SoftwareX 6 (2017) 150–154

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

Mammut: High-level management of system knobs and sensors

Daniele De Sensi [*], Massimo Torquati, Marco Danelutto

Department of Computer Science, University of Pisa, Pisa, Italy

ARTICLE INFO

Article history:
Received 20 October 2016
Received in revised form 16 June 2017
Accepted 30 June 2017

Keywords:
Energy profiling
DVFS
System management
Remote management

ABSTRACT

Managing low-level architectural features for controlling performance and power consumption is a growing demand in the parallel computing community. Such features include, but are not limited to: energy profiling, platform topology analysis, CPU cores disabling and frequency scaling. However, these low-level mechanisms are usually managed by specific tools, without any interaction between each other, thus hampering their usability. More important, most existing tools can only be used through a command line interface and they do not provide any API. Moreover, in most cases, they only allow monitoring and managing the same machine on which the tools are used. MAMMUT provides and integrates architectural management utilities through a high-level and easy-to-use object-oriented interface. By using MAMMUT, is possible to link together different collected information and to exploit them on both local and remote systems, to build architecture-aware applications.

Code metadata

Current code version	v1.0.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-16-00087
Legal Code License	MIT License
Code versioning system used	git
Software code languages, tools, and services used	C++, Makefile
Compilation requirements, operating environments & dependencies	C++11 compiler, Linux
If available Link to developer documentation/manual	http://danieledesensi.github.io/mammut/manual.html
Support email for questions	d.desensi.software@gmail.com

1. Motivation and significance

Tuning the hardware related control mechanisms (i.e. *knobs*) offered by modern computing platforms, and monitoring their impact on performance and power consumption of applications is an important field of computer science research. Providing high-level views of such mechanisms is of paramount importance, to enable a better understanding of their potentiality and to drive the development of new applications and runtime systems. Recently, a significant effort has been made to exploit all the low-level features offered by modern multi-core CPUs (Central Processing Units) to find good trade-offs between energy consumption and absolute performance of parallel applications.

For example, researches have explored the use of clock frequency scaling (also known as DVFS — Dynamic Voltage and Frequency Scaling) [1], cores disabling [2], threads pinning [3], hyperthreading [4] and idle states management [5]. Such research is still ongoing [6–8] and new types of low-level knobs like cache memories reconfiguration have been proposed [9]. When those mechanisms are used together in a synergistic way, better results can be achieved [3,10].

However, the tools provided by the operating system are often targeted towards one specific architectural aspect, without any possibility to interact and extract information obtained with other tools, hampering their effectiveness. For example, the `taskset` Linux command, provides the possibility to map an application to a particular set of cores but does not allow to change their clock frequency or to bind memory allocation to specific Non-Uniform Memory Access (NUMA) nodes. Conversely, in other cases, no tools

[*] Corresponding author.
E-mail address: desensi@di.unipi.it (D. De Sensi).

http://dx.doi.org/10.1016/j.softx.2017.06.005
2352-7110/© 2017 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

are directly provided by the platform, and the interactions with the Operating System (OS) must be performed by reading/writing data from/to specific system files. In addition to that, some interactions are only possible on specific platforms, impairing the portability of the software and demanding to the programmer the explicit management of such situations.

In general, low-level architectural mechanisms are managed via an appropriate Application Programming Interface (API). However, when the API is present, it usually does not provide a sufficiently high abstraction level and therefore the integration of the APIs from different tools may result in a time-consuming task. To better clarify this limitation, consider the case of energy consumption monitoring for a given application. On currently available multi-core platforms, it is not possible to monitor each physical core independently, but only groups of physical cores can be monitored (e.g., all the cores on the same CPU). Accordingly, the application programmer should be able to decide on which CPU the application should be executed, monitor its energy consumption and maybe change the clock frequency of the CPU cores. Nevertheless, available energy monitoring APIs [11–14] do not usually interact with other hardware mechanisms, thus only providing the possibility to measure energy consumption. When all these operations need to be performed at once, the user needs to convert data representation between all these tools and force them to work together. Moreover, a standard requirement [15,16] is the possibility to monitor and manage remote systems. However, this feature is not available in most existing solutions.

We propose Mammut, an object-oriented, open-source C++ framework, that provides an high-level interface for the management of hardware related mechanisms on local and remote Linux systems. It aims to solve the issues mentioned earlier, by easing application and software runtime system development. Mammut provides an easy-to-use API to correlate and integrate data coming from different architectural sources, hiding portability issues and providing a homogeneous interface to the user. By using Mammut, the programmer is relieved of the burden of dealing with the details of the specific platform and tool, since these details are handled transparently by the framework. Thanks to its modular design and its open-source nature, Mammut can be easily extended to add new features and mechanisms. Mammut targets multi-core machines and has been successfully tested on different modern Intel, ARM and PowerPC architectures.

The tool most similar to ours is probably *Likwid* [17]. However, it is mostly focused on performance monitoring and does not provide some of the mechanisms available in Mammut (like cores unplugging or processes management). It is mostly designed as a set of tools to be used through a command line interface. Albeit an API exists, it does not provide all the functionalities provided by their command line tools. Moreover, by using *likwid* is not possible to monitor remote machines. This last point is considered a key feature for integrating remote energy-measurement functionality and dynamic management of resources capabilities in modern computing devices like the ones that are becoming increasingly popular in the IoT and Fog computing systems [18]

2. Software description

Mammut is about 10 thousand lines of C++ code structured as a set of modules, each of them managing a given set of functionality. Currently, the following modules are available: Topology (Section 2.2.1), Energy (Section 2.2.2), Cpufreq (Section 2.2.3) and Task (Section 2.2.4).

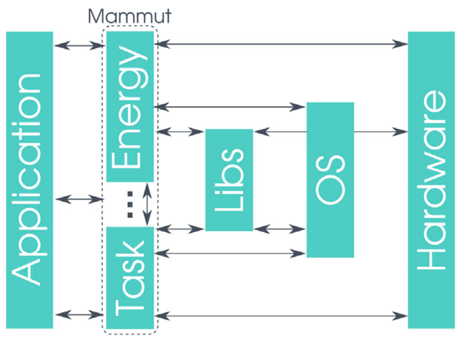

Fig. 1. Mammut architecture.

2.1. Software architecture

The starting point is the `Mammut` object. The application programmer interacts with this object to obtain handles to the different available modules via a `getInstance[ModuleName]` call. Indeed, since different implementations of the same module may exist (e.g. one implementation for each supported architecture or operating system), the `Mammut` object will provide the most suitable implementation according to the underlying system. Each module can provide its functionalities by a direct interaction with the OS, with the hardware, with other modules or by exploiting third-party libraries, as shown in Fig. 1. This allows Mammut to expose a homogeneous interface to the application programmer independently from the underlying system.

2.2. Modules architecture and functionalities

We now briefly describe the main structure of the modules and their main functionalities. Some of the following features (e.g., changing the clock frequency) can only be used by users with privileged rights.

2.2.1. Topology module

This module allows the management of the physical topology of the underlying architecture. We organised the HW components in a hierarchical way: *CPUs*, *Physical Cores* (i.e. the basic computation unit of a CPU) and *Virtual Cores*. *Virtual Cores* represent the different contexts of a *Physical Core* (e.g., Intel's HyperThreading contexts). Indeed, modern CPUs are usually able to provide multiple abstract (*virtual*) cores for each physical core present on the CPU. Each hardware component instance is represented by an object, which can be used to retrieve its sub-components, to turn it off or to get information about the specific version and capabilities of that component. Moreover, *Virtual Cores* allows to explicitly manage idle behaviour, i.e. the number of physical components that should be turned off when that *Virtual Core* has no process to execute. This is a commonly used feature to reduce the energy consumption of applications [5].

2.2.2. Energy module

By using the energy module is possible to read the power consumption of the system or, of some of its components. It is possible to know which types of energy counters are available on the system and to retrieve one of them by querying the module. Each counter provides a `reset()` call to set the cumulative counter to 0 and a `getJoules()` call to retrieve the energy consumed since the last `reset()` call. Each counter may provide additional calls. For example, energy counters available on Intel SandyBridge, IvyBridge and Haswell provide specific functions to read energy consumption of each CPU, of the *Physical Cores* on the CPU, of the

Table 1
Parameters controlled by the different mammut modules.

Module	Topology				Energy	Frequency	Task
Class	CPU	Physical Cores	Virtual Cores	Idle Level	**Counter**	Domain	Process/Thread
Parameters	Identifier Utilisation On/Off Physical Cores Virtual Cores	Identifier Utilisation On/Off Virtual Cores	Identifier Utilisation On/Off Idle Level Flags	Name Description Enable/Disable Exit Latency Time Count Consumed Power	Energy (Entire Machine) Energy (CPU) Energy (OffCores) Energy (DRAM Controller) Energy (GPU)	Governor Frequency TurboBoost Voltage	Priority Mapping to CPU/Cores CPU Utilisation Assembler Instructions Count

DRAM (Dynamic Random Access Memory) controller, and of the integrated graphic card. Thus, by combining information coming from this module and from the Topology module it is possible to monitor energy consumption of specific hardware components.

2.2.3. *Cpufreq module*

This module allows to read and change the frequency and the *governors* of the CPU cores. The *governors* are algorithms used by the OS to manage the clock frequency of the CPUs. If the user needs to implement custom policies and does not want to rely on those provided by the operating system, it is possible to change the clock frequency manually. In general, it is not possible to change the frequency of each core individually. For this reason, we provide the concept of *Domain*, i.e. a set of *Virtual Cores* that must run at the same frequency. Therefore, the user may read and change the *governor*, the frequency or the voltage of a *Domain*. Accordingly, by leveraging on the data provided by the Topology module, is possible to know which cores will be influenced by such operations.

2.2.4. *Task module*

This module allows managing processes and threads running on the system. For example, it is possible to move threads between *Physical* or *Virtual cores*, to change their priority or to read statistics about their execution.

2.3. *Controlled parameters*

Table 1 shows, for each module, a list of exposed parameters. The availability of such parameters depends on the specific architecture.

2.4. *Remote management*

To monitor a remote system, a `mammut-server` should run on that system. From the client side, it is sufficient to specify the address and the port on which the server is listening to when the `Mammut` object is created. All the other calls and interactions do not require any modification. In this way, the user, by simply changing a single line of code, can seamlessly reuse the code written for local system management to manage a remote one. This is possible because the `getInstance[ModuleName]` call, in this case, will return an object with the same interface of the one used for local management but that will act as a client towards the remote server. The interaction with the server is performed via *libprotobuf*[1] library. Note that this type of interaction can also be used to provide access to privileged features (e.g. changing clock frequency) to users with non-privileged rights. For example, we could start a `mammut-server` with privileged rights on a system and run a non-privileged client on the same system to access privileged features. For this reason, `mammut-server` can be executed with a limited set of modules. We are planning to include a more fine-grained capabilities control in the next versions of the library.

[1] https://github.com/google/protobuf

3. Illustrative example

In the following code snippet, we provide a full running example, showing how it is possible to leverage on the information provided by the different modules to shutdown unneeded CPUs (lines 19–23), move the application on a specific CPU (lines 25–2), change its governor and frequency (lines 29–33) and read its power consumption (lines 35–38).

To monitor remote systems, is sufficient to replace `Mammut m` with `Mammut m(new CommunicatorTcp(ipAddress, port))` where `ipAddress` is the address of the remote system and `port` is the port on which the `mammut-server` is listening.

4. Results and comparison

To better understand the advantages of using Mammut, we implemented the same code snippet by using standard tools provided by the operating system and it was composed by almost 700 lines of code (against the 40 of Mammut). Despite being a basic example, this allowed us to estimate the advantages of Mammut and we can expect this gap to significantly increase for scenarios using the more complex features provided by Mammut. Moreover, modifying such code for monitoring remote machines would introduce additional complexity and would lack flexibility, since the serialisation and transfer of the data should be explicitly implemented by the user. On the other hand, to monitor a remote computing node in Mammut only requires specifying the remote address during the initialisation of the library. Mammut allows the application programmer to focus on the concrete part of its application, minimising the effort required to monitor or control the computing architecture. We compared the memory usage and execution time of the Mammut API calls (using the demo applications provided) with that of several command line tools (e.g. `likwid`, `cpupower`, `lstopo`, `turbostat` and `taskset`), showing the results in Table 2.

The first thing we can notice is that most tools are focused on a single feature. On the other hand, the more general tools (Mammut and *likwid*) have a slightly higher memory footprint and execution time, since they need to address different problems and user's necessities. Mammut has a lower memory footprint than *likwid* since it adopts a *lazy initialization* and when using a single module only the data needed for that module is allocated. However, in all the cases the memory footprint is in the order of few hundreds of KiloBytes, since the different tools only need to map some information already available on the filesystem or provided by the operating system in internal data structures.

Differently from Mammut, *likwid* does not have any means to monitor remote architectures. This is an important feature due to the capillary diffusion of computing devices, like in IoT and Fog systems. Moreover, Mammut provides a flexible API, that can be used by the programmer to enhance his application by exploiting information about the underlying architecture. On the contrary, *likwid* was mainly designed for system administrators, since it provides a set of tools to be used from a command line interface. Despite an API has been later added to *likwid*, differently from Mammut, it is not object oriented. Providing an object oriented

```
#include <mammut/mammut.hpp>
#include <iostream>
#include <unistd.h>

using namespace mammut;
using namespace mammut::energy;
using namespace mammut::task;
using namespace mammut::topology;
using namespace mammut::cpufreq;
using namespace std;

int main(int argc, char** argv){
    Mammut m;
    Energy* energy = m.getInstanceEnergy();
    CpuFreq* frequency = m.getInstanceCpuFreq();
    Topology* topology = m.getInstanceTopology();
    TasksManager* tasks = m.getInstanceTask();

    Cpu* cpu = topology->getCpu(0);
    vector<Cpu*> cpus = topology->getCpus();
    for(size_t i = 0; i < cpus.size(); i++){
        if(cpus[i] != cpu){ cpus[i]->hotUnplug(); }
    }

    ProcessHandler* process = tasks->getProcessHandler(getpid());
    process->move(cpu);
    tasks->releaseProcessHandler(process);

    vector<Domain*> domains = frequency->getDomains(cpu);
    for(size_t i = 0; i < domains.size(); i++){
        domains[i]->setGovernor(GOVERNOR_USERSPACE);
        domains[i]->setHighestFrequencyUserspace();
    }

    CounterCpus* c = (CounterCpus*) energy->getCounter(COUNTER_CPUS);
    if(c){ c->reset(); }
    // Do work
    if(c){ cout << "Joules: " << c->getJoulesCpu(cpu) << endl; }
}
```

abstraction is of paramount importance, since it captures a model of the real world and leads to improved maintainability and understandability of the code written by the framework's user [19].

5. Impact

Mammut eases the development and rapid prototyping of algorithms and applications that need to operate on system's knobs or to monitor the system they are running on (or a remote one). This allows the researchers to focus on the algorithm development while the management of such data is performed in an intuitive way by using the high-level API provided by Mammut, shortening the development time.

Mammut has been used by researchers to optimise power consumption of parallel applications and to develop models for the prediction of power consumption and performance of parallel applications [8]. In this context, Mammut has been used to operate on some system parameters (e.g., cores' clock frequency and number of active cores) and to correlate the effect of these parameters on the observed power consumption. This led to the design and development of efficient algorithms to dynamically adapt application's power consumption to the varying workload conditions [20–22]. Mammut also allowed researchers to improve the energy efficiency of *Data Stream Processing* (DaSP) applications by allowing the developers to easily increase or decrease the clock frequency of the CPU [23]. Moreover, Mammut have been integrated into the Nornir framework.[2] Nornir is a framework which can be used to enforce specific constraints in terms of performance and/or power consumption on parallel applications. More recently, it has been used in the RePhrase EU H2020 project[3] as low-level runtime tool for collecting power consumption and other statistics (e.g. intensity of memory accesses) of parallel applications [24].[4] These information are used by the runtime system for deciding which architecture is most suited to execute a specific parallel application. Mammut has also been recently used to measure and optimise power consumption of query processing in web search engines [25]. Moreover, it has been used to evaluate power consumption of parallel benchmarks for multicore architectures [26][5]

2 https://github.com/DanieleDeSensi/nornir
3 http://rephrase-ict.eu/
4 http://rephrase-eu.weebly.com/uploads/3/1/0/9/31098995/d4-1.pdf

Table 2
Execution time (in seconds) and memory footprint (in KiloBytes) comparison of different tools. N.A. = Not available (because the functionality is not provided by the tool).

Module/Functionality	mammut	likwid	lstopo	cpupower	turbostat	taskset
Read Energy	0.022 (2012 KB)	1.3 (11176 KB)	N.A.	N.A.	0.018 (776 KB)	N.A.
Topology	0.011 (2004 KB)	0.055 (11128 KB)	0.046 (1664 KB)	N.A.	N.A.	N.A.
CPU Frequency	0.029 (1784 KB)	0.232 (11088 KB)	N.A.	0.008 (1004 KB)	N.A.	N.A.
Process/Thread	0.020 (2652 KB)	0.056 (11076 KB)	N.A.	N.A.	N.A.	0.002 (680K)

and to properly allocate the benchmarks' threads on the target architecture. OCaml and C bindings of Mammut have been recently implemented and released as open source[6] by researchers at University of Orleans.

Finally, Mammut has been selected as a power meter in the parallel runtime framework FastFlow (version 2.1.3),[7] which targets heterogeneous multi-cores platforms. In FastFlow, Mammut provides information regarding power consumption of application parallelized by using parallel patterns.

Given the short life of the project, the increasing interest it is receiving from different research communities implies the need of such a tool. Its simplicity and flexibility allows the users to exploit Mammut in different contexts, helping the programmer in building and optimising architecture-aware software.

6. Conclusions and future directions

By using Mammut, developers can easily access and modify information provided by the hardware and OS through an intuitive object-oriented interface, without dealing with portability issues when moving their code on a different system. Moreover, Mammut seamlessly allows the management of remote systems. We are currently planning to extend Mammut with other modules for the management of caches, memory, and Graphics Processing Units (GPUs). In addition to that, we are considering the possibility to support machines running Windows operating systems.

References

[1] Miyoshi A, Lefurgy C, Van Hensbergen E, Rajamony R, Rajkumar R. Critical power slope. In: Proceedings of the 16th international conference on supercomputing. New York, New York, USA: ACM Press; 2002. p. 35.

[2] Chadha G, Mahlke S, Narayanasamy S. When less is more (LIMO):controlled parallelism forimproved efficiency. In: Proceedings of the 2012 international conference on compilers, architectures and synthesis for embedded systems. ACM Press, USA; 2012. p. 141.

[3] Cochran R, Hankendi C, Coskun AK, Reda S. Pack & Cap: adaptive DVFS and thread packing under power caps. In: Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture. USA: ACM Press, New York, New York; 2011. p. 175.

[4] Matthew Curtis-maury DSN, Blagojevic Filip, Antonopoulos Christos D. Prediction-based power-performance adaptation of multithreaded scientific codes. IEEE Trans Parallel Distrib Syst 2008.

[5] Gandhi A, Harchol-Balter M, Das R, Kephart J, Lefurgy C. Power Capping Via Forced Idleness. In: Proceedings of workshop on energy-efficient design Austin, Texas. 2009.

[6] Mishra N, Zhang H, Lafferty JD, Hoffmann H. A probabilistic graphical model-based approach for minimizing energy under performance constraints. ACM SIGARCH Comput. Architec. News 2015;43(1):267–81.

[7] Wang W, Porterfield A, Cavazos J, Bhalachandra S. Using per-loop CPU clock modulation for energy efficiency in OpenMP applications. In: 2015 44th international conference on parallel processing. IEEE; 2015. p. 629–38.

[8] De Sensi D. Predicting performance and power consumption of parallel applications. In: 2016 24th euromicro international conference on parallel, distributed, and network-based processing. 2016. p. 200–7.

[9] Totoni E, Torrellas J, Kalé LV. Using an adaptive HPC runtime system to reconfigure the cache hierarchy. In: International conference for high performance computing, networking, storage and analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014. 2014. p. 1047–58.

[10] Curtis-Maury MM, Shah AA, Blagojevic FF, Nikolopoulos DSD, De Supinski BBR, Schulz MM. Prediction models for multi-dimensional power-performance optimization on many cores. Parallel architectures and compilation techniques - conference proceedings 2008;250–9.

[11] Weaver VM, Johnson M, Kasichayanula K, Ralph J, Luszczek P, Terpstra D, Moore S. Measuring energy and power with PAPI. In: 2012 41st international conference on parallel processing workshops. 2012. p. 262–8.

[12] Li Lu, Kessler Christoph. MeterPU: A generic measurement abstraction API enabling energy-tuned skeleton backend selection. In: Proc. international workshop on reengineering for parallelism in heterogeneous parallel platforms (REPARA-2015) at ISPA-2015. 2015.

[13] Alonso P, Badia RM, Labarta J, Barreda M, Dolz MF, Mayo R, Quintana-Ort ES, Reyes R. Tools for power-energy modelling and analysis of parallel scientific applications. In: 2012 41st international conference on parallel processing. 2012. p. 420–9.

[14] Cabrera A, Almeida F, Arteaga J, Blanco V. Measuring energy consumption using EML (Energy Measurement Library). Comput Sci 2015;30(2):135–43.

[15] Liu Y, Zhu H, Lu K, Liu Y. A power provision and capping architecture for large scale systems. In: Parallel and distributed processing symposium workshops phd forum, 2012 IEEE 26th International. 2012. p. 954–63.

[16] Kimura H, Sato M, Hotta Y, Boku T, Takahashi D. Emprical study on reducing energy of parallel programs using slack reclamation by dvfs in a power-scalable high performance cluster. In: 2006 IEEE international conference on cluster computing. 2006. p. 1–10.

[17] Treibig J, Hager G, Wellein G. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In: proceedings of PSTI2010, the first international workshop on parallel software tools and tool infrastructures, San Diego CA. 2010.

[18] Mehdipour F, Javadi B, Mahanti A. FOG-Engine: towards big data analytics in the fog. In: 2016 IEEE 2nd intl conf on big data intelligence and computing and cyber science and technology congress. 2016. p. 640–6.

[19] Booch G. Object-oriented development. IEEE Trans Softw Eng 1986;SE-12(2):211–21.

[20] De Sensi D, Torquati M, Danelutto M. A reconfiguration algorithm for power-aware parallel applications. ACM Trans Archit Code Optim (TACO) 2016; 13(4):43:1–25.

[21] De Matteis T, Mencagli G. Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing. In: Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming. 2016. p. 13:1–13:12.

[22] Danelutto M, De Sensi D, Torquati M. A power-aware, self-adaptive macro data flow framework. Parallel Process Lett 2017;27(01):1740004.

[23] De Matteis T, Mencagli G. Proactive elasticity and energy awareness in data stream processing. J Syst Softw 2017;127(C):302–19.

[24] Janjic V, et al. D4.1: software for homogeneous adaptivity for initial patterns. RePhrase - refactoring parallel heterogeneous resource-aware applications (Project Number: 644235). 2016;EU H2020-ICT-09-2014-1(D4.1).

[25] Catena M, Tonellotto N. Energy-Efficient query processing in web search engines. IEEE Trans Knowl Data Eng 2017;29(7):1412–25.

[26] Danelutto M, De Matteis T, De Sensi D, Mencagli G, Torquati M. P3ARSEC: towards parallel patterns benchmarking. In: Proceedings of the symposium on applied computing. USA: ACM, New York, NY; 2017. p. 1582–9.

[5] https://github.com/ParaGroup/p3arsec
[6] https://github.com/mathiasbourgoin/ocaml_mammut
[7] http://mc-fastflow.sourceforge.net