Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR

Article · November 2015

Source: arXiv

CITATIONS
16

READS
123

7 authors, including:

Umer Liqat
Madrid Institute for Advanced Studies
16 PUBLICATIONS **129** CITATIONS
SEE PROFILE

Kyriakos Georgiou
University of Bristol
26 PUBLICATIONS **202** CITATIONS
SEE PROFILE

Steve Kerrison
MicroSec Pte Ltd, Singapore
30 PUBLICATIONS **251** CITATIONS
SEE PROFILE

Pedro López-García
Madrid Institute for Advanced Studies
77 PUBLICATIONS **1,126** CITATIONS
SEE PROFILE

Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR

U. Liqat[1], K. Georgiou[2], S. Kerrison[2], P. Lopez-Garcia[1,3], John P. Gallagher[5], M.V. Hermenegildo[1,4], and K. Eder[2]

[1] IMDEA Software Institute, Madrid, Spain
{umer.liqat,pedro.lopez,manuel.hermenegildo}@imdea.org
[2] University of Bristol, Bristol, UK
{kyriakos.georgiou,steve.kerrison,kerstin.eder}@bristol.ac.uk
[3] Spanish Council for Scientific Research (CSIC), Madrid, Spain
[4] Universidad Politécnica de Madrid (UPM), Madrid, Spain
[5] Roskilde University, Roskilde, Denmark
jpg@ruc.dk

Abstract. The static estimation of the energy consumed by program executions is an important challenge, which has applications in program optimization and verification, and is instrumental in energy-aware software development. Our objective is to estimate such energy consumption in the form of *functions on the input data sizes of programs*. We have developed a tool for experimentation with static analysis which infers such energy functions at two levels, the instruction set architecture (ISA) and the intermediate code (LLVM IR) levels, and reflects it upwards to the higher source code level. This required the development of a translation from LLVM IR to an intermediate representation and its integration with existing components, a translation from ISA to the same representation, a resource analyzer, an ISA-level energy model, and a mapping from this model to LLVM IR. The approach has been applied to programs written in the XC language running on XCore architectures, but is general enough to be applied to other languages. Experimental results show that our LLVM IR level analysis is reasonably accurate (less than 6.4% average error vs. hardware measurements) and more powerful than analysis at the ISA level. This paper provides insights into the trade-off of precision versus analyzability at these levels.

Keywords: Energy Consumption Analysis, Resource Usage Analysis, Static Analysis, Embedded Systems.

1 Introduction

Energy consumption and the environmental impact of computing technologies have become a major worldwide concern. It is an important issue in high-performance computing, distributed applications, and data centers. There is also

increased demand for complex computing systems which have to operate on batteries, such as implantable/portable medical devices or mobile phones. Despite advances in power-efficient hardware, more energy savings can be achieved by improving the way current software technologies make use of such hardware.

The process of developing energy-efficient software can benefit greatly from static analyses that estimate the energy consumed by program executions without actually running them. Such estimations can be used for different software-development tasks, such as performing automatic optimizations, verifying energy-related specifications, and helping system developers to better understand the impact of their designs on energy consumption. These tasks often relate to the source code level. For example, source-to-source transformations to produce optimized programs are quite common. Specifications included in the source code can be proved or disproved by comparing them with safe information inferred by analysis. Such information, when referred to the procedures in the source code can be useful for example to detect which are the most energy-consuming ones and replace them by more energy-efficient implementations. On the other hand, energy consumption analysis must typically be performed at lower levels in order to take into account the effect of compiler optimizations and to link to an energy model. Thus, the inference of energy consumption information for lower levels such as the Instruction Set Architecture (ISA) or intermediate compiler representations (such as LLVM IR [21]) is fundamental for two reasons: 1) It is an intermediate step that allows propagation of energy consumption information from such lower levels up to the source code level; and 2) it enables optimizations or other applications at the ISA and LLVM IR levels.

In this paper (an improved version of [22]) we propose a static analysis approach that infers energy consumption information at the ISA and LLVM IR levels, 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* that are inserted in the program representation at each of these levels. The user (i.e., the "energy-efficient software developer") can customize the system by selecting the level at which the analysis will be performed (ISA or LLVM IR) and the level at which energy information will be output (ISA, LLVM IR or source code). As we will show later, the selection of analysis level has an impact on the analysis accuracy and on the class of programs that can be analyzed.

The main goal of this paper is to study the feasibility and practicability of the proposed analysis approach and perform an initial experimental assessment to shed light on the trade-offs implied by performing the analysis at the ISA or LLVM levels. In our experiments we focus on the energy analysis of programs written in XC [33] running on the XMOS XS1-L architecture. However, the concepts presented here are neither language nor architecture dependent and thus can be applied to the analysis of other programming languages (and associated lower level program representations) and architectures as well. XC is a high-level C-based programming language that includes extensions for concurrency, communication, input/output operations, and real-time behavior. In order to potentially support different programming languages and different program rep-

resentations at different levels of compilation (e.g., LLVM IR and ISA) in the same analysis framework we differentiate between the *input language* (which can be XC source, LLVM IR, or ISA) and the *intermediate semantic program representation* that the resource analysis operates on. The latter is a series of connected code blocks, represented by Horn Clauses, that we will refer to as "HC IR" from now on. We then propose a transformation from each *input language* into the HC IR and passing it to a resource analyzer. The HC IR representation as well as a transformation from LLVM IR into HC IR will be explained in Section 3. In our implementation we use an extension of the CiaoPP [13] resource analyzer. This analyzer always deals with the HC IR in the same way, independent of its origin, inferring energy consumption functions for all procedures in the HC IR program. The main reason for choosing Horn Clauses as the intermediate representation is that it offers a good number of features that make it very convenient for the analysis [24]. For instance, it supports naturally Static Single Assignment (SSA) and recursive forms, as will be explained later. In fact, there is a current trend favoring the use of Horn Clause programs as intermediate representations in analysis and verification tools [9,17,5,4].

Although our experiments are based on single-threaded XC programs (which do not use pointers, since XC does not support them), our claim about the generality and feasibility of our proposed approach for static resource analysis is supported by existing tools based on the Horn Clause representation that can successfully deal with C source programs that exhibit interesting features such as the use of pointers, arrays, shared-memory, or concurrency in order to analyze and verify a wide range of properties [9,17,11]. For example [11] is a tool for the verification of safety properties of C programs which can reason about scalars and pointer addresses, as well as memory contents. It represents the bytecode corresponding to a C program by using (constraint) Horn clauses.

Both static analysis and energy models can potentially relate to any language level (such as XC source, LLVM IR, or ISA). Performing the analysis at a given level means that the representation of the program at that level is transformed into the HC IR, and the analyzer "mimics" the semantics of instructions at that level. The energy model at a given level provides basic information on the energy cost of instructions at that level. The analysis results at a given level can be mapped upwards to a higher level, e.g. from ISA or LLVM IR to XC. Furthermore, it is possible to perform analysis at a given level with an energy model for a lower level. In this case the energy model must be reflected up to the analysis level.

Our hypothesis is that the choice of level will affect the accuracy of the energy models and the precision of the analysis in opposite ways: energy models at lower levels (e.g. at the ISA level) will be 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 on the hardware. However, at lower levels more program structure and data type/shape information is lost due to lower-level representations, and we expect a corresponding loss of analysis accuracy. We could devise mechanisms to represent such higher-level information and pass it

down to the lower-level ISA, or to recover it by analysing the ISA. However, our goal is to compare the analysis at the LLVM IR and ISA levels without introducing such mechanisms, which might be complex or not effective in some cases (e.g., in abstracting memory operations or recovering type information).

This hypothesis about the analysis/modelling level trade-off (and potential choices) is illustrated in Figure 1. The possible choices are classified into two groups: those that analyze and model at the same level, and those that operate at different levels. For the latter, the problem is finding good mappings between software segments from the level at which the model is defined up to the level at which the analysis is performed, in a way that does not lose accuracy in the energy information.

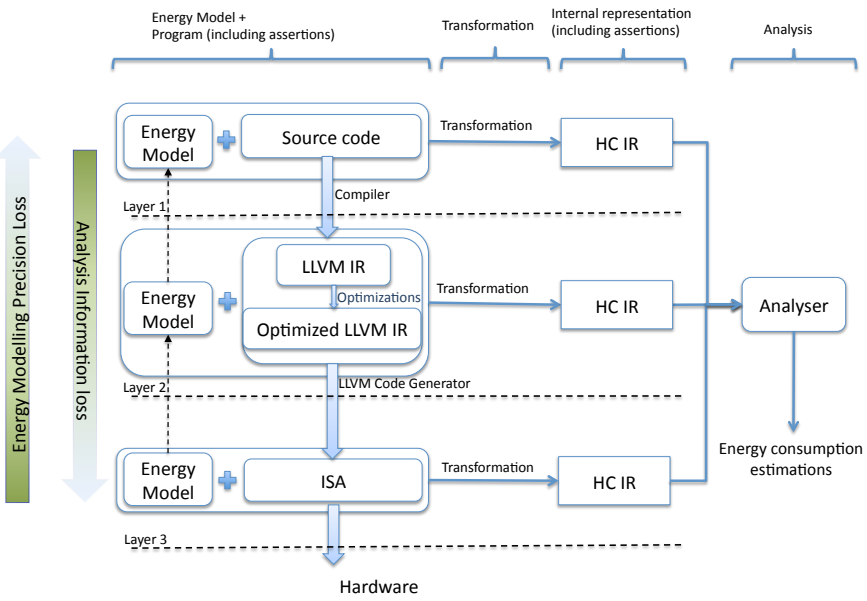

Fig. 1: Analysis/modelling level trade-off and potential choices.

In this paper we concentrate on two of these choices and their comparison, to see if our hypothesis holds. In particular, the first approach (choice 1) is represented by analysing the generated ISA-level code using models defined at the ISA level that express the energy consumed by the execution of individual ISA instructions. This approach was explored in [23]. It used the precise ISA-level energy models presented in [19], which when used in the static analysis of [23] for a number of small numerical programs resulted in the inference of functions that provide reasonably accurate energy consumption estimations for any input data

size (3.9% average error vs. hardware measurements). However, when dealing with programs involving structured types such as arrays, it also pointed out that, due to the loss of information related to program structure and types of arguments at the ISA level (since it is compiled away and no longer relates cleanly to source code), the power of the analysis was limited. In this paper we start by exploring an alternative approach: the analysis of the generated LLVM IR (which retains much more of such information, enabling more direct analysis as well as mapping of the analysis information back to source level) together with techniques that map segments of ISA instructions to LLVM IR blocks [8] (choice 2). This mapping is used to propagate the energy model information defined at the ISA level up to the level at which the analysis is performed, the LLVM IR level. In order to complete the LLVM IR-level analysis, we have also developed and implemented a transformation from LLVM IR into HC IR and used the CiaoPP resource analyzer. This results in a parametric analysis that similarly to [23] infers energy consumption functions, but operating on the LLVM IR level rather than the ISA level.

We have performed an experimental comparison of the two choices for generating energy consumption functions. Our results support our intuitions about the trade-offs involved. They also provide evidence that the LLVM IR-level analysis (choice 2) offers a good compromise within the level hierarchy, since it broadens the class of programs that can be analyzed without the need for developing complex techniques for recovering type information and abstracting memory operations, and without significant loss of accuracy.

In summary, the original contributions of this paper are:

1. A translation from LLVM IR to HC IR (Section 3).
2. The integration of all components into an experimental tool architecture, enabling the static inference of energy consumption information in the form of *functions on input data sizes* and the experimentation with the trade-offs described above (Section 2). The components are: LLVM IR and ISA translations, ISA-level energy model and mapping technique (Section 4 and [19,8]), and analysis tools (Section 5 and [27,30]).
3. The experimental results and evidence of trade-off of precision versus analyzability (Section 6).
4. A sketch of how the static analysis system can be integrated in a source-level Integrated Development Environment (IDE) (Section 2).

Finally, some related work is discussed in Section 7, and Section 8 summarises our conclusions and comments on ongoing and future work.

2 Overview of the Analysis at the LLVM IR Level

An overview of the proposed analysis system at the LLVM IR level using models at the ISA level is depicted in Figure 2. The system takes as input an XC source program that can (optionally) contain assertions (used to provide useful hints and information to the analyzer), from which a *Transformation and Mapping*

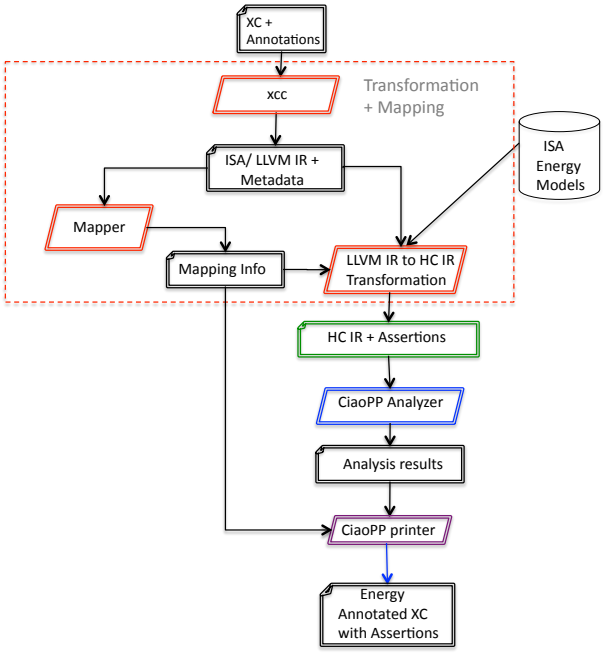

Fig. 2: An overview of the analysis at the LLVM IR level using ISA models.

process (dotted red box) generates first its associated LLVM IR using the xcc compiler. Then, a transformation from LLVM IR into HC IR is performed (explained in Section 3) obtaining the intermediate representation (green box) that is supplied to the CiaoPP analyzer. This representation includes assertions that express the energy consumed by the LLVM IR blocks, generated from the information produced by the mapper tool (as explained in Section 4). The *CiaoPP analyzer* (blue box, described in Section 5) takes the HC IR, together with the assertions which express the energy consumed by LLVM IR blocks, and possibly some additional (trusted) information, and processes them, producing the analysis results, which are expressed also using assertions. Based on the procedural interpretation of these HC IR programs and the resource-related information contained in the assertions, the resource analysis can infer static bounds on the energy consumption of the HC IR programs that are applicable to the original LLVM IR and, hence, to their corresponding XC programs. The analysis results include energy consumption information expressed as functions on data sizes for the whole program and for all the procedures and functions in it. Such results are then processed by the *CiaoPP printer* (purple box) which presents the information to the program developer in a user-friendly format.

3 LLVM IR to HC IR Transformation

In this section we describe the LLVM IR to HC IR transformation that we have developed in order to achieve the complete analysis system at the LLVM IR level proposed in the paper (as already mentioned in the overview given in Section 2 and depicted in Figure 2).

A Horn clause (HC) is a first-order predicate logic formula of the form $\forall(S_1 \wedge \ldots \wedge S_n \rightarrow S_0)$ where all variables in the clause are universally quantified over the whole formula, and $S_0, S_1, \ldots, S_n$ are atomic formulas, also called literals. It is usually written $S_0 \ :- \ S_1, \ldots, S_n$.

The HC IR representation consists of a sequence of *blocks* where each block is represented as a *Horn clause*:

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

Each block has an entry point, that we call the *head* of the block (to the left of the $:-$ symbol), with a number of parameters $< params >$, and a sequence of steps (the *body*, to the right of the $:-$ symbol). Each of these S_i steps (or *literals*) is either (the representation of) an LLVM IR *instruction*, or a *call* to another (or the same) block. The analyzer deals with the HC IR always in the same way, independent 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 LLVM IR programs.

The transformation also passes energy values for the LLVM IR level for different programs based on the ISA/LLVM IR mapping information that express the energy consumed by the LLVM IR blocks, as explained in Section 4. Such information is represented by means of *trust* assertions (in the Ciao assertion language [14]) that are included in the HC IR. In general, *trust* assertions can be used to provide information about the program and its constituent parts (e.g., individual instructions or whole procedures or functions) to be trusted by the analysis system, i.e., they provide base information assumed to be true by the inference mechanism of the analysis in order to propagate it throughout the program and obtain information for the rest of its constituent parts.

LLVM IR programs are expressed using typed assembly-like instructions. Each function is in SSA form, represented as a sequence of basic blocks. Each basic block is a sequence of LLVM IR instructions that are guaranteed to be executed in the same order. Each block ends in either a branching or a return instruction. In order to transform an LLVM IR program into the HC IR, we follow a similar approach as in a previous ISA-level transformation [23]. However, the LLVM IR includes an additional type transformation as well as better memory modelling.

The following subsections describe the main aspects of the transformation.

3.1 Inferring Block Arguments

As described before, a *block* in the HC IR has an entry point (head) with input/output parameters, and a body containing a sequence of steps (here, repre-

sentations of LLVM IR instructions). Since the scope of the variables in LLVM IR blocks is at the function level, the blocks are not required to pass parameters while making jumps to other blocks. Thus, in order to represent LLVM IR blocks as HC IR blocks, we need to infer input/output parameters for each block.

For entry blocks, the input and output arguments are the same as the ones to the function. We define the functions $param_{in}$ and $param_{out}$ which infer input and output parameters to a block respectively. These are recomputed according to the following definitions until a fixpoint is reached:

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

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

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

where $def(k)$ and $ref(k)$ denote the variables written or referred to at a node (instruction) k in the block, respectively, and n is the number of nodes in the block.

Note that the LLVM IR is in SSA form at the function level, which means that blocks may have ϕ nodes which are created while transforming the program into SSA form. A ϕ node is essentially a function defining a new variable by selecting one of the multiple instances of the same variable coming from multiple predecessor blocks:

$$x = \phi(x_1, x_2, ..., x_n)$$

def and *ref* for this instruction are $\{x\}$ and $\{x_1, x_2, ..., x_n\}$ respectively. An interesting feature of our approach is that ϕ nodes are not needed. Once the input/output parameters are inferred for each block as explained above, a post-process gets rid of all ϕ nodes by modifying block input arguments in such a way that blocks receive x directly as an input and an appropriate x_i is passed by the call site. This will be illustrated later in Section 3.3.

Consider the example in Figure 4 (left), where the LLVM IR block *looptest* is defined. The body of the block reads from 2 variables without previously defining them in the same block. The fixpoint analysis would yield:

$$params_{in}(looptest) = \{Arr, I\}$$

which is used to construct the HC IR representation of the *looptest* block shown in Figure 4 (right), line 3.

3.2 Translating LLVM IR Types into HC IR Types

LLVM IR is a typed representation which allows retaining much more of the (source) program information than the ISA representation (e.g., types defining compound data structures). As already mentioned, this enables a more direct analysis as well as mapping of the analysis information back to source level. Thus, we define a mechanism to translate LLVM IR types into their counterparts in HC IR.

The LLVM type system defines primitive and derived types. The primitive types are the fundamental building blocks of the type system. Primitive types include *label, void, integer, character, floating point, x86mmx,* and *metadata*. The *x86mmx* type represents a value held in an MMX register on an x86 machine and the *metadata* type represents embedded metadata. The derived types are created from primitive types or other derived types. They include *array, function, pointer, structure, vector, opaque*. Since the XCore platform supports neither pointers nor floating point data types, the LLVM IR code generated from XC programs uses only a subset of the LLVM types.

At the HC IR level we use *regular types*, one of the type systems supported by CiaoPP [13]. Translating LLVM IR primitive types into regular types is straightforward. The *integer* and *character* types are abstracted as *num* regular type, whereas the *label*, *void*, and *metadata* types are represented as *atm* (atoms).

For derived types, corresponding non-primitive regular types are constructed during the transformation phase. Supporting non-primitive types is important because it enables the analysis to infer energy consumption functions that depend on the sizes of internal parts of complex data structures. The array, vector, and structure types are represented as follows:

$$\begin{gathered} array\_type \rightarrow (nested)list \\ vector\_type \rightarrow (nested)list \\ structure\_type \rightarrow functor\_term \end{gathered}$$

Both the *array* and *vector* types are represented by the *list* type in CiaoPP which is a special case of compound term. The type of the elements of such lists can be again a primitive or a derived type. The *structure* type is represented by a compound term which is composed of an atom (called the *functor*, which gives a name to the structure) and a number of *arguments*, which are again either primitive or derived types. LLVM also introduces pointer types in the intermediate representation, even if the front-end language does not support them (as in the case of XC, as mentioned before). Pointers are used in the pass-by-reference mechanism for arguments, in memory allocations in *alloca* blocks, and in memory load and store operations. The types of these pointer variables in the HC IR are the same as the types of the data these pointers point to.

Consider for example the types in the XC program shown in Figure 3. The type of argument *Arg* of the *print* function is an array of *mystruct* elements. *mystruct* is further composed of an integer and an array of integers. The LLVM IR code generated by xcc for the function signature *print* in Figure 3 (left) is:

$$\text{define void @print(} [0 \times \{i32, [5 \times i32]\}]^* \text{ noalias nocapture)}$$

```
struct mystruct{
  int x;
  int arr[5];
};

void print(struct mystruct [] Arg, int N)
{
  ...
}
```

```
:- regtype array1/1.
array1:=[] | [~struct|array1].

:- regtype struct/1.

struct:=mystruct(~num,~array2).

:- regtype array2/1.
array2:=[] | [~num|array2].
```

Fig. 3: An XC program and its type transformation into HC IR.

The function argument type in the LLVM IR ($[0 \times \{i32, [5 \times i32]\}]$) is the typed representation of the argument *Arg* to the function in the XC program. It represents an array of arbitrary length with elements of $\{i32, [5 \times i32]\}$ structure type which is further composed of an $i32$ integer type and a $[5 \times i32]$ array type, i.e., an array of 5 elements of $i32$ integer type.[6]

This type is represented in the HC IR using the set of regular types illustrated in Figure 3 (right). The regular type *array*1, is a list of *struct* elements (which can also be simply written as `array1 := list(struct)`). Each *struct* type element is represented as a functor *mystruct*/2 where the first argument is a *num* and the second is another list type *array*2. The type *array*2 is defined to be a list of *num* (which, again, can also be simply written as `array2 := list(num)`).

3.3 Transforming LLVM IR Blocks/Instructions into HC IR

In order to represent an LLVM IR function by an HC IR function (i.e., a predicate), we need to represent each LLVM IR block by an HC IR block (i.e., a Horn clause) and hence each LLVM IR instruction by an HC IR literal.

```
1  alloca:
2   br label looptest
3  looptest:
4   %I=phi i32[%N,%alloca],
                    [%I1,%loopbody]
5   %Zcmp=icmp ne i32 %I, 0
6   br i1 %Zcmp, label %loopbody,
                 label %loopend
7  loopbody:
8   %Elm=getelementptr [0xi32]*%Arr,
                       i32 0,i32 %I
9   //process array element 'Elm'
10  %I1=sub i32 %I, 1
11  br label %looptest
12 loopend:
13  ret void
```

```
1  alloca(N, Arr):-
2   looptest(N, Arr).
3  looptest(I, Arr):-
4   icmp_ne(I, 0, Zcmp),
5   loopbody_loopend(Zcmp,I,Arr).
6  icmp_ne(X, Y, 1):- X \= Y.
7  icmp_ne(X, Y, 0):- X = Y.
8  loopbody_loopend(Zcmp,I,Arr):-
9   Zcmp=1,
10  nth(I, Arr, Elm),
11  //process list element 'Elm'
12  I1 is I - 1, sub(I,1,I1),
13  looptest(I1, Arr).
14 loopbody_loopend(Zcmp,I,Arr):-
15  Zcmp=0.
```

Fig. 4: LLVM IR Array traversal example (left) and its HC IR representation (right)

[6] $[0 \times i32]$ specifies an arbitrary length array of $i32$ integer type elements.

The LLVM IR instructions are transformed into equivalent HC IR literals where the semantics of the execution of the LLVM IR instructions are either described using trust assertions or by giving definition to HC IR literals. The *phi assignment* instructions are removed and the semantics of the *phi assignment* are preserved on the call sites. For example, the *phi assignment* is removed from the HC IR block in Figure 4 (right) and the semantics of the *phi assignment* is preserved on the call sites of the *looptest* (lines 2 and 14). The call sites *alloca* (line 2) and *loopbody* (line 13) pass the corresponding value as an argument to *looptest*, which is received by *looptest* in its first argument I.

Consider the instruction *getelementptr* at line 8 in Figure 4 (left), which computes the address of an element of an array *%Arr* indexed by *%I* and assigns it to a variable *%Elm*. Such an instruction is represented by a call to an abstract predicate *nth/3*, which extracts a reference to an element from a list, and whose effect of execution on energy consumption as well as the relationship between the sizes of input and output arguments is described using trust assertions. For example, the assertion:

```
:- trust pred nth(I, L, Elem)
  :(num(I), list(L, num), var(Elem))
  => ( num(I), list(L, num), num(Elem),
       rsize(I, num(IL, IU)),
       rsize(L, list(LL, LU, num(EL, EU))),
       rsize(Elem, num(EL, EU)) )
       + (resource(avg, energy, 1215439) ).
```

indicates that if the `nth(I, L, Elem)` predicate (representing the *getelementptr* LLVM IR instruction) is called with I and L bound to an integer and a list of numbers respectively, and *Elem* an unbound variable (precondition field "`:`"), then, after the successful completion of the call (postcondition field "`=>`"), *Elem* is an integer number and the lower and upper bounds on its size are equal to the lower and upper bounds on the sizes of the elements of the list L. The sizes of the arguments to *nth/3* are expressed using the property *rsize* in the assertion language. The lower and upper bounds on the length of the list L are LL and LU respectively. Similarly, the lower and upper bounds on the elements of the list are EL and EU respectively, which are also the bounds for *Elem*. The *resource* property (global computational properties field +) expresses that the energy consumption for the instruction is an average value (1215439 nano-joules[7]).

The branching instructions in LLVM IR are transformed into calls to target blocks in HC IR. For example, the branching instruction at line 6 in Figure 4 (left), which jumps to one of the two blocks *loopbody* or *loopend* based on the Boolean variable *Zcmp*, is transformed into a call to a predicate with two clauses (line 5 in Figure 4 (right)). The name of the predicate is the concatenation of the names of the two LLVM IR blocks mentioned above. The two clauses of the predicate defined at lines 8-13 and 14-15 in Figure 4 (right) represent the LLVM IR blocks *loopbody* and *loopend* respectively. The test on the conditional variable is placed in both clauses to preserve the semantics of the conditional branch.

[7] nJ, 10^{-9} joules

4 Obtaining the Energy Consumption of LLVM IR Blocks

Our approach requires producing assertions that express the energy consumed by each call to an LLVM IR block (or parts of it) when it is executed. To achieve this we take as starting point the energy consumption information available from an existing XS1-L ISA Energy Model produced in our previous work of ISA level analysis [23] using the techniques described in [19]. We refer the reader to [19] for a detailed study of the energy consumption behaviour of the XS1-L architecture, containing a description of the test and measurement process along with the construction and full evaluation of such model. In the experiments performed in this paper a single, constant energy value is assigned to each instruction in the ISA based on this model.

A mechanism is then needed to propagate such ISA-level energy information up to the LLVM IR level and obtain energy values for LLVM IR blocks. A set of mapping techniques serve this purpose by creating a fine-grained mapping between segments of ISA instructions and LLVM IR code segments, in order to enable the energy characterization of each LLVM IR instruction in a program, by aggregating the energy consumption of the ISA instructions mapped to it. Then, the energy value assigned to each LLVM IR block is obtained by aggregating the energy consumption of all its LLVM IR instructions. The mapping is done by using the debug mechanism where the debug information, preserved during the lowering phase of the compilation from LLVM IR to ISA, is used to track ISA instructions against LLVM IR instructions. A full description and formalization of the mapping techniques is given in [8].

5 Resource Analysis with CiaoPP

In order to perform the global energy consumption analysis, our approach leverages the CiaoPP tool [13], the preprocessor of the Ciao programming environment [14]. CiaoPP includes a global static analyzer which is parametric with respect to resources and type of approximation (lower and upper bounds) [27,30]. The framework can be instantiated to infer bounds on a very general notion of resources, which we adapt in our case to the inference of energy consumption. As mentioned before, the resource analysis in CiaoPP works on the intermediate block-based representation language, which we have called HC IR in this paper. Each block is represented as a Horn Clause, so that, in essence, the HC IR is a pure Horn clause subset (pure logic programming subset) of the Ciao programming language. 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, number of calls to a predicate, number of accesses to a database, etc. The instantiation of the framework for energy consumption (or any other resource) is done by means of an assertion language that allows the user to define resources and other parameters of the analysis by means of assertions. Such assertions are

used to assign basic resource usage functions to elementary operations and certain program constructs of the base language, thus expressing how the execution of such operations and constructs affects the usage of a particular resource. The resource consumption provided can be a constant or a function of some input data values or sizes. The same mechanism is used as well to provide resource consumption information for procedures from libraries or external code when code is not available or to increase the precision of the analysis.

For example, in order to instantiate the CiaoPP general analysis framework for estimating bounds on energy consumption, we start by defining the identifier ("counter") associated to the energy consumption resource, through the following Ciao declaration:

```
:- resource energy.
```

We then provide assertions for each HC IR block expressing the energy consumed by the corresponding LLVM IR block, determined from the energy model, as explained in Section 4. Based on this information, the global static analysis can then infer bounds on the resource usage of the whole program (as well as procedures and functions in it) as functions of input data sizes. A full description of how this is done can be found in [30].

Consider the example in Figure 4 (right). Let P_e denote the energy consumption function for a predicate P in the HC IR representation (set of blocks with the same name). Let c_b represent the energy cost of an LLVM IR block b. Then, the inferred equations for the HC IR blocks in Figure 4 (right) are:

$$alloca_e(N, Arr) = c_{alloca} + looptest_e(N, Arr)$$

$$looptest_e(N, Arr) = c_{looptest} + loopbody\_loopend_e(0 \neq N, N, Arr)$$

$$loopbody\_loopend_e(B, N, Arr) = \begin{cases} looptest_e(N-1, Arr) \;\; \text{if } B \text{ is } \texttt{true} \\ + \; c_{loopbody} \\ \\ c_{loopend} \qquad\qquad\qquad \text{if } B \text{ is } \texttt{false} \end{cases}$$

If we assume (for simplicity of exposition) that each LLVM IR block has unitary cost, i.e., $c_b = 1$ for all LLVM IR blocks b, solving the above recurrence equations, we obtain the energy consumed by `alloca` as a function of its input data size (N):

$$alloca_e(N, Arr) = 2 \times N + 3$$

Note that using average energy 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 (possibly below it). Thus, theoretically, to ensure that the analysis infers an upper bound, we need to use upper bounds as well in the energy models. This is not a trivial task as the worst case energy consumption depends on the data processed, is likely to be different for different instructions, and unlikely to occur frequently in subsequent instructions. A first investigation into the effect of different data on the energy consumption of individual instructions, instruction sequences and full programs

is presented in [28]. A refinement of the energy model to capture upper bounds for individual instructions, or a selected subset of instructions, is currently being investigated, extending the first experiments into the impact of data into worst case energy consumption at instruction level as described in Section 5.5 of [19].

6 Experimental Evaluation

We have performed an experimental evaluation of our techniques on a number of selected benchmarks. Power measurement data was collected for the XCore platform by using appropriately instrumented power supplies, a power-sense chip, and an embedded system for controlling the measurements and collecting the power data. Details about the power monitoring setup used to run our benchmarks and measure their energy consumption can be found in [19]. The main goal of our experiments was to shed light on the trade-offs implied by performing the analysis at the ISA level (without using complex mechanisms for propagating type information and representing memory) and at the LLVM level using models defined at the ISA level together with a mapping mechanism.

There are two groups of benchmarks that we have used in our experimental study. The first group is composed of four small recursive numerical programs that have a variety of user defined functions, arguments, and calling patterns (first four benchmarks in Table 2). These benchmarks only operate over primitive data types and do not involve any structured types. The second group of benchmarks (the last five benchmarks in Table 2) differs from the first group in the sense that they all involve structured types. These are recursive or iterative.

The second group of benchmarks includes two filter benchmarks namely *Biquad* and *Finite Impulse Response (FIR)*. A filter program attenuates or amplifies one specific frequency range of a given input signal. The `fir(N)` benchmark computes the inner-product of two vectors: a vector of input samples, and a vector of coefficients. The more coefficients, the higher the fidelity, and the lower the frequencies that can be filtered. On the other hand, the Biquad benchmark is an equaliser running Biquad filtering. An equaliser takes a signal and attenuates/amplifies different frequency bands. In the case of an audio signal, such as in a speaker or microphone, this corrects the frequency response. The `biquad(N)` benchmark uses a cascade of Biquad filters where each filter attenuates or amplifies one specific frequency range. The energy consumed depends on the number of banks `N`, typically between 3 and 30 for an audio equaliser. A higher number of banks enables a designer to create more precise frequency response curves.

None of the XC benchmarks contain any assertions that provide information to help the analyzer. Table 1 shows detailed experimental results. Column **SA energy function** shows the energy consumption functions, which depend on input data sizes, inferred for each program by the static analyses performed at the ISA and LLVM IR levels (denoted with subscripts *isa* and *llvm* respectively). We can see that the analysis is able to infer different kinds of functions (polynomial, exponential, etc.). Column **HW** shows the actual energy consumption in nano-joules measured on the hardware corresponding to the execution of the

programs with input data of different sizes (shown in column **Input Data Size**). **Estimated** presents the energy consumption estimated by static analysis. This is obtained by evaluating the functions in column **SA energy function** for the input data sizes in column **Input Data Size**. The value N/A in such column means that the analysis has not been able to infer any useful energy consumption function and, thus, no estimated value is obtained. Column **Err vs. HW** shows the error of the values estimated by the static analysis with respect to the actual energy consumption measured on the hardware, calculated as follows: $\textbf{Err vs. HW} = (\frac{\textbf{LLVM}(or\ \textbf{ISA}) - \textbf{HW}}{\textbf{HW}} \times 100)\%$. Finally, the last column shows the ratio between the estimations of the analysis at the ISA and LLVM IR levels.

Table 2 shows a summary of results. The first two columns show the name and short description of the benchmarks. The columns under **Err vs. HW** show the average error obtained from the values given in Table 1 for different input data sizes. The last row of the table shows the average error over the number of benchmarks analyzed at each level.

The experimental results show that:

- For the benchmarks in the first group, both the ISA- and LLVM IR-level analyses are able to infer useful energy consumption functions. On average, the analysis performed at either level is reasonably accurate and the relative error between the two analyses at different levels is small. ISA-level estimations are slightly more accurate than the ones at the LLVM IR level (3.9% vs. 9% error on average with respect to the actual energy consumption measured on the hardware, respectively). This is because the ISA-level analysis uses very accurate energy models, obtained from measuring directly at the ISA level, whereas at the LLVM IR level, such ISA-level model needs to be propagated up to the LLVM IR level using (approximated) mapping information. This causes a slight loss of accuracy.
- For the second group of benchmarks, the ISA level analysis is not able to infer useful energy functions. This is due to the fact that significant program structure and data type/shape information is lost due to lower-level representations, which sometimes makes the analysis at the ISA level very difficult or impossible. In order to overcome this limitation and improve analysis accuracy, significantly more complex techniques for recovering type information and representing memory in the HC IR would be needed. In contrast, type/shape information is preserved at the LLVM IR level, which allows analyzing programs using data structures (e.g., arrays). In particular, all the benchmarks in the second group are analyzed at the LLVM IR level with reasonable accuracy (3% error on average). In this sense, the LLVM IR-level analysis is more powerful than the one at the ISA level. The analysis is also reasonably efficient, with analysis times of about 5 to 6 seconds on average, despite the naive implementation of the interface with external recurrence equation solvers, which can be improved significantly. The scalability of the analysis follows from the fact that it is compositional and can be performed in a modular way, making use of the Ciao assertion language to store results of previously analyzed modules.

SA energy function (nJ)	Input Size	HW (nJ)	Estimated (nJ)		Err vs. HW%		isa/ llvm
			llvm	isa	llvm	isa	
$Fact_{isa}(N)=$	N=8	227	237	212	4.6	-6.4	0.9
$24.26\ N + 18.43$	N=16	426	453	406	6.5	-4.5	0.9
$Fact_{llvm}(N)=$	N=32	824	886	794	7.6	-3.5	0.9
$27.03\ N + 21.28$	N=64	1690	1751	1571	3.6	-7.0	0.9
$Fib_{isa}(N)$[a]$=26.88 fib(N)$	N=2	75	74	65	-1.16	-12.5	0.89
$+22.85\ lucas(N)$[b]-30.04	N=4	219	241	210	10	-4.1	0.87
	N=8	1615	1853	1608	14.75	-0.4	0.87
$Fib_{llvm}(N)$[a]$=32.5 fib(N)$	N=15	47×10^3	54×10^3	47×10^3	16.47	1.2	0.87
$+25.6\ lucas(N)^b - 35.65$	N=26	9.30×10^6	10.9×10^6	9.5×10^6	17.3	1.74	0.87
$Sqr_{isa}(N)=$	N=9	1242	1302	1148	4.8	-7.5	0.88
$8.6\ N^2 + 48.7\ N + 15.6$	N=27	8135	8734	7579	7.4	-6.8	0.87
	N=73	52×10^3	57×10^3	49×10^3	8.5	-6.5	0.86
$Sqr_{llvm}(N)=$	N=144	19.7×10^4	21.4×10^4	18.4×10^4	8.89	-6.4	0.86
$10\ N^2 + 53\ N + 15.6$	N=234	51×10^4	56×10^4	48×10^4	9.61	-5.86	0.86
	N=360	11.89×10^5	13×10^5	11.2×10^5	10.49	-5.16	0.86
	N=3	326	344	3.6	5.7	-6.0	0.89
$PowerOfTwo_{isa}(N)=$	N=6	2729	2965	2631	8.7	3.6	0.89
$41.5 \times 2^N - 25.9$	N=9	21.9×10^3	23.9×10^3	21.2×10^3	9	3.3	0.89
$PowerOfTwo_{llvm}(N) =$	N=12	17.57×10^4	19.1×10^4	17×10^4	9	-3.3	0.89
$46.8 \times 2^N - 29.9$	N=15	13.8×10^5	15.3×10^5	13.6×10^5	11	-1.5	0.89
	N=57	1138	1179	N/A	3.60	N/A	N/A
$reverse_{llvm}(N)=$	N=160	3125	3185	N/A	1.91	N/A	N/A
$19.47\ N + 69.33$	N=320	6189	6301	N/A	1.82	N/A	N/A
	N=720	13848	14092	N/A	1.76	N/A	N/A
	N=1280	24634	24998	N/A	1.48	N/A	N/A
	N=5	7453	7569	N/A	-2	N/A	N/A
$matmult_{llvm}(N)=$	N=15	15.79×10^4	15.9×10^4	N/A	1.03	N/A	N/A
$42.47\ N^3 + 68.85\ N^2 +$	N=20	36.29×10^4	36.8×10^4	N/A	1.51	N/A	N/A
$49.9\ N + 24.22$	N=25	69.56×10^4	70.8×10^4	N/A	1.77	N/A	N/A
	N=31	13.07×10^5	13.3×10^5	N/A	1.98	N/A	N/A
	N=131; M=69	14.5×10^3	13.2×10^3	N/A	8.65	N/A	N/A
$concat_{llvm}(N, M)=$	N=170; M=182	25.44×10^3	23.3×10^3	N/A	8.60	N/A	N/A
$65.7\ N + 65.7\ M + 137$	N=188; M=2	13.8×10^3	12.6×10^3	N/A	8.59	N/A	N/A
	N=13; M=134	10.7×10^3	9.79×10^3	N/A	8.74	N/A	N/A
$biquad_{llvm}(N)=$	N=5	871	836	N/A	-4	N/A	N/A
$157\ N + 51.7$	N=7	1187	1151	N/A	-3.1	N/A	N/A
	N=10	1660	1622	N/A	-2.31	N/A	N/A
	N=14	2290	2250	N/A	-1.75	N/A	N/A
$fir_{llvm}(N)=$	N=85	2999	2839	N/A	-5.3	N/A	N/A
$31.8\ N + 137$	N=97	3404	3221	N/A	-5.37	N/A	N/A
	N=109	3812	3602	N/A	-5.5	N/A	N/A
	N=121	4227	3984	N/A	-5.7	N/A	N/A

Table 1: Comparison of the accuracy of energy analyses at the LLVM IR and ISA levels.

[a] It uses mathematical functions fib and $lucas$, a function expansion would yield:
$Fib_{isa}(N)=34.87 \times 1.62^N + 10.8 \times (-0.62)^N - 30$
$Fib_{llvm}(N)=40.13 \times 1.62^N + 11.1 \times (-0.62)^N - 35.65$

[b] $Lucas(n)$ satisfy the recurrence relation $L_n = L_{n-1} + L_{n-2}$ with $L_1 = 1, L_2 = 3$

Program	Description	Err vs. HW		isa/
		llvm	isa	llvm
`fact(N)`	Calculates N!	5.6%	5.3%	0.89
`fibonacci(N)`	Nth Fibonacci number	11.9%	4%	0.87
`sqr(N)`	Computes N^2 performing additions	9.3%	3.1%	0.86
`pow_of_two(N)`	Calculates 2^N without multiplication	9.4%	3.3%	0.89
`Average`		**9%**	**3.9%**	**0.92**
`reverse(N, M)`	Reverses an array	2.18%	N/A	N/A
`concat(N, M)`	Concatenation of arrays	8.71%	N/A	N/A
`matmult(N, M)`	Matrix multiplication	1.47%	N/A	N/A
`fir(N)`	Finite Impulse Response filter	5.47%	N/A	N/A
`biquad(N)`	Biquad equaliser	3.70%	N/A	N/A
`Average`		**3.0%**	**N/A**	**N/A**
`Overall average`		**6.4%**	**3.9%**	**0.92**

Table 2: LLVM IR- vs. ISA-level analysis accuracy.

7 Related Work

Few papers can be found in the literature focusing on static analysis of energy consumption. As mentioned before, the approach presented in this paper builds on our previously developed analysis of XC programs [23] based on transforming the corresponding ISA code into a Horn Clause representation that is supplied, together with an ISA-level energy model, to the CiaoPP [13] resource analyzer. In this work we have increased the power of the analysis by transforming and analyzing the corresponding LLVM IR, and using techniques for reflecting the ISA-level energy model upwards to the LLVM IR level. We also offer novel results supported by our experimental study that shed light on the trade-offs implied by performing the analysis at each of these two levels. Our approach now enables the analysis of a wider range of benchmarks. We obtained promising results for a good number of benchmarks for which [23] was not able to produce useful energy functions. A similar approach was proposed for upper-bound energy analysis of Java bytecode programs in [26], where the Jimple (a typed three-address code) representation of Java bytecode was transformed into Horn Clauses, and a simple energy model at the Java bytecode level [20] was used. However, this work did not compare the results with actual, measured energy consumption.

In all the approaches mentioned above, instantiations for energy consumption of general resource analyzers are used, namely [27] in [26] and [23], and [30] in this paper. Such resource analyzers are based on setting up and solving recurrence equations, an approach proposed by Wegbreit [34] that has been developed significantly in subsequent work [29,6,7,32,27,1,30]. Other approaches to static analysis based on the transformation of the analyzed code into another (intermediate) representation have been proposed for analyzing low-level languages [12] and Java (by means of a transformation into Java bytecode) [2]. In [2], cost relations are inferred directly for these bytecode programs, whereas in [26] the bytecode is first transformed into Horn Clauses. The general resource analyzer in [27] was also instantiated in [25] for the estimation of execution times of

logic programs running on a bytecode-based abstract machine. The approach used timing models at the bytecode instruction level, for each particular platform, and program-specific mappings to lift such models up to the Horn Clause level, at which the analysis was performed. The timing model was automatically produced in a one-time, program-independent profiling stage by using a set of synthetic calibration programs and setting up a system of linear equations.

By contrast to the generic approach based on CiaoPP, an approach operating directly on the LLVM IR representation is explored in [10]. Though relying on similar analysis techniques, the approach can be integrated more directly in the LLVM toolchain and is in principle applicable to any languages targeting this toolchain. The approach uses the same LLVM IR energy model and mapping technique as the one applied in this paper.

There exist other approaches to cost analysis such as those using dependent types [16], SMT solvers [3], or *size change abstraction* [36]

A number of static analyses are also aimed at worst case execution time (WCET), usually for imperative languages in different application domains (see e.g., [35] and its references). The worst-case analysis presented in [18], which is not based on recurrence equation solving, distinguishes instruction-specific (not proportional to time, but to data) from pipeline-specific (roughly proportional to time) energy consumption. However, in contrast to the work presented here and in [25], these worst case analysis methods do not infer cost functions on input data sizes but rather absolute maximum values, and they generally require the manual annotation of loops to express an upper-bound on the number of iterations. An alternative approach to WCET was presented in [15]. It is based on the idea of amortisation, which allows to infer more accurate yet safe upper bounds by averaging the worst execution time of operations over time. It was applied to a functional language, but the approach is in principle generally applicable. A timing analysis based on game-theoretic learning was presented in [31]. The approach combines static analysis to find a set of basic paths which are then tested. In principle, such approach could be adapted to infer energy usage. Its main advantage is that this analysis can infer distributions on time, not only average values.

8 Conclusions and Future Work

We have presented techniques for extending to the LLVM IR level our tool chain for estimating energy consumption as functions on program input data sizes. The approach uses a mapping technique that leverages the existing debugging mechanisms in the XMOS XCore compiler tool chain to propagate an ISA-level energy model to the LLVM IR level. A new transformation constructs a block representation that is supplied, together with the propagated energy values, to a parametric resource analyzer that infers the program energy cost as functions on the input data sizes.

Our results suggest that performing the static analysis at the LLVM IR level is a reasonable compromise, since 1) LLVM IR is close enough to the source code

level to preserve most of the program information needed by the static analysis, and 2) the LLVM IR is close enough to the ISA level to allow the propagation of the ISA energy model up to the LLVM IR level without significant loss of accuracy for the examples studied. Our experiments are based on single-threaded programs. We also have focused on the study of the energy consumption due to computation, so that we have not tested programs where storage and networking is important. However, this could potentially be done in future work, by using the CiaoPP static analysis, which already infers bounds on data sizes, and combining such information with appropriate energy models of communication and storage. Although the analysis infers sound bound representations in the form of recurrence equations, sometimes the external solvers it uses are not able to find closed form functions for such equations. This is a limitation in applications where such closed forms are needed. Techniques to address such limitation are included in our plans for future work. Our static analysis will also benefit from any improvement of the Computer Algebra Systems used for solving recurrence equations.

It remains to be seen whether the results would carry over to other classes of programs, such as multi-threaded programs and programs where timing is more important. In this sense our results are preliminary, yet they are promising enough to continue research into analysis at LLVM IR level and into ISA-LLVM IR energy mapping techniques to enable the analysis of a wider class of programs, especially multi-threaded programs.

Acknowledgements

This research has received funding from the European Union 7th Framework Program agreement no 318337, ENTRA, Spanish MINECO TIN'12-39391 *StrongSoft* project, and the Madrid M141047003 *N-GREENS* program.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In R. D. Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.
3. D. E. Alonso-Blas and S. Genaim. On the Limits of the Classical Approach to Cost Analysis. In A. Miné and D. Schmidt, editors, *19th International Symposium on Static Analysis, SAS 2012*, volume 7460 of *LNCS*, pages 405–421. Springer, 2012.
4. N. Bjørner, F. Fioravanti, A. Rybalchenko, and V. Senni, editors. *Proceedings of First Workshop on Horn Clauses for Verification and Synthesis*, volume 169 of *EPTCS*, July 2014.
5. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis*

of Systems, 14th International Conference, TACAS 2008, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
6. 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.
7. 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.
8. K. Georgiou, S. Kerrison, and K. Eder. On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs. *ArXiv e-prints:1510.07095*, Oct. 2015.
9. S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution). In C. Flanagan and B. König, editors, *TACAS*, volume 7214 of *LNCS*, pages 549–551. Springer, 2012.
10. N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder. Static analysis of energy consumption for LLVM IR programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, New York, NY, USA, 2015. ACM.
11. A. Gurfinkel, T. Kahsai, and J. A. Navas. Seahorn: A framework for verifying C programs (competition contribution). In *Proc. of TACAS 2015*, volume 9035 of *LNCS*, pages 447–450. Springer, 2015.
12. 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.
13. M. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
14. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012.
15. C. Herrmann, A. Bonenfant, K. Hammond, S. Jost, H.-W. Loidl, and R. Pointon. Automatic Amortised Worst-Case Execution Time Analysis. In *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OASIcs*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007.
16. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.
17. H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A Verification Toolkit for Numerical Transition Systems - Tool Paper. In *Proc. of FM 2012*, volume 7436 of *LNCS*, pages 247–251. Springer, 2012.
18. 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.
19. S. Kerrison and K. Eder. Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Transactions on Embedded Computing Systems*, 14(3):1–25, April 2015.
20. S. Lafond and J. Lilius. Energy consumption analysis for two embedded Java virtual machines. *J. Syst. Archit.*, 53(5-6):328–337, 2007.

21. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 75–88. IEEE Computer Society, March 2004.
22. U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Energy Consumption at Different Software Levels: ISA vs. LLVM IR. Technical report, FET 318337 ENTRA Project, April 2014. Appendix D3.2.4 of Deliverable D3.2. Available at `http://entraproject.eu`.
23. U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 72–90. Springer, 2014.
24. M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in LNCS, pages 154–168. Springer-Verlag, August 2007.
25. E. Mera, P. López-García, M. Carro, and M. Hermenegildo. Towards Execution Time Estimation in Abstract Machine-Based Languages. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press, July 2008.
26. 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.
27. 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.
28. J. Pallister, S. Kerrison, J. Morse, and K. Eder. Data dependent energy modeling for worst case energy consumption analysis. *arXiv preprint arXiv:1505.03374*, 2015.
29. M. Rosendahl. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*. ACM Press, 1989.
30. 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.
31. S. A. Seshia and J. Kotker. Gametime: A toolkit for timing analysis of software. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 388–392. Springer, 2011.
32. P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *15th International Workshop on Implementation of Functional Languages (IFL'03), Revised Papers*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, Sep 2005.
33. D. Watt. *Programming XC on XMOS Devices.* XMOS Limited, 2009.
34. B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.
35. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - Overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

36. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction (extended version). *CoRR*, abs/1203.5303, 2012.