Inferring Energy Bounds via Static Program Analysis and Evolutionary Modeling of Basic Blocks

Umer Liqat[1,3] , Zorana Banković[1], Pedro Lopez-Garcia[1,2(✉)]   ,
and Manuel V. Hermenegildo[1,3]

[1] IMDEA Software Institute, Madrid, Spain
{umer.liqat,zorana.bankovic,pedro.lopez,manuel.hermenegildo}@imdea.org
[2] Spanish Council for Scientific Research (CSIC), Madrid, Spain
[3] Universidad Politécnica de Madrid, Madrid, Spain

Abstract. The ever increasing number and complexity of energy-bound devices (such as the ones used in *Internet of Things* applications, smart phones, and mission critical systems) pose an important challenge on techniques to optimize their energy consumption and to verify that they will perform their function within the available energy budget. In this work we address this challenge from the software point of view and propose a novel approach to estimating accurate parametric bounds on the energy consumed by program executions that are practical for their application to energy verification and optimization. Our approach divides a program into basic (branchless) blocks and performs a *best effort* mod-eling to estimate upper and lower bounds on the energy consumption for each block using an evolutionary algorithm. Then it combines the obtained values according to the program control flow, using a *safe* static analysis, to infer functions that give both upper and lower bounds on the energy consumption of the whole program and its procedures as functions on input data sizes. We have tested our approach on (C-like) embed-ded programs running on the XMOS hardware platform. However, our method is general enough to be applied to other microprocessor architec-tures and programming languages. The bounds obtained by our proto-type implementation on a set of benchmarks were always safe and quite accurate. This supports our hypothesis that our approach offers a good compromise between safety and accuracy, and can be applied in practice for energy verification and optimization.

1   Introduction

Reducing and controlling the energy consumption and the environmental impact of computing technologies has become a challenging problem worldwide.

It is a significant issue in systems ranging from large data centers and high-performance computing systems down to small *Internet of Things (IoT)* devices, sensors, smart watches, smart phones, portable/implantable medical devices, wearables, etc. Such devices often rely on small batteries or energy harvested from the environment, and may have to operate and intercommunicate continuously for long periods of time, 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 *IoT* and other energy-bound applications. In addition, for many of these IoT applications (as well as others, such as, e.g., space systems or implantable/portable medical devices), beyond optimizing energy consumption, it is actually crucial to guarantee that execution will complete within a specified energy budget, i.e., before the available system energy runs out, or that the system will function for a given period of time.

In spite of the recent rapid advances in energy-efficient hardware, it is software that controls the hardware, so that far more energy savings remain to be tapped by improving the software that runs on these devices. In this work we address these challenges from the software point of view, focusing on the *static* estimation of the energy consumed by program executions (i.e., at compile time, without actually running the programs with concrete data), 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 [14,15]. Unfortunately, current approaches that guarantee that the bounds are always safe tend to compromise their accuracy with overly conservative bounds, which may not be useful in practice. With this safety/accuracy trade-off in mind, we propose an approach that combines a *best effort* modeling with a *safe* static analysis, to infer accurate bounds that are on the safe side in most cases, in order to be practical for verification applications, in addition to energy optimization.

Describing how energy *verification* is performed is out of the scope of this paper, and we refer the reader to [13,14] for a description on how upper and lower bounds on resource usage in general can be used for verification within the CiaoPP system [6], and to [15] for a specialization to energy consumption verification. Herein we focus instead on the *inference* of energy bounds. Nevertheless, in the following we provide the intuition on how these bounds are used in our system for verification and certification: assume that E_l and E_u are (strict) safe lower and upper bounds (respectively) on the energy consumption of a program, and that E_b is an energy budget expressed by a program specification, e.g., defined by the capacity of the battery. Then:

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

2. If $E_l \leq E_b \leq E_u$, it might be possible to complete the execution of the program, but we cannot claim it for certain.

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

Of the small number of static energy analyses proposed to date, only a few [11,12,20] 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 information regarding 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 instruction-level models are [10], at the Java bytecode level, or [9], at the Instruction Set Architecture (ISA) level.

Clearly, the safety of the bounds inferred by analysis depends on the safety of the low-level models. Unfortunately, instruction-level models such as [9,10] provide *average* energy consumption values or functions, which are not really suitable for safe 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 supplied 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 [11,12,20] used instead the already mentioned instruction level average energy models [9,10]. 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 finding a good compromise and providing a *best effort* 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 by 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 ISA instructions as the modeling unit instead of individual instructions. We divide the (ISA) 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 an upper (resp. lower) bound on the energy consumption of each block. In this way the inter-instruction data dependence discussed above and other factors are accounted for within each block. The inter-instruction dependencies between blocks are still modeled in a conservative way, and hence can be one of the sources of inaccuracy. However, such modeling does not affect the safety of the energy bounds. The energy values obtained for each block are supplied to our

(safe) static resource analysis, which combines them according to the program control flow and produces functions that give both (practical) upper and lower bounds on the energy consumption of the whole program and its procedures, as functions on input data sizes.

In order to find bounds on the energy consumption of each basic block we use an evolutionary algorithm (EA), varying the basic block's input values and taking energy measurements directly from the hardware for each input combination. This way, we take advantage of the fast search space exploration provided by EAs. The approach in [22] also uses EAs for estimating worst case energy consumption. However, it is applied to *whole* programs, rather than at the basic block level. A major disadvantage of such an approach is that, if there are data-dependent branches in the programs, as is often the case, the EA quickly loses accuracy, and does not converge since different input combinations can trigger different sets of instructions [22]. This can make the problem intractable. In contrast, our approach combines EAs and static analysis techniques in order to get the best of both worlds. Our approach takes out the treatment of data-dependent branches from the EA, so that the same sequence of instructions is always executed in each basic block. This way, the EA converges and estimates the worst (resp. best) case energy of the basic blocks with higher accuracy. We take care of the program control flow dependencies by using static analysis instead.

For concreteness, in our experiments we focus on the energy analysis of programs written in XC [25], running on the XS1-L architecture [17], designed by XMOS.[1] However, our approach is general enough to be applied as well to the analysis of other architectures and other programming languages and their associated lower-level program representations. 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, 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].

The results of our experiments suggest that our *best effort* approach is quite accurate, in the sense that the inferred energy bounds are close to the actual upper and lower bounds. Furthermore, 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 highest energy consumptions and the inferred lower bounds below the actual lowest energy consumptions). We argue thus that our analysis provides a good practical compromise.

In summary, the main contributions of this paper are:
– A novel approach that combines dynamic and static analysis techniques for inferring more accurate upper and lower bounds on the energy consumption of program executions as functions of input data sizes. The dynamic part is based on EAs, and produces low-level energy models that contain *(best effort)*

---

[1] http://www.xmos.com/.

upper and lower bounds on the cost of the elementary operations, as opposed to just average values.

– The proposal of a new abstraction level at which to perform the energy modeling of program components, namely at the level of basic (branchless) blocks of ISA instructions, and a method based on EAs to dynamically obtain upper and lower bounds on the energy of such basic blocks with a good safety/accuracy compromise.
– A prototype implementation and experimental study that supports our claims.

In the rest of the paper, Sect. 2 explains our *best effort* technique for energy modeling of program basic blocks. Section 3 shows how these models are used by the safe static analysis to infer (practical) upper and lower bounds on the energy consumed by programs as functions of their input data sizes. Section 4 reports on an experimental evaluation of our approach. Related work is discussed in Sect. 5, and finally Sect. 6 summarizes our conclusions.

2 Modeling the Energy Consumption 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. We first identify all the basic blocks of the program, and then perform a *best effort* profiling of the energy consumption of each basic block for different input data using an EA. These steps are explained in the following sections.

2.1 Identifying 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, the program is first compiled to a lower-level representation, ISA in our case. A dataflow 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 their energy consumption 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*, *entsp*) are omitted from the block. The cost of such instructions is measured separately and added to the cost of the block or the function.

3. The harness function that runs the blocks in isolation provides the context to each block needed for the results to be applicable to the original program. For example the memory accesses in each block are transformed into accesses to a fixed address in the local memory of the harness function. The initial values placed in this local memory are the inputs to the block that the EA explores.
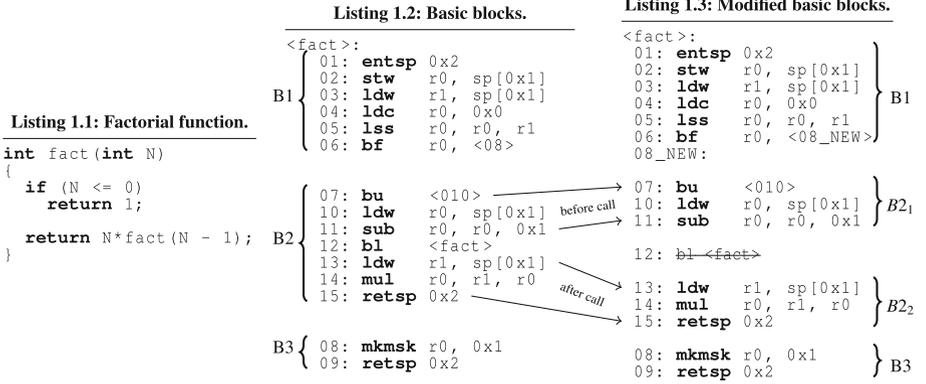
Listing 1.2: Basic blocks.

Listing 1.3: Modified basic blocks.

Listing 1.1: Factorial function.

```
int fact(int N)
{
  if (N <= 0)
    return 1;

  return N*fact(N - 1);
}
```

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

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

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

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

```
07: bu     <010>
10: ldw    r0, sp[0x1]
11: sub    r0, r0, 0x1
```
$B2_1$

```
12: bl <fact>
```

```
13: ldw    r1, sp[0x1]
14: mul    r0, r1, r0
15: retsp 0x2
```
$B2_2$

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

before call, *after call*

Fig. 1. Example: basic block modifications

An example of modifications 1 and 2 above is shown in Fig. 1. Listing 1.2 shows 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 1.3, such that the instructions before and after the function call form two blocks $B2_1$ and $B2_2$ respectively, and the call instruction (bl) is omitted. 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 $B2_1$, and $B2_2$ blocks, and the bl ISA instruction respectively, 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 (minimize) 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:

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

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 Listing 1.2 (Fig. 1), the input arguments are $gen(B1) = \{r0\}$, $gen(B2_1) = \{sp[0x1]\}$, $gen(B2_2) = \{sp[0x1],r0\}$, and $gen(B3) = \emptyset$.

2.2 Evolutionary Algorithm for Finding Energy Bounds for Basic Blocks

We now detail the main aspects of the EA used for approximating the upper-bound (i.e., worst case) and lower-bound (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 one 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 consumption of each block. The set of input variables to a block is inferred using a dataflow analysis (as explained in the previous section). Thus, an individual is simply an array of input values given in the order of their appearance in the block. In the initial population, the input values to an individual are randomly assigned to 32-bit numbers. In addition, some corner cases that are known to cause high or low energy consumption for particular instructions are included.[2]
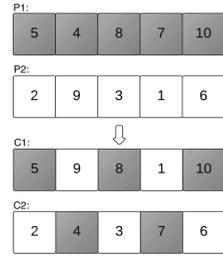
Fig. 2. Crossover

Crossover. The crossover operation is implemented as an even-odd crossover, since it provides more variability than a standard n-point crossover. The process is depicted in Fig. 2, where $P1$ and $P2$ are the parents, and $C1$ and $C2$ are their children created by the crossover operation.
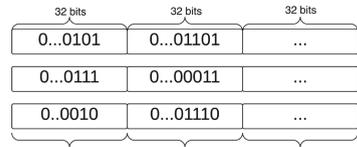
Fig. 3. Mutation

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 best way to explore the search space is by performing some bit flipping in the mutation operation. This is implemented as follows. For each gene component (i.e., for each input value to the basic block):

1. We create a random 32-bit integer (a random mask).
2. Then we perform the XOR operation of that integer and the corresponding gene. This results in a random flipping of the bits of each gene: only the bits of the gene at positions where the value of the random mask is 1 are flipped.

---

[2] For example, all 1s for high energy consumption, or all 0s for low energy consumption as operands to a multiply ISA instruction.

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

In the ISA representation of the program, the type structure is implicit and each operand (e.g., register) of an ISA instruction is a 32-bit value that either represents data or a memory address holding data. Since the input variables to a block holds data (memory accesses are transformed as described in the previous section), the mutation and crossover operators could generate data that such input variables would never take if the block were to run as part of the whole program. Thus, this conservative modeling of inter-block data dependencies could be one source of inaccuracy.

Objective function. The objective function that we want to maximize/minimize is the energy of a basic block, which is measured directly from the chip. The concrete measurement setting will be explained in Sect. 4.

The XMOS XS1 architecture used in our experiments is a *cache-less, by design-predictable architecture*, and it does not exhibit pipeline effects such as stalls (to resolve pipeline hazards), since exactly one instruction per thread is executed in a 4-stage pipeline. However, in general, pipeline effects, 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 a block, and have to be taken into account. Intra-block pipeline effects are accounted for by our approach just because the dependencies among the instructions within a block are captured. The inter-block pipeline effects could 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 to the execution time of the block). Similarly, for the lower bound estimation a zero stall penalty could be used. To approximate these effects, in [2], the authors characterize each block through pairwise executions with all of its possible predecessors. Each basic block pair is characterized by executing it on an Instruction Set Simulation (ISS) to collect cycle counts. A similar reasoning would apply to cache effects due to module boundaries. These effects could also be bounded using cache and pipeline analysis techniques [16].

3 Static Analysis of the Program Energy Consumption

Once (best effort) energy models are obtained for each basic block of the program, the energy consumption of the whole program is bounded by a (safe) static analyzer that takes into account the control flow of the program and infers (practical) upper/lower bounds on its energy consumption. We have implemented such an analyzer by specializing the generic resource analysis framework provided by CiaoPP [3, 4, 21, 23], for programs written in the XC programming language [25] and running on the XMOS XS1-L architecture. This includes the use of a transformation [11, 12] of the ISA code into an intermediate representation for analysis which is a series of connected code blocks, represented as Horn Clauses (HC IR). Such a transformation is shown in Fig. 4 where the ISA representation of the factorial function from Listing 1.2 (Fig. 1) is shown. It transforms the blocks into

clauses and instructions into clause literals. Conditional branching is modeled by predicates with two clauses, one with the condition true and the other false. The input/output arguments of each block are inferred via a dataflow analysis. The final step transforms the blocks into Static Single Assignment (SSA) form where each variable is assigned exactly once. The analyzer deals with this HC IR always in the same way, independently of where it originates from. We have also written the necessary code (i.e., assertions [7]) to feed such analyzer with the block-level upper/lower bound energy model obtained by using the technique explained in Sect. 2. The analyzer 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 automatically sets up a system of recurrence (cost) equations that capture the cost (energy consumption) of P as a function of the sizes of its input arguments $\bar{x}$. Typical metrics used for data sizes in this context are the actual value of a number, the length of a list or array, etc. [21,23].

```
1   <fact>:                          1   fact(R0,R0_3):-
2   01: entsp 0x2                    2     entsp(0x2),
3   02: stw   r0, sp[0x1]            3     stw(R0,Sp0x1),
4   03: ldw   r1, sp[0x1]            4     ldw(R1,Sp0x1),
5   04: ldc   r0, 0x0                5     ldc(R0_1,0x0),
6   05: lss   r0, r0, r1             6     lss(R0_2,R0_1,R1),
7   06: bf    r0, <008>             7a     bf(R0_2,0x8),
                                    7b     fact_aux(R0_2,Sp0x1,R0_3,R1_1).

                                    10  fact_aux(1,Sp0x1,R0_4,R1):-
11  07: bu    <010>                11     bu(0x0A),
12  10: ldw   r0, sp[0x1]          12     ldw(R0_1,Sp0x1),
13  11: sub   r0, r0, 0x1          13     sub(R0_2,R0_1,0x1),
14  12: bl    <fact>              14a     bl(fact),
                                  14b     fact(R0_2,R0_3),
16  13: ldw   r1, sp[0x1]          16     ldw(R1,Sp0x1),
17  14: mul   r0, r1, r0           17     mul(R0_4,R1,R0_3),
18  15: retsp 0x2                  18     retsp(0x2).

                                   20  fact_aux(0,Sp0x1,R0,R1):-
21  08: mkmsk r0, 0x1              21     mkmsk(R0,0x1),
22  09: retsp 0x2                  22     retsp(0x2).
```

Fig. 4. An ISA (factorial) program (left) and its Horn-clause representation (right)

Consider the example in Fig. 4 (right). The following cost equations are set up over the function *fact* that characterize the energy consumption of the whole function using the approximation A (e.g., upper/lower) of each block inferred by the EA, as a function of its input data size $R0$ (in this case the metric is the actual value of $R0$):

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

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

These inferred recurrence relations/equations are then passed on to a computer algebra system (e.g., CiaoPP's internal solver or an external solver such

as Mathematica, both used for the results presented in this paper) in order to obtain a closed form function for them. 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 $R0$ as: $fact_e(R0) = R0 + 1$.

The functions inferred by the static analysis are arithmetic (including polynomial, exponential, logarithmic, etc.), and their arguments (the input data sizes) are natural numbers. The generic resource analyzer ensures that the inferred bounds are strict/safe if it is supplied with energy models which provide safe bounds.

4 Experimental Assessment

The main goal of our experimental assessment is to perform a first comparison of the actual upper and lower bounds on energy consumption measured on the hardware against the respective bounds obtained by evaluating the functions inferred by our proposed approach (which depend on input data sizes), for each program considered and for a range of input data sizes. As mentioned before, the experiments have been performed with XC programs running on the XMOS XS1-L architecture [17]. The particular (development) hardware for which we derive the branchless-block-level model is a dual-tile board that contains an XS1-A16-128-FB217 processor.

In order to take power measurements during execution on real hardware, record and/or display them in real time, we use the hardware and software harness designed by XMOS, as an extension of the XMOS toolchain, which includes:

– A (hardware) debug adapter (xTAG v3.0) that enables power to be measured.[3] The basic principle consists in placing a small shunt resistor of R_{shunt} ohm in series within the supply line. By measuring the voltage drop on the shunt V_{shunt}, the current is calculated as $I_{shunt} = V_{shunt}/R_{shunt}$ (Ohm's law), which is also the current of the power supply $I_{sup} = I_{shunt}$. Then the power consumption is estimated as $V_{sup} \times I_{sup}$, where V_{sup} is the voltage of the power supply. The xTAG v3.0 adapter has an extra connector that carries the analog signals required 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.[4]

– A (software) tool (`xgdb`, the debugger), which collects data from the xTAG to be used by the analysis, by connecting to it over a USB interface (using libusb), and reading both ordinary xSCOPE traffic and voltage/current measurements.

The selected benchmarks, shown in Table 1, are either iterative or recursive. For conciseness, the first column only shows the names of the programs and

---

[3] https://www.xmos.com/download/private/xTAG-3-Hardware-Manual.pdf.
[4] https://www.xmos.com/download/private/Trace-data-with-XScope(X9923H).pdf.

Table 1. Accuracy of upper- and lower-bound estimations

| Program	DDBr	Upper/Lower bounds (nJ) $\times 10^3$	vs. HW
$fact(N)$	n	$ub = 5.1\ N + 4.2$	$+7\%$
		$lb = 4.1\ N + 3.8$	-11.7%
$fibonacci(N)$	n	$ub^a = 5.2\ lucas(N) + 6\ fib(N) - 6.6$	$+8.71\%$
		$lb = 4.5\ lucas(N) + 5\ fib(N) - 4.2$	-4.69%
$reverse(A)$	n	$ub = 3.7\ N + 13.3$ (N = length of array A)	$+8\%$
		$lb = 3\ N + 12.5$	-8.8%
$findMax(A)$	y	$ub = 5\ N + 6.9$ (N = length of array A)	$+8.7\%$
		$lb = 3.3\ N + 5.6$	-9.1%
$selectionSort(A)$	y	$ub = 30\ N^2 + 41.4\ N + 10$ (N = length of array A)	$+8.7\%$
		$lb = 16.8\ N^2 + 28.5\ N + 8$	-9.1%
$fir(N)$	y	$ub = 6\ N + 26.4$	$+8.9\%$
		$lb = 4.8\ N + 22.9$	-9.7%
$biquad(N)$	y	$ub = 29.6\ N + 10$	$+9.8\%$
		$lb = 23.5\ N + 9$	-11.9%

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

the arguments that are relevant for their energy-bound functions. The first two benchmarks are small arithmetic programs, and the third one reverses elements of an input array A of size N ($reverse(A)$). A sorting algorithm ($selectionsort$) and a simple program for finding the maximum number in an array ($findMax$) are also included. The latter, which is also part of the former, is a program where data-dependent branching can bring significant variations in the worst- and best-case energy consumption for a given input data size. We have also studied two audio signal processing benchmarks, $biquad$ and fir (Finite Impulse Response), provided by XMOS as representatives of XS1 application kernels. Both programs perform filtering tasks that attenuate or amplify specific frequency ranges of a given input signal. The **DDBr** column expresses whether a benchmark has data-dependent branching or not (y/n). The third column shows the upper- and lower-bound energy functions (on input data sizes) estimated by our approach, as well as the size metric used. When an input argument (in the first column) is numeric, its size metric is its actual value (and is omitted in the third column). Column **vs. HW** shows the average deviation of the energy estimations obtained by evaluating such functions, with respect to the actual bounds measured on the hardware as explained above. A deviation is positive (resp. negative) if the estimated value is over (resp. under) the actual measurement. For a given input data size n the actual upper and lower bounds measured on the hardware are obtained by using data of size n that exhibit the worst and best cases respectively.
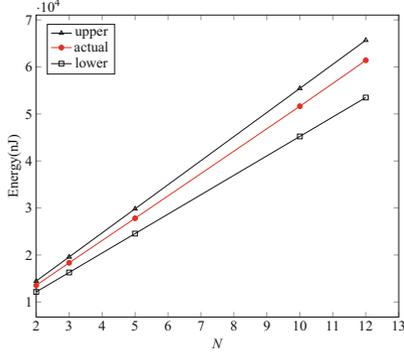
(a) $fact$
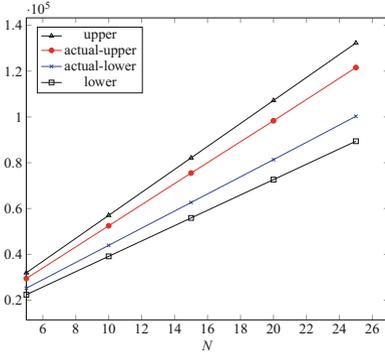
(b) $findMax$
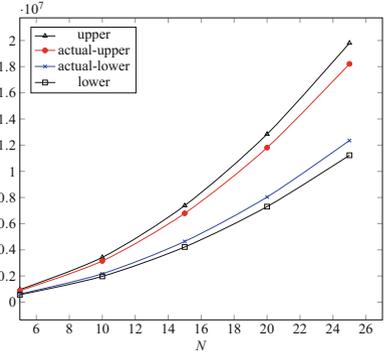
(c) $selectionsort$

Fig. 5. Estimated energy upper/lower bounds vs. actual measurements (Color figure online)

Figure 5(a) depicts the upper- and lower-bound energy functions estimated by the analysis, as well as the actual bounds measured on the hardware for the $fact(N)$ program (taking different values of N). In this case, both the actual upper- and lower-bounds coincide, as shown by the middle curve (in red), which plots the actual measurements on the hardware. It can be observed that the values of the upper-bound function estimated by the static analysis supplied with the model obtained by the EA always over-approximate the actual hardware measurements (by 7%, as given by Table 1), whereas the lower-bound values under-approximate them (by 11.7%).

Similarly, the $findMax$ benchmark is shown in Fig. 5(b). Unlike $fact$, the actual upper- and lower-bound functions of $findMax$, depending on input arrays of length N, do not coincide, due to the data-dependent branching. The actual energy consumption of $findMax$ not only depends on the length of the input

array, but also on its contents, and thus cannot be captured exactly by a function that depends on data sizes only (i.e., by abstracting the data by their sizes). A call to $findMax$ with a sorted array in ascending order (of a given length N) will discover a new max element in each iteration, and hence update the current max variable, resulting in the actual upper bound (i.e., worst case of the algorithm). In contrast, if the array is sorted in descending order, 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 lower bound (i.e., best case). Thus, Fig. 5(b) depicts four curves: the upper- and lower-bound energy functions estimated by our approach for $findMax$, as well as the two actual energy bound curves measured on the hardware. The former are obtained by evaluating the energy functions in Table 1 for different array-lengths N, as before. The latter are obtained with actual arrays of length N that give the worst and best cases, as explained above. Note that it is not always trivial to find data that exhibit program worst and best case behaviors. Table 1 shows that the estimated upper- (resp. lower-) bounds over- (resp. under-) approximate the actual upper- (resp. lower-) bounds measured on the hardware by 8.7% (resp. 9.1%). Figure 5(c) for *selectionsort* shows a similar behavior but with quadratic bounds.

The inaccuracies in the energy estimations of our technique come mainly from two sources: the modeling, which assigns an energy value to each basic block as described in Sect. 2, and the static analysis, described in Sect. 3, which estimates the number of times that the basic blocks are executed depending on the input data sizes, and hence, the energy consumption of the whole program. Table 2 shows part of the results of our study in order to quantify the inaccuracy originating from those sources. Different executions of the $findMax$ benchmark are shown for different input arrays of length **N** (Column **N**). The table is divided into two parts. The first part uses randomly generated input arrays of length **N**, while the second part (three lower rows) uses input arrays that cause the worst- and best-case energy consumption. Column **Cost App** indicates the type of approximation of the automatically inferred energy functions: upper bound (**ub**) and lower bound (**lb**). Such energy functions are shown in Table 1. We have then compared the energy consumption estimations obtained by evaluating the energy function (Column **Est**) with the observed energy consumption 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:

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

Column **Prof** shows the result of estimating the energy consumption using the energy model and assuming that the static analysis 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 the concrete input arrays, 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 values provided by the energy model for each basic block, and adding all of them. Col-

Table 2. Source of inaccuracies in $findMax$: analysis vs. modeling

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

umn **PrD%** represents the inaccuracy due to the energy modeling of basic blocks using the EA, which has been quantified as the relative harmonic difference between **Prof** and the observed energy consumption **Obs**. The difference between **D%** and **PrD%** represents the inaccuracy due to the static analysis.

Although the first part of the table, using random data, may give the impression that both the static analysis and the energy modeling contribute to the inaccuracy of the energy estimation of the whole program, the second (lower) part of the table indicates that the inaccuracy only comes from the energy modeling. This is because in the lower part the comparison was performed with input arrays that make $findMax$ exhibit its actual upper- and lower-bounds (depending on the length of the array). In this case, Columns **Est** and **Prof** show the same values, which means that there was no inaccuracy due to the static analysis (regarding the inference of the *actual upper- and lower-bound functions*), and that the overall inaccuracy is due to the over- and under-approximation in the EA to model energy consumption of each basic block.

Table 3 shows a similar experiment for the *reverse* program, which has no data-dependent branching. Since the number of operations performed by *reverse* is actually a function of the length of its input array (not of its contents), Columns **Est** and **Prof** show the same values for random data (unlike for $findMax$), which means that no inaccuracy comes from the static analysis part.

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

Table 3. Source of inaccuracies in *reverse*: analysis vs. modeling

N	Cost	Energy(nJ) $\times 10^3$			D%	PrD%
	App	Est	Prof	Obs		
Random array data						
5	lb	28	28	29	−3.5	−3.5
	ub	31.8	31.8		9.2	9.2
15	lb	59	59	64	−8.1	−8.1
	ub	68.8	68.8		7.2	7.2
25	lb	90	90	98	−8.5	−8.5
	ub	105.8	105.8		7.7	7.7

all the experiments the *biquad* benchmark took the most time (a maximum time of 230 min) for maximizing the energy consumption. In contrast, the *fact* benchmark took the least time (a maximum time of 121 min). The times remained within the 150–200 min range on average. Time speed-ups 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 s on average, despite the naive implementation of the interface with external recurrence equation solvers, which can be improved significantly.

5   Related Work

Static analysis of the energy consumed by program executions has received relatively little attention until recently. An analysis of Java bytecode programs that estimated 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 [18], and a simple energy model at the Java bytecode level [10] was used. However the energy model for the Java opcodes used *average* estimations, which are not suitable for verification applications. Also, the results were not compared with actual, measured energy consumption. A similar approach was proposed in [12] for XC programs, using an ISA-level model. This work did compare to actual energy consumptions, obtaining promising results, but the ISA-level model also provided average energy values, which implied the same problem for verification.

Other approaches to static analysis based on the transformation of the analyzed code into another representation have been proposed for analyzing low-level languages [5] and Java bytecode [1]. In [1], cost relations are inferred directly for these bytecode programs, whereas in [20] the bytecode is first transformed into Horn Clauses [18].

Other work is based on 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., [26] 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 [8]. It distinguishes instruction-specific (proportional to data) from pipeline-specific (proportional to time) energy consumption, and also takes into account branch prediction and cache misses. However, it requires the user to identify the input which will trigger the maximal energy consumption. In [24] the approach is refined for estimating *hard* (i.e., over-approximated) energy bounds using *relative* energy models (at the LLVM level), where the energy of instructions is given *in relation to each other* (e.g., if all instructions have relative energy 1, then they all consume the same absolute energy), which does not depend on the specific hardware, but can be applied if a mapping between LLVM and low-level ISA instructions exists. If the energy bounds are not *hard* (i.e., the application allows their violation) a genetic algorithm is used 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 sets 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, which are required to not have any data-dependent branching. The authors further introduce probability distributions for the transition costs among pairs of independent instructions, which can 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 [19], 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). Loop bound inference techniques can also be applied but require that all loop counts can be resolved. All of this essentially reduces the case to that of programs with no loops.

6   Conclusions

We have proposed a combined static/dynamic approach for estimating parametric upper and lower bounds on the energy consumption of a program.

The dynamic part, based on an evolutionary algorithm, is a *best effort* approach to approximating the maximum/minimum energy consumption of the basic blocks in the program. Such blocks contain multiple instructions, which allows this phase to capture inter-instruction dependencies. Moreover, the basic blocks are branchless, which makes the evolutionary algorithm approach quite practical and efficient, and the energy values inferred by it are accurate, since no control flow-related variations occur. A safe static analysis is then used to combine the energy values obtained for the blocks according to the program control flow, and estimate energy consumption bounds of the whole program that depend on input data sizes. In the experiments we performed on a set of benchmarks, the upper and lower bounds obtained were always safe and quite accurate. Such results suggest that our approach offers a good safety/accuracy compromise, estimating bounds that are practical for its application to energy verification and optimization.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of Java bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_12
2. Chakravarty, S., Zhao, Z., Gerstlauer, A.: Automated, retargetable back-annotation for host compiled performance and power modeling. In: Proceedings of CODES+ISSS 2013, pp. 36:1–36:10. IEEE Press, USA (2013). http://dl.acm.org/citation.cfm?id=2555692.2555728
3. Debray, S.K., Lin, N.W., Hermenegildo, M.V.: Task granularity analysis in logic programs. In: Proceedings of PLDI 1990, pp. 174–188. ACM, June 1990
4. Debray, S.K., López-García, P., Hermenegildo, M.V., Lin, N.W.: Lower bound cost estimation for logic programs. In: ILPS 1997, pp. 291–305. MIT Press (1997)
5. Henriksen, K.S., Gallagher, J.P.: Abstract interpretation of PIC programs through logic programming. In: Proceedings of SCAM 2006, pp. 184–196. IEEE Computer Society (2006)
6. Hermenegildo, M., Puebla, G., Bueno, F., García, P.L.: Integrated program debugging, verification, and optimization using abstract interpretation (and The Ciao System Preprocessor). Sci. Comput. Program. **58**(1–2), 115–140 (2005)
7. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., Puebla, G.: An overview of Ciao and its design philosophy. TPLP **12**(1–2), 219–252 (2012). http://arxiv.org/abs/1102.5497
8. Jayaseelan, R., Mitra, T., Li, X.: Estimating the worst-case energy consumption of embedded software. In: Proceedings of IEEE RTAS, pp. 81–90. IEEE Computer Society (2006). http://dx.doi.org/10.1109/RTAS.2006.17
9. Kerrison, S., Eder, K.: Energy modeling of software for a hardware multithreaded embedded microprocessor. ACM TECS **14**(3), 1–25 (2015). http://doi.acm.org/10.1145/2700104

10. Lafond, S., Lilius, J.: Energy consumption analysis for two embedded Java virtual machines. J. Syst. Archit. **53**(5–6), 328–337 (2007)
11. Liqat, U., Georgiou, K., Kerrison, S., Lopez-Garcia, P., Hermenegildo, M.V., Gallagher, J.P., Eder, K.: Inferring parametric energy consumption functions at different software levels: ISA vs. LLVM IR. In: van Eekelen, M., Dal Lago, U. (eds.) Proceedings of FOPARA. LNCS, vol. 9964, pp. 81–100. Springer, Heidelberg (2016)
12. Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M.V., Eder, K.: Energy consumption analysis of programs based on XMOS ISA-level models. In: Gupta, G., Peña, R. (eds.) LOPSTR 2013. LNCS, vol. 8901, pp. 72–90. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14125-1_5
13. López-García, P., Darmawan, L., Bueno, F.: A framework for verification and debugging of resource usage properties. In: Technical Communications of ICLP. LIPIcs, vol. 7, pp. 104–113. Schloss Dagstuhl (July 2010)
14. Lopez-Garcia, P., Darmawan, L., Bueno, F., Hermenegildo, M.: Interval-based resource usage verification: formalization and prototype. In: Peña, R., van Eekelen, M., Shkaravska, O. (eds.) FOPARA 2011. LNCS, vol. 7177, pp. 54–71. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32495-6_4
15. Lopez-Garcia, P., Haemmerlé, R., Klemen, M., Liqat, U., Hermenegildo, M.V.: Towards energy consumption verification via static analysis. In: WS on High Performance Energy Efficient Embedded Systems (HIP3ES) (2015)
16. Lv, M., Guan, N., Reineke, J., Wilhelm, R., Yi, W.: A survey on static cache analysis for real-time systems. LITES **3**(1), 5:1–5:48 (2016)
17. May, D.: The XMOS XS1 architecture (2013). http://www.xmos.com/published/xmos-xs1-architecture
18. Méndez-Lojo, M., Navas, J., Hermenegildo, M.V.: A flexible, (C)LP-based approach to the analysis of object-oriented programs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 154–168. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78769-3_11
19. Mera, E., López-García, P., Carro, M., Hermenegildo, M.V.: Towards execution time estimation in abstract machine-based languages. In: PPDP 2008, pp. 174–184. ACM Press, July 2008
20. Navas, J., Méndez-Lojo, M., Hermenegildo, M.: Safe upper-bounds inference of energy consumption for Java bytecode applications. In: NASA LFM 2008, pp. 29–32, April 2008
21. Navas, J., Mera, E., López-García, P., Hermenegildo, M.V.: User-definable resource bounds analysis for logic programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 348–363. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74610-2_24
22. Pallister, J., Kerrison, S., Morse, J., Eder, K.: Data dependent energy modeling for worst case energy consumption analysis. Technical report, May 2015. http://arxiv.org/abs/1505.03374
23. Serrano, A., Lopez-Garcia, P., Hermenegildo, M.V.: Resource usage analysis of logic programs via abstract interpretation using sized types. TPLP **14**(4–5), 739–754 (2014). ICLP 2014 Special Issue
24. Wagemann, P., Distler, T., Honig, T., Janker, H., Kapitza, R., Schroder-Preikschat, W.: Worst-case energy consumption analysis for energy-constrained embedded systems. In: 2015 27th Euromicro Conference on Real-Time Systems (ECRTS), pp. 105–114, July 2015

25. Watt, D.: Programming XC on XMOS devices. XMOS Limited (2009). http://books.google.co.uk/books?id=81klKQEACAAJ
26. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. **7**(3), 1–53 (2008)