A compositional model to characterize software and hardware from their resource usage

Davide Morelli and Antonio Cisternino

Computer Science Department, University of Pisa
Largo B. Pontecorvo 3, Italy
(morelli|cisterni)@di.unipi.it

Abstract

Since the introduction of laptops and mobile devices, there has been a strong research focus towards the energy efficiency of hardware. Many papers, both from academia and industrial research labs, focus on methods and ideas to lower power consumption in order to lengthen the battery life of portable device components. Much less effort has been spent on defining the responsibility of software in the overall computational system's energy consumption.

Some attempts have been made to describe the energy behaviour of software, but none of them abstract from the physical machine where the measurements were taken. In our opinion this is a strong drawback because results can not be generalized. We propose a measuring method and a set of algebraic tools that can be applied to resource usage measurements. These tools are expressive and show insights on how the hardware consumes energy (or other resources), but are equally able to describe how efficiently the software exploits hardware characteristics. The method is based on the idea of decomposing arbitrary programs into linear combinations of benchmarks of a test-bed without the need to analyse a program's source code by employing a black box approach, measuring only its resource usage.

1998 ACM Subject Classification D.2.8 Metrics, D.4.8 Performance, B.8.2 Performance Analysis and Design Aids

Keywords and phrases Performance, Metrics, Energy consumption

Digital Object Identifier 10.4230/OASIcs.ICCSW.2012.95

1 Introduction

Energy consumption is a global concern; as the Environmental Protection Agency of the U.S. stated in a report [8] dated 2007, the energy consumed by data centres and servers alone can account for 1.5% of the global energy use, and is doubling every five years.

The adoption rate of portable devices raised the attention towards energy efficiency of hardware components (in order to lengthen battery life) and network protocols. Much less effort has been spent on defining the responsibility of software in the overall computational system's energy consumption, none of them abstract from the physical machine where the measures are taken. In our opinion this is a strong drawback because results can not be generalized.

We propose a measuring method and a set of algebraic tools that can be applied to resource usage measurements (energy consumption and completion time being just two instances of resource usage). These tools are expressive and show insights on how the hardware consumes energy (or other resources). They are also able to describe how efficiently the software exploits the hardware characteristics.

Typically, measurement techniques proposed in the literature aim to break down the energy consumption to atomic components, associating an average cost to every instruction [16, 15,

2012 Imperial College Computing Student Workshop (ICCSW'12).
Editor: Andrew V. Jones; pp. 95–101

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

9, 2, 17]. We believe that a different approach should be used, as modern computational systems tend towards complex systems. Hardware started developing parallelism when the CPU frequency reached its cap, and software systems are becoming increasingly complex. The result of this process is that the resource usage of a single instruction is hardly predictable and depends on the execution context. For example, the time required to load a location depends on both the program's memory access pattern, the hardware characteristics, and the memory used by the other processes running on the same computational system, leading to almost *non deterministic* results. For this reasons we have chosen to follow a black box approach, measuring the resource usage of the software running on a computational system as a whole, following the approach proposed in [11], instead of trying to profile the software [9], simulating the execution of algorithms on modified virtual machines or power level performance simulators [2, 17]. This is because these approaches either require source code access or are only feasible for simple architectures, where a cycle accurate simulator is possible and for relatively small programs. A black box approach measures the software as a whole without trying to break down the energy consumption of single instructions. This approach is the most simple to implement, does not need modifications to the operative system and works with a simple ammeter with the simplest possible approach: the current is measured in AC from the wall outlet [14, 17, 4].

Metrics for software similarity are very interesting because they allow us to predict the behaviour of programs using measurements of similar programs, and allow their characterization. Yamamoto proposes a metric of similarity based on source code analysis in [18]. For the scope of our research, we are interested in methods that do not require access to the source code because we want to be able to characterize software as a black box.

In a 1992 paper [12] we found an approach that was particularly inspiring for our work: a model was proposed to characterize both hardware and software. The overall completion time of a program was modelled as a linear decomposition of abstract operations. Our model is very similar to their with respect to the linear composition model. Nevertheless, there are many important differences:

- they use abstract operations as computational patterns (e.g. *add*, *store*, *divide*, etc.); we allow more articulate entities, long combinations of instructions
- the program analysis is performed by means of static analysis and instrumentation of the source code; we don't require the source code to analyse the program
- their model focuses on completion time solely; our model is capable of describing the usage of every measurable resource, we are most interested in completion time and energy consumption, but could be applied to any other metric (i.e. memory usage, CPU time, etc.)

[17] also proposed an energy characterization model for both hardware and software.

In [1], the use of performance counters leverages the characterization of software, a technique that is becoming a de facto standard [5, 10, 7, 6]. Benchmarks are analyzed, PCA is used to reduce the solution space and clustering techniques are used to identify families of programs and to find a representative workload for a certain task.

Two other key concepts are the idea that the environment where the program is run must be taken into account [3] and the need to find a model capable of offering results resilient with change of hardware: Sherwood [13] characterized software with a model consistent with the change of architecture; he proposed a high level approach (not at instruction level), but he did not focus on energy consumption.

2 A compositional model

Programs are composed of instructions that once executed affect the resource consumption of the system. There are many different kinds of computational resources a program can consume (CPU time, memory, network, etc.). We define *computational pattern* a sequence of instructions that expose a peculiar resource usage, that is subject to change as we change the computational system where the pattern is executed on, e.g. on a processor family FPU operation may consume more energy to complete with respect to a different class of processors.

In our model we assume that actual programs can be seen as composed of computational patterns, i.e. a matrix multiplication algorithm will read data from memory (showing a peculiar memory read pattern with cache hit and miss), perform FPU then write the result back to memory.

A computational pattern will have a different resource usage on each computational system, e.g. the same memory pattern of data read could rise to a much lower number of cache miss if a processor is capable of predicting the pattern and prefetching data. The composition of a program from the point of view of the computational patterns does not change when the program is run on a different computational system, but the resource usage behaviour will change because the computational pattern the program is composed of will have a different resource usage profile.

Therefore, we chose a set of synthetic benchmarks as our *test-bed*, where every benchmark is intended to capture a particular computational pattern that we expect to find in different quantities in every program we intend to analyse, with zero being a legitimate quantity.

3 Linear algebra model

We define the *measurement matrix* a computational system S as a $\mathbb{R}^{m\times n}$ matrix where n is the number of benchmarks in our test-bed and m is the number of attributes (resource usage) we measure for each program. Each one of these matrices holds the knowledge we have about a particular computational system. The i^{th} column of $\mathbf{M}$ shows the resource usage of the i^{th} benchmark of our test-bed. The j^{th} cell of that column holds the measurement of the resource usage of the j^{th} attribute (e.g. CPU time, cache hit, etc.).

When we want to decompose a program p using the benchmarks of our test-bed as the building block we measure the resources usage of p running on S and we build a vector μ_p with those measures. The j^{th} element of μ_p holds the measurement of the resource usage of the j^{th} attribute; μ_p is like a column of $\mathbf{M}$ but is composed of measurements of a program that is not part of the test-bed. Now consider the following linear system:

$$\mathbf{M}\cdot\mathbf{s}_p = \mu_p \tag{1}$$

We call $\mathbf{s}_p$ the *split-up* of p, it holds a decomposition of p using the benchmarks of our test-bed as building blocks.

Standard vector algebra can be used to analyse and interpret measures, splitups and programs. We can analyse vectors using vector norm:

$$\|v\| = \sqrt{\sum_{i=1}^{n}(v_i)^2} \tag{2}$$

and vector similarity:

$$cos(\theta) = \frac{v_1 \cdot v_2}{\|v_1\|\|v_2\|} \tag{3}$$

3.1 Measurements space

The columns of M can be seen as vectors in an m dimensional vector space that we call the *measurements space*. The position of the i^{th} vector shows the resource usage of the i^{th} benchmark of the test-bed. More generally speaking, the μ_p vector shows the resource usage of the program p in a particular system S.

The norm can be used to get an insight on the overall resource usage of a benchmark, i.e. more resource demanding benchmarks will have a higher norm than less resource demanding ones.

Vector similarity will tell us how similar two programs are: if the angle between the vectors is small it means that they may use more or less resources in absolute terms (have different norms), but their resource usage behaviour is similar.

3.2 Splitup space

Splitups can be seen as vectors in a n dimensional vector space that we call the *splitup space*. The position of a vector in this space shows the composition of the program using the benchmarks as building blocks.

When comparing the splitup vectors of two programs we can say that the one with a higher norm is the more resources demanding. If the vector similarity is very close to 1 but the norms are different we are probably looking at the same program running with different input sizes, e.g. if p_1 and p_2 are the same sorting algorithm with p_1 running on half the array size of p_2 we'll probably see $\|\mathbf{s}_{p_2}\| = 2\|\mathbf{s}_{p_1}\|$ and $\frac{\mathbf{s}_{p_1} \cdot \mathbf{s}_{p_2}}{\|\mathbf{s}_{p_1}\|\|\mathbf{s}_{p_2}\|} = 1$

When a program is run on different input sizes the balance of the computational patterns used may vary, e.g. the cache hit ratio could grow logarithmically while the FPU usage may grow linearly. Analysing how the splitup of a program changes with the input size is highly informative of the program structure. When the splitup does not change with the input size we call the program *uniform*; and if it changes, we call it *non uniform*.

If the test-bed is well formed the splitup of a program has to be the same when we run it on different computational systems. If it is different it means that this program is capturing a resource usage behaviour not captured by any benchmark in the test-bed, in other words this program contains an unknown computational pattern, therefore it should be added to our test-bed.

3.3 Benckmark space

$\mathbf{M}_S$ is the $\mathbf{M}$ of a system S. Its rows can be seen as vectors in a n^{th} dimensional space that we call the *benchmark space*. We can see how resource usage changes when we change computational system from S_a to S_b analysing the position of the i^{th} row of $\mathbf{M}_{S_a}$ (where i is the index of the resource we are interested in) against the position of the i^{th} row of $\mathbf{M}_{S_b}$ in the benchmark space.

E.g. when the norm of the energy consumption vector of S_1 is higher than the norm of the energy consumption vector of S_2 it means that S_1 is (generally speaking) less energy efficient than S_2. If the angle between their completion time vectors is small it means that S_1 and S_2 have a similar architecture and probably one is just more efficient than the other (i.e.

a newer machine). If the angle is large it means that the systems have a different architecture that makes some of the benchmarks in the test-bed more efficient than others; the direction of the difference between the vectors tells us how S_1 is different from S_2.

4 Real data

Measuring resource attribute will usually involve error, i.e. the accuracy of the measuring tool, sampling frequency, etc. Equation 1 could be not solvable and has to be rewritten in order to minimise a norm of the error vector:

$$\epsilon = |\mathbf{M} \cdot \mathbf{s}_p - \mu_p| \tag{4}$$

If we use a Manhattan norm instead of an Euclidean norm, this is a linear programming problem that can be solved using the simplex algorithm.

We want all the elements of $\mathbf{s}_p$ to be non negative numbers, because each of them expresses an estimation of the number of iterations of the respective benchmark, as present in p. The benchmark space is therefore not a vector space but a convex cone. This limitation does not change the approach needed to find the splitup, since we just need to add a few conditions to the simplex.

Being the benchmark space a convex cone, the number of vectors (the benchmark in the test-bed) that form a basis is not generally known, but the process of selection of the benchmarks in the test-bed can be incremental and automatic: if the splitup of a program falls within the convex cone it means that it can be expressed as linear decomposition of known computational patterns, if it falls outside the cone it means that it should be added to the test-bed, widening the range of programs that can be expressed algebraically.

We can choose the level of detail we want to get with the decomposition of programs. I.e., we might want to have a computational pattern for every major memory read pattern or just a general one. In the former case we would be able to discriminate how the program uses memory, but we would need a lot of experimental data to solve the system. In the latter, we would need few experimental data but might only see a raw estimate of the program's behaviour. The number of rows of the measurement matrix needs to be larger than the number of columns, which means that we need to measure at least as many resources as the number of the benchmarks in the test-bed. This could be difficult if we want to have a large test-bed, in which case we could create a new measurement matrix with more rows just merging measurement matrices of multiple systems.

5 Experimental data

As an example we present data of a preliminary test: we measured the completion time and the energy consumption of a small set of programs running on a desktop computer equipped with a CoreDuo processor with 2MB L2 cache and 1 GB RAM (from now on referred to as S). We prepared two synthetic benchmarks: *cpu* is a simple *add* assembler instruction executed 10^6 times; *mem* is a program that sums a fixed number of random locations from a large array. We used *cpu* and *mem* as our test-bed and measured mergesort (from now on referred to as p) sorting arrays of different sizes (1M, 2M, 4M, 8M, 16M, 32M).

	cpu	mem	p(1M)	p(2M)	p(4M)	p(8M)	p(16M)	p(32M)
time	2.14 s	7.26 s	0.22 s	0.33 s	0.67 s	1.39 s	2.85 s	5.79 s
energy	81.46 J	304.00 J	8.60 J	13.20 J	27.49	58.29 J	121.79 J	254.82 J

$\mathbf{M}_S$ is composed of the first two columns of the above table and the measurement vectors for mergesort at various input sizes are:

$$\mu_{p(1M)} = \begin{pmatrix} 0.22\,\mathrm{s} \\ 8.60\,\mathrm{J} \end{pmatrix} \quad \mu_{p(2M)} = \begin{pmatrix} 0.33\,\mathrm{s} \\ 13.20\,\mathrm{J} \end{pmatrix} \quad \mu_{p(4M)} = \begin{pmatrix} 0.67\,\mathrm{s} \\ 27.49\,\mathrm{J} \end{pmatrix}$$
$$\mu_{p(8M)} = \begin{pmatrix} 1.39\,\mathrm{s} \\ 58.29\,\mathrm{J} \end{pmatrix} \quad \mu_{p(16M)} = \begin{pmatrix} 2.85\,\mathrm{s} \\ 121.79\,\mathrm{J} \end{pmatrix} \quad \mu_{p(32M)} = \begin{pmatrix} 5.79\,\mathrm{s} \\ 254.82\,\mathrm{J} \end{pmatrix}$$

The resulting splitup vectors (calculated minimizing formula 4 using the simplex algorithm) are:

$$\mathbf{s}_{p(1M)} = \begin{pmatrix} 0.075118 \\ 0.008161 \end{pmatrix} \quad \mathbf{s}_{p(2M)} = \begin{pmatrix} 0.075862 \\ 0.023093 \end{pmatrix} \quad \mathbf{s}_{p(4M)} = \begin{pmatrix} 0.069347 \\ 0.071845 \end{pmatrix}$$
$$\mathbf{s}_{p(10M)} = \begin{pmatrix} 0 \\ 0.248125 \end{pmatrix} \quad \mathbf{s}_{p(20M)} = \begin{pmatrix} 0 \\ 0.528191 \end{pmatrix} \quad \mathbf{s}_{p(30M)} = \begin{pmatrix} 0 \\ 0.780954 \end{pmatrix}$$

The splitup vectors show how quickly mergesort gets dominated by memory usage as the input size grows. This is expected because as the array grows it will not fit into cache and a lot of cache miss will occur, therefore most of the time and energy will be spent accessing memory.

6 Conclusion

We have presented a method to decompose arbitrary programs into linear combinations of benchmarks of a test-bed by employing a black box approach, measuring only its resource usage, without the need to analyse a program's source code. Valid metrics of resource usage are both performance counters and energy consumption (or completion time). Performance counters can therefore be used to build a model capable of predicting the energy consumption (or completion time) of the same program on a different computational system. The same method also gives us useful information about what the differences between computational systems are, thereby showing which computational patterns consume more resources. We intend to apply this method to heterogeneous computing (CPU/GPU), virtual machines and cloud systems to provide realtime analysis and forecasting of energy consumption (as well as completion time) of software, without prior knowledge of its source code.

References

1 Jan Lodewijk Bonebakker. Finding representative workloads for computer system design. Technical report, Mountain View, CA, USA, 2007.

2 David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 83–94, New York, NY, USA, 2000. ACM.

3 Fay Chang, Keith Farkas, and Parthasarathy Ranganathan. Energy-driven statistical sampling: Detecting software hotspots. In Babak Falsafi and T. Vijaykumar, editors, *Power-Aware Computer Systems*, volume 2325 of *Lecture Notes in Computer Science*, pages 105–108. Springer Berlin / Heidelberg, 2003.

4 A. Cisternino, P. Ferragina, D. Morelli, and M. Coppola. Information processing at work: On energy-aware algorithm design. In *Green Computing Conference, 2010 International*, pages 407 –415, aug. 2010.

5 Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 157–166, New York, NY, USA, 2006. ACM.

6 Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 220–, Washington, DC, USA, 2003. IEEE Computer Society.

7 Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Workload design: Selecting representative program-input pairs. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:83, 2002.

8 R. Brown et al. Report to congress on server and data center energy efficiency: Public law 109-431, 2008.

9 Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, WMCSA '99, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.

10 Aashish Shreedhar Phansalkar. *Measuring program similarity for efficient benchmarking and performance analysis of computer systems*. PhD thesis, Austin, TX, USA, 2007. AAI3285977.

11 Suzanne Marion Rivoire. *Models and metrics for energy-efficient computer systems*. PhD thesis, Stanford, CA, USA, 2008. AAI3313649.

12 Rafael H. Saavedra and Alan J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst.*, 14:344–384, November 1996.

13 Tomothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36:45–57, October 2002.

14 Amit Sinha and Anantha P. Chandrakasan. Jouletrack - a web based tool for software energy profiling. In *In Design Automation Conference*, pages 220–225, 2001.

15 Stefan Steinke, Markus Knauer, Lars Wehmeyer, and Peter Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *in Proc. Int. Wkshp Power and Timing Modeling, Optimization and Simulation (PATMOS*, 2001.

16 Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: a first step towards software power minimization. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, ICCAD '94, pages 384–390, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

17 N. Vijaykrishnan, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. pages 95–106, 2000.

18 Tetsuo Yamamoto, Makoto Matsushita, Toshihiro Kamiya, and Katsuro Inoue. Measuring similarity of large software systems based on source code correspondence. In Frank Bomarius and Seija Komi-Sirviö, editors, *Product Focused Software Process Improvement*, volume 3547 of *Lecture Notes in Computer Science*, pages 179–208. Springer Berlin / Heidelberg, 2005.