An Incremental Methodology for Energy Measurement and Modeling

Abdelhafid Mazouz*
Intel Corporation
Champaign, IL, USA
first.last@gmail.com

David C. Wong
Intel Corporation
Austin, TX, USA
first.c.last.@intel.com

David Kuck
Intel Corporation
Austin, TX, USA
first.last@intel.com

William Jalby
UVSQ/ECR
Versailles, France
first.last@uvsq.fr

ABSTRACT

This paper presents an empirical approach to measuring and modeling the energy consumption of multicore processors. The modeling approach allows us to find a breakdown of the energy consumption among a set of key hardware components, also called HW nodes. We explicitly model the front-end and the back-end in terms of the number of instructions executed. We also model the L1, L2 and L3 caches. Furthermore, we explicitly model the static and dynamic energy consumed by the the uncore and core components. From a software perspective, our methodology allows us to correlate energy to the executed code, which helps find opportunities for code optimization and tuning.

We use binary analysis and hardware counters for performance characterization. Although, we use the on-chip counters (RAPL) for energy measurement, our methodology does not rely on a specific method for energy measurement. Thus, it is portable and easy to deploy in various computing environments. We validate our energy model using two Intel processors with a set of HPC codelets, where data sizes are varied to come from the L1, L2 and L3 caches and show 3% average modeling error. We present a comprehensive analysis and show energy consumption differences between kernels and relate those differences to the algorithms that are implemented. Finally, we discuss how vectorization leads to energy savings compared to non-vectorized codes.

CCS Concepts

•Computing methodologies → Modeling methodologies; •Hardware → Power and energy; •Computer systems organization → *Multicore architectures;*

*Abdelhafid Mazouz contributed to this work as an Intel employee. He is now at Bull/Atos.

ICPE'17, April 22-26, 2017, L'Aquila, Italy
© 2017 ACM. ISBN 978-1-4503-4404-3/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3030207.3030224

Keywords

Energy modeling; Performance evaluation, RAPL

1. INTRODUCTION

Power or energy models are widely used in the context of dynamic power management and DVFS controllers [1, 3, 22]. They can serve as a tool to dynamically find hardware (HW) parameters that are best suited for a given workload in a computing system. Accurate power and energy modeling of HW are also important components for software (SW) development tools and HW/SW codesign tools [11, 12]. Generally, one needs a model in which power or energy is related to the amount of each HW resource used in a given computation. We define HW resources (called HW nodes) as HW components that can enhance the overall performance. This allows expression of system or subsystem power or energy as the sum of individual HW node contributions, so for each computation the contributions of each node can be understood. This paper gives a general procedure for generating such models by iteratively refining high-level measurements down to lower-level HW details, and in the limit to individual operations and instructions.

Our methodology is portable, as we rely on HW counters for performance and energy measurement, which are available on most modern general purpose processors. In this paper, we apply our methodology to estimate energy consumption for multicore processors. We use the *Running Average Power Limit* [5, 10] (RAPL) interfaces for energy measurement and estimation. As we focus on Intel microprocessors, such HW interfaces are provided to estimate the core and uncore energy. Although errors are observed in such HW estimates [9, 16], it allows us to deploy and apply our methodology in various computing environments without the need for physical HW probes. If physical HW probes or an accurate high-level simulator were available for such, our methodology could be applied directly to those physical energy measurements. The general procedure incrementally produces as much detail as can be isolated by microbenchmark measurements and HW counters. The two key parameters in our model are static power and dynamic energy consumption. We give lumped static power estimates for core and uncore, as well as dynamic energy contribution down to low-level nodes.

The paper discusses our overall methodology and validation of the model, as well as some observations about the results. These include comparisons of two HSW microprocessors, comparison of numerical codelets, and relative energy consumption of scalar, SSE and AVX vector versions of each codelet. This allows one to understand problems with a HW design, e.g. which nodes are most important to improve in a new system for a given workload. Also, the results can be used in SW development tools to understand what changes to make to each codelet to improve its overall energy or performance/energy ratio for an existing system.

The main contributions of the paper are:

1. A way of separating static and dynamic power at the level of the core and uncore:
   (a) Using clock modulation [10] to estimate static power of the core. Unlike DVFS, clock modulation allows us to vary frequency while voltage is kept constant.
   (b) Using multicore runs to estimate uncore static power. This allows us to vary the number of active vs non-active cores. When all cores are non active (deep `C-States` [10]), their voltage is set to zero, exposing the uncore power consumption.
2. A general iterative model that starts with high-level energy and performance measurements and derives lower level node energy information by a series of micro-benchmarking steps:
   (a) Dynamic energy breakdown among all key HW nodes, including those impacting performance.
   (b) Multi-step methodology allowing isolation of the energy contribution of all modeled HW nodes.
3. Energy comparison:
   (a) Low level analysis of energy consumption by different codelets.
   (b) Instruction set comparison.
   (c) Architectural comparison.

This paper is organized as follows. Section 2 presents our experimental setup and measurement methodology. Section 3 motivates our energy modeling work. Section 4 describes the different steps of our energy modeling methodology. Section 5 presents our experimental results. Section 6 presents some background on energy modeling. Finally, Section 7 gives our conclusions/perspective.

2. EXPERIMENTAL SETUP

2.1 Hardware platforms

For our evaluation, we use two Intel test platforms, one is a server machine and the other is a workstation/client. The two machines exhibit differences in terms of the maximal supported clock frequency and the size of the L3 cache.

- **Haswell** client machine (`HSW-CL`). The machine consists of a single `Xeon` E3-1270 v3 processor. The processor has 4 cores running at 3.5 GHz maximal non-turbo frequency. The four cores share an L3 cache of 8 MB. In each core, the L1 instruction cache size is 32 KB, the L1 data cache size is 32KB and L2 cache size is 256KB. The process technology is 22 nm.
- **Haswell** server machine (`HSW-SE`). The machine consists of a single `Xeon` E5-2630 v3 processor. The processor has 8 cores running at 2.4 GHz maximal non-turbo frequency. The eight cores share an L3 cache of 20 MB. In each core, the L1 instruction cache size is 32 KB, the L1 data cache size is 32KB and L2 cache size is 256KB. The process technology is 22 nm.

2.2 Measurement methodology

To ensure the predictability and the reproducibility of our results [17], we turned off Hyper-Threading (HT) and Turbo-boost (TB) in all our experiments. We also fixed the CPU frequency in each processor to the maximal non-Turbo frequency using the Linux *user space* governor. Consequently, we prevent the OS/HW from changing the frequency at runtime. Because prefetch can help or hinder performance across codelets, for simplicity, we only run experiments with prefetch off. While turning on Hyper-Threading and HW prefetch does not limit the applicability of our methodology, turning on Turbo-boost on can be a challenge for any modeling methodology as voltage and frequency are dynamically varied by the HW depending on various runtime parameters without being fully exposed to the OS and applications. Though our modeling approach includes multicore training runs, this paper presents only unicore validation results and leaves multicore validation for future work.

To build the energy model, we created a set of dedicated assembly micro-benchmarks to stress various HW components. As for model validation, We run a set of kernels that we call codelets. They are extracted from numerical recipes (NR) [19, 20], where many families of algorithms are represented: Linear Algebraic Equations, Eigensystems, Fast Fourier Transform and Partial Differential Equations. All the codelets were carefully selected and they cover a wide range of performance and code structure characteristics [19]: single, double and mixed precision data, non-vectorized, partially vectorized and fully vectorized, 1D and 2D loops, 1D and 2D arrays, unit stride and non-unit stride memory accesses. There are also scalar, vector-vector, matrix-vector and matrix-matrix types of computations. For each codelet, we can control the data sizes, so we picked data sizes that fit the L1, L2 and L3 caches.

The codelets were compiled using the `Intel Icc` compiler version 15.0.0. In order to study the relationship between vectorized code and energy efficiency, we created three versions of each codelet whenever possible.

- **Scalar SSE** (`SC`). Each codelet is compiled using the `-O3 -xSSE4.2 -no-vec` flag. The compiler generates pure scalar code using SSE4.2 instructions (the `-no-vec` is necessary for codelets where the compiler can generate vector code by default).
- **Vector SSE** (`SSE`). Each codelet is compiled using the `-O3 -xSSE4.2` flag. The compiler generates vectorized code using the `SSE4.2` instruction set.
- **Vector AVX** (`AVX2`) Each codelet is compiled using the `-O3 -xcore-avx2` flag. The compiler generates vectorized code using the `AVX2` instruction set.

There are 26 NR codelets, 19 are vectorizable and 7 are purely scalar. As we can use the `-no-vec` flag, we can generate a total of 26 `SC`,19 `SSE`, 19 `AVX2` codelets. So we used a total of 64 codelets in our experiments.

For energy measurements, we use the RAPL [10] interfaces present in each of our processors. They are a set of non-architectural HW counters[1] used for power management like power limiting. Depending on the platform type (client or server), these interfaces are organized into multiple domains: the package (PKG), the cores (PP0), the graphics (PP1) and the memory domain (DRAM). In our paper, we use only the energy measurement capability of RAPL and focus on the PKG domain. The PKG domain estimates energy for all the chip including all cores and uncore components. While we rely on RAPL, our methodology is not bound to such interfaces. If physical HW probes for energy measurement were available, then our methodology could be applied directly to those physical energy measurements.

We use multiple metrics to express performance, power and energy. We express performance rate as computational capacity C[operations/second] (we also use flop in place of operation), power as W[Watts], and energy as E[Joules], E[Watt-hour]. As we deal with fine grained codelets, we also express E in milli-Joules (mJ) or nano-Joules (nJ). The quality of a computation (a codelet running on an architecture) can be expressed in many ways and we use the quality metrics: C, C/W and C/E [4, 7]. C/W expresses a traditional HW metric $C/W[operations/energy] = O/E$, and C/E is a traditional system-level metric expressed as C/E[perf/operating cost]. While traditional metrics are sometimes the reciprocals of these, we prefer graphs in which higher is better.

3. MOTIVATION

Unlike many research efforts [1, 3, 22], we focus on energy modeling instead of power because energy exhibits much more variation than power. We focus on energy because its dynamic range is much greater than the power range (numerically), as shown in Figure 1. Figure 1 illustrates why we consider energy instead of power. It also shows similarity and monotonic order of total energy E and power W measurements of all NR codelets regardless of the instruction set. While this data is related to measurements on `HSW-CL`, the same conclusions result from measurements on `HSW-SE`.

Figure 1 shows that while E has an approximate range of 45X, power has only an approximate range of 1.4X between the observed minimal and maximal power consumption values. Although, these codelets were run with data sizes chosen to come from the L3 cache, one may think that it is unfair to compare energy values of different codes as these codelets implement various algorithms with various operation counts. To tackle this issue, Figure 2 compares normalized energy values against performance numbers for all NR codelets. We make clear distinction between codelets that belong to each instruction set category: `SC`, `SSE` and `AVX`. On one side, the y-axis reports the E/O metric which represents the energy necessary to perform one floating point operation. In fact, $E/O = W/C$, so energy per operation is equivalent to the ratio of power by performance. The x-axis shows performance C and it is expressed in terms of GFlops.

These normalized metrics allow comparing multiple codelets. Figure 2 also shows that there is wide range between $E/O = W/C$ values on one hand, and C values on the other one. In fact, W/C exhibits a 60 X range and C exhibits an 80 X range. Furthermore, the hyperbolas fitted to each instruction set show less than 1 watt separating the `SC`, `SSE` and `AVX2` instruction sets (see equation numerators in Figure 2), where the `SSE` curve is slightly above `SC`, and `AVX2` is slightly above `SSE`. This suggests that an energy model more easily exhibits codelet differences than a power model.

[1]Also referred to as MSR.

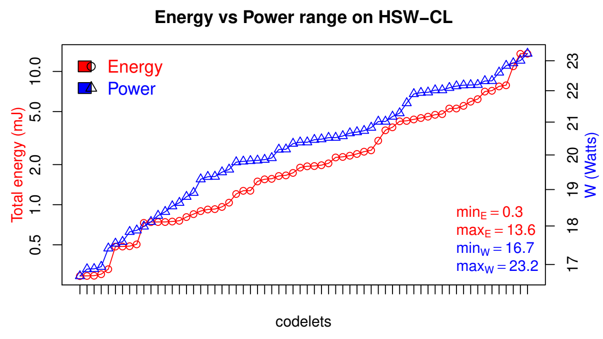

Figure 1: Energy and power range per codelet on `HSW-CL` of 64 NR codelets in rank order

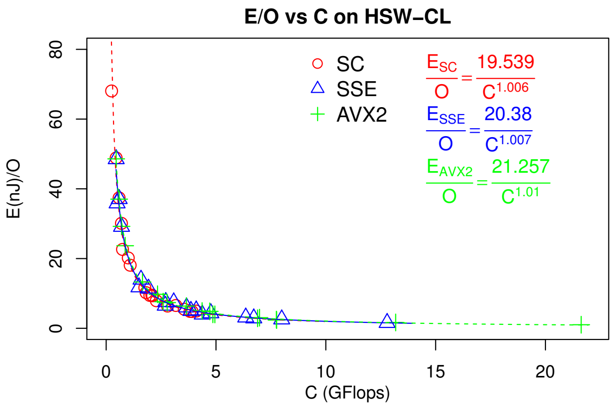

Figure 2: Comparison of normalized energy to performance per codelet on `HSW-CL`

4. ENERGY CONSUMPTION ESTIMATION

The idea behind our modeling approach is to identify key HW components that impact performance and lead to observable energy consumption. Once these HW nodes are identified, it is possible to associate energy weights proportionally to their usage. The profiling information about HW node usage can be obtained using static and dynamic analysis. Static analysis is performed on the extracted assembly code using the MAQAO [15] framework. It allows us to obtain metrics about the mix of instructions that are executed. Dynamic analysis relies on hardware performance counters and allows us to collect the memory traffic while executing a code. Memory traffic can be defined as the cache line movement between the L1 and L2 caches on one hand, and between the L2 and L3 caches on the other hand.

4.1 Modeled HW nodes

We consider that a CPU consists of a set of cores, each with private L1 and L2 caches and an uncore component.

The uncore is shared between all the cores and it includes the L3 cache. Moreover, each core consists of a front-end for instruction decoding and a back-end for instruction execution. As we target HPC codes composed of multiple loop-nests, we can classify the modeled HW nodes into four main categories: floating-point execution units (FPU), integer execution units (INT), front-end (FE) and memory (Caches), all modeling the dynamic energy of a CPU.

- **FE** nodes. This class models the front-end stage. It consists of a single node that models the energy consumption when issuing micro-ops or instructions from the front-end to the back-end.
- **INT** node. This is a single node that models the energy consumption when executing integer instructions, e.g. used in the control flow of loops.
- **FPU** nodes. This class represents mostly floating-point operations that are executed in the back-end (dispatch). The modeled nodes/operations are `ADD`, `MUL`, `DIV` and `FMA`. For all of these, a distinction is made in terms of the number of bytes processed by each instruction, e.g. we model ADD32, ADD64, ADD128 and ADD256. This class has a total of 17 nodes.
- **Cache** nodes. This class models memory traffic between the core and the hierarchy of memory caches. First, we model `L1` nodes representing the energy consumption of executing memory loads (`LD`) and stores `ST` in the L1 cache. There are four `LD` and four `ST` nodes; each captures the energy consumed when varying the data width of operations (32, 64, 128 or 256 bits). Next, are the `L2` and `L3` node. The former models the energy consumption for the data movement between the L1 and L2 caches. The latter models the energy consumption of cache line movement between the L2 and L3 caches. Both account for read and write traffic.

Besides the nodes that model the dynamic energy, we also explicitly model the static energy that is consumed in the uncore and in the core. The static energy of the core is a single quantity that accounts for the OoO engine, FE logic, execution units, plus the L1 and L2 caches. The static energy of the uncore on the other hand covers the L3, IMC and other components.

4.2 Modeling methodology

To precisely isolate the energy contribution of each modeled HW node, we rely on an incremental methodology. As depicted in Figure 3, we distinguish two main steps: 1) static energy (E^{static}) estimation and 2) dynamic energy ($E^{dynamic}$) estimation. Consequently, we express the total energy using Equation 1.

$$E^{total}_{CPU} = E^{static}_{CPU} + E^{dynamic}_{CPU} \tag{1}$$

For the static energy, we first model the static energy of the uncore E^{static}_{uncore}, then we model the static energy of each active core E^{static}_{core}. For a given codelet, dynamic energy consumption represents the aggregate energy of all the HW nodes that are stressed while running that codelet. As discussed in Section 4.1, we consider FE, INT, FPU and cache nodes to model the dynamic energy of a CPU.

All of the steps we follow rely on a set of dedicated micro-benchmarks written in assembly code to stress the various HW components that we want to model. Using this methodology, we believe that we can find a breakdown of the energy consumption of any workload, regardless of the measurement method . For example, we use the RAPL interface to measure total energy consumption, but energy measurements using physical probes or simulation numbers could be used.

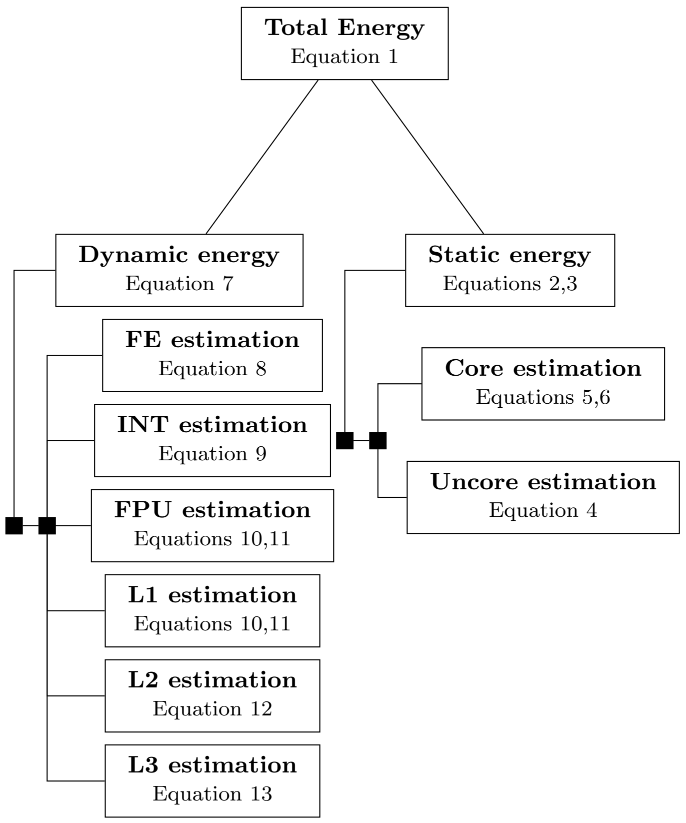

Figure 3: Energy modeling diagram

4.2.1 Static energy estimation

Instead of estimating directly the static energy, which is workload dependent, we estimate the static power W^{static}_{CPU}. W^{static}_{CPU} of a multicore processor is modeled using Equation 2. The static energy of a workload can be computed as the product of the static power W^{static}_{CPU} and the measured time duration T of the workload (Equation 3)

$$W^{static}_{CPU} = ActiveCores \times W^{static}_{core} + W^{static}_{uncore} \tag{2}$$

$$E^{static}_{CPU} = W^{static}_{CPU} \times T \tag{3}$$

While we rely on varying the number of active cores to estimate the static power of the uncore, we rely on the duty cycle modulation [10] feature to estimate the static power of the core. Duty cycle modulation also called `T`-states[2] is a mechanism for active power management available on Intel processors starting from `Pentium 4`. It is a throttling mechanism used to modulate processor core frequency, mainly for

[2]T-states refer also to thermal states.

thermal management techniques. Duty cycle modulation allows processor power and temperature reduction by lowering core activity to some predefined level.

Static power of the uncore. To estimate W^{static}_{uncore}, we run a pure floating-point code (no memory accesses). By excluding memory accesses, we avoid having dynamic power consumed by L1, L2 or L3 accesses. Consequently, the total power consumed by the chip consists of W^{static}_{uncore}, W^{static}_{core} and $W^{dynamic}_{core}$. We can then measure the total power consumption of the chip while varying the number of active cores that run the same FP code simultaneously. We model the static power of the uncore using Equation 4. W^{static}_{uncore} is estimated as the intercept of the simple linear regression when the number of active cores equals 0.

$$W^{total}_{CPU} = ActiveCores \times (W^{dynamic}_{core} + W^{static}_{core}) + W^{static}_{uncore} \tag{4}$$

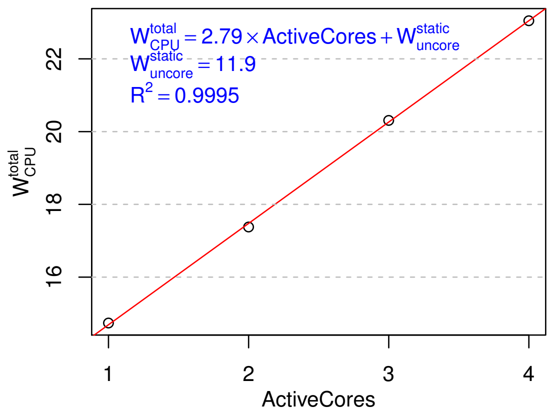

Figure 4: Estimation of uncore static power on HSW-CL

Figure 4 reports the static power of the uncore on the HSW-CL machine when the number of active cores is varied from 1 to 4. We assume that when C-States [10] is enabled and no workloads assigned to the cores (not active), they enter into deep sleep states where voltage is set to 0. We then consider that since all the cores are turned off, the measured power consumption is due to the uncore component. So we define that power as the uncore static power.

Static power of the core. Once we estimate the power consumption of the uncore, the second step is to model the static power of the core. We consider that the static power of the core includes all the core logic plus the static power of the L1 and L2 caches. We make two main assumptions. First, to avoid any measurement noise related to memory operations, we use pure FP micro-benchmarks. Second, we run a copy of our designed FP micro-benchmark on all the available cores on the machine simultaneously, to ensure that no core enters deep sleep states that may lead to wrong conclusions about the static power estimation.

If we run codes that do not have memory requests, then the total power of a CPU can be modeled using Equation 5. For a fixed voltage V, $\beta(V) = W^{static}_{CPU}$ (Equation 5). While $\beta(V)$ expresses the static power of all active cores plus the static power of the uncore as a function of V, $\alpha(V) \times F$ expresses the dynamic power of the core in terms of V and F. This means that in order to estimate $W^{static}_{core}(V)$, we need to fix V and vary F such that $W^{dynamic}_{core}(F, V) = 0$. In the context of our test machines, changing CPU frequency[3] leads to V and F being varied simultaneously. To overcome this limitation, we use clock modulation to control the effective frequency F_{eff} while keeping the voltage constant.

$$W^{total}_{CPU} = \alpha(V) \times F_{eff} + \beta(V) \tag{5}$$

From Equation 5, if $F_{eff} = 0$, then no dynamic power is consumed by the CPU. This means that $W^{total}_{CPU} = \beta(V) = W^{static}_{CPU}$. From Equation 2, we can compute the static power of each active core using Equation 6. Figure 5 shows an example of how W^{static}_{core} is estimated on the HSW-CL machine.

$$W^{static}_{core} = \frac{W^{static}_{CPU} - W^{static}_{uncore}}{ActiveCores} \tag{6}$$

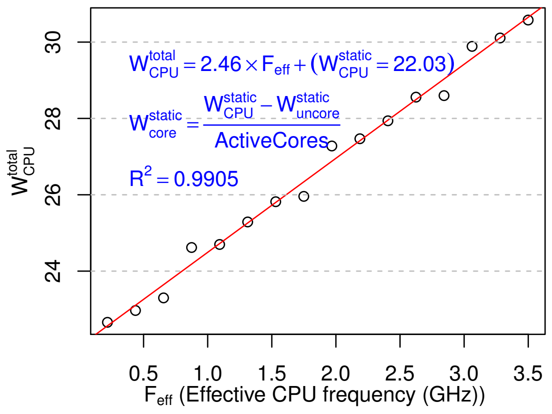

Figure 5: Estimation of core static power on HSW-CL

Table 1 reports estimated uncore static power values and static power of each individual core on both HSW machines.

Static power	HSWE3 (3.5GHz)	HSWEP (2.4GHz)
Uncore (nJ/Cycle)	3.42	6.55
Uncore (Watt)	11.97	15.72
Per core (nJ/Cycle)	0.72	0.79
Per core (Watt)	2.52	1.89

Table 1: Static power estimated values on the HSW-CL and HSW-SE machines. Values are expressed in terms of nano-Joules/Cycle and in terms of Watts

4.2.2 Dynamic energy estimation

After estimating static power of each active core and static power of the uncore, we focus on modeling the total dynamic energy. It consists of the sum of energy consumption of all the HW nodes (N) that we consider as impacting performance. Equation 7 summarizes the dynamic energy of a CPU. Energy consumption of each individual node is proportional to its usage/activity. For example, instruction/FPU

[3]CPU frequency change is also called a DVFS pair change.

nodes are modeled in terms of the number of instructions that are executed.

$$E^{dynamic}_{CPU} = \sum_{node}^{N} E^{dynamic}_{node} \tag{7}$$

FE and INT nodes estimation. Our first step in modeling dynamic energy consumption is to estimate 1) the FE and the INT nodes. The former models the energy consumption in the FE ($E^{dynamic}_{FE}$) in terms of the number of instructions that are issued to the back-end I_{FE}. The second models the energy consumption ($E^{dynamic}_{INT}$) of executing any number of integer loop-control instructions I_{INT}.

We use Equations 8 and 9 to estimate the energy consumption of the FE and INT components. To evaluate $E^{dynamic}_{FE}$, we create micro-benchmarks where we vary the number of non-operation instructions. Such instructions allow us to isolate the behavior of the FE issue stage. While these instructions are issued by the FE and retired/committed by the back-end, they are not dispatched to execution units/ports. We emulate non-operations by running a set of **XCHG %AX,%AX** instructions. Similarly, to evaluate $E^{dynamic}_{INT}$, we use a micro-benchmark where we vary the number of executed integer instructions of the kind **ADD %RAX,%RCX**. These INT instructions stress both the FE and back-end as they are dispatched and use execution ports/units. Solving the linear regression expressed by Equations 8 and 9, allows us to compute the energy for decoding a single instruction e_{FE}, and energy to execute an integer instruction e_{INT}.

$$E^{dynamic}_{FE} = e_{FE} \times I_{FE} \tag{8}$$

$$E^{dynamic}_{INT} = e_{INT} \times I_{INT} \tag{9}$$

LD, ST and FP nodes estimation. At this level, we focus on estimating the dynamic energy of executing multiple floating-point and memory (load and store) operations that hit the L1 cache. We cover the cases of operations using 32, 64, 128 and 256 bits data width. For each desired data width, we use a set of micro-benchmarks where various levels of instruction mixes are created. While it is possible to create a specific micro-benchmark for each instruction for which we want to estimate energy, our observations show that having a mix of instructions allows us to accurately estimate the cost of various instructions. In fact, using mixed instruction micro-benchmarks helps to capture the complex interactions arising from the Out-of-Order execution engine.

For each family of instructions that use a certain portion of the available width, we solve systems expressed by Equation 10. For each instruction $I \in$ {LD, ST, MUL, ADD, DIV, FMA} with a width $w \in \{32, 64, 128, 256\}$, we can express its energy by Equation 11.

$$E^{dynamic}_{FPU} = \sum_{I}\sum_{w} E^{dynamic}_{I,w} \tag{10}$$

$$E^{dynamic}_{I,w} = e_{I,w} \times NbInst_{I,w} \tag{11}$$

L2 energy estimation. Workloads with memory accesses that miss L1 but are L2 hits require modeling these L2 accesses explicitly. To this end, we designed micro-benchmarks composed of a set of strided (64 byte granularity) memory loads and stores. Each access triggers a distinct cache line movement between the L1 and the L2 caches $MemTr_{L2}$, defined as the memory traffic between the L1 and L2 caches.

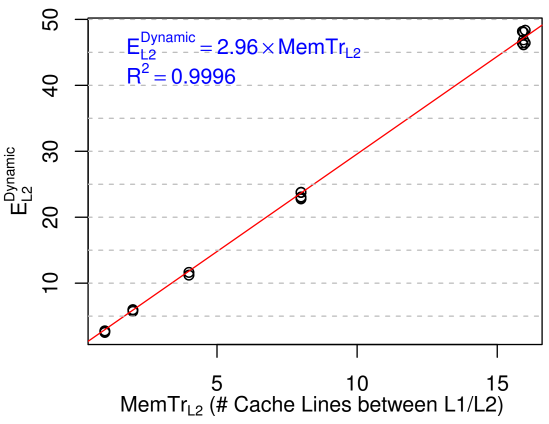

Figure 6: Dynamic energy estimation of L2 accesses per microbenchmark on HSW-CL

Once we measure the total energy and the number of requests to L2, it is possible to solve a linear regression represented by Equation 12, where e_{L2} represents the energy consumption regression variable for moving a single cache line between the L1 and L2 caches. Since the micro-benchmarks are composed of memory loads and stores plus loop-control integer instructions, the LHS of Equation 12 is formed by summing over the energy contributions of FE, INT and L1 nodes from the RHS of Equation 7. Figure 6 shows the L2 access energy estimation on HSW-CL when we vary $MemTr_{L2}$, the number of cache line requests to the L2 from the L1 cache.

$$E^{dynamic}_{L2} = e_{L2} \times MemTr_{L2} \tag{12}$$

L3 energy estimation. The final step in processor dynamic energy consumption focuses on L3 memory access hits. For model training purposes, we use the same micro-benchmarks as for the L2 training, but we choose data sizes to trigger distinct cache line requests to L3, so all the memory accesses will miss L1 and L2 but will hit L3.

To model the energy consumption for cache line movement between the L2 and L3 caches, we define such traffic as $MemTr_{L3}$. We vary the number of distinct cache line requests to L3 while measuring total energy and solve the linear regression (Equation 13) to compute e_{L3} which is the energy necessary to move one cache line between the L2 and L3 caches. As in L2 modeling, the LHS of Equation 13 is formed by summing over the energy contribution of FE, INT, L1 and L2 nodes from the RHS of Equation 7. Figure 7 shows the L3 access energy estimation on HSW-CL when we vary $MemTr_{L3}$, the number of cache line requests to L3, where all the accesses are L1 and L2 misses.

$$E_{L3} = e_{L3} \times MemTr_{L3} \tag{13}$$

Energy coefficient for modeled machines. By applying this methodology, it is possible to assign energy consumption to individual HW node usage. Table 2 reports the

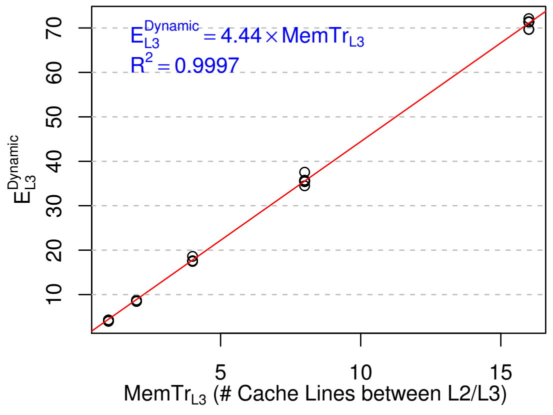

Figure 7: Dynamic energy estimation of L3 accesses per microbenchmark on HSW-CL

energy coefficient of the various HW nodes that we model on both the HSW-CL and HSW-SE machines.

Energy (nJ)/ HW node	HSWE3 (3.5GHz)	HSWEP (2.4GHz)
FE (nJ/Inst)	0.11	0.08
INT (nJ/Inst)	0.14	0.13
ADD (SS/SD) 32/64 (nJ/Inst)	0.33	0.33
ADD (PS/PD) 128 (nJ/Inst)	0.33	0.33
ADD (PS/PD) 256 (nJ/Inst)	0.60	0.45
MUL (SS/SD) 32/64 (nJ/Inst)	0.16	0.13
MUL (PS/PD) 128 (nJ/Inst)	0.30	0.15
MUL (PS/PD) 256 (nJ/Inst)	0.33	0.25
DIV (SS) 32 (nJ/Inst)	3.76	4.19
DIV (SD) 64 (nJ/Inst)	5.59	5.46
DIV (PS) 128 (nJ/Inst)	5.21	4.80
DIV (PD) 128 (nJ/Inst)	6.32	6.15
DIV (PS) 256 (nJ/Inst)	11.83	11.20
DIV (PD) 256 (nJ/Inst)	14.09	17.05
FMA (PS/PD) 256 (nJ/Inst)	0.70	0.63
LD/L1 (SS/SD) 32/64 (nJ/Inst)	0.30	0.15
LD/L1 (PS/PD) 128 (nJ/Inst)	0.33	0.18
LD/L1 (PS/PD) 256 (nJ/Inst)	0.45	0.34
ST/L1 (SS/SD) 32/64 (nJ/Inst)	0.69	0.69
ST/L1 (PS/PD) 128 (nJ/Inst)	0.69	0.69
ST/L1 (PS/PD) 256 (nJ/Inst)	0.78	0.69
L2 (Read/Write) (nJ/64 bytes)	2.96	2.12
L3 (Read/Write) (nJ/64 bytes)	4.59	4.70

Table 2: Energy coefficients expressed in nano-Joule

5. VALIDATION AND RESULTS

We evaluated the accuracy of our energy model using the NR codelets with data sizes chosen to run from L1, L2 and the caches. All of our experimental results were obtained by turning the HW prefetcher off. As the range of errors of all codelets is similar regardless of the data sizes that are chosen, this paper discusses only results of L3 data sizes due to lack of space. As we assume uniform consumption among cores, the presented numbers express unicore run results on multicore systems.

5.1 Energy model error prediction analysis

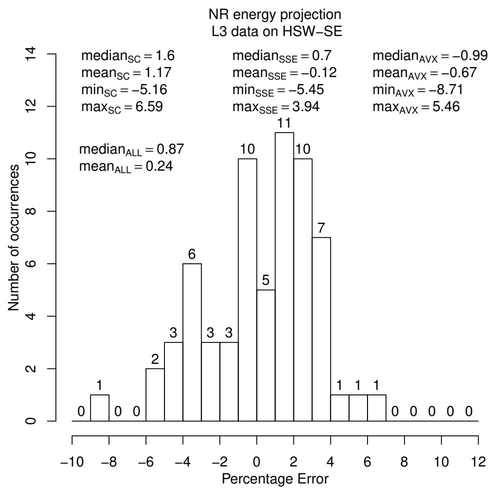

Figure 8: Energy prediction errors when running SC, SSE and AVX2 codelets out of L3 on HSW-SE

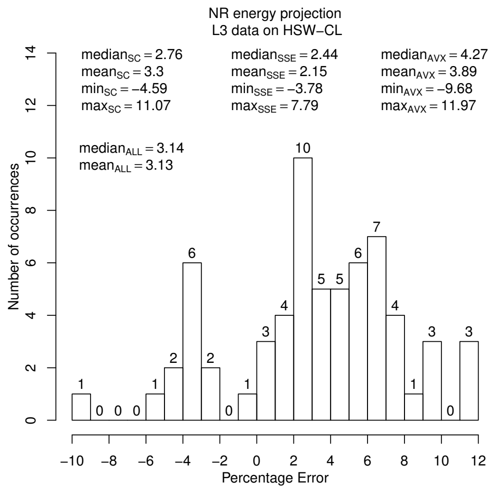

Figure 9: Energy prediction errors when running SC, SSE and AVX2 codelets out of L3 on HSW-CL

Figures 8 and 9 summarize the errors observed when comparing energy measurement to energy values computed by our model for HSW-SE and HSW-CL respectively. Though both histograms report all the error independently of the used instruction set, we also show median, average, minimal and maximal errors of the SC, SSE and AVX2 codelets. We can make the following observations.

1. When errors are greater than 0, the model underestimates energy consumption, otherwise, the model overestimates energy consumption. While the errors are evenly split around the 0% error point on HSW-SE, they are evenly split around the 3% point on HSW-CL.

2. While the proportion of codelets for which energy is overestimated is less than 20% on `HSW-CL`, that proportion is less than 44% on `HSW-SE` though overestimation does not exceed 5% in most cases.
3. On both machines, prediction accuracy varies little across instruction sets. Observed minimal and maximal errors fall in the same range for `SC`, `SSE` and `AVX2`.
4. On both machines, the model overestimates energy consumption for the same set of codelets, which we can split into two sub groups: 1) unit stride and 2) non-unit stride codelets. All codelets belonging to these sub groups share the same memory access and computation patterns.
5. For this study, while energy underestimation is mainly due to non-modeled instructions (like single to double precision conversion instructions), we believe that energy overestimation is a consequence of the complexity of isolating the energy consumption of each modeled HW node independently.

5.2 Total energy characterization results

We can use the energy model in multiple ways. In this paper, we limit our study to analyze the energy consumption of different codelets on one hand, and differences in energy between different instruction sets on the other hand.

As our codelets were compiled to generate a scalar (`SC`) and two vector versions (`SSE` and `AVX2`), we study the energy efficiency of codelets using the quality metrics of Section 2. Figure 10 reports the average total CPU energy E, C/E and C/W for all codelets with the same instruction set on both test platforms. As expected, the more a codelet uses the available vector width, the lower is the energy consumption, and the higher are the ratios C/E and C/W. Figure 10 confirms that vectorization leads to higher performance and lower energy.

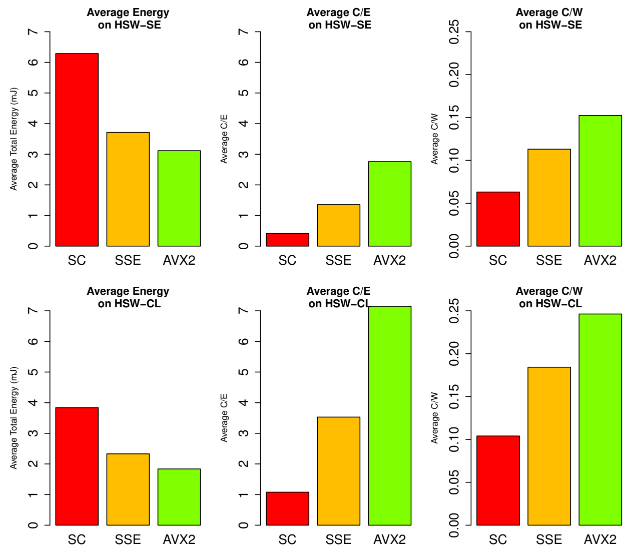

Figure 10: Average total energy (lower is better), C/E (higher is better) and C/W (higher is better) metrics across instruction sets on the `HSW-SE` and `HSW-CL`

Looking at C/E and C/W per codelet and per instruction set in Figures 11 and 12 shows that according to the C/E metric, almost all `SSE` codelets outperform `SC` versions by a factor between 2 to 10X. Similarly, except for few codelets, `AVX2` outperforms `SSE` by up to 3X. Though the magnitudes differ, we also observe the same trend in the C/W metric, and all codelets behave similarly on both machines.

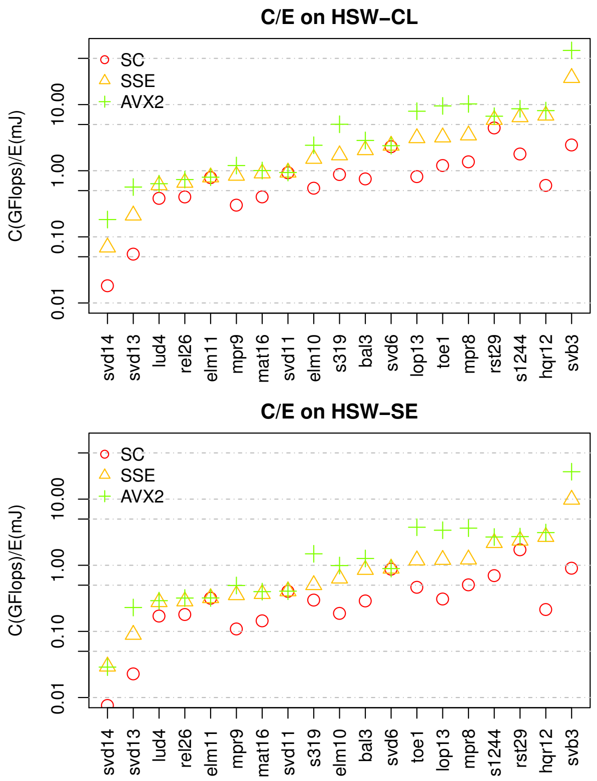

Figure 11: C/E per codelet and per instruction set on the `HSW-SE` and `HSW-CL`

As the analysis of C/E and C/W leads to the same overall observations, we focus on C/E and analyze the source of variation between codelets. To do so, we consider `SSE` codes running on `HSW-CL`. Figure 13 shows the C/E values for each codelet sorted in increasing order. It also shows corresponding C and E values using multiple y-axes.

Table 3 compares the details of 6 codelets selected from Figure 13 for their diversity, arranged in increasing C/E order (over a range of 300X). While high C/E represents an attractive system metric, specific C and E values may vary widely. The table shows a range of 32X for performance and 110X for energy. Table 3 also shows the origin of variations in C and E values. The total amount of arithmetic (flop count), and the performance limiters (one or more nodes whose performance enhancements will enhance total system performance for a given codelet) contribute to performance as does memory activity. While all data sizes were chosen to run from L3, the energy contributions give another view of the HW nodes. Energy comparisons relative to these 6 codelets and absolute node energy relative to a given codelet are shown in the last two columns. Relative and absolute energy contributions of each node are also highlighted in Figure 14.

Svd13 has the lowest C/E value, which clearly arises from both poor performance and high energy because of its dom-

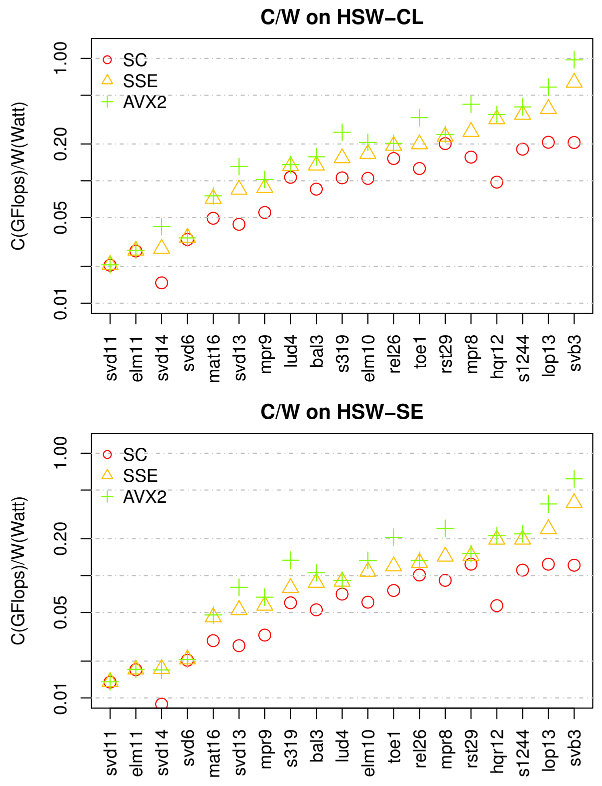

Figure 12: C/W per codelet and per instruction set on `HSW-SE` and `HSW-CL`

ination by division. In fact its relative static energy dominates that of the other codelets, as shown in Table 3 and Figure 14. Relative to other nodes dynamic energy, the FPU (division dominated) and L3 consume most energy.

At the other extreme, `svb3` has the highest C/E and performance and low energy. `svb3` implements a matrix-vector multiply using single precision elements. Its performance is limited by the RS (reservation station) and LM (load matrix). Relative to other codelets, its energy is dominated by L3 and to some extent the FE. Relative to other nodes, its dynamic energy is quite balanced, though L3 energy is slightly higher.

Rstrct is another codelet with high C/E and performance, and relatively low energy. *rstrct* is a finite difference operator stencil that does red/black computation (stride 2 in 2 dimensions), so indexing covers only $\frac{1}{4}$ of the total data per iteration, reducing it from *relax* (also a finite difference operator but using a divide). Performance is limited by the PRF (physical register file) driven by its arithmetic. Its relative energy is dominated by L1 and the front-end, which must process many arithmetic instructions, relative to other codelets. Its dynamic energy is quite well balanced across cache levels and FE.

The other 3 codelets fall in the middle with diverse characteristics. One to note is *svd11* which has the poorest performance due to poor memory behavior, indicated by LFB (line fill buffer) as its performance limiter. It is a 2d array indexed to stride by the array size, which drives the cache energy up.

Because the uncore is shared across many cores on chip, the percentage of dynamic energy will actually increase by a factor proportional to the number of cores used for a multicore run.

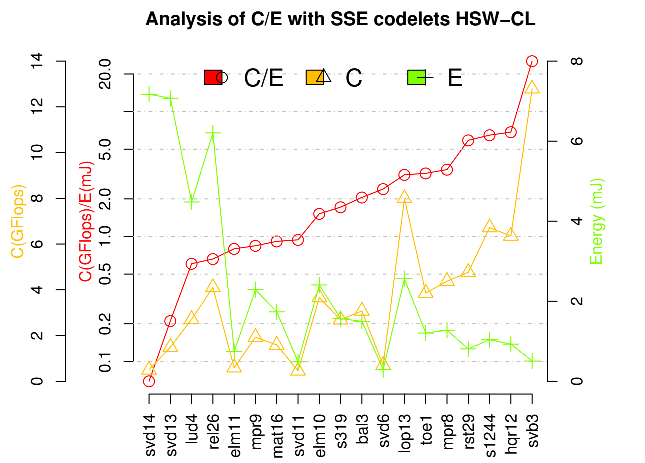

Figure 13: Computed C/E, C and total E for `SSE` codes running on the `HSW-CL` machine

Codelet	C/E	Perf (C)	MFlops	Perf limiter	E	E high rel to codelets	Dyn E high rel to nodes
Svd13	0.2	1.5	0.6	Divide	7.1	Static uncore, FPU	FPU, L3
Relax	0.7	4.1	1.2	PRF, some /	6.2	L2, FPU	L2, FPU
Svd11	0.9	0.5	0.01	LFB, LDA	0.5	L2, L3	L3
Toep1	3.2	3.9	0.24	LM	1.2	L3	L3
Rstrct	5.9	4.8	0.19	PRF	0.8	L1, FE	L3
Svb3	25.4	12.8	0.32	LM	0.5	L3,FE	L3

Table 3: Energy and performance comparison of 6 `SSE` codelets selected from the C/E range

5.3 Vectorization analysis results

It is a common belief that vectorization is most likely to lead to better performance and energy savings. Figure 15 shows energy consumption of `SC`, `SSE` and `AVX2`. Codelets are sorted relative to increasing order of `SC` energy. While the y-axis on the left side reports energy values, the y-axis on the right side reports the ratio of energy between `SC` to `SSE`, between `SC` to `AVX2` and between `SSE` to `AVX2`.

Clearly, vectorization causes important variations in energy reductions. We observe both extremes, nearly no effect and high energy savings (about 4X). Moreover, as we reported in Figure 10, Figure 15 shows that `AVX2` and `SSE` codelets are 2.2X and 1.6X more energy efficient on average than `SC` codelets. However, it shows as far as L3 data is concerned that `AVX2` codelets are more energy efficient than `SSE` ones by only 1.3X.

To highlight the HW nodes that benefit most from vectorization in terms of energy, consider the codelets discussed in Table 3. As these codelets exhibit a wide range in E, they are good candidates to illustrate energy efficiency due to vectorization. Table 4 shows the absolute energy contribution

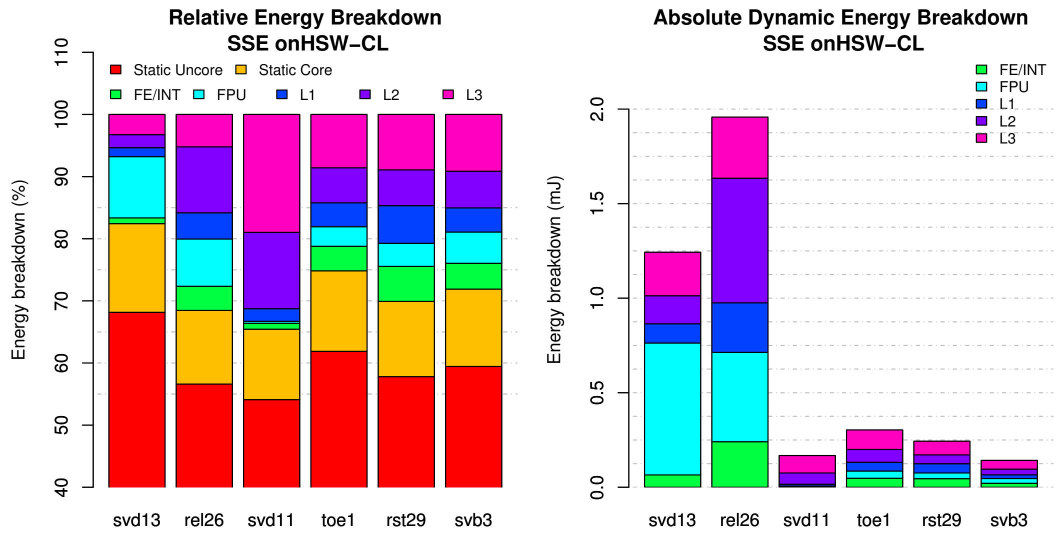

Figure 14: Relative and absolute energy contribution of HW nodes on 6 selected SSE codelets.

of HW nodes for each of the selected codelets on the HSW-CL machine. Energy breakdown information is provided for each of SC, SSE and AVX2 versions of each selected codelet.

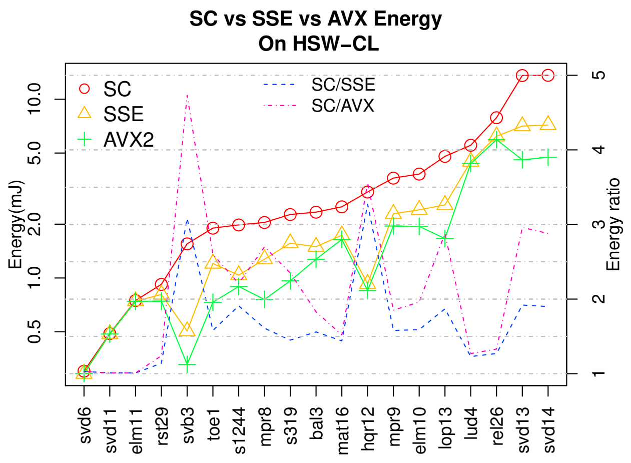

Figure 15: Comparing total E of SC, SSE and AVX codes on the HSW-CL machine

Overall, analyzing the energy breakdown of the 6 codelets allows us to see that except for svd11, both static and dynamic energy are reduced when going from scalar to vectorized codelet. As far as dynamic energy is concerned, Table 4 shows that the dynamic energy of the L2 and L3 caches remains unchanged regardless of the instruction set. This is true as the memory traffic between the L1 and L2 caches and the memory traffic between the L2 and L3 caches remains the same (the same number of cache lines is required to do the computations). In practice, most energy savings are due to the FE, FPU and the L1. This can be explained easily by the fact that these nodes correlate very well with the number of instructions that are executed and vectorized codelets usually lead to lower numbers of instructions than scalar codelets, so energy is reduced accordingly.

Svd11 is among the codelets where vectorization does not lead to energy savings. Svd11 is an LDA (leading dimension access) codelet, where only the arithmetic operations are vectorized. Therefore, it is dominated by memory operations. Since the data size is chosen to come from the L3 cache, its dynamic energy is dominated by L2 and L3 accesses. Consequently, partial vectorization brings neither performance nor energy savings. **relax** and **rstrct** (like **svd11**) have a non-negligible number of scalar memory operations, but unlike svd11, these stencil codes also have vectorized memory operations. Consequently, vectorization brings only unimportant energy improvements.

One of the codelets that benefits most from vectorization is svb3, a matrix-vector multiply codelet with unit stride memory accesses. Though its 2D data fits the L3 cache, the vector part fits in the L1. So only the matrix elements need to be brought to the L1. Moreover, as the matrix and the vector are single precision floating-point elements, SSE and AVX2 versions are able to process 4 and 8 elements in each vector, respectively. This leads to the observed energy savings. Though AVX2 instructions have twice the width of SSE instructions, energy saved on this codelet does not reflect that. In fact, the vectorized version of svb3 is RS and LM performance limited. Consequently, SC to AVX2 energy savings are only slightly higher than those of SC to SSE.

Vectorization is also effective on toep1. However, unlike svb3, toep1 uses double precision elements instead of single. Finally, even a high energy consumption codelet like svd13 benefits from vectorization. In fact, as the L1, FPU and FE energy drop to almost half, its static energy is also reduced in the same proportion due to time reductions from vectorization.

6. RELATED WORK

The need for dynamic power management techniques [1, 3, 22] or HW design trade-offs [13] has motivated many research efforts to develop power and energy models for processors. These models can be classified in terms of whether they rely on some performance metrics (hardware performance counters) or rely on cycle-accurate/RTL simulators.

Codelet	Static E			FE			FPU			L1			L2			L3			Total Dynamic		
	SC	SSE	AVX	SC	SSE	AVX	SC	SSE	AVX	SC	SSE	AVX	SC	SSE	AVX	SC	SSE	AVX	SC	SSE	AVX
Svd13	11.7	5.83	3.35	.13	.07	.05	1.22	.70	.74	.20	.10	.06	.15	.15	.15	.23	.23	.23	1.93	1.24	1.23
Relax	5.49	4.25	3.99	.26	.24	.29	.83	.47	.44	.29	.26	.23	.70	.66	.67	.32	.32	.32	2.40	1.96	1.95
Svd11	.32	.32	.32	.01	.00	.01	.00	.00	.00	.01	.01	.01	.06	.06	.06	.09	.09	.09	.17	.17	.17
Toep1	1.52	.90	.50	.05	.05	.02	.06	.04	.02	.10	.05	.02	.07	.07	.07	.10	.10	.10	.38	.30	.23
Rstrct	.66	.57	.52	.03	.05	.04	.05	.03	.02	.06	.05	.04	.05	.05	.05	.07	.07	.07	.26	.24	.22
Svb3	1.21	.36	.21	.06	.02	.01	.08	.03	.02	.12	.02	.01	.03	.03	.03	.05	.05	.05	.34	.14	.11

Table 4: Absolute energy breakdown (mJ) of 6 selected codelets on `HSW-CL` machine

Architecture-level simulators like McPAT or CACTI-P [13, 14] can provide detailed and accurate information about energy or power consumption of modern multicore processors. However, simulation speed can be a limitation for software tuning and optimization.

To drive power management policies at runtime, many power models were built using hardware performance counters. Similar to our work, the idea is to associate energy weights to activity factors computed using hardware counters that correlate the most with performance. Such approaches are developed in [3, 22] and applied in the context of DVFS controllers to adjust CPU frequency to the demand on the machine. Though useful, these techniques do not provide an accurate distribution of energy among the different HW components or associate energy consumption with code characteristics. As these models rely on general linear regressions, some power or energy coefficients have negative slopes which are meaningless from a physics point of view. Consequently, it is hard to understand and estimate the amount of power or energy that micro-architectural components consume.

Bertran et al. [1] discuss an incremental methodology to build a power model to isolate the power consumption of main CPU components. The proposed model does not make explicit separation between the static and dynamic power consumption. Goel et al. [6] propose a methodology that explicitly separates between dynamic and static power consumption. However, the methodology they follow requires some special boards (overclocked CPU for the gaming community) to model the static power. Such custom HW allows explicit voltage tuning for core and uncore components.

McCullough et al.[18] compare the effectiveness of various power modeling techniques for system components. Like us, they turned off Turbo-Boost and HT. They conclude that CPU power-models can lead to non-negligible prediction errors due to hidden states, and advice to not rely on these models. We agree that parameters that are not explicitly exposed by HW may lead to prediction errors, but unlike their tested modeling techniques, an incremental approach can help isolating the energy contribution of various components compared to single linear or non-linear regressions.

Sahoa et al. [21] discuss a methodology where energy consumption is estimated in terms of the mix of vector instructions that are executed on the Xeon Phi processor. In fact, no separation between the various operations is made as the cost of a multiply can be very different from the one for an add operation. Unlike these previous research efforts, we correlate energy consumption to software characteristics like the different instruction sets that the hardware supports. We also make an explicit distinction between static and dynamic energy. This methodology allows us to be more fine grained when associating energy to the generated code.

Other research efforts focused on studying energy efficiency of different HW. Czechowski et al. [4] discuss energy and power improvement for various generations of Intel processors. Gupta et al. [8] study energy efficiency of the uncore component compared to the core component. All these studies focus on hardware and do not provide insight on how effective the HW can be for a particular code.

From a software tuning perspective, Cebrián et al. [2] discuss energy savings due to vectorization using two benchmarks. While they show and confirm that vectorization improves performance while decreasing energy consumption, they do not show which the components benefit the most from vectorization. Thanks to our methodology, we can give a detailed breakdown of energy consumption and drive software tuning to generate better code.

7. CONCLUSION

In this paper, we presented a methodology to build linear energy models for multicore processors. Our method starts with high-level energy and performance measurement and drives low-level energy breakdown information among a set of HW nodes. Also, our methodology presents a new way of separating static and dynamic power at the level of the core and the uncore, using the clock modulation feature and a series of multicore runs. Thus, our approach has the ability to associate the energy consumption of various HW nodes to the quality of the generated code, helping developers in the process of code optimization and tuning.

Using a set of codelets, our experimental evaluation shows an average error of 3%, regardless of the test machine, data size (footprint) or the instruction set we use. These results show that our methodology can be the basis of tools to analyze the energy efficiency of hardware and the quality of the analyzed code. We show that while total static energy can be as high as 80%, the static energy of the uncore component can contribute up to 70% (Figure 14). Our numbers suggest that this requires much attention as it is a good candidate for energy consumption reduction.

Using a set of quality metrics, we analyze differences between various codelets and show that HW nodes consuming most energy correlate well with performance limiters. In fact, our method allows us to link energy consumption to the various algorithms that are implemented in the codelets we use. We can also show and explain why vectorization using the `AVX2` and `SSE` instruction sets leads to higher performance and lower energy consumption. Though few codelets have similar energy consumption regardless of the instruction set, the general trend shows that `AVX2` is more energy efficient than `SC` or `SSE`.

Our work has been limited to multicore systems in throughput mode, with prefetch, HT, and Turbo-boost disabled. We plan to extend the model to deal with prefetch and HT on. Since Turbo-boost has hidden HW state activities, controlled measurement and modeling are probably too difficult to do with low, predictable errors. We do plan to extend our methodology to parallel workloads on multicore systems, and to accelerator-like systems.

8. REFERENCES

[1] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade. Decomposable and responsive power models for multicore processors using performance counters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 147–158, New York, NY, USA, 2010. ACM.

[2] J. M. Cebrián, L. Natvig, and J. C. Meyer. Performance and energy impact of parallelization and vectorization techniques in modern microprocessors. *Computing*, 96(12):1179–1193, Dec. 2014.

[3] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 19(10):1396–1410, Oct 2008.

[4] K. Czechowski, V. W. Lee, E. Grochowski, R. Ronen, R. Singhal, R. Vuduc, and P. Dubey. Improving the energy efficiency of big cores. *SIGARCH Comput. Archit. News*, 42(3):493–504, June 2014.

[5] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 189–194, New York, NY, USA, 2010. ACM.

[6] B. Goel and S. A. McKee. A methodology for modeling dynamic and static power consumption for multicore processors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 273–282, May 2016.

[7] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose processors. In *Low Power Electronics, 1995., IEEE Symposium on*, pages 12–13. IEEE, 1995.

[8] V. Gupta, P. Brett, D. Koufaty, D. Reddy, S. Hahn, K. Schwan, and G. Srinivasa. The forgotten 'uncore': On the energy-efficiency of heterogeneous cores. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 34–34, Berkeley, CA, USA, 2012. USENIX Association.

[9] D. Hackenberg, T. Ilsche, R. Schone, D. Molka, M. Schmidt, and W. E. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 0:194–204, 2013.

[10] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual: System programming guide, September 2016.

[11] W. Jalby, D. C. Wong, D. J. Kuck, J. Acquaviva, and J. C. Beyler. Measuring computer performance. In M. W. Berry, K. A. Gallivan, E. Gallopoulos, A. Grama, and B. Philippe, editors, *High-Performance Scientific Computing - Algorithms and Applications.*, pages 75–95. Springer, 2012.

[12] D. J. Kuck. *Computational Capacity-Based Codesign of Computer Systems*, pages 45–73. Springer London, London, 2012.

[13] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[14] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, pages 694–701, Piscataway, NJ, USA, 2011. IEEE Press.

[15] MAQAO. Maqao project, 2016.

[16] A. Mazouz, B. Pradelle, and W. Jalby. Statistical validation methodology of cpu power probes. In *Revised Selected Papers, Part I, of the Euro-Par 2014 International Workshops on Parallel Processing - Volume 8805*, pages 487–498, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[17] A. Mazouz, S. A. A. Touati, and D. Barthou. Study of variations of native program execution times on multi-core architectures. In L. Barolli, F. Xhafa, S. Vitabile, and H. Hsu, editors, *CISIS 2010, The Fourth International Conference on Complex, Intelligent and Software Intensive Systems, Krakow, Poland, 15-18 February 2010*, pages 919–924. IEEE Computer Society, 2010.

[18] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta. Evaluating the effectiveness of model-based power characterization. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[19] J. Noudohouenou, V. Palomares, W. Jalby, D. C. Wong, D. J. Kuck, and J. C. Beyler. Simsys: A performance simulation framework. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM.

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA, 1992.

[21] Y. S. Shao and D. M. Brooks. Energy characterization and instruction-level energy model of intel's xeon phi processor. In *International Symposium on Low Power Electronics and Design (ISLPED), Beijing, China, September 4-6, 2013*, pages 389–394, 2013.

[22] V. Spiliopoulos, A. Sembrant, and S. Kaxiras. Power-sleuth: A tool for investigating your program's power behavior. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 241–250, Aug 2012.