A Study on the Influence of Software and Hardware Features on Program Energy *

Ajitha Rajan
School of Informatics
University of Edinburgh, UK
arajan@staffmail.ed.ac.uk

Adel Noureddine
School of Architecture, Computing and Engineering
University of East London, UK
a.noureddine@uel.ac.uk

Panagiotis Stratis
School of Informatics
University of Edinburgh, UK
s1329012@sms.ed.ac.uk

ABSTRACT

Software energy consumption has emerged as a growing concern in recent years. Managing the energy consumed by a software is, however, a difficult challenge due to the large number of factors affecting it – namely, features of the processor, memory, cache, and other hardware components, characteristics of the program and the workload running, OS routines, compiler optimisations, among others. In this paper we study the relevance of numerous architectural and program features (static and dynamic) to the energy consumed by software. The motivation behind the study is to gain an understanding of the features affecting software energy and to provide recommendations on features to optimise for energy efficiency.

In our study we used 58 subject desktop programs, each with their own workload, and from different application domains. We collected over 100 hardware and software metrics, statically and dynamically, using existing tools for program analysis, instrumentation and run time monitoring. We then performed statistical feature selection to extract the features relevant to energy consumption. We discuss potential optimisations for the selected features. We also examine whether the energy-relevant features are different from those known to affect software performance. The features commonly selected in our experiments were execution time, cache accesses, memory instructions, context switches, CPU migrations, and program length (Halstead metric). All of these features are known to affect software performance, in terms of running time, power consumed and latency.

1. INTRODUCTION

With rising energy costs and the need for environmental improvements, one of the key challenges faced by the ICT industry now and in the coming years will be reducing its carbon footprint and energy consumption while still delivering the performance and quality that users expect. Energy consumption is not only an issue for the mobile embedded systems domain, but also for servers, desktops and laptops which constitute more than a third of ICT energy consumption [38].

*This research is funded by EPSRC First Grant EP/L024624/1.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Measuring and managing the energy consumed by a software is, however, a difficult challenge due to the large number of factors affecting energy consumed – namely, characteristics of the processor (clock frequency, voltage, pipeline depth), memory (instruction and data cache reads, writes and misses), communication network, GPU, sensors for mobile devices, compiler and optimisations enabled, features of the program and workload running (number of instructions, input data, number of branches, number of operators and operands, algorithmic complexity, multi-threading) among others. Before we embark on proposing optimisations for software energy consumption, we believe it is imperative to first *identify and understand* the factors that contribute to energy consumption. Our ultimate goal is to use this understanding to guide techniques in choosing factors to optimise for energy. We would also like to ascertain if the *factors affecting software energy are the **same** as those affecting software performance*, in which case there is no need to propose new optimisation techniques specifically targeting energy. Existing optimisation techniques targeting performance may also be adequate for energy.

Existing literature, particularly in the embedded systems domain, has estimated energy consumption based on several architectural features [44, 47, 42]. Recent literature in the mobile devices domain estimated energy consumption using features related to the application and several other integrated hardware components [26, 21]. The goal in all of these studies has been to profile or estimate energy based on architectural and, in some cases, application features [43, 47]. Our empirical study does not aim to estimate energy but instead tries to understand the statistical relevance of features, both architectural and program specific when taken together, to energy consumed. We then select the relevant features as targets for energy optimization techniques. The selection technique we use takes *interaction between features* into account and *eliminates redundancy* in the selection.

Our empirical study has the following salient features

- We collect data on energy consumption and over a hundred features, static and dynamic, related to the architecture and the program.
- Our data is collected using 58 subject programs, each with their own workload, from different domains. Many of these programs are part of industry standard benchmarks.
- We use a statistical machine learning technique called ***feature selection*** to select the features relevant to energy consumption.
- We analyse the features selected, discuss their use as potential targets for optimisation and determine if they are different from optimisations proposed for software performance.

The feature selection technique we use is the LASSO technique [48]. To validate the usefulness of the features selected by Lasso, we compared accuracy of energy predictions using

only the *selected* features against energy predictions from a linear regression model using *all* the features. We found the features selected in our study were the same as known features in existing optimisations for execution time.

Our paper is organised as follows. Section 2 discusses related work. In Section 3 we describe the experimental setup to gather hardware and software features over different programs. We detail and discuss the results of our experiments in Section 4. We provide more insights into the features selected by our experiments, and the effect of existing optimisations in Section 5. Threats to validity in our experiment and conclusions are presented in Sections 6 and 7.

2. RELATED WORK

Monitoring and modeling energy consumption of applications has been and still is an active research area. As stated earlier, the goal in our work is not to model or estimate energy but to understand the relevance of many hardware and software features, when taken together, to energy to guide optimisations. Nevertheless, we present relevant literature on energy modeling since that is the most closely related body of work.

Previous work in the embedded systems domain has focussed extensively on estimating power consumed by the processor [49, 44, 47]. The estimations rely on a complete characterisation of the underlying instruction set architecture and inter-instruction effects to estimate power consumption. Cycle-accurate simulators developed to estimate power can simulate the actions of a processor at an architecture-level [9, 34, 52]. They identify the architectural blocks that are active in each clock cycle and record the input operands seen by each architecural block. Power consumption is then estimated for each architectural block as a function of the input data values. Both instruction-level characterisation and cycle-accurate simulator estimation methods are slow, compute-intensive, and infeasible for applications outside the embedded systems domain.

Several tools have been developed to estimate energy using hardware component characteristics. pTop [16] estimates energy consumption using a model based on the characteristics of hardware components (such as CPU, hard disk, ethernet). PowerAPI and BitWatts [11] use hardware performance counters to learn the energy model of hardware components. They build the model by benchmarking the computer and collecting power measures using a power meter. Jalen [32] is built on top of PowerAPI's energy models and maps energy estimations to an application's source code. In [43], Seo et al. estimate energy consumption of distributed Java applications by benchmarking Java byte code and native methods, measuring their energy consumption with a multimeter. Collected results are then used to construct an energy model for Java components and applications.

For mobile devices, integrated hardware components, such as the GPS, screen, or the 3G communication interface, introduce additional complexity into the energy models. Recent literature [36, 26, 21] has focussed on energy models for mobile devices. Eprof [36] maps back energy consumption of hardware components to applications by adopting a *last-trigger policy*. For example, it associates *tail energy drains* of a hardware component [1] to the last application that triggered the usage of that component. Hao et al. [21] and Li et al. [26] propose an approach called eLens to estimate energy usage at the source code level. They estimate the energy consumption of mobile applications using program analysis and linear regression analysis. Applications are first instrumented to determine paths traversed during execution. This along with instruction energy costs are used to predict source line energy.

[1]Tail energy happens when a component, such as the GPS, continues to be active and consumes energy even after the application that triggered its usage has ended.

3. EXPERIMENT DESCRIPTION

The experiment we conducted is comprised of two phases,
Phase 1: Data Collection - We ran a series of experiments on programs from three different benchmark suites to measure energy consumed and features related to the hardware and program. The benchmarks, features measured, and tools used for measurements are explained in the following sections.
Phase 2: Prediction Models - Using the measurements in the data collection phase, we trained our model to select features[2] that are relevant to the energy consumed. The statistical feature selection technique we use is Lasso regression and is described in Section 3.4. In order to validate the accuracy and usefulness of the selected features, we also trained and constructed a Linear regression model using *all* the features. We then compared the energy prediction error of Lasso regression (with the selected features) against that of Linear regression using all the features.
Phases 1 and 2 are described in the following sections.

We ran our experiments using a desktop computer powered by an Intel Core 2 Duo E8400 processor at 3 GHz, 2 GB of DDR2 memory at 800 MHz, 128 KB of L1 cache and 6 MB of L2 cache. The machine was running Ubuntu Server 14.04 with Linux kernel 3.16.0.33.

Phase 1 - Data Collection

3.1 Benchmarks

The programs in our experiments are from a range of application domains such as image processing, biomolecular simulation, networking, automotive, fluid dynamics, operating system, signal processing among others. The programs perform diverse tasks varying in the type and intensity of computation, file I/O, amount and frequency of memory operations, and several other features. For test cases and benchmarks that run in just a few milliseconds, we run them in a loop (ranging from 1 to 10,000 cycles) to get execution times that are large enough for measurement. Collected data is then averaged for a single program run and compared with the other programs. The programs and benchmark families used in our experiments are as follows.

3.1.1 SIR

We used 9 programs from the SIR (*Software-artifact Infrastructure Repository*) repository [15]. Programs include GNU projects (gzip 1.0.7 and sed 1.17), two lexical analysers (printtokens and printtokens2), two priority schedulers (schedule and schedule 2), a pattern matching program (replace), a statistics program (totinfo), and an aircraft collision avoidance system (tcas). Test cases were either provided, or generated using the Make-test-script program and test plans described in universe files. The test cases exercise the parameters and behaviour of the programs as described in [15]. We run our experiments and collected metrics for test cases over each SIR program. We performed two different analysis on the SIR programs, (1) For each SIR program, we used 300 randomly selected test runs to collect observations. We performed feature selection over these observations for *each* SIR program, (2) We randomly selected 30 test runs for each SIR program, resulting in a total of *270 observations* over all 9 SIR programs. We then performed feature selection across observations over *all* SIR programs. We limited the test runs in order for the total number of

[2]Features in our estimation model refers to the hardware and program related metrics

observations per benchmark family to be comparable. It is worth noting that each *observation* in our context is a record of the energy consumed along with measurements of the different hardware and program features for that test run.

3.1.2 Parboil Benchmark

We used all 11 programs from the Parboil benchmark [46], which is an open source benchmark suite with benchmarks collected from throughput computing application researchers in many different scientific and commercial fields including image processing, biomolecular simulation, fluid dynamics, and astronomy. Applications in the benchmarks perform a variety of computations including ray tracing, finite-difference time-domain simulation, magnetic resonance imaging, 3-D stencil operation. Parboil provides applications implemented as both serial C++ and OpenMP, along with workloads specific to the programming model. We run both the base and parallel OpenMP versions of these programs. We collected *51 observations* over all the programs in Parboil.

3.1.3 EEMBC Benchmark

Embedded Microprocessor Benchmark Consortium (EEMBC) [37] provides a diverse suite of processor benchmarks organised into categories that span numerous real-world applications, namely automotive, digital media, networking, office automation and signal processing, among others. There are a total of 38 programs – 17 from automotive domain, 6 from consumer, 4 in networking, 5 in office, and 6 from telecom. We used the workload in the benchmarks to generate a total of *304 observations* over all 38 programs.

The features measured and tools used are explained in the next section.

3.2 Measurements

For every run of every program, we used existing run time monitoring and program analysis tools to measure,

Hardware performance features reflecting CPU, cache and memory performance. We measure these features since they are known to impact power and energy [9, 13, 16, 18].

Dynamic program features relating to types of program instructions generated, memory operations performed, and register utilisation. Different types of instructions are associated with different base energy costs [49]. Inter-instruction effects, and operands also affect the energy consumed. Memory operations are also associated with high energy costs. Finally, effective register utilisation during machine code translation is shown to help improve energy efficiency [42].

Static program features such as number of basic blocks, number of loops, branches, global variables, cyclomatic complexity. We measure these high-level program features since they have been shown to be useful in estimating energy consumed by software [47], and are the primary guide in creating and measuring adequacy of test runs (over which execution time and energy are measured) [45, 22, 41].

In addition to the above features, we also measured the execution time and energy consumed by each of the program runs.

3.2.1 Hardware Features

We use Linux's Perf tool [2] to collect 37 hardware features. Perf uses special-purpose CPU registers to count the number of events, such as CPU misses or mispredicted branches. Perf runs in the user space and is part of the kernel. It has a low overhead compared to instrumenting profilers [51]. The hardware features collected with Perf are as follows:

Performance features. We collected 8 features that provide CPU performance information for the program, such as the number of instructions executed between each performance sampling, total CPU cycles, total bus cycles, the total time spent by the application or it's tasks on the CPU (CPU clock and task clock). Additional metrics include the number of context switches and the number of CPU migrations. Number of context switches corresponds to the number of times the CPU restored the context (*e.g.*, state) of a process or thread. Context switches can invalidate some TLB[3] entries, since the virtual-to-physical address mapping is different. This results in misses in the TLB for a memory reference. The number of CPU migrations counts the number of times the executing process has migrated to a new CPU or a new CPU core.

Branch Instruction features. We collected 4 features that count the number of executed branch instructions, mispredicted branch instructions, number of branch instructions with a load access, and the number of those loads that resulted in a cache miss.

Cache Related features. Cache effects are known to have a significant impact on performance [18]. We collected 19 features that record the number of loads and stores (in addition to load misses and store misses) for Level 1 Cache (L1), Last Level Cache (LLC) and Translation Lookaside Buffer (TLB). For L1 cache and TLB buffer, we collect separate metrics for data and instruction cache.

Page Fault features. We collected 6 features including, total page faults, minor and major page faults, alignment and emulation faults. When a page fault occurs, the thread that experienced the page fault is put into a Wait state while the operating system finds the specific page on disk and restores it to physical memory. Page faults can cost millions of CPU clock cyles because of disk access and, as a result, severely affect performance.

In addition to the above hardware features, we also measured execution time of the program using the "time" Linux command.

3.2.2 Dynamic Program Features

We use the Pin tool [28] (version 2.14) to collect 42 dynamic program features. Pin is a dynamic binary instrumentation framework where instrumentation is performed at run-time on the compiled binary files. We built instrumentation tools in this framework to collect features in the following categories:

Program Instructions. We used 4 features to count the number of executed instructions, the number and size of basic blocks executed, and the number of threads.

Memory Instructions. We used 6 features counting the number of times instructions were read or written from/to memory, and the number of malloc/realloc/calloc and free calls.

Processor Registers. We used 32 features to count the number of read and write accesses to the different processor registers.

3.2.3 Static Program Features

We collect 25 static program features using Frama-C's [5] metrics plugin [1] (Sodium release). The Frama-C platform helps analyse the source code for C programs. We collect the following metrics:

Halstead Complexity measures [20]. We have 13 features that count the total and distinct number of operators and

[3]A Translation Lookaside Buffer (TLB) cache stores recent translations of virtual addresses to physical addresses, both for instructions and data, to enable faster retrieval of information.

operands, the program length (which describes the size of the abstracted program by removing everything except operators and operands), the program volume (which models the number of bits required to store the abstracted program), and program level (which defines the ratio between the program volume and the volume of its most compacted implementation).

McCabe's Cyclomatic complexity [29]. We use this feature to provide an indication of the complexity of a program by measuring the number of linearly independent paths in a program.

Other static progam features. We have 11 features representing the number of source lines of code, number of loops, Ifs, Gotos, functions declared, exit points (*e.g.*, return statements), the number of decision points, global variables and the number of pointer dereferences.

3.2.4 Energy Metric

We measure the energy consumed by the computer for each program run using a Watts Up? .Net power meter [4]. The power meter is connected to another computer that collects energy observations at runtime. We also measured and continuously monitored the idle energy of the computer. We used the server edition of Ubuntu as our operating system with no additional services running. Energy consumption of a program run is the difference between the measured energy and idle energy. It is also worth noting that our energy measurements do not include the energy consumed by the monitor since the monitor we use has its own independent power supply.

In summary, we observed a total of 106 features, both hardware and program specific, that we believed will be relevant to the energy consumed by the program. In the next sections, we describe the regression models and feature selection technique we use in our experiments.

Phase 2 - Prediction Models

3.3 Linear Regression

One of the most widely used regression models is linear regression. This asserts that the response we wish to predict, y ("dependent variable") is a linear function of the input vector x ("independent variables" or features). In this paper, x is a vector of hardware and software features measured over the program that is used to compute y, the energy consumed by the program. Linear regression is expressed by Murphy et al. [31] as

$$y(x) \;=\; w^T x \;+\; \epsilon \;=\; \sum_{j=1}^{D} w_j x_j \;+\; \epsilon \tag{1}$$

where $w^T x$ represents the scalar product between the input vector x and model's weight vector w, also referred to as vector of coefficients for x. D is the length of input vector x. ϵ is the residual error between our linear predictions and the true response. Now given this regression model and a training data with N number of observations, we need to find an optimal setting of the coefficients vector, w, for best predicting y from x. This optimisation is typically done by minimizing the residual sum of squares (RSS) or sum of squared errors (SSE) as defined by,

$$RSS(w) \doteq \sum_{i=1}^{N} (y_i - w^T x_i)^2 \tag{2}$$

This method is referred to as least squares. The values of w_i computed for minimal RSS represent the regression coefficients to be used in the model. We used the WEKA (Waikato Environment for Knowledge Analysis) tool [19] to build the Linear regression models.

3.4 Feature Selection using Lasso

We use a machine learning technique, called feature selection, to select a subset of statistically relevant features to improve the overall performance of the learning model by preventing overfitting. In the context of this paper, we apply feature selection to (1) understand the relationship between the features and energy consumed, and (2) to select the set of relevant features (among all hardware and program features) that can provide insight on where to target energy optimisation efforts. Note that we are interested in the effect of features taken together, rather than individually, on the energy consumed. Feature selection also eliminates the need for measuring features that are not relevant, or are redundant to the energy consumed. We use an L_1 regularization method called **LASSO** to apply feature selection in the *linear regression model* [31], [48]. Lasso is an embedded algorithm performing feature selection as part of the model construction process. Regularization methods introduce a penalty term for the size of the weights that is tuned empirically for minimizing the RSS. Lasso regression is often preferred because it produces sparse models effectively, i.e., the regression coefficients for most *irrelevant or redundant features* are shrunk to zero, thereby achieving feature selection [27]. *Interactions among features* are also taken into account by Lasso. We achieve Lasso feature selection in our experiments using the *'glmnet'* package [17] in R [40], a software environment for statistical computing.

3.5 Validation

We use cross validation, in particular, **Leave-One Out Cross Validation (LOOCV)** to evaluate the performance of the different prediction models. Cross validation technique works as follows; suppose the original dataset has N observations, then, for each $i \in \{1, ..., N\}$, we train the regression model on all except the ith observation, test the model on the ith observation and compute the error. This process is repeated in a round-robin fashion and the error averaged over all i, i.e, N validation runs. In our case, every test case run from every program results in an observation of vector x of features and response y of the energy consumed. For each regression model, we compute, using LOOCV, the Root Mean Squared Error,

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y_i})^2} \tag{3}$$

which is the square root of the mean of the squared errors over n different predictions. We then normalize the RMSE to the range of the observed response using the minimum and maximum actual response, i.e., Normalised RMSE

$$NRMSE = \frac{RMSE}{y_{max} - y_{min}} \tag{4}$$

NRMSE is a popular error metric to compare numerical prediction errors between models. Compared to mean absolute error, RMSE amplifies and severely punishes large errors, such as those from outliers.

4. EXPERIMENTAL RESULTS AND ANALYSIS

We collected observations of energy and features for programs with their respective workloads in the three benchmark families as described in Section 3. The results are organised and analysed as follows,

1. **Per SIR program**. We conducted analysis separately for each of the 9 SIR programs.

Features selected by LASSO		
Gzip	Printtokens	Printtokens2
execution time cpu.clock task.clock context.switches cpu.migrations LLC.load.misses LLC.store.misses free - - -	execution time cache.misses cpu.clock task.clock page.faults context.switches minor.faults LLC.load.misses LLC.store.misses malloc free	execution time cache.misses task.clock page.faults context.switches cpu.migrations minor.faults LLC.load.misses LLC.store.misses free -
Replace	Schedule	Schedule2
execution time cache.misses cpu.clock task.clock context.switches cpu.migrations LLC.load.misses LLC.store.misses -	execution time cpu.clock page.faults context.switches cpu.migrations LLC.load.misses LLC.store.misses malloc free	execution time cpu.clock context.switches cpu.migrations LLC.load.misses LLC.store.misses iTLB.load.misses malloc free
Sed	Tcas	Totinfo
execution time cpu.clock task.clock page.faults context.switches cpu.migrations major.faults LLC.load.misses LLC.store.misses malloc realloc free	execution time cache.misses cpu.clock task.clock page.faults context.switches cpu.migrations minor.faults LLC.load.misses LLC.store.misses read.access.memory write.access.memory	execution time cpu.clock context.switches cpu.migrations major.faults LLC.load.misses malloc free - - - -

Table 1: Features selected by LASSO for individual SIR programs (when applying Lasso using all features)

2. **Per benchmark family**. Within each benchmark family, we analysed the results from runs of all programs.

Each analysis was carried out as follows: We used the Lasso learning technique to select a subset of features from all the hardware and program specific features observed. We assessed the usefulness of the selected features by comparing the NRMSE for energy prediction using Lasso against that of Linear regression with all features. We list the relevant features selected by Lasso and their coefficients. The coefficient values of the different features cannot be compared directly since the magnitudes and units of the measurements are not comparable. For instance, execution time is measured in milliseconds, while a Halstead metric gives the number of distinct operators and operands.

We did not conduct per program analysis for EEMBC and Parboil benchmarks since the number of test runs available per program was significantly smaller than the number of features measured. Predictions for such programs in our high-dimensional feature space will be highly inaccurate due to the curse of dimensionality [6]. As a result, we only conducted per program analysis for SIR programs that had greater number of test runs than features measured.

4.1 Individual SIR Programs

For each of the SIR programs, we ran 300 randomly selected test cases. The NRMSE results for the individual SIR programs are presented in Figure 1. Lasso regression performs well with a lower NRMSE across all SIR programs. Linear regression has a slightly higher NRMSE (less than 0.2%).

Table 1 lists for each of the 9 SIR programs, the features selected by LASSO for energy estimation. It is important to note that the Lasso feature selection technique will *suppress* features that are relevant to energy but redundant with respect to another feature selected.

Execution time is a feature selected in all SIR programs. This was to be expected since execution time has a high impact on energy consumption, and aligns with observations in previous investigations [12, 36, 24].

For all the SIR programs, CPU migrations and context switches are among the features selected. Both these features are known to cause overheads in terms of running time [25, 14]. CPU clock and/or task clock are also selected since the time spent in the CPU by the program (CPU clock) and its threads[4] (task clock) is high.

Features related to LLC cache and memory instructions in the source code (malloc/calloc/realloc and free) also appear in Table 1. This is because the programs frequently read and write data resulting in frequent accesses to the cache. Some of these cache accesses result in cache misses that cost extra clock cycles. Static software features are not selected for any of the SIR programs since for runs within each program, these features remain constant. Lasso shrinks the weight of these features to zero since they do not exhibit any deviation between observations.

4.2 Benchmark Families

We now discuss results from runs of all programs within each benchmark family.

4.2.1 SIR Benchmarks

Similar to our observation in individual SIR programs, NRMSE numbers are low across both prediction models (see Table 2), 0.00097 for linear, and Lasso is lower with 0.00082. The features selected with Lasso are marginally more effective in predicting energy than all the features in the linear model. Table 3 shows the features selected by Lasso.

	Linear	LASSO
SIR	0.000971891	0.000827693
EEMBC	0.004287477	0.005514576
Parboil	0.051841246	0.074844556

Table 2: Normalized Root mean square error (NRMSE) for all benchmarks

Features selected	Absolute Weight
execution.time	66.70251726
cache.misses	0.000154405
cpu.migrations	0.234642543
realloc	0.001386825
calloc	0.001374207
Program.level	0.138383951
Function	0.000190104

Table 3: Features selected by LASSO for all SIR benchmarks

Most of the features present in Table 3, are also present in the selection for individual SIR programs (see Table 1) and the reasons for the selection are similar. Table 3 also

[4]This refers to threads created by the OS for multitasking, rather than user created threads in the program.

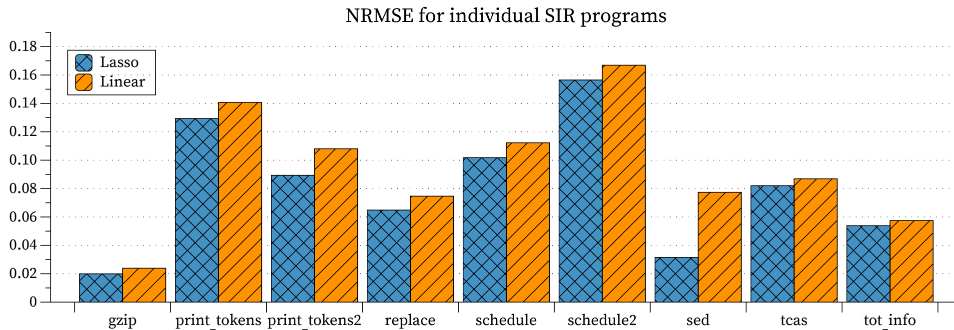

Figure 1: Normalized Root mean square error (NRMSE) for SIR programs

shows the coefficient values for the selected features. The coefficient value for execution time is 10^2 to 10^5 times larger than coefficient values for other features. Although, coefficient values in such models cannot be directly compared due to the different units of measurements, it provides a hint on the importance of execution time for energy prediction in the SIR benchmarks. To assess the importance of execution time for energy prediction, we compared LASSO prediction errors with and without execution time in the feature set. Table 4 lists the NRMSE for LASSO with "All" features,"All - Time", "Only Software", and "Only Hardware". As can be seen, NRMSE increases by 3 times when execution time is removed from the observations. *execution.time* is, thus, a key consideration, and cannot be approximated by any set of hardware or software features for SIR benchmarks.

Table 4 also shows that prediction accuracy is approximately the same for "All - Time", "Only Software" and "Only Hardware" with "Only Hardware" being marginally better. For SIR programs, measuring either only the program characteristics or only the hardware features, along with execution time would suffice to predict energy.

A feature that is different from what we saw for individual SIR programs is the size of the program indicated by the Halstead metric, *Program.level*. This implies that the number of distinct operators and operands in the source code for the SIR programs has an effect on energy. As mentioned earlier, we did not see this static program metric for individual programs since it does not change across runs of the same program. *cpu.clock* and/or *task.clock* are selected for all individual SIR programs, but is not seen in the LASSO selection across all programs. The software metrics, *Program.level*, *Function*, and *calloc* have effectively replaced *cpu.clock* and *task.clock*. This is confirmed again in Table 4 when we perform Lasso selection with *Only Software* metrics which has the same prediction accuracy as Lasso using $All - Time$.

4.2.2 EEMBC Benchmarks

For the 38 programs in the EEMBC benchmark,the Linear regression model performs better with an NRMSE of 0.0042 compared to the Lasso model with an NRMSE of 0.0055 (see Table 2). The prediction errors are, however, comparable and as a result, the features selected can be considered further for optimisations.

Table 5 shows the features selected by Lasso. As with the SIR benchmarks, execution time is selected and the coefficient value (without normalising) is larger than the other features. However, execution time is not as significant for energy prediction as in the case of SIR benchmarks. This is evident from Table 6 where the prediction error of Lasso using *All* features is comparable to Lasso using $All - Time$ features. *Only Software* and *Only Hardware* features also have comparable accuracy in energy prediction. This implies that targeting optimisations at one of these sets of features will be effective in achieving energy efficiency.

Features selected	Absolute Weight
execution.time	4.834222931
cpu.clock	0.004460450
task.clock	0.004313552
context.switches	0.040315249
cpu.migrations	1.755253195
branch.misses	6.10E-08
bus.cycles	1.38E-08

Table 5: Features selected by LASSO for all EEMBC benchmarks

cpu.clock and *task.clock* are among the other features selected. Contrary to the execution time - which includes wait time, these two clock features monitor time spent using the CPU. The CPU intensive nature of some of the EEMBC benchmarks drives these observations. Examples of such CPU intensive benchmarks are the image processing, and automotive applications. Branch misses is selected and this may be due to the extensive use of loops in all the EEMBC programs. We believe the feature, *bus.cycles*, was selected due to frequent communications between the CPU and the main memory. Programs in the telecommunications domain, Encoder, Decoder, and Bit allocation, make heavy use of such communications.

4.2.3 Parboil Benchmarks

For the Parboil benchmarks, Linear regression model does better with an NRMSE of 0.0518, as seen in Table 2. Lasso has a slightly higher error of 0.07484. Among the 3 benchmark families seen thus far, NRMSE is highest for Parboil across prediction models. We believe there are two reasons for the high NRMSE values for Parboil, (1) The number of observations available for Parboil is fewer than the other two benchmarks (51 for Parboil versus 304 for EEMBC and 270 for SIR), resulting in the dimensional feature space being much bigger. As a result, the effect of the curse of dimensionality is more significant in Parboil. (2) Parboil benchmarks have both the serial and parallel versions. *Number of threads* is the only feature in our measurements that reflects the parallel nature of benchmarks. It may be the case that this single feature is inadequate in capturing the effects of parallelism on energy. Table 7 shows the features selected by Lasso and the absolute weights of the coefficients.

The coefficient value of execution time is significantly larger than that of the other selected features. As can be seen in Table 8, prediction error almost doubles when execution time is not included ($All - Time$) in the observations, reflecting its importance for energy estimation

	All	All - Time	Only Software	Only Hardware
NRMSE	0.000827685	0.003193454	0.003193454	0.003183806
Selected features	execution.time, cache.misses, cpu.migrations, realloc, calloc, Program.level, Function	cpu.migrations, Program.level, cpu.clock, context.switches, task.clock, calloc, LLC.load.misses, Exit.point	Program.level, calloc, Exit.point, Free	cpu.migrations, context.switches, cpu.clock, task.clock, LLC.load.misses, minor.faults

Table 4: NMRSE and feature selected by LASSO for SIR benchmarks

	All	All - Time	Only Software	Only Hardware
NRMSE	0.005921402	0.005999306	0.006079861	0.006016273
Selected features	execution.time, cpu.clock, task.clock, context.switches, cpu.migrations, branch.misses, bus.cycles	cpu.migrations, context.switches, cpu.clock, task.clock	Sloc, basic.blocks	cpu.migrations, context.switches, cpu.clock, task.clock, LLC.load.misses

Table 6: NMRSE and feature selected by LASSO for EEMBC benchmarks

Features selected	Absolute Weight
execution.time	58.93799739
cpu.clock	0.001014987
task.clock	0.001215820
L1.dcache.prefetches	6.34E-07
dTLB.load.misses	8.24E-07

Table 7: Features selected by LASSO for all Parboil benchmarks

over Parboil programs. It is also worth noting that measuring *Only Hardware* features outperforms measuring *Only Software* features. We can safely say that energy consumed by Parboil programs are characterised better by hardware features than software. Many of the Parboil programs are CPU intensive, resulting in *cpu.clock* and *task.clock* being selected. The programs use large input data sets. Frequent accesses to these large data sets without temporal and spatial locality result in data cache misses being an important feature for energy estimation.

5. DISCUSSION

From our experiments, we find that the overall accuracy of the Lasso feature selection technique is comparable to the Linear regression model with all the features. The Lasso technique did not always outperform the Linear model and there was no consistent best model across all experiments. Given the high dimensional feature space, we believe comparable prediction accuracy using the Lasso model is a good start and the selected features can be used as a basis for discussion on potential optimisations for energy. It is worth noting that feature selection using Lasso takes correlation between measured features into account and eliminates redundancy. For instance, features like basic blocks and global variables have high coefficient values in the linear regression model but are not selected by Lasso since they are redundant with respect to execution time. When correlated features are selected, it implies that the features together better predict energy than any one of them (they are not redundant with respect to each other or a combination of other selected features).

We found that some hardware and program features were repeatedly selected across all our experiments. We list these features (or sets of features) and discuss their potential as energy optimisation targets. We also go on to discuss if the selected features and optimisations are any different from those already known for software performance.

Execution Time: Execution time has often been used as a representative for energy [12, 36, 24]. Lasso feature selection over all benchmarks resulted in coefficient values for execution time being 10 to 100 times larger than coefficient values for other features. Although, coefficient values in such models cannot be directly compared due to the different units of measurements, it provides a hint on the importance of execution time for energy. It is clearly valuable to target execution time for energy optimisation. However, it is important to bear in mind that energy is dependent on both execution time and power, and reducing one while increasing the other will offset the gains achieved.

Cache and Memory instructions: Our experiments consistently selected cache (L1, LLC, and TLB) features, and dynamic memory instruction features. Missed cache hits, on both the internal and external cache, as well as page faults slow the performance of a program [49, 50] by significantly increasing memory latency and, therefore, execution time. Note, however, that the cache features being selected by Lasso implies they are *not redundant* with respect to execution time. The selection implies that it is potentially beneficial to optimise cache features *in addition* to optimising execution time. For instance, if we have a memory intensive process whose performance is limited by the available memory bandwidth(cache and memory access speeds are slower than processor speeds), we will gain limited execution time gains from increasing processor clock frequency.The increased clock frequency will result in increased power consumption that may offset the limited execution time gains. Thus, it may be beneficial to move memory intensive processes to lower frequency cores, while executing less memory intensive processes on higher frequency cores.

To reduce cache misses and memory latency, one can place related data close in memory and use algorithms and data structures that exploit the principle of locality. For instance, elements of std::vector in C++ are stored in contiguous memory and accessing them is more cache-friendly since it exploits spatial locality, than accessing elements in std::list that does not have this feature.

Compiler researchers have also proposed the use of reuse distance as a metric to approximate cache misses [8, 39].

	All	All - Time	Only Software	Only Hardware
NRMSE	0.074839977	0.11318387	0.277245127	0.11318387
Features	execution.time, task.clock, cpu.clock, dTLB.load.misses, L1.dcache.pref	cpu.migrations, task.clock, cpu.clock, dTLB.store.misses, dTLB.load.misses, L1.dcache.pref, LLC.load.misses, branch.misses	Global.variables, Goto, Function.call, rdx.written, Pointer.dereferencing, rcx.written, rdx.read, assignment	cpu.migrations, task.clock, cpu.clock

Table 8: NMRSE and feature selected by LASSO for Parboil benchmarks

Beyls et al. [8] state reuse distance of a memory access as "the number of accesses to unique addresses made since the last reference to the requested data". In a fully associative cache with n lines, a reference with reuse distance $d < n$ will hit, and with $d \geq n$ will miss. To improve data locality based on reuse distance within loop structures, loop transformations, namely loop tiling and fusion, have been proposed and implemented in compilers [7, 10, 3].

We use the *Polly* data locality loop optimiser [3] in LLVM over SIR and the automotive EEMBC benchmarks. The motivation was to assess the energy gains from optimising data locality within loops in these benchmarks. Table 9 lists the energy gains observed over SIR benchmarks along with execution time and power gains.

SIR prog.	Energy gain (%)	Exec.time gain (%)	Power gain (%)
gzip	3.38	3.71	-0.37
printtokens	5.06	4.99	0.28
printtokens2	6.36	6.10	0.16
replace	-0.16	0	-0.17
schedule	0.24	0	0.25
schedule2	0.22	0.10	0.03
sed	4.59	4.95	-0.31
tcas	0.03	0	0.04
totinfo	-0.04	-0.2	0.12

Table 9: Energy gains in SIR benchmarks from Polly loop optimiser

It is evident from Table 9 that the energy gains achieved over SIR benchmarks using Polly for loop optimisations is relatively low, maximum being 6.36% (over *printtokens*2) This is primarily because in SIR programs, loops are used sparringly with relatively few data and array references within them. As a result, there is not enough opportunity for Polly to improve data locality. For instance, in the case of the *tcas* program where almost no energy gain was observed, loop structures are not present in the program and as a result Polly has no effect. *printtokens*, on the other hand, is a program which accepts a file containing a stream of characters and for each character (outer loop) it performs an expensive search to find the corresponding token (inner loop). This loop structure provides relatively more opportunities for Polly to optimise than in the case of *tcas*, resulting in the higher energy gain of 5.06%.

As seen in Table 9, energy gains follow execution time gains for most SIR programs. This is because, improved data locality and thus, reduced cache misses helps reduce execution time and not power. For *schedule* and *tcas* programs, we observe no gain in execution time, however negligible gains in energy because of power is observed. It is difficult to pinpoint the exact reason for the slight gain in power and can often be simply attributed to small fluctuations in the processor environment.

For the EEMBC benchmarks, we ran the Polly optimiser over the automotive domain comprising of 16 programs and associated workloads. Table 10 lists the energy gains with

EEMBC prog.	Energy gain (%)	Exec.time gain (%)	Power gain (%)
idctrn01	83.94	83.53	2.48
a2time01	45.74	44.96	1.41
aifftr01	63.73	63.79	-0.18
aifirf01	66.18	65.91	0.77
aiifft01	60.65	60.45	0.49
basefp01	66.50	64.96	4.37
bitmnp01	51.82	50.36	2.94
cacheb01	61.02	60.70	0.82
canrdr01	78.27	78.10	0.76
iirflt01	72.10	71.65	1.58
matrix01	71.79	72.72	-3.41
pntrch01	61.95	62.98	-2.79
puwmod01	67.54	67.45	0.25
rspeed01	63.92	63.82	0.28
tblook01	49.48	48.54	1.83
ttsprk01	64.79	64.06	2.03

Table 10: Energy gains in EEMBC benchmarks from Polly loop optimiser

Polly. In complete contrast to the SIR benchmarks, we observed massive energy gains with Polly, ranging from 45.74% to 83.94%. The large gains can be explained by the program structure of the EEMBC benchmarks, predominantly comprising of statements in for loops and array accesses. Polly's loop tiling and loop fusion optimisations perform extremely well over such program structures. Polly's data locality optimisations resulted in execution time gains that were similar to the energy gains observed, ranging from 44.96% to 83.53%. Power gains over the EEMBC programs were relatively low, and even negative for 3 of the 16 programs. The observations in Table 10 supports our earlier inference from Table 9 that energy gains are driven by execution time, rather than power, gains when using the Polly optimiser.

Context switches and CPU migrations. Both these features are relevant to energy consumption and are known to cause overheads in terms of running time [25, 14] and power [13]. Switching a process context or migrating it to a different CPU core is a costly task and tends to generate many cache misses (since the process may not find valid data on the new migrated core's cache). CPU scheduling algorithms have been proposed to help tackle this problem to reduce the power and energy consumed [13]. Some multi-core programming models (like OpenMP and MPI) allow developers to use environment variables that pin processes to CPU to prevent migration to a different core.

Program Level This halstead metric was selected for the SIR benchmark family. It is worth noting that for EEMBC, the linear regression model assigned the highest coefficient value to this feature. Lasso did not select this metric since it was, likely, redundant with respect to execution time. We hypothesize that reducing the number of operators, operands and function calls (that are used to compute this metric) in a program will help

in reducing energy consumption. In a previous study, we found that optimising design patterns by reducing function calls improved energy consumption [33].

Optimising for Energy versus Performance.
The features picked by our empirical study that affect software energy, namely execution time, cache misses, context switches, CPU migrations, program length (in terms of number of operators, operands and function calls) are **no different** from those known to affect software performance with respect to running time, memory latency, and power consumed. Datta and Patel propose CPU scheduling algorithms to mitigate performance loss, in terms of *power*, incurred from context switches, and CPU migrations [13]. David et al. and Li et at. have also studied the performance overhead introduced by context switches [14, 25]. *Cache misses* are widely known to impact performance in terms of execution time [53] and existing optimisations for data re-use like Polly achieve significant energy gains by reducing cache misses. Compiler optimisations built into existing compilers, like the GCC optimisation flags, attempt to reduce the *code size* and *execution time* of the program [23, 35].

All of the existing optimisation techniques proposed for performance, either focus on power or on execution time, but *not* both. Existing performance optimisations are effective in reducing energy and this has been observed in several studies [35, 47, 13, 52, 30, 7, 33]. It is, however, important to bear in mind that execution time gains do not always translate to energy gains. An example of this, is frequency scaling in processors that reduces execution time by running at full clock speed. This technique, however, proportionally increases power consumed as seen from the equation,

$$P = \frac{1}{2}CV^2f \tag{5}$$

where C is capacitance, V voltage and f is the frequency. Miyoshi et al. observed that scaling CPU frequency, although reducing execution time, can cost more energy than running at a slower clock speed [30]. Typical techniques for reducing power are power management software and dynamic voltage frequency scaling in processors. Our study does not consider these techniques since they focus on processors rather than a specific application.

The features affecting energy selected in our study lead us to believe that existing performance optimisations will suffice if one looks to reduce energy by reducing either only power consumed or only execution time. To achieve energy gains that goes beyond existing optimisations, we need to explore techniques that target *both* power and execution time. We plan to explore this problem in our future research.

6. THREATS TO VALIDITY

Our experiments suffer from the following internal threats to validity,

- We did not enable and examine the effect of standard compiler optimizations on energy consumption. We did, however, use Polly for data locality optimisations in loops. Enabling compiler optimisations has been shown to reduce energy consumption by reducing program execution time [23, 35].
- In our experimental setup, we do not completely control the services, unrelated to the benchmark being measured, run by the operating system. We used the server version of the Ubuntu distribution in order to limit the number of applications and background services running concurrently. However, some operating system services might still be running (such as checking for - but not performing, new updates). We did not specifically disable these services.

The external threat to validity in our experiment is using only a single hardware platform for running benchmarks and collecting observations. Our results cannot be generalized to other hardware architectures since the hardware and program specific features observed and measured will be very different. A similar experiment will have to be conducted to learn the features relevant to energy consumption for each hardware platform and each application domain. This is a consequence of the **no free lunch theorem** [31] that states that "there is no single best model that works optimally for all kinds of problems". The reason for this is that a set of assumptions that works well in one domain or achitecture may work poorly in another.

7. CONCLUSIONS

We have studied energy consumption of programs and workloads from three different benchmark families measuring over 100 different software and hardware features. The feature selection technique, Lasso, effectively picks five to seven features for each of the benchmark families that are relevant to energy consumption. The accuracy of energy predictions with the selected features was comparable to energy predictions using a linear regression model with all the features. The features commonly selected in our experiments were execution time, cache accesses, memory instructions, context switches, CPU migrations, and program length (Halstead metric). All of these features are known to affect software performance, in terms of running time, power consumed and latency. As a result, we can use existing performance optimisation techniques that reduce power or execution time to also achieve energy gains. We confirmed with the use of Polly, a compiler optimisation for data locality, that significant energy gains were possible by reducing latency and execution time. We believe the future in energy optimisation lies in techniques that can reduce *both* power and execution time.

8. REFERENCES

[1] Frama-C Metrics plugin. http://frama-c.com/metrics.html.
[2] Perf tool. https://perf.wiki.kernel.org/.
[3] Polly LLVM library. http://polly.llvm.org/index.html.
[4] Watts Up? power meter. https://www.wattsupmeters.com.
[5] Frama-c. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*. 2012.
[6] Richard Bellman. *Adaptive Control Processes: A Guided Tour*. 'Rand Corporation. Research studies. Princeton University Press, 1961.
[7] Kristof Beyls and Erik H D'Hollander. Refactoring for data locality. *Computer*, 42(2):62–71, 2009.
[8] Kristof Beyls and Erik DâĂŹHollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360, 2001.
[9] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th Intl. Symposium on*, pages 83–94, June 2000.
[10] Steve Carr, Kathryn S McKinley, and Chau-Wen Tseng. *Compiler optimizations for improving data locality*, volume 28. ACM, 1994.
[11] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. Process-level power estimation in vm-based systems. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 14:1–14:14, 2015.
[12] Luis Corral, Anton B. Georgiev, Alberto Sillitti, and Giancarlo Succi. Can execution time describe accurately the energy consumption of mobile apps? an experiment in android. In *Proceedings of the 3rd Intl. Workshop GREENS 2014*.
[13] A.K. Datta and R. Patel. Cpu scheduling for power/energy management on multicore processors using cache miss and

context switch data. *Parallel and Distributed Systems, IEEE Transactions on*, 25(5):1190–1199, May 2014.

[14] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. Context switch overheads for linux on arm platforms. In *Proceedings of the 2007 Workshop ExpCS*.

[15] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An Intl. Journal*, 10(4):405–435, 2005.

[16] Thanh Do, Suhib Rawshdeh, and Weisong Shi. ptop: A process-level power profiling tool. In *Proceedings of the 2nd Workshop on Power Aware Computing and Systems*, 2009.

[17] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22, 2010.

[18] Tony D Givargis, Jörg Henkel, and Frank Vahid. Interface and cache power exploration for core-based embedded system design. In *Proceedings of the 1999 IEEE/ACM Intl. conference on Computer-aided design*, pages 270–273. IEEE Press, 1999.

[19] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[20] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[21] Shuai Hao, Ding Li, W G J Halfond, and R Govindan. Estimating mobile application energy consumption using program analysis. In *35th ICSE*, pages 92–101.

[22] Mats PE Heimdahl, Michael W Whalen, Ajitha Rajan, and Matt Staats. On mc/dc and implementation structure: An empirical study. In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 5–B. IEEE, 2008.

[23] Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Wu Ye. Influence of compiler optimizations on system power. In *Proceedings of the 37th Annual Design Automation Conference*, pages 304–307. ACM, 2000.

[24] O. Landsiedel, K. Wehrle, and S. GoÌĹtz. Accurate prediction of power consumption in sensor networks. In *The Second IEEE Workshop on Embedded Networked Sensors*, pages 37–44, May 2005.

[25] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop ExpCS*.

[26] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. Calculating source line level energy information for Android applications. *Proceedings of the ISSTA 2013*, page 78.

[27] Fan Li, Yiming Yang, and Eric P Xing. From lasso regression to feature vector machine. In *Advances in Neural Information Processing Systems*, pages 779–786, 2005.

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[29] T.J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.

[30] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *Proceedings of the 16th Intl. conference on Supercomputing*, pages 35–44. ACM, 2002.

[31] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[32] Adel Noureddine. *Towards a Better Understanding of the Energy Consumption of Software Systems*. Theses, Université des Sciences et Technologie de Lille - Lille I, March 2014.

[33] Adel Noureddine and Ajitha Rajan. Optimising energy consumption of design patterns. In *The 37th ICSE'15, NIER track*, 2015.

[34] Ping-Wen Ong and Ran-Hong Yan. Power-conscious software design-a framework for modeling software on hardware. In *Low Power Electronics, Digest of Technical Papers., IEEE Symposium*, pages 36–37, 1994.

[35] James Pallister, Simon J Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 58(1):95–109, 2015.

[36] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. Where is the energy spent inside my app? In *Proceedings of EuroSys '12*, page 29.

[37] J.A. Poovey, M Levy, S Gal-On, and T Conte. A benchmark characterization of the eembc benchmark suite. *Micro, IEEE*, PP(99):1–1, 2009.

[38] EU FP7 project-288021. Network of Excellence in Internet Science. Overview of ict energy consumption. http://www.internet-science.eu/sites/eins/files/biblio/EINS_D8 1_final.pdf.

[39] Changwoo Pyo, Kyung-Woo Lee, Hye-Kyung Han, and Gyungho Lee. Reference distance as a metric for data locality. In *In HPC Asia'97*, pages 151–156. IEEE, 1997.

[40] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[41] Ajitha Rajan. Coverage metrics to measure adequacy of black-box test suites. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 335–338. IEEE, 2006.

[42] Jeffry T Russell and Margarida F Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *In Proceedings. ICCD'98*, pages 328–333. IEEE, 1998.

[43] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. *Lecture Notes in Computer Science*.

[44] Amit Sinha and A.P. Chandrakasan. JouleTrack-a Web based tool for software energy profiling. *Proceedings of the 38th Design Automation Conference*, (617), 2001.

[45] Matt Staats, Michael W Whalen, Ajitha Rajan, and Mats Per Erik Heimdahl. Coverage metrics for requirements-based testing: Evaluation of effectiveness. In *NASA Formal Methods*, pages 161–170, 2010.

[46] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, V. Chang, N. Anssari, G. Liu, and W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, IMPACT Technical Report, IMPACT-12-01, University of Illinois, at Urbana-Champaign, 03 2012.

[47] T.K. Tan, A. Raghunathan, G. Lakshminarayana, and N.K. Jha. High-level software energy macro-modeling. In *Design Automation Conference, 2001. Proceedings*, pages 605–610, 2001.

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

[49] V. Tiwari, S. Malik, A. Wolfe, and M.T.-C. Lee. Instruction level power analysis and optimization of software. In *VLSI Design, Proceedings., Ninth Intl. Conference on*, pages 326–328, Jan 1996.

[50] N. Vijaykrishnan, M. Kandemir, M.J. Irwin, H.S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *Computer Architecture, 2000. Proceedings of the 27th Intl. Symposium on*, pages 95–106, June 2000.

[51] Roberto A. Vitillo. Performance tools developments. In *Future computing in particle physics*, pages 36–37, Edinburgh, UK, 2011. e-Science Institute.

[52] W. Ye, N. Vijaykrishnan, M. Kandemir, and MJ. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proceedings of the 37th Annual Design Automation Conference*, pages 340–345. ACM, 2000.

[53] Yijun Yu, Kristof Beyls, and Erik H D'Hollander. Visualizing the impact of the cache on program execution. In *Information Visualisation, 2001. Proceedings. Fifth Intl. Conference on*, pages 336–341. IEEE, 2001.