the use of co-processors and external boards to allow the collection of power information. In the case of ARM, various cycle-accurate simulator usually also report power data based on architecture specifications.

- Many studies have focused on power estimation using various approaches from machine learning to online power models [9, 72, 59, 60].

Chapter 3

Related Work

Profiling and trace visualization are standard tools for performance analysis, debugging of large-scale systems, and modern parallel software. In this chapter, we explore some of these tools, observe what their benefits are, and compare them to our proposed approach: SnuMAP.

Since the inception of state-of-the-art operating systems profiling tools and performance analyzing tools have been are the core of design and implementation decisions. Particularly during the last few years, many tracing tools have been proposed to facilitate the detection of performance problems and bottlenecks on large power-hungry many-core systems. Two of the most well-known tools are Dtrace [25] and KProbes [49]. The Linux kernel itself comes with a useful but limited set of tracers such as SystemTap [70], Ftrace [28], and Ptrace [34]; these are usually used for debugging and analysis via script-like queries to define specific event(s) tracing. They are in most cases sufficient, and many other visualization tools are built on top of them [48, 73, 83]. Four of the most recent profilers for Linux focus on NUMA architectures, performance and

power analysis, and visualization of system invariants.

Memprof [50] is a profiler that targets NUMA architectures that analyzes the interaction between threads and objects in an executing program. Memprof achieves this by collecting thread events and object flows. NUMA characteristics are understood by collecting memory objects using a similar approach as SnuMAP via library interpositioning to intercept calls to the dynamic memory allocator. Furthermore, Memprof also implements a kernel module (as SnuMAP and other approaches do [17]) to communicate with the Linux kernel for tracking the life cycle of code sections and global static variables by overloading `perf_event_mmap`. However, Memprof differs from SnuMAP in the means to collect profile data. Instead of relying on performance counters to measure memory accesses, Memprof uses Instruction Based Sampling (IBS) [46] to periodically select a random instruction and record its performance information.

Another powerful profiler and graphical tool is Aftermath [23], an open-source graphical tool for performance analysis and debugging. Aftermath allows the joint visualization of task durations, hardware counters and data exchanges in an interactive user interface. As opposed to Memprof, and using a similar approach to SnuMAP [1], Aftermath reads performance counters directly from the target architecture via the PAPI software interface [71]. Additionally, Aftermath provides a powerful user-graphical interface for visualizing the consolidated data and makes it easy to apply filters to enhance essential sections.

Lastly, HPCToolKit [1]. This is a framework for analyzing the performance and scalability bottlenecks of parallel programs. It enables powerful profiling without code instrumentation similar to SnuMAP. HPCToolKit allows the discovery and quantification of performance bottlenecks via binary analysis, top-

[1]It is crucial to note that SnuMAP uses its own custom interface which consists of Linux `perf`, Intel PCM [39], and direct I/O of MSR devices.

down performance analysis and scalability of measurements. Their approach for data collection is fundamentally different than the one from SnuMAP, using statistical sampling, event triggers, and stack unwinding.

Most of the surveyed tools focus on performance analysis of single isolated programs and disregard an integration between performance and energy behaviors as a single view of a parallel application execution. Multiple co-located workloads are no considered, and the interaction between the target application, the parallel runtimes, and the operating system is ignored. It is often the case that performance bottlenecks raise due to complex and intertwined interactions between resource managers and running applications in sophisticated architectures. That is why one of SnuMAP's key features enables the visualization of executing co-located programs interacting with the underlying resource managers and hardware.

A similar approach was recently taken by Lozi *et. al* [55] to explore different types of performance bugs in the Linux scheduler. The authors realized that as hardware architectures become more complex, so does the resource manager of the Linux kernel. However, new features built on top of old assumptions have let the Linux scheduler evolve into a complex beast. The authors used generalized data collection techniques to gather information from the kernel without targeting a specific application within the Linux kernel. This approach allowed them to observe the effect the Linux scheduler has on different types of scenarios involving parallel applications.

Another compelling data point for modern profilers is power. Power consumption is the driving force in many fields in computer systems and will continue to do so. The most representative profilers in this field are PowetTOP [75], a Linux tool used to diagnose and correct power related issues. PowerTOP di-

rectly communicates with the ACPI[2] subsystem and provides specialized support for Intel drivers like `intel_pstate` for P-states control and report, and `intel_idle` for C-states. Most of power and energy statistics are presented to the user on a text-based graphical user interface (GUI) which can be somewhat difficult to understand. As discussed on chapter 5 SnuMAP uses Intel PCM to queries advanced Intel-specific hardware counters including power and energy information. All these counters can be easily exported into `ksysguard`-suitable log files which can be consumed by a `ksysguard` instance and displayed as time-series plots.

Besides these recent contributions, through the years multiple tools have been proposed using different techniques for data collection and analysis. In contrast, SnuMAP tries to be a general solution that uses data available regardless of the programming model to create insightful visualizations of the system and each program's execution. Important proposed tracing tools [15, 26, 61, 78, 80] enabled the collection of information; however, the analysis of traces is a difficult task, and most of the time traces alone prohibit the discovery of performance problems. A more general and powerful approach are profilers [5, 18, 61, 54, 32]. Despite offering more than tracing information, most of them focus on general and old architectures, overseeing the complexity and specific characteristics of modern architectures and the interaction with the target application. Lastly, some tools [5, 58, 15, 29, 67, 76, 79] focus on specific programming models which allows them to make strong assumptions allowing for a better understanding of the performance of the application. However, this approach focuses on the application as an isolated entity and does not observe the interaction with the underlying software and hardware.

[2]Advanced Configuration and Power Interface (ACPI): open industry specification for power management from the operating system.

Chapter 4

Overview and Design

In this chapter, we present a brief overview of SnuMAP's components and design choices. As Figure 4.1 denotes, SnuMAP is comprised of five de-coupled and modular components that enable profiling on both single-threaded and co-located parallel workloads:

- `SnuMAP-main`: main entry point / `SnuMAP-shared` library.
- SnuMAP-enabled Linux kernel.
- Custom kernel module and context-switch tracker.
- Performance and power monitoring unit.
- Visualizers.

We decided to free the software developer from code instrumentation while using SnuMAP as much as possible by intercepting calls to the standard `libc` library's `main` and `exit` routines via library interpositioning in order to signal the start and end of profiling. We opted to interface via `libc` as it is a universal and low-level system programming library; consequently, most programming

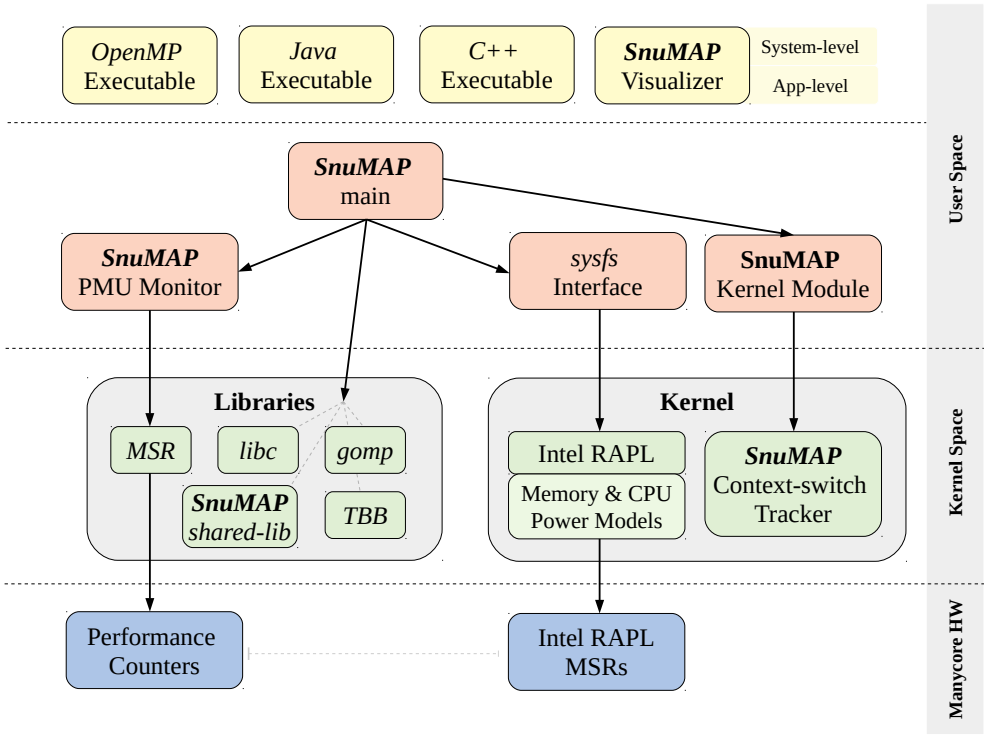


Figure 4.1: SnuMAP software stack.

languages and frameworks provide bindings or foreign function interfaces (FFI) for it. Similarly, we used library interpositioning for applications that also use several parallel run-time frameworks such as OpenMP [31], MPI [27], and Intel TBB [64] in order to collect framework-dependent information. For example, currently, SnuMAP is able to clearly distinguish between all parallel and sequential sections within an OpenMP application that is being profiled.

SnuMAP also includes a kernel module and a performance and power monitoring unit in order to collect the required profile information. The custom kernel module acts as a vehicle to communicate when to start, stop, or dump event logs from SnuMAP's context-switch tracker. Core-allocation information is populated via a custom context-switch tracker. As Figure 5.1 shows, the con-

text switch logic remains untouched and the only extension happens between context-switches of two tasks; the suspension and resume timestamp of both tasks is recorded by SnuMAP. Note that this information is core-dependent and so we need to differentiate the three data points core id, task id, and time stamp. This tracker resides deeply in the core of the Linux kernel and uses the previously described kernel module to activate itself; therefore, having no effect on tasks not being profiled by SnuMAP. We decided to allow kernel source modification as this allows very precise timing of context-switch events for every task on all the cores of a system with very low overhead.

The performance and power monitors constantly obtain important performance counter and energy information from the system and correlates this information to the parallel applications of interest using the core-allocation data. These two monitoring binaries are triggered when profiling is started and write event logs to a trace file during their execution; providing an interface to hardware performance counters and system level information that can be correlated with the application(s) being profiled under SnuMAP.

Due to the fact that not all programming languages nor runtime frameworks can be supported, SnuMAP includes a shared library to enable other programs to call a single and simple API to signal the start and termination of profiling. This method is used for example to profile Java programs (see section 5.3). With this shared library and library interpositioning virtually any binary can be profiled by SnuMAP. As soon as the target program starts, SnuMAP begins collecting data for any new thread the parallel program creates, system level utilization and power data. All data collected by SnuMAP can be selected before starting execution; therefore, only the SnuMAP modules required to collect the specified data will be enabled at runtime. After the parallel application finishes, various log/trace files are created.

Chapter 5

Implementation

In this section, we describe the different components that construct SnuMAP at some level of detail. As discussed in chapter 4; SnuMAP can profile core allocation, performance or resource utilization, and power.

We have divided this chapter into four relevant sections. The first two sections explore details regarding the components needed to profile core allocation information, performance/resource utilization, and power data. The third section discusses the various user-level interfaces SnuMAP provides to communicate with the target applications to allow profiling. The last section of this chapter discusses all the tools used to display and analyze the profiled data via the powerful visualization tools.

5.1 Core Allocation

This sections lists and explains all modules necessary to collect and construct core-allocation profile data. Starting with a custom Linux kernel patch to a kernel module for communication.

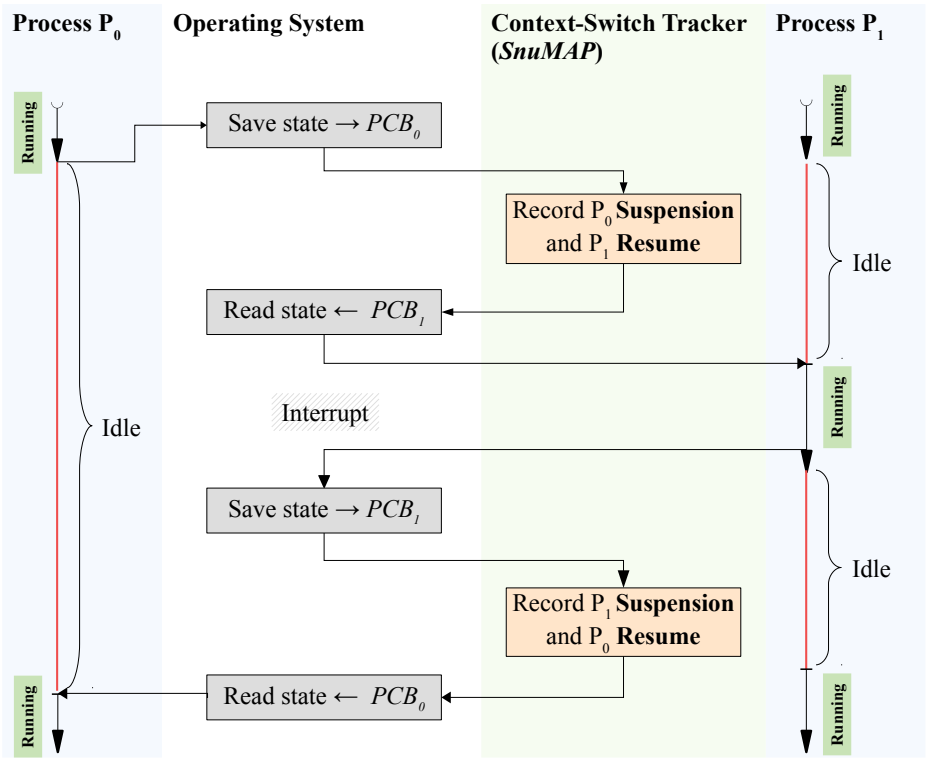


Figure 5.1: Regular context-swich vs. SnuMAP's context-swtich tracker.

5.1.1 Kernel Patch: context-switch Tracker

In order to create high-definition plots, we opted to allow small changes to the Linux kernel to support the collection of context-switch data. The principal source of information for SnuMAP cor eallocation relies on the elapsed time at each context-switch. In the Linux kernel, each core has a list of runnable threads [1]. When a thread performs a context-switch, SnuMAP stores the time when the thread is suspended and when it is resumed.

Since our main focus is many-core architectures, SnuMAP uses `jiffies`, as

[1] In the Linux operating system the term *thread* and *task* are used interchangeably

timing mechanism, instead of each core's internal clock as this allows SnuMAP to have a global synchronized view of time with respect to all other cores in the system. SnuMAP's context-switch tracker patch defines two additional states for a thread in the system: (a) paused, or (b) running. A *paused* thread, is the one that has been context-switched out by another thread in the same core. In contrast, a *running* thread relates to a thread that is currently running on the system on an identifiable core. The timing information produced by tracking changes to a thread state is stored for every core using a dynamic data structure. This data structure starts its life with a fix-size to avoid overflowing the kernel's memory. A list of empty timestamps buckets is generated statically and populated at runtime. If all the buckets are utilized at runtime then a new list is created and appended to the previous list.

All these modifications are rather minor and mainly involve the scheduler methods that handle the creation and destruction of a thread. During the design of SnuMAP, we were well aware that trace profile tools like ptrace already provide means to collect context-switch events information for a given application. However, this information is not complete as it only shows the number of context-switches that happen for a thread within the kernel but does not correlate this information in a time series in relation with the core that triggered the event.

5.1.2 Kernel Module

A kernel module is a piece of code which can be loaded/unloaded dynamically into the Linux kernel. This is usually the way developers use to extend some functionality without the need to re-start the system. We opted to include a kernel module to enable/disable profiling, specially the communication and activation of the context-swtich tracker as it completely eliminates the overhead

of our kernel patch on threads which are not being profiled. SnuMAP's kernel module can be seen as the glue that enables the communication between the Linux kernel context-switch tracker and either the dynamic (library interpositioning) or static SnuMAP interface.

Custom `IOCTL` calls are used to directly interact with each thread data structures within the kernel (`task_struct`) and signal the start and stop of context-switch tracking. A simple flag in each `task_struct` is used to determine if this thread should be profiled or not. Thus, all the initialization of data structures needed for context-switch tracking and communication mailboxes is done in this kernel module. Once profiling has been completed the kernel module is notified and it start moving data from kernel space to the user space in the form of a trace file. Reducing SnuMAP overhead even further by doing all file IO after the execution of the profiled application. Note that the application will seem as not being completed yet, as SnuMAP will take control and dump the context-switch profile data, so timing information can be left to SnuMAP's trace files.

5.2 Performance and Energy Monitoring Unit

In this section, the two intertwined monitoring units for performance and energy/power are discussed. Logically they are based off the same design pattern. They sample performance counters and at-run-time information from hardware vendors and log this information to trace files. However; the two use different approaches, especially as energy measurements are not readily available on all hardware. Both the CPU and memory performance monitors use a mixture of various libraries and tools to gather information (see Figure 5.2). Linux *perf* tool is used to abstract access to common and useful hardware counters via the `perf_open_event` system call. Intel PCM [39] is included within SnuMAP

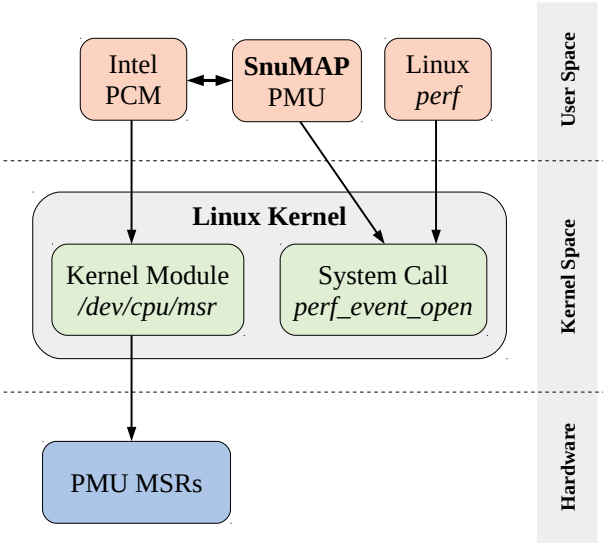


Figure 5.2: SnuMAP's performance and energy monitoring unit overview.

to allow querying more complex events from Intel processors. This open-source library abstract the direct access of MSRs exposed at the `/dev/cpu/msr` path. Similarly, for non-Intel architectures; MSRs are read directly based on each hardware provider's developer manuals. So far, SnuMAP supports the following architectures: AMD, Intel, and Tilera.

5.2.1 CPU Performance Monitoring

CPU performance has been studied in much detail. On most architectures different hardware performance counters are well supported, allowing for a better understanding on how a workload behave on the CPU. SnuMAP queries the following CPU related performance counters via its PMU:

- Work cycles.
- Stall cycles.
- IPS.
- Interconnect utilization.

- Frequency, C and P-states residency.
- Branch information.

5.2.2 Memory Performance Monitoring

Memory plays a critical role in today's complex architectures. It accounts for approximately 25% of the total power consumption of a system [21] and, if used incorrectly, can affect performance severely. As the number of cores increases in multi-core platforms, insufficient memory bandwidth has become an increasingly important issue. In NUMA architectures, it is often the case that if an application incurs multiple remote accesses to non-local memory this will highly affect performance. In other words, if an application incurs multiple remote accesses, the application should be restructured to use (neighboring) local memory(ies). For these reasons and the importance of understanding the effects of an application on the memory hierarchy SnuMAP collects a large range of performance counters regarding memory and cache information. Among the most important ones we have:

- Inter-node requests.
- Caches hits, and misses.
- Caches and TLBs' loads, stores and prefetches.
- Caches residency.
- DRAM clocks.
- Bytes written/read to/from each memory controller.

The collected information is time stamped using Linux `jiffies` which allows for it to be synchronized with other collected performance data.

5.2.3 Energy/Power Monitoring

Energy and power are difficult metrics to collect from current systems. There are no actual performance counters that expose this information. Instead, external power meters are used. Yet, the sampling rate of these approaches are low and the process error-prone. Additionally, it is difficult to synchronize between the execution of an application and readings from an external piece of hardware. A small set of expensive motherboards include a power meter as part of its Intelligent Platform Management Interface (IPMI) firmware which allows readings of power and energy at a higher sampling rate than external power meters. However, these tools have proven not to be accurate enough [47]. SnuMAP takes another approach. It focuses only on Intel architectures and uses RAPL power models to measure energy and power. As described on chapter 2 RAPL, despite not being accurate, if compared to external hardware, it understands the behavior of power on changing workloads and can predict energy usage for CPU, memory and graphics accurately [20, 22, 35, 66]. SnuMAP enables the measurement of power and energy of CPU, memory, and graphics via RAPL through its `sysfs` interface. It then logs and synchronizes this information with the timestamp of the target parallel application. SnuMAP can also correlate the power consumption of individual applications as it understands the core allocation and power consumption at a given set of times thought the execution of the application on the system.

5.3 User-level Interfaces

This section explains the various interfaces which expose SnuMAP profiling to virtually any parallel application independent of the programming language or runtime framework used.

5.3.1 Dynamic Interface: Library Interpositionining

As mentioned in previous sections, SnuMAP refers to as a *dynamic* interface to its library-interpositioning module that allows the interception of entry and exit points of any supported parallel application. This interface is shown to the user in the form of an executable (`snumap_main`) which allows the easy execution of binaries. For example, if we have a binary called `foo` which can be executed on the terminal using the command: `$ ./foo`, the only modification necessary to profile this application under SnuMAP is to prepend the main entry point component: `$ snumap_main foo`.

This dynamic interface can be extended further depending on the runtime framework used. SnuMAP supports the addition of methods to perform library interpositioning in order to collect runtime-specific data. For example, OpenMP applications can be further characterized by what parallel section each thread belongs to within a parallel application. This information is available from the library implementation of OpenMP. The de facto GNU implementation of OpenMP on most Linux boxes is GOMP [31]. This library implements two routines (`GOMP_parallel_start` and `GOMP_parallel_end`) that are intercepted by SnuMAP and allow it to determine when a parallel section is started or ends. Similar to this example other runtime-frameworks can add additional columns of data to SnuMAP trace files and use the same visualization tools to correlate profile information even further.

5.3.2 Static Interface: Shared Library

SnuMAP tries to be general enough by providing a shared library implementation as a gateway to enable an application to be profiled. Binaries can bind to this application programming interface (API) either with code instrumentation, C-bindings or Foreign Function Interfaces (FFI). This API is simple and

Table 5.1: SnuMAP communication API.

Function Signature	Description
`snumap_start_profiling (char* conf_path)`	Marks this parallel application to track its context switches, and according to the given configuration file it will also set sampling rates and modules to load to perform profiling. This configuration file is in `TOML` format and it is described in depth in SnuMAP's user-documentation.
`snumap_stop_profiling (void)`	Marks the termination of this parallel application, and prepares the trace log files for later visualization and analysis.

minimal as it only signals the start and end of a parallel application. Table 5.1 describes this API.

Other cases: Example Java `JNI`

Java applications run on top of the Java Virtual Machine (JVM) so SnuMAP is not able to provide an easy way to profile these applications. The solution is the Java Native Interface (JNI); which allows Java programs to execute native code via a dynamic linker. However, it requires a very particular and error-prone format. SnuMAP open-source site provides multiple examples we include multiple examples that show how to profile Hadoop map-reduce jobs, like `wordCount`, using a collaboration between JNI and the static SnuMAP interface.

A shared library is created with specific function names that depend on the project namescope and class names. With this naming convention the Java dynamic linker is able to find the SnuMAP interface methods, and call them from a Java program. Due to this unusual format, a general solution is not

possible for Java applications. Instead, extra modifications to match a target Java application hierarchy (package name, and class name) in the shared library function names are required[2]. Additionally, it is also required to load the shared library statically, and explicitly call the SnuMAP interface methods from within the target Java program to enable profiling.

5.4 Visualizations

SnuMAP also provides a set of graphical tools that enable the analysis and visualization of all profile data. In this section, we present a list of possible graphs that can be generated by using SnuMAP as a profile and visualization tool. All visualization are constructed as Python Matplotlib Figures [37]. Therefore, SnuMAP visualizations inherit all the dynamic behavior of Matploblib; zoom, scrolling, cutting, saving, changing axis, etc.

All collected performance counters including memory and power consumption information can be represented in time-series plots. Therefore, according to SnuMAP given configuration each performance counter creates its own axis within the same plot instance with the X-axis synchronized across all figures. This allows direct correlation among different types of information.

Additionally, SnuMAP allows filtering information on an application-level correlating system level information on top of each application that took part of a profile execution. Note that not all collected hardware performance counters can be correlated to each individual application as only system level information is available. For example, power consumption per application can be estimated but not accurately displayed as power measurement are only available at the package level.

[2]Note that modifying only the method name is sufficient.

5.4.1 Core vs. Time

Core vs. time visualizations are based on Gantt charts and show, on the y-axis, every core of the target architecture, and, on the x-axis, the time (increment in `jiffies`). With this setup, SnuMAP can plot the elapsed time an application's thread executed on each core As for programs that use one of the supported runtime programming models, SnuMAP can produce a core vs. time plot to distinguish each parallel section. This makes it easier to diagnose the possible causes of performance bottlenecks, as each colored parallel section is identified by its function's memory address. These visualizations are two powerful tools that can enable a deeper understanding of parallel application patterns in the underlying architecture for both developer and resource managers.

Chapter 6

Overhead

An important aspect of any profiler is the overhead it introduces on both the system and the target applications. This overhead can not only slow down the normal execution of an application but also affect the accuracy of measurements and therefore lead to wrong conclusions and correlations. We have claimed through this thesis that SnuMAP is lightweight. In this chapter, we elaborate on this via experiments and discuss possible ways SnuMAP can improve upon to avoid or completely eliminate the minor overhead that there still exists. All data shown in this chapter was obtained from a 20-core Intel platform described in Table 7.1 using eight OpenMP applications from NPB3.3 and PARSEC parallel benchmarks.

Profiling an application with SnuMAP is a three-step process:

1. Start application.
2. Collect performance information.
    - Track context-swithces.

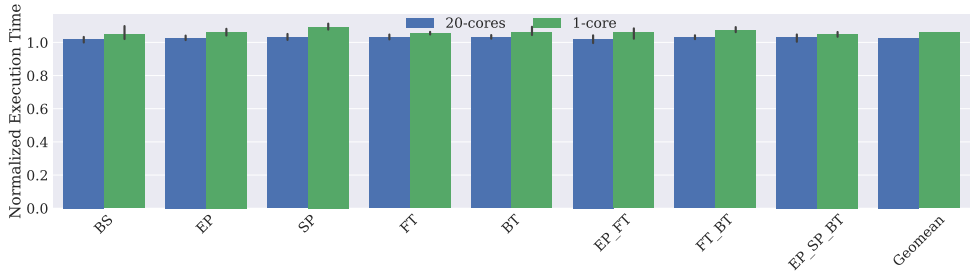


Figure 6.1: Normalized execution time of SnuMAP's context-switch tracker overhead across multiple applications for 20-cores vs. 1-core cases.

- Read performance counters and write count to a file.

3. On application termination write context-switch tracking information to a trace file.

Note that tracking the context-switches and reading performance counters happen in parallel thus their overhead can be hidden somewhat, regardless of this we have divided this chapter into two sections that discuss the overhead of SnuMAP's context-switch tracker and PMU monitor separately.

6.1 Context-switch Tracker

As discussed in Chapter 5 SnuMAP requires a kernel-patch in order to support context-switch tracking capabilities. This adds a small latency to the target application as it stores the timestamp for every context switch. In Figure 6.1 we show the normalized execution time of eight OpenMP parallel applications. In average SnuMAP's context-switch tracker generates a shy 3% of overhead in terms of execution time. This overhead is highly dependant on how the scheduler enqueues the tasks and the number of cores. The number of cores will affect this latency as for a parallel application depending on how scalable it is the work that can be distributed to multiple cores will reduce the total overhead as the

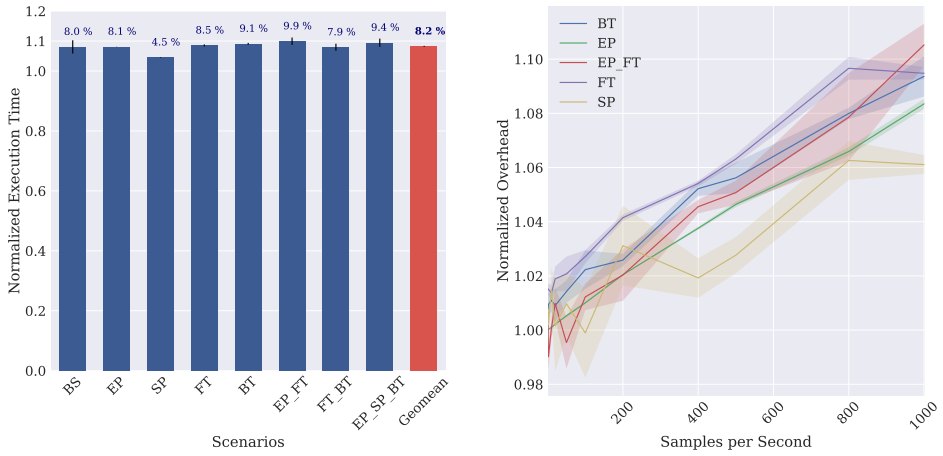
context-switches will happen in parallel. This is shown clearly in Figure 6.1 there is a small difference between the overhead when restricting the applications to a single core as opposed when running on multiples cores on all applications used. In average this difference is around 2% of the overall execution time. After further experiments, we discover that our metric of time (jiffies) sometimes is not precise enough and context switches can happen within a single increment of the clock. This is why it would be possible to reduce this overhead slightly by merging multiple context-switch time stamps that fall within the same clock increment. Other than the previously mention optimization fine-grain code and data-structure optimization could be implemented to further reduce the latency introduced by the context-switch tracker.

6.2 PMU Monitor

6.2.1 Reading Performance Counters

One of the major sources of overhead from SnuMAP's PMU monitor is the communication between SnuMAP and low-level kernel interfaces that enable reading performance counters. Regardless of the intermediate interface used to read hardware-dependant counters most of them require to pass via a sub-system or kernel-interface. For example, SnuMAP currently uses the perf subsystem to read general and commonly used performance counters. Reading perf events eventually result in calls to the `read()` system call which introduces large latency as system calls are entirely managed by the operating system. A solution to bypass the operating system exists for `x86` architectures using the `RDPMC`[1] instruction. This instruction allows reading performance counters directly from the physical register and storing the current count on specified regions of memory enabling a similar user-level interface as perf events. This methodology is

[1]RDPMC stands for: Read Performance-Monitoring Counters.



(a) Normalized overhead.

(b) Sampling rates' effect.

Figure 6.2: Experimental results of SnuMAP PMU monitor's overhead on various workloads and sampling rates.

somewhat similar to reading MSR registers. Nevertheless, note that usually MSR registers are read using Linux default interface (`/dev/msr`) which completely defeats the purpose as it requires to context-switch into the operating system.

6.2.2 Discussion

On Figure 6.2 we explore the overhead introduced by SnuMAP's PMU monitor from two perspectives. First, on Figure 6.2a we show the normalized performance loss (in terms of total execution time) of eight OpenMP scenarios. As can be seen, using SnuMAP's PMU monitor at the highest sampling rate (1 millisecond) the execution time of the target applications increases by 8.2%, which can be think of as a minor overhead considering that this includes the time it takes to write the register's counts to a file (writing trace files overhead). Second, on Figure 6.2b we show the overhead introduced by SnuMAP in specific scenarios with different sampling rates. It is clear that if not high-resolution at

the millisecond level is needed it is possible to reduce the overhead from 8.2% to approximately 1% in the case of `EP_FT` by simply changing the PMU monitor sampling rate from one to five milliseconds. However, it must be noted from the figures that despite we correlate the overhead to be dependant on the target application, in fact, we expect this overhead to affect any application equally and only depend on the sampling rate as shown in Figure 6.2b, and interference introduced by the operating system and additional overhead when writing measurements to trace files. Despite not being a negligible overhead, the worst case results were presented and the modular capabilities of SnuMAP enable it to reduce this overhead further by only profiling performance counters that are required. Additionally, a binary format representation for SnuMAP trace files will reduce the amount of data needed to be written to a file at each sampling rate reducing the overhead accordingly.

Chapter 7

Use Cases and Evaluation

In this chapter we present a use-case based evaluation of the proposed profiler, using a large variety of platforms and architectures. SnuMAP proved to be of great help on all of them and aided the analysis and understanding of performance bugs, the interaction with the underlying system and the effects of the scheduler and load balancer.

7.1 Target Architectures

We evaluated multiple use cases using five different many-core systems, detailed described in table 7.1. The AMD and Intel servers represent NUMA systems with many-core and multi-socket capabilities, while the Tile-Gx38 platform represents a mesh-style many-core processor.

7.2 Target Applications

We selected several OpenMP applications from the SNU-NPB3.3 [68] implementation of the NAS parallel benchmark [6] and PARSEC 3 [10] to showcase

Table 7.1: Description of Target Architectures.

Name	Processor	Frequency	DRAM	Cores	Nodes	Linux Kernel
AMD32 [3]	Opteron 6276	2.4GHz	32GB	32	2	3.19
AMD64 [4]	Opteron 6380	2.5GHz	126GB	64	8	4.4
Tilera [56]	Tile-Gx8036	1.2GHz	32GB	36	2	3.14
Intel72 [38]	Xeon E7-8870 v3	2.1GHz	504GB	72	4	4.16
Intel20 [38]	Xeon E5-2630 v4	2.2GHz	126GB	20	2	4.16

the capabilities of SnuMAP on massively parallel OpenMP workloads. Additionally, we also included other.

7.3 Space-shared Scheduling Scenario

Modern operating system and parallel programming models are known to have conflicting goals and assumptions. Space-shared scheduling has been proposed as a means to address performance and interference issues [74]. This approach splits the scheduler into two: (a) *a coarse grained* resource manager, represented by a space-shared scheduler, and (b) *a fined-grained* resource manager, representing parallel programming models that provide at-runtime decisions for resource allocation. This is an exciting approach, and multiple prototypes have been proposed [7, 53, 77, 16].

We decided to use SnuMAP core vs. time plotting capabilities to see in detail how one of this approaches implements a space-shared runtime system that works on top of the Linux kernel and scheduler. The runtime implementation chosen has a simple scalability model based on CPU utilization, so with increase CPU utilization more cores should be given to an application. Similarly, if the CPU utilization falls below a threshold, the runtime system salvages some of those under-utilized cores. Figure 7.1 shows the core vs. time plot for a co-located scenario with three application (EP, BT, and SP). From the Fig-

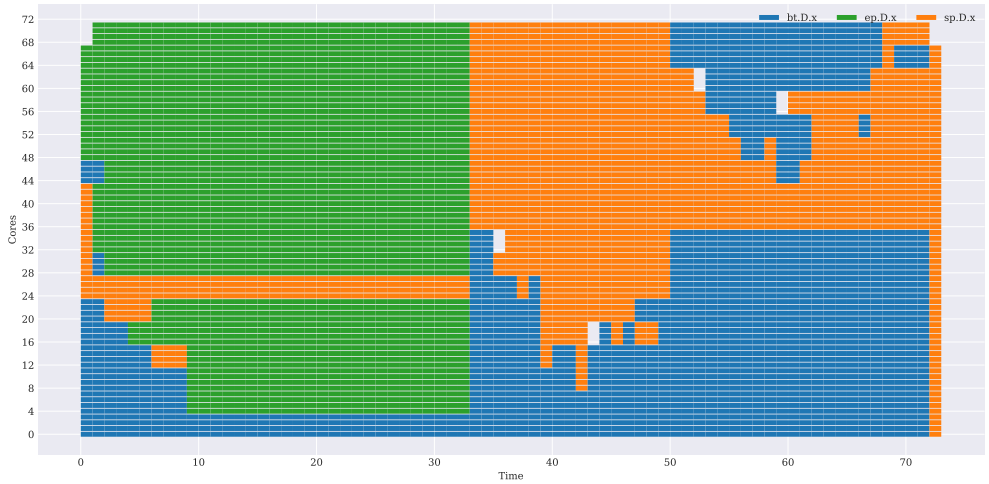


Figure 7.1: SnuMAP core vs. time plot for three-concurrent applications (EP, FT and SP all class D) on the Intel72 platform running on top of a space-shared runtime system based on a simple scalability model.

ure, we can observe that because threads from all the parallel applications are fixed to specific cores, there are not nearly as much thread migrations as in the other cases when using the default Linux scheduler. Also, note how all cores are utilized by all application but the number that is assigned to each application changes based on the phase of each application dynamically. This simple scenario demonstrates how useful SnuMAP can be by resource managers, nor only when developing new scheduling algorithms but also as a way to test and corroborate the proposed approach.

Similar to core resource, SnuMAP allowed us to visualize the power consumption impact the runtime system has for both CPU and DRAM. In Figure 7.2 nodes two and three are the ones that account for must of the DRAMs power consumption this can be explained by the fact that SP is a memory bound application, and it is fully allocated on parts of nodes three, two and one during its execution. Moreover, the DRAM power consumption of nodes zero

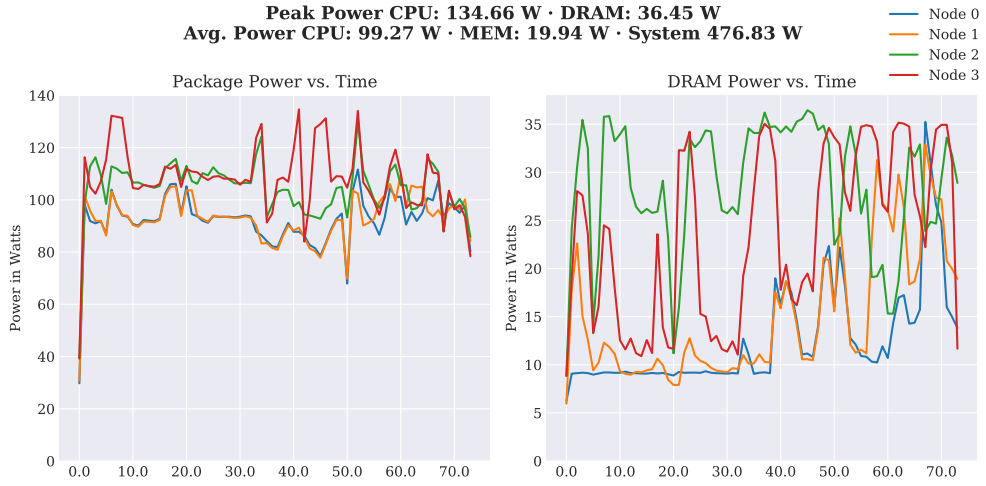


Figure 7.2: SnuMAP CPU and DRAM power consumption plots for three-concurrent applications (EP, FT, and SP all class D) on the Intel72 platform running on top of a space-shared runtime system based on a simple scalability mode.

and one remains relatively low until approximately second 40. We can infer this happens because before that point most of the cores on nodes zero and one are allocated for application EP which is a CPU intensive application.

7.4 Multiple Applications Scenarios

In this section we present various OpenMP co-located scenarios. We show how SnuMAP can be used to better understand the interaction with the underlying platform, other co-located applications and each application's patterns regarding power consumption, CPU utilization and memory usage.

For the OpenMP workloads BT, MG, and LU represent memory intensive scalable applications, whereas EP is CPU intensive and CG issues irregular memory accesses. Each application computes over a specific problem size, also known as the class size. We executed all possible permutations of pairs of exist-

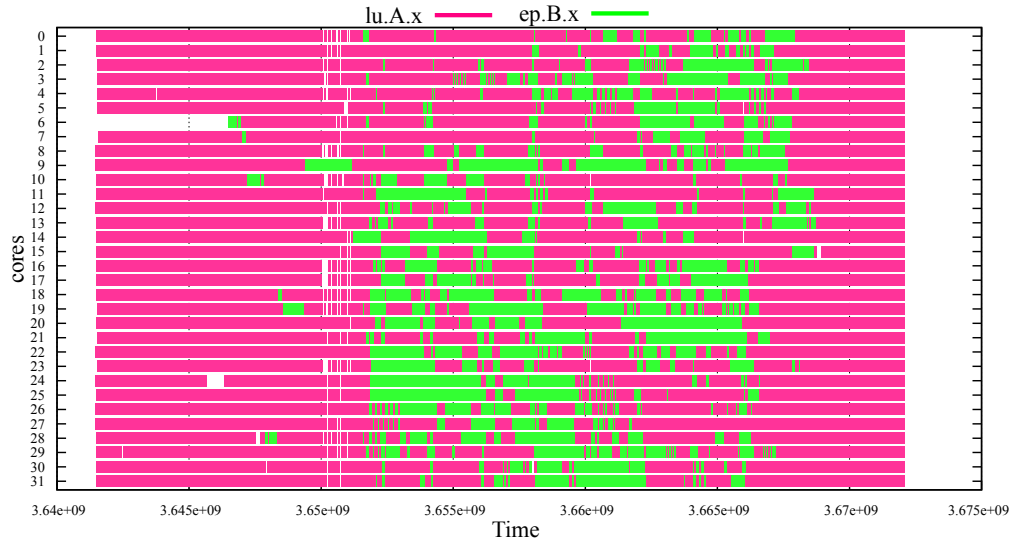


Figure 7.3: SnuMAP core vs. time plot for two-concurrent applications (LU class A, and EP class B) on the AMD 32 platform.

ing applications of the SNU-NPB3.3 benchmark suite on all architectures but present here a subset of the most interesting co-located scenarios.

7.4.1 Two Applications on AMD32

Figure 7.3 shows a core vs. time plot generated by SnuMAP after profiling the concurrent execution of LU (class size A) and EP (class size B) on the AMD 32 architecture. A detail interaction between these two apps and the underlying system is easily noted via SnuMAP plot. The effects of the Linux scheduler and the load balancer are visible. Note how the work is balanced among all the cores, and how the scheduler seems to favor LU at the beginning of the execution, to later start making room for EP tasks. An interesting particularity is the lack of work for a short period on core number six. SnuMAP core vs. time plots are especially useful for visualizing synchronization bottlenecks; at the center-left of the graph we observe that LU seems to stop executing for

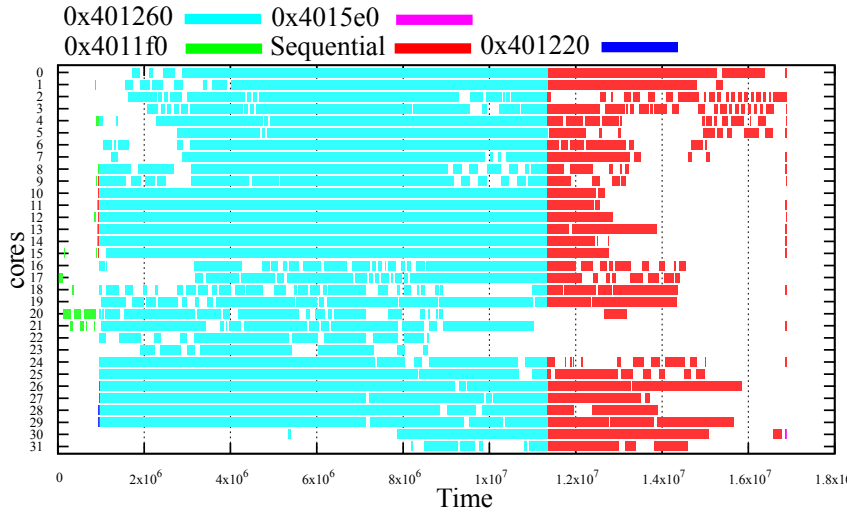


Figure 7.4: SnuMAP core vs. times plot distinguishing individual parallel sections of a single application (EP class B) in a concurrent applications scenario (EP class B, and LU class A) on the AMD 32 architecture.

short periods of time, this is due to the many synchronizations generated by many small parallel sections.

Additionally, Figure 7.4 shows the core vs. time plot generated by SnuMAP after profiling the concurrent execution of EP (class size B) and LU (class size C) on the AMD 32 architecture. These are the same trace data as used for Figure 7.3. Nevertheless, in this case, we isolated the profile data of **only** the EP application, and by using the extra trace file generated for this OpenMP application, SnuMAP can plot individual parallel sections using different colors. From this type of plot, it becomes apparent which code sections is executed at a given time. By looking at the results, it seems that the parallel sections labeled `sequential`[1], and `0x401260` are the longest running sections. Also, the interaction between parallel sections is minimal as they appear to execute in

[1] *sequential* refers to the code section that does not spawn more tasks. Instead, it is the one called from a parallel code section. In the case of OpenMP, this happens on nested for loops which are parallelized via nested *pragmas*.

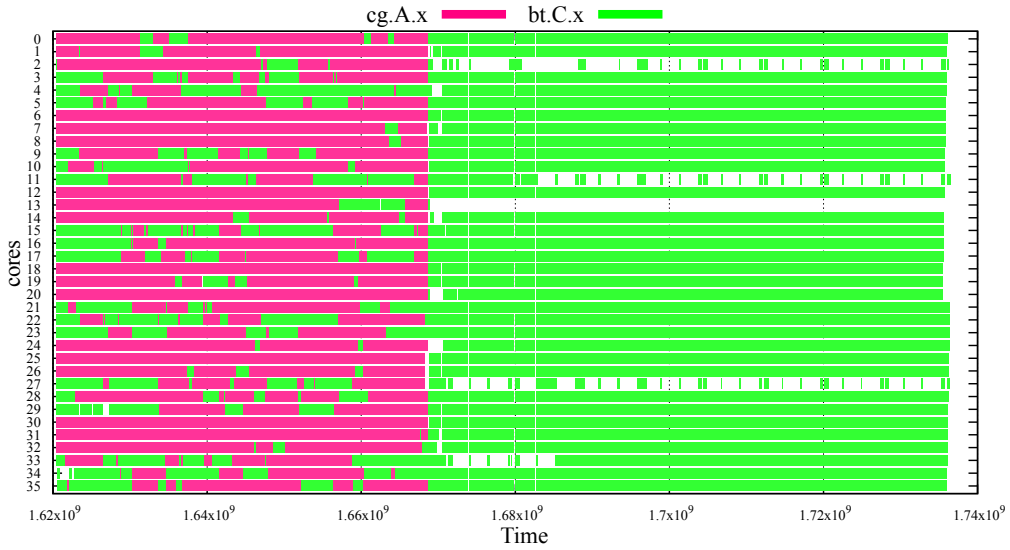


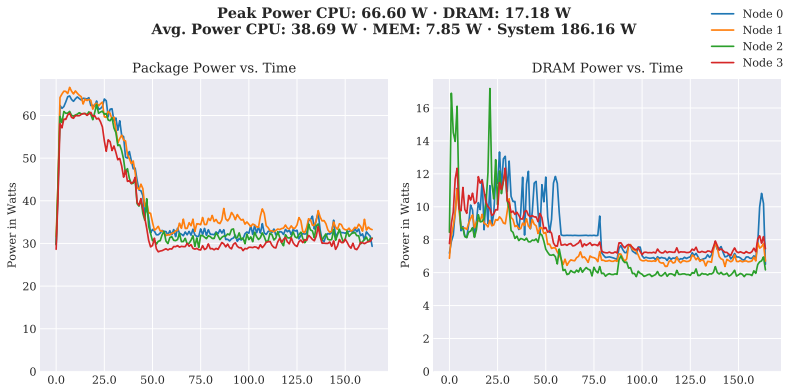
Figure 7.5: SnuMAP core vs. times plot for two-concurrent applications workload (CG class A, and BT class C) on the Tilera platform.

batch (one after another).

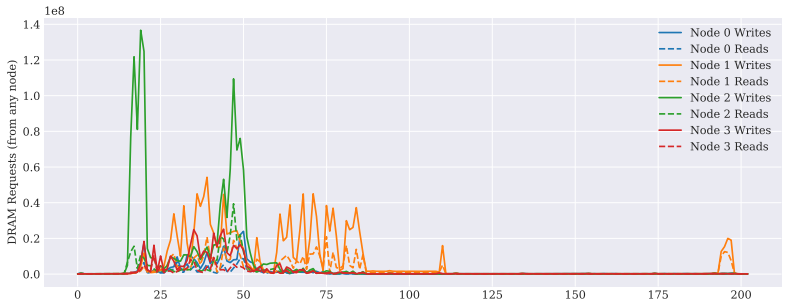
7.4.2 Two Applications on Tilera

Figure 7.5 shows the core vs. time plot generated by SnuMAP after profiling the concurrent execution of CG (class size A) and BT (class size C) on the Tile-Gx36 architecture. This is an interesting result because it shows how the architecture and the scheduler allocate cores to both applications with high resolution. The overall allocation of CG interleaves the allocation of BT in sets of three cores. The result of such an allocation is an equal division of the processor in half. This might be the case so that each application is closer (in hop distance) to the nearest memory bank Also, note the center of the plot; two empty vertical lines are visible, these represent two synchronization points of the BT application.

7.5 Python Scenario



(a) CPU Package and DRAM power consumption.



(b) DRAM requests.

Figure 7.6: SnuMAP's plots for Python workload on the Intel72 system.

Python provides out of the box support for a FFI library called CFFI. This library enables direct communication with SnuMAP allowing profiling on Python workloads very easily. To showcase this, we instrumented and executed a simple Python parallel workload that uses the `multiprocessing` library to chunkenize work into as many tasks as there are cores on the system. As can be seen in Listing 7.5 only a few lines of code need to be added to enable the communication with SnuMAP shared library (lines 10-16 and 22-23).

Listing 7.1: Instrumentation example to enable SnuMAP profiling on a Python script via the CFFI standard library.

```
import multiprocessing
from cffi import FFI
from .factorize import factorize, distribute_work

ffi = FFI()

# allocate a char* for SnuMAP's config path & return a pointer to it
snumap_config = ffi.new('char []', './snumap_config.toml')

# define SnuMAP's API
ffi.cdef('void snumap_start_profling(char* conf_path);')
ffi.cdef('void snumap_stop_profling(void);')

# open SnuMAP shared library and start profiling
snumap_lib = ffi.dlopen('libSnuMAP.so')
snumap_lib.snumap_start_profling(snumap_config)

# actual work: factorize all ints in workset on all cores
work_set = list(range(0, 950000))
distribute_work(work_set, multiprocessing.cpu_count())

# stop profiling
snumap_lib.snumap_stop_profiling()
```

In Figure 7.6 we present part of SnuMAP's profiling results after executing the Python workload on the Intel72 platform. As can be seen in Figure 7.6(a) the most power consumed by both DRAM and CPU for this workload happens within the first 30 seconds of execution. This is an interesting finding because despite that the total execution time of this workload is much longer than this initial 30 seconds most of the power is consumed within the first section. If we analyze the workload code in more detail, we realize that the `factorize` function despite being slightly CPU intensive it executes rather quickly. However, it needs a large input work set. This work-set, the division of work, scheduling, and creation of tasks is all done before the actual execution by the `multiprocessing` library. Hence, there is only spikes on both power and memory requests (Figure 7.6(b)) at the beginning of the execution. Note also the strong correlation between DRAM power consumption and memory requests for each one of the nodes in the system.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, we introduced SnuMAP: a light-weight many-core system profiler. The proposed profiler is capable of collecting core allocation information, performance information from both CPU and DRAM, as well as power information for single or co-located parallel applications. We have evaluated the functionalities of SnuMAP on four different Linux-based manycore platforms. Various case studies show the amount of useful information that SnuMAP can provide, even at a short glance. Software developers can efficiently use a single set of tools all included in SnuMAP's framework to better understand performance bottlenecks, power consumption and memory behavior on multiprocessing systems. Additionally, SnuMAP automatically profiles any binary without the need for code instrumentation in most cases via library interpositioning and provides a simple shared library gateway as API to enable profiling.

8.2 Future Work

SnuMAP profiles various important hardware performance counters that are critical in understanding performance and interaction with the underlying architecture. However, not all hardware vendors provide the same clean interface. Various improvements can be made to support more architectures and more hardware performance counters. Moreover, application-dependant information is usually more useful than a holistic view of the system, so we plan to extend SnuMAP capabilities to correlate different data points to application-dependent information. Also, we would like to explore the use of different type of visualization to represent information better, as well as experiment with other types of data storage formats like binary files or pseudo files systems.

Bibliography

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency Computation Practice and Experience*, 22(6):685–701, 2010.

[2] Advanced Micro Devices, Inc. White paper: ACP — The Truth About Power Consumption Starts Here. Technical report, 2010.

[3] AMD. AMD Opteron 62000 Series Processors Reference Document. `https://www.amd.com/Documents/Opteron_6000_QRG.pdf`.

[4] AMD. AMD Opteron 6300 Series Processors. `http://www.amd.com/en-us/products/server/opteron/6000/6300`.

[5] T. E. Anderson and E. D. Lazowska. Quartz: A tool for tuning parallel program performance. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '90, pages 115–125, New York, NY, USA, 1990. ACM.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S.

Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[8] F. Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, Sept. 17–20 2000.

[9] J. L. Berral, I. n. Goiri, R. Nou, F. Julià, J. Guitart, R. Gavaldà, and J. Torres. Towards energy-aware scheduling in data centers using machine learning. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, pages 215–224, New York, NY, USA, 2010. ACM.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[11] W. L. Bircher and L. K. John. Complete system power estimation using processor performance events. *IEEE Trans. Comput.*, 61(4):563–577, Apr. 2012.

[12] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the*

19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM.

[13] J. Breitbart, J. Weidendorfer, and C. Trinitis. Automatic co-scheduling based on main memory bandwidth usage. In *Proceedings of the 20th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, JSSPP '16, May 2016.

[14] R. Buyya, C. Vecchiola, and S. T. Selvi. *Mastering Cloud Computing: Foundations and Applications Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[15] J. Caubet, J. Gimenez, J. Labarta, L. D. Rose, and J. S. Vetter. *A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications*, pages 53–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[16] Y. Cho, S. Oh, and B. Egger. Adaptive space-shared scheduling for shared-memory parallel programs. In *JSSPP*, 2016.

[17] W. Choi, H. Kim, W. Song, J. Song, and J. Kim. epro-mp: A tool for profiling and optimizing energy and performance of mobile multiprocessor applications. *Sci. Program.*, 17(4):285–294, Dec. 2009.

[18] Cortesi D. and Fier J. and Wilson J. and Boney J. Origin 2000 and onyx2 performance tuning and optimization guide. Technical report, 2001.

[19] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194, Aug 2010.

[20] S. Desrochers, C. Paradis, and V. M. Weaver. A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, pages 455–470, New York, NY, USA, 2016. ACM.

[21] B. Diniz, D. Guedes, W. Meira, Jr., and R. Bianchini. Limiting the power consumption of main memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 290–301, New York, NY, USA, 2007. ACM.

[22] J. Dongarra, H. Ltaief, P. Luszczek, and V. M. Weaver. Energy footprint of advanced dense numerical linear algebra using tile algorithms on multicore architectures. In *2012 Second International Conference on Cloud and Green Computing*, pages 274–281, Nov 2012.

[23] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach-temam. Aftermath : A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. *7th workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2014)*, (1):1–13, 2014.

[24] U. Drepper. What every programmer should know about memory, 2007.

[25] Dtrace. `http://dtrace.org`.

[26] G. Eisenhauer, K. Schwan, W. Gu, and N. Mallavarupu. Falcon-toward interactive parallel programs: the on-line steering of a molecular dynamics application. In *High Performance Distributed Computing, 1994., Proceedings of the Third IEEE International Symposium on*, pages 26–33, Aug 1994.

[27] M. P. Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[28] Ftrace. `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`.

[29] K. Fürlinger and M. Gerndt. *ompP: A Profiling Tool for OpenMP*, pages 15–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[30] T. Gleixner and D. Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Proceedings of the Ottawa Linux Symposium*, 2006.

[31] GNU Project. GNU libgomp. `http://gcc.gnu.org/onlinedocs/libgomp/`, 2018.

[32] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[33] B. Gregg. The flame graph. *Commun. ACM*, 59(6):48–57, May 2016.

[34] M. Haardt and M. Coleman.

[35] D. Hackenberg, T. Ilsche, R. Schöne, D. Molka, M. Schmidt, and W. E. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204, April 2013.

[36] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[37] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[38] Intel. Intel Xeon Processor E7-8870 v3. `http://ark.intel.com/products/84682/Intel-Xeon-Processor-E7-8870-v3-45M-Cache-2_10-GHz`, 2018.

[39] Intel Corporation. Intel Performance Counter Monitor - A better way to measure CPU utilization. `www.intel.com/software/pcm`.

[40] Intel Corporation. Intel X-series Processors. `https://ark.intel.com/products/series/123588/Intel-Core-X-series-Processors`.

[41] Intel Corporation. Intel® Turbo Boost Technology in Intel® Core™ Microarchitecture (Nehalem) Based Processors.

[42] Intel Corporation. RAPL Power Meter. `https://01.org/rapl-power-meter`, 2010.

[43] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer Manuals. `https://software.intel.com/en-us/articles/intel-sdm`, 2018.

[44] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 347–358, Washington, DC, USA, 2006. IEEE Computer Society.

[45] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO 36, pages 93–, Washington, DC, USA, 2003. IEEE Computer Society.

[46] D. P. J. Instruction-based sampling: A new performance analysis technique for amd family processors. 2007.

[47] R. Kavanagh, D. Armstrong, and K. Djemame. Accuracy of energy model calibration with ipmi. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 648–655, June 2016.

[48] J. Keniston. Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps abstract. In *Proceedings of the Ottawa Linux Symposium*, June 2007.

[49] KProbes. `https://www.kernel.org/doc/Documentation/kprobes.txt`.

[50] R. Lachaize, B. Lepers, and V. Quéma. MemProf: A Memory Profiler for NUMA Multicore Systems. *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, page 5, 2012.

[51] S. Li, T. Hoefler, and M. Snir. Numa-aware shared-memory collective communication for mpi. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 85–96, New York, NY, USA, 2013. ACM.

[52] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. Numa-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[53] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client os. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.

[54] I. Ljubuncic. *Problem-solving in High Performance Computing: A Situational Awareness Approach with Linux*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016.

[55] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova. The Linux scheduler: a decade of wasted cores. *Proceedings of the Eleventh European Conference on Computer Systems - EuroSys '16*, pages 1–16, 2016.

[56] Mellanox. Tile-Gx Processor Family. `http://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx36.pdf`.

[57] K. Meng, R. Joseph, R. P. Dick, and L. Shang. Multi-optimization power management for chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 177–186, New York, NY, USA, 2008. ACM.

[58] Mohr B. and Malony AD. and Hoppe H-C. and Schlimbach F. and Haab G. and Hoeflinger J. and Shah S. A performance monitoring interface for openmp. 2002.

[59] T. Mück, S. Sarma, and N. Dutt. Run-dmc: Runtime dynamic heterogeneous multicore performance and power estimation for energy efficiency. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, CODES '15, pages 173–182, Piscataway, NJ, USA, 2015. IEEE Press.

[60] C. Möbius, W. Dargie, and A. Schill. Power consumption estimation models for processors, virtual machines, and servers. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1600–1614, June 2014.

[61] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, 1996.

[62] NVIDIA Corporation. NVIDIA Management Library (NVML), 2018.

[63] A.-C. Orgerie, M. D. d. Assuncao, and L. Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Comput. Surv.*, 46(4):47:1–47:31, Mar. 2014.

[64] C. Pheatt. Intel® threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, Apr. 2008.

[65] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, Mar. 2012.

[66] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, March 2012.

[67] M. Schulz and B. R. de Supinski. Pnmpi tools: a whole lot greater than the sum of their parts. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–10, Nov 2007.

[68] S. Seo, G. Jo, and J. Lee. Performance characterization of the nas parallel benchmarks in opencl. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 137–148, Nov 2011.

[69] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[70] SystemTap. `https://sourceware.org/systemtab/`.

[71] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with papi-c. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[72] G. Tesauro, R. Das, H. Chan, J. O. Kephart, C. Lefurgy, D. W. Levine, and F. Rawson. Managing power consumption and performance of computing systems using reinforcement learning. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, pages 1497–1504, USA, 2007. Curran Associates Inc.

[73] Vadim Filanovsky, Brendan Gregg, Martin Spier, and Ed Hunter. Netflix flamescope, 2018.

[74] A. Vajda. *Programming Many-Core Chips*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[75] A. van de Ven and A. Kok. powertop(8). Linux man pages., 2007.

[76] J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 240–250, New York, NY, USA, 2002. ACM.

[77] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 3–14. ACM, 2010.

[78] F. Wolf and B. Mohr. EPILOG binary trace-data format). Technical Report ZAM-IB-2003-06, Jülich.

[79] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 50–50, Nov 2000.

[80] O. Zaki. Toward scalable performance visualization with Jumpshot, 1999.

[81] K. Zhang, R. Chen, and H. Chen. Numa-aware graph-structured analytics. *SIGPLAN Not.*, 50(8):183–193, Jan. 2015.

[82] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 629–644, Broomfield, CO, Oct. 2014. USENIX Association.

[83] M. Zheng, M. Sun, and J. C. S. Lui. Droidtrace: A ptrace based android dynamic analysis system with forward execution capability. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 128–133, Aug 2014.

요약

이 논문에서는 매니코어 시스템을 위한 모듈 방식의 공개 소스 추적 프로파일러인 SnuMAP를 제안한다. SnuMAP은 응용 프로그램마다 코어 분배, 파워, CPU, 그리고 메모리 이용률과 같은 관심있는 다양한 데이터에 관한 전반적인 시스템 관점을 제공한다. 게다가, SnuMAP은 가볍고 해당 응용 프로그램의 소스 코드의 수정이 필요 없으며, 또한 병렬 처리 응용 프로그램의 성능도 감소시키지 않는다. 대신, 응용 프로그램 개발자와 매니코어 자원 관리자로부터 가치 있는 정보 및 이해를 필요로 한다. 이러한 종류의 도구는 시스템 이용률을 증가시키는 데 목적을 둔 현대의 매니코어 시스템의 많은 병렬적 워크로드의 동시 스케줄링처럼 점점 더 중요해지고 있다. 우리는 SnuMAP을 다양한 연구 과제에 사용할 수 있도록 하며 이 논문에서 SnuMAP에서 제공하는 시각 자료와 데이터로 찾을 수 있는 중요한 결과 예시를 보여준다. 이 과제는 원래 간단한 공개 소스 프로파일러에서 출발하였고 점점 복잡한 분석 도구로 발전하였다. 더 많은 정보는 `http://csap.snu.ac.kr/software/snumap` 에서 확인할 수 있다.

주요어: 프로파일러, 매니코어 시스템, NUMA, 파워, Intel RAPL

학번: 2016-22100