Energy Usage for Parallel Haskell Programs

Yasir Alguwaifli
School of Computer Science
University of St Andrews, UK
ya8@st-andrews.ac.uk

Kevin Hammond
School of Computer Science
University of St Andrews, UK
kevin@kevinhammond.net

Theodoros Dimopoulos
School of Computer Science
University of St Andrews, UK
td41@st-andrews.ac.uk

Christopher Schwaab
School of Computer Science
University of St Andrews, UK
cjs26@st-andrews.ac.uk

ABSTRACT

Understanding and controlling software energy usage is an increasing concern in many settings. To date, there has been little work, however, on understanding how energy relates to high level programming constructs, or on dealing with real parallel systems, especially at a significant scale. In this paper, we measure and correlate the energy usage of several parallel Haskell programs against execution time and other runtime system (RTS) metrics, produced using the standard Haskell compiler, GHC. We show how energy usage relates to the number of cores, giving performance results on a recent Intel Xeon x86 system that has 28 physical cores and supports hyperthreading. From these results, we construct an energy model and relate this model to the parallel structure of the program source. Given a suitable structural model of parallelism, this provides information that can be used to predict the energy usage for a concrete Haskell program running a real multicore system.

KEYWORDS

Parallelism, Multicore, Energy Usage, Functional Programming, Haskell, Performance Modelling

ACM Reference Format:
Yasir Alguwaifli, Kevin Hammond, Theodoros Dimopoulos, and Christopher Schwaab. 2017. Energy Usage for Parallel Haskell Programs. In *Proceedings of Intl. Symp. on Implementation and Application of Functional Langs., Bristol, UK, Aug. 2017 (IFL'17),* 12 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

There has recently been an increased focus on the energy that is used by computing devices. Predicting the energy that is used by a computation is important in settings such as the Internet of Things (IoT), where devices may rely on limited battery or solar power to operate effectively. In some situations (e.g. drones), exceeding energy limitations can lead to mission failure, system loss, or even safety issues. We are exploring these issues in the newly funded EU Horizon 2020 TeamPlay project, which aims to investigate trade-offs between time, energy and safety/security.

There has been significant work on hardware-directed analysis of energy/power costs and on different methods for scaling and reducing energy usage [2, 13]. These measurement-based approaches can provide useful information about the energy that is used by a program. However, they lack analytical and predictive power. Building on such measurements, some authors have related energy usage to a machine's instruction set architecture [5, 11, 14, 16]. However, such approaches typically deal only with fixed execution paths and closed program inputs. Loops and recursion can be problematic, for example. They also do not deal well with the complex micro-architectural features that are found in more advanced processor designs, such as caches and pipelines, and are generally restricted to sequential systems. There has been very little work to date on correlating energy usage with higher levels of programming abstraction, to provide models of energy usage that can be directly used by the programmer to the program source, or on extending the energy models to properly consider parallel execution. In this paper, we consider the energy that is used by parallel Haskell programs, compiled using a recent version of the state-of-the-art GHC compiler and runtime system. We study a number of examples taken from the well-known **NoFib** benchmarking suite [6], constructing statistical energy models for our benchmarks using measurements taken by the **RAPL** hardware energy measurement system [17] for a recent 28-core Intel Xeon processor.

1.1 Source-Level Energy Analysis

Energy analysis of software is still a relatively new area. Work has mostly focused either on simple program-level measurements of a program's energy usage, or else on relating energy usage to an instruction-level representation of the program (often without consider high-level language features such as looping, ignoring data dependencies, and without considering common, but hard-to-understand, micro-architectural features, such as pipelines etc.). For example, [5, 11, 14, 16] have given an overview of a framework that can predict energy for various languages that use LLVM. This work has involved analysing, predicting and minimising energy consumption in imperative languages by considering instruction level costs. While GHC can generate code from *Cmm* via LLVM, this is not the default option for code generation. Moreover, this method remains tightly coupled to the LLVM compiler infrastructure and cannot link execution costs to high level program constructs or to the program as a whole. For example, Georgiou *et al.* have developed and applied an energy consumption static analysis methodology (*ECSA*) that maps the intermediate representation (*IR*) of LLVM to low-level assembly language constructs [8]. Using this approach, they have been able to analyse the energy consumption of software running on an embedded system with a cache-less ARM processor. While the authors claim that this approach achieves accurate prediction, it remains tightly coupled to a specific compiler infrastructure, a specific IR, and a specific hardware implementation. Other work, by Chowdury *el al.* [3], has measured the the overall energy consumption for a number of industrial software systems and constructed a system-level model. Their approach uses *system calls* (syscalls) to model the energy usage, and they have built a large dataset of examples. However, the approach fails for applications that do not make regular system calls, or which involve significant amounts of computation or memory accesses between calls. Finally, in a Haskell setting, Lima *et al.* [12], have investigated energy consumption patterns over a number of data-structures taken from the *Edison* library of purely functional data structures. They present results that show how to minimise energy consumption when switching between alternative mutable primitives (*TMVar* & *MVar*), and report non-trivial improvements in energy consumption for certain data structures when specific operations are used. In contrast, in this paper we consider all kinds of Haskell applications, constructing whole-program energy models that can be related to scalable parallel program execution.

1.2 Novel Contributions

The main contributions of this paper are:

(1) we provide an empirical evaluation of the energy usage of a number of parallel Haskell programs, compiled using GHC and running on a modern 28-core Intel Xeon system, and relate this to total execution time and the parallel speedups that we obtain (Section 3);
(2) based on our evaluation, we show that there is a strong correlation between energy usage and total execution times (Section 4);
(3) we construct new statistical models for the energy usage of our benchmark programs (Section 4);
(4) confirming Dimopoulos's previous work [4], we show that the benchmarks fall into a number of clearly identifiable classes in terms of their patterns of energy usage (Section 4);
(5) we show how garbage collection impacts energy consumption, and show that reducing garbage collection can have beneficial energy properties, even if execution time is increased (Section 5).

2 METHODOLOGY & ENVIRONMENT

Our work initially focused on understanding the relationship between *energy consumption* and execution time for compiled Haskell programs in a simple, memory-limited, but parallel system, as might be used in a high-end Internet-of-Things application, for example. We used a Raspberry Pi model 3 for our experiments, running 4 ARM Cortex-A53 processor cores at 1.2 GHz, on an Asus Tinkerboard, and used a USB multimeter to collect various power/energy details in real-time. Our aim was to infer *energy* consumption statically by building a test dataset of different execution costs per processor core. Our approach involved running a program and observing the total energy usage for the system, using a real-time energy monitor, then applying *linear interpolation* to predict a program's total energy consumption for different numbers of cores, based on the *productivity* metric i.e. the recorded CPU utilisation) obtained from the *RTS*. Unfortunately, we found that this did not always give accurate predictions of total energy usage. Table 1 shows the results that we obtained using this method for three simple compute-intensive applications. While our model gave reasonably accurate (under-)predictions of energy usage for `queens` and `fibonacci`, there was significant divergence for `partee`. Since we experienced a number of problems with our experimental setup, including issues of frequency scaling, overheating and memory limitations, and we were also interested in scalability to larger numbers of cores, we therefore decided to instead consider a new 28-core Intel Xeon system at the University of St Andrews.

2.1 System Configuration

Our experimental testbed machine is a 2-socket shared memory system at the University of St Andrews, *corryvreckan*, containing two Xeon E5-2690, each of which has 14 physical cores (28 virtual cores), giving a total of 28 physical cores (56 virtual cores). It has 256GB of physical DRAM, which is shared among all the cores. We have configured this system to run at a stable *2.601 GHz*, eliminating thermal throttling and clock boosting drivers e.g. as used by Intel's

	No. Cores			
Tested Program	1	2	3	4
queens	1.7130 (**1.6914**)	2.2903 (**2.2604**)	2.8042 (**2.7415**)	3.3208 (**3.2300**)
fibonacci	1.6897 (**1.588**)	2.0846 (**1.8314**)	2.4304 (**2.1565**)	2.7363 (**2.4560**)
partree	1.9181 (**1.4674**)	2.6609 (**1.6262**)	3.0856 (**1.8059**)	3.5033 (**1.9942**)

Table 1: Predicted versus measured energy usage (in Watts) for three benchmarks running on the Raspberry Pi model 3. Figures in parentheses represent the predictions using our original (discarded) model.

P-State. The machine runs *Red Hat 4.8.5-11* with kernel version *3.10.0-514.21.2.el7.x86_64*, and to stabilise our results, we have restricted it so that it is used only for our experimental purposes. We have used GHC 7.6.3, the default version of the Haskell compiler that is installed on this system, and run all our experiments with optimisations turned on (`-O2`).

2.2 Measurements

We collect basic energy measurements for each benchmark in terms of the total number of *Joules* of energy (*J*) that are used by each *processor package*. That is, we measure the total energy that is drawn by a single hardware CPU socket within the computing system. In order to obtain representative general results, we use four different problem sizes for each benchmark. We use Intel's *Running Average Power Limit* (RAPL) [9] system to collect this information. Access to the RAPL measurements is made available through model-specific registers (*MSR*). These provide accurate [17, 19] power readings for various systems components within each package e.g. the GPU and DIMMs. The same mechanism also allows energy consumption to be controlled/capped. We sample processes with an interval of *100 milliseconds*. Each measurement is computed for a single execution of each of our parallel benchmark programs. RAPL records power usage in terms of *watts*. We calculate energy usage from this by computing the rate of change in power per unit of time as shown below:

$$\begin{aligned} \text{Energy} &= \text{Power} \times \text{Time} \\ \text{Joules} &= \text{Watts} \times (\text{Unit of time: seconds, mins., } \ldots) \end{aligned}$$

2.3 Experimental Approach

We constructed our dataset of base results by collecting basic execution metrics from the GHC runtime system (*RTS*) and RAPL. GHC's RTS provides a number of key program metrics that are related to a specific program execution, including total execution time, the split into *mutation time*[1] and *garbage collection* times, and the percentage *productivity* for a parallel system in terms of the overall improvement in *mutation* time (i.e. the average useful CPU utilisation, where 100% represents perfect utilisation on a single CPU core). For our test cases, we also collect the *total elapsed* time, which represents the wall-clock time, and *power*, which is the wattage drawn per unit of time. Our dataset comprises multiple samples that have been collected from several different executions of each benchmark program. We sample each program in our described environment over four different inputs for each number of processor cores. This means that we start with a certain problem size for a given program and study its sequential execution costs. We then increment the number of cores in subsequent executions: this helps mitigate the effect of caching and variations in core temperatures, so avoiding misleading energy results. Each sample has been collected using four different problem sizes.

2.4 Benchmarks

We have used a number of examples taken from the widely-used **Nofib** parallel benchmark suite:

- **Sum Euler** (*sumeuler*): A parallel implementation of Euler's Totient function which finds the *coprimes* below some integer, *n*. The implementation uses *splitAtN*, as explained in [1].
- **Ray Tracer** (*ray*): A parallel implementation of the ray tracing algorithm as described in [10]. This uses a form of *divide-and-conquer* algorithm, exploiting structured parallelism and the Haskell *Strategies* library.
- **N-Queens** (*queens*): A parallel implementation of the classical *eight queens* problem, extended to arbitrary board sizes ($N \times N$). The implementation evaluates solutions using a parallel list strategy, *parList*.
- **N-Body** (*nbody*): A parallel implementation that finds the motion of celestial objects in 3D space. The implementation uses Intel's Concurrent Collections for Haskell [15] (*CnC*).
- **Parallel Map Over A Tree** (*partree*): A parallel algorithm that generates a binary tree and applies a worker function to its leaves in parallel using the Haskell *Strategies* library.

2.5 Regression Analysis

Since, in order to perform an effective analysis of the energy that is used by a program, we need to find the most significant factors that govern its energy usage, we will investigate our empirical results using a number of different

[1]The time that is spent on directly running the Haskell program.

regression analysis techniques. Specifically, we will study several methods that aim to minimise the number of *features* that are used to determine our energy model. The features that we have chosen to measure are: *productivity*, *duration*, and *power*. We use the model described in Equation (1), below, where β_0, $\beta_1 \ldots \beta_{p-1}$ refers to the coefficient of the features that are used in the model and p refers to the number of coefficients in the model. $p-1$ is thus the number of coefficients including the intercept and the i^{th} value refers to individual observations of the x−variables. Finally ϵ_i has a normal distribution for the i^{th} population error with mean of zero and constant variance of σ^2. Using the model, we can compute least squares estimates $\hat{\beta}_0 + \hat{\beta}_1 + \ldots + \hat{\beta}_p$ that minimise the "sum of squares error" (*SSE*). Finally, the e_i compute the difference in the expected value of $E(Y_i)$ (*a.k.a* $\hat{Y}_i$) and the actual value of Y_i.

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \ldots + \beta_{p-1} x_{i,p-1} + \epsilon_i \tag{1}$$

$$\begin{aligned} \hat{Y}_i &= \hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \hat{\beta}_2 x_{i2} + \ldots + \hat{\beta}_p X_{ip} \\ e_i &= Y_i - \hat{Y}_i \\ SSE &= \sum_{i=1}^{n} e_i^2 \end{aligned} \tag{2}$$

LASSO: Least Absolute Shrinkage Selector Operator

Least Absolute Shrinkage Selector Operator (LASSO) [18] is a regression analysis method that penalises the additional features in a given dataset that ultimately reduce some β coefficients to zero, and so reduce the number of features in a model. This is a form of regularisation that is called *l1*. LASSO adds a λ component to the *sum of squares error* (SSE) in Equation (3) to the *sum of squares error* with a value of $\lambda > 0$ that shrinks the β coefficients to zero for some of the features that are less explanatory of the model.

$$\underset{\beta}{\arg\min} \left\{ SSE + \lambda \textstyle\sum_{i=1}^{p} |\beta|_i \right\} \tag{3}$$

Ridge: Ridge regression

Ridge regression (Equation (4)) is another method to shrink the components of highly correlated independent variables that potentially do not help explain the model. Unlike LASSO, Ridge does not exclude any coefficients from the model. However, it still uses the λ cost function to shrink highly correlated variables.

$$\underset{\beta}{\arg\min} \left\{ SSE + \lambda \textstyle\sum_{i=1}^{p} |\beta|_i^2 \right\} \tag{4}$$

Elastic Net

Elastic Net [20] is another regularisation method that combines techniques from *LASSO* and *Ridge*. In equation (5), the two *shrinkage* (β) terms of λ_1 and λ_2 are used to combine the penalty to features and to exclude insignificant ones. The α term is defined as the threshold that controls the bias between LASSO and Ridge, where $0 \le a \le 1$ (1 being full LASSO, and 0 being full Ridge).

$$\begin{aligned} & \underset{\beta}{\text{argmin}} \left\{ SSE + \lambda_2 |\beta|_i^2 \; + \lambda_1 |\beta|_i \right\} \\ \Rightarrow \quad & \underset{\beta}{\text{argmin}} \left\{ SSE + \; (1-\alpha)|\beta|_i^2 \; + \alpha|\beta|_i \right\} \\ & \text{where} \quad \alpha = \frac{\lambda_2}{\lambda_2 + \lambda_1} \end{aligned} \tag{5}$$

3 RESULTS

Plotting the collected samples of parallel speedup and energy usage for the default GHC execution parameters (i.e. without core affinity and using the default stack/heap configuration for GHC – 8MB of stack and 512KB of heap) gives the results shown in Figures 1–5. For the first three benchmarks: *sumeuler* (Figure 1); *queens* (Figure 2); and *nbody* (Figure 3), we can observe a consistent pattern of parallel speedups that are proportional to the number of cores in use, with corresponding reductions in energy usage. In all three cases, this behaviour is consistent across multiple problem sizes.

sumeuler: For *sumeuler* (Figure 1), we have chosen the *intervals* to range from *75,000* to *90,000* with a chunk size of *400*. In all the cases that we have measured, we observe that we obtain near linear speedups when we are using the 28 physical cores (22.45 on 28 physical cores for *n=90,000*, and 22.01 for *n=75,000*). When using virtual cores, however, the speedups continue to improve slightly, but not linearly. When we reach 56 virtual cores, we observe maximum speedups of 32.46 for *n=90,000* and 30.95 for *n=75,000*). As we increase the core count, we can also clearly see corresponding reductions in energy usage, from a maximum of 16084.77J on one core to a minimum of 1656.38J on 56 cores at *n=90,000*, and from a maximum of 10964.95J on one core to a minimum of 1159.46J on 56 cores at *n=75,000*.

queens: For *queens*, we have chosen the number of queens to range from *13* to *16*. In all the cases that we have measured, we observe that we obtain near linear speedups when we are using the 28 physical cores (10.07 on 28 physical cores for *n=16*, and 4.61 for *n=13*). When using virtual cores, however, the speedups continue to improve slightly, but not linearly. When we reach 56 virtual cores, we observe maximum speedups of 11.22 for *n=16* and 3.74 for *n=13*). As we increase

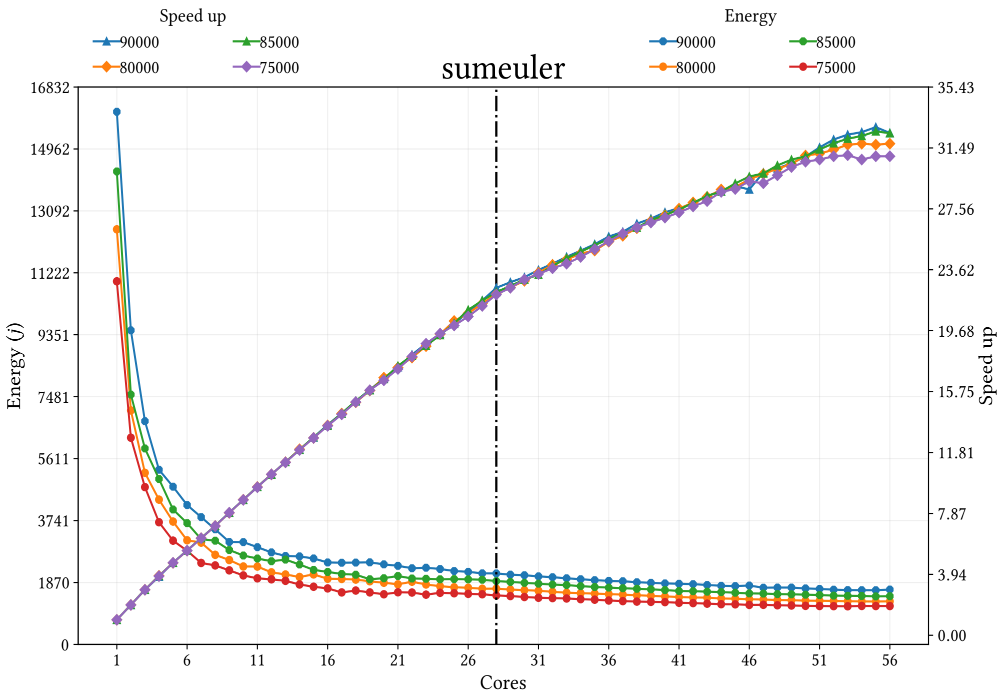

Figure 1: Sumeuler – *Energy vs. Speed up* for intervals from 75,000 to 90,000; chunk size 400

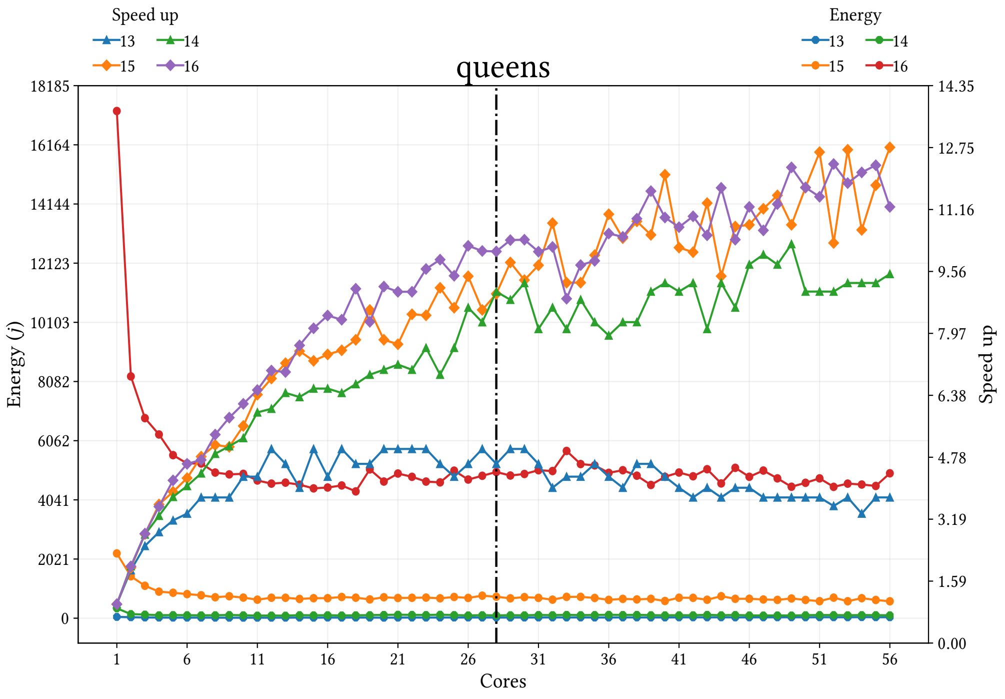

Figure 2: Queens – *Energy vs. Speed up* for numbers of queens from 13 to 16

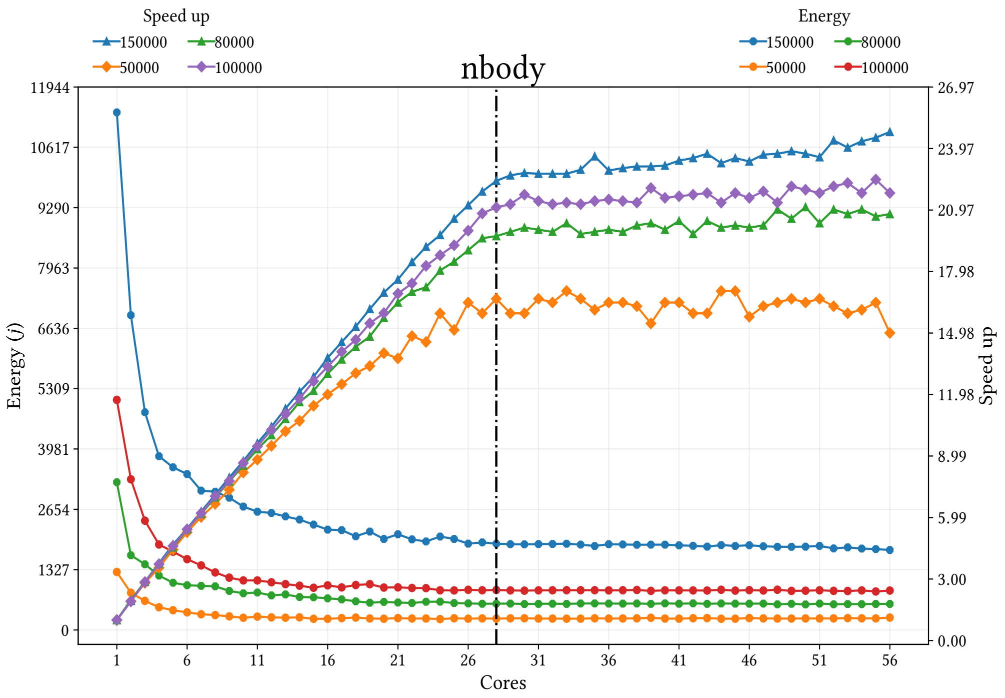

Figure 3: Nbody – *Energy vs. Speed up* for numbers of bodies from 50,000 to 150,000

the core count, we can also clearly see corresponding reductions in energy usage. As we increase the core count, we can also clearly see corresponding reductions in energy usage, from a maximum of 17318.85J on one core to a minimum of 4948.31J on 56 cores for *16* queens, and from a maximum of 45.01J on one core to a minimum of 34.23J on 56 cores for *13* queens.

nbody: For *nbody* (Figure 3), we have chosen the number of particles to range from *50,000* to *150,000.* In all the cases that we have measured, we observe that we obtain near linear speedups when we are using the 28 physical cores (22.39 on 28 physical cores for *n=150,000*, and 16.64 for *n=50,000*). When using virtual cores, however, the speedups continue to improve slightly, but not linearly. When we reach 56 virtual cores, we observe maximum speedups of 24.77 for *n=150,000* and 14.98 for *n=50,000*). As we increase the core count, we can also clearly see corresponding reductions in energy usage, from a maximum of 11385.90J on one core to a minimum of 1759.25J on 56 cores at *n=150,000*, and from a maximum of 1274.95J on one core to a minimum of 272.72J on 56 cores at *n=50,000.*

ray: The other two benchmarks: *ray* (Figure 4) and *partree* (Figure 5) show rather different results. For *ray* (Figure 4), we have chosen the *detail/resolution* to range from *1300* to *1600.* Although we observe some speedups on 3-6 cores in all cases (up to 1.69 times the sequential case), performance degrades thereafter to give a real slow-down, and there is also a curious reduction in performance when we use 2 cores, and a discontinuity at 10 cores. As we increase the core count, we can also clearly see the opposite effect of the former examples: energy usage increases rather than decreases as the number of cores is increased, reaching a maximum of 9535.86J for 1600 details on 55 cores. Investigation of our samples shows that this benchmark had significant levels of garbage collection. This behaviour reveals additional factors that impact energy consumption. We will discuss this further in Section 5.

partree: For *partree* (Figure 5), we have chosen the number of *tree nodes* to range from *600* to *800* with number of *workers per node* to be 350. Although the performance continues to improve as we add physical cores (up to a maximum of 4.47 times speedup for 28 cores for 650 tree nodes), the speedups decrease as further virtual cores are added. Although the energy usage initially decreases as we add cores (from 9285.73J on one core, for 800 tree nodes), beyond about 6 cores, the energy usage plateaus, before increasing beyond 12 cores. The energy usage is particularly unstable when virtual cores are used. As with *ray*, detailed inspection of the samples revealed significant levels of garbage collection. This will also be discussed in Section 5.

Core Affinity: Although we deliberately measured results with and without the use of core affinity to pin tasks to particular cores, using the GHC -qa flag, although we observed that the performance and energy usage was slightly lower and more erratic, the speedups and energy consumption were broadly similar in both cases. Although this initially seems counter-intuitive, since we would expect energy advantages from avoiding cache flushes and context switches, for example, we conclude that there is no overall energy advantage to using the core affinity settings for the benchmarks that we have studied.

4 MODELLING

The energy models that we derive are produced using *glmnet* [7]. This is a library for the statistical language *R*, which implements the multiple regularisation techniques that described above. Using *cross-validation* from *glmnet* we can find the best minimum λ values. for each of our regularisation runs. For *nbody*, we see high R^2 values across all the three models (Table 2), scoring between 80.3% and 81.3%. In a similar way, *sumeuler* has demonstrated stronger R^2 values, both lasso and elastic net showed a 91.91% fit for the model with the best λ values, Ridge was similar with 90.25%. Unlike *nbody*, *sumeuler* seemed to over-fit with certain high values of all the three analyses. *ray* demonstrated one of the best fits across all regularisation analyses, with R^2 of 97.57%, 97.57%, 97.11% for elastic net, lasso and ridge respectively. *partree* and *queens* were more unpredictable with relatively lower R^2 percentages. In particular, *partree* showed relatively weak predictability, with regularisation results for elastic net of 66.67%, lasso of 66.67%, and ridge of 63.99%. For *queens*, the results were for elastic net 74.83%, lasso 74.84%, and ridge: 74.09%. The coefficients of the best fitted models vary as shown in Table 2. However, we observe a pattern that conforms with what the regularisation analysis also revealed: the duration takes precedence in determining the overall power consumption for some of our benchmarks. *sumeuler* $\hat{\beta}$ has a stronger correlation towards *power* in both of the lasso and elastic net models, in ridge however it returns to *duration*. In the case of ray tracer, the coefficients are skewed towards duration across all the 3 models. This relationship is the opposite of our hypothesis, which is *energy* $\propto$ *duration*. *queens* is similar to *sumeuler*, where the generated models have signified *power* in elastic net and lasso, and in ridge *duration* had the higher value. Although *partree* has the lowest R^2 value, it produced high values for *power* in all models.

Table 3 compares our results. The root mean square error or *RMSE* corresponds to the response value scale on Y axis or what is known as the *dependent variable*. Here, for *sumeuler*, *nbody*, and *ray* the values of R^2 and *RMSE* are relatively smaller than those of *queens* and *partree*. This indicates that the fitness of the model matches the R^2 values.

5 GARBAGE COLLECTION AND ENERGY

Further analysis of our samples that show heavy GC usage was correlated with higher energy consumption. For example, Listing 1 shows the raw statistics we have obtained for *partree* running on 28 cores, with 800 tree nodes. It is obvious that the majority of the execution time is devoted to garbage collection. While not being directly comparable, of course, the energy usage of 7541.6J is also significantly higher than for other programs that we ran on similar numbers of cores: *nbody* consumed 1914.49J and *sumeuler* consumed 2141.45J, for example.

```
INIT    time    0.03s  (  0.01s elapsed)
MUT     time   45.81s  (  1.51s elapsed)
GC      time  1848.95s ( 66.23s elapsed)
EXIT    time    0.22s  (  0.18s elapsed)
Total   time  1895.01s ( 67.93s elapsed)

Alloc rate    2,432,513,376 bytes per MUT
    second

Productivity   2.4% of total user, 67.8% of
    total elapsed

gc_alloc_block_sync: 50078149
whitehole_spin: 0
gen[0].sync: 5209
gen[1].sync: 127516
PARENT: Child's exit code is: 0

Total Energy:   7541.6 J
Average Power:  110.918 W
Time:   67.9925 sec
```

Listing 1: Original sample for *partree* on 28 cores

In order to reassess the cause of such a high level of garbage collection, we therefore revisited the *partree* benchmark. We executed the program again with the same core count and problem size. However, we fixed the stack and heap sizes using the following GHC flags:

- **-H:** this flag sets a suggested heap size that is expandable during the life cycle of the program execution. The default is 0 which indicates that memory is allocated as needed based on the value of -A. When -H is set it expands -A to meet the specified heap size.
- **-K:** this flag sets the maximum stack size for individual threads, the default is 8 megabytes.

Using these flags, we repeated the program execution using new settings of 22GB for the heap (-H22G), and 8.1GB for the stack (-K8.1G). Since the base memory usage of the

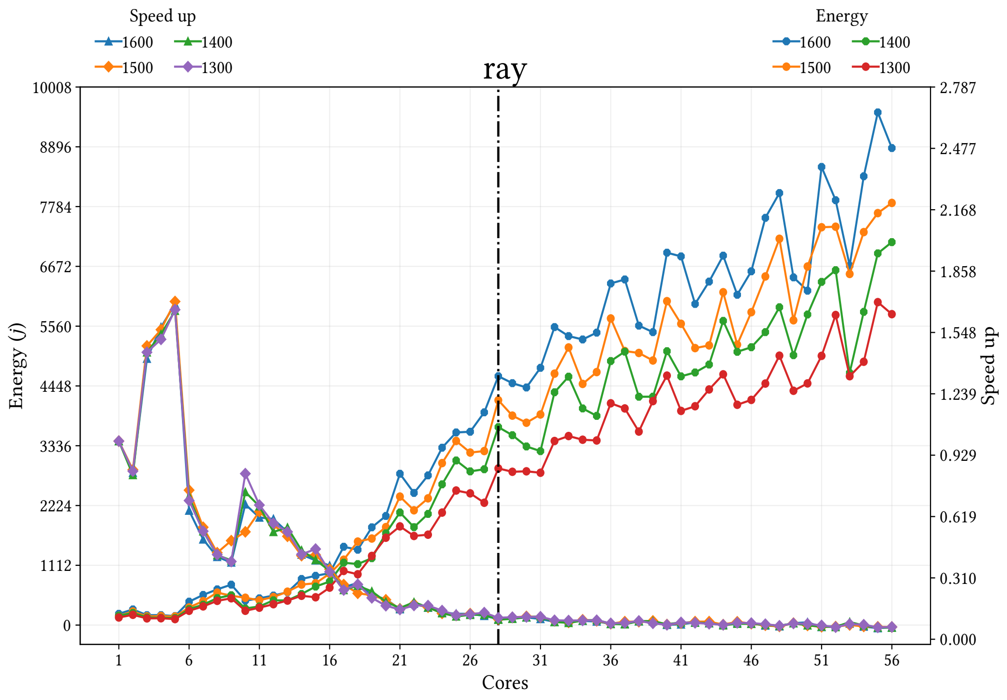

Figure 4: Ray – *Energy vs. Speed up*, detail ranging from 1300 to 1600

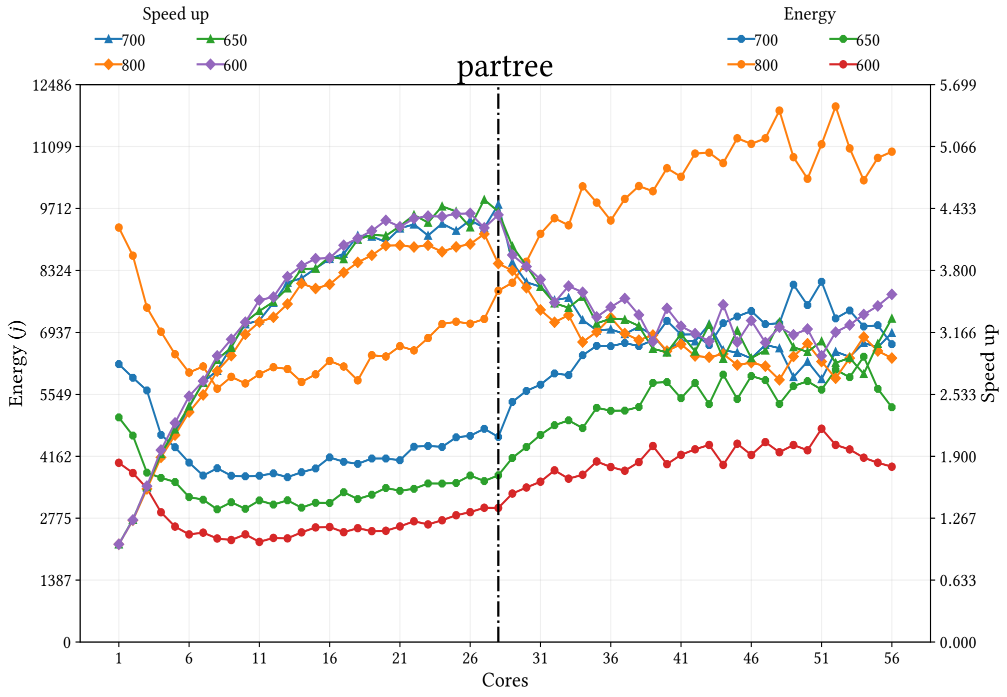

Figure 5: Partree – *Energy vs. Speed up*, tree nodes ranging from 600 to 800, workers per node fixed at 350

Program	Nbody			Sumeuler			Ray			Queens			Partree		
Coefficient	$l1$	$l2$	$Enet$	$l1$	$l2$	$Enet$	$l1$	$l2$	$Enet$	$l1$	$l2$	$Enet$	$l1$	$l2$	$Enet$
$\hat{\beta}_0$ $(Intercept)$	289.9	605.7	313.5	888.2	1771	888	−595	−760	−641	−719	−317	−693	−3173	−1148	−3132
$\hat{\beta}_1$ $(productivity)$	−0.20	−0.16	−0.20	−0.68	−0.43	−0.68	1	0.93	1	−0.76	−0.64	−0.74	−30	−12	−29
$\hat{\beta}_2$ $(duration)$	26.2	23.5	26.1	20.8	18.1	20.8	48.1	44.8	48.1	29.9	27.3	29.8	41	35	41
$\hat{\beta}_3$ $(power)$	10.1	6.1	9.7	26.9	9.6	27	1.4	8.8	2.4	32.2	26.1	31.7	90	60	90

Table 2: $\hat{\beta}$ coefficients for the complete dataset (core affinity, no core affinity, cpu utilisation capped). *Enet*: **elastic net,** *l*1: **LASSO,** *l*2: **ridge**

	Nbody		Sumeuler		Ray		Queens		Partree	
	R^2	$RMSE$	R^2	$RMSE$	R^2	$RMSE$	R^2	$RMSE$	R^2	$RMSE$
Elnet	81.28%	689	91.91%	642	97.57%	386	74.83%	1405	66.67%	1573
$l1$	81.3%	696	91.91%	638	97.57%	387	74.84%	1411	66.67%	1553
$l2$	80.3%	704	90.25%	698	97.11%	421	74.09%	1438	63.99%	1615

Table 3: R^2 and root mean square error values for the complete dataset (core affinity, no core affinity, cpu utilisation capped). *Enet*: **elastic net,** *l*1: **LASSO,** *l*2: **ridge**

test environment does not exceed `2.5 gigabytes`, we are then left with `249.5 gigabytes` of free memory. This was divided as described above. Listing 2 shows the decrease in energy consumption that results from this change, from 7541.6J to 3288.J, a reduction of 56%. Although mutation time has increased to 103.68s from 45.81s, total execution time has decreased to 1034.54s from 1895.01s, since GC time has decreased from 1848.95s to 930.70s. This accounts almost precisely for the improved energy consumption.

```
INIT    time    0.01s  (  0.00s elapsed)
MUT     time  103.68s  (  3.70s elapsed)
GC      time  930.70s  ( 33.32s elapsed)
EXIT    time    0.08s  (  0.08s elapsed)
Total   time  1034.54s ( 37.18s elapsed)

Alloc rate    1,060,460,689 bytes per MUT
    second

Productivity  10.0% of total user, 279.3%
    of total elapsed

gc_alloc_block_sync: 104435303
whitehole_spin: 0
gen[0].sync: 874835
gen[1].sync: 3411590
PARENT: Child's exit code is: 0

Total Energy:   3288 J
Average Power:  87.1017 W
Time:   37.7489 sec
```

Listing 2: Modified stack and heap sample for *partree* on 28 cores

To further verify this, we apply the same approach to *quicksort*: a parallel implementation of the standard Quick Sort sorting algorithm. We sort a dataset of over `50` million randomly generated integers and set the recursion depth parameter to 8. Listing 3, clearly shows that garbage collection takes around 92% percent of total execution time `63.01s`.

```
INIT    time    0.01s  (  0.00s elapsed)
MUT     time  107.15s  (  5.14s elapsed)
GC      time  1762.94s  ( 63.01s elapsed)
EXIT    time    0.06s  (  0.06s elapsed)
Total   time  1870.16s  ( 68.22s elapsed)

Alloc rate    2,809,289,660 bytes per MUT
    second

Productivity   5.7% of total user, 157.2%
    of total elapsed
```

```
gc_alloc_block_sync: 89018579
whitehole_spin: 0
gen[0].sync: 91
gen[1].sync: 3246573
PARENT: Child's exit code is: 0

Total Energy:   6342.34 J
Average Power:  92.886 W
Time:   68.2809 sec
```

Listing 3: Original sample for *quicksort* on 28 cores

Listing 4 shows the results for *quicksort*, using -H50G and -K6.3G. Once again, although mutation time has increased, garbage collection time, and thus total execution time, have been reduced by a more significant amount. In this case, the overall energy consumption has been reduced by a factor of 3.6, simply by changing the default stack and heap sizes.

```
INIT    time    0.01s  (  0.00s elapsed)
MUT     time  275.76s  ( 10.52s elapsed)
GC      time  192.27s  (  7.05s elapsed)
EXIT    time    0.19s  (  0.18s elapsed)
Total   time  468.34s  ( 17.88s elapsed)

Alloc rate    1,182,819,903 bytes per MUT
    second

Productivity  58.9% of total user, 1543.6%
    of total elapsed

gc_alloc_block_sync: 11487701
whitehole_spin: 0
gen[0].sync: 37150
gen[1].sync: 266770
PARENT: Child's exit code is: 0

Total Energy:   1732.22 J
Average Power:  83.1426 W
Time:   20.8343 sec
```

Listing 4: Modified stack and heap sample for *quicksort* on 28 cores

Mutation Time and GC Time versus Energy Usage

Figures 6– 11 compare garbage collection time and mutation time against energy consumption on varying numbers of cores for two examples: *quicksort* and *sudoku* - a program that solves a number of Sudoku puzzles. The first set of graphs show the default stack and heap configurations for each example. Subsequent graphs show the results that we obtain when we use -H20G and -H60G respectively. The stack was set to be 10GB smaller than the heap, i.e. -K10G, -K50G. We can clearly see from these graphs that using the default settings there is a strong correlation between GC time and energy usage, but only a weak correlation between execution time and energy usage. When using -H20G and -60G, however, not only is total energy reduced, but there is a much closer correlation between mutation time and energy usage. This is because the system has achieved a better balance: both garbage collection and mutation contribute to total execution time, and thus to overall energy usage.

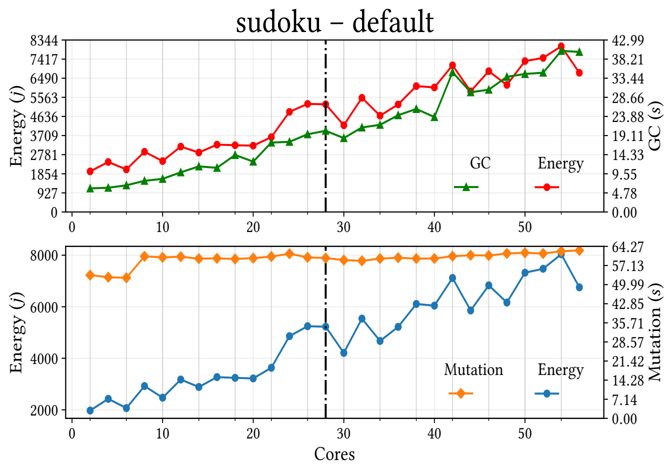

Figure 6: Sudoku with default GHC parameters

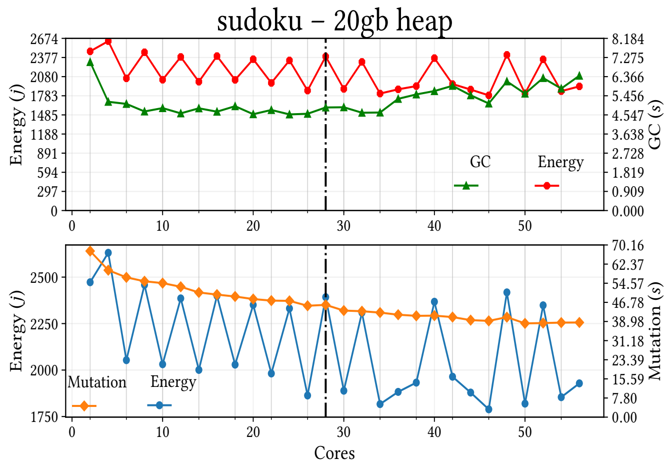

Figure 7: Sudoku using a Heap size of 20GB

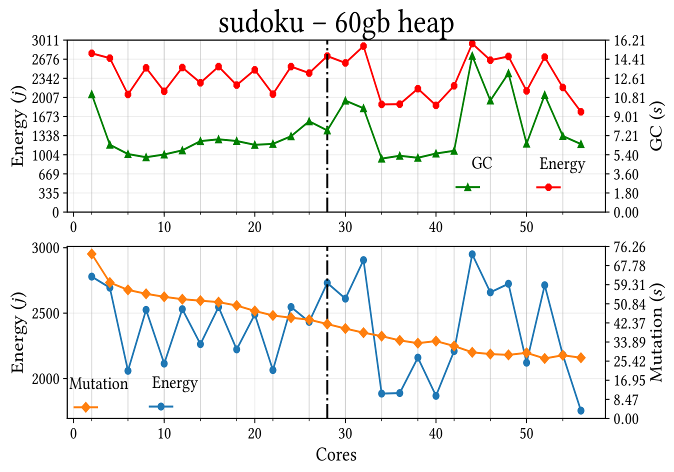

Figure 8: Sudoku using a Heap size of 60GB

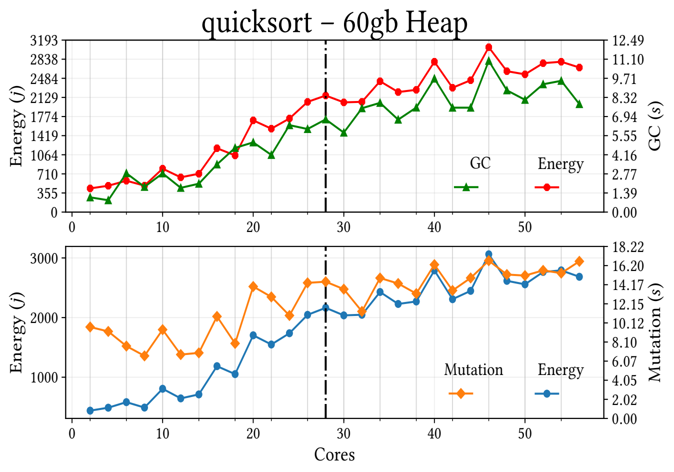

Figure 11: Quicksort using a Heap size of 60GB

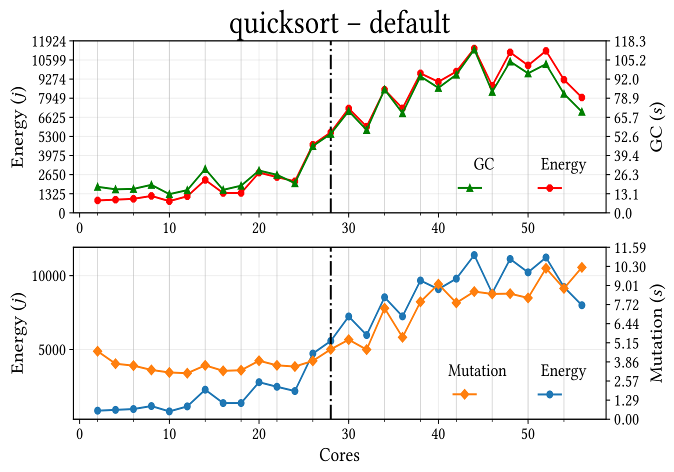

Figure 9: Quicksort with default GHC parameters

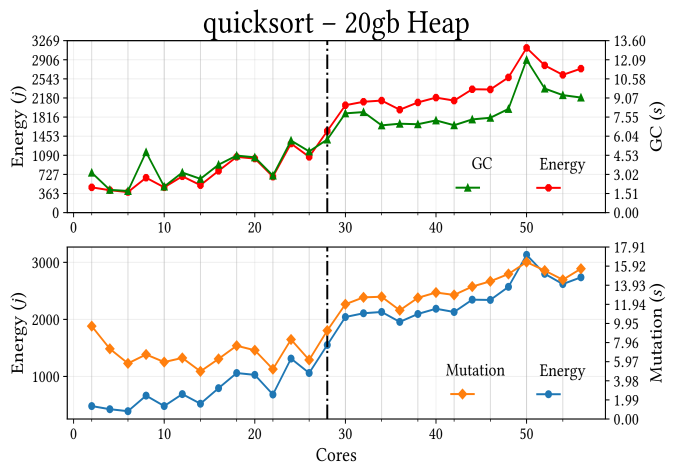

Figure 10: Quicksort using a Heap size of 20GB

6 CONCLUSIONS & FUTURE WORK

This paper has investigated the energy usage for parallel Haskell programs. We have studied the energy usage for a number of representative Haskell benchmarks taken from the standard Nofib suite, and compiled using GHC. For the examples we have studied, we have found that there is a generally strong correlation between energy usage and execution time. We have also found that using more processor cores can improve energy usage. Garbage collection can have a major impact on energy usage. We found that benchmarks that had significant levels of garbage collection could have poor energy usage. Reducing garbage collection costs could be highly beneficial for energy usage (and total execution time) even if this resulted in increased mutation times.

In future work, we will first aim to extend the number of benchmarks that we consider, and to consider larger Haskell program fragments. This will enable us to confirm the results that have been described here. We will also consider different processor architectures, including e.g. ARM and AMD processor designs. This will allow us to confirm whether our results hold generically for all processor designs (as we surmise). This may involve using different measurement technology: at present RAPL only exists on (some) Intel x96 processors. It would also be interesting to compare the results that we have obtained with similar results from the GHC *LLVM* backend. Finally, our ultimate goal is to use the information about energy usage that we have obtained to provide *a-priori* predictions of the energy usage for Haskell programs, and to relate these to execution time. This will involve constructing new models that can relate energy costs to program level source, and that can compose costs from different program components, perhaps through some type-based mechanisms.

REFERENCES

[1] AD Al Zain, Philip W Trinder, Kevin Hammond, Alexander Konovalov, Steve Linton, and Jost Berthold. 2008. Parallelism without pain: orchestrating computational algebra components into a high-performance parallel system. In *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on.* IEEE, 99–112.

[2] Zorana Banković and Pedro López-García. 2015. Energy Efficient Allocation and Scheduling for DVFS-enabled Multicore Environments Using a Multiobjective Evolutionary Algorithm. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO Companion '15).* ACM, New York, NY, USA, 1353–1354. https://doi.org/10.1145/2739482.2764645

[3] Shaiful Alam Chowdhury and Abram Hindle. 2016. GreenOracle: Estimating Software Energy Consumption with Energy Measurement Corpora. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16).* ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/2901739.2901763

[4] Theodoros Dimopoulos. 2017. *CPU Energy Consumption Analysis of Parallel HaskellApplications.* Master's thesis. University of St Andrews.

[5] Kerstin Eder, John P. Gallagher, Pedro Lopez-Garcia, Henk Muller, Zorana Bankovic, Kyriakos Georgiou, Remy Haemmerle, Manuel V. Hermenegildo, Bishoksan Kafle, Steve Kerrison, Maja Kirkeby, Maximiliano Klemen, Xueliang Li, Umer Liqat, Jeremy Morse, Morten Rhiger, and Mads Rosendahl. 2016. ENTRA: Whole-Systems Energy Transparency. (2016). arXiv:1606.04074 http://arxiv.org/abs/1606.04074

[6] Simon Marlow et al. 2017. NoFib: Haskell Benchmark Suite. (nov 2017). https://github.com/ghc/nofib

[7] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2010. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of Statistical Software* 33, 1 (2010), 1–22. http://www.jstatsoft.org/v33/i01/

[8] Kyriakos Georgiou, Steve Kerrison, and Kerstin Eder. 2015. On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs. *CoRR* abs/1510.07095 (2015). arXiv:1510.07095 http://arxiv.org/abs/1510.07095

[9] Intel 2016. *Intel 64 and IA-32 Architectures Software Developer Manual: Vol 3* (3 ed.). Intel.

[10] Paul H Kelly. 1989. *Functional programming for loosely-coupled multiprocessors.* MIT Press.

[11] Steve Kerrison and Kerstin Eder. 2015. EDER_ Energy Model - Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Transactions on Embedded Computing Systems* 14, 3 (2015), 56:1–56:25. https://doi.org/10.1145/2700104

[12] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes. 2016. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 517–528. https://doi.org/10.1109/SANER.2016.85

[13] Ami Marowka. 2017. Energy-Aware Modeling of Scaled Heterogeneous Systems. *Int. J. Parallel Program.* 45, 5 (Oct. 2017), 1026–1045. https://doi.org/10.1007/s10766-016-0453-2

[14] Jeremy Morse, Steve Kerrison, and Kerstin Eder. 2016. On the infeasibility of analysing worst-case dynamic energy. (2016). arXiv:1603.02580 http://arxiv.org/abs/1603.02580

[15] Ryan Newton, Chih-Ping Chen, Simon Marlow, et al. 2011. Intel concurrent collections for haskell. (2011).

[16] James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. 2015. Data dependent energy modelling for worst case energy consumption analysis. *eprint arXiv:1505.03374* (2015). arXiv:1505.03374 http://adsabs.harvard.edu/abs/2015arXiv150503374P

[17] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. 2012. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro* 32, 2 (March 2012), 20–27. https://doi.org/10.1109/MM.2012.12

[18] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), 267–288.

[19] Vincent M. Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. 2012. Measuring Energy and Power with PAPI. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops (ICPPW '12).* IEEE Computer Society, Washington, DC, USA, 262–268. https://doi.org/10.1109/ICPPW.2012.39

[20] Hui Zou and Trevor Hastie. 2005. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67, 2 (2005), 301–320.