# Inferring Energy Bounds Statically
# by Evolutionary Analysis of Basic Blocks

U. Liqat [†]
umer.liqat@imdea.org

Z. Banković [†]
zorana.bankovic@imdea.org

P. Lopez-Garcia [*] [†]
pedro.lopez@imdea.org

M.V. Hermenegildo [†] [‡]
manuel.hermenegildo@imdea.org

## ABSTRACT

We are currently witnessing an increasing number of energy-bound devices, including in some cases mission critical systems, for which there is a need to optimize their energy consumption and verify that they will perform their function within the available energy budget. In this work we propose a novel parametric approach to estimating tight energy bounds (both upper and lower) that are practical for energy verification and optimization applications in embedded systems. Our approach consists in dividing a program into basic ("branchless") blocks, establishing the maximal (resp. minimal) energy consumption for each block using an evolutionary algorithm, and combining the obtained values according to the program control flow, using static analysis, to produce energy bound functions. Such functions depend on input data sizes, and return upper or lower bounds on the energy consumption of the program for any given set of input values of those sizes, without running the program. The approach has been tested on XMOS chips, but is general enough to be applied to any microprocessor and programming language. Our experimental results show that the bounds obtained by our prototype tool can be tight while remaining on the safe side of budgets in practice.

## Keywords

Energy Consumption Analysis, Energy Modeling, Embedded Systems, Static Analysis, Evolutionary Algorithms.

## 1. INTRODUCTION

We are witnessing an ever-increasing performance and ubiquity of battery- and/or harvested energy-powered devices. An important trend in this context is the so called *Internet of Things* paradigm. It is estimated that by the year 2020, about 50 billion small autonomous devices, embedded in all kind of objects, in our clothes, or stuck to our bodies will operate and intercommunicate continuously for long periods of time, such as years. Such devices rely on small batteries or energy harvested from the environment, which implies that their energy consumption should be very low.

Although there have been improvements in battery and energy harvesting technology, they alone are often not enough to achieve the required level of energy consumption to fully support *Internet of Things* and other energy-bound applications. Thus, better techniques for optimizing the energy consumption of embedded systems are needed. While many energy-saving features have been de-

veloped for hardware, far more energy savings remain to be tapped by improving the software that runs on these devices. In addition, there are many critical embedded applications (e.g., sensor-based) for which, beyond optimizing energy consumption, it is actually crucial to guarantee that execution will complete within a specified energy budget, e.g., before the available system energy runs out.

In this work we focus on the *static* estimation of the energy consumed by program executions (i.e., at compile time, before actually running them), as a basis for energy optimization and verification. Such estimations are given as functions on input data sizes, since data sizes typically influence the energy consumed by a program, but are not known at compile time. This approach allows abstracting away such sizes and inferring energy consumption in a way that is parametric on them.

Different types of resource usage estimations are possible, such as, e.g., probabilistic, average, or safe bounds. However, not all types of estimations are valid or useful for a given application. For example, in order to verify/certify energy budgets, *safe upper- and lower-bounds* on energy consumption are required [16, 15]. Unfortunately, current approaches that guarantee that the bounds are always safe tend to compromise their tightness seriously, inferring overly conservative bounds, which are not useful in practice. With this safety/tightness trade-off in mind, our goal is the development of an analysis that infers tight bounds that are on the safe side in most cases, in order to be practical for verification applications, as well as for energy optimization.

Of the small number of static energy analyses proposed to date, only a few [20, 13, 12] use resource analysis frameworks that are aimed at inferring safe upper and lower bounds on the resources used by program executions. A crucial component in order for such frameworks to infer hardware-dependent resources, and, in particular, energy, is a low-level resource usage model, such as, e.g., a model of the energy consumption of individual instructions. Examples of such models are [11], at the Java bytecode level, or [10], at the assembly level.

Clearly, the accuracy of the bounds inferred by analysis depends on the nature and accuracy of the low-level models. Unfortunately, models such as [11, 10] provide *average* energy consumption values or functions, which are not really suitable for upper- or lower-bounds analysis. Furthermore, trying to obtain instruction-level models that provide strict safe energy bounds would result in very conservative bounds. Although when fed with such models the static analysis would infer high-level energy consumption functions providing strictly safe bounds, these bounds would not be useful in general because of their large inaccuracy. For this reason, the analyses in [20, 13, 12] used instead the already mentioned instruction level average energy models [11, 10]. However, this meant

that the energy functions inferred for the whole program were not strict bounds, but rather approximations of the actual bounds, and could possibly be below or above. This trade-off between safety and accuracy is a major challenge in energy analysis. In this paper we address this challenge by providing a technique for the generation of lower-level energy models which are useful and effective in practice for verification-type applications.

The main source of inaccuracy in current instruction-level energy models is inter-instruction dependence (including also data dependence), which is not captured in most models. On the other hand, the concrete sequences of instructions that appear in programs exhibit worst cases that are not as pessimistic as considering the worst case for each of the individual intervening instructions. Based on this, we decided to use *branchless blocks* of assembly instructions as the modeling unit instead of individual instructions. We divide the (assembly) program into such *basic blocks*, each a straight-line code sequence with exactly one entry to the block (the first instruction) and one exit from the block (the last instruction). We then measure the energy consumption of these basic blocks, and determine a maximal (resp. minimal) energy consumption for each block. In this way the inter-instruction data dependence discussed above and other factors are accounted for. The energy values obtained for each block are fed to our static resource analysis, which combines them according to the program control flow and produces the energy bound functions.

In order to find the maximum and minimum energy consumption of each basic block we use an evolutionary algorithm (EA). We vary the input values and take energy measurements directly from the hardware for each input combination. This way, we take advantage of the fast search space exploration provided by EAs. EAs have also been used for estimating the worst case energy consumption of *whole* programs [22], due to their fast exploration of the search space. However, if there are data-dependent branches in the programs, which is often the case, applying this approach to whole programs (or program segments that contain branches) quickly loses accuracy, since different input combinations can trigger different sets of instructions [22]. In contrast, our approach combines EAs and static analysis techniques in order to get the best of both worlds. We take out the treatment of data-dependent branches from the EA, so that the same sequence of instructions is always executed in each basic block. The worst (resp. best) case energy of the basic blocks is estimated by the EA with higher accuracy since, not having any branches, the most important deficiency of the EA is avoided. The program control flow dependencies are taken care instead by the static analysis.

In our experiments we focus for concreteness on the energy analysis of programs written in XC [28], running on the XMOS XS1-L architecture. However, our approach is general enough be applied to the analysis of other programming languages (and associated lower level program representations) and architectures as well. XC is a high-level C-based programming language that includes extensions for concurrency, communication, input/output operations, and real-time behavior. Our experimental setup infers energy consumption information by processing the ISA (Instruction Set Architecture) code compiled from XC [28], and reflects it up to the source code level. Such information is provided in the form of *functions on input data sizes*, and is expressed by means of *assertions* [7].

In these experiments, the energy estimations produced by our approach were always safe, in the sense that they over-approximated the actual bounds (i.e., the inferred upper bounds were above the actual upper bounds and the inferred lower bounds below the actual lower bounds). This suggests that, even if we cannot assure

formally that such estimations are always safe, they are quite accurate in the sense that the inferred energy bounds are close to the actual bounds, and that in practice they will also be safe/strict in most cases. We argue that our analysis provides a good practical compromise for the verification/certification of energy budgets.

In summary, the main contributions of this paper are:

- A novel approach that combines dynamic and static analysis techniques for inferring the energy consumption of program executions. The dynamic part is based on EAs, and produces low-level energy models.

- The proposal of a new abstraction level at which to perform the energy modeling of program components using dynamic techniques: basic (branchless) blocks of assembly instructions.

- A method based on EAs to dynamically (i.e., by profiling) obtain practical upper and lower bounds on the energy of such basic blocks, with a good safety/accuracy balance.

- The use of a static analysis that takes care of the program control flow, in order to determine how many times blocks are executed, which combined with the information provided by the block models, infers functions that give the energy of a program and its procedures as functions of input data sizes.

- An experimental study that supports our claims.

In the rest of the paper, Section 2 explains how the information inferred by our approach can be used for the energy consumption verification application. Section 3 explains our technique for energy modeling of program basic blocks. Section 4 shows how these models are used by the static analysis to infer upper- and lower-bounds on the energy consumed by programs as functions of their input data sizes. Section 5 reports on an experimental evaluation of our approach. Related work is discussed in Section 6, and finally Section 7 summarises our conclusions.

## 2. ENERGY CONSUMPTION VERIFICATION/CERTIFICATION

The lower ($E_l$) and the upper bound ($E_u$) inferred by our analysis can be used for energy consumption verification and certification. We refer the reader to [14, 15] for a detailed description on how static analysis information can be used for general resource usage verification within the CiaoPP system, and to [16] for how it can be specialized for verifying energy consumption specifications of embedded programs.

Here we only give some intuitive ideas. Assume that a program specification expresses energy budget $E_b$, e.g., defined by the capacity of the battery, we can conclude the following:

1. $E_u \leq E_b \implies$ the given program can be safely executed within the existing energy budget.

2. $E_l \leq E_b \leq E_u \implies$ it might be possible to execute the program, but we cannot claim it for certain.

3. $E_b < E_l \implies$ it it not possible to execute the program (the system will run out of batteries before program execution is completed.

## 3. ENERGY MODELING OF BLOCKS

As mentioned before, the first step of our energy bounds analysis is to determine upper and lower bounds on the energy consumption of each basic ("branchless") program block. We perform the modeling at this level rather than at the instruction level in order to cater for inter-instruction dependencies. In order to determine such bounds first all the basic blocks of the program are identified, and then the energy consumption of each of these blocks is profiled for different input data using an EA. These steps are explained in the following sections.

### 3.1 Generating the Basic Blocks to be Modeled

A *basic block* over an inter-procedural control flow graph (CFG) is a maximal sequence of distinct instructions, $S_1$ through $S_n$, such that all instructions $S_k, 1 < k < n$ have exactly one in-edge and one out-edge (excluding call/return edges), $S_1$ has one out-edge, and $S_n$ has one in-edge. A basic block therefore has exactly one entry point at $S_1$ and one exit point at $S_n$.

In order to divide a program into such *basic blocks* (for which an upper bound on the energy consumption of the program will be determined using the EA), the program is first compiled to the lower representation, ISA in our case. A data flow analysis of the ISA representation yields an inter-procedural control flow graph (CFG). A final control flow analysis is carried out to infer *basic blocks* from the CFG. These basic blocks are further modified so that they can be run and measured independently by the EA. Modifications for each basic block include:

1. A basic block with $k$ function call instructions is divided into $k + 1$ basic blocks without the function call instructions.

2. A number of special ISA instructions (e.g., *return*, *call*) are omitted from the block. The cost of such instructions is measured separately and added to the cost of the block.

3. Memory read/write instructions are abstracted to a fixed memory region available to each basic block in order to avoid memory violations.

An example of the modification 1 above is shown in Figure 1, Listing 1, which is an ISA representation of a recursive factorial program where the instructions are grouped together into 3 *basic blocks* $B1$, $B2$, and $B3$. Consider *basic block* $B2$. Since it has a (recursive) function call to *fact* at address 12, it is divided further into two blocks in Listing 2, such that the instructions before and after the function call form two blocks $B2_1$ and $B2_2$, respectively. The energy consumption of these two blocks is maximized (minimized) by providing values to the input arguments to the block (see below) using the EA. The energy consumption of $B2$ can then be characterized as:

$$B2_e^A = B2_{1e}^A + B2_{2e}^A + bl_e^A$$

where $B2_{1e}^A$, $B2_{2e}^A$, and $bl_e^A$ denote the energy consumption of the $B2_1$, $B2_2$ blocks and the $bl$ ISA instruction, with approximation $A$ (where $A$=upper or $A$=lower).

For each modified basic block, a set of input arguments is inferred. This set is used for an individual representation to drive the EA algorithm to maximize the energy consumption of the block. For the entry block, the input arguments are derived from the signature of the function. The set $gen(B)$ characterizes the set of variables read without being previously defined in block $B$. It is defined

as:

$$gen(b) = \bigcup_{k=1}^{n} \{v \mid v \in ref(k) \land \forall (j < k).v \notin def(j)\}$$

where $ref(n)$ and $def(n)$ denote the variables referred to and defined/updated at a node $n$ in block $b$ respectively.

For the basic blocks in Figure 1 in Listing 1, the set of input arguments are $gen(B1)$={r0}, $gen(B2_1)$={sp[0x1]}, $gen(B2_2)$={sp[0x1],r0} and $gen(B3) = \emptyset$.

### 3.2 EA for Estimating the Energy of a Basic Block

In the following we detail the most important aspects of the EA used for estimating the maximal (i.e., worst case) and minimal (i.e., best case) energy consumption of a basic block. The only difference between the two algorithms is the way we interpret the objective function: in the first case we want to maximize it, while in the second we want to minimize it.

**Individual.** The search space dimensions are the different input variables to the blocks. Our goal is to find the combination of input values which maximizes (minimizes) the energy of each block. The set of input variables to a block is inferred using a dataflow analysis (explained in the next section). Thus, an individual is simply an array of input values given in the order of their appearance in the block. The input values in an individual are coded as integers, since they represent 32-bit values stored in different hardware registers.

The majority of individuals are initialised with random 32-bit numbers. However, we also include corner cases to the initial population that are known to cause high (low) energy consumption for particular instructions. For example all 1s for high energy consumption, or all 0s for low energy consumption as operands to a multiply ISA instruction. This speeds up the EA algorithm in finding inputs to some basic blocks that maximize/minimize their overall energy consumption.

**Crossover.** The crossover operation is implemented as an even-odd crossover, since it provides more variability than a standard $n$-point crossover. In this crossover the first child is created by taking the first element and every other one after it from one of the parents, e.g., the mother. The second element and every other one come from the other parent, i.e., the father. The second child is created in the opposite way: the first element and every other one after it are taken from the father, while the second and every other one come from the mother. The process is depicted in Figure 2, where $P1$ and $P2$ are the parents, and $C1$ and $C2$ are their children created by the crossover operation.

**Mutation.** For the purpose of this work we have created a custom mutation operator. Since the energy consumption in digital circuits is mainly the result of bit flipping, we believe that the most optimal way to explore the search space is by performing some bit flipping in the mutation operation. This is implemented in the following way. For each gene (i.e., input value to the basic block):

1. We create a random 32-bit integer, i.e., a random mask.

2. Then we perform the XOR operation of that integer and the corresponding gene. This way, we perform random flipping of the bits of each gene, since we only flip the bits of the gene at positions where the value of the random mask is 1.

The process is depicted in Figure 3, where the input values are given as binary numbers.

**Objective function.** The objective function that we want to maximize (minimize) is the energy of a basic block, which is measured

## Listing 1: Basic blocks of a factorial function.

```
<fact>:
   01: entsp 0x2
   02: stw   r0, sp[0x1]
B1 03: ldw   r1, sp[0x1]
   04: ldc   r0, 0x0
   05: lss   r0, r0, r1
   06: bf    r0, <08>

   07: bu    <010>
   10: ldw   r0, sp[0x1]
   11: sub   r0, r0, 0x1
B2 12: bl    <fact>
   13: ldw   r1, sp[0x1]
   14: mul   r0, r1, r0
   15: retsp 0x2

B3 08: mkmsk r0, 0x1
   09: retsp 0x2
```

## Listing 2: Modified basic blocks.

```
<fact>:
 01: entsp 0x2
 02: stw   r0, sp[0x1]
 03: ldw   r1, sp[0x1]      B1
 04: ldc   r0, 0x0
 05: lss   r0, r0, r1
 06: bf    r0, <08_NEW>
08_NEW:

 07: bu    <010>
 10: ldw   r0, sp[0x1]      B2₁
 11: sub   r0, r0, 0x1

 12: bl <fact>

 13: ldw   r1, sp[0x1]
 14: mul   r0, r1, r0       B2₂
 15: retsp 0x2

 08: mkmsk r0, 0x1
 09: retsp 0x2             B3
```

block before call

block after call

**Figure 1: Example: Basic block modifications.**



**Figure 2: Example of even-odd crossover.**



**Figure 3: Mutation.**

by executing it on an Instruction Set Simulation (ISS) to collect cycle counts.

The XMOS XS1 architecture used in our experiments does not have these pipeline effects by design, since exactly one instruction per thread is executed in a 4-stage pipeline (more details in Section 5.1).

## 4. ENERGY CONSUMPTION OF THE PROGRAM

Once the energy models of each basic block of the program are known, the energy consumption of the whole program is bounded by a static analyser that takes into account the control flow of the program and infers safe upper/lower bounds on its energy consumption. We have implemented such analyser by specialising the generic resource analysis framework provided by CiaoPP [24] for programs written in the XC programming language [28] and running on the XMOS XS1-L architecture. We have also written the necessary code (i.e., assertions [7]) to feed such analyser with the block-level upper/lower bound energy model obtained by using the technique explained in Section 3.

The generic resource analyser ensures that the inferred bounds are strict/safe if it is fed with energy models providing safe bounds. As mentioned in the introduction, in [13] we performed a previous instantiation of such generic analyser by using the instruction-level energy model described in [10], which provided average energy
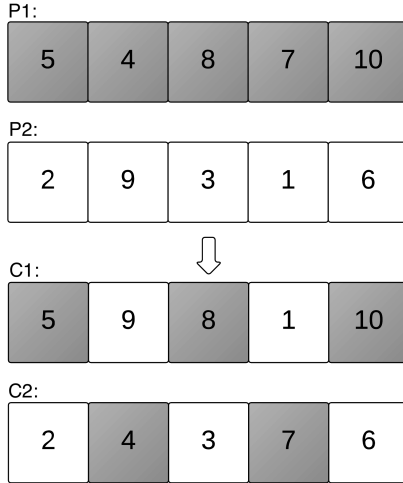
directly from the chip. The concrete setting of the experiment will be explained in the following section.

In general, pipeline effects such as stalls (to resolve pipeline hazards), which depend on the state of the processor at the start of the execution of a basic block, can affect the upper/lower bound estimated on the energy consumption of such block. In our approach intra-block pipeline effects are accounted for, since, the dependences among the instructions within a block are preserved. However, the inter-block pipeline effects need to be accounted for. These can be modeled in a conservative way by assuming a maximum stall penalty for the upper bound estimation of each block (e.g., by adding a stall penalty, say three cycles, to the execution time of the block). Similarly, for the lower bound estimation a zero stall penalty can be used. To approximate this effect, in [3], the authors characterize each block through pairwise executions with all of its possible predecessors. Each basic block pair is characterized
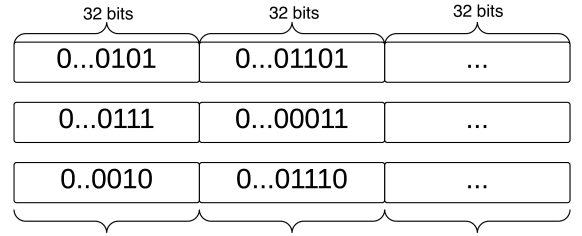
values. As a result, the analysis inferred an upper-bound energy function for the whole program that was an approximation of the actual upper bound, and could possibly be below it.

The analysis is general enough to be applied to other programming languages and architectures (see [13, 12] for details) provided that energy models for each architecture exist. It enables a programmer to symbolically bound the energy consumption of a program $P$ on input data $\bar{x}$ without actually running $P(\bar{x})$. It 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 $\bar{x}$. The transformation-based analysis framework of [13, 12] transforms the assembly (or LLVM IR) representation of the program into an intermediate semantic program representation (HC IR), that the analysis operates on, which is a series of connected code blocks, represented as Horn Clauses. The analyser deals with this HC IR always in the same way, independent of where it originates from, setting up cost equations for all code blocks (predicates).

Consider the example in Listing 1. The recursive cost equations are set up over the function *fact* that characterize the energy consumption of the whole function using the approximation $A$ of each block inferred by the EA:

$$fact_e^A(R0) \quad = \quad B1_e^A + fact\_aux_e^A(0 \le R0, R0)$$

$$fact\_aux_e^A(B, R0) = \begin{cases} B2_e^A + fact_e^A(R0-1) & \text{if } B \text{ is } \texttt{true} \\ B3_e^A & \text{if } B \text{ is } \texttt{false} \end{cases}$$

The cost of the *fact* function is captured by the equation $fact_e^A(R0)$ under an approximation $A$ (e.g., upper/lower) which in turn depends on $B1_e^A$ (i.e., the energy consumption of block $B1$) and the equation $fact\_aux_e^A$, which represents the branching originated from the last instruction of block *B1*. It captures the cost of blocks *B2* and *B3* based on the condition on the input size *R0*.

If we assume (for simplicity of exposition) that each basic block has unitary cost in terms of energy consumption, i.e., $Bi_e = 1$ for all $i$, we obtain the energy consumed by `fact` as a function of its input data size $(R0)$: $fact_e(R0) = R0 + 1$.

The functions inferred by the static analysis are arithmetic functions (polynomial, exponential, logarithmic, etc.) that depend on input data sizes (natural numbers).

# 5. EXPERIMENTAL EVALUATION

In this Section we report on an experimental evaluation of our approach to inferring both upper and lower bounds on the energy consumed by program executions, given as functions on input data sizes. The experiments have been performed with programs written in XC running on the XMOS XS1-L architecture. However, as already said, our approach is general enough be applied to the analysis of other programming languages (and associated lower level program representations) and architectures as well.

## 5.1 Evaluation Platform

We use a hardware and software platform created by XMOS that enables us to measure the energy [19], time, and power used during program executions on real hardware. The developed board is a dual-tile board that contains an XS1-A16-128-FB217 processor. The board is fed with a 3.3 V power supply, and supports voltage scaling, although both tiles have to run at the same voltage supply. It also supports frequency scaling, where the tiles can have different frequencies. The XMOS XS1 [17] is a cache-less, predictable architecture by design and manages threads on the hardware. The threads are executed in a round-robin fashion, using a 4-stage pipeline which only permits a single instruction per thread

to be active within the pipeline at the same time. This restriction avoids pipeline hazards.

In order to support the process of measuring power, the following has been implemented:

- An extension to the XMOS toolchain that allows power measurements to be recorded and/or displayed in real time. In essence, a small shunt resistor has been added in series with the voltage supply. By measuring the voltage drop on the shunt, we can calculate the current $I$, which is also the current of the voltage supply, since the shunt is connected in series. In this way, we estimate the power consumption as $V_{sup} \cdot I$, where $V_{sup}$ is the voltage of the power supply.

- A variant of the XTAG-2 debug adapter (called XTAG3) that enables power to be measured [31]. Basically, it has an extra connector that carries analog signals necessary to estimate the power consumption, as explained above. The measurements regarding these signals are transported to the host computer over USB using the xSCOPE interface [32]. In addition, a protocol that enables power measurements and application probing to be performed simultaneously, and data to be transported simultaneously over the USB connection to the host computer, has been designed.

The tool that collects data from the XTAG is `xgdb`, the debugger that is part of the XMOS toolchain. `xgdb` connects to the XTAG over a USB interface (using libusb), and reads both ordinary xSCOPE traffic and voltage/current measurements. The collected data is normally stored in an XML file, or instead, `xgdb` can pipe the data directly into an analysis program that can only record data that is relevant (between start and end) and only compute the relevant metrics (maximum current, total energy, etc.).

## 5.2 Results and Discussion

The aim of the experimental evaluation is to perform a first comparison of actual hardware energy measurements against the upper- and lower-bounds on energy consumption obtained by evaluating the functions inferred by our proposed approach (which depend on input data sizes), for each program considered and for different input data sizes. The actual energy consumption of the programs, for each value of input data sizes, is measured with the evaluation platform, i.e., the same used to build the upper- and lower-bound models of the blocks of each program.

| Program | Upper/Lower Bounds (nJ)$\times 10^3$ | vs. HW |
|---|---|---|
| $fact(N)$ | $ub = 5.1\,N + 4.2$ | 7% |
| | $lb = 4.1\,N + 3.8$ | -11.7% |
| $fibonacci(N)$ | $ub = 5.2\,lucas(N)^1 + 6\,fib(N) - 6.6$ | 8.71% |
| | $lb = 4.5\,lucas(N) + 5\,fib(N) - 4.2$ | -4.69% |
| $reverse(N)$ | $ub = 3.7\,N + 13.3$ | 8% |
| | $lb = 2.95\,N + 12$ | -8.8% |
| $findMax(N)$ | $ub = 5\,N + 6.9$ | 8.7% |
| | $lb = 3.3\,N + 5.6$ | -9.1% |
| $fir(N)$ | $ub = 6\,N + 26.4$ | 8.9% |
| | $lb = 4.8\,N + 22.9$ | -9.7% |
| $biquad(N)$ | $ub = 29.6\,N + 10$ | 9.8% |
| | $lb = 23.5\,N + 9$ | -11.9% |

**Table 1: Upper and lower bounds accuracy.**

---

[1]The mathematical function $lucas(n)$ satisfies the recurrence relation $lucas(n) = lucas(n-1) + lucas(n-2)$ with $lucas(1) = 1$ and $lucas(2) = 3$.

A number of selected benchmarks are shown in Table 1 that are either iterative or recursive. The **Upper/lower Bounds** column depicts the energy estimation functions (on input data sizes) for upper and lower bounds. The column **vs. HW** shows the average over- and under-approximations of the estimation versus the actual measurements on the hardware.

The first two benchmarks are small arithmetic benchmarks. The third benchmark *reverse*(N) reverses elements of an input array of size *N*. The list of benchmarks also includes two filter benchmarks, namely *biquad* and *fir* (Finite Impulse Response). Both programs attenuate or amplify specific frequency ranges of a given input signal. The *fir*(N) benchmark computes the inner-product of two vectors: a vector of input samples, and a vector of coefficients. The more coefficients, the higher the fidelity, and the lower the frequencies that can be filtered. The *biquad*(N) benchmark is an equaliser, i.e., it takes a signal and attenuates/amplifies different frequency bands. It uses a cascade of Biquad filters where each filter attenuates or amplifies one specific frequency range. The energy consumed depends on the number of banks *N*, typically between 3 and 30 for an audio equaliser. A higher number of banks enables a designer to create more precise frequency response curves. A simple *findMax* benchmark (finding the maximum number in an array) is also included in the list. This is a program where data-dependent branching can bring significant variations of the worst (best) case energy consumption. Note that unlike the first three benchmarks, *fir*, *biquad*, and *findMax* all have data-dependent branches..
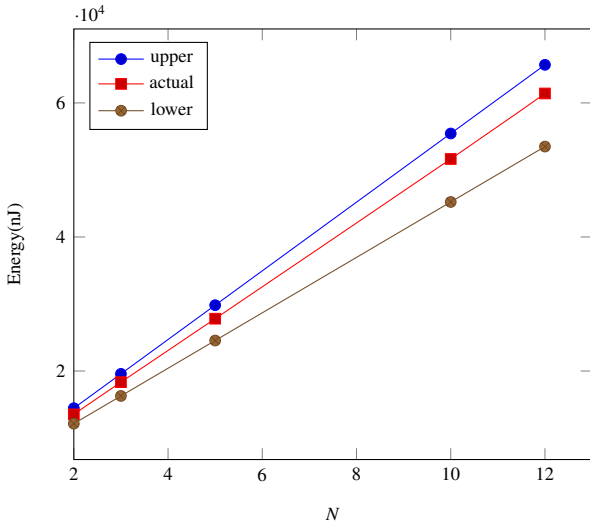


**Figure 4:** *fact* **upper/lower bounds vs. actual measurement.**

Figure 4 depicts the upper/lower bound inferred by the analysis vs. the actual measurement on the hardware for the factorial program. The actual program consumption is measured for several values of *N*, the input value, resulting in the middle curve. The other two curves are the result of plotting the upper- and lower-bound energy functions for different sizes (in the case of an integer, its size is its value).

The upper bound values inferred by the static analysis and the EA over-approximate the actual hardware measurements by 7%, whereas the lower-bound values under-approximate the actual measurements by 11.7% –see Table 1.

The *findMax* benchmark, which, as mentioned before, has significant data dependent branching, is shown in Figure 5. Unlike *fact*, the upper and lower-bounds in *findMax* are more distant due
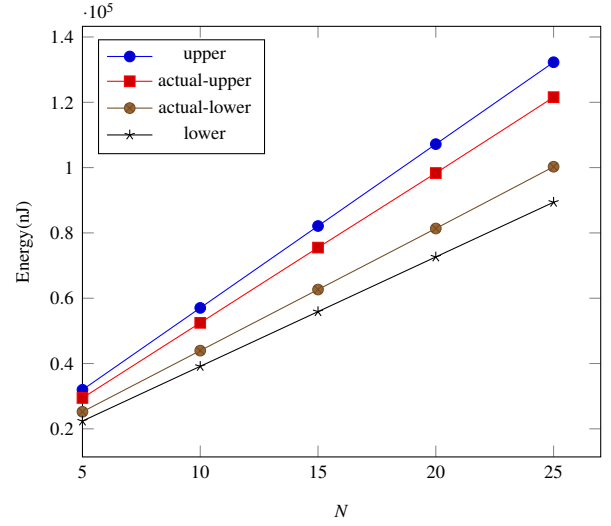


**Figure 5:** *findMax* **example upper/lower bounds vs actual-lower and upper energy consumption measurement based on data.**

to the data sensitive branching. A call to *findMax* with a sorted array in ascending order will discover a new *max* element in each iteration and hence update the current *max* variable resulting in an actual worst case of the algorithm. In contrast, if the array is sorted in descending order then the algorithm will find the *max* element in the first iteration and the rest of the iterations will never update the current *max* variable, resulting in the actual best case.

Figure 5 depicts the upper and lower bounds inferred by the static analysis as well as the actual worst and best case measurements of *findMax* (first with ascending order and then with descending order array data). The upper and lower bounds inferred are compared against the actual worst and best case measurements. The upper bound over-approximates by 8.7% whereas the lower bound under-approximates by 9.1%. Note that it is not always trivial to find data that exhibit program worst and best case behaviors.

In Table 2, different executions of the *findMax* benchmark are shown for particular input sizes **N** but using a random data input array in one case and actual worst/best case array input data in the other case. Column **N** shows the size of the input array. Column **Cost App** indicates the type of approximation of the automatically inferred cost functions which estimate energy consumption (depending on input data size **N**): upper bound (**U**) and lower bound (**L**). Such energy functions for the *findMax* benchmark are shown in Table 1. In order to assess the accuracy of the cost functions we have evaluated them for particular sets of input data corresponding to different input (array) sizes (*N*), yielding different energy consumption estimations. We have then compared such estimations (column **Est**) with the observed energy consumptions of the hardware measurements (column **Obs**). Column **D** shows the relative harmonic difference between the estimated and the observed energy consumption, given by the formula:

$$rel\_harmonic\_diff(Est, Obs) = \frac{(Est - Obs) \times (\frac{1}{Est} + \frac{1}{Obs})}{2}$$

The inaccuracies in the energy estimations of our technique come mainly from two sources: the energy model, which assigns an energy value to each basic block as described in Section 3, and the Static Resource Analysis (SRA), described in Section 4, which estimates the number of times that the basic blocks are executed de-

pending on the input data sizes.

In order to investigate the source(s) of inaccuracies, we have also introduced Column **Prof**. It shows the result of estimating the energy consumption using the energy model and assuming that the SRA was perfect and estimated the exact number of times that the basic blocks were executed. This obviously represents the case in which all loss of accuracy must be attributed to the energy model. The values in Column **Prof** have been obtained by profiling actual executions of the program with particular input data, where the profiler has been instrumented to record the number of times each basic block is executed. The energy consumption of the program is then obtained by multiplying such numbers by the energy values provided by the energy model for each basic block, and adding all of them. Column **PrD** shows the relative harmonic difference between **Prof** and the observed energy consumption **Obs**, which represents the inaccuracy due to the energy modeling of basic blocks using the EA.

In the case of random data, both the SRA and the energy modeling contribute to the inaccuracy of the energy estimation for the whole program. In contrast, in the second case two sets of array data are used: one that makes $findMax$ exhibit its worst case behavior and another that makes it exhibit the best. These are then compared against the upper- and lower-bound estimations. Since Columns **Est** and **Prof** show the same values in this case, it means that there was no inaccuracy due to the SRA, and that the overall inaccuracy is due to the over- and under-approximation in the EA to model energy consumption of each basic block. In other words, the analysis of the $findMax$ program provides accurate bounds for each data size $N$.

| N | Cost App | Energy(nJ)$\times 10^3$ | | | D % | PrD % |
|---|---|---|---|---|---|---|
| | | Est | Prof | Obs | | |
| Random array data | | | | | | |
| 5 | L | 22.3 | 24.9 | 27.3 | -20.1 | -9.2 |
| | U | 31.9 | 30.2 | | 15.6 | 10 |
| 15 | L | 55.9 | 61.8 | 69.1 | -17 | -11 |
| | U | 82.1 | 75.1 | | 21 | 8.3 |
| 25 | L | 89.4 | 99.6 | 110.9 | -17.6 | -10.7 |
| | U | 132.2 | 120.8 | | 21.7 | 8.5 |
| Actual worst- and best-case array data | | | | | | |
| 5 | L | 22.3 | 22.3 | 25.2 | -12.2 | -12.2 |
| | U | 31.9 | 31.9 | 29.4 | 8.1 | 8.1 |
| 15 | L | 55.9 | 55.9 | 62.6 | -11.3 | -11.3 |
| | U | 82.1 | 82.1 | 75.5 | 8.3 | 8.3 |
| 25 | L | 89.4 | 89.4 | 100.2 | -11.4 | -11.4 |
| | U | 132.2 | 132.2 | 121.5 | 8.4 | 8.4 |

**Table 2:** $findMax$**: Source of inaccuracies in prediction: static analysis vs. energy modeling.**

Regarding the time taken by the EA, it can vary depending on the parameters it is initialized with, as well as the initial population. This population is different every time the EA is initiated, except for a fixed number of individuals that represent corner cases. In the experiments, the EA is run for up to a maximum of 20 generations, and is stopped when the fitness value does not improve for four consecutive generations. In all the experiments the *biquad* benchmark took the most time (a maximum time of 230 minutes) for maximizing the energy consumption. In contrast, the *fact* benchmark took the least time (a maximum time of 121 minutes). The times remained within the 150-200 minutes range on average. Time speedups were also achieved by reusing the EA results for sequences of instructions that were already processed in a previous benchmark

(e.g., return blocks, loop header blocks, etc.). This makes us believe that our approach could be used in practice in an iterative development process, where the developer gets feedback from our tool and modifies the program in order to reduce its energy consumption. The first time the EA is run would take the highest time, since it would have to determine the energy consumption of all the program blocks. After a focused modification of the program that only affects a small number of blocks, most of the results from the previous run could be reused, so that the EA would run much faster during this development process. In other words, the EA processing can easily be made incremental.

The static analysis, on the other hand, is quite efficient, with analysis times of about 4 to 5 seconds on average, despite the naive implementation of the interface with external recurrence equation solvers, which can be improved significantly.

# 6. RELATED WORK

Static dataflow analysis of the energy consumed by program executions has received relatively little attention until recently. An analysis of Java bytecode programs for inferring upper-bounds on energy consumption as functions on input data sizes was proposed in [20], where the Jimple (a typed three-address code) representation of Java bytecode was transformed into Horn Clauses, and a simple energy model at the Java bytecode level [11] was used. However the energy model used average estimations of the Java opcodes and an opcode cost verification found the estimation to be between -5% and 10%. Furthermore, this work did not compare the results with actual, measured energy consumption. A similar approach was proposed in [13] for the analysis of a C-based programming language. It performed a transformation of the assembly code generated by the compilation of the source program into Horn Clauses, which were then analyzed by using the accurate assembly level energy models presented in [10]. The experiments, performed for a number of small numerical programs, showed for the first time that energy bound functions inferred statically from low-level model could be inferred that provided energy consumption estimations were reasonably accurate with respect to actual executions for any input data size.

Similarly to the work presented here, the approaches mentioned above used instantiations for energy consumption of general resource analyzers, namely [21] in [20] and [13], and [24] in [12] and this paper. Such resource analyzers are based on setting up and solving recurrence equations, an approach proposed by Wegbreit [29] that has been developed significantly in subsequent work [23, 4, 5, 26, 21, 1, 24]. Other approaches to static analysis based on the transformation of the analyzed code into another (intermediate) representation have been proposed for analyzing low-level languages [6] and Java (by means of a transformation into Java bytecode) [2]. In [2], cost relations are inferred directly for these bytecode programs, whereas in [20] the bytecode is first transformed into Horn Clauses. The general resource analyzer in [21] was also instantiated in [18] for the estimation of execution times of logic programs running on a bytecode-based abstract machine. The approach used timing models at the bytecode instruction level, for each particular platform, and program-specific mappings to lift such models up to the Horn Clause level, at which the analysis was performed.

Other work has taken as its starting point techniques referred to generally as "WCET" (worst case execution time analyses), which have been applied, usually for imperative languages, in different application domains (see e.g., [30] and its references). These techniques generally require the programmer to bound the number of iterations of loops, and then apply an Implicit Path Enumeration

technique to identify the path of maximal consumption in the control flow graph of the resulting loop-less program. This approach has inspired some worst case energy analyses, such as the one presented in [9]. It distinguishes instruction-specific (not proportional to time, but to data) from pipeline-specific (roughly proportional to time) energy consumption. The approach also takes into account complex issues such as branch prediction and cache misses. However, they rely on the user to identify the input which will trigger the maximal energy consumption. In [27] the same approach is applied and further refined for estimating *hard* (i.e., over-approximated) energy bounds. The main novelty of this work consists in introducing relative energy models (implemented at the LLVM level in this case), where the energy of each instruction is given *in relation to each other* (e.g., if we assume that all the instructions have relative energy 1, this means that they all have the same absolute energy), which does not depend on the specific hardware, but can be applied for all the platforms where a mapping between LLVM and low-level assembly instructions exists. On the other hand, in the situations when the energy bounds are not *hard* (i.e., the application allows their violation) they use a genetic algorithm to obtain an under-approximation of the energy bounds. However, this approach loses accuracy when there are data dependent branches present in the program, since different inputs can lead to the execution of different set of instructions.

A similar approach is used in [22] to find the worst-case energy consumption of two benchmarks using a genetic algorithm. In contrast to our approach, the evolutionary algorithm is applied to whole programs, and these do not have any data-dependent branching. The authors further introduce probability distributions for the transition costs among pairs of independent instructions, which can be then be convolved to give a probability distribution of the energy for a sequence of instructions.

In contrast to the work presented here and in [18], all these WCET-style methods (either for execution time or energy) do not infer cost functions on input data sizes but rather absolute maximum values, and, as mentioned before, they generally require the manual annotation of all loops to express an upper bound on the number of iterations, which can be tedious (or impossible) and effectively reduces the case to that of programs with no loops.

Another alternative approach to WCET-style methods was presented in [8]. It is based on the idea of amortization, which allows inferring more accurate yet safe upper bounds by averaging the worst execution time of operations over time. It was applied to a functional language, but the approach is in principle generally applicable. A timing analysis based on game-theoretic learning was presented in [25]. The approach combines static analysis to find a set of basic paths which are then tested. Its main advantage is that it can infer distributions on time, not only average values. In principle, both approaches could be adapted to infer energy usage.

## 7. CONCLUSIONS

We have proposed an approach for inferring parametric upper and lower bounds on the energy consumption of a program using a combination of static and dynamic techniques. The dynamic technique, based on an evolutionary algorithm, is used to determine the maximal / minimal energy consumption of each basic block. Such blocks contain multiple instructions, which allows this phase to take into account inter-instruction dependencies. Since such basic blocks are branchless, the evolutionary algorithm approach is more practical and efficient and the technique infers energy values that are accurate since no control flow-related variations occur. A static analysis is then used to combine the energy values obtained for the blocks according to the program control flow, and produce energy consumption bounds of the whole program. We also carried out an experimental evaluation to validate the upper and lower bounds on a set of benchmarks. The results support our hypothesis that the bounds inferred in this way are indeed safe and quite accurate, and the technique practical.

## 9. REFERENCES

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In R. D. Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[3] S. Chakravarty, Z. Zhao, and A. Gerstlauer. Automated, Retargetable Back-annotation for Host Compiled Performance and Power Modeling. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 36:1–36:10, USA, 2013. IEEE Press.

[4] 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*, pages 174–188. ACM Press, June 1990.

[5] 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*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[6] K. S. Henriksen and J. P. Gallagher. Abstract interpretation of PIC programs through logic programming. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 184–196. IEEE Computer Society, 2006.

[7] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012.

[8] C. Herrmann, A. Bonenfant, K. Hammond, S. Jost, H.-W. Loidl, and R. Pointon. Automatic Amortised Worst-Case Execution Time Analysis. In *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OASIcs*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007.

[9] R. Jayaseelan, T. Mitra, and X. Li. Estimating the Worst-Case Energy Consumption of Embedded Software. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*, pages 81–90. IEEE Computer Society, 2006.

[10] 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.

[11] S. Lafond and J. Lilius. Energy consumption analysis for two embedded Java virtual machines. *J. Syst. Archit.*, 53(5-6):328–337, 2007.

[12] 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.

[13] 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 *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.

[14] P. López-García, L. Darmawan, and F. Bueno. A Framework for Verification and Debugging of Resource Usage Properties. In M. Hermenegildo and T. Schaub, editors, *Technical Communications of the 26th Int'l. Conference on Logic Programming (ICLP'10)*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 104–113, Dagstuhl, Germany, July 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] P. Lopez-Garcia, L. Darmawan, F. Bueno, and M. Hermenegildo. Interval-Based Resource Usage Verification: Formalization and Prototype. In R. P. na, M. Eekelen, and O. Shkaravska, editors, *Foundational and Practical Aspects of Resource Analysis. Second Iternational Workshop FOPARA 2011, Revised Selected Papers*, volume 7177 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, 2012.

[16] P. Lopez-Garcia, R. Haemmerlé, M. Klemen, U. Liqat, and M. V. Hermenegildo. Towards Energy Consumption Verification via Static Analysis. In *Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2015), arXiv: 1501.03064*, 2015.

[17] D. May. The XMOS XS1 architecture. available online: http://www.xmos.com/published/xmos-xs1-architecture, 2013.

[18] E. Mera, P. López-García, M. Carro, and M. Hermenegildo. Towards Execution Time Estimation in Abstract Machine-Based Languages. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press, July 2008.

[19] H. Muller, editor. *Metrics and Case Studies*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013. Deliverable 6.1, http://entraproject.eu.

[20] 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)*, pages 29–32, April 2008. Extended Abstract.

[21] 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, pages 348–363. Springer, 2007.

[22] J. Pallister, S. Kerrison, J. Morse, and K. Eder. Data dependent energy modeling for worst case energy consumption analysis. *arXiv preprint arXiv:1505.03374*, 2015.

[23] M. Rosendahl. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 144–156. ACM Press, 1989.

[24] A. Serrano, P. Lopez-Garcia, and M. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, 14(4-5):739–754, 2014.

[25] S. A. Seshia and J. Kotker. Gametime: A toolkit for timing analysis of software. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 388–392. Springer, 2011.

[26] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *15th International Workshop on Implementation of Functional Languages (IFL'03), Revised Papers*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, Sep 2005.

[27] P. Wagemann, T. Distler, T. Honig, H. Janker, R. Kapitza, and W. Schroder-Preikschat. Worst-case energy consumption analysis for energy-constrained embedded systems. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 105–114, July 2015.

[28] D. Watt. *Programming XC on XMOS Devices*. XMOS Limited, 2009.

[29] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[30] 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. Embedded Comput. Syst.*, 7(3), 2008.

[31] XMOS. The XTAG-2 Hardware Manual, September 2009. Available online at: https://www.xmos.com/download/private/XTAG-2-Hardware-Manual

[32] XMOS. Use xTIMEcomposer and xSCOPE to trace data in real-time, 2013. Available online at: https://www.xmos.com/download/public/Trace-data-with-XScope(X9923H).pdf.