ENTRA

318337

Whole-Systems ENergy TRAnsparency

High-Level Energy Models

Deliverable number:	D2.3
Work package:	Energy Modelling Through the System Layers (WP2)
Delivery date:	1 July 2015 (33 months)
Actual date:	11 September 2015 (revised 01.03.2016)
Nature:	Report
Dissemination level:	PU
Lead beneficiary:	University of Bristol
Partners contributed:	Roskilde University, University of Bristol, IMDEA Software Institute, XMOS Limited

Project funded by the European Union under the Seventh Framework Programme, FP7-ICT-2011-8 FET Proactive call.

Short description:
This deliverable presents energy models that apply higher levels of abstractions or new levels of information and detail, building upon the work conducted earlier in the project and previously presented in D2.1. The work herein comprises extensions into several areas, each of which provides new dimensions for further design space exploration by software developers. It feeds into several tasks of other work packages, namely energy aware tools (T1.1) in energy-aware software engineering (WP1), along with energy and quality trade-offs (T4.2) and energy aware scheduling (T4.3) in optimization (WP4).

There are four main topics that this deliverable contributes to: energy modelling at the intermediate representation used during code compilation, multi-core communication modelling, worst case data-aware energy modelling, and modelling of application source code.

This deliverable includes the following attachments:

- D2.3.1. *Modelling software energy consumption in a multi-core network of embedded multi-threaded processors*. In preparation for conference submission.
- D2.3.2. *Data dependent energy modelling: A worst case perspective*. In preparation for conference submission.
- D2.3.3. *On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs*. arXiv preprint arXiv:1510.07095, 2015.
- D2.3.4. *On the infeasibility of analysing worst-case dynamic energy*. To be submitted to the ACM Journal of Transactions on Embedded Computing (TECS).

Contents

1 Introduction

The ENTRA project strives to communicate energy consumption data through the many layers of abstraction that are present in the engineering of modern embedded systems. The previous deliverable from work package 2, D2.2, presented low-level energy modelling techniques, that could be used to inform analysis tools, potentially directed by the common assertion language defined in deliverable D2.1. This deliverable extends the modelling upwards, into higher levels of abstraction, as well as looking at new parameters that can aid design space exploration.

Several areas of work have contributed to achieving this. The accuracy of higher level modelling and analysis is often dependent upon the accuracy of the underlying models, so refinements to low-level models and simulators have been made in order to improve the foundational aspects of this work. The parametrisation of models has also been broadened, in order to explore opportunities for wider ranging design exploration, as well as the potential to define more detailed bounds that can be used for example with the common assertion language. These include data-dependent energy modelling, where the data values processed by a device have an influence on the energy consumption, not just the instructions (Section 4), and voltage frequency scaling (VFS) parameters that can be applied to processors to constrain performance whilst reducing power (Section 3).

Three higher level approaches are detailed. The first (Section 2) demonstrates and evaluates an application of a mapping from the LLVM compiler's Intermediate Representation (IR) to the low level energy model of a processor's Instruction Set Architecture (ISA). The second approach extends our core energy modelling efforts to multiple networked cores (Section 3), with new simulation improvements as well as higher level modelling and visualisation of energy consumption. Finally, we propose a new method for modelling energy consumption at application source code level, specifically for software targeting the Android Operating System (Section 5).

2 Energy modelling at the LLVM IR level

In deliverable D3.1, we introduced a novel mapping technique to lift our ISA-level energy model to a higher level, the intermediate representation of the compiler, namely LLVM IR [LA04], implemented within the LLVM tool chain [LLV14]. This enables Energy Consumption Static Analysis (ECSA) to be performed at a higher level than Instruction Set Architecture (ISA), thus introducing energy transparency into the compiler tool chain by making energy consumption information accessible directly to the optimizer.

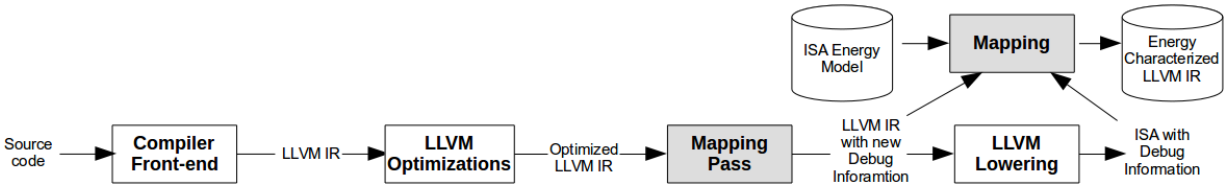


Figure 1: Overview of the mapping process.

A prototype version of a mapping tool implementing the mapping techniques was described in Deliverable D1.1. An overview of this tool is given in Figure 1. Our mapping pass is introduced into the compilation process after LLVM optimizations to tag each LLVM IR instruction with a unique debug location. The mapping phase runs after the LLVM lowering phase and maps LLVM IR instructions with the new debug locations to the emitted ISA instructions. The ISA energy model is then used to accumulate the energy value of each LLVM IR instruction based on its mapped ISA instructions. This technique was further calibrated and evaluated using a set of single- and multi-threaded benchmarks, mainly selected from a number of industrial embedded applications. For the evaluation we performed ECSA based on two techniques.

The first ECSA technique is based on setting up a system of recursive cost equations over a program P that capture its cost (energy consumption) as a function of the sizes of its input arguments. This work has been performed for single threaded benchmarks and the results were presented in deliverable D3.2 (Attachment D3.2.4) as well as in deliverable D1.1 and integrated into CiaoPP tool. An improved version of this work has been accepted for publication [LGK$^+$15]. This work studies the ECSA analysis at different levels (ISA vs. LLVM IR), the results show that LLVM IR level analysis using mapping techniques to lift an ISA model to LLVM IR level is a good compromise since 1) much of the program information (e.g. types) is available at LLVM IR level, unlike at ISA level, allowing analysis of bigger class of programs, and 2) the analysis performed at either level is reasonably accurate and the relative error between the two analyses at different levels is small. ISA-level estimations are slightly more accurate than the ones at the

LLVM IR level (3.9% vs. 6.4% error on average with respect to the actual energy consumption measured on the hardware, respectively).

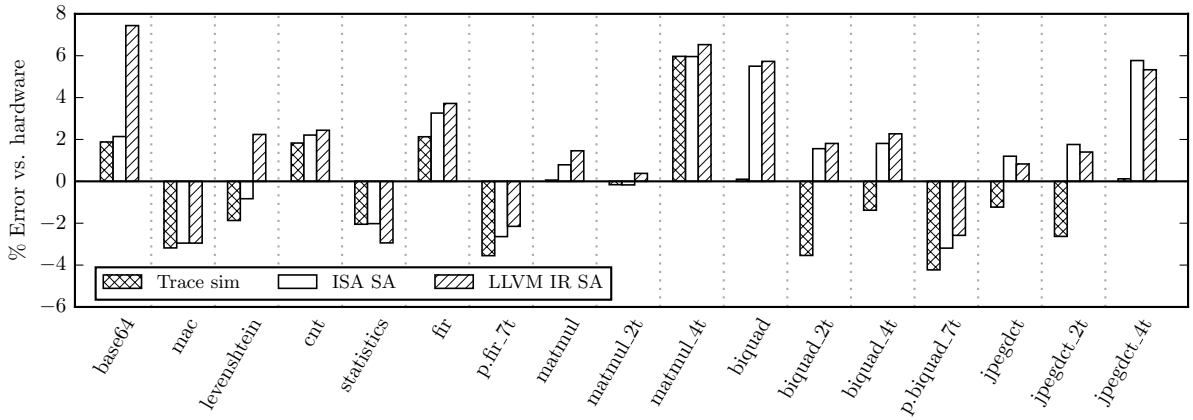
The second ECSA technique is based on Implicit Path Enumeration Technique (IPET) [LM95] and performed for both single- and multi-threaded benchmarks, which we introduced in deliverable D1.1 under Section 4, *Work in Progress*. Our results show that the mapping technique allowed for energy consumption transparency at the LLVM IR level, with accuracy keeping within 1% of ISA-level estimations in most cases. The rest of this section provides an evaluation of this technique.

2.1 Benchmarks and results

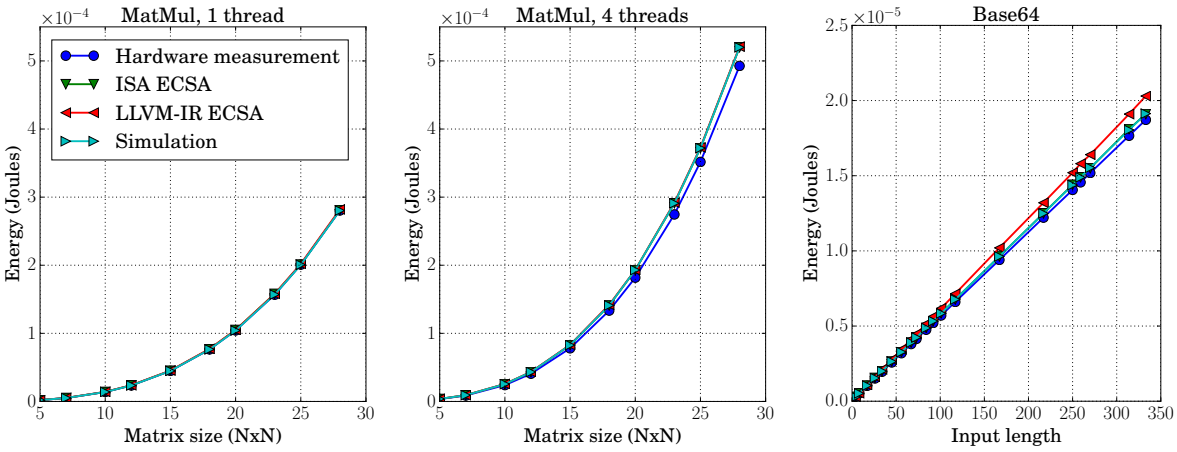
Our objective is to calibrate and evaluate the effectiveness of our mapping technique on common industrial, deeply embedded applications. Table 1 summarizes all of the benchmarks' attributes. The meaning of the columns is as follows: *Source* indicates where the benchmark was obtained, *Description* provides insight into the benchmarks' functionality, *NCSL* is the number of non-commentary source lines of code, *T* is the number of threads used, and the remaining columns indicate the presence of loops (*L*), nested loops (*N*), arrays and/or matrices (*A*), bitwise operations (*B*), a complex CFG structure (*CP*), multiple functions (*MF*) and thread communications (*C*).

To do so, three analysis techniques were compared to hardware energy measurements for our benchmarks. Figure 2a presents the error margin of ISA energy model combined with the three analysis techniques compared to hardware energy measurements for our benchmarks. Trace Sim produces instruction traces from simulation, ISA ECSA uses the model for static analysis at the ISA level and LLVM IR ECSA uses our mapping technique to apply the model and analysis at LLVM IR level. For all benchmarks with multiple test parameters, the geometric mean of the errors is used. Figure 2b compares energy estimates to hardware measurements for a range of parameters in three appropriate benchmarks. Benchmarks were compiled with `xcc` version 12 [XMO14] at optimization level `O2`, which is the default for most compilers.

For the software division and floating point benchmarks, ECSA provides a constant energy consumption across all test cases, as they contain no loops that are directly affected by the functions' arguments. Figure 3 demonstrates this for `Radix4Div` and `B. Radix4Div`. Considering that IPET is intended to provide bounds based on a given cost model, in our case it tries to select the worst case execution paths in terms of the energy consumption, and therefore ECSA estimations seen in Figure 3 represent a loose upper bound on the energy consumption of each benchmark. Similar figures were also retrieved for the two `SoftFloat` benchmarks. These bounds in most cases can not be considered safe, as they might be undermined by the use of a non data sensitive energy model and analysis. However, they can still give the application



(a) All benchmarks.



(b) Parametric benchmarks.

Figure 2: Hardware measurements compared to ECSA and ISA trace estimation.

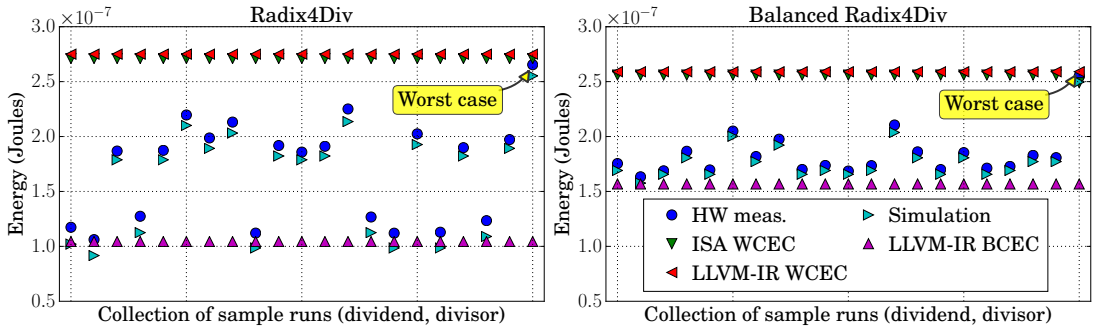


Figure 3: Results for benchmarks with constant ECSA estimations across all test cases.

programmer valuable guidance towards energy aware software development in the absence of energy measurements.

Generally, for all results shown here a proportion of error is present in both forms of static analysis as well as simulation based energy modelling. The error in the simulation based model is a baseline for the best achievable error in static analysis, as simulation has more accurate execution information available to it. For all the benchmarks, the ISA ECSA results are over-approximating the trace based energy estimations. This applies also to the LLVM IR ECSA results with exception of the `statistics` benchmark. This over-approximation is a product of the bound analysis used which is trying to select the most energy costly CFG path based on the provided cost model. A smaller difference between the results of the ISA ECSA results and the trace based energy estimations indicates that the execution path selected by the IPET fits better the actual execution path of a benchmark than LLVM IR ECSA.

2.2 LLVM IR analysis accuracy

This form of analysis is solely dependent on the accuracy of the mapping technique. As can been seen in Figure 2a, for all benchmarks the LLVM IR ECSA results are within one percentage point error of ISA ECSA results, except for `Base64` benchmark with a further 5.3 percentage points error. In this case the CFGs of the two levels were significantly different due to basic blocks introduced from branches in the ISA level CFG. This is one of the few cases where the mapper was unable to accurately track the differences between the two CFGs.

2.3 LLVM IR level ECSA applications

Our LLVM IR analysis results demonstrate a high accuracy with a deviation in the range of 1% from the ISA ECSA. Some LLVM IR estimations may not always be as accurate as at ISA level,

but they are still of value to developers. The LLVM optimizer and code emitter are the natural place for compiler optimizations. Transparency of energy consumption at this level enables programmers to investigate how optimizations affect their program's energy consumption, or even help introduce new low energy optimizations. This is more applicable at the LLVM IR level than at the ISA level, because more program information exists at that level, such as types and loop structures. The presented mapping techniques and analysis framework at the LLVM IR level are applicable to any compilers that use the LLVM common optimizer, provided that an energy model for the target architecture is available.

For some programs, indirect jumps that are introduced at the ISA level can make it impossible to extract a CFG. While this prevents ISA level ECSA, it can still be performed for these programs at LLVM IR, allowing programmers to gain energy consumption insight even where ISA level analysis is not feasible.

2.4 Related publications and dissemination

The mapping techniques for LLVM IR energy characterization have been used into two papers, for energy static analysis at the LLVM IR level. The first one, [GGP+15], was presented at SCOPES 2015 [Sco], and the second one, [LGK+15], is accepted to appear in *Foundational and Practical Aspects of Resource Analysis (FOPARA) 2015* [Fop]. A paper detailing the mapping techniques and their evaluation is attached as D2.3.3.

3 Multi-core modelling with communication costs

In a multi-core system of communicating software, both the core energy consumption and the communication energy consumption must be considered. Further, the timing impact of multi-core communication, and the increased latency in the movement of data, can have an impact upon execution time and therefore total system energy consumption.

Work during the D2.3 period has endeavoured to improve existing energy models and extend them to support these types of systems and software. An overview of this work is given here, and a paper detailing the research contributions and results from this work is attached as D2.3.1.

3.1 Improved core energy model

The work of [KE15a] is expanded upon in order to produce a more refined core energy model. This is achieved by performing additional profiling, capturing the energy consumption of a larger number of instructions. The remaining un-profiled instructions are modelled using a decision tree regression technique, where key instruction features are used to estimate energy, based on the demonstrated impact of those features when present in profiled instructions.

This work resulted in an improvement in average error to 2.67 % and also reduced the standard deviation of the results in the same set of benchmarks used on the original model. These model improvements can be fed into the higher level analysis that have been presented in deliverables D3.1 and D3.2, such as those published as [LKS+14] and [GGP+15].

3.2 VFS energy consumption modelling

Although not specific to multi-core or communication modelling, part of the modelling effort was put into voltage and frequency scaling (VFS), in response to the identification of savings that could be had when applying VFS in the examples given deliverable D5.1. Modelling that can be parametrised by operating voltage and frequency allows a broader range of design space exploration for the software developer.

A VFS capable model was created by profiling a dual-core XMOS device with voltage scaling capabilities, the SLICEKIT-A16. This device contains two of the XS1-L cores modelled in [KE15a] and so provides a good base upon which to perform this further work. In addition to the processor cores, an analogue peripheral block sits on the processor network, one of the features of which is the control of the cores' 1 V supplies.

A set of profiling tests were constructed that exercised the SLICEKIT-A16 at idle and under load in various VFS states. The power data from this was then used to construct a characteristic

equation, parametrised by voltage, top-level system frequency and divided core frequency. The characteristic equations and their coefficients are given in the attachment D2.3.1.

The performance of the VFS modelling is good, with the model yielding a mean squared error against the profiling data of 2.6 % and total error range of 15.72 %. Given that the focus of research effort was put into multi-core and communication costs, further effort would be required in order to refine this version of the model and also demonstrate how it can be exploited. This is proposed as future work in D2.3.1.

3.3 Multi-core energy consumption and communication

The multi-core energy modelling effort seeks to achieve two things:

1. Accurately estimate the overall energy consumption of a multi-core system when running communicating multi-core software.

2. Demonstrate effective methods for conveying *where* energy is consumed in the system, in order to guide the optimisation efforts of either tools or the developer.

Additional profiling was performed to establish the cost of communicating data tokens across the XMOS XS1's inter-node links. Simple models are then produced for the switch and interconnect based on the quantity of traffic that they carry.

A multi-core model is then created by assembling a graph representing the target XMOS system, reading data from the platform specification "XN" files used in the XMOS toolchain. Nodes in the graph are either cores or switches, and have the appropriate energy models attached to them. Edges represent links between these components, which again can have a model attached.

The modelling is enabled by trace data from the `axe` instruction set simulator. Several enhancements were made to `axe`, including:

- Accurately tracing *fetch no-op* `FNOP` activity in the core, to improve core model accuracy.

- Provide more accurate instruction traces, with finer-grained timing precision, to assist the core model's pipeline occupancy parametrisation.

- Issue traces for tokens traversing links, allowing them to be modelled.

- Implement the credit based flow control used within the XS1 switch network, providing more accurate multi-core communication timing and contention behaviours.

The latter point is particularly important as without it, no instruction set simulator for the XS1 architecture correctly simulates the time taken to communicate between cores. Our contributions use the link parameters given in the XN file to simulate the same communication speeds and latencies as realised in the hardware. This significantly improves timing accuracy of multi-core simulations, where previously the timing error would lead to very large, potentially unusable energy modelling errors.

The timing model is demonstrated to have a sub-1 % error in the characterisation tests, and in larger benchmarks maintains single digit percentage error. This is two orders of magnitude better than what was previously available.

The simulator improvements are combined with the new graph-based network energy model and tested under the FIR and Biquad benchmarks presented in deliverable D5.1. The energy model error is shown to be less than ± 10 %, with an average of -4.92 % error.

In addition, a unique presentation method is shown, that highlights where the model has determined energy is being consumed. This is divided into cores, switches and links. This provides finer grained detail than normal hardware measurements provide, which typically give the overall system energy consumption, or a limited number of measurement points that capture multiple components simultaneously. This is used to demonstrate how a developer can identify the best way in which the software can utilise the hardware, considering optimal core utilisation as well as communication patterns. An explanation with examples, in the context of the Biquad filter benchmark, is given in attachment D2.3.1.

3.4 Dissemination

The work presented in this section is captured an attachment as D2.3.1 and is also available as a technical report [KE15b]. It will also shortly be submitted for peer-review with the intent to publish at an appropriate conference venue.

4 Worst case energy and defining data-aware energy models

Prior to concerns about energy consumption, execution time was an essential consideration for embedded systems software. Significant work has been presented that seeks to analyse programs to determine properties such as Worst-Case Execution Time (WCET) [WEE+08]. The ENTRA project's research into estimation of software energy consumption, can be framed in a similar way, whereby a worst case energy consumption is sought from the analysis of a program.

However, by the nature of energy consumption's relationship to power and time where $E = P \cdot T$, an extra dimension of complexity is introduced. This is because the power term, P, cannot be determined solely with respect to the instructions that a processor executes; the data that is used as input to, and produced as output from an instruction, has an effect on P even if T does not change.

Examination of prior work shows that data-dependant energy models for worst case or otherwise bounded energy consumption metrics have not been given any in-depth exploration. Thus, for more sophisticated, data dependant energy models to be applicable with confidence, further investigation was needed. In answer to this, in [PKME15] we present our methods and findings for the exploration of data dependant energy behaviour, as well as first steps towards modelling in this dimension. At the time of writing, this paper is currently being prepared for conference submission, and has been published in pre-print on arXiv. It is also included with this deliverable as attachment D2.3.2.

However, further research has revealed, that in the most general case, worst case analysis of data switching costs is infeasibly complex, both for a precise analysis and for any level of approximation. The optimization problem corresponding to calculating the worst case switching cost can be shown to be NP-hard, and in addition it can be shown that any approximation algorithm that guarantees a level of accuracy must also be NP-hard.

This theoretical result mandates a certain level of uncertainty in any worst-case energy estimation as the switching costs cannot be accurately estimated. Fundamentally, it must be asked whether worst case analysis can deliver a useful energy estimate given this limitation, and whether other avenues of exploration are worth following. Techniques such as the statistical analysis of data-dependent energy may become more significant given this uncertainty.

Our work on this topic is included with this deliverable as attachment D2.3.4, including background and analysis of the significance of data switching costs. It is currently under consideration for acceptance to a journal.

5 Energy modelling of application source code

A source-code level energy model is motivated by the gap that must be bridged in order to relate low level energy behaviour to the high level software. This work focuses on complex Java code forming a library for mobile gaming on Android devices. As such, this is substantially different to the pure C, embedded real time code examined in other parts of this and previous deliverables. However, the goals of the project still apply to this type of software development scenario and the techniques investigated in this section could be applied to other high-level languages.

This work establishes an energy model at the source code level, extracting energy costs for operations performed in the code from empirical measurement, which are then used to build a vector of energy costs for those operations. This model can then be applied to other applications that utilise the library to estimate their energy consumption characteristics.

This work reported in this deliverable is ongoing work.

5.1 Introduction and Motivation

The ENTRA project aims to support energy-aware software development, in contrast to the energy-oblivious manner typical today. Throughout the engineering life-cycle, developers are blind to the energy usage of the code written by themselves. On the other hand, it is estimated that energy saving by a factor of three to five could be achieved solely by software optimization [Edw11]. The approach of the ENTRA project is to achieve energy transparency through a combination of static analysis and energy modelling techniques, enabling the developer to understand the energy distribution among different parts of the source code.

In this section we focus on energy modelling. Traditional energy modelling methodology [TMW94, vBDMH00], including approaches adopted in the ENTRA project, is bottom-to-top. In this approach an energy model for machine level instructions is constructed, and this low-level model is mapped upwards to intermediate code and source code. This approach faces obstacles when the software stack consists of a number of abstract layers.

An alternative approach could be called top-to-bottom, and aims to construct a model directly for the source code, without any explicit low-level model. Intuitively, given a target device and power management strategy, the source code completely determines the energy consumption. The overall goal is to identify the basic energy-consuming operations from the source code and find the correlations to energy cost by analyzing a large amount of test cases. The resulting energy model implicitly includes the effect of all the layers of the software stack down to the hardware.

5.1.1 An experiment in source-code energy modelling

The target of the experiments described below is the Android platform. In February 2015, the penetration of smartphones was about 75% in the U.S. This figure is expected to reach 85% by December 2015 [Mar15]. With the improvement of hardware processing capability and software libraries for smartphones, applications are becoming heavier and more PC-like. At the same time, users are annoyed by limited battery capacity, as parallel-running applications could easily drain the full-charged battery within 24 hours. Thus energy-aware development for smartphones is becoming a critical requirement.

On the Android platform, say, the source code is in Java and translated to Java byte-code, further to Dalvik [Andb] (simplified Java virtual machine for Android) byte-code, native code and machine code and finally has chance to execute on the processors. Consequently, the modelling and mapping tasks in a bottom-to-top approach would have to address the complex problem of characterizing the links among all the layers. For this reason we attempt a top-to-bottom modelling approach for Android source code.

In the experiment, our target platform is an Android development board with two ARM quad-core CPUs, and the employed source code is a game engine which is for constructing games, demos and other interactive applications. The result shows that the inference model achieves an accuracy about 80%. Based on this model, we aim to capture the energy properties of the source code, such as energy hotspots, energy bugs and other opportunities for optimizing energy consumption.

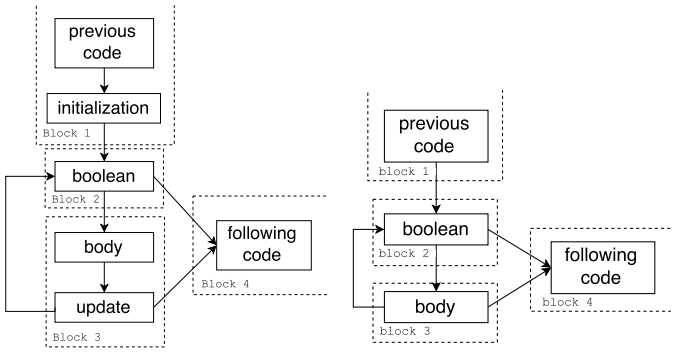
5.2 Code to Energy

We build the energy model by analyzing a large set of execution cases. The brief procedure of mapping the source code to energy is 1) obtaining the precise execution path, 2) tracing the corresponding power consumption, 3) labelling test cases with the energy cost and 4) employing the labelled data to train the energy model.

Three factors play significant roles in building the inference model, which are execution-path obtaining, statement breakdown and model training. In the following sections, we will illustrate how to accurately acquire executed source code, why we need to breakdown statements and approaches in model construction.

5.2.1 Block Division

We designed a large range of execution cases (in Section 5.3) to guarantee most corners in the source code are able to be covered. Not only the breadth of code coverage, the user interaction



(a) **For** loop (b) **While** loop

Figure 4: Block division of **for** and **while** loops in control flow graph

is also a significant dimension in the design space. As game applications are highly interactive, distinct input sequences result in huge varieties. The details of case design will be presented in Section 5.3.

Definition 1 *A block is a set of gathered statements. In the block, each node has only one in-edge and one out-edge in the control flow graph, but the start point of the block could have more than one in-edge and the end point could have more than one out-edge.*

A block is a fixed execution unit. That means, always if one part of the block is processed, the rest certainly will be executed. Its concept is declared in Definition 1. We choose the block as our basic sampling unit because tracing statements one by one has a vital impact on energy consuming and running time which damages the sampling precision. On the other hand, functions or classes are unstable execution unit, since we can not point out which certain parts the function or class will be active during run time.

For individual syntax structures, we deal with block division case by case. **For** loop an **while** loop are taken for examples as shown in Figure 4. In a **for** loop, the header usually has three segments which are *initialization*, *boolean* and *update*. According to Definition 1, the segments are divided into three different blocks. Following the same logic, we set the **while** header itself as a block.

The complete set of reached code points is acquired by recording the executions of blocks. We instrument the source code with a log instruction at the beginning of each blocks. It generates a block ID and a time stamp as one record in the log file which will be analyzed in later stage to obtain the execution path. As shown in Section 5.2.4, one execution case is corresponding to one input of the training model. The input features are extracted based on the path.

5.2.2 Basic Energy Operations

It is impossible to characterize the energy cost straightly based on individual statements because they varies largely, any pair of statements in the code are probably distinct. An arithmetic expression, say, could have two operators or three or more which could be additions or multiplications or mixed. In contrast, if we go to the function level, the model will be restricted to the domain of our target source code, since it's unlikely to find identical functions in different applications. We employ the "energy operation" as the basic modelling unit, such as arithmetic operations, comparison operations, method invocations etc. At the end, we in fact model the energy of individual operations. The entire energy consumption is made up of the cost of all operations in the code.

The operations (ops) are grouped into eight classes as shown in Table 2. In practice, these ops refer to one or more virtual machine instructions, but it is not important for the method to know exactly which ones. For example, the *multiplication* op with integer operands could be implemented by the *imul* instruction in the Java virtual machine instruction set [Ora] and then by other Dalvik opcodes [Andb]. The *block goto* op corresponds to the *jsr* instruction whose job is leading the CPU to an aimed subroutine. A basic assumption of our approach is that each of the source-level ops corresponds to a set of machine instruction which use a certain set of hardware components, which results in approximately the same energy cost each time it is called. This assumption may not always hold and other factors might also lead to different energy costs for different occurrences of the same op, leading to imprecision in the model, but it is an assumption that allows a reasonable estimate of energy consumption. Our experimental results show that the energy op is an appropriate intermediate representation to bridge the gap from source code to machine instructions, with regard to energy consumption.

5.2.3 Data Collection

$$N_e(op_i) = \sum^{block_j \in BLK(op_i)} \{N_e(block_j) \cdot N_o(op_i,\ block_j)\} \tag{1}$$

$$where \quad op_i \in Energy\ Ops$$

We developed a parser to extract the energy ops from the source code. All of the ops are labelled with the ID of the block where it resides. Note that, in the model building stage one record of training data couples one execution case, which is an one-on-one relation. One record consists of the numbers of executions of individual ops which are figured up according to Equation (1). $BLK(op_i)$ is the set of blocks that contain op_i. $N_o(op_i, block_j)$ means the occurrence of op_i in $block_j$. $N_e(block_j)$ is the executions of $block_j$. Basically, the executions of op_i is equivalent to

the sum of the products of $N_e(block_j)$ and $N_o(op_i, block_j)$.

Android applications utilize a rich range of Android's APIs and Java library classes where the fine-grained execution path is hard to capture. On the other hand, only a small proportion is frequently used during run time. We list the highly-referred library functions (Lib Funcs) in table 3, which are treated as special energy ops in the training stage. In particular, the `GL10` class is the key interface for applications to implement their graphic computing.

5.2.4 Power Tracing & Model

In the experiment stage (Section 5.3), each execution case runs twice. In the first run, we record the execution path without power measuring. In the second time, we only trace power and disable the log instructions. We split the path and power obtaining work because the log instructions take up a part of the entire energy consumption which could not be neglected.

$$E = \sum_{i=1}^{n} power(t_i) \cdot \Delta_i, \quad (t_0 \leq t_1 \leq t_2 \cdots \leq t_{n-1} \leq t_n) \tag{2}$$

The power trace is acquired by the measurement equipment, after which we approximate the energy cost (E) by calculating the integral of power, as shown in Equation 2. $p = power(t)$ is the power-vs-time function, so $power(t_i)$ is the measured power value at time stamp t_i. We let $\Delta_i = t_i - t_{i-1}$ which is the interval between two sequential samplings.

$$E = \sum^{op_i \in Energy\ Ops} Cost_{op_i} \cdot N_e(op_i) \tag{3}$$
$$+ \sum^{func_i \in Lib\ Funcs} Cost_{func_i} \cdot N_e(func_i) + Idle\ Cost$$

The aimed model is formalized in Equation 3. The entire energy consumption consists of the sum of the costs of operations and library functions and idle cost. Notice that the idle costs of individual cases are different, since they are executed in distinct sequences of inputs, thus the lengths of sessions are also varying.

5.3 Experiment Setup

5.3.1 Target & Power Measurement

Experiment target: Odroid-XU+E development board [Odr]. It possesses two ARM quad-core CPUs, Cortex-A15 with 2.0Ghz clock rate and Cortex-A7 with 1.5Ghz. The eight cores are

grouped into four pairs. Each pair consists of one big and one small core. So in the view of operation system, there are four logic cores. In our experiment, we turn off the small cores and only run workload on big cores at a fixed clock frequency of 1.1Ghz. This is for removing the influence of voltage, clock rate and CPU performance on power usage.

Power Measurement: Odroid-XU+E has a built-in power monitor tool to measure the voltage and current of CPUs with a sampling frequency of 30Hz and update the values in a file. We wrote a script to obtain the readings from the file every 0.1 second. During the test case, we run the script on a different core from where the application runs to minimize the interruption.

5.3.2 Source Code & Case Design

Our target source code is the Cocos2d-Android [Goo] game engine, a framework for building games, demos and other interactive applications. It also implements a fully-featured physics engine. The game is one of the main applications on smart devices, which has developed more and more PC-game-like, requiring a high CPU performance. The energy modelling and analysis research in this paper shows the opportunity to improve the source code and guide the software development towards energy efficiency.

We designed a large range of execution cases to simulate the game scenarios under different sequences of user inputs. We script with the Android Debug Bridge [Anda] (adb) , a command line tool connecting target device to the host, to automatically feed the input sequences to the target board. For instance, in the `Click & move` scenario, the sprite (the character in the game) moves to the position where the tap is. We designed 10 input sequences with distinct tap positions and intervals. Each sequence is executed three times repeatedly. So for the `Click & move`, we have 30 execution cases.

5.4 Model Construction

The model construction is on the strength of machine learning, finding out the correlation between energy ops and costs from a large amount of data. We set out our collected data in the following matrices. The first one (N) from left is execution numbers of l ops (including energy ops and library funcs) in m test cases, which is observed by logging and calculating, as shown in Section 5.2. Each row indicates one test case. The vector ($\vec{cost}$) in the middle contains the costs of l ops, which are the values we are aiming to obtain. The vector ($\vec{e}$) on the right of equation mark is the energy cost computed by Equation 2. So for each test case, the energy cost is the sum of the costs of all the ops. It should be noticed that the energy cost has excluded the idle cost which is measured when no application workload is processing.

$$\begin{pmatrix} n_1^{(1)} & n_2^{(1)} & ... & n_l^{(1)} \\ n_1^{(2)} & n_2^{(2)} & ... & n_l^{(2)} \\ & ... & ... & \\ n_1^{(m-1)} & n_2^{(m-1)} & ... & n_l^{(m-1)} \\ n_1^{(m)} & n_2^{(m)} & ... & n_l^{(m)} \end{pmatrix} \times \begin{pmatrix} cost_1 \\ cost_2 \\ ... \\ cost_l \end{pmatrix} = \begin{pmatrix} e_1 \\ e_2 \\ ... \\ e_{m-1} \\ e_m \end{pmatrix} \tag{4}$$

Inevitably, the execution tracing and power measurement is not absolutely accurate. Meanwhile, the energy model is not exactly subject to the linear property. As a result, the equation above is unsolvable since the vector $\vec{e}$ is out of the column space of N. To address this problem, we employ the well-known gradient descent algorithm [Ng12] to approximate the values of $\vec{cost}$.

The values of $\vec{cost}$ are randomly set and then improved by the gradient descent algorithm step by step. We first introduce the error function J (Equation 5) which indicates the quality of the model. $\vec{n^{(i)}}$ is the ith row in the first matrix, $\vec{cost}$ is the middle vector above. $\vec{n^{(i)}} \times \vec{cost}$ is the predicted energy cost for the ith test case, $e^{(i)}$ is the observed energy cost. J is the sum of the squared errors of all the test cases, which is afterwards divided by $2m$ to get the average values. The reason why $2m$ is applied, but not m, is that $2m$ is convenient for the derivative computation later. So the smaller J is, the better $\vec{cost}$ is.

$$J(cost_1, cost_2, ...cost_l) = \frac{1}{2m}\sum_{i=1}^{m}(\vec{n^{(i)}} \times \vec{cost} - e^{(i)})^2 \tag{5}$$

The idea of gradient descent is to minimize J by repeatedly updating each element in $\vec{cost}$ with Equation 6 until convergence. The partial derivative of J function on $cost_j$ gives the direction in which increasing or decreasing $cost_j$ will cut down J. The value α determines how large the stride is in each iteration. If it is too large, the extremum value possibly will be missed; if too tiny, the minimizing process will be time-consuming. It needs to be manually tuned. Theoretically, the gradient descent algorithm could only find the local optimal value. In practice, the initial values in $\vec{cost}$ are randomly set several times to look for the global optimization.

$$\begin{aligned} cost_j &:= cost_j - \alpha\frac{\partial J(cost_1, ...cost_j, ...cost_l)}{\partial cost_j} \\ &= cost_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(\vec{n^{(i)}} \times \vec{cost}) \cdot n_j^{(i)} \\ &j = 1, 2, ...l \end{aligned} \tag{6}$$

The algorithm above may produce $\vec{cost}$ with negative elements, however as a matter of fact, the energy costs should be above zero. In view of this, we customize the original error functions

by adding a correction part for negative values, as shown in Equation 7. Smaller the negative cost is, larger the penalty is. The λ value balances the weight of correction part and that of the original part. The ρ value determines how aggressive the correction is. Both λ and ρ should be adjusted by hand. Consequently, the Equation 6 is re-written as Equation 8.

$$J = \frac{1}{2m}\sum_{i=1}^{m}(\vec{n^{(i)}} \times \vec{cost} - e^{(i)})^2 + \lambda\frac{1}{l}\sum_{j=1}^{l}\rho^{-cost_j} \tag{7}$$

$$\rho > 1$$

$$cost_j := cost_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(\vec{n^{(i)}} \times \vec{cost}) \cdot n_j^{(i)} + \alpha\lambda\frac{ln(\rho)}{l}\rho^{-cost_j} \tag{8}$$

$$j = 1, 2, \cdots, l$$

5.5 Preliminary Results (Inference Accuracy)

The key point of case design is to vary the executions of individual blocks, so we are able to enlarge the column space of the N matrix (in Section 5.4) to raise the possibilities to solve all the values in $\vec{cost}$. We try to achieve it by commenting out different sets of blocks in each test case. With the data collected in training cases, we obtain the approximate $\vec{cost}$. It is not trivial to note that the session lengths of test cases are about 5s, those of training cases are rather various from 5s to 40s, so as to broaden the view of the model. The inference accuracy of $\vec{cost}$ in training and test cases is shown in Figure 5. The model fit the training cases quite well with an error margin of 2.2%. However, the error rate is unacceptably high in test, which is around 43.0%.

We find that the over-fit in train stage is because in most situations a set of energy ops are executed together, they are very hard to separate, for example, the `comparison` ops are always coupled with a `block-goto` op. To settle this, we apply an feature selection procedure. According to the training data, we put the op with strong linear execution correlations (above 0.9 and below 1.1) in the same group. The correlation is the covariance over the product of the standard deviations of two variables (op executions).

We then similarly treat one group of ops as one op to train the model. Figure 6 demonstrates that the inference error rate in training stage is higher than that before feature selection, about 7.2% and in test stage is much lower as 21.7%.

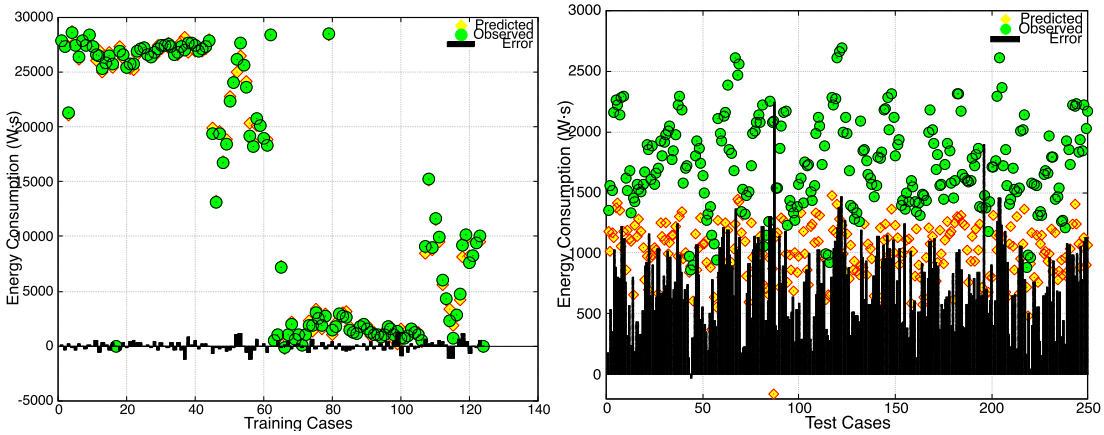


Figure 5: The predicted, observed energy consumption in training and test cases BEFORE feature selection. The bars show the errors of predicted values.

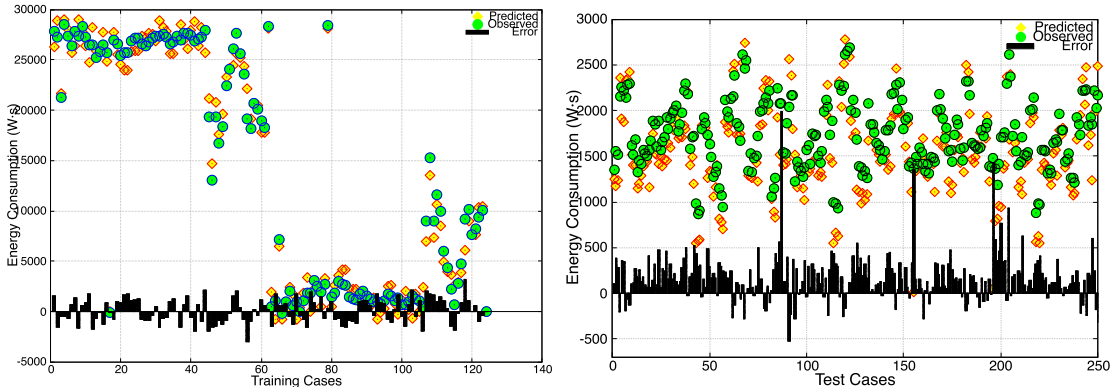


Figure 6: The predicted, observed energy consumption in training and test cases AFTER feature selection. The bars show the errors of predicted values

5.6 Dissemination

An article based on the work presented in this section has been submitted for peer-reviewed publication at a leading international conference venue.

Benchmark	Source	Description	Code structure & characteristics								
			NCSL	T	L	N	A	B	CP	MF	C
Base64	Online[1]	Computes the base64 encoding	32	1	✓		✓	✓			
Mac	MDH WCET	Dot product of two vectors and sum of squares	11	1	✓		✓				
Levenshtein	BEEBS	Measures the difference between two strings	26	1	✓	✓	✓	✓		✓	
Radix4Div	Online[2]	Optimized Software Division for platforms without hardware support	63	1	✓			✓	✓		
B. Radix4Div	Online, modified	Software Division for platforms without hardware support	37	1	✓			✓	✓		
Cnt	MDH WCET	Counts non-negative numbers in a matrix	29	1	✓	✓	✓				
Statistics	MDH WCET	Statistics program	85	1			✓	✓		✓	
FIR	XMOS	Finite Impulse Response filter	40	1	✓		✓	✓			
P. FIR_7T			103	7	✓		✓	✓			✓
MatMul	MDH WCET	Matrix multiplication of two square matrices	15	1	✓	✓	✓				
MatMul_2T			25	2	✓	✓	✓				
MatMul_4T			27	4	✓	✓	✓				
Biquad	XMOS	Signal equaliser using biquad filtering	49	1			✓	✓			
Biquad_2T			55	2			✓	✓		✓	
Biquad_4T			57	4			✓	✓		✓	
P. Biquad_7T			94	7			✓	✓		✓	✓
Jpegdct	MDH WCET	Performs a JPEG discrete cosine transform	35	1	✓	✓	✓	✓			
Jpegdct_2T			43	2	✓	✓	✓	✓		✓	
Jpegdct_4T			45	4	✓	✓	✓	✓		✓	

NCSL: Non-comment source-lines **T**: Number of threads **L**: Contains loops **N**: Nested loops **A**: Uses arrays/matrices **B**: Bitwise operations **CP**: Complex CFG structure **MF**: Multiple functions **C**: Contains thread communications.

[1] Retrieved from `http://stackoverflow.com/questions/342409`, Nov 2014.

[2] Retrieved from http://tinyurl.com/ld7exmd, Nov 2014.

Table 1: Description and attributes of benchmarks.

Table 2: Energy Operations

Arithmetic Ops	Addition, Subtraction Multiplication, Division Increment, Decrement
Boolean Ops	And, Or, Not
Comparison Ops	Greater, Less, Equal Greater or equal Less or equal
Bitwise Ops	BitAnd, BitOr SignedBitShiftRight SignedBitShiftLeft
Reference Ops	Array reference Field reference
Function Ops	Argument passing Returning value
Control Ops	Block goto Function Invocation
Others	Declaration Type conversion

Table 3: Library Functions

Class	Functions
ArrayList	add, get, size, isEmpty, remove
GL10	glBindTexture, glDisableClientState glDrawElements, glEnableClientState glMultMatrixf, glTexCoordPointer glPopMatrix, glPushMatrix glTexParameterx, glVertexPointer
Math	max, pow, sqrt, random
FloatBuffer	position, put

References

[ACM] Acm transactions on architecture and code optimization, at http://taco.acm.org/.

[Anda] Android. Android debug bridge. [Online: accessed 10.08.2015].

[Andb] Android. Dalvik virtual machine. [Online: accessed 10.08.2015].

[Edw11] Chris Edwards. Lack of software support marks the low power scorecard at DAC. In *Electronics Weekly.*, pages 15–21, June 2011.

[Fop] Foundational and practical aspects of resource analysis, at http://resourceanalysis.cs.ru.nl/fopara/.

[GGP+15] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Static analysis of energy consumption for llvm ir programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, New York, NY, USA, 2015. ACM.

[Goo] Google. Cocos2d-android. [Online: accessed 10.08.2015].

[KE15a] S. Kerrison and K. Eder. Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Transactions on Embedded Computing Systems*, 14(3):1–25, April 2015.

[KE15b] S. Kerrison and K. Eder. Modeling and visualizing networked multi-core embedded software energy consumption. *ArXiv e-prints*, September 2015.

[LA04] C. Lattner and V.S. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 75–88. IEEE Computer Society, March 2004.

[LGK+15] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Energy Consumption at Different Software Levels: ISA vs. LLVM IR. In *Proc. of the Foundational and Practical Aspects of Resource Analysis*, LNCS. Springer, 2015. To Appear.

[LKS+14] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M.V. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Logic-Based Program Synthesis and Transformation,*

23rd International Symposium, LOPSTR 2013, Revised Selected Papers, volume 8901 of *Lecture Notes in Computer Science*, pages 72–90. Springer, 2014.

[LLV14] LLVMorg. The LLVM Compiler Infrastructure, November 2014.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 88–98, 1995.

[Mar15] Marketing Land. Report: U.S. smartphone penetration now at 75 percent, 2015. [Online: accessed 10.08.2015].

[Ng12] Andrew Ng. CS229 lecture notes, 2012. [Online: accessed 10.08.2015].

[Odr] Odroid. Odroid-XUE. [Online: accessed 10.08.2015].

[Ora] Oracle. Java virtual machine instruction set. [Online: accessed 10.08.2015].

[PKME15] James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Data dependent energy modelling: A worst case perspective. *arXiv preprint arXiv:1505.03374, Submitted to PATMOS 2015, under review*, 2015.

[Sco] 18th International Workshop on Software and Compilers for Embedded Systems, at http://www.scopesconf.org/scopes-15/.

[TMW94] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, Dec 1994.

[vBDMH00] Tajana Šimunić, Luca Benini, Giovanni De Micheli, and Mat Hans. Source code optimization and profiling of energy consumption in embedded systems. In *Proceedings of the 13th International Symposium on System Synthesis*, ISSS '00, pages 193–198, Washington, DC, USA, 2000. IEEE Computer Society.

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D.B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P.P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - Overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[XMO14] XMOS. xTimecomposer, November 2014.

Attachments

The attachments referenced throughout this deliverable are included in this appendix.

Attachment D2.3.1

Modelling software energy consumption in a multi-core network of embedded multi-threaded processors

In draft, awaiting submission.

Modeling and visualizing networked multi-core embedded software energy consumption

Steve Kerrison and Kerstin Eder
University of Bristol, United Kingdom
firstname.lastname@bristol.ac.uk

Technical Report, August 2015

Abstract

In this report we present a network-level multi-core energy model and a software development process workflow that allows software developers to estimate the energy consumption of multi-core embedded programs. This work focuses on a high performance, cache-less and timing predictable embedded processor architecture, XS1. Prior modelling work is improved to increase accuracy, then extended to be parametric with respect to voltage and frequency scaling (VFS) and then integrated into a larger scale model of a network of interconnected cores. The modelling is supported by enhancements to an open source instruction set simulator to provide the first network timing aware simulations of the target architecture. Simulation based modelling techniques are combined with methods of results presentation to demonstrate how such work can be integrated into a software developer's workflow, enabling the developer to make informed, energy aware coding decisions. A set of single-, multi-threaded and multi-core benchmarks are used to exercise and evaluate the models and provide use case examples for how results can be presented and interpreted. The models all yield accuracy within an average ±5 % error margin.

1 Introduction

An increasing number of embedded systems now express their workloads through concurrent software. The parallelism present in modern devices, in forms such as multi-threading and multiple cores, allow this concurrency to be exploited. This progression towards parallel systems has two main motivations. The first is in response to hitting operating frequency limits, where more work must now be done per clock in order to achieve performance gains in each new device generation. The other uses parallelism to allow work to be completed on time at a lower operating frequency, which can yield significant energy reductions.

However, parallel systems and concurrent software introduce complexities over traditional sequential variants that simply valued "straight-line speed". In particular, synchronisation of and exchanging information between concurrent components can negatively impact parallel performance if done inefficiently as per the well known *Amdahl's Law*. A good understanding of the software's behaviour, coupled with appropriate underlying hardware can overcome this if used correctly.

In embedded systems software, predictability is essential, both in terms of execution time, where real-time deadlines must be met, and in terms of energy consumption, where the supply of energy may be scarce. Time and energy are related through power, and while significant effort is put into timing predictable software, there remains both a lack of intuition and a lack of tools to help software developers determine the energy consumption of their modern embedded software components.

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 318337, ENTRA - Whole-Systems Energy Transparency.

This report presents an energy model for a family of cache-less, time-deterministic, hardware multi-threaded embedded processors, the XMOS XS1-L series, which implements the XS1 architecture. These processors are programmed in a C-like language with message passing present in both the architecture and the programming model. The processors can be assembled into networks of interconnected cores, where the communication paradigm then extends across this network. The energy model must therefore be able to account for software energy consumption within each core as well as the timing and power effects of network traffic. To achieve this and also give developers better energy estimation tools, the following contributions are made:

- A multi-threaded energy model for the XS1-L [15] is extended to include more accurate instruction energy data, through greater instruction profiling and regression tree techniques.
- Support for Voltage and Frequency Scaling (VFS) is integrated into the model, the provide a richer environment for design space exploration by software developers.
- Several new features are added to `axe`, an Instruction Set Simulator (ISS) for the XS1-L, improving its core timing accuracy and introducing network timing behaviour, which has until now not been present in any simulators for these devices.
- The energy consumption of network communication is profiled, in order to extend the energy model to account for communication between multi-cores.

These contributions allow traces from the `axe` ISS to be analysed by the modelling framework, producing both text reports and visualisation of energy consumption across the network of processors in the system. The accuracy of this work is established through a series of multi-threaded, multi-core embedded software benchmarks. These are used to evaluate the effectiveness of the modelling and detail how it

can be used to aid a developer's design and implementation decisions.

Results show that the core model average error is 2.67 % with a standard deviation of 4.40 %, improving upon the prior work. The network capable model demonstrate an average error of −4.92 %, with a standard deviation of 3.92 %, supported by the VFS model with a mean squared error of 2.60 % and total error range of 15.72 %. The network model is shown to be suitable for determining the best approach for implementing two concurrent signal processing tasks on a target dual-core XS1-L platform.

Structure

The rest of this report is structured as follows. Related work is presented in Section 2, which looks at energy modelling of modern embedded processors, multi-core communication techniques and parallelism in embedded architectures, and summarises the particular implementations used in the XS1-L processor. The core- and network-level energy models are explained in Section 3, then the necessary ISS changes to support the model are presented in Section 4. Results from benchmarks exercising various parts of the model and simulation framework are discussed in Section 5 along with an evaluation of their performance in terms of accuracy and usability. Finally, Section 6 draws conclusions from this research and proposes future work.

2 Related work and background

Energy modelling of software is motivated by a need to reduce global ICT energy consumption as well as to enable devices such as embedded systems to provide more features and last longer on limited source of energy. Although hardware actually consumes energy, it does so at the behest of software, which can be inefficient if the software does not fit well to the target hardware, or does not allow the hardware to exploit its own energy saving features [20].

Multi-core systems have proliferated through ICT, from servers in datacenters down to smart phones, and now even deeply embedded systems. Any endeavour to provide software energy consumption metrics must therefore be multi-core aware. In the rest of this section we discuss related work in three background areas. First is multi-core processors in embedded systems, next is energy modelling of processors, with a focus on software level energy consumption, and finally we introduce the XS1-L processor, the particular micro-architecture used as a case study for this research.

2.1 Parallelism and multi-core embedded processors

There are various ways of realising parallelism in processors. In embedded systems, many methods have been used. VLIW (Very Long Instruction Word) has been used for some time, particularly in DSPs (Digital Signal Processors), where instruction packets enable software pipelining to be parallelised.Multi-core is becoming more prevalent, where it is beneficial to replicate a core several times and distribute work between cores. This has become necessary to provide performance gains as frequency increases have become harder to realise within practical power budgets [13].

High performance embedded processors, such as those found in smart phones, can feature multiple cores with different micro-architectures. ARM's *big.LITTLE* is the seminal example of this, where programs can be scheduled onto simpler cores when low-energy operation is necessary or appropriate. The *little* cores are slower, but can be operated at a lower voltage and frequency point than their *big* counterparts, consuming significantly less energy. In big.LITTLE, significant effort is put into cache coherency between the cores, and migrating tasks can require flushing and copying of core-local caches in order to keep consistent state.

Smaller processors, such as ARM's Cortex-M series, can also be used in multi-core, but the implementation is defined by the manufacturer. ARM has made recommendations on how to construct such devices, including cache and memory arbitration mechanisms [27]. Older generation ARM9 processors have been assembled in their thousands in the *SpiNNaker* system [5].

Rather than connecting processors via a cache hierarchy and memory bus, some systems implement a network of cores. Devices such as the Adapteva Epiphany [1] and EZChip TILE [4] processors feature many cores in a Network-on-Chip, with a grid topology of interconnects between them. In both of these processors there are multiple networks, each serving a unique purpose, such as I/O, cache coherency and direct inter-tile communication. The Intel Xeon Phi [12] uses a ring network and a hierarchy of processors, caches, tag directories and memory controllers to create a NoC that can also be viewed as a traditional memory hierarchy. Its use is not in embedded systems, but rather as a high performance computing accelerator.

The XS1-L processor features no cache hierarchy and can be assembled into a network of cores where channel style communication is possible both on- or off-chip. This is discussed in more detail in Section 2.3.

2.2 Energy modelling of processors

A program's energy consumption is the integral of a device's power dissipation during the course of execution:

$$E = \int_{t=0}^{T} P(t)\ dt, \tag{1}$$

although this is frequently represented using an average power, giving $E = P \times T$. To energy model a processor, P must be estimated over the course of T with sufficient granularity and precision to provide a desired accuracy. At the hardware level, detailed transistor or CMOS device models can be used, and every change in circuit state simulated to determine a fine-grained power estimation. This is time consuming and requires access to the RTL description of a processor, making this form of analysis infeasible for software developers.

Higher level models can be used instead, such as those modelling the processor as functional blocks. Instructions issued by the processor trigger activity in the functional blocks, and a cost is associated with that, which can be used to estimate the energy consumption of a sequence of instructions. At this level, the instructions are an essential part, as these drive the modelling, but also form a connection to the software — the instruction sequences for a given architecture are related to the software developer's

program via transformation by a compiler. The ISA therefore provides a good level at which to perform analysis of hardware energy consumption at the behest of software.

Seminal work in ISA level energy modelling includes that of Tiwari et al. [26], where sequences of instructions are assigned costs, as well as the transitions between instructions, which causes circuit switching as new control paths are enabled. This work has been drawn up upon to enable energy consumption simulation frameworks such as Wattch [3] and SimPanalyzer [24]. This style of ISA level modelling has also been refined to include finer grained detail on the activity along the processor data path, where data value changes also influence energy consumption [25].

Energy modelling has been performed for a wide variety of processors with various micro-architectural characteristics, for example VLIW DSP devices [10], both simple and high performance ARM variants, as well as very large processors such modern server grade x86 devices [8] and the 61 core Xeon Phi [23]. These all draw from similar background, but account for different processor features, and obtain their model data from different sources. For example, high performance ARM and x86 models can use hardware performance counters to model activities such as cache misses, which have a significant impact on energy consumption. Simpler devices may not be so affected, and thus direct instruction level costs can be attributed. Parametrised energy models that consider properties such as operating frequency and voltage have been created for other processors, such as the Intel Xeon in [2], to inform a model predictive controller in order to smooth thermal hotspots in such dense multi core devices.

A single core model of the XMOS XS1-L architecture is presented in [15], which uses data from a series of instruction energy profiling tests in order to build the model. The architecture's hardware multi-threading is accounted for, with the level of parallelism (active threads) contributing to energy consumption during the course of the analysed program. This model has been applied using instruction set simulation, and also via static analysis at the ISA level [7, 16] as well as the LLVM IR level [6].

2.3 The XS1-L processor and network

The XS1-L family is a group of processors implementing the XMOS XS1 ISA in a 65 nm process technology, featuring a configurable network upon which arbitrary topologies of interconnected processors can be built. Each core has a four stage pipeline and support for up to eight hardware scheduled threads. A thread can have no more than one instruction in the pipeline at any given time, therefore the XS1-L parallelism is only fully utilised if four or more threads are active.

These processors include 64 KiB of single cycle SRAM and have no cache, therefore the memory subsystem is flat and requires no special considerations with respect to timing. The majority of instructions complete in four clock cycles, with the exception of the divide and remainder instructions and any instructions that block on some form of I/O. If more than four threads are active, then the instruction issue rate per thread will reduce proportionally, but the instruction throughput of the processor remains the same. This makes timing analysis of the processor very

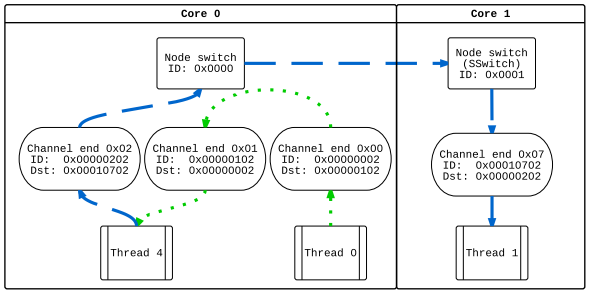


Figure 1: Visualisation of channel based communication between threads both locally and between cores.

predictable, allowing tight bounds or even exact values to be produced.

The XS1 instruction set includes provisions for *resource operations*. These are interactions with peripheral devices, such as I/O ports, synchronisers and communication channel endpoints (chanends). As such, activities such as I/O are a first class member of the instruction set. Other instruction sets, such as x86, have similar provisions [11, pp.115,176]. However, the XS1 architecture takes this further, and places these peripherals outside of the memory space, such that I/O and other resource operations are not translated into memory mapped reads and writes, but are instead a completely separate data path. This separation of memory and resources aids in the modelling processor, particularly when communication between threads and cores is considered.

On a single core, it is possible for threads to communicate or access common data using shared memory paradigms. This can be expressed in software through appropriate use of regular pointers in C, or through specially attributed pointers in version 2 of the *XC* language that was developed to complement XS1. However, CSP style channel communication is more prevalent in XC. Channels in XC translate into channel endpoints in the XS1 architecture, where two chanends are logically connected together. Communication then takes the form of `in` and `out` instructions. Control tokens can be used to provide synchronisation, and instructions will block if buffers are full or no data is available to read. This paradigm extends beyond core-local communication and out onto a network of cores. Therefore, concurrent programs can grow to use multiple processors with relative ease.

A network of XS1-L processors consists of multiple cores each connected to their own integrated switch. This switch provides a number of links, which can be connected to other switches, either on- or off-chip. These links can operate in either five- or two-wire mode in each direction, where the former can carry two bits per symbol and the latter one bit per symbol in an 8b/10b encoding. The five wire mode is therefore faster at the same frequency, but requires ten wires total per link. Each link possesses a receive buffer and credit based flow control is used to prevent overrun. When a link is first enabled, the sending switch must solicit credit from the switch at the other end of the link with a *hello token*. During normal operation, *credit* tokens are

sent from the receiver to the sender as buffer space becomes available.

Routing between switches is configurable based on IDs assigned to each node, where a node is a switch and its associated core. When a message is first sent from the chanend of a core, the ID of the destination node is prepended to the message. Receiving switches then compare this ID to their own. If they are different, the first bit that is different is used to determine the *direction* along which the message will be routed. A direction can be assigned one or more links, and the next available link in that direction will then be used for forwarding. Typically, dimension-order routing is used to create a deadlock avoiding network, but this is dependent upon the topology network that is physically assembled. Links are held exclusively by the source chanend either indefinitely, or until a closing control token is transmitted from the source. Through this approach, both wormhole routed packets and permanently reserved streaming routes can be created.

A high level view of threads communicating through chanends and switches, both locally and between cores is shown in Figure 1. The precise implementation details and configuration parameters are detailed in [18, 19]. Examples of multi-core XS1 implementations include the XMP-64, which features 64 cores, using the older XS1-G family, and the Swallow project, which assembles multiple dual-core XS1-L family processors into a system of hundreds of cores [9]. These use hypercube and lattice network topologies, respectively.

3 XS1-L core and network energy model

In this report, the modelling effort of [15] is extended in several ways. Firstly, more instructions are directly energy profiled, and for those that cannot, a regression tree approach is implemented to estimate their energy cost. Secondly, additional voltage and frequency profiling is performed, using a suitable variant of the XS1-L, to produce a VFS aware model version, retaining good error bounds. Finally, network communication costs are considered, through further profiling, and a network level, communication ware model produced, integrating core, switch and interconnect components within the model.

3.1 Regression tree

The prior work of [15] investigated grouping instructions by operand count in order to provide an energy estimate for un-profiled instructions, as well as to reduce model complexity. However, the evaluation showed that this was not suitably fined grained or sufficiently accurate. Instead, each profiled instruction is accounted for individually, and un-profiled instructions are assigned a *default* value, based on the observed average of all profiled instructions.

Here, a different approach is used, where a set of instruction features are used to classify each instruction. This is combined with the direct instruction profiling data into a regression tree, allowing un-profiled instructions to receive an energy estimate based on profiled instructions with similar features.

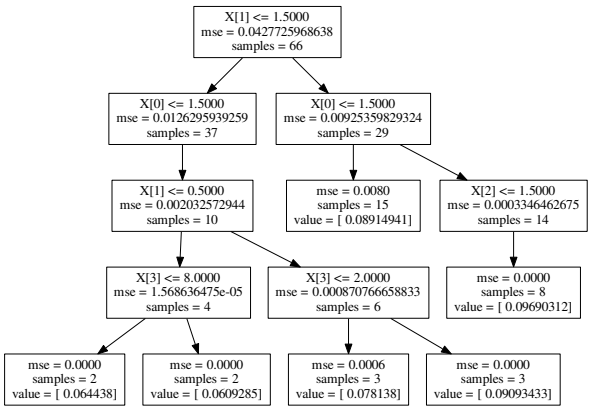


Figure 2: Visualisation of part of the model regression tree. Leaves provide energy estimations, all other nodes are decisions based on a particular instruction feature $X[f]$. Not all branches are shown; the full tree is 29 nodes.

Instr.	Features						Energy
	L	**S**	**D**	**I**	**M**	**R**	
`add_3r`	1	2	1	0	0	0	185 mW
`ldc_lru6`	2	0	1	10	0	0	160 mW
`outct_rus`	1	1	0	4	0	1	134 mW

Table 1: Input data for regression tree constructor

3.1.1 Tree construction

First, a set of features are identified, which from empirical data, demonstrate a correlation with energy consumption. These are specific to the XS1-L processor, although can be re-defined for other processors in order to re-use the technique. In the case of the XS1-L, the features are:

L: Instruction length (short or long: 1 or 2).

S: Number of source registers (count: 0–4).

D: Number of destination registers (count: 0–2).

I: Length of immediate operand (num. bits: 4–16).

M: A memory operation is performed (Boolean).

R: A resource operation is performed (Boolean).

The *Scikit-Learn* `DecisionTreeRegressor` [21] is used to build the regression tree. The data is presented as an matrix of instruction features and a vector of measured energy for each profiled instruction. From this, a regression based decision tree is constructed. A sample of the input data is provided in Table 1.

The regression tree construction library uses floating point feature parameters. For the given feature set, both the integer and Boolean features can be converted to their nearest floating point equivalent without consequence.

3.1.2 Tree traversal

A cutting of the regression tree for our energy model is depicted in Figure 2. When the energy cost of an instruction must be determined, a check first determines if a direct energy measurement exists. If so, it can be used within the model equation. If not, then the instruction's features are used to traverse the decision tree. Each feature is indexed numerically by the `DecisionTreeRegressor`, and map to the features in the order we have declared them.

For example, the first decision, at the root of tree, is dependent upon the number of source operands. Those with fewer than two (or ≤ 1.5) follow the left branch, whilst those with two or more follow the right branch. The instruction length is considered next. However, descending into the tree further, the feature selection that minimises the mean squared error (mse), will differ depending on the instruction and the collected energy data. This makes the decision tree more versatile than a flat ordinary least squares regression. For example, there are no instructions with four source operands that use memory, therefore $X[4]$ or feature $\mathbf{M}$ has no influence upon such instructions. The tree is also unbalanced; some branches reach leaves in fewer levels, due to no variation in features beyond a certain decision point.

The accuracy of this approach is tested and evaluated in Section 5, where a reduction in both error and variance is shown when compared to the previous model.

3.2 VFS modelling

The XS1-L series of processors can dynamically adjust their core clock frequency when idle, and some devices support variable voltage. The low speed of voltage adjustment makes dynamic voltage and frequency scaling (DVFS) impractical for most of the real-time embedded tasks targeted to the XS1-L. However, it is still possible to statically select a best voltage and frequency for a given set of tasks, and so there is motivation to provide an energy model can support this exploration in order to determine what savings can be made.

An XMOS *SLICEKIT-A16* is used for VFS profiling, a board containing an XS1-A16 processor. The A16 contains two XS1-L family processor cores, as well as an analogue component block containing components such as ADCs and most importantly configurable DC-DC power supplies, one of which services the cores. The SLICEKIT-A16 board provides two shunt resistors for power sensing, one for the 3.3 V I/O used by the chip and one for the 3.3 V supply fed to the on-chip voltage regulators. Measurements are performed with a MAGEEC power measurement board [17], which provides 2 MSPS at 12-bit resolution with a noise floor at approximately 0.1 % of measured power in our test setup, depending on the current supplied. The measurement setup and power supply structure is different to that used in [15], so the power supplies must be considered in the new model.

3.2.1 Profiling method

VFS profiling is performed by a series of tests both at idle and high power, where each test is performed at different voltage and frequency operating points. Three configurable parameters are exercised:

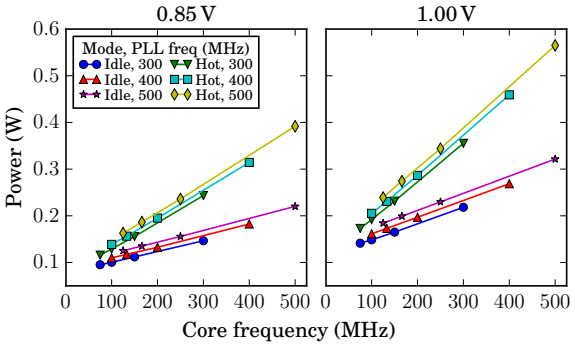


Figure 3: Power measurements at two voltage points for idle and high power tests over a range of system frequencies and dividers.

System frequency The frequency produced by the PLL, which is integrated into each processor's switch. This sets the frequency of the switch and core, each of which can then be divided further.

Core divider The divider applied to the system frequency to produce the core frequency. Typically this is zero. The specified divider can be applied dynamically when there are no active threads, or permanently.

Core voltage The power supply to the device's cores is configurable in 10 mV steps.

Other parameters, such as the reference clock, can be also be changed. However, the reference clock is used for timing ports and other synchronisation activities, thus we keep it at its default of 100 MHz to prevent unexpected timing changes in programs. As a result of this, profiling is limited to system frequencies of 500, 400 and 300 MHz and a core divider in the range 0–3. Error free operation was achieved with a core voltage range of 0.85–1.15 V. However, the vendor only certifies devices for operation at 1.0 V, and extensive testing at each voltage point was not performed; they were used purely for VFS characterisation.

3.2.2 VFS profiling data

Figure 3 shows the profiling data for two of the voltage points that were tested. Each plot shows six series; three for idle tests and three for high power tests, each at one of three system frequencies. Points along the x axis determine the core frequency after the divider is applied.

The 100 MHz operating point is achieved twice during tests, with $F = \frac{400}{4}$ and $F = \frac{300}{3}$. From this we see that there is an overhead in having a higher system clock, regardless of the resultant core clock. This is intuitive, as there is still some part of the system operating at the higher frequency.

3.2.3 Energy model

To produce a VFS capable energy model, we incorporate the configurable parameters defined in Section 3.2.1 into a suitably modified model equation. Curve fitting is used to

determine the contribution that these parameters have to the equation.

SciPy's Nelder-Mead method [22] is used to minimize the error of the function Equation (2) against the idle test profile data collected, for the following parameters. C_{pll} is the characteristic capacitance present at the system frequency (or PLL frequency). C_{idle} is the characteristic capacitance in the core at idle. I_{leak} is the static leakage current. Finally, I_{ext} captures other effects parametric to the supply voltage and scaled by the power dissipated in the device, approximating power supply efficiency.

$$F = (V^2 C_{\text{pll}} F_{\text{pll}} + V^2 C_{\text{idle}} F_{\text{core}} + V I_{\text{leak}}) \times V I_{\text{ext}} \quad (2)$$

The resultant parameters are:

$$C_{\text{pll}} = 675 \times 10^{-12}, \qquad C_{\text{idle}} = 1.68 \times 10^{-9}$$
$$I_{\text{leak}} = 334 \times 10^{-3}, \qquad I_{\text{ext}} = 106 \times 10^{-3}. \quad (3)$$

A parameter is also determined for the high-power tests, $C_{\text{hot}} = 2.15 \times 10^{-9}$, although it is used only for validation, and not in the final energy model. Testing these parameters against the profiling data, a mean squared error of 2.60 % is achieved. The minimum error is −3.58 % and the maximum 12.14 %, giving a full error range of 15.72 %.

These parameters are then used in a modified version of the instruction level energy model from [15]. This yields Equation (4) as the new model:

$$\begin{aligned} E_{\text{instr}} = (\ & V^2 C_{\text{pll}} F_{\text{pll}} \\ & + V^2 F_{\text{core}} (C_{\text{idle}} + C_{\text{instr}}) M_{N_{\text{pipe}}} O \\ & + V I_{\text{leak}}\ ) \times V I_{\text{ext}} \times 4 T_{\text{clk}} \quad (4) \\ \text{where} \quad M_{\text{pipe}} = & \min(4, N_t). \end{aligned}$$

This captures the previous components of the model, plus the frequency and voltage dependent parameters. N_{pipe} is the number of threads present in the pipeline when the instruction's energy is measured, which is the minimum of 4 and the number of active threads. This is used select a scaling factor due to parallelism, M. An average inter-instruction overhead, O, is also included, as per the original model.

The remainder of this report focuses more on network aware energy modelling, rather than VFS design space exploration. Future work could include exercising the VFS aspect of the energy model more heavily, and so is included here for the benefit of such endeavours.

3.3 Network modelling

A system level model of a network of XS1-L processors is comprised of multiple core model instances, as well additional modelling components to capture network switch and link activity. The core model requires either simulated instruction sequences or appropriately parametrised static analysis. A system level model must support the interaction between multiple cores. The implementation details of this at a simulation level, are covered in Section 4.

3.3.1 Parameters

Communication costs must be accounted for in three system components. Firstly, the core, where the `in` and `out` instructions are executed. These are already captured in the core energy model. Second is the switch, which consumes energy as it routes tokens through it. Finally, the interconnects over which tokens are transmitted must be considered.

Switch energy consumption data is acquired from profiling of the larger Swallow XS1-L system [14, p. 124]. Link energy for the SLICEKIT-A16 is determined from direct profiling. These are shown in terms of Joules per token in Equation (5).

$$E_{\text{switch}} = 70.8 \times 10^{-12}\,\text{J}, \quad E_{\text{link}} = 221 \times 10^{-12}\,\text{J} \quad (5)$$

Currently, these are fixed values. However, it is possible to parametrise these by link length (where longer wiring has a higher capacitance), as well as by switch frequency and voltage. This may form future work, joining well with the proposed further work on VFS modelling.

3.3.2 Construction

The network level model is constructed using the `NetworkX` library for Python, which allows networks with nodes and edges that have arbitrary attributes. The XML file used by the developer or vendor to describe an XMOS based system (the XN file), is read by the energy modelling framework and used to construct a graph of the system's cores, switches and links.

When a simulation trace is analysed by the modelling tool, energy is incremented in each graph node or edge as appropriate. Instructions increase the core energy of the relevant core, whilst token traces increase the source switch and traversed link energy. For this trace analysis to account for network activity, the trace must include network activity that identifies tokens traversing links. This change was made to `axe` as part of the modifications described in Section 4.

At the end of the modelling run, this data can be aggregated into a text report, broken down by core, or as a visualisation. These will be shown in Section 5.

4 ISS network and timing implementation

XS1-L energy modelling has been demonstrated using statistics from instruction set simulation as well as various levels of static analysis. Using full instruction set simulation traces provide more detail, at the cost of simulation time. However, by improving analysis of traces to complete once a *function or section of interest* has completed, simulation time can be kept low. The same triggering methods used by the hardware measurement, can easily be used to define sections of interest by identifying the relevant I/O resource instructions in a trace. This means single iterations of functions or algorithms can be observed by modelling, where repeated iterations are required for physical measurement. The slowdown of simulation is mitigated to some degree by this. This is mitigated further by the use of `axe`, an open source XS1 simulator that is faster than its closed source `xsim` counterpart, although it can be less accurate. A number of `axe` enhancements are detailed in this section that improve its accuracy, whilst preserving some of its performance advantage.

Enabling full trace simulation allows better debugging of the energy model, as well as the opportunity to more closely scrutinise where energy is being consumed. To that end, this work focuses on full traces. However, the models underpinning this work can be adapted for use at other levels of abstraction, as with previous model versions.

4.1 Instruction scheduling

The modified version of `axe` enforces strict instruction scheduling, where each active thread may only issue one instruction before the next queued thread is given an opportunity. The timestamps of subsequent instructions in a thread are incremented by $\min(4, N_t)$, to reflect the four-stage, hazard free pipeline.

This more closely follows the micro-architecture, whereas the original `axe` implementation may issue multiple instructions from one active thread even when another thread is also in an active state. This also ensures that the timestamps in instruction traces are ordered, greatly simplifying the process of determining pipeline occupation during energy modelling.

4.2 `FNOP` simulation

In addition to instruction scheduling changes, occurrences of fetch no-ops (`FNOP`s) are also recorded in the modified simulator. A simple model of the processor's instruction buffers is used to determine when a thread must stall in order to fetch the next instruction word. The conditions leading to an `FNOP` include:

- Sequences of memory operations in a thread, preventing any instructions being fetched for that thread during the memory stage of the pipeline.
- Branching to an unaligned 4-byte instruction, where only the first half of the instruction is fetched during the memory stage of the branch instruction.

Many `FNOP`s can be avoided by re-scheduling long instructions and memory instructions amongst short, non-memory instructions, as well as word aligning entry points to loops where the first instruction is long. However, the compiler does not currently do all of these automatically. The impact of `FNOP`s can be significant if they are present in tight loops, increasing execution time and therefore energy. Thus, is it important to correctly simulate this behaviour for accurate energy modelling.

4.3 Switch and link control flow

Both the `axe` and `xsim` XS1-L instruction set simulators support channel communication in multi-core programs. However, even with accurate core-local instruction scheduling, neither include accurate simulation of network behaviour. At a functional level, link utilisation and route reservation are simulated, such that protocol violations can be detected and appropriate exceptions raised, as well as some degree of performance limiting due to route contention. However, multi-core message tokens are transmitted in zero time. This can create significant timing error when simulating communicating multi-core programs.

To address this, we have added a model of the link control flow mechanisms from XS1-L into `axe`'s switch, link and channel end code. Control tokens such as `HELLO` and the initial `CREDIT` issue [19, pp. 10–13] are now handled rather than ignored. Symbol and token delays on links, as specified in the XN platform description file at compile time, are also obeyed. This ensures that messages traverse the network in a realistic time-scale, and that as buffers fill, network throughput is throttled.

Test	Recorded time (µs)			Error (%)	
	HW	`xsim`	`axe`	`xsim`	`axe`
One core	2.270	1.118	2.272	-50.75	0.09
Two core	8.300	2.150	8.287	-74.10	-0.16

Table 2: 1024-word message timing, comparing dual-core hardware to `xsim` and modified `axe` simulators.

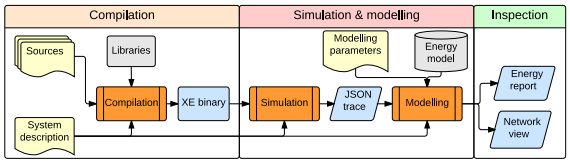


Figure 4: Energy aware multi-core software development workflow.

The current changes are not completely faithful to the hardware, but yield a significant improvement on the previous simulation capabilities. This is evident in Table 2, which compares a 1024-word transfer between two channel ends on a SLICEKIT-A16 in both core-local and dual-core variants, with respect to the actual hardware timing, `xsim` simulation and the modified `axe` simulation. The error in `xsim` exceeds 50 % in both cases, whereas `axe` achieves less than 0.2 %. Over a broader range of similar tests, with different message lengths and producer/consume rates, `axe` is able to maintain an average error of 0.80 % with a standard deviation of 1.26 %.

To aid modelling, switch and link activity are added to `axe` simulation traces. These resemble the switch tracing present in the vendor's `xsim` simulator with the `--tracing-switch` parameter set, but are in a JSON format that is more readily consumable by the energy modelling framework.

5 Benchmarking and evaluation

This evaluation considers enhancements to the core model as well as the multi-core communication model. However, VFS modelling, as discussed in Section 3.2, remains for future work, due to the current instruction set simulation framework not supporting configurable operating frequencies and clock dividers without significant further development. This does not exclude the VFS model from use in other forms of non-simulation based analysis, however.

5.1 Core benchmarks

The extended core energy model is evaluated using the same benchmark suite as the original model in [15] and on the same single core XS1-L hardware. This provides direct

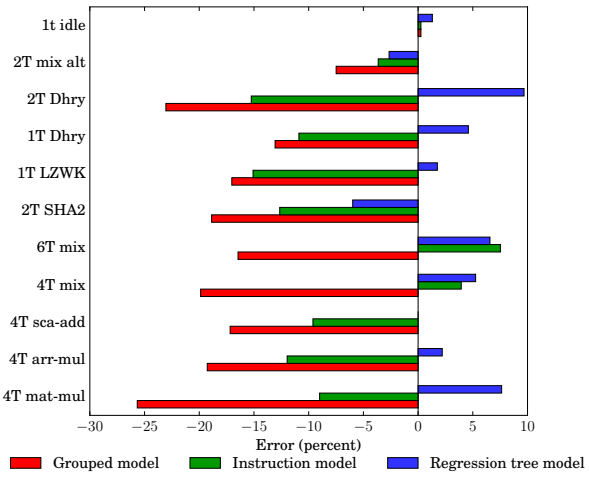


Figure 5: Error of previous (grouped, instruction) models versus new (regression tree) model.

Model version	Error (%)	Std. dev. (%)
Grouped	−16.42	6.91
Instruction	−7.23	7.45
Regression tree	2.67	4.40

Table 3: Geometric mean model error and standard deviation of the tested energy models.

comparison between the accuracy of both model versions with respect to the target hardware. The benchmarks used include the system at idle, various audio sample mixing variants, operating on multiple independent streams with various levels of concurrency, one and two concurrent Dhrystone instances, as well as multi-threaded parallel matrix operations. They are explained in more detail in [15, pp. 20–21].

Figure 5 shows the model error for each benchmark with respect to the hardware measured energy consumption. The regression tree model performs better than both of the previous model versions in the majority of benchmarks. Where the original instruction model out-performs the regression tree model, the difference is approximately a single percentage point of error. The average and standard deviation of the errors is summarised in Table 3, where it is evident that the regression tree model improves overall accuracy whilst reducing variance and the overall range of error across benchmarks.

5.2 Multi-core benchmarks

To test the multi-core model, suitable benchmarks must be used. Those used to test the core model do not lend themselves to multi-core deployment, due to their structure and limited, if any, use of channel communication. Instead, two new applications are used for multi-core benchmarking. These are a Finite Impulse Reponse filter (FIR), and the Infinite Impulse Response (IIR) Biquad filter, which will be referred to `fir` and `biq` respectively.

Both of these benchmarks are used for applying signal processing, in this case to streams of audio samples. This is

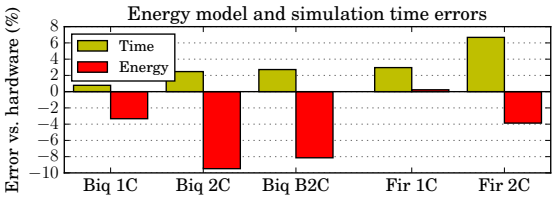


Figure 6: Time and energy errors for `fir` and `biq` benchmarks in single (1C) dual core (2C) and for `biq`, bad dual core (B2C) configurations.

Property	G. mean (%)	Std. dev. (%)
Time	3.10	2.16
Energy	-4.92	3.92

Table 4: Geometric mean and standard deviation of timing simulation and energy model errors for all `biq` and `fir` benchmarks.

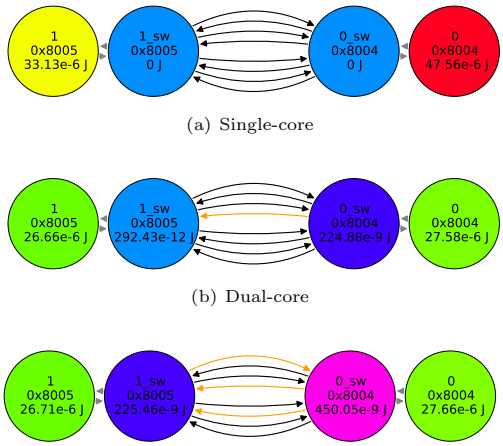
a typical application area for the target processor, and so a good benchmark selection. Both benchmarks feature multiple stages, `fir` implementing seven taps and `biq` featuring seven individual biquads.

The concurrent implementation of these applications represent each stage as a thread, with the progressively filtered samples passed over channels between threads. The seven threads can be allocated onto a single core the SLICEKIT-A16, or distributed across the device's two cores. We test both $7:0$ and $4:3$ thread distributions between the two cores, giving consideration to the fact that both thread and processor instruction throughput are optimal when a core has four active threads. We also implement a poorly allocated version of `biq` where threads communicate between cores sub-optimally.

In Figure 6 the energy and timing errors for the benchmarks are presented. We observe that in all cases, the simulation over-estimates execution time, but by less than 7 %. the energy model under predicts in the majority of cases, but remains within a 10 % error margin.

The overall results are summarised in Table 4, which demonstrates single-digit percentage mean and standard deviations for the errors. Note that the timing over-prediction counteracts the energy model under-prediction. Improving one in isolation may in fact reduce the visible accuracy of the process overall. It is therefore essential to examine the multiple dimensions of error that are present, in order to direct effort appropriately.

Figure 7 shows a visualisation of energy consumption in the cores, switces and interconnect of the SLICEKIT-A16. Cores are annotated with the modelled energy consumption, and switches show their energy consumption as well as the aggregated energy consumed on outbound links. Links are also coloured by energy. The colouring of the graphs has been scaled to be directly comparable. Hot/cold are represented as pink/blue for switches and green/red for cores. Links turn orange as they consume more energy. Only links between switches are energy modelled, as the



(a) Single-core

(b) Dual-core

(c) Dual-core with poorly allocated threads

Figure 7: Network level energy consumption visualisations the `biq` benchmark.

core to switch links are captured implicitly within the core model.

Although visualisation is a less precise representation, it does allow for comparison and inspection in order to determine *where* energy is consumed. From these examples, we see that the single core implementation in fig. 7(a) is the least efficient, taking more energy on the active core, and resulting in significant energy consumption from leakage in the otherwise idle core. In fig. 7(b), the benchmark completes quicker and work is distributed, so the cores consume less total energy. The communication cost is insignificant in comparison; some three orders of magnitude less. Finally, fig. 7(c) is again dual core, but allocates the software pipeline stages poorly, resulting in three times more core-to-core communication. The cores consume slightly more energy due to a marginally longer run-time, and the network cost is three times higher. Not only does this demonstrate the desirability of distributing the workload across the available cores, it also demonstrates that energy inefficiency can be introduced with minimal timing impact, where communication latency may be hidden.

6 Conclusions and future work

In this work, a single core, multi-threaded energy model is presented with an average error of less than 5 %. This is enabled through both instruction set simulator enhancements and a regression tree approach to modelling instructions that cannot be directly energy profiled.

A multi-core model is then described and tested, again supported by instruction set simulation enhancements. The timing error of the simulator is shown to be within 7 % and the energy estimation error within −10 % for two multi-core audio filtering benchmarks, with average errors of 3.10 % and −4.92 % respectively.

This combination of tools and the demonstrated workflow allows for multi-threaded, multi-core software design space exploration, in order to establish which software definable properties, such as thread allocation and communication patterns, impact the energy consumption of a target device.

A voltage and frequency scaling adaptation of the core energy model is also presented, with a mean squared error of 2.6 %. In future work, we propose that the `axe` simulation tool can be further improved to support frequency selection, allowing instruction set simulation, VFS-aware energy modelling, and thus design exploration including VFS parameters.

Additional opportunities for future work include incorporating these new models into static analysis techniques. Some static analysis techniques have been demonstrated against prior work on multi-threaded energy models [16, 7, 6], and so there is a strong case for extending such work to the multi-core level. Simulation based modelling could also be further extended by examining larger systems, such as the many-core Swallow system [9], which assembles potentially hundreds of XS1-L processors into a compute grid. However, appropriately sized benchmark applications, or compositions of smaller related tasks, would need to be identified , adapted or developed in order to exercise the model and such a system in an appropriate context.

References

[1] Adapteva. *Ephiphany Introduction*. 2015. URL: `http://www.adapteva.com/introduction/` (visited on 03/30/2015).

[2] A. Bartolini et al. “A Distributed and Self-Calibrating Model-Predictive Controller for Energy and Thermal management of High-Performance Multicores”. In: *Energy* (2011).

[3] D. Brooks, V. Tiwari, and M. Martonosi. *Wattch: A Framework for Architectural-Level Power Analysis and Optimizations*. May 2000. DOI: `10.1145/342001.339657`. URL: `http://portal.acm.org/citation.cfm?doid=342001.339657`.

[4] EZChip Semicondutor. *TILE-Gx72 Processor*. Tech. rep. 2009, pp. 1–2.

[5] S Furber et al. “Overview of the SpiNNaker System Architecture”. In: *IEEE Transactions on Computers* 62.12 (2013), pp. 2454–2467. URL: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6226357`.

[6] K. Georgiou, S. Kerrison, and K. Eder. *A Multi-level Worst Case Energy Consumption Static Analysis for Single and Multi-threaded Embedded Programs*. Tech. rep. University of Bristol, 2014. URL: `http://www.cs.bris.ac.uk/Publications/pub_master.jsp?id=2001701`.

[7] N. Grech et al. “Static analysis of energy consumption for LLVM IR programs”. In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. SCOPES ’15. Sankt Goar, Germany: ACM, 2015. DOI: `10.1145/2764967.2764974`.

[8] W. Heirman et al. “Power-aware multi-core simulation for early design stage hardware/software co-optimization”. In: *Proceedings of the 21st international conferencge hardware/software co-optimizatione on Parallel architectures and compilation techniques - PACT ’12*. New York, New York, USA: ACM Press, 2012, p. 3. ISBN: 9781450311823. DOI: `10.1145/2370816.2370820`. URL: `http://dl.acm.org/citation.cfm?doid=2370816.2370820`.

[9] S. J. Hollis and S. Kerrison. “Overview of Swallow - A Scalable 480-core System for Investigating the Performance and Energy Efficiency of Many-core Applications and Operating Systems”. In: *arXiv* (2015). URL: `http://arxiv.org/abs/1504.06357`.

[10] M. Ibrahim, M. Rupp, and S. Habib. *Power consumption model at functional level for VLIW digital signal processors*. Tech. rep. 1. 2008, pp. 2–7. URL: http://www.researchgate.net/publication/228947933_Power_consumption_model_at_functional_level_for_VLIW_digital_signal_processors/file/e0b49521c2bc72bd43.pdf.

[11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes*. December. 2011, p. 3463.

[12] Intel Corporation. *Intel Xeon Phi Coprocessor*. Tech. rep. 2013.

[13] A. B. Kahng. "The ITRS design technology and system drivers roadmap". In: *Proceedings of the 50th Annual Design Automation Conference on - DAC '13*. New York, New York, USA: ACM Press, 2013, p. 1. ISBN: 9781450320719. DOI: 10.1145/2463209.2488776. URL: http://dl.acm.org/citation.cfm?doid=2463209.2488776.

[14] S. Kerrison. "Energy modelling of multi-threaded, multi-core software for embedded systems". PhD thesis. University of Bristol, Dept. of Computer Science, 2015.

[15] S. Kerrison and K. Eder. "Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor". In: *ACM Transactions on Embedded Computing Systems* 14.3 (Apr. 2015), 56:1–56:25. ISSN: 1539-9087. DOI: 10.1145/2700104. URL: http://doi.acm.org/10.1145/2700104.

[16] U. Liqat et al. "Energy Consumption Analysis of Programs based on XMOS ISA-Level Models". In: *23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*. Springer, Sept. 2015.

[17] MAGEEC Project. *MAGEEC Power Measurement Board*. 2014. URL: http://mageec.org/wiki/Power_Measurement_Board (visited on 07/04/2015).

[18] D. May. *The XMOS XS1 Architecture*. 2009. ISBN: 9781907361012.

[19] D. May et al. *XS1-L System Specification*. 2008.

[20] K. Roy and M. C. Johnson. "Software design for low power". In: *Low power design in deep submicron electronics*. Kluwer Academic Publishers, 1997. Chap. 6, pp. 433–460. ISBN: 0-7923-4569-X. URL: http://dl.acm.org/citation.cfm?id=265902.

[21] Scikit-Learn. *Scikit-Learn Decision Trees*. 2015. URL: http://scikit-learn.org/stable/modules/tree.html (visited on 03/18/2015).

[22] SciPy. *scipy.optimize.minimize — SciPy v0.15.1 Reference Guide*. 2015. URL: http://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.optimize.minimize.html (visited on 08/12/2015).

[23] Y. S. Shao and D. Brooks. "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor". In: *International Symposium on Low Power Electronics and Design (ISLPED)*. November. IEEE, Sept. 2013, pp. 389–394. ISBN: 978-1-4799-1235-3. DOI: 10.1109/ISLPED.2013.6629328. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6629328.

[24] Sim-Panalyser. *Sim-Panalyser 2.0 Reference Manual*. 2004, pp. 1–54.

[25] S. Steinke et al. "An accurate and fine grain instruction-level energy model supporting software optimizations". In: *Proc. of PATMOS*. Citeseer, 2001. DOI: 10.1.1.115.3528. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.6971\&rep=rep1\&type=pdf.

[26] V. Tiwari, S. Malik, and A. Wolfe. "Compilation techniques for low energy: An overview". In: *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*. IEEE, 1994, pp. 38–39. ISBN: 0780319532. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=573195.

[27] J. Yiu and I. Johnson. *Multi-core microcontroller design with Cortex-M processors and CoreSight SoC*. Tech. rep. ARM, 2013.

Attachment D2.3.2

Data dependent energy modelling: A worst case perspective

In draft, awaiting submission.

arXiv:1505.03374v1 [cs.PF] 13 May 2015

Data dependent energy modelling: A worst case perspective

James Pallister, Steve Kerrison, Jeremy Morse and Kerstin Eder
Dept. Computer Science, Merchant Venturers Building,
Bristol, BS8 1UB. Email: firstname.lastname@bristol.ac.uk

***Abstract*—Energy consumption of the software running on a device has become increasingly important as a growing number of devices rely on batteries or other limited sources of power. Of particular interest is constructing a bounded measure of the energy consumption — the maximum energy a program could consume for any input given to it.**

We explore the effect of different data on the energy consumption of individual instructions, instruction sequences and full programs. The whole program energy consumption of two benchmarks is analysed over random and hand-crafted data, and maximized with genetic algorithms for two embedded processors. We find that the worst case can be predicted from the distribution created by the random data, however, hand-crafted data can often achieve lower energy consumption.

A model is constructed that allows the worst case energy for a sequence of instructions to be predicted. This is based on the observation that the transition between instructions is important and thus is not a single energy cost — it is a distribution dependent on the input and output values of the two consecutive instructions. We characterise the transition distributions for several instructions in the AVR instruction set, and show that this gives a useful upper bound on the energy consumption. We explore the effect that the transfer function of the instruction has on the data, and give an example which leads to a bimodal energy distribution. Finally, we conclude that a probabilistic approach is appropriate for estimating the energy consumption of programs.

I. Introduction

The energy consumption of our embedded devices is becoming ever more important to characterize and account for, since battery capacity has not increased along with our energy needs. To combat this, many methods of reducing energy consumption have been proposed, both in hardware and in software. Fundamentally, the software controls the hardware, thus any technique implemented in hardware must also be complemented with software support. It is all too easy for the software to ignore hardware optimizations, negating any beneficial impact. As such, it is important to enable the software to make effective use of the hardware to minimize energy.

Energy modelling is a technique which allows the energy consumption of software to be estimated without measuring a physical device. For example, a model may take the form of assigning an energy value to each instruction [1], an energy value to each state the processor [2], or a detailed approach utilising a large amount of the processor's state, including data for each instruction [3]. Although taking measurements is typically superior to using an energy model in terms of accuracy, a model is much more versatile and can be used in many more situations, such as statically predicting energy consumption [4] and making optimization decisions to reduce energy consumption [5].

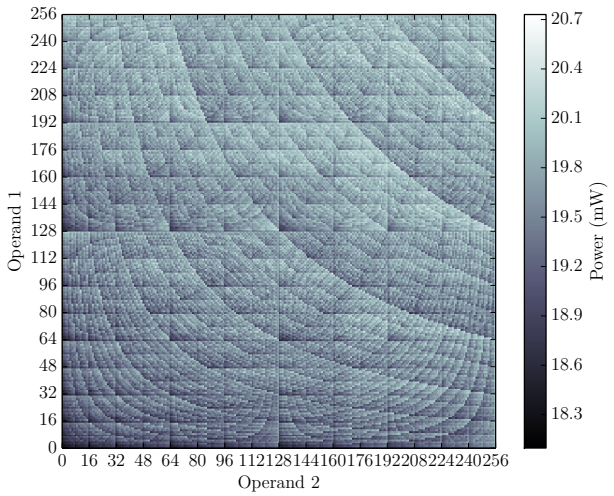


Fig. 1: A heat map showing how data affects the energy consumed by a `mul` instruction.

In real-time embedded systems, the execution time of a program must be bounded. This can provide guarantees that tasks within a program will meet hard deadlines. Recently, efforts have been made to ensure the energy consumption can also be given an upper bound, with the intent of guaranteeing that a task will complete within an available energy budget. However, these efforts are often based upon energy models that do not explicitly consider the dynamic power drawn by switching of data, instead estimating an upper bound using either average or otherwise scaled instruction models.

The change in energy consumption caused by different data can have a significant impact on the overall energy consumption of a program. Previous work [6] has reported up to 20% difference in energy consumption with different data being operated on. This work finds 15% energy difference in a simple AVR [7] processor. As an example of the variability within an instruction, Figure 1 shows the difference in power for a single cycle, 8-bit multiply instruction in this processor[1], where the worst case input for this instruction is 15% higher than the most energy efficient input. The diagram was constructed by taking measurements for every possible input, which was only feasible due to the processor being 8-bit.

Accounting for data dependent effects in an energy model is a challenging task, which we split into two parts. Firstly, the energy effect of applying an instruction to the processor state needs to be modelled. This is an infeasible amount of data

[1]All measurements in this paper are taken on physical hardware.

to exhaustively collect, for instance for a 32-bit architecture with a three-operand instruction, 2^{96} combinations of data values are used in the instruction. Secondly, a technique is required to derive the energy consumption for a sequence of instructions from such a model. The composition of data dependent instruction energy models is a particularly difficult task, as the data causing maximum energy consumption for one instruction may minimize the cost in a subsequent, dependent instruction. Finding the greatest cost for such sequences would require searching for inputs that maxmimize a property after an arbitary computation, which is an infeasibly large task. Overapproximating by summing the worst possible data dependent consumption of each instruction in a sequence, regardless of whether such a computation can occur, would lead to a very inaccurate upper bound.

We explore the effect of data on the maximal energy consumption of programs, and instructions, performing probabilistic analysis on the distributions of energy consumption obtained. The data's effect on the entire program is explored with random data, a genetic algorithm, and carefully crafted data, finding that random data can form a distribution from which a maximum energy can be estimated. Individual instructions are analysed, and several probabilistic modelling approaches are being explored to accurately determine the maximum energy consumption of sequences of instructions. This analysis highlights how correlations between the data reduce the maximum energy. A degenerate case is discovered, where the sequence of instructions results in a bimodal energy distribution.

This paper is organised as follows. The next section discusses related work. In Section III the effect of data on two full programs is explored for two different embedded processors. Section IV examines and models individual and sequences of instructions in the AVR, including a sequence of instructions which causes a bimodal distribution. Section VI concludes and gives an outlook on future work.

II. Related work

Worst Case Execution Time (WCET) analysis attempts to find an upper bound on the time taken for an arbitrary program to execute [8, 9]. A key approach is a method called Implicit Path Enumeration Technique (IPET) [10], which estimates an upper bound given information about a program's control flow graph. Of recent interest has been work on Worst Case Energy Consumption (WCEC), utilizing methods from the WCET area, and combining them with energy modelling techniques to bound the program's energy consumption [11]. However in many of these studies, the energy model used is not tailored for the worst case, nor is the impact of data on energy consumption adequately reflected. This can lead to unsafe results.

Much work has gone into creating energy models, so that the overhead of taking physical measurements can be avoided. The most common form of model for embedded systems is an instruction level model. The instruction level model proposed by Tiwari [12] uses an energy cost for each instruction, and an energy cost for the circuit switching effect between each instruction, as well as a final catch-all factor to cover other effects. The model does not consider data at all, instead assuming that all of the data dependent effects have been captured in the other coefficients.

Steinke et al. [3] construct a more detailed energy model that does consider the effects of data as well as the instructions. The data energy is based on several variables, both using the Hamming distance between consecutive values and their Hamming weights. The pertinent variables used in this calculation are the value in the register, the data and the address of any memory access. The technique achieved impressive results, with only 1.7% error. However, it is not known if the Hamming distance between register values is a sufficiently detailed indicator to capture the full energy behaviour.

Many studies agree that the Hamming distance between consecutive operand values has a positive correlation with power dissipation, however, the correlation is only moderate, with many other factors also having an effect. Park et al. [13] consider how different operand values affect the energy consumption, using a range of values between `0x0000` and `0xFFFF` to ensure that there is a large number of different Hamming distances between operands. A similar approach, based on the Hamming weight is used in [14].

Further studies have extensively used the Hamming weight to account for data energy [15]. The study notes that the Hamming distance and weight are particularly useful for subsequent values on buses in the processor, and less useful for combinatorial instructions, such as arithmetic. Ascia et al. [16] build upon the approach, exploring how the data transitions from 0 to 1 and 1 to 0 can be given different energy costs. In a study of the Leon3 processor [17], taking data into account was found to reduce the model error when a 'typical' number of switching bits was factored in.

Kojima et al. [18] measure the data's effect on power for the adder and the multiplier in a DSP, as well as the register file. The register file power was found to show linear dependence on the Hamming weight of the data operand, while the adder shows moderate correlation with the number of transitions in the input data (i.e. Hamming distance between successive operands). However, the multiplier shows very little correlation with the Hamming distance, except when one of the inputs is held constant. This backs up the suggestion in [15] that combinatorial blocks require parameters other than the Hamming distance and weight.

Similar conclusions have been reached in studies which attempt to find the maximum power a circuit may trigger [19]. Many studies attempt to maximize the power consumption of a circuit, using a weighted maximum satisfiability approach [20], and genetic algorithms [21].

The reachability of a particular state has large implications for maximum energy consumption. Hsiao et al. [22] use a genetic algorithm to determine the maximum power per cycle in VLSI circuits, discovering that the peak power for a single cycle was higher than the peak sustainable power. This is due to the state that would be triggering maximum power dissipation not being reachable from the current circuit state, and instead the only reachable states dissipate less power. This reasoning can be applied to processor instructions too — the data triggering highest energy consumption in one instruction may be transformed in such a way the the subsequent instructions cannot consume maximal energy.

Probability theory has also been used to characterize how circuits dissipate power. Burch et al. [23] take a Monte Carlo approach, simulating the power of different input patterns to a circuit. The paper hypothesizes that the distribution of powers can frequently be approximated by a normal distribution, as a consequence of the central limit theorem [24]. While the central portions of the probability distribution fit well to a

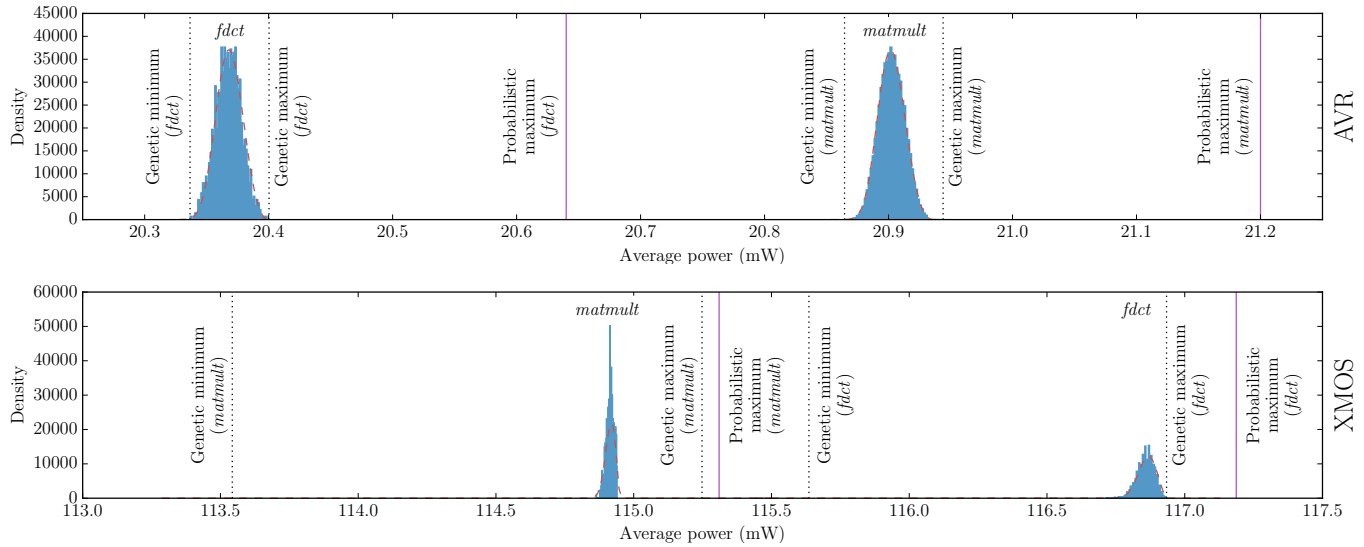


Fig. 2: The distributions obtained for random datasets with the *fdct* and *matmult-int* benchmarks, on the AVR and XMOS processors.

TABLE I: The number of possible datasets for each benchmarks.

Benchmark	Elements	Size (bits)	Total space (bits)
matmult	800	8	6,400
fdct	64	16	1,024

normal distribution, the tails diverge, implying that a different distribution would be a better fit when maximum power is of interest. Studies have used extreme value theory to rectify this issue. The extreme value distribution is of importance when the maximum of a series of random variables is needed. This has been applied to maximum power estimation in VLSI circuits [25, 26], enabling a probabilistic estimate of the maximum power with only a limited number of simulations. Probabilistic modelling for power dissipation has also been performed at high levels [27], with server processors and network interface cards.

In summary, energy consumption and data dependency has been considered at the VLSI level using a variety of techniques. However, there has been little exploration of data dependency at the instruction or application level for the worst case energy consumption.

III. Whole Program Data Dependence

In this section we examine how a program's energy consumption changes as data is being processed, giving a measure of how much a program's energy consumption depends on its data. Programs that have no data dependent branches are chosen, therefore changes in energy are purely due to different data progressing through the computational path in the processor. Having no data dependent branches simplifies the analysis, since choosing data which takes a different execution path changes which instructions are executed, which would skew the average power and energy consumption.

The benchmarks used for this test are *fdct* and *matmult-int*, taken from BEEBS, an embedded benchmark suite [28]. These tests are purely integer, because the target processors in this work have no hardware floating-point support and soft floating-point libraries often branch for specific corner cases during computation. Neither chosen benchmarks have data dependent branches, thus their execution time is identical even with different input data.

These benchmarks are run on two distinct architectures, Atmel AVR and XMOS XS1 [29]. The AVR has an 8-bit data-path whereas XS1 is 32 bits wide. Additionally, the XS1 features hardware multi-threading, in this case with a four-stage pipeline. The specific XS1 device under test is the single core, eight thread XS1-L1, operating at 400 MHz. Using single threaded benchmarks, the pipeline is only 25% utilised. However, the effects of data on these benchmarks is still measurable. An example of synthetically constructed worst case data for power in a multi-threaded scenario is given for the XS1-L1 in [6].

The entire data space of the program cannot be explored exhaustively. For example, in a 20 by 20 matrix multiplication of 8-bit integers there are $3,200$ bits of data for each matrix, which is an infeasibly large space to fully explore. The following sections apply several search methods for maximizing a program's data dependent energy consumption. First, a profile of the typical energy consumption is built by using random data. Then, the energy is minimized and maximized using a genetic algorithm to guide the search. Finally, hand crafted test patterns are used to trigger different energy consumption.

We fit the Weibull distribution to random data, under the hypothesis that the switching and hence power dissipation caused by random data will be close to the maximum. The distribution can then be examined to estimate an upper bound. The reversed Weibull distribution is used in extreme value theory, however, it may underestimate since it has a finite cut-off point which must be estimated accurately. Empirically, it is found that the regular Weibull distribution can model the distributions found (see Figure 4 for a comparison between the two). The Weibull cumulative probability distribution is given by,

$$F(x;k,\mu,\sigma) = 1 - e^{-\left(\frac{x-\mu}{\sigma}\right)^k}. \quad (1)$$

This is in contrast to the type III extreme value distribution,

the reversed Weibull,

$$F(x; k, \mu, \sigma) = 1 - e^{-\left(\frac{-x-\mu}{\sigma}\right)^k}, \quad (2)$$

which is only defined for x up to μ.

A. Random data

Figure 2 shows the average power obtained when the *fdct* and *matmult-int* benchmarks use random data. The red line shows the Weibull distribution fitted to these data. Overall, the distributions are narrow, indicating a low variation caused by the data. The variations for both benchmarks on AVR are similar, however, each has a different mean, since different instructions are executed, each with a different average power.

Using the distribution calculated, and the total size of the input data space, an estimate of the maximum possible average power can be calculated,

$$(1 - CDF(x)) \cdot S = 1, \quad (3)$$

where $CDF(x)$ is the cumulative density function of the probability distribution, and S is the total size of the data space. Intuitively, this is equivalent to finding the value of the percentile representing the highest power dataset in the entire data space.

Fitting these parameters to the distribution parameters for each of the benchmarks results in an estimation of the maximum achievable average power for each benchmark. For example the probabilistic maximums for AVR are,

$$fdct = 20.64 \text{ mW} \quad (4)$$

$$matmult = 21.20 \text{ mW}. \quad (5)$$

These upper bounds are shown by the solid vertical lines on the graph.

B. Genetic algorithm

The related work showed that genetic algorithms were an effective technique to finding the maximum power dissipation for a circuit [22]. Genetic algorithms can also be applied to the data a program operates on. In this paper, a genetic algorithm is instantiated that attempts to find a dataset which increases the energy or power for the entire program.

The results of this are included in Figure 2 as vertical dotted lines. These data points are slightly higher and lower than the points found by the random data — the guidance provided by the genetic algorithm allows both higher and lower solutions to be found quickly. Since the parameters to the Weibull distribution were found for each distribution, the probability of finding a more extreme solution can be calculated, e.g. for AVR,

$$P(x > \textit{fdct}_{max}) = 6.92 \times 10^{-10} \quad (6)$$

$$P(x > \textit{matmult-int}_{max}) = 1.00 \times 10^{-14}. \quad (7)$$

The probability of finding a solution more extreme than the current ones is very low, provided the assumption of the distribution being a good fit holds. However, the size of the data input space is so large that there are many possible states which may trigger a larger energy consumption.

C. Hand-crafted data

Due to the extremely large number of input states to both of the benchmarks, there are certain configurations of input that are never considered by the random search or the genetic algorithm. This includes data such as every bit set and every bit cleared, could be important and trigger an unusually high or low energy consumption.

The types of hand-crafted data fed into each benchmark are listed below.

All bits zero. All of data values are set to zero.

All bits one. All of the bits in the data values are set.

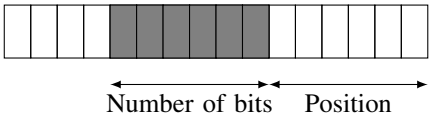
Strided data. The data element is set to one at various strides, such as every 2, 4, 8 or 16 bytes.

Strided random data. The data is set to a random value at various strides, such as every 2, 4, 8 or 16 bytes.

Patterns. Some patterns are known to cause high energy consumption for particular instructions. For example, `0xaaaaaaaa` and `0x55555555` are known to be power intensive for certain multipliers [6].

Sparse data. Very sparse data, such as only one element being set to one in various positions is tested.

Restricted bit-width. Setting random values to a restricted portion of the element is tested. The diagram below shows which bits are set randomly.



All elements the same. Every element in the data is set to the same value. A range of values are tested.

Figure 3 shows the average power when all of these hand-crafted sets of data are measured on each benchmark. There are many different components of these graphs — each caused by a different part of the hand-crafted data.

- A This mode is around the lowest average power achievable for the *matmult* benchmark, caused by the all bits zero data and the sparse data.
- B The distribution at B consists of data sets for which most elements are 1, and few elements are set to zero. This causes a low average power since the matrix multiply is most frequently performing 1×1 — an operation which takes little power compared to multiplying larger numbers (but more than 0×0).
- C There are a spread of points at this location, across a wide range of power values — these are the other tests which involve more dense data.
- D The highest consumption in the non-sparse tests is 21.04 mW for AVR, and is caused by data which has the same value in all the elements. The values of the elements for the top results are 247, 253, 181, 221 and 245 — close to having all bits set. These are the only tests which significantly exceed the distribution obtained from random data. For the XS1-L, a larger proportion of tests dissipate a higher power, visible in the form of a third peak.
- E For the *fdct*, there are three data points which are far lower than any other. These are the all zero data, and two instances of strided data — when the first in every 32 elements is one and all elements are zero. This is sparse data, however any of the other sparse data still triggers much higher power. This characteristic is

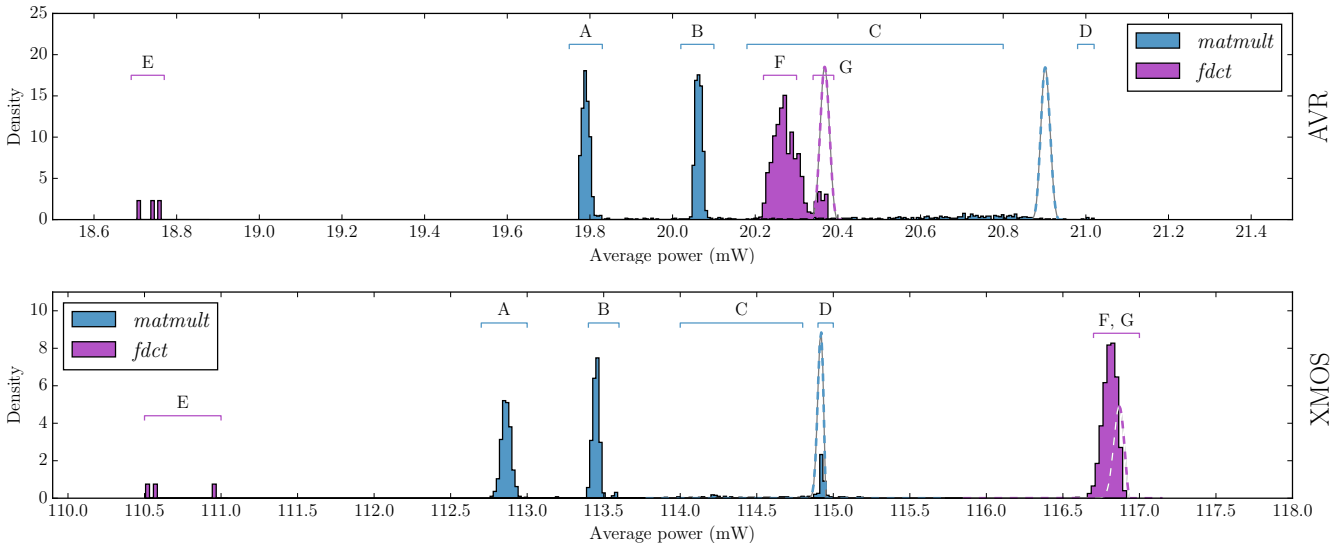


Fig. 3: Distribution of average power measured for both benchmarks on both platforms, when run with hand-crafted datasets.

observed on both architectures.

F The majority of tests occur in this bracket, below the expectation given by random data. Since the AVR is an 8-bit processor, the 16-bit arithmetic is emulated with at-least two instructions per operation. Many of the hand-crafted data sets used zero or close to zero value data, resulting in the second operation having zero value and thus lower power. This is not the case with the XS1-L, giving a possible explanation for the presence of a single peak in its case.

G These data points tend to be triggered by high data — the *fdct* operates on 16-bit data. With high valued data, the second operation in the emulated arithmetic (for the upper bits) has non-zero value — corresponding to a higher average power.

Overall, there is a trend towards higher average power as the data becomes more random or dense. The distribution predicted by purely random data is good as an estimation of the upper bound — very few tests exceeded the limits found earlier with genetic algorithms, and all were bounded by the probabilistic highest value. This suggests that the distribution obtained from random data can be used to estimate a worst case energy consumption, but not a best case.

Comparing the characteristics observed for each benchmark on AVR versus XS1-L, the distributions take similar forms for both *matmult* and *fdct*. The XS1-L1 dissipates more power, but is a more complex device with a higher operating frequency. However, the separation between the distributions A and B in *matmult* are within the same order of magnitude for both devices, at approximately 25 μW and 65 μW for AVR and XS1-L1 respectively. Similarly, the widths of the features denoted F in *fdct* differs by a comparable amount.

IV. Modelling

It has been seen that the entire program can be modelled using the Weibull distribution, however, this is at a very coarse granularity, and will reduce accuracy with programs that have data dependent branches. To cope with data dependent branches, each basic block in the program should have an energy distribution associated with it. These can then be combined

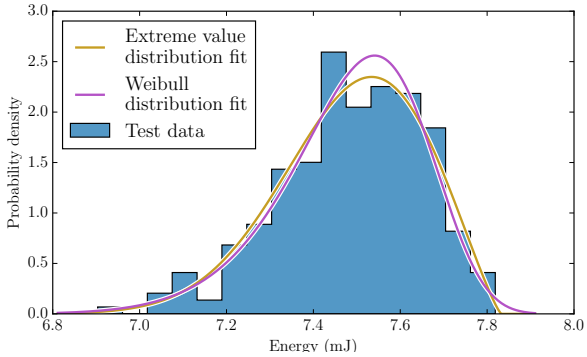


Fig. 4: Distribution of energy values for the `lsl` instruction.

using standard techniques to estimate the worst case, such as IPET [10]. This section covers the creation of a model that will output an energy distribution for each basic block.

A simplistic method to generate each basic block's energy distribution would be to measure each block in isolation, and input random data. The energy cost of each block could then be measured and a distribution built up. However, even for small programs this is a large amount of work, and the amount of work scales up as the program size increases. A more tractable approach is to model each instruction in the basic block and compose these models. Information about each instruction can be characterized, and then reused for each basic block, requiring no further measurements than the initial collection of data for each instruction.

In most cases, a typical instruction has too many possible data values to exhaustively explore all of them. The key insight in this section is that, as with the total energy for a program, the overall distribution of instruction's energy consumption typically conforms to the Weibull distribution.

Initial attempts to model a *single* instruction distribution with the Weibull, and the extreme value distribution are successful, as seen in Figure 4. The distribution was constructed by repeatedly placing random data into each register the

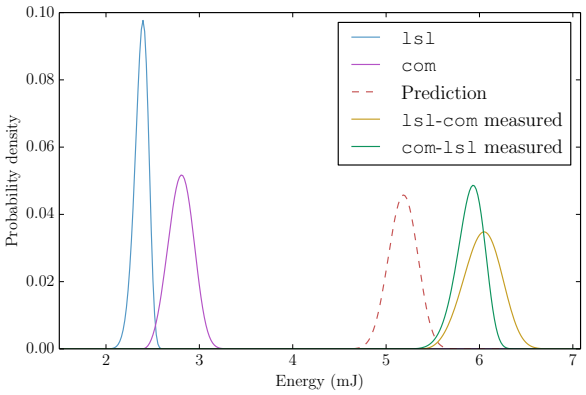


Fig. 5: Comparison of prediction energy for the combination of a `com` and a `lsl` instruction with a simplistic model.

instruction operates on, and measuring each test. By convolving the individual distributions of instructions together, a prediction of multiple instruction can be constructed. Figure 5 shows the distributions for the instructions, `com` (bitwise complement) and `lsl` (logical shift left). The dashed curve shows the expected distribution. The two curves marked in green and orange show the actual distributions of the energy for each instruction — one for `com`, then `lsl`, and the second for `lsl`, then `com`. These distributions are not similar, and more importantly are higher than the prediction resulting in an underestimate of the worst case energy consumption.

The difference in distributions stems from the surrounding instructions — to evaluate the instructions, the sequence is prefixed with a `mov` instruction to set up the values going into `com` and `lsl`. This suggests that the actual switching of data between the instructions can have a significant impact on not only the average energy, but the shape of the distribution too.

By using a model based on Tiwari et al. [1], a transition distribution to represent the data dependent transition between instructions can be used,

$$E_p = \sum_{i,j \in I} E_{i,j} \tag{8}$$

$$E_{i,j} \sim \text{Weibull distribution.} \tag{9}$$

E_p can be calculated by convolving the individual probability distributions,

$$f_p(x) = \bigotimes_{i,j \in I} f(x; k_{i,j}, \mu_{i,j}, \sigma_{i,j}). \tag{10}$$

where μ, σ and k are the parameters into the Weibull probability density function, f, $\bigotimes$ is the convolution operator, and f_p is the probability density function of the instruction sequence. The convolution of two Weibull probability density functions is not known to have an analytical solution, so is solved numerically for the purposes of this study.

A. Data collection

The collection of transition distributions for each pair of instructions is particularly challenging. The most simplistic way to approach this is to repeat a pair of instructions with specified data, and measure the energy,

```
add r0, r1, r2
sub r3, r4, r5.
```

However, after the first repetition `r0` and `r3` will not exhibit the same switching as they did in the first iteration — the value in the register will not change. This means that the values in the register should be randomised before and after the execution of the instructions,

```
mov r0, X          ⎤ E_mov,mov
mov r3, Y          ⎥ E_mov,add
add r0, r1, r2     ⎥ E_add,sub
sub r3, r4, r5     ⎥ E_sub,mov
...                ⎦
```

where X and Y are independent, uniform random variables. In addition to X and Y, the registers `r1`, `r2`, `r4` and `r5` are initialised to random values. This ensures that all the variables that could affect the transition distribution between two instructions are random. As seen before this should lead to each transition distribution conforming to the Weibull distribution. The above test forms the distributions seen to the right of the instructions.

By adding in the extra instructions to ensure each register contains a random variable, additional `mov` instructions are inserted. These are convolved with the distribution that is of interest; they must first be found so that they can be removed.

A large number of different values can then be assigned to all the variables in the sequence, and the energy, E_s measured for each. This forms the following equation which must be solved to find $E_{add,sub}$,

$$E_s = E_{mov,mov} \otimes E_{mov,add} \otimes E_{add,sub} \otimes E_{sub,mov}. \tag{11}$$

Equation 11 can be solved by first finding the distribution for $E_{mov,mov}$, and then finding the distributions for $E_{mov,i}$, where i is an instruction from the instruction set. For simplicity it is assumed that the distribution $E_{i,j}$ is identical to the distribution $E_{j,i}$. The $E_{mov,mov}$ distribution can be found by finding the distribution for the following code sequence,

```
mov r0, X   ⎤ E_mov,mov(Z, X, W, Y)
mov r1, Y   ⎥ E_mov,mov(W, Y, X, Z)
mov r0, Z   ⎥ E_mov,mov(X, Z, Y, W)
mov r1, W   ⎥ E_mov,mov(Y, W, Z, X)
...         ⎦
```

The resulting distribution of energy values is $E_{mov,mov}$ convolved with itself 4 times. Solving for this distribution, and then using this along with similarly formed tests to find $E_{mov,i}$ and $E_{mov,j}$, the transition distribution for any $E_{i,j}$ can be found. The above sequence also displays the random variables that affect the distributions — two for the first `mov` instruction (the values in the output and the input registers) and two for the subsequent `mov` instruction.

The transition distributions for a subset of the AVR's instruction set have been found, and are using in the following section to predicting distributions for multiple instructions.

B. Instruction sequence tests

Using the transition distributions between consecutive instructions, a prediction for a sequence of instructions can be made. Figure 6 shows the predicted distributions for three short instruction sequences, as marked on the graph. In all cases

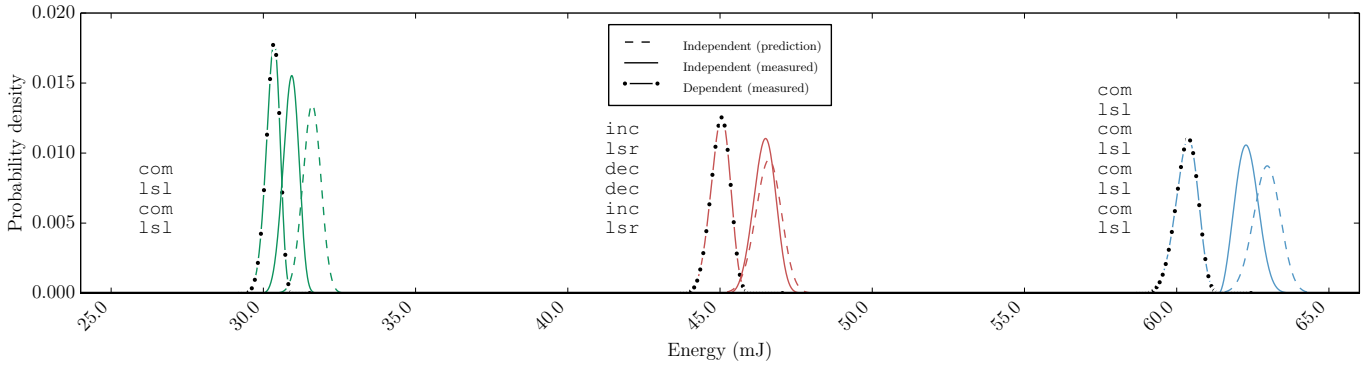


Fig. 6: Comparison of prediction for three instruction sequences, using transition distributions. For the independent tests, each register in the sequence has an independent value (independent random variables). For the dependent test, each register is `r0`, and thus the data value operated on by successive instructions is dependent on the previous instruction.

```
mov r3, r20
mov r4, r21
mul r3, r4
mov r2, r0
mul r2, r3
mov r4, r0
mul r4, r2
mov r3, r0
```

Repeat 3 times

Fig. 7: Sequence of `mov` and `mul` instructions causing a bimodal distribution. A `mul` implicitly writes to `r0`.

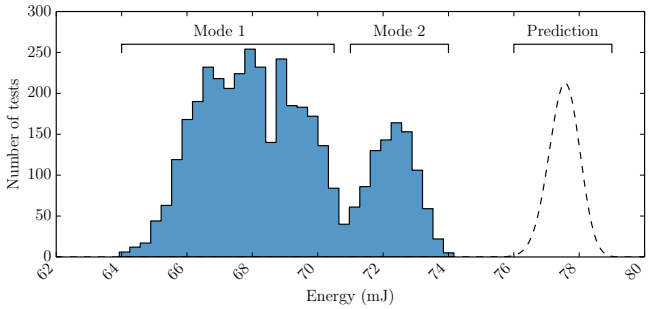


Fig. 8: A histogram showing the distribution of energy values for all data values when executing a sequence of multiplies.

the prediction is conservative — the mean of the distribution is overestimated. This makes it useful as a worst case energy model, since the 99^{th} percentile can be taken as a probabilistic estimation of maximum energy (for example).

The figure also demonstrates the case where the values in the registers are not randomly distributed, and are instead dependent on the transformations by previous instructions. All of these distributions have a smaller mean — the correlation between registers causes a lower energy overall and the worst case bound holds.

The tests in this section only showed arithmetic instructions. However, the distributions for load and store instructions are similar and can be composed similarly. It is expected that branch instructions will be simple to characterize — while there are often no direct inputs to a conditional branch, the state of the control flags influences the direction of the branch.

V. Data dependency between instructions

The previous section mentioned that the effects of the computation may impact the location of the distribution. This section presents a case where this occurs when certain sequences of multiply instructions are used. Figure 7 shows a sequence of `mul` and `mov` instructions, which calculates $a^{13} \cdot b^8$, where `r2` $= a$ and `r1` $= b$.

The sequence was measured for its energy under different inputs, and a histogram of these data is shown in Figure 8. The number of data values causing each energy consumption is on the y-axis. The distribution has two large peaks, labelled with two modes in the figure. In this particular example the lower energy peak is caused by the computation collapsing to a 0 value. When a 0 is fed into the multiply it is typically very low energy — and the output is 0, ensuring that in a sequence of multiplies this will achieve a very low energy.

The higher mode is caused by neither of the inputs to any of the multiplies being zero, which occurs for every multiply when both inputs are odd. Overall the higher energy requirements of this computation cause a separate mode to occur.

The sequence of instructions is unusual in that it may not typically appear in a program. However, this kind of behaviour could possibly be triggered by other instructions, causing particularly high or low energy to occur. The bimodal behaviour is perhaps more likely to occur when the instruction itself is bimodal. These instructions are typically comparisons which output two possible values[2], and the results of these operations are not typically used for computation other than deciding whether or not to take a branch.

While this type of behaviour will affect the tightness of the energy's upper bound, it does not affect its safety, since it is the upper mode that is captured by the model.

VI. Conclusion and future work

This paper has analysed how the data a processor operates on affects its energy consumption. Initial analysis for a full program suggests that using random data to create a Weibull

[2]Another example is the `lss` instruction on the XMOS processor, which places a one or zero in the register, based on the result of a signed less than comparison.

distribution allows a probabilistic worst case for that program to be estimated. The probabilistic worst case was higher than could be found using random data, a genetic algorithm, or hand-crafted data. The hand-crafted data more often resulted in an energy consumption that was significantly lower than expected — these data fed to these tests often caused little bit-switching, and so took a smaller amount of energy.

While the upper bound of these two benchmarks could be modelled this way, other programs with data dependent branches would be much more challenging to model. In an effort to create a composable analysis, the transition between each instruction was modelled as a Weibull distribution. Once each distribution for each pair of instructions has been characterized, the distributions can be convolved, giving a probability distribution for a sequence of instructions.

Several instruction sequences were tested, comparing the predictions to the actual measured distributions. The prediction is close, and overestimates the energy consumption in all cases, providing a conservative estimate of the worst case energy consumption. The prediction assumes that all of the instructions are independent of effects upon each other, however in a real program this is not true. The measurements are repeated, for when the same instruction sequence had dependencies between instructions, finding that added correlation between the values always decreased the total energy consumption — the prediction still provides an upper bound.

The correlation between data values input and output from instruction can lead to unusual energy behaviour. One sequence was explored, a series of multiplications and data movement, which resulted in bimodal energy behaviour across a range of random data. The bimodal distribution was caused from some register values 'collapsing' to zero partway through the computation (when at least one input is even), and remaining at zero due to the multiplication. Since a zero operand causes much lower than average energy consumption, the tests where this happens achieve lower energy. In this case of extreme correlation between data values, the modelling effect will over predict, but is still a safe upper bound, and these contrived cases are unlikely to occur frequently in realistic programs.

The next step is to gather these distributions for the entire instruction set, and combine it with a technique such as implicit path enumeration, so that larger programs with data dependent branches can be analysed, and a probabilistic worst case given to the developer.

Another observation is that programs have differing degrees of data dependency — some instructions in the program are purely control overhead, and do not operate on the data input to the program. A static analysis could find just the instructions which are in the data path of the program, and an estimate of the total variability due to data could be constructed from these instructions which operate on data, and the transition distributions.

VII. Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 318337, ENTRA - Whole-Systems Energy Transparency. This study was partly sponsored by EPSRCs Doctoral Training Account EP/K502996/1. The energy measurement hardware (MAGEEC WAND) used for measuring the energy of the AVR hardware was funded by Innovate UK, award 131198.

References

[1] Tiwari et al. "Power analysis of embedded software: a first step towards software power minimization". In: *IEEE Transactions on VLSI Systems* 2.4 (Dec. 1994), pp. 437–445.

[2] Nunez-Yanez et al. "Enabling accurate modeling of power and energy consumption in an ARM-based System-on-Chip". In: *Microprocessors and Microsystems* 37.3 (May 2013), pp. 319–332.

[3] Steinke et al. "An accurate and fine grain instruction-level energy model supporting software optimizations". In: *Proc. PATMOS*. 2001.

[4] Liqat et al. "Energy Consumption Analysis of Programs based on XMOS ISA-Level Models". In: *23rd Int. Symp. Logic-Based Program Synthesis and Transformation*. Springer, Sept. 2015.

[5] Pallister et al. "Optimizing the flash-RAM energy trade-off in deeply embedded systems". In: *Proc. 13th Annual IEEE/ACM Int. Symp. on Code Generation and Optimization* abs/1406.0 (May 2015), pp. 115–124. arXiv:1406.0403.

[6] Kerrison et al. "Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor". In: *ACM Trans. Embed. Comput. Syst.* 14.3 (Apr. 2015), 56:1–56:25.

[7] Atmel. *Atmel 8-bit Microcontroller*. 2013.

[8] Hahn et al. "Towards compositionality in execution time analysis". In: *ACM SIGBED Review* 12.1 (Mar. 2015), pp. 28–36.

[9] Engblom et al. "Comparing different worst-case execution time analysis methods". In: *Proc. Work-in-progress Session at the 21st Real-Time Systems*. 2000, pp. 1–4.

[10] Li et al. "Performance analysis of embedded software using implicit path enumeration". In: *ACM SIGPLAN Notices* 30.11 (Nov. 1995), pp. 88–98.

[11] Jayaseelan et al. "Estimating the Worst-Case Energy Consumption of Embedded Software". In: *Symp. Real-Time and Embedded Technology and Applications*. IEEE, 2006, pp. 81–90.

[12] Tiwari et al. "Instruction level power analysis and optimization of software". In: *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 13.2-3 (1996), pp. 223–238.

[13] Park et al. "A Multi-Granularity Power Modeling Methodology for Embedded Processors". In: *IEEE Trans. VLSI Systems* 19.4 (Apr. 2011), pp. 668–681.

[14] Zaid A. M. Al-Khatib. "Operand Value Based Modeling and Estimation of Dynamic Energy Consumption of Soft Processors in FPGA". PhD thesis. Concordia University, Montreal, Canada, 2013.

[15] Sarta et al. "A data dependent approach to instruction level power estimation". In: *Proc. Workshop on Low-Power Design*. IEEE, 1999, pp. 182–190.

[16] Ascia et al. "An Instruction-Level Power Analysis Model with Data Dependency". In: *VLSI Design* 12.2 (2001), pp. 245–273.

[17] Penolazzi et al. "Energy and Performance Model of a SPARC Leon3 Processor". In: *Euromicro Conf. Digital System Design, Architectures, Methods and Tools* (Aug. 2009), pp. 651–656.

[18] Kojima et al. "Power analysis of a programmable DSP for architecture/program optimization". In: *Symp. Low Power Electronics. Digest of Technical Papers*. IEEE, 1995, pp. 26–27.

[19] F.N. Najm. "A survey of power estimation techniques in VLSI circuits". In: *Trans. VLSI Systems* 2.4 (Dec. 1994), pp. 446–455.

[20] Devadas et al. "Estimation of power dissipation in CMOS combinational circuits using boolean function manipulation". In: *Trans. Computer-Aided Design* II.3 (1992).

[21] Michael S. Hsiao. "Peak Power Estimation Using Genetic Spot Optimization for Large VLSI Circuits". In: *Design, Automation and Test in Europe*. March. 1999, pp. 175–179.

[22] Hsiao et al. "K2: an estimator for peak sustainable power of VLSI circuits". In: *Low Power Electronics and Design* (1997).

[23] Burch et al. "A Monte Carlo approach for power estimation". In: *Trans. VLSI Systems* 1.1 (Mar. 1993), pp. 63–71.

[24] W. Feller. “The fundamental limit theorems in probability”. In: *Bulletin of the American Mathematical Society* 51.11 (Nov. 1945), pp. 800–833.

[25] Evmorfopoulos et al. “A Monte Carlo approach for maximum power estimation based on extreme value theory”. In: *Trans. Computer-Aided Design of Integrated Circuits and Systems* 21.4 (Apr. 2002), pp. 415–432.

[26] M. Pedram. “Maximum power estimation using the limiting distributions of extreme order statistics”. In: *Proc. 1998 Design and Automation Conference.* Ieee, 1998, pp. 684–689.

[27] Dargie et al. “A Probabilistic Model for Estimating the Power Consumption of Processors and Network Interface Cards”. In: *Int. Conf. Trust, Security and Privacy in Computing and Communications* (July 2013), pp. 845–852.

[28] Pallister et al. “BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms”. 2013.

[29] David May. *The XMOS XS1 Architecture*. 2009.

Attachment D2.3.3

On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs

Under review at the ACM journal *Transactions on Architecture and Code Optimization (TACO).*

arXiv:1510.07095v1 [cs.PL] 24 Oct 2015

On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs

Kyriakos Georgiou, Steve Kerrison, Kerstin Eder

University of Bristol

Abstract. There is growing interest in lowering the energy consumption of computation. Energy transparency is a concept that makes a program's energy consumption visible from software to hardware through the different system layers. Such transparency can enable energy optimizations at each layer and between layers, and help both programmers and operating systems make energy aware decisions. The common methodology of extracting the energy consumption of a program is through direct measurement of the target hardware. This usually involves specialized equipment and knowledge most programmers do not have. In this paper, we examine how existing methods for static resource analysis and energy modeling can be utilized to perform Energy Consumption Static Analysis (ECSA) for deeply embedded programs. To investigate this, we have developed ECSA techniques that work at the instruction set level and at a higher level, the LLVM IR, through a novel mapping technique. We apply our ECSA to a comprehensive set of mainly industrial benchmarks, including single-threaded and also multi-threaded embedded programs from two commonly used concurrency patterns, task farms and pipelines. We compare our ECSA results to hardware measurements and predictions obtained based on simulation traces. We discuss a number of application scenarios for which ECSA results can provide energy transparency and conclude with a set of new research questions for future work.

1 Introduction

A substantial amount of effort has been invested into predicting the execution time of a program. However, there is little in the complementary area of energy consumption. Such information can be of significant value during the development and life time of critical systems. For example, energy consumption information can be crucial for devices that depend on unreliable, limited sources of power such as energy harvesters. Giving consideration to the energy consumption of a system at development time can avoid potential system failures due to inadequate energy supply at runtime. For systems that operate on a battery, this can provide a good approximation of the time frame in which the battery needs replacement.

The energy consumption of a program on specific hardware can always be determined through physical measurements. Although this is potentially the most accurate method, it is often not easily accessible. Measuring energy consumption can involve sophisticated equipment and special hardware knowledge. Custom modifications may be needed to probe the power supply. These conditions make it very difficult for the majority of software developers to assess a program's energy consumption.

Static Resource Analysis (SRA) provides an alternative to measurement. Significant progress has been made in the area of Worst Case Execution Time (WCET) prediction using static techniques that determine safe upper bounds for the execution time of programs. This naturally leads to the question of whether similar techniques can be used to bound the energy consumption of programs, and, if so, how effective they can be. A popular approach used for WCET is the Implicit Path Enumeration Technique (IPET), which retrieves the worst case control flow path of programs based on a timing cost model. Instead, in [1], an energy model that assigns energy values to blocks of Instruction Set Architecture (ISA) code is used, and the authors claim to statically estimate Worst Case Energy Consumption (WCEC).

However, in contrast to timing, energy consumption is data sensitive, i.e. the energy cost of executing an instruction varies depending on (the circuit switching activity caused by) the operands used. This effect is not captured in non data sensitive energy models, i.e. models that assign a single energy consumption value to each entity, e.g. to each instruction. Such models typically are characterized based on averages obtained from measuring the energy consumed when random data is being processed [2]. Alternatively, the highest energy consumption measured could be used for model characterization. As a consequence, when a non data sensitive energy model is used, the safety of the bounds retrieved from a worst case path static analysis might be undermined by worst case data scenarios for models that provide average energy consumption costs. On the other hand, the use of worst case models is known to lead to over-estimations [3] affecting the tightness of the retrieved bounds, because it is unlikely that the data that triggers the worst case energy consumption for one instruction also does this for all subsequent instructions in a program. This problem applies to all previous works that perform static analysis for energy consumption, as they combine non data sensitive bound analysis techniques with non data sensitive energy models. In [1] static analysis for WCEC is claimed by maximising the switching activity factor for each simulated component. However, the model abstraction level used does not guarantee that a physical implementation would behave in this way. We use a model with a similar constraint, where the data input that would trigger the worst case per instruction is not known, and so cannot assert the results to be WCEC.

In this paper we thoroughly investigate the value and limitations of using IPET in combination with non data sensitive energy models to perform Energy Consumption Static Analysis (ECSA) in the context of deeply embedded hardware, in our case the XMOS XS1-L "Xcore" [4]. The Xcore is a multi-threaded

deeply embedded processor with time-deterministic instruction execution. Such systems are simpler than general purpose processors and favor predictability and low energy consumption over maximizing performance. The absence of performance enhancing complexity at the hardware level, such as caches, provides us with an ideal setting to evaluate ECSA.

We base our investigation on an ISA-level multi-threaded energy model for the Xcore [5]. This model was characterized using constrained pseudorandom input data and associates a single averaged energy cost with each instruction in the XMOS ISA. We refined this model to one that is well suited for ECSA as it represents both static and dynamic power contributions to better reflect inter-instruction and inter-thread overheads; this improved model accuracy by an average of 4%. In addition to using this model for ECSA, we also used it to compare ECSA results with predictions based on statistics obtained from simulation traces.

For our study we have developed an IPET-based ECSA, which we use together with the non data sensitive ISA-level energy model described above, to predict the energy consumption of single and multi-threaded programs. With respect to the latter we focus on two commonly used concurrency patterns in embedded programs, task farms and pipelined programs with evenly distributed workloads across threads. In addition, we have developed a novel mapping technique to lift our ISA-level energy model to a higher level, the intermediate representation of the compiler, namely LLVM IR [6], implemented within the LLVM tool chain [7]. This enables ECSA to be performed at a higher level than ISA, thus introducing energy transparency into the compiler tool chain by making energy consumption information accessible directly to the optimizer.

Performing ECSA on multi-threaded programs and at the LLVM IR allows a comprehensive analysis of the energy consumption predictions that can be obtained using this technique. Our ECSA technique is evaluated using a set of single- and multi-threaded benchmarks, mainly selected from a number of industrial embedded applications. Our results show that accurate energy estimations can be retrieved at the ISA level. The mapping technique allowed for energy consumption transparency at the LLVM IR level, with accuracy keeping within 1% of ISA-level estimations in most cases. The main contributions of this paper are:

1. Modeling the target architecture to capture its behavior statically, including refinement of an existing ISA-level energy model, improving its accuracy by around 4% (Section 3.1);
2. Formalization and implementation of a novel mapping technique that lifts an ISA-level energy model to a higher level, the intermediate representation of the LLVM compiler, which allows ECSA of programs at the LLVM IR level (Section 3.2);
3. ECSA on a set of multi-threaded programs (Section 3.5), focusing on task farms and pipelines, two commonly used concurrency patterns in embedded computing;

4. Comprehensive evaluation of our ECSA on a set of industrial benchmarks and detailed analysis of results (Section 4.2);
5. Discussion of the practical value and limitations of how such analysis can be useful for software developers, compiler engineers, development tools and Real Time Operating Systems (RTOS) (Section 4.3).

The rest of the paper is organized as follows. Section 2 critically reviews previous work on energy modeling and SRA, with a focus on SRA for energy consumption and the effects of combining non data sensitive bound analysis techniques with non data sensitive energy models. Section 3 introduces in detail the components of our analysis, in particular the formalization and implementation of our mapping technique, and how ECSA can be applied to multi-threaded programs. Our experimental evaluation methodology, benchmarks and results are presented and discussed in Section 4. Section 5 concludes the paper, outlines opportunities for future work and raises a number of research questions to stimulate further research in ECSA.

2 Background

The work presented in this paper builds upon two areas: processor energy modeling and SRA. This section establishes the background work of both.

2.1 Energy modeling of embedded processors

Energy modeling can be performed at various levels of abstraction, from gate- or transistor-level in detailed hardware simulation [8], up to high-level modeling of whole applications. Although the hardware components are responsible for power dissipation and thus consumption of energy, the behavior of that hardware is largely controlled by the software running upon it. As such, writing software that makes efficient use of the underlying hardware has been identified as the most important step in energy efficient software development [9]. For energy modeling to be useful to a software developer, models must convey information that can be related to the code the developer is writing.

The ISA is a practical level of abstraction for energy modeling of software, because it expresses the underlying operations performed by the hardware and its relationship with the intent of the software. In [2] an ISA-level energy model is proposed that obtained energy consumption data through hardware measurements of large loops of individual instructions. The total cost of a program is composed of instruction costs, inter-instruction costs (the effects of switching from one instruction to the next), and externally modeled behaviour such as activity in the memory hierarchy.

This work was initially applied to x86 and SPARC architecture processors, operating with an accuracy of within 10% of the hardware. It was extended to form a framework for architecture-level power analysis, Wattch [10]. The Sim-Panalyser [11] uses a similar approach, built on top of the SimpleScalar architecture simulation framework [12].

If additional characteristics of processor activity are considered, such as bit-flips in the data-path, a more accurate data-dependent model can be produced, such as that of [13, 14]. This requires more detailed information from simulation in order to supply additional model parameters, but has been demonstrated to bring accuracy to within 1.7% of the hardware. It is still an abstraction away from the internal switching activity of functional units, however. Observing the results in [15], some functional units may be more dependent on their internal structure than input/output Hamming weight with respect to data-dependent power.

Using similar approaches to Steinke and Tiwari, additional processor architectures such as VLIW DSPs have also been modeled, with 4.8% [16] and 1.05% [17] accuracy. Alternative approaches to modeling include representing activity in terms of the processor's functional blocks [17], energy profiling of the most commonly used software library functions [18], and construction of model parameters through linear regression [19].

In [1], a micro-architectural energy model was created, considering functional units activity, clock gating and pipeline progression for a simulated processor. This model was used for WCEC static analysis. To retrieve safe bounds, the switching activity factor was set to the maximum, 1.0, for each component. This led to significant energy consumption over-estimations in some cases, up to 33%, and assumes that the model accurately reflects a physical implementation.

In architectures where performance counters are available, these can be used to characterize the processor energy consumption based on the conditions affecting these counters, such as cache misses and pre-fetches. Simulations that model these performance events can then be used to predict the energy consumption of an application. This has been applied to processors of various levels, from embedded XScale [20] to Xeon Phi accelerators [21].

The discussed approaches achieve varying levels of accuracy, all within a 10% error margin. The comparison points vary between methods, so the accuracies are not necessarily directly comparable. However, the prior work motivates new models to achieve a similar margin. In many of the above examples, the models target a 'typical' energy characterization, where the modeled energy consumption is based on random or non-exhaustive input data sets. For a given application, some additional error margin will be introduced based on the particular characteristics of its dataset. This forms a part of the model error, in addition to the errors arising from the abstractions applied in each model type. The work presented in this paper, which examines multiple abstraction levels, seeks to identify each point at which inaccuracies may be introduced into the estimation process. This is important to assesses usefulness of estimations produced by static analysis, and will be discussed in Section 4.2

2.2 Static Resource Analysis

SRA is a methodology to determine the usage of a resource (usually time or energy or both) for a specific task when executed on a piece of hardware, without actually executing the task. This requires accurate modeling of the hardware's

behavior in order to capture the dynamic functional and non-functional properties of task execution. Determining these properties accurately is known to be undecidable in general. Therefore, to extract safe values for the resource usage of a task, a sound approximation is needed [22, 23].

SRA has been mainly driven by the timing analysis community. Static cost analysis techniques based on setting up and solving recurrence equations date back to Wegbreit's [24] seminal paper, and have been developed significantly in subsequent work [25, 26, 27, 28, 29, 30]. Other classes of approaches to cost analysis use dependent types [31], SMT solvers [32], or size change abstraction [33].

For performing an accurate WCET static analysis, there are four essential components [22]:

1. Value analysis: mainly used to analyze the behavior of the data cache.
2. Control flow analysis: used to identify the dynamic behavior of a program.
3. Low level or processor behavior analysis: attempts to retrieve timing costs for each atomic unit on a given hardware platform, such as an instruction or a basic block (BB) in a Control Flow Graph (CFG) for a processor.
4. Calculation: uses the results from the two previous components to estimate the WCET. Most common techniques used for calculation of the WCET are the IPET, the path-based techniques and the tree-based methods [34].

Three of the above components, namely the control flow analysis, low level analysis and calculation, are adopted in our work and will be further explained in Section 3.

IPET is one of the most popular methods used for WCET analysis [35, 36, 37, 34, 38]. In this approach, the CFG of a program is expressed as an Integer Linear Programming (ILP) system, where the objective function represents the execution time of the program. The problem then becomes a search for the WCET by maximizing the retrieved objective function under some constraints on the execution counts of the CFG's basic blocks. The main advantage of this technique is the ability to determine the basic blocks in the worst case execution path and their respective execution counts without the need to extract the explicit worst execution path (ordered list of the executed basic blocks). This is more efficient than path based techniques for retrieving WCET bounds [34].

In the presence of caches or a complex processor pipeline, the ILP solving complexity can increase dramatically, making IPET not practical for WCET. Abstract interpretation [39], a technique used to facilitate data flow analysis, can then be used in conjunction with IPET to allow WCET in such cases [36].

Although significant research has been conducted in static analysis for the execution time estimation of a program, there is little on energy consumption. One of the few approaches [40] seeks to statically infer the energy consumption of Java programs as functions of input data sizes, by specializing a generic resource analyzer [29, 41] to Java bytecode analysis [42]. However, a comparison of the results to actual measurements was not performed. Later, in [43], the same generic resource analyzer was instantiated to perform energy analysis of XC programs [44] at the ISA level based on ISA-level energy models and including a comparison to actual hardware measurements. However, the scope of this

particular analysis approach was limited to a small set of simple benchmarks because information required for the analysis of more complex programs, such as program structure and types, is not available at the ISA level. The analysis presented in this paper does not rely on such information. A similar approach, using cost functions, was used in [45]. The analysis was performed at the LLVM IR level, using the mapping technique that we formalise and describe in full detail for the first time in this paper. Although the range of programs that could be analyzed was improved compared to [43], the complexity of solving recurrence equations for analysing larger programs proved a limiting factor.

In [1] the WCEC for a program was inferred by using the IPET first introduced in [35]. They claim WCEC analysis, and experimental results indicate that all energy estimations over-approximate the energy consumptions retrieved from simulation. However, infeasible paths were not excluded from analysis, and there is no guarantee that the comparison test cases used in simulation were the actual worst cases.

Similarly, in [46] the authors attempt to perform static worst case energy consumption analysis for a simple embedded processor, the `Cortex M0+`. This analysis is also based on IPET combined with a so called absolute energy model, an energy model that is said to provide the "maximum energy consumption of each instruction". The authors argue that they can retrieve a safe bound. However, this is demonstrated on a single benchmark, `bubblesort`, only. The bound is 19% above a single hardware measurement; the authors acknowledge that this approach leads to over-approximations. Furthermore, the hardware measurement used as a base line to evaluate the prediction obtained from static analysis only captures the algorithm's worst case complexity scenario, no information is given on the actual data to provide insight into the effect of data switching activity on energy consumption. This can be misleading, since two sets of different input data might have the same algorithmic worst case behavior, but can be very different with respect to their total energy consumption. In practice, this gives rise to a range of energy consumption measurements for different input data all triggering the algorithmic worst case path. For instance, for the Xcore architecture the energy consumption of the `MatMult 4` threads benchmark [47] for the same size of matrices, ranges from 4.1 to 4.9 nJ depending on the used data. We have closely investigated this and discuss our findings in Figure 5.

All of the reviewed previous works for static energy consumption analysis used worst case path analysis methods combined with non input pattern-depended energy models. Currently, there is no practical method to perform average case static analysis [48]. One of the most recent works towards average case SRA, demonstrates that compositionality combined with the capacity for tracking data distributions unlocks the average case analysis, but novel language features and hardware designs are required to support these properties [49]. Furthermore, developing a data sensitive energy model requires detailed knowledge of and access to the RTL, since the power dissipation is highly depended on the switching activity inside the circuits [50]; this is a challenge in itself. This situation has motivated us to conduct a comprehensive study to fully understand the

value and limitations of ECSA, using IPET-based analysis in combination with a single cost energy model, for both single and multi-threaded code at the ISA and LLVM IR levels of abstraction.

3 Energy Consumption Static Analysis

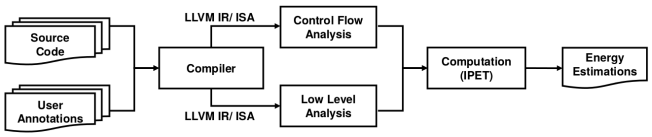


Fig. 1: Overview of our energy consumption static analysis.

Figure 1 shows the ECSA process for both, analysis at ISA and LLVM IR level. The source code together with any user annotations (e.g. to provide loop bounds) is sent to the compiler which emits the LLVM IR and the ISA code. Low level analysis, analysis of program control flow and computation of the energy consumption estimations is then applied on both levels. For the LLVM IR analysis an extra step is required at the compilation phase for the energy characterization of the LLVM IR instructions as detailed in Section 3.2. In the rest of this section we briefly introduce each ECSA stage.

3.1 Low Level Analysis

This stage aims to model the micro-architecture dynamic behavior of the processor based on an ISA-level energy model.

XMOS Xcore ISA level Energy Modeling The Xcore processor is hardware multi-threaded, providing inter-thread communication and I/O port control directly in the ISA. It is event-driven; busy waiting is avoided in favor of hardware scheduled idle periods. This makes the Xcore well suited to embedded applications requiring multiple hardware interfaces with real-time responsiveness.

The underlying energy model for this work is captured at the ISA level. Individual instructions from the ISA are assigned a single cost each. These can then be used to compute power or energy for sequences of instructions. The model also captures the cost of thread scheduling performed by the hardware, in accordance with a series of profiling tests and measurements, because it influences the energy consumption of program execution. Instructions from runnable threads are scheduled round-robin by the hardware. To avoid data hazards, the processor's four stage pipeline may only contain one instruction from each thread. If the number of runnable threads is less than four, there will be empty pipeline stages.

The modeling technique is built upon [2], as discussed in Section 2.1, which is adapted and extended to consider the scheduling behavior and pipeline characteristics of the Xcore [5]. A new version of this model that is well suited for static analysis has been developed. It represents energy in terms of static and dynamic power components to better reflect inter-instruction and inter-thread overheads. This has improved model accuracy by an average of 4%.

$$E_{\text{prg}} = (P_s + P_{di}) \cdot T_{\text{idl}} + \sum_{i \in \text{prg}} \left(\frac{P_s + P_i M_{N_p} O}{N_p} \cdot 4 \cdot T_{\text{clk}} \right), \text{ where } N_p = \min(N_t, 4) \tag{1}$$

In Equation (1), E_{prg} is the energy of a program, formed by adding the energy consumed at idle to the energy consumed by every instruction, i, executed in the program. At idle, only a base processor power, the sum of its static, P_s, and dynamic idle power, P_{di}, is dissipated for the total idle time, T_{idl}. For each instruction, static power is again considered, with additional dynamic power for each particular instruction, P_i. The dynamic power contribution is then multiplied by a constant inter-instruction overhead, O, that has been established as the average overhead of instruction interleaving. This is then multiplied by a scaling factor to account for the number of threads in the pipeline, M_{N_p}. The result is divided by the number of instructions in the pipeline, which is at most four and is dependent upon the number of active threads, N_t. Each instruction completes in four cycles, so $4 \cdot T_{\text{clk}}$ gives the energy contribution of the given instruction, based on the calculated power.

When more than four threads are active, the issue rate of instructions per thread will be reduced. The energy model accounts for this with the min term in Equation (1). From a purely timing perspective, the latency between instruction issues for a thread is $\max(N_t, 4) \cdot T_{\text{clk}}$. This property means that instructions are time-deterministic, provided the number of active threads is known. A thread may stall in order to fetch the next instruction. This is also deterministic and can be statically identified [51, pp. 8–10]. These instruction timing rules have been used in simulation based energy estimation, and are also utilized in the multi-threaded static analysis performed in this paper. Both simulation and static analysis must be able to determine N_t, the number of active threads, in order to accurately estimate energy consumption.

A limited number of instructions can be exceptions to these timing rules. The divide and remainder instructions are bit-serial and take up to 32 cycles to complete. Resource instructions may block if a condition of their execution is not met, e.g. waiting on inbound communication causes the instruction's thread to be de-scheduled until the condition becomes satisfied. This paper focuses its contributions on fully predictable instructions, with timing disturbances from communication forming future work.

The cost associated with an instruction represents the average energy consumption obtained from measuring the energy consumed during instruction execution based on constrained pseudo-randomly generated operands using the setup described in [5]. Thus, this model does not explicitly consider the range of

input data values and how this may affect consumption. Empirical evidence indicates that such factors can contribute to the dynamic energy consumption [3]. The implications of using a random data constructed single value energy model with a bound SRA for ECSA are discussed in Section 4.2.

Utilizing the Xcore energy model in static analysis To determine the energy consumption of a program based on Equation (1) the program's instruction sequence, $\langle i_1, \ldots, i_n \rangle$, the idle time T_{idl}, and the number of active threads N_p during instruction execution must be known. In [5] Instruction Set Simulation (ISS) was used to gather full trace data or execution statistics to obtain these parameters. In this work we use ISS only as a reference for comparison of ECSA results, with a second reference being direct hardware measurement.

ECSA thus needs to extract the CFGs for each thread and identify the interleavings between them. This allows for each instruction in the program to identify the N_p component in Equation (1). It also allows to estimate the total idle time, T_{idl} of the program. For single-threaded programs the energy characterization of the CFG is straightforward as there is no thread interleaving. The IPET can be directly applied on the energy characterized CFG to extract a path that bounds the energy consumption of the program, as described in Section 3.4. For arbitrary multi-threaded programs, energy characterizing the CFGs of each thread using static analysis is challenging. We have therefore concentrated on two commonly used concurrency patterns, task farms and pipelines, which we use with evenly distributed workloads across threads.

In addition to the instructions defined in the ISA, a Fetch No-Op (FNOP) can also be issued by the processor. These occur deterministically [51, pp. 8–10]. FNOPs can have a significant impact on energy consumption, particularly within loops. To account for FNOPs in static analysis, the program's CFG at ISA level is analyzed. An instruction buffer model is used to determine where FNOPs will occur in a basic block. However, one particular FNOP case is dependent on the dynamic branching behavior of the program, in which case we overestimate FNOPs. Further implementation details on FNOPs modeling can be found in [52].

3.2 Mapping an ISA Energy Model to LLVM IR

Although substantial effort has been devoted to ISA energy modeling, there is little research into modeling at higher levels of program representation, where precision can decrease. In [53], statistical analysis and characterization of LLVM IR code is performed. This is combined with instrumentation and execution on a target host machine to estimate the performance and energy requirements in embedded software. Transferring the LLVM IR energy model to a new platform requires performing the statistical analysis again. The mapping technique we present here is fully portable. It requires only the adjustment of the LLVM mapping pass to the new architecture. Furthermore, our LLVM IR mapping technique provides on-the-fly energy characterization that allows to take into

consideration the compiler behavior, CFG structure, types and other aspects of instructions.

Formal specification of the mapping Our mapping technique determines the energy characteristics of LLVM IR instructions. Thus, mapping links LLVM IR instructions with machine specific ISA instructions. ISA level energy models can then be propagated up to LLVM IR level, allowing energy consumption estimation of programs at that level. We formalize the mapping as follows. For a program P, let

$$\text{IRprog}_L = \{1, 2, ..., n\} \tag{2}$$

be the ID numbers of P's LLVM IR instructions and therefore

$$\text{IRprog} = \langle ir_1, ir_2, ..., ir_n \rangle \tag{3}$$

is the sequence of LLVM IR instructions for P.

$$\text{T}_{arch}(\text{IRprog}) = \text{ISAprog} \tag{4}$$

is an architecture specific compiler back end that can translate the IRprog to

$$\text{ISAprog} = \langle (isa_1, m_1), (isa_2, m_2), ..., (isa_k, m_l) \rangle \text{ where } m_1, m_2, ..., m_l \in \text{IRprog}_L \tag{5}$$

which is the sequence of ISA instructions for P, together with the ID of the LLVMIR instruction from which each isa_k originated. If an isa_k comes from more than one LLVM IR instructions, then T_{arch} chooses the ID of one of them to assign to isa_k.

$$\text{M}(ir_i) = \{isa_j | ir_i \in \text{IRprog} \wedge \text{ISAprog} = \text{T}_{arch}(\text{IRprog}) \wedge (isa_j, i) \in \text{ISAprog}\} \text{ and } \forall\ ir_n, ir_k \in \text{IRprog} \wedge ir_n \neq ir_k \text{ then } \text{M}(ir_n) \cap \text{M}(ir_k) = \emptyset \tag{6}$$

is a mapping function that captures a 1:m relation from IRprog to ISAprog instructions. Therefore,

$$\text{E}(ir_i) = \sum_{isa_j \in S} \text{E}(isa_j) \text{ where } ir_i \in \text{IRprog} \wedge isa_j \in \text{ISAprog} \wedge \text{S} = \text{M}(ir_i) \tag{7}$$

represents the energy consumption of an LLVM IR instruction as the sum of the energy consumed by all ISA instructions associated with that LLVM IR instruction.

By instantiating the above mapping to a specific architecture, LLVM IR energy characterization can be retrieved. The accuracy of this characterization can vary for different architectures. If the accuracy is not adequate, then a tunning phase can be introduced to account for any specific compiler or architecture behavior. An example of such tunning is given in the next section, which accounts for *phi-nodes* and FNOPs.

Xcore mapping instantiation and tuning In our case, the T_{arch} function is the XMOS tool chain lowering phase that translates the LLVM IR to Xcore specific ISA. Our mapping implementation leverages the debug mechanism in the XMOS compiler tool chain, in order to enable T_{arch} to assign to each ISA instruction the ID of the LLVM IR instruction it originated from. This is typically used by the programmer to identify and fix problems in application code. Debug symbols are created during compilation to assist with this. These symbols are propagated to all intermediate code layers and down to the ISA code. Debug symbols can express which programming language constructs generated a specific piece of machine code in a given executable module. In our case, these symbols are generated by the front end of the XMOS compiler in standard DWARF format [54]. These are transformed to LLVM metadata [55] and attached to the LLVM IR.

During the lowering phase of compilation, LLVM IR code is transformed to a target ISA by the back end of the compiler, with debug information stored alongside it as LLVM metadata. Naturally, the accuracy of debug information in the output executable is reduced if the number of optimization passes is increased. This is due to portions of the initial LLVM IR either being discarded or merged during these passes.

Tracking this information gives an $n : m$ relationship between instructions at the different layers, because source code instructions can be translated to many LLVM IR instructions, and these again into many ISA instructions. This $n : m$ relation prevents ECSA from providing accurate energy values and therefore the mapping introduced in Section 3.2, requires Equation (6) to create an 1:m relation between the LLVM IR and ISA code.

To address this issue, we created an LLVM pass that traverses the LLVM IR and replaces source location information with LLVM IR location information. The location information represents the IRprog_L in Equation (2). The LLVM pass runs after all optimization passes, just before emitting ISA code. The optimized LLVM IR is closer in structure to the ISA code than the unoptimized version. Using this method a $1 : m$ mapping between LLVM IR instructions and ISA instructions can be extracted by Equation (6). Once the mapping has been performed for a program, the energy values for groups of ISA instructions are aggregated and then associated with their single corresponding LLVM IR instruction using Equation (7).

An example mapping is given in Figure 2. On the left hand side is a part of the LLVM IR CFG of a program, which represents the IRprog in Equation (3), along with the debug location, LLVMIR_L in Equation (2), for each LLVM IR instruction. The right hand side shows the corresponding ISA CFG, together with the debug locations for each ISA instruction, given by T_{arch}. The coloring of the instructions demonstrates the mapping between the two CFGs' instructions using Equation (6). Now, one LLVM IR instruction is matched to many ISA instructions, but each ISA instruction is mapped to only one LLVM IR instruction. Some LLVM IR instructions are not mapped, because they are removed during the lowering phase of the compiler. This mapping also guarantees that all

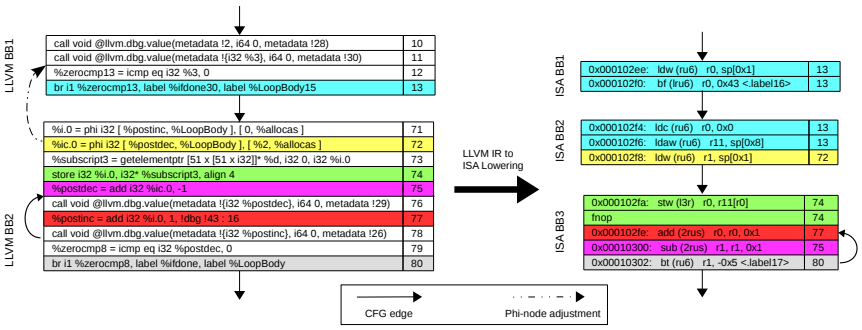


Fig. 2: Fine grained 1:m mapping including our LLVM mapping pass.

ISA instructions are mapped to LLVM IR, so there is no loss of recorded energy between the two levels.

Additional optimizations are performed during the lowering phase from LLVM IR to ISA, such as peephole optimizations and target specific optimizations. These can affect the mapping, but not to the same degree as the LLVM optimizations. A tuning phase can be introduced after the mapping, to account for them.

The mapping instantiation for the Xcore architecture was able to provide an average energy estimation deviation of 6% from the predictions on the ISA level. An additional tuning phase is introduced after the mapping, to account for specific compiler and architecture behavior. This improved the mapping accuracy, narrowing the gap between ISA and LLVM IR energy predictions to an average of 1% as discussed in 4.3.

LLVM IR *phi-nodes* are an example of such tuning. *Phi-nodes* can be introduced at the start of a BB as a side effect of the Single Static Assignment (SSA) used for variables in the LLVM IR. A *phi-node* takes a list of pairs, where each pair contains a reference to the predecessor block together with the variable that is propagated from there to the current block. The number of pairs is equal to the number of predecessor blocks to the current block. A *phi-node* can create inaccuracies in the mapping when LLVM IR is lowered to ISA code that no longer supports SSA, because it can be hoisted out from its current block to the corresponding predecessor block. For blocks in loops this can lead to a significant analysis error.

Whenever the tuning phase is able to track these cases, it can adjust the energy figures for each LLVM IR BB accordingly. An example of this is given in Figure 2 at debug location number 72. Its corresponding ISA instruction is hoisted out from the loop BB `ISA BB3` and into `ISA BB2`. This is tracked by the mapping, and the equivalent hoisting is done at LLVM IR level, thus correctly assigning energy values to each LLVM IR block. Similar errors can be introduced by branching LLVM IR instructions with multiple targets, since in the Xcore

ISA only single target branches are supported. This is also handled during the mapping phase.

As discussed in Equation (1), FNOPs can be issued by the processor and this can be statically determined at the ISA level. LLVM IR has no way to represent this. Ignoring them can therefore lead to a significant underestimation of energy at LLVM IR level. To address this, FNOPs in the lowered ISA code are assigned the debug location of an adjacent ISA instruction in the same BB by the tuning phase, thus they are accounted for in the mapped LLVM IR block.

LLVM IR instructions can be combined into a single ISA instruction. An example of such instructions are the add and multiplication ones which can be translated to the Xcore `macc` (multiply-accumulate) ISA instruction. The T_{arch} will assign the energy cost to only one of the LLVM IR instructions. Although, this is adequate for the energy characterization of LLVM IR basic blocks, if needed the tuning phase allows to associate the cost with both instructions.

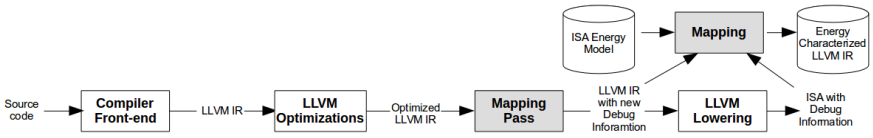


Fig. 3: Overview of the mapping process.

An overview of the mapping technique is given in Figure 3. Our mapping pass is introduced into the compilation process after LLVM optimizations. The pass also includes tuning. The mapping phase implements Relations 6 and 7. It runs after the LLVM lowering phase and maps LLVM IR instructions with the new debug locations to the emitted ISA instructions. The ISA energy model is then used to accumulate the energy value of each LLVM IR instruction based on its mapped ISA instructions.

3.3 Control Flow Analysis

This component aims to capture the dynamic behavior of the program and associates CFG BBs with the information needed for the computation step of analysis. IPET requires the CFG and call graph of a program to be constructed at the same level as the analysis. At LLVM IR level, the compiler can generate them. At ISA level a tool was created to construct them. To detect BBs that belong to a loop or recursion, we adopted and extended the algorithm in [56]. The CFGs are annotated according to the needs of the IPET described in Section 3.4. Finally, the annotated CFGs are used in the computation step to produce ILP formulations and constraints.

3.4 Computation

The IPET adopted in our work to estimate the energy consumption of a program is based on [57]. To construct the ILP system needed for IPET, we use information produced from the previous two components. The method of ILP formulation along with the constraints needed to bound the problem and optimize it's solution can be found in the seminal paper [57]. To infer the energy consumption, instead of using the time cost of a CFG basic block we are using its energy cost, as provided by the respective energy model.

Constraints are used to capture information that can affect a program's dynamic behavior, such as bounded loop iterations, or path information, such as infeasible paths. Usually, this information can only be specified by the programmer, as it depends on the program semantics and cannot be extracted by the static analysis. The minimum required user input to enable bounding of the problem is the declaration of loop bounds. This is also standard practice in timing analysis [22]. Providing this kind of information is usually easy, as the loop bounds are typically known by programmers of timing critical embedded programs. Further constraints, such as denoting infeasible paths in a CFG, can be provided to extract more accurate estimations. The user provides this information as source code annotations. The annotation language used in this work can be found in [58].

3.5 Analysis of multi-threaded programs

In this paper we present the first steps towards ECSA of multi-threaded programs. Two concurrency patterns are considered: replicated threads with no inter-thread communication, working on different sets of data (task farms), and pipelines of communicating threads. For both cases, we consider evenly distributed, balanced work loads. In the former case, an example use is simultaneously processing multiple independent data. In the latter case, pipelining enables parallelism to be used to improve performance when processing a single data stream.

There is a fundamental difference when statically predicting the case of interest (worst, best, average case) for time and for energy for multi-threaded programs. Generally, for time only the computations that contribute to the path forming the case of interest must be considered. For energy, all computations taking place during the case of interest must be considered. For instance, in an unbalanced task farm, the WCET will be equivalent to the longest running thread. To bound energy, the energy consumption of each thread needs to be aggregated. This is harder since the static analysis needs to determine the number of active threads at each point in time in order to apply the energy model from Equation (1) and characterize the CFG of each thread. Then, IPET can be applied to each thread's CFG, extracting energy consumption bounds. Aggregating these together will give a loose upper bound on the program's energy consumption, meaning that the safety of the bound cannot be guaranteed.

In our balanced task farm examples, all task threads are active in parallel for the duration of the test. Thus, the number of active threads is constant, giving a constant N_t, used to determine the pipeline occupancy scaling factor, M, in Equation (1). For balanced pipelined programs, we consider the continuous, streaming data use case, so the same constant thread count property holds. In both cases IPET can be performed on each thread's CFG and the results aggregated to retrieve the total energy consumption. In this work, core-local communication is considered, which uses the same instructions as off-core communication, but no external link energy needs to be accounted for. Therefore the core energy model provides sufficient data.

For multi-threaded programs with synchronous communications, to retrieve a WCET, IPET can be applied on a global graph, connecting the CFGs of all threads along communication edges. The communication edges can be treated by the IPET as normal CFG edges and WCET can be extracted by solving the formulated problem [38]. This will return a single worst case path across the global graph. Bounding energy in this way is not possible, as parallel thread activity over time needs to be considered. Here the task is even harder in comparison to programs without communication, as activity can be blocked if the threads' workloads are unbalanced. In this case, statically determining the number of active threads at each point in time is a hard challenge.

Although the concurrency patterns addressed here can be considered as easy targets for the ECSA, they are typical embedded use cases, and as is explained in Table 1, ECSA can provide sufficiently accurate information to enable energy aware decision making. Building on this, more complex programs will be analyzed in future work, such as unique non-communicating threads rather than replicated threads, unbalanced farm and pipeline workloads and other concurrency patterns. Such programs will feature varying numbers of active threads over the course of execution. In these cases the ECSA must be extended to perform analysis that extracts all the possible combinations of thread interleaving.

This work focuses on multi-threaded communication on a single core. However, for communicating threads, the channel communication paradigms that are used by the programs at the source code level and within the ISA can also be used in a multi-core environment, creating scope for the analysis of larger systems.

4 Experimental Evaluation

To evaluate our ECSA, a series of mainly industrial benchmarks were selected with representative test cases. Both our ECSA results and estimations from ISS using the same energy model are compared to hardware measurements. The benchmarks, evaluation methodology, results and further observations are discussed in this section.

4.1 Benchmarks

Our objective is to demonstrate the value of our ECSA for common industrial, deeply embedded applications. A complete list of all the 21 benchmarks' code and summary of their attributes, can be found in [47]. Benchmarks were compiled with `xcc` version 12 [59] at optimization level `O2`; the default for most compilers.

Deeply embedded processors do not typically have hardware support for division or floating point operations, using software libraries instead. Software implementations are usually far less efficient than their hardware equivalent, both in terms of execution time and energy consumption. The effect of these software implementations on energy consumption should be known by developers, therefore we include soft division and soft float benchmarks.

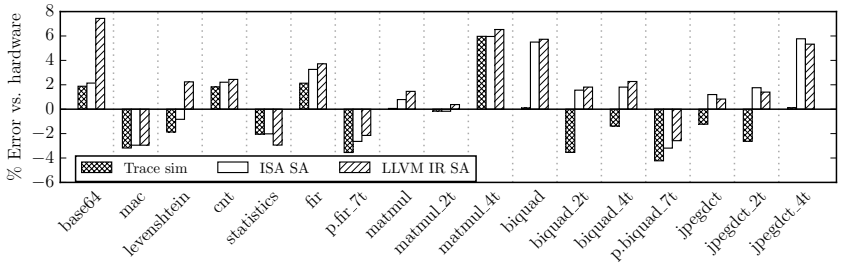
A radix-4 software divider, `Radix4Div` [60], is used. A less efficient version, `B.Radix4Div`, is added for comparison. This version omits an early return when the dividend is greater than 255. A consequence of excluding this optimization is that CFG paths become more balanced, with less variation between the possible execution paths. The effect of this on the energy consumption is discussed later in this section. For software floating point, single precision `SFloatAdd32bit` and `SFloatSub32bit` operations from [61] are analyzed.

To represent common signal processing tasks, `FIR` and `Biquad` benchmarks written for the Xcore processor [62], are analyzed. In addition, a series of open source benchmarks of core algorithmic functions were selected from the `MDH WCET` benchmark suite [63]. They were modified to work with our test harness and, in some cases, to make them more parametric to function input arguments. Some were extended to be multi-threaded task farms, where the same code runs on two or four threads. To extend our analysis to multi-threaded communication programs, we analyze pipelined versions of `FIR` and `Biquad`, each formed of seven threads. These programs are the preferred form for Xcore, as spreading the computation across threads allows the voltage and frequency of the core to be lowered, significantly reducing energy consumption with the same performance as the single threaded version.

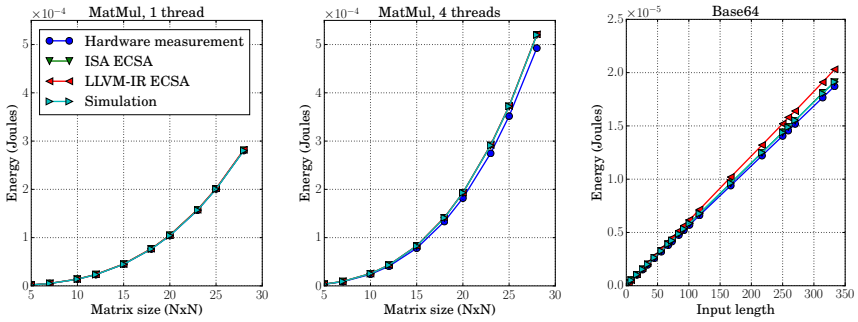
4.2 Results Analysis

The experimental results show several features, influenced by the level of multi-threading, the properties of the benchmarks, and the levels at which ECSA and modeling are performed. In this section we examine all of these in order to determine what influences ECSA accuracy at each level, highlighting both strengths and limitations.

Figure 4a presents the error margin of using our energy model with three energy estimation techniques compared to hardware energy measurements for our benchmarks. *Trace Sim* produces instruction traces from ISS, *ISA ECSA* uses the model for static analysis at the ISA level and *LLVM IR ECSA* uses our mapping technique to apply the model and analysis at LLVM IR level. For all benchmarks with multiple test parameters, the geometric mean of the errors



(a) All benchmarks.



(b) Parametric benchmarks.

Fig. 4: Hardware measurements compared to ECSA and ISA trace estimation.

is used. Figure 4b compares energy estimates to hardware measurements for a range of parameters in three parametric benchmarks.

For `Levenshtein`, `MatMult 1,2,4`, `Mac`, `Cnt` and `Base64` parametric energy consumption estimations can be determined, as discussed in Section 4.3. These are expressed in terms of a function over the number of loop iterations.

The parametric benchmarks are also more data sensitive, due to the use of matrices. The hardware energy measurements for all the benchmarks using matrices were obtained by using random data to initialize them. In order to investigate the effect of different random data, the measurements were repeated 500 times for each benchmark using a different seed each time for data generation. The maximum variation observed was in the range of the measurement error, less than 0.5%, and therefore the average of these measurements was used to compare against the predicted results. The effect of using non random data will be investigated in Figure 5. For the more industry oriented benchmarks (all the `FIR`, `Biquad` and `Jpegdct` versions) real sample data where used for the hardware measurements.

For the software division and floating point benchmarks, ECSA provides a constant energy consumption upper bound across all test cases, as they con-

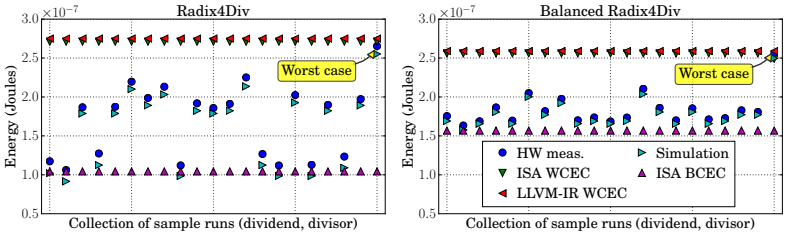


Fig. 5: Results for benchmarks with constant ECSA estimations across all test cases.

tain no loops that are directly affected by the functions' arguments. Figure 5 demonstrates this for `Radix4Div` and `B.Radix4Div`. Considering that IPET is intended to provide bounds based on a given cost model, in our case it tries to select the worst case execution paths in terms of the energy consumption. Therefore, the ECSA estimations seen in Figure 5 represent a loose upper bound on the benchmarks' energy consumption. Similar figures were also retrieved for the two `SoftFloat` benchmarks. These bounds, in most cases cannot be considered safe, as they might be undermined by the use of a non data sensitive energy model and analysis. However, they can still give the application programmer valuable guidance towards energy aware software development, as discussed in Section 4.3.

For the benchmarks in Figure 5, we sought test cases that exercise the average, best- and worst-case scenarios of each benchmark's algorithm, to compare the resultant range of energy consumption with our ECSA predictions. A good understanding of the underlying algorithms and information collected from the ISS traces was necessary to identify tests covering each scenario with certainty. This poses a challenge in guaranteeing that the cases of interest, such as worst case, have been exercised. For example, the `Radix4Div` benchmark takes two 16-bit parameters, forming a search-space of 2^{32} test cases. This was suitably small to perform an exhaustive search in order to capture the worst case empirically. However, the time cost of an exhaustive search precludes doing the same for many other benchmarks. For both `Radix4Div` variants, the upper bounds inferred by the IPET analysis are not only very close to the worst case retrieved by exhaustively searching the possible test cases, but are also safe.

Generally, for all results shown, a proportion of error is present in both forms of static analysis as well as simulation based energy estimation. The error in the ISS based estimation is a baseline for the best achievable error in static analysis, as ISS produces more accurate execution information. For all the benchmarks, the ISA ECSA results are over-approximating the trace based energy estimations. This applies also to the LLVM IR ECSA results with exception of the `statistics` benchmark. This over-approximation is a product of the bound analysis used which is trying to select the most energy costly CFG path based

on the provided cost model. A smaller difference between the ECSA results and the trace based energy estimations indicates that the execution path selected by the IPET fits better the actual execution path of a benchmark.

Measurement error analysis To assess the accuracy of ECSA predictions, reliable hardware measurements are required. We use a shunt resistor current sense circuit and data sampling hardware to obtain power dissipation with sub-milliwatt accuracy. The data capture process is explained in more detail in [5].

Measurements are subject to errors introduced through environmental factors. In particular, temperature and electrical noise can result in variations of the measured energy consumption for multiple runs of the same test. To measure the effect of these factors on our platform, we executed the `MatMult 4` thread benchmark 100,000 times. This benchmark was selected because it is particularly power intensive and likely to affect the device temperature the most. The variation observed on our hardware was less than 0.7% which we consider negligible and close to the error margin of our measurement equipment. These factors could have a more significant impact on other platforms. It is therefore important to examine them when performing ECSA.

The test harness introduces a small error by repeatedly calling the benchmark function within a loop. This is necessary to ensure an adequate number of power samples are taken during the test. However, the loop surrounding the call to the benchmark, together with the function call itself, introduce an overhead. This overhead can be significant, especially when the amount of computation in the loop body is low. To mitigate this overhead, we ensure that the loop is as efficient as possible and each benchmark sufficiently large in size. Finally, measurements were taken several times to ensure that results obtained were consistent, with less than 0.5% variation.

LLVM IR analysis accuracy This form of analysis is solely dependent on the accuracy of the mapping techniques presented in Section 3.2. As shown in Figures 4a and 5, for all benchmarks the LLVM IR ECSA results are within one percentage point error of ISA ECSA results, except for the `Base64` benchmark with a further 5.3 percentage points error. In this case the CFGs of the two levels were significantly different due to BBs introduced from branches in the ISA level CFG. This is one of the few cases where the mapper was unable to accurately track the differences between the two CFGs.

Multi-threading accuracy Three benchmarks, `MatMul`, `Biquad` and `JpegDCT`, were extended to multi-threaded versions, where each thread executes the same program and processes its own data stream. The computation performed and the energy consumption increases with the number of active threads, with a negligible change in execution time. The underlying energy model is parametric to the occupancy of the execution pipeline, which is determined by the number of running threads. As such, the estimations from the model and their relative

errors can differ when the number of threads is changed. For any given number of threads, the accuracy of the ECSA is influenced by the accuracy of the ISA level energy model.

In the case of pipelined benchmarks, `p.fir_7t` and `p.biquad_7t`, the energy model underestimates energy consumption by approximately 5%. This error is inherited by the ECSA. Further calibration of the model is required to achieve better accuracy for multi-threaded programs with communications.

Data effect Since a non data sensitive ECSA will provide a single energy estimation regardless of input data values, comparing this to hardware measurements may give a different error for each input data. To examine this, we used one of our most data sensitive benchmarks, `MatMul 4`. The smallest over-estimation when compared with the hardware measurements was 5.96% for both the ISA ECSA and trace based energy estimations. This was obtained for matrices that were initialized with randomly generated data. Since our energy model was characterized using pseudo-randomly generated data, it provides a good fit to the data used for measurement, thus this result meets our expectations. The maximum over-estimation found was 25%, by initiating both matrices with zero data, minimizing the processor's switching activity. By using the same random data in the two matrices, the over-estimation was between the two previous cases, approximately 15%. This is because the processor switching activity is less than in the case of different random data initialized matrices, and more than the case with the zero initialized matrices. Thus, users must be cautious when using ECSA with data sensitive benchmarks, as we will discuss in Section 4.3.

These findings lead to two new research questions. Firstly, for convenience, many energy models are constructed from random input data. However, as we demonstrated, the closer the data used to characterize the energy model fits the data of the use case, the more accurate the ECSA estimation. For example, `MatMul` and `fdct` are heavily used in video processing applications with highly correlated data between frames in the video stream. Therefore, a random data constructed energy model for these applications may not be suitable. How can we construct energy models that are more fit for purpose? Secondly, if a data sensitive energy model were to be constructed, how would this model be composed to be useful for ECSA? These two research questions motivate future work in this area.

Static analysis limitations ECSA suffers from all the static analysis limitations that the timing analysis faces [64]. Many of the techniques used by the timing analysis community to tackle these limitations can also be adopted in ECSA. For example, infeasible paths can lead to unrealistic estimations in both cases, energy and time. Techniques such as symbolic execution [46] used in timing analysis to exclude infeasible paths, can be also used for ECSA. For this paper, source code annotations were translated to ILP constraints, in order to exclude infeasible paths from ECSA.

As already identified, ECSA can be more complicated than timing analysis. In Figure 5, we discussed that energy consumption is sensitive to the data related switching activity in the processor, which time is not affected from. In Section 3.5 we discussed, that for multi-threaded programs, timing analysis is considered only with a single path across all the threads, but ECSA has to consider all computations active during the case of interest.

In summary, the results show that static analysis, both at ISA and LLVM-IR level, can deliver practical energy consumption estimates for a good range of single and multi-threaded programs. The estimation error for both static and simulation based techniques can be reduced if the accuracy of the underlying energy model is improved.

4.3 ECSA applications

Precise energy measurements are often not easily accessible, requiring extra equipment and hardware knowledge as well as modifications to the target hardware. This makes it very difficult for most programmers to assess a program's energy consumption. ECSA overcomes these obstacles by providing energy transparency to users and systems with a useful level of accuracy.

Trace based energy estimation allows for a very precise estimation of energy consumption for a particular program run. The program is executed in simulation with a given set of input parameters. The exact sequence of instructions can be recorded during simulation and then used to estimate energy consumption. However, a change to the input may produce a new execution path, requiring a new simulation run to extract the correct instruction sequence. Simulation is typically several orders of magnitude slower than hardware execution, making repeated simulations undesirable as a means for tuning or optimizing a program. ECSA does not depend on repeated simulation. It does not require trace data in order to provide an energy estimation. This allows for much faster estimation of a program's energy consumption.

The main difference between energy measurements, trace simulation based energy predictions and ECSA, is that the first two methods estimate the cost of the actual executed path. ECSA, however, gives an upper bound based on the cost model used. Both ECSA and trace estimations rely on the accuracy of the energy model. Further, they cannot accurately account for energy due to data-sensitive switching activity. In the rest of this section we will provide a set of guidelines on how the ECSA results should be interpreted, and how they can influence energy aware decisions that can be made by software developers, compiler engineers, development tools and RTOS.

LLVM IR level ECSA The LLVM optimizer and code emitter are the natural place for compiler optimizations. Our LLVM IR analysis results demonstrate a high accuracy with a deviation in the range of 1% from the ISA ECSA. Some LLVM IR estimations may not always be as accurate as at ISA level, but they are still of value to developers. Transparency of energy consumption

at this level enables programmers to investigate how optimizations affect their program's energy consumption [65], or even help introduce new low energy optimizations [66, 67]. This is more applicable at the LLVM IR than at the ISA level, because more program information exists at that level, such as types and loop structures. Our mapping techniques and analysis framework at the LLVM IR level are applicable to any compiler that uses the LLVM common optimizer, provided that an energy model for the target architecture is available.

For some programs, indirect jumps that are introduced at the ISA level can make it impossible to extract a CFG. While this prevents ISA level ECSA, the analysis can still be performed for these programs at LLVM IR, allowing programmers to gain energy consumption insight even when ISA level analysis is not feasible.

ECSA bound use cases Given that we are using bound analysis with an energy model characterized with random input data, we must consider the ECSA estimations as loose upper bounds of the WCEC. Although, these bounds are not safe, in most cases they can provide useful information to the programmer, e.g. to determine whether or not an application is likely to exceed an available energy budget.

The modified `B.Radix4Div` benchmark avoids an early return when the dividend is greater than 255. Omitting this optimization is less efficient, but balances the CFG paths. The effect of this modification can be seen in Figure 5. The ISA level energy consumption lower bounds (the best case retrieved by IPET) are shown. In the optimized version, the energy consumption across different test cases varies significantly, creating a large range between the upper and lower energy consumption bounds. Conversely, the unoptimized version shows a lower variation, thus narrowing the margin between the upper and lower bounds, but has a higher average energy consumption.

Knowledge of such energy consumption behavior can be of value for applications like cryptography, where the power profile of systems can be monitored to reveal sensitive information in side channel attacks [68]. In these situations, ECSA analysis can help code developers to design code with low energy consumption variation, so that any potential leak of information that could be obtained from power monitoring can be obfuscated.

Parametric resource usage equations Regression analysis was applied to the ISA level static analysis results of the benchmarks `MatMult 1,2,4`, `Mac`, `Cnt` and `Base64`. The resultant upper bound equations are shown in Table 1. The second column shows the retrieved equations which return the energy consumption predictions in nano-Joules (nJ) as a function over x, as defined in the third column. `Levenshtein` is a multi-parametric energy consumption benchmark. However, the regression analysis was unable to determine a good parametric equation for it.

Parametric resource usage equations can be valuable for a programmer or user to predict energy consumption with specific parameter values. Moreover,

embedding such equations into an operating system can enable energy aware decisions for either scheduling tasks, or checking if the remaining energy budget is adequate to complete a task. If the application permits, the operating system may also downgrade the quality of service to complete the task within a lower energy budget.

Benchmark	Regression Analysis (nJ)	x
Base64	$f(x) = 19x + 94.2$	string length
Mac	$f(x) = 15x + 21.1$	length of two vectors
Cnt	$f(x) = 19.9x^2 + 5.7x + 34.6$	matrix size
MatMul	$f(x) = 12.2x^3 + 17.5x^2 + 4.7x + 33$	size of square matrices
MatMul_2T	$f(x) = 19.3x^3 + 21.4x^2 + 5.9x + 96.8$	size of square matrices
MatMul_4T	$f(x) = 22.7x^3 + 25x^2 + 6.5x + 157.7$	size of square matrices

Table 1: Benchmarks with parametric energy consumption.

Multi-threaded ECSA The first class of parallel programs to which ECSA was applied is replicated non-communicating threads. The user can make energy aware decisions on the number of threads to use, with respect to time and energy estimations retrieved by our analysis. For example, take four independent matrix multiplications on four pairs of equally sized matrices (28×28). Our analysis will show that a single thread will have an execution time of 4x the time needed to execute one matrix multiplication. However, two threads will half the execution time and decrease the energy by 54%. Four threads which will half the execution time again, and decrease the energy by 41% compared to the two-thread version. Using more threads increases the power dissipation, but the reduction in execution time saves energy on the platform under investigation. Although there is a different estimation error between different numbers of active threads, the error range of 6% is small enough to allow comparison between these different versions. The comparison can be also done by RTOS using the cost functions from Table 1 to make real time energy aware scheduling decisions.

The second class of parallel programs that our ECSA was applied to was streaming pipelines of communicating threads. There is a choice in how to spread the computation across threads to maximize throughput and therefore minimize execution time or lower the necessary device operating frequency. Having a number of available threads, a number of cores and the ability to apply voltage and frequency scaling, provides a wide range of configuration options in the design phase, with multiple optimization targets. This can range from optimizing for quality of service, time and energy, or a combination of all three. Our ECSA can take advantage of the fact that the energy model used can be parametric to voltage and frequency, to statically identify the most energy efficient configuration of the same program, among a number of different options that deliver the same required performance. The first step of analyzing the pipelined versions of industrial filter applications has been made in this paper. We are currently

working on extending our ECSA to automatically exploit the possible different configurations and provide the optimal solution, within the user's constraints.

Finally, the user needs to be aware of the potential effect of input data. When highly data sensitive applications are analyzed, the user can make some assumptions, based on the possible input data range, about the accuracy of the ECSA analysis. As explained in Figure 5, data that is close to random will lead to a smaller estimation error, when random data was used to build the energy model. From our findings, this variation can be up to 25%, but this has only been shown in short, contrived cases and is unlikely to be large in realistic programs.

5 Conclusion and Future Work

This work has given critical review of ECSA existing works that have overlooked the effect of using non data sensitive energy models and SRA bound techniques, on the retrieved energy estimations. In the absence of average case SRA and data sensitive energy models, we establish this effect in our experimental evaluation of ECSA on a set of mainly industrial benchmarks. We also demonstrate that such an analysis can still have a significant value for software developers, compiler engineers, development tools and RTOS, by establishing a number of ECSA applications in Section 4.3.

A technique was introduced to allow energy characterization of LLVM IR. It enables ECSA at this level with a small loss of accuracy, typically 1%, compared to ECSA at ISA level. ECSA is applied to a set of multi-threaded programs for the first time to our knowledge. This is a significant step beyond existing work that examines single-thread programs, because such an analysis can provide significant guidance for time-energy design space exploration between different numbers of threads and cores.

This work has generated new research questions. There is a clear need for non bounding SRA techniques that focus on average cases. Data sensitive energy models and SRA techniques are needed for ECSA to account for data sensitive switching activity in the processor. The majority of existing energy models are usually generated using random data. As we have discussed in Figure 5, alternative data energy models might be better for specific applications.

Future work aims to analyze more complex concurrent programs, such as distinct non-communicating threads rather than replicated threads, pipelines of threads with unbalanced workloads and other concurrency patterns. The ECSA can be combined with some more dynamic techniques such as abstract simulation to account for all the possible threads interleaving. Extending such analysis beyond deeply embedded systems, with more architectural performance enhancing features, might be done by exploiting more techniques from the WCET community, such as abstract interpretation and data cache analysis.

References

1. R. Jayaseelan, T. Mitra, and X. Li, "Estimating the worst-case energy consumption of embedded software," in *Real-Time and Embedded Technology and Applications*

Symposium, 2006. Proceedings of the 12th IEEE, pp. 81–90, April 2006.
2. V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee, "Instruction level power analysis and optimization of software," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 13, no. 2-3, pp. 223–238, 1996.
3. J. Pallister, S. Kerrison, J. Morse, and K. Eder, "Data dependent energy modelling: A worst case perspective," *CoRR*, vol. abs/1505.03374, 2015.
4. D. May, "XMOS XS1 Instruction Set Architecture," 2009.
5. S. Kerrison and K. Eder, "Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor," *ACM Transactions on Embedded Computing Systems*, vol. 14, pp. 56:1–56:25, Apr. 2015.
6. C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, pp. 75–88, 2004.
7. LLVMorg, "The LLVM Compiler Infrastructure," November 2014.
8. A. Bogliolo, L. Benini, G. Micheli, and B. Ricc, "Gate-Level Power and Current Simulation of CMOS Integrated Circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, no. 4, pp. 473–488, 1997.
9. K. Roy and M. Johnson, "Software design for low power," in *Low power design in deep submicron electronics*, ch. 6, pp. 433–460, Kluwer Academic Publishers, 1997.
10. D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 83–94, May 2000.
11. Sim-Panalyser, *Sim-Panalyser 2.0 Reference Manual*. 2004.
12. T. Austin, "SimpleScalar: An Infrastructure for computer system modeling," *IEEE Computer*, no. February, pp. 59–67, 2002.
13. S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel, "An accurate and fine grain instruction-level energy model supporting software optimizations," in *Proc. of PATMOS*, Citeseer, 2001.
14. D. Sarta, D. Trifone, and G. Ascia, "A data dependent approach to instruction level power estimation," in *Low-Power Design, 1999. Proceedings. IEEE Alessandro Volta Memorial Workshop on*, pp. 182–190, Mar 1999.
15. H. Kojima, D. Gorny, K. Nitta, and K. Sasaki, "Power analysis of a programmable dsp for architecture/program optimization," in *Low Power Electronics, 1995., IEEE Symposium on*, pp. 26–27, Oct 1995.
16. M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria, "An instruction-level energy model for embedded VLIW architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, pp. 998–1010, Sept. 2002.
17. M. a. Ibrahim, M. Rupp, and H. Fahmy, "Power estimation methodology for VLIW Digital Signal Processors," in *2008 42nd Asilomar Conference on Signals, Systems and Computers*, no. 1, pp. 1840–1844, IEEE, Oct. 2008.
18. G. Qu, N. Kawabe, K. Usami, and M. Potkonjak, "Function-level power estimation methodology for microprocessors," *Proceedings of the 37th conference on Design automation - DAC '00*, pp. 810–813, 2000.
19. S. Lee, A. Ermedahl, and S. Min, "An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors," *ACM SIGPLAN Notices*, vol. 36, pp. 1–10, Aug. 2001.
20. G. Contreras and M. Martonosi, "Power prediction for Intel XScale processors using performance monitoring unit events," in *ISLPED '05. Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, pp. 221–226, IEEE, 2005.

21. Y. Shao and D. Brooks, "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor," in *International Symposium on Low Power Electronics and Design (ISLPED)*, no. November, pp. 389–394, IEEE, Sept. 2013.
22. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.
23. G. Brat, J. Navas, N. Shi, and A. Venet, "Ikos: A framework for static analysis based on abstract interpretation," in *Software Engineering and Formal Methods*, pp. 271–277, Springer, 2014.
24. B. Wegbreit, "Mechanical program analysis," *Commun. ACM*, vol. 18, no. 9, pp. 528–539, 1975.
25. M. Rosendahl, "Automatic complexity analysis," in *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, (New York, NY, USA), pp. 144–156, ACM, 1989.
26. S. K. Debray, N.-W. Lin, and M. Hermenegildo, "Task Granularity Analysis in Logic Programs," in *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pp. 174–188, ACM Press, June 1990.
27. S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin, "Lower Bound Cost Estimation for Logic Programs," in *1997 International Logic Programming Symposium*, pp. 291–305, MIT Press, Cambridge, MA, October 1997.
28. P. Vasconcelos and K. Hammond, "Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs," in *Proceedings of the Workshop on Implementation of Functional Languages*, vol. 3145 of *Lecture Notes in Computer Science*, pp. 86–101, Springer-Verlag, September 2003.
29. J. Navas, E. Mera, P. López-García, and M. Hermenegildo, "User-Definable Resource Bounds Analysis for Logic Programs," in *International Conference on Logic Programming (ICLP'07)*, Lecture Notes in Computer Science, Springer, 2007.
30. E. Albert, P. Arenas, S. Genaim, and G. Puebla, "Closed-Form Upper Bounds in Static Cost Analysis," *Journal of Automated Reasoning*, vol. 46, pp. 161–203, February 2011.
31. J. Hoffmann, K. Aehlig, and M. Hofmann, "Multivariate amortized resource analysis," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 3, p. 14, 2012.
32. D. Alonso-Blas and S. Genaim, "On the limits of the classical approach to cost analysis," vol. 7460, pp. 405–421, 2012.
33. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith, "Bound analysis of imperative programs with the size-change abstraction (extended version)," *CoRR*, vol. abs/1203.5303, 2012.
34. J. Engblom, A. Ermedahl, and F. Stappert, "Comparing different worst-case execution time analysis methods," in *The Work-in-Progress session of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*, November 2000.
35. Y.-T. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, (New York, NY, USA), pp. 456–461, ACM, 1995.
36. H. Theiling and C. Ferdinand, "Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis," in *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pp. 144–153, Dec 1998.
37. G. Ottosson and M. Sjodin, "Worst-case execution time analysis for modern hardware architectures," in *In Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97*, pp. 47–55, 1997.

38. D. Potop-Butucaru and I. Puaut, "Integrated Worst-Case Execution Time Estimation of Multicore Applications," in *13th International Workshop on Worst-Case Execution Time Analysis* (C. Maiza, ed.), vol. 30 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 21–31, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.
39. P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Los Angeles, California), pp. 238–252, ACM Press, New York, NY, 1977.
40. J. Navas, M. Méndez-Lojo, and M. Hermenegildo, "Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications," in *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, April 2008. Extended Abstract.
41. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García, "Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)," *Science of Computer Programming*, vol. 58, no. 1–2, 2005.
42. J. Navas, M. Méndez-Lojo, and M. Hermenegildo, "User-Definable Resource Usage Bounds Analysis for Java Bytecode," in *Proceedings of BYTECODE*, vol. 253 of *Electronic Notes in Theoretical Computer Science*, pp. 65–82, Elsevier - North Holland, March 2009.
43. U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. Hermenegildo, and K. Eder, "Energy Consumption Analysis of Programs based on XMOS ISA-level Models," in *Proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, 2014.
44. D. Watt, *Programming XC on XMOS Devices*. XMOS Limited, 2009.
45. N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder, "Static analysis of energy consumption for llvm ir programs," in *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, (New York, NY, USA), ACM, 2015.
46. P. Wägemann, T. Distler, T. Hönig, H. Janker, R. Kapitza, and W. Schröder-Preikschat, "Worst-case energy consumption analysis for energy-constrained embedded systems," July 2015.
47. K. Georgiou, "On the value and limits of multi-level energy consumption static analysis for deeply embedded single and multi-threaded programs - benchmarks." `https://www.cs.bris.ac.uk/home/kg8280/benchmarks.html`, 2015.
48. J. M. Townley, "Practical programming for static average-case analysis: the moqa investigation," 2013.
49. M. Schellekens, "Moqa; unlocking the potential of compositional static average-case analysis," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 1, pp. 61 – 83, 2010. Speical Issue: Logic, Computability and Topology in Computer Science: A New Perspective for Old Disciplines.
50. F. Najm, "A survey of power estimation techniques in vlsi circuits," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, pp. 446–455, Dec 1994.
51. D. May, *The XMOS XS1 Architecture*. 2009.
52. K. Georgiou, "On the value and limits of multi-level energy consumption static analysis for deeply embedded single and multi-threaded programs - fnop modeling." `https://www.cs.bris.ac.uk/home/kg8280/fnops.html`, 2015.

53. C. Brandolese, S. Corbetta, and W. Fornaciari, "Software energy estimation based on statistical characterization of intermediate compilation code," in *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pp. 333–338, Aug 2011.
54. "The dwarf debugging standard," Oct. 2013. `http://dwarfstd.org/`.
55. C. Lattner and D. Patel, "Extensible metadata in llvm ir," Apr 2014.
56. T. Wei, J. Mao, W. Zou, and Y. Chen, "A new algorithm for identifying loops in decompilation," in *Static Analysis* (H. Nielson and G. Fil, eds.), vol. 4634 of *Lecture Notes in Computer Science*, pp. 170–183, Springer Berlin Heidelberg, 2007.
57. Y.-S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 16, pp. 1477–1487, Dec 1997.
58. K. Eder, K. Georgiou, and N. Grech, eds., *Common Assertion Language*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013. Deliverable 2.1, http://entraproject.eu.
59. XMOS, "xTimecomposer," November 2014.
60. M. Field, "Binary division," November 2014.
61. J. Hauser, "SoftFloat," November 2014.
62. XMOS, "Application Note: DSP performance on XS1-L device," November 2014.
63. J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks - past, present and future," in *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.
64. R. Wilhelm and D. Grund, "Computation takes time, but how much?," *Commun. ACM*, vol. 57, pp. 94–103, Feb. 2014.
65. C. Blackmore, O. Ray, and K. Eder, "A logic programming approach to predict effective compiler settings for embedded software," *Theory and Practice of Logic Programming*, vol. 15, pp. 481–494, 7 2015.
66. J. Pallister, K. Eder, and S. Hollis, "Optimizing the flash-ram energy trade-off in deeply embedded systems," in *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, pp. 115–124, Feb 2015.
67. J. Pallister, K. Eder, S. J. Hollis, and J. Bennett, "A high-level model of embedded flash energy consumption," in *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, (New York, NY, USA), pp. 20:1–20:9, ACM, 2014.
68. P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, (London, UK, UK), pp. 388–397, Springer-Verlag, 1999.

Acknowledgments

The research leading to these results has received funding from the European Union 7th Framework Programme (FP7/2007-2013) under grant agreement no 318337, ENTRA - Whole-Systems Energy Transparency. Special thanks to Intel for providing us with the equipment used for our power monitoring setup.

Attachment D2.3.4

On the infeasibility of analysing worst-case dynamic energy

Under review at the ACM journal *Transactions on Embedded Computing (TECS)*.

On the infeasibility of analysing worst-case dynamic energy

Jeremy Morse, Steve Kerrison and Kerstin Eder
University of Bristol

March 7, 2016

Abstract

In this paper we study the sources of dynamic energy during the execution of software on microprocessors suited for the Internet of Things (IoT) domain. Estimating the energy consumed by executing software is typically achieved by determining the most costly path through the program according to some energy model of the processor. Few models, however, adequately tackle the matter of dynamic energy caused by operand data. We find that the contribution of operand data to overall power can be significant, prove that finding the worst-case input data is NP-hard, and further, that it cannot be estimated to any useful factor. Our work shows that accurate worst-case analysis of data dependent energy is infeasible, and that other techniques for energy estimation should be considered.

1 Introduction

A significant design constraint in the development of embedded systems is that of resource consumption. Software executing on such systems typically has very limited memory and computing power available, and yet must meet the requirements of the system. To aid the design process, analysis tools such as profilers or maximum-stack-depth estimators provide the developer with information allowing them to refine their designs and satisfy constraints.

A less well studied constraint is the limited energy budgets that deeply embedded systems possess. A typical example would be a wireless sensor powered by battery, that must operate for a minimum period without the battery being replaced. Other examples would be systems dependent on energy harvesting, or systems with low thermal design points that thus have a maximum power dissipation level. These constraints can also be approached with software analysis tools, and several techniques have been developed that allow the estimation of software's energy consumption [17, 7, 18].

Within energy estimation, focus has been given to *Worst Case* Energy Consumption (WCEC): determining the maximum amount of energy that can be consumed during the execution of the software. In this paper, we shall study the calculation of worst case energy, considering only the effects that different software and inputs can have on a system. The objective is to determine whether it is possible to establish an upper bound on energy that is tighter than over-estimating by, for example, using a maximum activity factor. Such a factor may be unachievable during the execution of a real program, because data that triggers the highest energy consumption in one instruction may, through data dependency and other constraints, preclude subsequent instructions from consuming their maximal energy [11].

Energy is the integral of power over a given period. The power dissipation of a processor can be apportioned in two parts: static and dynamic. *Static power* or leakage is the power dissipated for as long as the component is turned on, irrespective of its internal state or any changing inputs and outputs. *Dynamic power* or switching activity refers to power dissipation from the substance of execution: the switching of gates and charging of data buses, which all consume energy. We express these more formally in Section 3. Analysis of worst-case instantaneous dynamic power has been well studied in the literature, but here we consider worst-case energy, i.e. the power over a program execution.

Estimating worst case energy for a particular program naturally becomes the computation of these two distinct sources of energy consumption. Static power is directly controlled by the length of the program, measured in time. Numerous techniques have been developed by the *worst case execution time* (WCET) community to address this matter [33]. Dynamic power, however, has received much less attention. Several models of how systems consume energy have characterised the dynamic power only for specific inputs, averaged over all inputs, assumed a worst-case dynamic power for each instruction with few details, or assumed no dynamic power at all [12, 7, 32].

This paper demonstrates that for data-dependent dynamic energy, the calculation of the worst case input to a software execution is an NP-hard problem, and further, that its dynamic energy cannot be approximated to a useful factor. Our proof applies to processors in general, but we show on an example processor, the Xcore XS1-L [19], that the portion of energy that is infeasible to analyse contributes at least half of the processor's dynamic power.

The rest of the paper is structured as follows: in Section 2 we examine the current state of energy estimation, and related work. In Section 3 we study the Xcore processor and its dynamic energy consumption. Section 4 formalises the problem that we are dealing with, which is shown to be NP-hard in Section 5, and in Section 6 we demonstrate that the problem cannot be effectively approximated. We discuss the results in Section 7, and draw conclusions in Section 8 with an outlook on future work.

2 Background

This section identifies existing techniques for determining the energy consumption of software when executed, techniques for determining the maximum amount of energy a program can consume, and the theoretical definition of the MAXSAT problem.

2.1 Energy estimation techniques

Given the high complexity of microprocessors, energy analysis based on hardware designs tends to be resource intensive, and require access to proprietary design materials. Research has instead focused on using empirical techniques to model how processors consume energy. These models can then be used to estimate the consumption of a real-world system.

One of the most popular techniques is the instruction level energy model [30]. Various test patterns of instructions are executed on a processor and their power empirically measured, leading to a model of per instruction energy costs and the dynamic cost of switching between different instructions. Simulating an instruction sequence, or interpreting a trace of an execution, can then be combined with this energy model to produce a cost value for the execution. Steinke [27] extend this model to include the costs of circuit switching in instruction operands. These costs include the amount of switching occurring on data buses supplying input operands when executing instructions, and the switching on the output datapath when an operand or memory address is written to by an instruction.

Further modelling techniques for dynamic power go beyond the core part of the processor, such as analysing flash memory [21], caches [4] and DRAMs [16]. High performance processors feature hardware-provided counters that record metrics such as cache hit rates, which can be used by appropriately parametrised energy models [24]. In this paper, where embedded devices are the focus, we choose to only examine the dynamic power attributable to the core part of the processor.

2.2 Worst case energy consumption (WCEC)

WCEC is a form of energy estimate, where the aim is to find the maximum amount of energy that a piece of software will consume. The problem is thus made of two parts: modelling the energy consumption of the software under test, and searching for the execution of it that will lead to the greatest amount of energy consumed. This problem is similar to the worst case execution time problem (WCET) [33] where the execution time of software is modelled, and then the longest possible path found. For both problems, a *specific* worst case execution is sought. However, much interest is also shown in providing an upper bound on the worst case. Such a bound may be higher than the worst case, but may help demonstrate that a design constraint is met.

2.2.1 WCEC is not a simple reinterpretation of WCET

The worst case *energy* consumption problem goes beyond the worst case execution time problem, because the execution time of a single instruction is largely independent of its input data. This is because timing variability has mostly been eliminated "by design" through the use of synchronous logic and the limited propagation time associated with executing individual instructions.

In real-time embedded systems, timing-predictable processors execute instructions within a fixed number of clock cycles, irrespective of the data the operation works on. This is particularly beneficial to WCET analysis, which can then focus on identifying the worst case execution path which is determined by the control flow, rather than by the data flow of the computation. More advanced micro-architectural features, such as early-out of operations, or cache hierarchies, provide higher average performance at the cost of predictability. This makes WCET analysis far more challenging, as tight bounds firmly rely on timing predictability of the target architecture [28]. However, even operations that have a variable execution time can be quantized by the processor's clock period into a tractable number of discrete possibilities. The range may be in the order of tens, hundreds, or thousands of cycles, depending on the type of operation.

Energy depends on both, the execution time and the power dissipation of the operation. Power is not quantized in terms of the clock period, but instead by the number of transistor and interconnect state changes (i.e. switches) that *may* take place during an operation, depending on the data to be processed. The number of possible power dissipation levels is thus in the order of the number of transistors in the device. This is several orders of magnitude larger than the number of timing possibilities explored by WCET analysis.

For the techniques that are used in WCET to be directly transferable to WCEC, a set amount of energy per operation would need to be specified and realised in hardware, similar to specifying and ensuring, through timing analysis, that each operation fits into a fixed number of clock cycles. Consider the converse: A processor that presents a similar WCET analysis difficulty to determining WCEC, would be an asynchronous design, where the precise execution time is a non-trivial function of an operation's input data. Such devices may have an average delay, but actual performance for a given use case or tight bounds may be harder to determine [14].

2.2.2 Existing work on WCEC

The first publication to provide a technique for computing the WCEC of software was [12], where upper bounds on the energy consumption of several programs were inferred using energy models of software basic blocks and an ILP solver to find a maximal path through the program. The authors additionally debunk the suggestion that the execution path consuming the most time is always the path that also consumes the most energy. With regards to dynamic power, the authors assume that all circuits switch on every clock cycle rather than attempting to determine actual switching activity, their justification being that the contribution of dynamic power to overall energy is low, thus their approximation does not introduce significant imprecision. We address this in Section 3.

Resource analysis techniques that extract cost relations from programs have been employed to analyse energy consumption bounds [17, 7]. The costs used in these analyses represent energy consumption and are based on models that provide a single energy cost per instruction, obtained by averaging the energy measured from processing random data, constrained to yield valid operands for the respective instruction [15]. However, bounds obtained in this way cannot be considered safe, as executions would exist where the energy from operand data exceeds the average case.

More recently, [32] have presented techniques for estimating over and under approximations of WCEC through implicit path enumeration and genetic algorithms, respectively. They do not, however, comment on dynamic power at all: their absolute instruction energy model appears to assume maximum switching for each instruction cost. Their relative energy model does not consider real energy costs, instead estimating the difference in energy consumption between instructions, again with no explicit consideration of dynamic power.

Both Jayaseelan and Wägemann identify inefficiency as being a reason why they cannot compute accurate switching activities for circuits. As we will show in this paper, the problem is infeasibly complex under the P $\neq$ NP assumption.

2.3 Existing complexity results

Switching activity is a matter studied in detail by the VLSI community for circuit design, as the maximum instantaneous switching in a circuit can affect the power supply requirements [20]. This problem has been shown to be NP-hard [5] and numerous techniques have been developed to make an estimate of the worst case power consumption [8], allowing maximum power analysis.

Power estimation itself does not directly correspond with energy estimation. The objective of WCEC is finding the maximal amount of circuit activity over a period of time, rather than the instantaneous maximum, which itself may be incompatible with the circumstances that lead to maximum energy. In particular, software requires that computations be consistent with past inputs, creating additional constraints and dependencies.

Switching between instructions is a notable source of energy consumption, which can be controlled through the order in which instructions are executed. Techniques have been developed to reduce consumption through instruction scheduling [23], but this is known to be an NP-hard problem. Instruction scheduling uses pre-computed costs of switching between instructions to determine an optimal static schedule. It does not consider the data operands to instructions or any cost that does not have a fixed value.

None of these complexity results are directly applicable to the estimation of energy in data-dependent switching during software execution. To the best of our knowledge, we believe this is the first work to consider data-dependent switching costs.

2.4 Maximum satisfiability

The *Maximum satisfiability* problem "MAXSAT" [2, pp.613–631] is defined as the satisfiability problem where the number of clauses satisfied must be maximised by an assignment. Following the presentation of [13], define L to be a set of literals, and C a set of disjunctive form clauses:

$$L = \bigcup_{i>0} \{x_i, \overline{x}_i\}$$
$$c \in C, C = \{l_1 \vee ... \vee l_n \mid l_i \in L\},$$

where each x_i is a Boolean variable. A truth assignment defines each x_i or its negation to be true. A clause is deemed to be satisfied if at least one literal in the clause is assigned true. A MAXSAT problem is a set of literals and set of clauses $\langle L, C \rangle$, such that the solution is the truth assignment that causes the maximal number of clauses to be satisfied.

3 Circuit switching on Xcore

Prior WCEC papers have relied on the suggestion that the variation in dynamic switching is small in relation to other energy costs in a processor, at approximately 3 % [29]. Other work has presented a mixed picture: [26] found that the switched capacitance (i.e. switching cost) of a StrongARM processor had little variance across applications, suggesting that switching costs contribute little to overall program energy; while [1] observe that data switching accounts for up to 50% of processor core energy.

Here, we affirm that dynamic switching costs can be high by analysing the energy consumption of the Xcore [19] XS1-L, and demonstrating a significant energy variation due to dynamic switching.

3.1 Defining power dissipation in a micro-processor

The energy, E, of an electronic device is the integral of its power dissipation, P, over a given time period, T:

$$E = \int_{t=0}^{T} P(t)\ dt. \tag{1}$$

Power is an instantaneous measure of the rate of work. Typically, this is sampled repeatedly in order to discretise the integral, or the power is averaged, simplifying the equation to $E = P \times T$.

In digital devices such as processors, the total power dissipation of the device, P_{tot} is typically apportioned into two additive parts, termed static and dynamic, denoted here as P_s and P_d respectively:

$$P_{tot} = P_s + P_d \tag{2}$$

Elaborating on these, static power is determined by the operating voltage, V_{dd} of the device and I_{leak}, the leakage current present, which is itself dependent upon physical characteristics such as operating temperature, transistor feature size and the manufacturing process that is used.

$$P_s = V_{dd} I_{leak}, \quad \therefore \quad P_s \propto V_{dd} \tag{3}$$

Dynamic power is dependent upon the capacitance of the components that are being switched, C_{sw}, as well as the operating voltage and the frequency of switching, f. In a processor, f is

governed by the clock frequency. The proportion of the device that is switching is dependent upon the instruction and data being executed and related changes in state. This is represented by an *activity factor*, α, where each instruction or action performed by the processor may have a different α.

$$P_d = \alpha C_{sw} V_{dd}^2 f, \quad \therefore \quad P_d \propto V_{dd}^2 \tag{4}$$

There is a quadratic relationship between voltage and dynamic power. The necessary operating voltage is approximately linearly proportional to the operating frequency.

3.2 Apportioning dynamic power

When considering power per instruction, it is important to calculate an appropriate α per instruction, or some equivalent by abstraction. However, the instruction is not the sole influence upon the α value. The operands supplied to the processor's functional units (for example, arithmetic unit), will affect the amount of switching. This includes changes to the input and output data buses, as well as internal switching within the functional unit as the new result is computed. As such, one instruction may have a range of possible α values that are dependent on the input data.

Prior work [12] has suggested that this variation in α is small and therefore not significant enough to consider when constructing a worst-case energy model. However, we demonstrate that variation in input data can be responsible for as much as 42 % of a core's power dissipation and thus becomes a relevant contributor to the model. This is pertinent to systems with minimal additional components, such as those that are deeply embedded, where the processor is the major consumer of energy. In larger, more complex systems, with many external devices and power supplies, the variation in total system energy due to data values is proportionally smaller.

Internal processor data buses are one of the largest contributors of dynamic power. These buses interconnect various internal units, and so changing values on these buses indicate the charge or discharge of connections between a number of gate inputs and outputs, which may have different loads depending on their fan-in or fan-out and connection length. The [27] energy model explores this and discovers that approximately 20 % of overall processor power can be attributed to the Hamming distance on buses.

To determine the dynamic power cost on our target device (the Xcore XS1-L), we performed experiments in the manner of [27]. For a set of instructions, we tested every combination of operand inputs from zero to 255 for each operand, creating a sequence of tests, $\mathbb{P}$. We alternate between instructions with this data set and all-zero operands, to ensure we measured the Hamming distance on each cycle. The Xcore is a cache-less multi-threaded processor with time-deterministic execution. Test sequences were constructed in such a way to ensure we exercised the processor datapaths in every instruction cycle. Although the processor has a 32-bit datapath, exhaustive testing over 8-bit data is sufficient to expose the behaviours of interest to this work.

The device is operated with a 1.0 V core power supply and 500 MHz clock frequency. Power is sampled at the 3.3 V input to the DC-DC converter that supplies the cores and is done so using a vendor-supplied sampling and debug device that uses a shunt resistor to determine current. The tests are each run repeatedly for a 0.5 s duration in order to acquire several thousand power samples, then taking the average.

The device under test is a dual-core component, tested with single-core code. As such, we must remove the additional energy consumption that would not be present if a single-core version of the component were to be used. This is established through the following steps. First, measure the power of the dual-core device when idle, defining P_{tdual} to ascertain $P_{tsingle}$:

$$P_{tsingle} = \frac{P_{tdual}}{2}. \tag{5}$$

Executing instruction power tests on one core, leaving the remaining core idle, produces a sequence of test results, $\mathbb{P} = \{P_0, \ldots, P_n\}$. Define the dynamic power contribution of the lowest and highest power test cases as P_{dmin} and P_{dmax} respectively, and the dynamic power range, P_{drng}:

$$P_{dmin} = \min(\mathbb{P}) - P_{tdual}, \tag{6}$$
$$P_{dmax} = \max(\mathbb{P}) - P_{tdual}, \tag{7}$$
$$P_{drng} = P_{dmax} - P_{dmin}. \tag{8}$$

We observe for the device under test that $P_{tdual} = 328\,\text{mW}$ and therefore $P_{tsingle} = 164\,\text{mW}$. Any additional power observed during tests is used to determine how much dynamic power variation is possible for the set of input values tested. This is not solely static power, because even at idle, switching in components such as the clock tree is taking place, contributing to dynamic power. Thus, the difference in power observed during instruction and data tests is not the total dynamic power contribution, but does establish the degree of variation in dynamic power that can take place, and what proportion of total core power this amounts to.

For the `add` instruction, $P_{dmin} = 34\,\text{mW}$ and $P_{dmax} = 96\,\text{mW}$, giving $P_{drng} = 62\,\text{mW}$. This demonstrates that for `add`, up to 27 % of the core's power dissipation is governed by operand values. In a system where processor power is significant, this is a substantial variation, inaccurate predictions of which may be undesirable.

To aid analysis of the results of these experiments, we present a series of "heat-map" figures, showing measured dynamic power in colour and datapath Hamming weights in greyscale. These plots use measurements from tests of the `add` instruction.

Figure 1 shows total dynamic power for `add` with all combinations of two 8-bit operands. The diagonal striping indicates a strong correlation with the number of bits set to `1` in the result of the computation. This is observable due to alternating between test `add` operations and all-zero operations. The Hamming weight of the output is shown in Figure 2. This is determined to represent 4.4 mW per output bit set.

Subtracting the calculated switching power per output bit from the original dynamic power measurements gives Figure 3. This reveals a second pattern that was previously obscured by the dominant effects of the output Hamming weight. Intuitively, this corresponds to the Hamming weight of both input operands, demonstrated in Figure 4. We determine this to be 1.3 mW per input bit set. Repeating this process and subtracting the calculated power per input bit gives Figure 5, which closely corresponds to the Hamming weight shown in Figure 2 as previously stated.

Finally, by subtracting both of the input and output bit dynamic powers produces Figure 6, which shows that the remaining variation in dynamic power is an order of magnitude lower than the effect of these Hamming weights, ranging from 12 mW to 0 mW. Expressed as a series of matrix operations, where P is the measured dynamic power and the input and output Hamming weights are presented as H_i and H_o respectively, the remaining unaccounted for dynamic power, D of Figure 6, is:

$$D = P - (H_i \cdot 1.3) - (H_o \cdot 4.4) \quad \text{mW}. \tag{9}$$

In a real-world program, switching between an operation and all zeroes would not take place. However, with each new instruction, a Hamming distance would be present between the previous and current input values, as well as the previous and current output value. As such, the properties described here naturally translate from Hamming weights into Hamming distances.

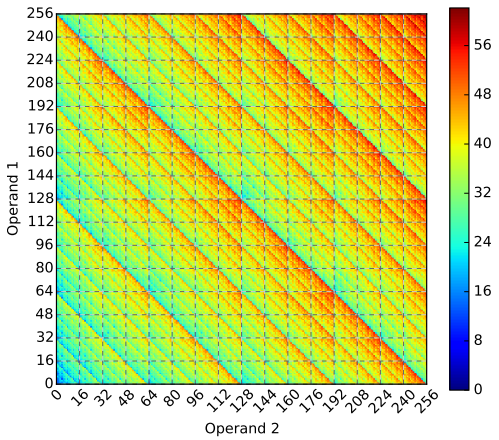


Figure 1: Dynamic power in milliwatts for add instruction.

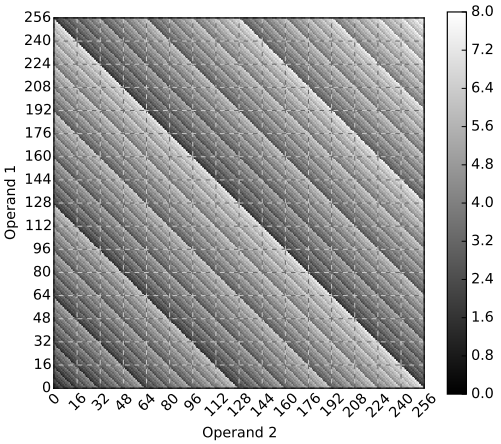


Figure 2: Hamming weight of output datapath of an add instruction, in number of bits set.

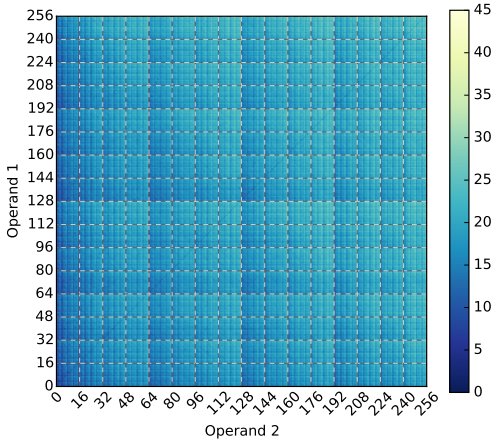


Figure 3: Dynamic power in milliwatts for add instruction, with output datapath cost subtracted (assuming 4.4mW per bit).

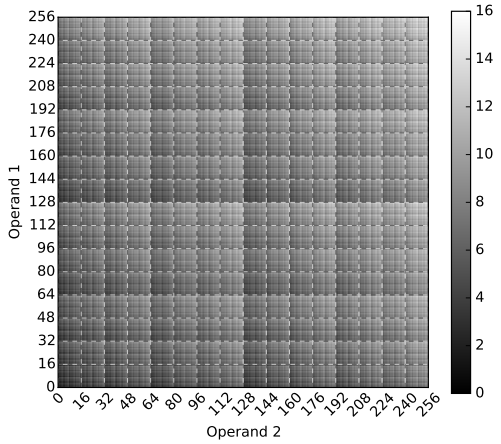


Figure 4: Hamming weight of both input operands to an add instruction, in number of bits set.

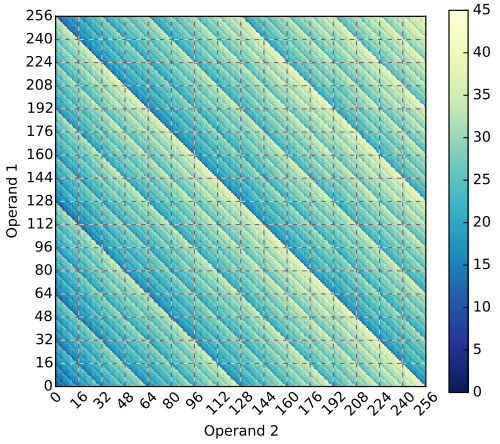


Figure 5: Dynamic power in milliwatts for add instruction, with input datapath cost subtracted (assuming 1.3mW per bit).

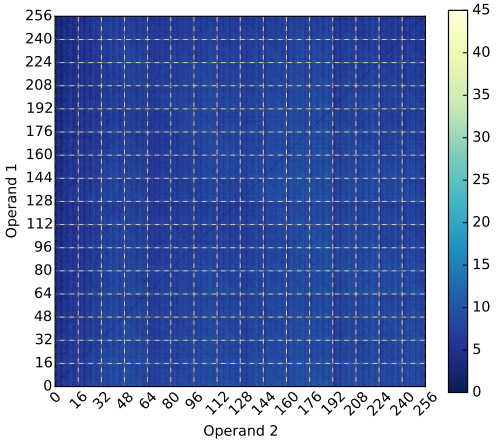


Figure 6: Dynamic power in milliwatts for add instruction, with output and input datapath costs subtracted.

Across all of our experiments, the maximum dynamic power observed was 123 mW, caused by the `sub` instruction. This is due to `sub` producing a negative two's complement output that results in all bits being set in the output operand, causing maximal Hamming distance in the output datapath. However, compared to the base instruction cost this means that on the Xcore the dynamic switching contributes as much as 42 % of the total processor power. Similar work for 8 bit AVR [22] shows dynamic power making up 15 % of processor power.

This data demonstrates that, at least on the Xcore and the AVR, the contribution of dynamic power to the full processor cost is non-trivial, and certainly a significant contributor to calculating the worst case energy in a program. We also observe that the output datapath for our particular processor is the most significant contributor to dynamic power. Accordingly, and for simplicity, we focus only on this component in subsequent sections: specifically, the Hamming distance between values on the output datapath between subsequent cycles.

3.3 Summary and Discussion

With regard to prior work that analyses the significance of dynamic power in software execution, we have demonstrated that on the Xcore dynamic power can be a large proportion of overall energy consumption by the processor, but cannot discount prior work that found little contribution on other platforms. This suggests that dynamic power contribution *can* be significant, but that it varies from processor to processor.

The system context should also be considered, for two main reasons. Firstly, a system that features a display and backlight component will have its total energy consumption dominated by these over all other components [3]. Looking beyond embedded systems, large multi-core processors such as the Xeon Phi [25] consume significantly more energy in caches and memories than in computation. This will of course significantly reduce the impact of any variation in processor energy. Secondly, the type of system and its performance requirements will influence processor choice, and the amount of power variation of the chosen processor will determine whether it is necessary to consider it. If this is the case, the computational workload placed

upon the system will then determine how much each part of the processor is exercised. It is shown in [9] that both processor choice and workload change how processor subcomponents such as the register file and functional units contribute to total energy consumption.

With this in mind, we observe that consideration of dynamic power caused by data operands is most relevant for applications in the Internet of Things (IoT) domain. Such applications typically have energy budgets as a primary concern, have some non-trivial processing task that requires a microcontroller, but do not use a large processor featuring caches and other performance enhancing hardware. When attempting to meet design constraints such as battery lifetime, determining the worst case energy consumption of software would be of interest, and thus determining the impact of data operands on dynamic energy consumption.

4 Formalising the circuit switching problem

As illustrated in the previous section, the matter we consider is the amount of energy caused by circuit switching, specifically the switching occurring on the output datapath in a processor. Here, we formalise our problem, which we name the "Circuit SWitching Problem" (CSWP), discussing its limitations and generality. Our objective is to take a program, determine the maximum amount of output datapath switching activity that can occur in that program, and, in the process, find the program input that triggers it.

Because we are concerned with the amount of circuit switching that can occur, we choose to avoid any facility for varying the length of a program in this formalisation, i.e. the number of instructions executed. A CSWP program thus cannot have any branch instructions or conditional execution ability: it corresponds closely with a trace of a general program execution, or a general program that has been unrolled and all conditional branches eliminated. Dealing with programs of varying length would unnecessarily involve searching different paths through the program.

Formally, we consider a CSWP program, P, to be a finite sequence of n instructions, x_i, such that $P = x_1, x_2, ..., x_n$. Each instruction is a 3-tuple $\langle m, i, o \rangle$, where m is a mnemonic $m \in M$, i is a set of inputs (discussed below), and o is an output operand. Both input and output operands (discussed further below) are considered to be bit-vectors of width w.

A CSWP program executes on an abstract machine with a monotonically incrementing program counter, an infinite number of registers, and a memory store of finite size. Memory is considered to be an array of size 2^w with each memory cell a bit-vector of width w. For each instruction x_i in the CSWP program the machine takes the input operands, computes an output according to the function of the instruction mnemonic, and writes the result to the output operand. The objective function of CSWP is then to compute:

$$\sum_{i=1}^{n-1} h(o_i, o_{i+1}) \ , \tag{10}$$

where h is a function computing Hamming distance between two values, i.e. the output values of each subsequent instruction, corresponding to the output datapath of the abstract machine.

Each mnemonic $m \in M$ represents a function over the input operands, resulting in a single output. In line with the constraints detailed above, CSWP programs only perform arithmetic computations, mapping input operands to an output. There are no branch mnemonics, neither are there any instructions that induce side effects of any form (such as changing some state or the program counter). We do not define a set of mnemonics that a CSWP program may use, however for the purposes of this paper we write listings using standard RISC mnemonics such as `add`, `sub`, `ldr`, `mov` [10].

Each input operand is permitted to be one of four classes of sources:

- Free inputs, which we denote with the text *freebit*.
- Constant values, which we write in hexadecimal.
- A memory access to a fixed address $m[x]$, with x the address.
- The output operand of a prior instruction, written o_i, where for the current instruction x_j, $i < j$.

The value of every input is always a bit-vector of width w. Free inputs may take any value, likewise constants may only have one value, defined in the instruction being executed. In our examples below, we further assume that all free inputs only take the values zero or one. Memory accesses evaluate to the contents of a memory cell, but for simplicity we only permit the addressing of fixed memory addresses. Prior output operands correspond to the output of each instruction being written to one of the infinite registers, which may then be read as an input to another instruction.

All instructions are considered to have an output of bit-width w, i.e., they all write some value to the output datapath of the machine. A `nop` (no-operation) instruction would be any instruction that repeats the output value of the previous instruction, causing no switching activity on the output datapath. Outputs may optionally be written to a memory cell $m[x]$, where x is a fixed address for the output value to be written to. In this circumstance, the output value may still be referred to as o_i, as a store to memory still causes the bits in the machine's output datapath to flip.

This formalisation has a number of limitations, most notably that without an infinite data store or ability to programmatically address it, it is not Turing complete. Given that our aim is to find the maximum switching for a particular path through a general program, this is a suitable restriction. The formalisation does not correspond to a particular machine, although with additional restrictions it may correctly model the execution trace of existing processors. The memory array may be considered to be superfluous given the lack of complex addressing, however it provides a useful mechanism for illustrating our examples through the rest of this paper.

We observe that CSWP is in class NP, as one may easily check the validity of a solution. Given the CSWP program and an input valuation for each free input, we can simulate the program with the given inputs, counting the number of bit flips at the same time process. The complexity of this process scales linearly with the number of instructions, n.

5 Reducing MAXSAT2 to circuit switching problem

To demonstrate that the CSWP is NP-hard, we must reduce any NP-hard problem to CSWP in polynomial time. For this, we turn to the MAXSAT problem, which is known to be NP-hard [2]. Specifically, we work with the MAXSAT2 variant, where each clause is limited to having at most two literals. Despite 2SAT being solvable in polynomial time, MAXSAT2 is still known to be NP-hard [31].

We reduce MAXSAT2 to CSWP by simulating MAXSAT2 in the switching activity of an instruction sequence, where the input that causes the maximum amount of circuit switching corresonds to an assignment to the Boolean variables that causes the maximum number of clauses to be satisfied. The reduction is illustrated in Algorithm 1, which takes the number of Boolean variables and the set of clauses as input, and outputs a CSWP program that simulates MAXSAT2. Here, we assume that the function `PrintInsn` causes a CSWP instruction to be emitted from the algorithm, with the instruction mnemonic, set of variables, and optional output

destination as its respective arguments. The return value identifies the output operand of the instruction.

Algorithm 1: Algorithm for encoding of MAXSAT2 formula within a CSWP program, printed via PrintInsn.

Input: Number of variables n and set of clauses C
Output: CSWP program encoding MAXSAT2 problem
var_addr = 0;
for $i = 0$; $i < n$; $i{+}{+}$ **do**
    out1 = `PrintInsn(`*"mov", [freebit]*`)`;
    out2 = `PrintInsn(`*"xor", [out1, 0x1]*`)`;
    `PrintInsn(`*"store", [out1], m[var_addr++]*`)`;
    `PrintInsn(`*"store", [out2], m[var_addr++]*`)`;
    `PrintInsn(`*"mov", [0]*`)`;
end
foreach $c \in C$ **do**
    $< l1, l2 >= c$;
    laddr1 = `LitToMemAddr(`*l1*`)`;
    laddr2 = `LitToMemAddr(`*l2*`)`;
    lit1 = `PrintInsn(`*"load", [m[laddr1]]*`)`;
    `PrintInsn(`*"xor", [lit1, 0x1]*`)`;
    `PrintInsn(`*"mov", [0]*`)`;
    lit2 = `PrintInsn(`*"load", [m[laddr2]]*`)`;
    `PrintInsn(`*"xor", [lit2, 0x1*`)`;
    `PrintInsn(`*"mov", [0]*`)`;
    `PrintInsn(`*"or", [lit1, lit2]*`)`;
    `PrintInsn(`*"mov", [0]*`)`;
end

First, we read a series of free input values that we assume lie in the range $[0, 1]$, i.e. represent true or false in the lowest bit of the bit-vector. We consider each of these bits to be an assignment to a Boolean variable in the MAXSAT2 problem. Each bit, and its compliment, are stored to a location in memory. This creates an array of values corresponding to the truth of each literal. At the end of this process we insert a `mov` instruction that loads a zero value, for the purpose of resetting the value on the output datapath to zero. The net effect is that for each Boolean variable read, a constant amount of switching activity occurs. Consider each value the free variable may have:

1. `True`: Reading the input switches the lowest output datapath bit to on, the subsequent `xor` switches it to off, and the final `mov` causes no switching.

2. `False`: Reading the input causes no switching, the `xor` switches the lowest output datapath bit to on, and the subsequent `mov` switches it back to off.

Thus, for each Boolean variable read, the CSWP program always causes two bit flips.

We then proceed to use the memory region prepared with literal valuations to simulate the MAXSAT2 problem. We assume a mapping between each literal of the Boolean variables and the address of its valuation in the memory array, and use the function `LitToMemAddr` to translate from literal to memory address. Then, for each clause, we produce an instruction sequence that loads each literal valuation using the constant-switching technique used to read free inputs. Once

the literals are loaded, they are `or`'d together, after which the output datapath is loaded with zero again.

The CSWP program produced by Algorithm 1 has both a constant and data dependent portion of switching activity. Two bit-flips occur for each Boolean variable in the input MAXSAT2 problem, and four for each clause. The switching activity from the `or` instruction, however, directly corresponds to the satisfiability of the clauses: if a clause is satisfiable (i.e., one of the literals is true) then the `or` and following `mov` will cause two additional bit-flips. If a clause is not satisfiable, the same instructions will cause no switching. As a result, the maximum amount of switching in the program is caused by the maximum number of clauses being satisfied. The assignment to the free variables which causes this is also an assignment to the Boolean variables of the MAXSAT2 problem that causes the maximum number of clauses to be satisfied. As a result, CSWP must be at least as hard as MAXSAT2 (i.e. NP-hard). As we know CSWP is also in class NP (Section 4), CWSP is NP-hard. □

We observe that the reduction is performed in polynomial time, as it scales linearly with the number of Boolean variables n and the number of clauses, of which there can be at most n^2.

Given this result, we can conclude that there cannot be an efficient algorithm that solves the CSWP, unless P = NP. Thus, given that general programs can be unrolled and reduced to a CSWP, it is infeasible to determine the worst case datapath switching in a program, defeating energy estimation techniques that would rely on such a model. However, given such a limitation, there could still be algorithms that approximate the worst case switching to a certain degree of accuracy, allowing worst case switching to be narrowed down to a small range of values. We address this in the next section.

6 Inapproximability

Algorithm 2: Algorithm encoding a SAT problem into CSWP, with an output gap governed by satisfiability

```
Input: Number of variables n and set of clauses C
Output: CSWP program with switching gap
/* Decision phase                                                    */
base_var_addr = var_addr = 0;
insn_count = 0;
for i = 0 ; i < n ; i++ do
    out1 = PrintInsn("mov", [freebit]);
    PrintInsn("store", [out1], m[var_addr++]]);
end
result = CheckSat(base_var_addr, C);
bit_pattern = PrintInsn("ite", [result, 0xFFFFFFFF, 0]);
/* Switching phase                                                   */
decision_insn_count = insn_count;
for i = 0 ; i < ((decision_insn_count)/2 + 1) ; i++ do
    PrintInsn("mov", [bit_pattern]);
    PrintInsn("mov", [0]);
end
```

Having shown that CSWP is NP-hard, we will now show that it also cannot be approximated to any useful factor. We demonstrate that there is no constant ε for which an approximation

factor of $1 - \varepsilon$ can be achieved, and then that polynomial approximation factors also cannot be achieved. Intuitively, this is because each bit flip caused by the program is the product of an arbitrary computation, meaning there is no structure to the combinatorial problem that one can generally rely upon when constructing an approximation.

Formally, we demonstrate CSWPs inapproximability using a gap introducing reduction [31] from SAT to CSWP. Such a reduction transforms an NP-complete decision problem into an NP-hard optimisation problem, with a quantity (the "gap") of the feature being optimised governed by the truth of the decision problem. By demonstrating such a gap, one shows that a portion of the NP-hard problem cannot be approximated in polynomial time, as the approximation algorithm would have to solve a NP-complete problem in the process.

In the context of CSWP, we demonstrate that for any instance of SAT, problem p, we can reduce it to a CSWP program q where a portion of the switching activity is governed by the truth of whether p is satisfiable. The transformation is illustrated in Algorithm 2, which we divide into two discrete portions: the decision phase, and the switching phase. We use the same functions as in Algorithm 1, with the modification that the `PrintInsn` function increments a counter, `insn_count`, for every instruction printed.

Throughout the decision phase, we are not concerned with the switching activity that may occur, and do not seek to control it, in contrast with the previous algorithm. We begin by reading n free variables, which we assume to be bit-vectors valued either zero or one, and store them to fixed addresses in memory. We then pass the address of the variable valuations and the SAT clauses to the `CheckSat` function, which emits a CSWP program that evaluates the clauses over the Boolean variables stored at `base_var_addr`, and returns an output operand identifying whether the assignment satisfied the clauses. Significantly, we do not seek to define how `CheckSat` checks the satisfiability of the clauses, we only assume that it achieves it in a number of instructions polynomial in n, the number of Boolean variables. We know that SAT is in NP, so due to complexity theory we also know an assignment can be verified in a polynomial number of instructions.[1] We then produce an output, `bit_pattern`, using an "if-then-else" instruction that evaluates to zero if the Boolean variables do not satisfy the clauses, and has all bits set if they do.

For the switching phase, the CSWP instruction counter, `insn_count`, is read to learn how many instructions there are in the decision phase of the CSWP program. We then emit a pattern that repeatedly loads the variable `bit_pattern` and then zero. The effect of this is to produce a phase in the program that causes a large amount of switching if the SAT problem p was satisfied; and to not if it was unsatisfiable. In this sequence, a satisfying assignment will cause the switching phase to flip every bit in the output datapath, every instruction; while no switching will occur otherwise.

We have thus introduced a gap in the switching activity of the CSWP program q, that is governed by whether the SAT problem p is satisfiable or not. We use the length of the decision phase of the program to ensure that the switching phase is at least the length of the decision phase, plus one or two instructions. This ensures that, regardless of the amount of switching in the decision phase, the switching phase dominates the switching activity of the program. When solving CSWP, if the SAT problem p were satisfiable, then the maximum amount of switching would include the switching phase, and the CSWP solver would be obliged to yield an input to the program that satisfied the reduced SAT problem. If p is unsatisfiable, it would instead yield whatever input maximised the switching in the decision phase.

We use the size of the gap to demonstrate that CSWP cannot be approximated. In the previous example the switching phase constitutes at least $1/2$ of the possible switching activity: if one possessed an algorithm to approximate such a CSWP program to a factor of $1/2$, then it would

[1] We note that, as the inputs to `CheckSat` are free variables, we are essentially modelling a SAT solver.

be obliged to activate the switching phase of any CSWP program reduced from a satisfiable SAT formula, thus acting as an oracle for an NP-complete problem. Under the P $\neq$ NP assumption, such an algorithm does not exist. $\square$

Furthermore, we are able to extend this result to any constant factor. For any value of ε and SAT instance p, take the desired approximation factor $f = 1 - \varepsilon$ and set the length of the switching phase to be $declen \times (1/f)$, where $declen$ is the number of instructions in the decision phase of CSWP q. Such a program will have a gap of at least $1/f$ times the decision phase, that depends entirely on the satisfiability of p, and thus cannot be approximated. One need not limit this approach to a constant factor either: one may instead compute f to be some factor that is a polynomial function of the size of SAT problem p, for example n^2, and achieve the same result. This shows that there can be no useful approximation factor for CSWP.

The safety of this result depends on the reduction to q being polynomial in the number of variables n in p. Introducing the variables of p scales linearly with n, checking the satisfiability of a particular assignment is known to be checkable in polynomial time, and the evaluation of the result into `bit_pattern` is constant-time. The decision phase is thus a polynomial reduction. The switching phase is controlled by the length of the decision phase (which is polynomial), but also the desired approximation factor. Provided the approximation factor is polynomial, the full reduction is also polynomial.

7 Discussion

We consider here the scope of these results for the analysis of dynamic energy in general, their implications with regard to the feasibility of such analysis, and potential alternative methods for analysing energy in systems.

7.1 Scope

The immediate outcome of these proofs is that, with a program's switching activity shown to be NP-hard, calculating the worst-case dynamic energy for a program is infeasible. Our result relies on the analysis that the cost of switching activity is dominated by switching in the output datapath, meaning the majority of dynamic energy cannot be calculated. Clearly, the exact cost of such switching will vary between processors, however our result may be used as a basis for demonstrating that calculating the switching in other components of the processor is also infeasible. For example, because all inputs to instructions are inevitably the output of some other instruction, it is reasonable to assume that it is NP-hard to estimate the switching activity of input operands too.

Branch prediction and data caches will contribute dynamic energy too. These also depend on program inputs to an extent, but are not modelled by our CSWP formalisation. Other processor components may contribute dynamic energy that is not affected by the inputs to a program — the switching associated with instruction logic (decode, functional unit activation, instruction cache) will contribute dynamic energy regardless of the program input.

Finally, features in some processors, such as out-of-order execution may defeat this analysis. The circuit switching cost is still present, and its determination will still be NP-hard, however it may occur in an unpredictable fashion that depends on a processor-internal unobservable instruction execution schedule.

7.2 Implications

The infeasibility result for estimating dynamic energy over time prevents the construction of an instruction level energy model that identifies a worst case switching cost for each instruction in a given program. Existing techniques that apply WCEC analysis [12, 17, 7, 18, 6, 32] to software can thus never have an energy model that accurately accounts for worst case achievable dynamic energy of the given computation. One may instead, given an accurate model of the switching costs within a processor, assume that every circuit switches in every clock cycle, which will achieve a safe upper bound on the energy consumption [12, 32]. The over-approximation inherent with this approach will not yield a tight bound. For example, on the XMOS XS1-L, with dynamic energy contributing 42 % of energy consumption, one would have a similarly sized amount of potential over-approximation regarding the energy consumption of any execution.

7.3 Alternatives

Viable techniques for estimating dynamic energy consumption can come from a variety of fields: in particular, statistical methods [22] may be effective for determining the distribution of energy consumption under normal operation. Such a model may be used by assuming that the most energy the program can consume occurs only 1 % of the time, and taking the energy value corresponding to that probability as the program's energy consumption. This does not provide a safe upper bound on the program's energy consumption as it is based on normal operation. However, on the balance of probability it is very likely to present an upper bound. Depending on the use case such bound may be more useful in making energy consumption of software transparent to developers than gross over-approximation.

Another alternative is to initially assume that the maximum dynamic energy is dissipated by every instruction, and then use information gained by static analysis to lower this figure. For example, if one can determine the integer interval of a variable, then one can potentially determine the maximum switching of a specific instruction to be lower than its general maximum. The closeness of the bound determined would depend heavily on the capabilities of the static analyses applied, and the extent to which the program lends itself to static analysis. However, our results show that this technique cannot generally provide an accurate bound.

In all circumstances, alternative estimation techniques will posses some level of unsafeness or incompleteness, otherwise they will be NP-hard in the general case as proven in this work.

8 Conclusions

In this paper we have considered the energy consumption in a processor that can directly be attributed to the data or inputs to the software being executed, and demonstrate that the general analysis of circuit switching in processor datapaths — the "circuit switching problem" — is NP-hard. Further, we demonstrate that there is no efficient algorithm for approximating the circuit switching problem to any constant or polynomial factor. We conclude that the analysis of worst-case energy as caused by software cannot be achieved in an efficient manner, leaving a necessary uncertainty factor corresponding to the amount of dynamic energy controlled by processor datapaths. In addition, we consider alternate questions that one could pose that do not amount to worst-case analysis and how they can contribute to understanding software energy consumption.

In the future we believe that work is best focused on statistical methods of modelling program energy consumption, or otherwise characterising the way in which software operates. Critically, we cannot continue to think in terms of "worst case" behaviour, but must instead turn to empirical methods for analysing program behaviours rather than formally proving them.

Acknowledgements

We would like to thank David May, Benjamin Sach, Kyriakos Georgiou and James Pallister for their insights into and motivation of this work. The research leading to these results has received funding from the European Union 7th Framework Programme (FP7/2007-2013) under grant agreement no 318337, ENTRA - Whole-Systems Energy Transparency.

References

[1] G. Ascia et al. "An instruction-level power analysis model with data dependency". English. In: *VLSI DESIGN* 12.2 (2001), 245–273. ISSN: 1065-514X. DOI: {10.1155/2001/82129}.

[2] A. Biere et al. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009, pp. 611–632. ISBN: 1586039296, 9781586039295.

[3] A. Carroll and G. Heiser. "An analysis of power consumption in a smartphone". In: *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, p. 21. URL: http://portal.acm.org/citation.cfm?id=1855840.1855861.

[4] L. Chandra and S. Roy. "Estimation of energy consumed by software in processor caches". In: *2008 IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, Apr. 2008, pp. 21–24. ISBN: 978-1-4244-1616-5. DOI: 10.1109/VDAT.2008.4542403. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4542403.

[5] A. T. Freitas, H. C. Neto, and A. L. Oliveira. "On the cmoplexity of Power Estimation Problems". 2004.

[6] K. Georgiou, S. Kerrison, and K. Eder. *On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs*. Tech. rep. University of Bristol, 2015.

[7] N. Grech et al. "Static Analysis of Energy Consumption for LLVM IR Programs". In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '15. Sankt Goar, Germany: ACM, 2015, pp. 12–21. ISBN: 978-1-4503-3593-5.

[8] H. Hajimiri, K. Rahmani, and P. Mishra. "Efficient Peak Power Estimation Using Probabilistic Cost-Benefit Analysis". In: *VLSI Design (VLSID), 2015 28th International Conference on*. 2015, pp. 369–374. DOI: 10.1109/VLSID.2015.68.

[9] R. Hameed et al. "Understanding sources of inefficiency in general-purpose chips". In: *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10* (2010), p. 37. DOI: 10.1145/1815961.1815968. URL: http://portal.acm.org/citation.cfm?doid=1815961.1815968.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728.

[11] Hsiao et al. "K2: an estimator for peak sustainable power of VLSI circuits". In: *Low Power Electronics and Design* (1997).

[12] R. Jayaseelan, T. Mitra, and X. Li. "Estimating the Worst-Case Energy Consumption of Embedded Software". In: *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. RTAS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 81–90. ISBN: 0-7695-2516-4. DOI: 10.1109/RTAS.2006.17. URL: http://dx.doi.org/10.1109/RTAS.2006.17.

[13] D. S. Johnson. "Approximation Algorithms for Combinatorial Problems". In: *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*. STOC '73. Austin, Texas, USA: ACM, 1973, pp. 38–49. DOI: 10.1145/800125.804034. URL: http://doi.acm.org/10.1145/800125.804034.

[14] D. Kearney and N. W. Bergmann. "Performance evaluation of asynchronous logic pipelines with data dependent processing delays". In: *Asynchronous Design Methodologies, 1995. Proceedings., Second Working Conference on*. 1995, pp. 4–13. DOI: 10.1109/WCADM.1995.514637.

[15] S. Kerrison and K. Eder. "Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor". In: *ACM Trans. Embedded Comput. Syst.* 14.3 (2015), p. 56. DOI: 10.1145/2700104. URL: http://doi.acm.org/10.1145/2700104.

[16] Y. Lee and S. Kim. "DRAM energy reduction by prefetching-based memory traffic clustering". In: *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI - GLSVLSI '11* (2011), p. 103. DOI: 10.1145/1973009.1973031. URL: http://portal.acm.org/citation.cfm?doid=1973009.1973031.

[17] U. Liqat et al. "Energy Consumption Analysis of Programs based on XMOS ISA-Level Models". In: *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*. Vol. 8901. Lecture Notes in Computer Science. Springer, 2014, pp. 72–90. ISBN: 978-3-319-14124-4.

[18] U. Liqat et al. "Inferring Energy Consumption at Different Software Levels: ISA vs. LLVM IR". In: *Proc. of FOPARA*. LNCS. To Appear. Springer, 2015.

[19] D. May. *The XMOS XS1 Architecture. Available online: http://www.xmos.com/published/xmos-xs1-architecture*. 2013.

[20] P. M. Morgado, P. F. Flores, and L. M. Silveira. "Generating Realistic Stimuli for Accurate Power Grid Analysis". In: *ACM Trans. Des. Autom. Electron. Syst.* 14.3 (June 2009), 40:1–40:26. ISSN: 1084-4309. DOI: 10.1145/1529255.1529262. URL: http://doi.acm.org/10.1145/1529255.1529262.

[21] J. Pallister et al. "A high-level model of embedded flash energy consumption". In: *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems - CASES '14*. New York, New York, USA: ACM Press, 2014, pp. 1–9. ISBN: 9781450330503. DOI: 10.1145/2656106.2656108. URL: http://dl.acm.org/citation.cfm?doid=2656106.2656108.

[22] J. Pallister et al. "Data dependent energy modelling: A worst case perspective". In: *CoRR* abs/1505.03374 (2015). URL: http://arxiv.org/abs/1505.03374.

[23] A. Parikh et al. "Instruction Scheduling for Low Power". In: *Journal of VLSI signal processing systems for signal, image and video technology* 37.1 (2004), pp. 129–149. DOI: 10.1023/B:VLSI.0000017007.28247.f6.

[24] S. Rivoire, P. Ranganathan, and C. Kozyrakis. "A Comparison of High-level Full-system Power Models". In: *Proceedings of the 2008 Conference on Power Aware Computing and Systems*. HotPower'08. San Diego, California: USENIX Association, 2008, pp. 3–3. URL: http://dl.acm.org/citation.cfm?id=1855610.1855613.

[25] Y. S. Shao and D. Brooks. "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor". In: *International Symposium on Low Power Electronics and Design (ISLPED)*. November. IEEE, Sept. 2013, pp. 389–394. ISBN: 978-1-4799-1235-3. DOI: 10.1109/ISLPED.2013.6629328. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6629328.

[26] A. Sinha and A. P. Chandrakasan. "Energy Aware Software". In: *Proceedings of the 13th International Conference on VLSI Design*. VLSID '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 50–. ISBN: 0-7695-0487-6. URL: http://dl.acm.org/citation.cfm?id=580736.835252.

[27] S. Steinke et al. "An Accurate and Fine Grain Instruction-level Energy Model Supporting Software Optimizations". In: *Proceedings of PATMOS*. 2001.

[28] L. Thiele and R. Wilhelm. "Design for Timing Predictability". In: *Real-Time Syst.* 28.2-3 (Nov. 2004), pp. 157–177. ISSN: 0922-6443. DOI: 10.1023/B:TIME.0000045316.66276.6e. URL: http://dx.doi.org/10.1023/B:TIME.0000045316.66276.6e.

[29] V. Tiwari, S. Malik, and A. Wolfe. "Power analysis of embedded software: a first step towards software power minimization". In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 2.4 (1994), pp. 437–445. ISSN: 1063-8210. DOI: 10.1109/92.335012.

[30] V. Tiwari et al. "Instruction Level Power Analysis and Optimization of Software". In: *J. VLSI Signal Process. Syst.* 13.2-3 (Aug. 1996), pp. 223–238. ISSN: 0922-5773. DOI: 10.1007/BF01130407. URL: http://dx.doi.org/10.1007/BF01130407.

[31] V. V. Vazirani. *Approximation Algorithms*. New York, NY, USA: Springer-Verlag New York, Inc., 2001, pp. 306–311. ISBN: 3-540-65367-8.

[32] P. Wägemann et al. "Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems". In: *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*. 2015, pp. 105–114. DOI: 10.1109/ECRTS.2015.17.

[33] R. Wilhelm et al. "The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools". In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: 10.1145/1347375.1347389.