Energy Consumption Analysis of Programs based on XMOS ISA-Level Models

U. Liqat[1], S. Kerrison[2], A. Serrano[1], K. Georgiou[2], P. Lopez-Garcia[1,3], N. Grech[2], M.V. Hermenegildo[1,4], and K. Eder[2]

[1] IMDEA Software Institute
{umer.liqat,alejandro.serrano,
pedro.lopez,manuel.hermenegildo}@imdea.org
[2] University of Bristol
{steve.kerrison,kyriakos.georgiou,
n.grech,kerstin.eder}@bristol.ac.uk
[3] Spanish Council for Scientific Research (CSIC)
[4] Universidad Politécnica de Madrid (UPM)

Abstract. Energy consumption analysis of embedded programs requires the analysis of low-level program representations. This is challenging because the gap between the high-level program structure and the low-level energy models needs to be bridged. Here, we describe techniques for recreating the structure of low-level programs and transforming these into Horn clauses in order to make use of a generic resource analysis framework (CiaoPP). Our analysis, which makes use of an energy model we produce for the underlying hardware, characterises the energy consumption of the program, and returns energy formulae parametrised by the size of the input data. We have performed an initial experimental assessment and obtained encouraging results when comparing the statically inferred formulae to direct energy measurements from the hardware running a set of benchmarks. Static energy estimation has applications in program optimisation and enables more energy-awareness in software development.

Keywords: energy consumption analysis, energy models, resource usage analysis, static analysis.

1 Introduction

Energy consumption and the environmental impact of computing technologies are a major focus. Despite advances in power-efficient hardware, more energy savings can be achieved by improving the way current software technologies make use of such hardware. Many optimization techniques that can be used for producing energy-efficient software need estimations of the energy consumption of software segments prior to their execution, in order to make decisions about the optimal way of executing them. These a priori estimations are also very useful to software engineers to better understand the effect of their designs on the energy consumption early on during the software development process, and make

more informed design decisions (e.g., using the appropriate data structures), even when there are parts not developed yet.

In this paper we combine static analysis and *low level* energy modelling techniques to implement a tool capable of estimating the energy consumption of an embedded program (and its constituent parts, such as procedures and functions) as a function on several parameters of the input data (e.g., sizes), and the hardware platform where they are executed (e.g., clock frequency and voltage). We show the feasibility of our proposal with a concrete case study: analysis of ISA (Instruction Set Architecture) code compiled from XC [24]. XC is a high-level C-based programming language that includes extensions for concurrency, communication, input/output operations, and real-time behaviour. XC libraries share a common API with standard C libraries and therefore C code can commingle with XC code in a single application.

Since energy consumption analysis depends on the underlying hardware, the analyser requires information expressing the effect of the execution of a software segment (e.g., an assembly instruction) on the hardware. Such information is represented using *models*. In our approach these models express information using assertions. These are propagated during the static analysis process in order to infer information for higher-level entities such as functions. For instance, using assertions we abstract the operations in the language in terms of their effect on the size of the runtime data and the energy exerted. Energy models at lower levels (e.g., at the ISA level) are more precise than at higher levels (e.g., XC source code), since the closer to the hardware, the easier it is to determine the effect of the execution of the program on the hardware. For this reason, we have produced models for the ISA level, which we use when analysing ISA code generated by the XCC compiler.

Our approach leverages the CiaoPP tool [6], the preprocessor of the Ciao programming environment [7]. CiaoPP includes a generic, parametric analysis framework for resource usage that can be instantiated to infer bounds on resources of interest (energy consumption in our case), for different languages [14]. In CiaoPP, a resource is a user-defined *counter* representing a (numerical) non-functional global property, such as execution time, execution steps, number of bits sent or received by an application over a socket, etc. The CiaoPP resource analysis can infer upper and lower bounds on the usage made of such resources by programs by working on an intermediate block-based representation, the Horn clause (HC) IR. In this representation, each block is written as a *Horn clause*, i.e., a head followed by a sequence of primitive operations or calls to other blocks. Assertions describe the resources to be analyzed. We propose a transformation of the ISA program into this HC IR (containing Horn clauses and assertions), which allows us to analyse the transformed program with CiaoPP. The control and data flow encoded through the procedural interpretation of these Horn-clause programs, coupled with the resource-related information contained in the assertions (such as the energy consumption models at the ISA level), allow the resource analysis to infer static bounds on the energy consumption of the blocks that are directly applicable to the original ISA programs.

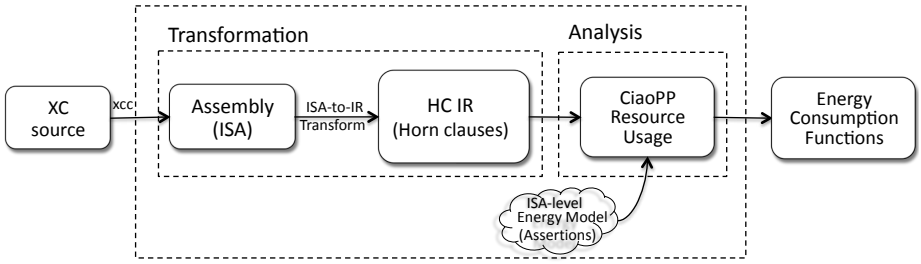


Fig. 1. Overview of the analysis framework for XC programs.

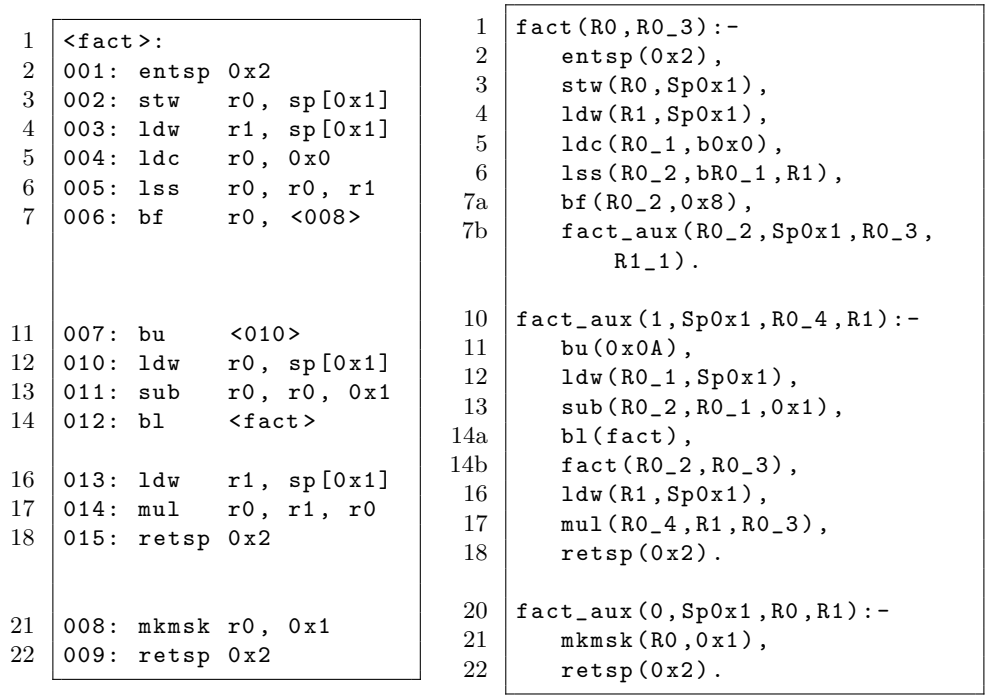
```
int fact(int N) {
  if (N <= 0) return 1;
  return N * fact(N - 1);
}
```

Fig. 2. An XC source (factorial) function.

Figure 1 shows the main steps of our approach for energy consumption analysis, which starts with an XC program (e.g., the `fact` function in Figure 2). The ISA program corresponding to it is generated using the XC compiler tool XCC (left hand side of Figure 3). The resulting ISA program is passed to a translator which generates the associated Horn clauses (right hand side of Figure 3). Such program, together with the information contained in the energy models at the ISA level (represented using the mentioned assertion language), is passed to the resource analysis which outputs the energy consumption for all procedures in the HC IR program. In our example, the resource analysis infers an estimation of the energy consumed by a call to `fact` as $(26.0\ N + 19.4)$ nano-Joules. This is parametric with N, the input argument to `fact`.

In this work we have successfully bridged the gap between researchers closer to the hardware area, needed to produce the low level energy models, and others from software, with expertise in static analysis techniques and tools. In this multidisciplinary research, we have faced some challenges and produced some original contributions that we describe in this paper and summarise as follows:

1. Development of an energy model for a multi-threaded architecture (XMOS XS1-L), that can be applied at instruction set simulation level or higher, with specialisation for high-level, single-threaded benchmarks.
2. Design and implementation of a translation from ISA programs into a Horn-clause representation (HC IR).
3. Instantiation of the CiaoPP general resource analysis framework to infer energy consumption using the low-level energy consumption model.
4. Overall design and implementation of a fully automatic system that statically estimates the energy consumption of functions and procedures written in a high-level, C-based programming language, giving the results as functions on input data sizes.
5. Experimental assessment of the developed energy usage static analyser.



```
1   <fact>:
2   001: entsp 0x2
3   002: stw   r0, sp[0x1]
4   003: ldw   r1, sp[0x1]
5   004: ldc   r0, 0x0
6   005: lss   r0, r0, r1
7   006: bf    r0, <008>



11  007: bu    <010>
12  010: ldw   r0, sp[0x1]
13  011: sub   r0, r0, 0x1
14  012: bl    <fact>

16  013: ldw   r1, sp[0x1]
17  014: mul   r0, r1, r0
18  015: retsp 0x2


21  008: mkmsk r0, 0x1
22  009: retsp 0x2
```

```
1   fact(R0,R0_3):-
2      entsp(0x2),
3      stw(R0,Sp0x1),
4      ldw(R1,Sp0x1),
5      ldc(R0_1,b0x0),
6      lss(R0_2,bR0_1,R1),
7a     bf(R0_2,0x8),
7b     fact_aux(R0_2,Sp0x1,R0_3,
           R1_1).

10  fact_aux(1,Sp0x1,R0_4,R1):-
11     bu(0x0A),
12     ldw(R0_1,Sp0x1),
13     sub(R0_2,R0_1,0x1),
14a    bl(fact),
14b    fact(R0_2,R0_3),
16     ldw(R1,Sp0x1),
17     mul(R0_4,R1,R0_3),
18     retsp(0x2).

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

Fig. 3. An ISA (factorial) program (left) and its Horn-clause representation (right).

Point 4 above may look simple at first sight, given that we have taken advantage of a number of existing tools, mainly the CiaoPP general resource analyser. However, in practice the implementation has required the development of a significant number of new modules and functionalities, as well as interfaces between these existing tools, all of which posed substantial design and implementation challenges and problems that we have successfully solved.

In the rest of the paper, energy characterisation and modelling for our case study architecture (XMOS XS1-L) is explained in Section 2. Then, Section 3 describes the translation from ISA programs into Horn clauses and Section 4 the instantiation of the CiaoPP general resource usage analysis framework. In Section 5, we have performed an experimental assessment of our approach, showing that the estimation of energy consumption is reasonably accurate. Section 6 comments on related work. Finally, Section 7 summarises our conclusions and comments on ongoing and future work.

2 Energy Characterization and Modelling

The assertion-based model uses power consumption data collected during hardware measurement. We have developed an ISA-level model that provides software energy consumption estimates based on Instruction Set Simulation (ISS) statistics. The hardware, the measurement process, as well as the construction of the

ISS-driven model, are detailed in [10], with the key components relevant to this paper explained in the rest of this section.

The practicality and accuracy of our approach to energy consumption analysis relies on a good characterisation of energy consumption and generating good energy consumption models. A trade-off needs to be found between the simplicity of the models, which improves the efficiency of the analysis, and the accuracy of the models, which improves the accuracy of the global analysis. Although we analyse single-threaded code, the energy profiling must consider the hardware multi-threading of the architecture, which has an energy impact even when only a single thread is executed.

Further, the nature of the architecture requires specific approaches in order to gather energy profiling data, but these same characteristics preclude certain energy effects from static analysis. For example, the effects of interleaving instructions or re-use of operands from the previous instruction become less relevant in a hardware multi-threaded pipeline, and impossible to determine statically. Although manifested in a specific way in this particular processor architecture, such traits also exist in other processors, such as super-scalar designs. In this paper we describe an initial proposal that offers a good compromise between the above issues, and also eliminates factors that are determined to be insignificant.

2.1 Energy Profiling Framework and Strategy

An energy profiling framework, `xmprofile`, is used to generate sequences of instructions under various constraints in order to profile the energy characteristics of the hardware. This data is essential for the accurate application of models at any analysis level. The hardware used is shown in Figure 4. A master processor issues test programs to and measures the power used by a slave processor, the Device Under Test (DUT).

Currently, a subset of the ISA, including arithmetic operations, logic operations, and condition tests, has been characterised. Other instructions are at the moment approximated using a single average value, based on typical observed behaviour.

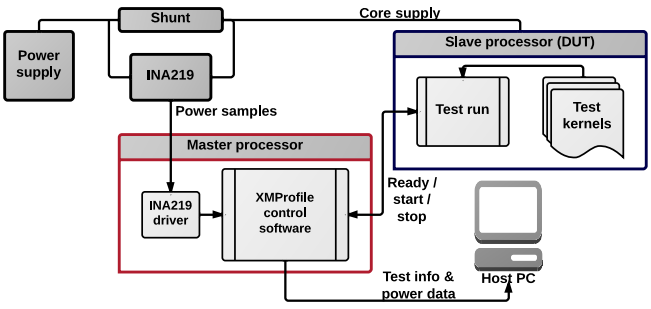


Fig. 4. Overview of test harness hardware and software structure, with a slave processor executing test kernels and a master processor collecting power samples.

2.2 ISA-level Model

An ISA-level model, `xmmodel`, gives an energy estimate for a program based on ISS output. Data from the measurement framework feeds this model.

Our model is based on that devised by Tiwari [22]. Tiwari's approach is shown in Equation (1). The energy of an ISA program, E_p, is characterised as the sum of base energy cost, B_i, for all ISA instructions, i, multiplied by the number of executions of each instruction, N_i. An inter-instruction overhead energy, $O_{i,j}$, is then accounted for by enumerating for all instruction combinations i, j and their frequency, $N_{i,j}$. Finally, additional contributions to program energy can be accounted for by k external effects, E_k, which may include externally modelled behaviours such as cache memory.

$$E_p = \sum_{i \in \text{ISA}} (B_i \times N_i) + \sum_{i,j \in \text{ISA}} (O_{i,j} \times N_{i,j}) + \sum_{k \in \text{ext}} E_k \tag{1}$$

The XS1 architecture is hardware multi-threaded. This necessitates a fundamental revision of the model equation. In addition, for performance reasons, the ISS collects instruction statistics rather than a full trace. This reduces the execution time by an order of magnitude, such that it is approximately 100 times slower than the hardware when simulation is run on a modern computer.

Equation (2) describes the energy of a program, E_p, using a similar method to Equation (1), but with several key differences. Time is an explicit component, multiplied by power terms in order to calculate energy. This separation enables future exploration of idle periods, external event timing, and variable operating frequencies. Inter-instruction overhead is represented as a single component, rather than considering it for all possible pairs of instructions, on account of a statistics-based approach rather than cycle-by-cycle instruction tracing. Finally, the level of concurrency must be accounted for, something that was not necessary for the architecture targeted by Equation (1). The concurrency level is the number of threads that are active at a given time. In the case of the XS1-L, the concurrency level represents how full the pipeline is and therefore how much activity is generated within it as each stage switches between instructions from the active threads.

$$E_\text{p} = P_\text{base} N_\text{idle} T_\text{clk} + \sum_{t=1}^{N_t} \sum_{i \in \text{ISA}} \left((M_t P_i O + P_\text{base}) \, N_{i,t} T_\text{clk} \right) \tag{2}$$

The base power, P_base, is present in both active and idle periods. The number of idle periods, N_idle, is counted and multiplied by the clock period, T_{clk}, to account for the energy consumed when no threads are active. For each number of concurrent threads, t, (based on the proportion of time each thread is active), and for each instruction, i, in the ISA, the instruction power, P_i, is multiplied by a constant inter-instruction power overhead, O, and a concurrency cost for the level of concurrency at which the processor is operating, M_t. These are all multiplied by the number of times this instruction occurs at this concurrency level, $N_{i,t}$, and the clock period. Combined with the idle energy, this gives a total energy estimate for the program run.

In the case where a single thread is running, with no idle periods, then the above can be simplified to Equation (3). The result is very similar to the single-threaded Tiwari equation, but with only a single, generic inter-instruction power overhead component, O, and with no external "k" components as the memory of the XS1-L is single-cycle with no cache, with no other effects that need to be considered at this point. There is only ever one active thread, so we use the concurrency cost for one thread, M_1. Again, in Equation (3), time is an explicit component. The overhead, O, is a constant because the inter-instruction effect cannot be known statically in the XS1 architecture, and during profiling the variation in inter-instruction effect was shown to be an order of magnitude less than the instruction cost and would average out over program runs.

$$E_{\mathrm{p}} = \sum_{i \in \mathrm{ISA}} ((M_1 P_i O + P_{\mathrm{base}}) \times (N_i T_{\mathrm{clk}})) \tag{3}$$

Our ISS-based model, using the same energy data as the static analysis, will be used as an additional comparison point between actual hardware energy measurements and the static analysis results.

3 Transforming ISA Programs into Horn Clauses

In this section we describe the transformation from ISA programs into Horn clauses (HC IR) mentioned in Section 1, which is used for analysis. Such representation consists of a sequence of *blocks* (as in the right hand side of Figure 3). Each block is represented as a *Horn clause*:

$$< block\_id > (< params >) :- \ S_1, \ \ldots \ , S_n.$$

which has an entry point, that we call the *head* of the block (to the left of the $:-$ symbol), including a number of parameters $< params >$, and a sequence of steps (the *body*, to the right of the $:-$ symbol), each of which is either, (the representation of) an ISA *instruction*, or a *call* to another (or the same) block. The analyser deals with the HC IR always in the same way, independently of its origin. The transformation ensures that the program information relevant to resource usage is preserved, so that the energy consumption functions of the HC IR programs inferred by the resource analysis are applicable to the original ISA programs.

ISA programs are expressed using the XS1 instruction set [13]. The transformation framework currently works on a subset of this instruction set. The ISA program is parsed and a control flow analysis is carried out, yielding an inter-procedural control flow graph (CFG). This process starts by identifying control transfer instructions such as branch or call instructions. Basic blocks are then constructed, which are annotated with input/output arguments and transformed into Static Single Assignment (SSA) form. Finally, the target HC IR (i.e., Horn clauses) is emitted.

A basic block over a 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. All call instructions are assumed to eventually return. Using the basic block definition a block control flow graph is constructed by the analyser, where each node represents a block. Edges between the blocks are derived from calls/jumps between blocks. This process involves iterating through the CFG of the ISA program and marking block boundaries, which are instructions that either begin or end a basic block.

Inferring Block Input/Output Parameters. In order to treat each block as a Horn clause, the block's input and output arguments need to be inferred. For the entry block, the input and output arguments are derived from the original function's signature. We define the functions $params_{in}$ and $params_{out}$, which infer input and output parameters of a block respectively. These perform a backwards analysis of the program, and are recomputed until a least fixpoint is reached on these functions.

$$params_{out}(b) = kill(b) \cup \bigcup_{b' \in next(b)} params_{out}(b')$$
$$params_{in}(b) = gen(b) \cup \bigcup_{b' \in next(b)} params_{in}(b')$$

where $next(b)$ denotes the set of immediate target blocks that can be reached from b with a call or jump, while $gen(k)$ and $kill(k)$ are the read and written variables in a block respectively, which we define as:

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

and $def(k)$ and $ref(k)$ denote the variables written or referred to at a node in the block respectively.

Our approach here is closely related to that of the live variable analysis (LVA) [18] used in compilers, and in dead code elimination in particular. A variable is live at a program point if it may get referenced later in the program (which is decided by considering the whole CFG of the program). In LVA, for each program point, a set of live variables is computed using functions similar to our *kill* and *gen* functions with data flow equations. In our approach however, instead of computing liveness information for each program point, we compute a least fixpoint of our $params_{out}$ and $params_{in}$ functions over the program's block control flow graph. This is an efficient solution that safely over-approximates the set of input/output arguments to each block, so that the extra arguments inferred for block heads due to such over-approximation do not affect the energy consumption estimations, since they are not used in the analysis of procedures corresponding to the original XC code.

Resolving Branching to Multiple Blocks. In the XS1 instruction set, conditional branch instructions (e.g., `bt`, `bf`) jump to one of the two target blocks based on the value of the branching variable. For example, in Figure 3, at line `7` the `bf` instruction (branch if fail) will jump to address `008` if $r0 = 0$, otherwise

to address `007`. In the HC IR this branch needs to be a call to one of the two blocks.

We use a similar approach to the one described in [14] to resolve branches to multiple blocks. The multiple target blocks of a jump instruction are assigned the same head, which essentially are clauses of the same HC IR predicate. This is achieved by merging the heads of the target clauses so that each clause has the same head. The algorithm is trivial, since we have already inferred the input/output parameters to each block's head. The input/output parameters to the new head of the clauses are the union of the input/output parameters of all the clauses along with the branching variable. This enables preservation of the branching semantics of the original ISA program in the HC IR form.

For example in Figure 3, the `bf` instruction at line 7 of the ISA program is changed to a dummy literal at line 7a in the HC IR, plus a predicate call to `fact_aux` on line 7b. The predicate `fact_aux` has two clauses, each representing one of the target blocks of the `bf` instruction. The dummy literal for the `bf` instruction is created so that the resource usage analysis can take it into account when inferring energy usage functions.

Static Single Assignment form (SSA). The last step is to convert the block representation into static single assignment (SSA) form, where each variable is assigned exactly once and multiple assignments to the same variable create new versions of that variable.

In compilers, the SSA form is generated at the function level (e.g., at LLVM [11] level) where a function might consist of multiple basic blocks. However, we follow the approach of generating the SSA form at the block level, and therefore we do not need to generate ϕ nodes. A ϕ node is an instruction used to select a version of the variable depending on the predecessor of the current block. Since each block is already annotated with input/output arguments, any predecessor block will pass the appropriate values as input parameters when making a call to the target block.

In Figure 3, the HC IR (right hand side) is already in SSA form, where each variable is defined exactly once and stack references are transformed to local variables. Each instruction is transformed into a HC IR literal with input/output variables.

Analysis on low level (ISA) representations, in general, suffers from the problem of extracting a precise control flow graph in the presence of indirect jumps and calls. The current implementation of our transformation is restricted to direct jumps and calls. We plan to integrate other techniques into the transformation tool to resolve such problems including recognizing code patterns used by compilers and performing static program analysis (see [26] and its references).

4 General Analysis Framework

In this section we introduce the CiaoPP general resource usage analysis framework and discuss how to instantiate it for the analysis of the HC IR programs resulting from the translation of ISA programs.

CiaoPP includes a global static analyser which is parametric with respect to resources and type of approximation (lower and upper bounds) [17]. The user can define the parameters of the analysis for a particular resource by means of assertions that associate basic cost functions with elementary operations of the base language and procedures in libraries, thus expressing how they affect the usage of a particular resource. The global static analysis can then infer bounds on the resource usage of all the procedures in the program, as functions of input data sizes.

In the rest of the section we use a running example to illustrate the main concepts and steps of the analysis framework. In particular, and for simplicity, assume that we are interested in estimating upper bounds on the energy consumed by the HC IR program in Figure 3 (right hand side) generated from its XC code in Figure 2.

4.1 Instantiating the General Framework

Defining Resources. We start by defining the identifier ("counter") associated to the energy consumption resource, through a declaration:

```
:- resource energy.
```

Expressing the Energy Model. In CiaoPP, the resource usage of primitive operations can be provided using "trust" assertions (see [7] and its references for a description of the assertion language). For example, we can write assertions for each predicate that represents an ISA instruction; these constitute the energy models. The following assertions (for the `add` and `sub` instructions) are part of the simple energy model that we used in the static analysis, which assigns a constant energy consumption to these ISA instructions (values 1215439 and 1210759 respectively):

```
:- trust pred add(X,Y,Z) + resource(avg, energy, 1215439).
:- trust pred sub(X,Y,Z) + resource(avg, energy, 1210759).
```

Note that the first argument (`avg`) of the `resource` property (in the global computational properties field "`+`" of the assertions) expresses that the given energy consumption for the ISA instructions is an average value. This model is obtained using the measurement process described in Section 2, based on Equation (3), so that the energy cost for an ISA instruction i is $c_i = (M_1\ P_i\ O + P_{\text{base}})\ T_{\text{clk}}$, expressed in the third argument of the `resource` property in femto-Joules (fJ, 10^{-15} Joules).

Assertions are also used to express other information that is instrumental in the resource usage analysis. For example, the assertion:

```
:- trust pred sub(X,Y,Z) : (var(X), int(Y), int(Z))
   => (int(X), int(Y), int(Z), size(ub,X,int(Y)-int(Z)),
         size(ub,Y,int(Y)), size(ub,Z,int(Z)))
    + (metric(X,int), metric(Y,int), metric(Z,int)).
```

indicates that if the `sub(X, Y, Z)` predicate (representing the "subtraction" ISA instruction) is called with `X` and `Y` bound to integer numbers and `Z` an unbound variable (precondition field "`:`"), after the successful completion of the call (postcondition field "`=>`"), `X` is an integer number whose size is the size of `Y` minus the size of `Z`. It also expresses that the size metric used for the three arguments is "int", the actual value of the integer numbers.

4.2 Performing the Analysis

Once the parameters of the general resource analysis framework have been defined, and assertions for primitives (representing the energy models) and library calls have been provided, the CiaoPP global static analysis can infer the resource usage of all the procedures/blocks in the program (as functions of input data sizes). A full description of how this is done can be found in [17].

Calling Mode Information. The resource analysis needs information referred to each argument in each predicate in the block representation (HC IR) that expresses whether it acts as an input or an output argument (its "mode"). In our approach no mode analysis is performed in order to obtain such information. The modes of the main blocks are extracted from the XC source code that the HC IR is originated from. This is possible because mode information is statically known at the XC language level and is propagated to the HC IR using (trust) assertions. There are also new intermediate predicates generated by the transformation from ISA programs into HC IR (described in Section 3), originated from conditional branching, which cannot be directly related to the XC source code. However, for such predicates information from the transformation phase, where the input/output arguments are determined for each predicate, is used, so that no mode analysis needs to be performed by CiaoPP.

Size Measure Analysis. CiaoPP uses type information to decide which metric to use to infer and express data sizes, from a set of predefined metrics (see [17] for details). As already said, our resource analysis is performed on a block-based representation (HC IR) of the ISA code generated by the XC compiler. Although XC is a typed language, most of the type information is lost in the ISA code generated by the compiler. There are a number of static and dynamic techniques developed by the reverse engineering community to reconstruct types/shape information from binaries (see [12] and its references). In our approach, we can recover and transfer types from the ISA code into some blocks (predicates) in the HC IR that are directly related to the ISA code, so that no type analysis is performed in those cases. However, we still need to perform some propagation of such types to any new intermediate blocks created by the transformation from ISA programs into Horn clauses. For example, our approach can determine that in the HC IR program in Figure 3 (right hand side) `fact` will be called with $R0$ bound to an integer and $R0\_3$ a free variable, and will succeed with $R0\_3$ bound to an integer. Also, `fact_aux` will be called with the first two arguments bound

to integers, and the rest free, and, upon success, all of them will be bound to integers. Given that information, the chosen metric for all the arguments will be *int*, i.e., the integer value of the argument.

Size Analysis. It determines the relative sizes of variable bindings at different program points. For each clause, size relations are propagated to express each output data size as a function of input data sizes. For recursive functions this is done symbolically, creating a set of recurrence relations that will be solved to get a closed form function.

For our running example, the recurrence relations set up for the size of the output argument $R0\_3$ of `fact` as a function of the size of the input argument $R0$ (denoted $fact_{R0\_3}(R0)$) as well as the corresponding one for `fact_aux` are:

$$fact_{R0\_3}(R0) = fact\_aux_{R0\_4}(0 \leq R0, R0)$$
$$fact\_aux_{R0\_4}(B, R0) = \begin{cases} R0 * fact_{R0\_3}(R0-1) & \text{if } B \text{ is } \texttt{true} \text{ (i.e., } 0 \leq R0) \\ 1 & \text{if } B \text{ is } \texttt{false} \text{ (i.e., } 0 > R0) \end{cases}$$

These inferred recurrence relations/equations are then fed into a computer algebra system (e.g., CiaoPP's internal solver or an external solver such as Mathematica, used for the results presented in this paper) that gives the following closed form function for it: $fact_{R0\_3}(R0) = R0!$

Resource Usage Analysis. It uses the size information inferred by the size analysis to set up recurrence equations representing the resource usage of predicates (blocks), and computes bounds to their solutions. Remember that c_i represents the energy cost of each instruction, taken from the energy model. Let b_e denote the energy consumption function for a predicate (block) `b`. Then, the inferred equations for `fact` are:

$$fact_e(R0) = fact\_aux_e(0 \leq R0, R0) + c_{entsp} + c_{stw} + c_{ldw} + c_{ldc} + c_{lss} + c_{bf}$$
$$fact\_aux_e(B, R0) = \begin{cases} fact_e(R0-1) + c_{bu} + 2\ c_{ldw} + c_{sub} + \\ \qquad + c_{bl} + c_{mul} + c_{retsp} & \text{if } B \text{ is } \texttt{true} \\ c_{mkmsk} + c_{retsp} & \text{if } B \text{ is } \texttt{false} \end{cases}$$

If we assume (for simplicity of exposition) that each instruction has unitary cost, i.e., $c_i = 1$ for all i, we obtain (using the mentioned computer algebra system) the energy consumed by `fact` as a function of its input data size ($R0$): $fact_e(R0) = 13\ R0 + 8$.

Note that our approach based on setting up recurrence equations and solving them using a computer algebra system allows inferring different types of (resource usage) functions, such as polynomial, factorial, exponential, logarithmic, and summatory.

Note also that using average values in the model implies that the energy function for the whole program inferred by the upper-bound resource analysis is an approximation of the actual upper bound that can possibly be below it. To ensure that the analysis infers a strict upper bound, we would need to use

Table 1. Description of benchmark functions used in experiments and their corresponding energy functions.

Function name	**Description**	**Energy function**
`fact(N)`	Calculates $N!$	$26.0\ N + 19.4$
`fibonacci(N)`	Nth Fibonacci no.	$30.1 + 35.6\ \phi^N + 11.0\ (1-\phi)^N$
`sqr(N)`	Computes N^2	$103.0\ N^2 + 205.8\ N + 188.32$
`poweroftwo(N)`	Calculates 2^N	$62.4 \cdot 2^N - 312.3$
`power(base,exp)`	Calculates $base^{exp}$	$6.3\ (\log_2 exp + 1) + 6.5$

strict upper bounds as well in the energy models. However, with the current models such bounds would be very conservative, causing a loss in accuracy that would make the analysis not useful in practice. Thus, the current approach is a practical compromise.

5 Benchmarks, Results and Evaluation

The aim of the experimental evaluation is to perform a first comparison of actual hardware energy measurements, in terms of accuracy, with the values obtained from both the low-level Instruction Set Simulation (ISS) model and the Static Resource Analysis (SRA) implemented within the CiaoPP framework, to obtain an early estimation of the feasibility of the approach. To this end, we describe a selection of currently analysable benchmarks, the method by which data was collected, and an evaluation of the analysis framework accuracy vs. the low-level ISS model and hardware measurements.

Benchmarks. For this type of evaluation we use as benchmarks mainly small mathematical functions. The structure of these programs is either iterative or recursive, with their cost depending on the function argument. For such programs state of the art solvers can easily provide the cost functions, by solving the system of recurrence relations provided by the SRA framework. Table 1 shows the benchmarks used in this comparison, their execution behaviour in relation to each function's parameters, and the cost function inferred.

Experimental method. Hardware energy readings were obtained by repeatedly executing a benchmark function over a 0.5 second period, T, collecting a set of power samples, P, whilst counting the number of executions, N_{fn}. From this, the energy of a single function call, $E_{\text{fn}} = \frac{\text{mean}(P) \times T}{N_{\text{fn}}}$ is calculated. This was performed using a similar method to the collection of energy model data described in Section 2, but was performed on separate hardware so as to de-couple modelling from testing.

ISS modelling involved simulating the same function a smaller number of times than on the hardware in order to keep simulation time adequately low.

Table 2. Actual and estimated energy consumption for the `fact(N)` function over a range of N.

SRA cost function(nJ)	N	HW measured energy (nJ)	Model energy (nJ)		Error vs. HW	
			ISS	SRA	ISS	SRA
$26.0\ N + 19.4$	1	53.1	62.8	45.3	1.18	0.85
	2	78.0	83.8	71.3	1.07	0.91
	4	127.7	125.7	123.1	0.98	0.96
	8	227.1	209.6	226.8	0.92	1.00
	16	426.0	377.4	434.2	0.89	1.02
	32	823.8	713.4	849.0	0.87	1.03
	64	1690.5	1387.0	1678.4	0.82	0.99

The instruction statistics were then processed in order to produce an energy figure, and then that figure divided by N_{fn} was used during ISS in order to extract the energy of a single call. The ISS modelling framework currently has a less efficient test loop than the hardware, potentially reducing accuracy for very short function calls. Similarly, if too few function calls are made during the simulation due to a long-executing function, overrun in the test time may skew low-level energy figures.

Static resource usage analysis was performed by evaluating the produced cost function for a given benchmark with respect to the input arguments, immediately providing the energy cost of a single function call.

Results. Table 2 provides an example of test data for the `fact` (factorial) function. The hardware (HW), low-level Instruction Set Simulation model (ISS), and Static Resource Analysis (SRA) model energy figures are compared. The relative errors of ISS and SRA are compared with respect to the HW energy and normalised as such. The cost function provided for this particular example demonstrates the relationship between the input parameter, N, and the SRA estimate of such a call. This, together with data for a number of further benchmarks are presented in graph form in Figure 5.

In Figure 5, hardware measured energy is compared directly to ISS and SRA energy predictions for the set of four benchmarks. The relative errors are also plotted. In all cases, the ISS model is seen to improve in accuracy as the input parameter N increases, in line with the expected inaccuracies arising from inefficiencies in the modelling loop used in simulation, as described in the previous subsection. In the case of the `poweroftwo` function, time limitations prevent the ISS model from approximating the function above $N = 13$, approaching which the error begins to increase markedly. The `power` function behaves in a similar way and demonstrates the relationship between multiple input arguments.

The CiaoPP SRA model does not suffer the same deficiencies, although it does incur a greater underestimation of energy for small values of N. The HW measurements unavoidably contain some loop code beyond the target function being examined and small N values will increase the effects of this in the measurement. ISS in fact models this inefficiency directly, whereas SRA does not,

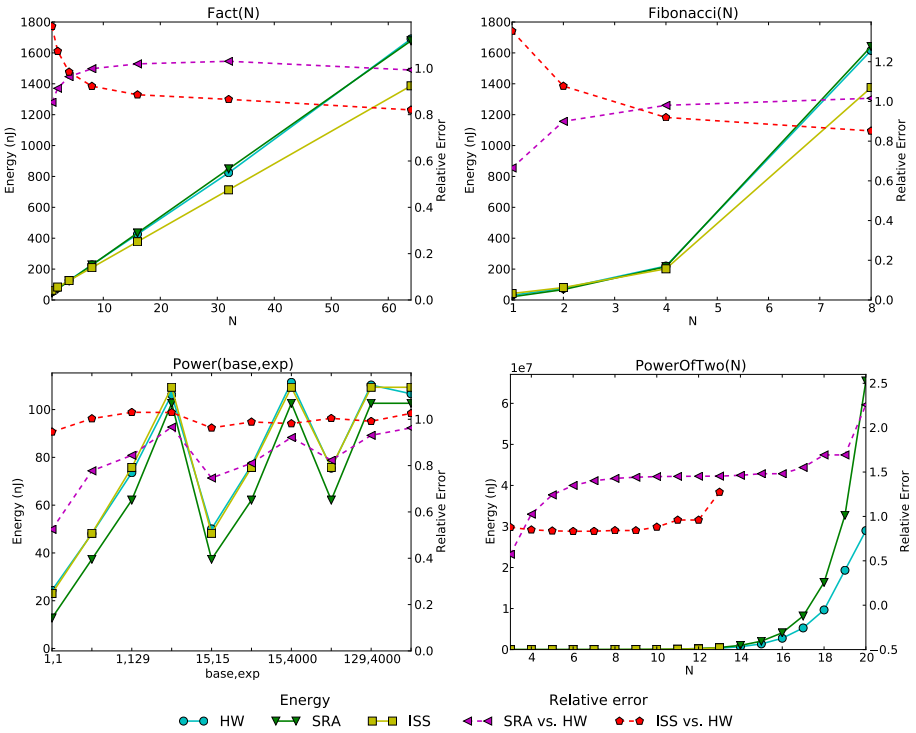


Fig. 5. Hardware energy, estimations and relative errors for (starting top-left, moving clock-wise) `fact`, `fibonacci`, `poweroftwo` and `power`.

hence the roughly symmetrical relative errors for the two models, particularly in the `fact` and `fibonacci` cases.

Both approaches are reliant on the same underlying instruction energy figures. Given that some instructions are not directly profiled and, instead, given an average value, accuracy is reduced when the distribution of instructions in a given program is such that the number of profiled instructions is low.

Overall, these results demonstrate both models' capabilities to estimate energy, with encouraging accuracy that can be improved upon. Further, the SRA approach is less restrictive, particularly in situations where simulation time might be prohibitively long.

6 Related Work

Static cost analysis techniques based on setting up and solving recurrence equations date back to Wegbreit's seminal paper [25], and have been developed significantly in subsequent work [19, 2, 3, 23, 17, 1, 16, 21]. This approach was first applied to energy consumption in [15], which inferred statically upper-bounds

on the energy consumption of Java programs as functions of input data sizes. As herein, this work used the generic framework of [17, 6], specializing it for Java bytecode [14, 16] by translating the Jimple (a typed three-address code) representation of Java bytecode into the Horn clause-based IR of the analyzer [14]. However, we employ transformations at lower level (XS1-ISA), irrespective of source language in general, where much of the program structure and typing information is trimmed away. Our transformation employs analysis techniques to reverse engineer ISA programs and reconstruct the control flow graph so that the equivalent HC IR safely approximates the semantics of the original ISA program. In addition, [15] did not compare the results with actual, measured energy consumptions and used a comparatively simple energy model.

Other approaches to cost analysis, such as, e.g., those based on the potential method [8], are limited to polynomial bounds, and do not allow inferring non-polynomial energy functions, as in the recurrence equation method. A number of static analyses are aimed at inferring worst case execution time (WCET, see, e.g., [4] and its references) and related techniques have been applied in [9] to derive a worst-case energy analysis. However, WCET methods typically do not infer cost functions on input data sizes but rather absolute maximum values, and they generally require manual annotation of loops with an upper bound on the number of iterations.

Other transformation-based approaches have been proposed in order to analyse low level microprocessor code [5] and Java source and bytecode [1] (outside the context of energy analysis).

Instruction Set Simulation can be used to estimate the energy of a program running on a suitably profiled hardware platform. Simple models for single-threaded architectures have been demonstrated [22]. These have then been expanded upon, leading to models capable of modelling more complex hardware such as that used in this paper, which comprises a multi-threaded architecture [10].

7 Conclusions and Future Work

In this paper we introduce an approach for estimating the energy consumption of programs compiled for the XS1 architecture, based on a Horn clause transformation and the use of ISA level models that we have produced. We have shown the feasibility of the approach with a prototype implementation within the CiaoPP system, which has been successful in statically finding a good approximation of the energy consumed by a set of selected programs in our experiments.

The XS1 architecture is inherently multi-threaded, and the simulation-based model is able to provide energy estimates for this. Statically analysing multiple concurrent threads adds a significant new dimension of complexity to the modelling exercise. This is a goal of further work in order to provide meaningful analysis for contemporary multi-threaded programs running on this architecture.

We also plan to produce and deal with energy models that take into account the switching cost among pairs of ISA instructions (i.e., the energy consumed

by bit flipping), since our analysis framework allows it. The improvement in accuracy from this approach can vary between architectures, for example research such as [20], shows that a simple model can be sufficient in some cases, due to bit flipping effects averaging out over time. Thus, the impact in the context of any target architectures must therefore be considered in this future work, in order to establish whether the increased complexity of analysis delivers a worthwhile gain in accuracy.

We also intend to improve upon the energy measurements of commonly used instructions, which involves more complex techniques such as linear regression. This technique can also be used to construct energy models of intermediate compiler representations such as LLVM IR [11], which would enable us to apply our analysis techniques to more structured program representations. Another method for analysing LLVM IR would involve mapping low-level program instruction segments to LLVM IR segments and reusing the ISA-level energy models.

Acknowledgements: The research leading to these results has received funding from the European Union 7th Framework Programme under grant agreement 318337, ENTRA - Whole-Systems Energy Transparency, Spanish MINECO TIN'12-39391 *StrongSoft* and TIN'08-05624 *DOVES* projects, and Madrid TIC-1465 *PROMETIDOS-CM* project. We also thank John Gallagher for useful and fruitful discussions and feedback in general, and in particular for his help on the implementation of a translation for removing multiple recursions in Horn clause programs, which is performed prior to setting up recurrence equations.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. of ESOP'07*, volume 4421 of *LNCS*. Springer, 2007.
2. 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.
3. 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.
4. Reinhard Wilhelm et al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.
5. 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.
6. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2), 2005.
7. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. 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. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.
9. R. Jayaseelan, T. Mitra, and X. Li. Estimating the worst-case energy consumption of embedded software. In *IEEE Real Time Technology and Applications Symposium*, pages 81–90. IEEE Computer Society, 2006.
10. S. Kerrison and K. Eder. Energy modelling and optimisation of software for a hardware multi-threaded embedded microprocessor. Submitted to ACM Transactions on Embedded Computing Systems, Sept. 2013, under review. Technical report, University of Bristol, June 2013.
11. C. Lattner and V.S. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, 2004.
12. JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011*. The Internet Society, February 2011.
13. D. May. The XMOS XS1 architecture. available online: http://www.xmos.com/published/xmos-xs1-architecture, 2013.
14. M. Mendez-Lojo, J.A. Navas, and M.V. Hermenegildo. A flexible, (c)lp-based approach to the analysis of object-oriented programs. In *LOPSTR*, pages 154–168, 2007.
15. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, April 2008. Extended Abstract.
16. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of BYTECODE*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 65–82. Elsevier - North Holland, March 2009.
17. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *International Conference on Logic Programming (ICLP'07)*, Lecture Notes in Computer Science. Springer, 2007.
18. F. Nielson, HR. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
19. M. Rosendahl. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*. ACM Press, 1989.
20. J. T. Russell and M. F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *ICCD*, pages 328–333, 1998.
21. 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.
22. V. Tiwari, S. Malik, A. Wolfe, and M. T. C. Lee. Instruction level power analysis and optimization of software. In *Proceedings of VLSI Design*, pages 326–328, 1996.
23. P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.
24. D. Watt. *Programming XC on XMOS Devices*. XMOS Limited, 2009.
25. B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.
26. L. Xu, F. Sun, and Z. Su. Constructing Precise Control Flow Graphs from Binaries. *University of California, Davis, Tech. Rep*, 2009.