Towards Energy Consumption Verification via Static Analysis

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

R. Haemmerlé [†]
remy.haemmerle@imdea.org

M. Klemen [†]
maximiliano.klemen@imdea.org

U. Liqat [†]
umer.liqat@imdea.org

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

ABSTRACT

In this paper we leverage an existing general framework for resource usage verification and specialize it for *verifying* energy consumption specifications of embedded programs. Such specifications can include both lower and upper bounds on energy usage, and they can express intervals within which energy usage is to be certified to be within such bounds. The bounds of the intervals can be given in general as functions on input data sizes. Our verification system can prove whether such energy usage specifications are met or not. It can also infer the particular conditions under which the specifications hold. To this end, these conditions are also expressed as intervals of functions of input data sizes, such that a given specification can be proved for some intervals but disproved for others. The specifications themselves can also include preconditions expressing intervals for input data sizes. We report on a prototype implementation of our approach within the CiaoPP system for the XC language and XS1-L architecture, and illustrate with an example how embedded software developers can use this tool, and in particular for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service.

Keywords: Energy consumption analysis and verification, resource usage analysis and verification, static analysis, verification.

1. INTRODUCTION

In an increasing number of applications, particularly those running on devices with limited resources, it is very important and sometimes essential to ensure conformance with respect to specifications expressing non-functional global properties such as energy consumption, maximum execution time, memory usage, or user-defined resources. For example, in a real-time application, a program completing an action later than required is as erroneous as a program not computing the correct answer. The same applies to an embedded application in a battery-operated device (e.g., a portable or implantable medical device, an autonomous space vehicle, or even a mobile phone) if the application makes the device run out of batteries earlier than required, making the whole system useless in practice.

In general, high performance embedded systems must control, react to, and survive in a given environment, and this in turn establishes constraints about the system's performance parameters including energy consumption and reaction times. Therefore, a mechanism is necessary in these systems in order to prove correctness with respect to specifications about such non-functional global properties.

To address this problem we leverage an existing general framework for resource usage analysis and verification [22, 23], and specialize it for *verifying* energy consumption specifications of embedded programs. As a case study, we focus on the energy verification of embedded programs written in the XC language [37] and running on the XMOS XS1-L architecture (XC is a high-level C-based programming language that includes extensions for communication, input/output operations, real-time behavior, and concurrency). However, the approach presented here can also be applied to the analysis of other programming languages and architectures. We will illustrate with an example how embedded software developers can use this tool, and in particular for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service.

2. OVERVIEW OF THE ENERGY VERIFICATION TOOL

In this section we give an overview of the prototype tool for *energy consumption verification* of XC programs running on the XMOS XS1-L architecture, which we have implemented within the CiaoPP system [13]. As in previous work [20, 25], we differentiate between the *input language*, which can be XC source, LLVM IR [17], or Instruction Set Architecture (ISA) code, and the *intermediate semantic program representation* that the CiaoPP core components (e.g., the analyzer) take as input. 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 perform a transformation from each *input language* into the HC IR and pass it to the corresponding CiaoPP component. 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 [25]. 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

[*]Spanish Council for Scientific Research (CSIC).
[†]IMDEA Software Institute, Madrid, Spain.
[‡]Universidad Politécnica de Madrid (UPM).

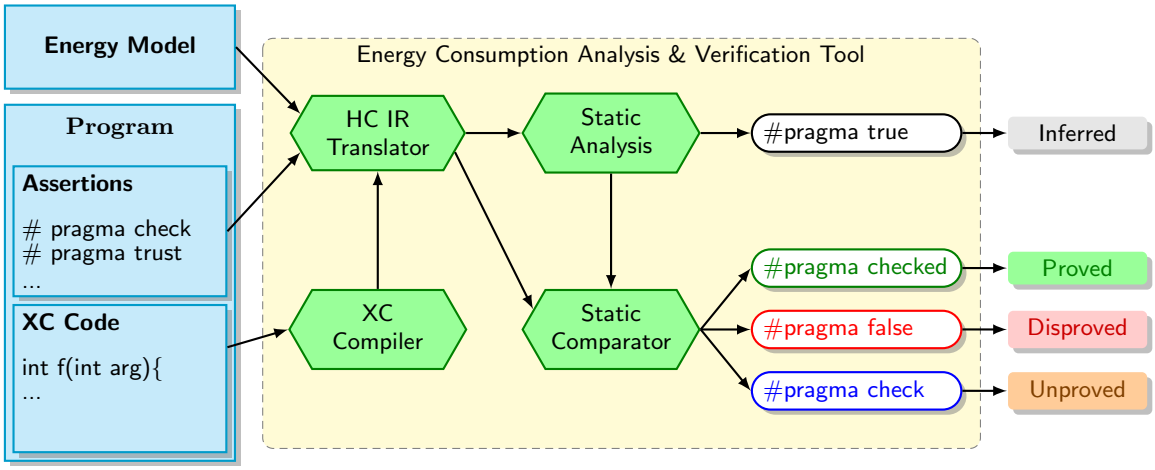

Figure 1: Energy consumption verification tool using CiaoPP.

as intermediate representations in analysis and verification tools [2].

Figure 1 shows an overview diagram of the architecture of the prototype tool we have developed. Hexagons represent different tool components and arrows indicate the communication paths among them.

The tool takes as input an XC source program (left part of Figure 1) that can optionally contain assertions in a C-style syntax. As we will see later, such assertions are translated into Ciao assertions, the internal representation used in the Ciao/CiaoPP system.

The energy specifications that the tool will try to prove or disprove are expressed by means of assertions with `check` *status*. These specifications can include both lower and upper bounds on energy usage, and they can express intervals within which energy usage is to be certified to be within such bounds. The bounds of the intervals can be given in general as functions on input data sizes. Our tool can prove whether such energy usage specifications are met or not. It can also infer the particular conditions under which the specifications hold. To this end, these conditions are also expressed as intervals of functions of input data sizes, such that a given specification can be proved for some intervals but disproved for others.

In addition, assertions can also express trusted information such as the energy usage of procedures that are not developed yet, or useful hints and information to the tool. In general, assertions with status `trust` 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.

In our tool the user can choose between performing the analysis at the ISA or LLVM IR levels (or both). We refer the reader to [19] for an experimental study that sheds light on the trade-offs implied by performing the analysis at each of these two levels, which can help the user to choose the level that fits the problem best.

The associated ISA and/or LLVM IR representations of the XC program are generated using the xcc compiler. Such representations include useful metadata. The *HC IR translator* component (described in Section 4) produces the internal representation used by the tool, HC IR, which includes the program and possibly specifications and/or trusted information (expressed in the Ciao assertion language [32, 15]).

The tool performs several tasks:

1. Transforming the ISA and/or LLVM IR into HC IR. Such transformation preserves the resource consumption semantics, in the sense that the resource usage information inferred by the tool is applicable to the original XC program.

2. Transforming specifications (and trusted information) written as C-like assertions into the Ciao assertion language.

3. Transforming the energy model at the ISA level [16], expressed in JSON format, into the Ciao assertion language. Such assertions express the energy consumed by individual ISA instruction representations, information which is required by the analyzer in order to propagate it during the static analysis of a program through code segments, conditionals, loops, recursions, etc., in order to infer analysis information (energy consumption functions) for higher-level entities such as procedures, functions, or loops in the program.

4. In the case that the analysis is performed at the LLVM IR level, the *HC IR translator* component produces a set of Ciao assertions expressing the energy consumption corresponding to LLVM IR block representations in HC IR. Such information is produced from a mapping of LLVM IR instructions with sequences of ISA instructions and the ISA-level energy model. The mapping information is produced by the *mapping tool* that was first outlined in [3] (Section 2 and Attachments D3.2.4 and D3.2.5) and is described in detail in [11].

Then, following the approach described in [20], the CiaoPP parametric static resource usage analyzer [30, 28, 34] takes

the HC IR, together with the assertions which express the energy consumed by LLVM IR blocks and/or individual ISA instructions, and possibly some additional (trusted) information, and processes them, producing the analysis results, which are expressed also using Ciao assertions. Such results include energy usage functions (which depend on input data sizes) for each block in the HC IR (i.e., for the whole program and for all the procedures and functions in it.). The analysis can infer different types of energy functions (e.g., polynomial, exponential, or logarithmic). The procedural interpretation of the HC IR programs, coupled with the resource-related information contained in the (Ciao) assertions, together allow the resource analysis to 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. Analysis results are given using the assertion language, to ensure interoperability and make them understandable by the programmer.

The verification of energy specifications is performed by a specialized component which compares the energy specifications with the (safe) approximated information inferred by the static resource analysis. Such component is based on our previous work on general resource usage verification presented in [22, 23], where we extended the criteria of correctness as the conformance of a program to a specification expressing non-functional global properties, such as upper and lower bounds on execution time, memory, energy, or user defined resources, given as functions on input data sizes. We also defined an abstract semantics for resource usage properties and operations to compare the (approximated) intended semantics of a program (i.e., the specification) with approximated semantics inferred by static analysis. These operations include the comparison of arithmetic functions (e.g., polynomial, exponential, or logarithmic functions) that may come from the specifications or from the analysis results. As a possible result of the comparison in the output of the tool, either:

1. The original (specification) assertion (i.e., with status `check`) is included with status `checked` (resp. `false`), meaning that the assertion is correct (resp. incorrect) for all input data meeting the precondition of the assertion,

2. the assertion is "split" into two or three assertions with different status (`checked`, `false`, or `check`) whose preconditions include a conjunct expressing that the size of the input data belongs to the interval(s) for which the assertion is correct (status `checked`), incorrect (status `false`), or the tool is not able to determine whether the assertion is correct or incorrect (status `check`), or

3. in the worst case, the assertion is included with status `check`, meaning that the tool is not able to prove nor to disprove (any part of) it.

If all assertions are `checked` then the program is *verified*. Otherwise, for assertions (or parts of them) that get `false` status, a *compile-time error* is reported. Even if a program contains no assertions, it can be checked against the assertions contained in the libraries used by the program, potentially catching bugs at compile time. Finally, and most importantly, for assertion (or parts of them) left with status `check`, the tool can optionally produce a *verification warning* (also referred to as an "alarm"). In addition, optional run-time checks can also be generated.

3. THE ASSERTION LANGUAGE

Two aspects of the assertion language are described here: the front-end language in which assertions are written and included in the XC programs to be verified, and the internal language in which such assertions are translated into and passed, together with the HC IR program representation, to the core analysis and verification tools, the Ciao assertion language.

3.1 The Ciao Assertion Language

We describe here the subset of the Ciao assertion language which allows expressing global "computational" properties and, in particular, resource usage. We refer the reader to [32, 13, 15] and their references for a full description of this assertion language.

For brevity, we only introduce here the class of **`pred` assertions**, which describes a particular predicate and, in general, follows the schema:

`:- pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-Props*]`.`

where *Pred* is a predicate symbol applied to distinct free variables while *Precond* and *Postcond* are logic formulae about execution states. An execution state is defined by variable/value bindings in a given execution step. The assertion indicates that in any call to *Pred*, if *Precond* holds in the calling state and the computation of the call succeeds, then *Postcond* also holds in the success state. Finally, the *Comp-Props* field is used to describe properties of the whole computation for calls to predicate *Pred* that meet *Precond*. In our application *Comp-Props* are precisely the resource usage properties.

For example, the following assertion for a typical `append/3` predicate:

```
:- pred append(A,B,C)
   : (list(A,num),list(B,num),var(C))
  => (list(C,num),
      rsize(A,list(ALb,AUb,num(ANl,ANu))),
      rsize(B,list(BLb,BUb,num(BNl,BNu))),
      rsize(C,
       list(ALb+BLb,AUb+BUb,
         num(min(ANl,BNl),max(ANu,BNu)))))
   + resource(steps,ALb+1, AUb+1).
```

states that for any call to predicate `append/3` with the first and second arguments bound to lists of numbers, and the third one unbound, if the call succeeds, then the third argument will also be bound to a list of numbers. It also states that an upper bound on the number of resolution steps required to execute any of such calls is $AUb + 1$, a function on the length of list A. The `rsize` terms are the *sized types* derived from the regular types, containing variables that represent explicitly lower and upper bounds on the size of terms and subterms appearing in arguments. See Section 5 for an overview of the general resource analysis framework and how sized types are used.

The global non-functional property `resource/3` (appearing in the "`+`" field), is used for expressing resource usages and follows the schema:

`resource(`*Res_Name***`,`** *Low_Arith_Expr, Upp_Arith_Expr)*

where *Res_Name* is a user-provided identifier for the resource the assertion refers to, *Low_Arith_Expr* and *Upp_Arith_Expr* are arithmetic functions that map input data sizes to re-

source usage, representing respectively lower and upper bounds on the resource consumption.

Each assertion can be in a particular *status*, marked with the following prefixes, placed just before the `pred` keyword: `check` (indicating the assertion needs to be checked), `checked` (it has been checked and proved correct by the system), `false` (it has been checked and proved incorrect by the system; a compile-time error is reported in this case), `trust` (it provides information coming from the programmer and needs to be trusted), or `true` (it is the result of static analysis and thus correct, i.e., safely approximated). The default status (i.e., if no status appears before `pred`) is `check`.

3.2 The XC Assertion Language

The assertions within XC files use instead a different syntax that is closer to standard C notation and friendlier for C developers. These assertions are transparently translated into Ciao assertions when XC files are loaded into the tool. The Ciao assertions output by the analysis are also translated back into XC assertions and added inline to a copy of the original XC file.

More concretely, the syntax of the XC assertions accepted by our tool is given by the following grammar, where the non-terminal ⟨*identifier*⟩ stands for a standard C identifier, ⟨*integer*⟩ stands for a standard C integer, and the non-terminal ⟨*ground-expr*⟩ for a ground expression, i.e., an expression of type ⟨*expr*⟩ that does not contain any C identifiers that appear in the assertion scope (the non-terminal ⟨*scope*⟩).

⟨*assertion*⟩ ::= '`#pragma`' ⟨*status*⟩ ⟨*scope*⟩ '`:`' ⟨*body*⟩

⟨*status*⟩ ::= '`check`' | '`trust`' | '`true`' | '`checked`' | '`false`'

⟨*scope*⟩ ::= ⟨*identifier*⟩ '`(`' '`)`'
| ⟨*identifier*⟩ '`(`' ⟨*arguments*⟩ '`)`'

⟨*arguments*⟩ ::= ⟨*identifier*⟩ | ⟨*arguments*⟩ '`,`' ⟨*identifier*⟩

⟨*body*⟩ ::= ⟨*precond*⟩ '`==>`' ⟨*cost-bounds*⟩ | ⟨*cost-bounds*⟩

⟨*precond*⟩ ::= ⟨*upper-cond*⟩ | ⟨*lower-cond*⟩
| ⟨*lower-cond*⟩ '`&&`' ⟨*upper-cond*⟩

⟨*lower-cond*⟩ ::= ⟨*ground-expr*⟩ '`<=`' ⟨*identifier*⟩

⟨*upper-cond*⟩ ::= ⟨*identifier*⟩ '`<=`' ⟨*ground-expr*⟩

⟨*cost-bounds*⟩ ::= ⟨*lower-bound*⟩ | ⟨*upper-bound*⟩
| ⟨*lower-bound*⟩ '`&&`' ⟨*upper-bound*⟩

⟨*lower-bound*⟩ := ⟨*expr*⟩ '`<=`' '`energy`'

⟨*upper-bound*⟩ := '`energy`' '`<=`' ⟨*expr*⟩

⟨*expr*⟩ := ⟨*expr*⟩ '`+`' ⟨*mult-expr*⟩
| ⟨*expr*⟩ '`-`' ⟨*mult-expr*⟩

⟨*mult-expr*⟩ := ⟨*mult-expr*⟩ '`*`' ⟨*unary-expr*⟩
| ⟨*mult-expr*⟩ '`/`' ⟨*unary-expr*⟩

⟨*unary-expr*⟩ := ⟨*identifier*⟩
| ⟨*integer*⟩
| '`sum`' '`(`' ⟨*identifier*⟩ '`,`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ '`)`'
| '`prod`' '`(`' ⟨*identifier*⟩ '`,`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩'`,`' ⟨*expr*⟩ '`)`'
| '`power`' '`(`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ '`)`'
| '`log`' '`(`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ '`)`'
| '`(`' ⟨*expr*⟩ '`)`'
| '`+`' ⟨*unary-expr*⟩
| '`-`' ⟨*unary-expr*⟩
| '`min`' '`(`' ⟨*identifier*⟩ '`)`'
| '`max`' '`(`' ⟨*identifier*⟩ '`)`'

XC assertions are directives starting with the token `#pragma` followed by the assertion *status*, the assertion *scope*, and the assertion *body*. The assertion *status* can take several values, including `check`, `checked`, `false`, `trust` or `true`, with the same meaning as in the Ciao assertions. Again, the default status is `check`.

The assertion scope identifies the function the assertion is referring to, and provides the local names for the arguments of the function to be used in the body of the assertion. For instance, the scope `biquadCascade(state, xn, N)` refers to the function `biquadCascade` and binds the arguments within the body of the assertion to the respective identifiers `state`, `xn`, `N`. While the arguments do not need to be named in a consistent way w.r.t. the function definition, it is highly recommended for the sake of clarity. The *body* of the assertion expresses bounds on the energy consumed by the function and optionally contains preconditions (the left hand side of the `==>` arrow) that constrain the argument sizes.

Within the body, expressions of type ⟨*expr*⟩ are built from standard integer arithmetic functions (i.e., `+`, `-`, `*`, `/`) plus the following extra functions:

- `power(base, exp)` is the exponentiation of `base` by `exp`;
- `log(base, expr)` is the logarithm of `expr` in base `base`;
- `sum(id, lower, upper, expr)` is the summation of the sequence of the values of `expr` for `id` ranging from `lower` to `upper`;
- `prod(id, lower, upper, expr)` is the product of the sequence of the values of `expr` for `id` ranging from `lower` to `upper`;
- `min(arr)` is the minimal value of the array `arr`;
- `max(arr)` is the maximal value of the array `arr`.

Note that the argument of `min` and `max` must be an identifier appearing in the assertion scope that corresponds to an array of integers (of arbitrary dimension).

4. ISA/LLVM IR TO HC IR TRANSFORMATION

In this section we describe briefly the HC IR representation and the transformations into it that we developed in order to achieve the verification tool presented in Section 2 and depicted in Figure 1. The transformation of ISA code into HC IR was described in [21]. We provide herein an overview of the LLVM IR to HC IR transformation.

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.

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 represent each of the basic blocks of the LLVM IR in the HC IR, we follow a similar approach as in the ISA-level transformation [21]. However, the LLVM IR includes an additional type transformation as well as better memory modelling. It is explained in detail in Appendix 5 of [3]. The main aspects of this process, are the following:

1. Infer input/output parameters to each block.
2. Transform LLVM IR types into HC IR types.
3. Represent each LLVM IR block as an HC IR block and each instruction in the LLVM IR block as a literal (S_i).
4. Resolve branching to multiple blocks by creating clauses with the same signature (i.e., the same name and arguments in the head), where each clause denotes one of the blocks the branch may jump to.

The translator component is also in charge of translating the XC assertions to Ciao assertions and back. Assuming the Ciao type of the input and output of the function is known, the translation of assertions from Ciao to XC (and back) is relatively straightforward. The *Pred* field of the Ciao assertion is obtained from the scope of the XC assertion to which an extra argument is added representing the output of the function. The *Precond* fields are produced directly from the type of the input arguments: to each input variable, its regular type and its regular type size are added to the precondition, while the added output argument is declared as a free variable. Finally the *Comp-Props* field is set to the usage of the resource `energy`, i.e., a literal of the form `resource(energy, Lower, Upper)` where `Lower` and `Upper` are the lower and upper bounds from the energy consumption specification.

5. ENERGY CONSUMPTION ANALYSIS

As already mentioned in Section 2, we use an existing static analysis to infer the energy consumption of XC programs [21]. It is a specialization of the generic resource analysis presented in [35] that uses the instruction-level models described in [16]. Such generic resource analysis is fully based on *abstract interpretation* [9], defining the resource analysis itself as an *abstract domain* that is integrated into the PLAI abstract interpretation framework [27, 33] of CiaoPP, obtaining features such as *multivariance*, efficient fixpoints, and assertion-based verification and user interaction.

In the rest of this section we give an overview of the general resource analysis, using the following `append/3` predicate as a running example:

```
append([],    S, S).
append([E|R], S, [E|T]) :- append(R,S,T).
```

The first step consists of obtaining the regular type of the arguments for each predicate. To this end, we use one of the type analyses present in the CiaoPP system [36]. In our example, the system infers that for any call to the predicate `append(X, Y, Z)` with `X` and `Y` bound to lists of numbers and `Z` a free variable, if the call succeeds, then `Z` also gets bound to a list of numbers. The regular type for representing "list of numbers" is defined as follows:

```
listnum := [] | [num | listnum].
```

From this type definition, sized type schemas are derived, which incorporate variables representing explicitly lower and upper bounds on the size of terms and subterms. For example, in the following sized type schema (named *listnum-s*):

$$listnum\text{-}s \rightarrow listnum^{(\alpha,\beta)}(num^{(\gamma,\delta)})$$

α and β represent lower and upper bounds on the length of the list, respectively, while γ and δ represent lower and upper bounds of the numbers in the list, respectively.

In a subsequent phase, these sized type schemas are put into relation, producing a system of recurrence equations where output argument sizes are expressed as functions of input argument sizes.

The resource analysis is in fact an extension of the sized type analysis that adds recurrence equations for each resource. As the HC IR representation is a logic program, it is necessary to consider that a predicate can fail or have more than one solution, so we need an auxiliary *cardinality analysis* to get more precise results.

We develop the `append` example for the simple case of the resource being the number of resolution steps performed by a call to `append/3` and we will only focus on upper bounds, r_U. For the first clause, we know that only one resolution step is needed, so:

$$r_U\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\right) \leq 1$$

The second clause performs one resolution step plus all the resolution steps performed by all possible backtrackings over the call in the body of the clause. This number can be bounded as a function of the number of solutions. After setting up and solving these equations we infer that an upper bound on the number of resolution steps is the (upper bound on) the length of the input list `X` plus one. This is expressed as:

$$r_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\right) \leq \beta_X + 1$$

We refer the reader to [35] for a full description of this analysis and tool.

6. THE GENERAL RESOURCE USAGE VERIFICATION FRAMEWORK

In this section we describe the general framework for (static) resource usage *verification* [22, 24] that we have specialized in this paper for verifying energy consumption specifications of XC programs.

The framework, that we introduced in [22], extends the criteria of correctness as the conformance of a program to a specification expressing non-functional global properties, such as upper and lower bounds on execution time, memory, energy, or user defined resources, given as functions on input data sizes.

Both program verification and debugging compare the *actual semantics* $[\![P]\!]$ of a program P with an *intended semantics* for the same program, which we will denote by I. This intended semantics embodies the user's requirements, i.e., it is an expression of the user's expectations. In the framework, both semantics are given in the form of (*safe*) approximations. The abstract (safe) approximation $[\![P]\!]_\alpha$ of the concrete semantics $[\![P]\!]$ of the program is actually computed by (abstract interpretation-based) *static analyses*, and compared directly to the (also approximate) specification, which is safely assumed to be also given as an abstract value I_α. Such approximated specification is expressed by *assertions* in the program. Program verification is then performed by comparing I_α and $[\![P]\!]_\alpha$.

In this paper, we assume that the program P is in HC IR form (i.e., a logic program), which is the result of the transformation of the ISA or LLVM IR code corresponding to an XC program. As already said, such transformation preserves the resource consumption semantics, in the sense that the resource usage information inferred by the static analysis (and hence the result of the verification process) is applicable to the original XC program.

Resource usage semantics.

Given a program p, let $\mathcal{C}_p$ be the set of all calls to p. The concrete resource usage semantics of a program p, for a particular resource of interest, $[\![P]\!]$, is a set of pairs $(p(\bar{t}), r)$ such that $\bar{t}$ is a tuple of data (either simple data such as numbers, or compound data structures), $p(\bar{t}) \in \mathcal{C}_p$ is a call to procedure[1] p with actual parameters $\bar{t}$, and r is a number expressing the amount of resource usage of the computation of the call $p(\bar{t})$. The concrete resource usage semantics can be defined as a function $[\![P]\!] : \mathcal{C}_p \mapsto \mathcal{R}$ where $\mathcal{R}$ is the set of real numbers (note that depending on the type of resource we can take other set of numbers, e.g., the set of natural numbers).

The abstract resource usage semantics is a set of 4-tuples:

$$(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$$

where $p(\bar{v}) : c(\bar{v})$ is an abstraction of a set of calls. $\bar{v}$ is a tuple of variables and $c(\bar{v})$ is an abstraction representing a set of tuples of data which are instances of $\bar{v}$. $c(\bar{v})$ is an element of some abstract domain expressing instantiation states. Φ is an abstraction of the resource usage of the calls represented by $p(\bar{v}) : c(\bar{v})$. We refer to it as a *resource usage interval function* for p, defined as follows:

- A *resource usage bound function* for p is a monotonic arithmetic function, $\Psi : S \mapsto \mathcal{R}_\infty$, for a given subset $S \subseteq \mathcal{R}^k$, where $\mathcal{R}$ is the set of real numbers, k is the number of input arguments to procedure p and $\mathcal{R}_\infty$ is the set of real numbers augmented with the special symbols ∞ and $-\infty$. We use such functions to express lower and upper bounds on the resource usage of procedure p depending on input data sizes.

- A *resource usage interval function* for p is an arithmetic function, $\Phi : S \mapsto \mathcal{RI}$, where S is defined as before and $\mathcal{RI}$ is the set of intervals of real numbers, such that $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$ for all $\bar{n} \in S$, where $\Phi^l(\bar{n})$ and $\Phi^u(\bar{n})$ are *resource usage bound functions* that denote the lower and upper endpoints of the interval $\Phi(\bar{n})$ respectively for the tuple of input data sizes $\bar{n}$. Although $\bar{n}$ is typically a tuple of natural numbers, we do not want to restrict our framework. We require that Φ be well defined so that $\forall \bar{n}\ (\Phi^l(\bar{n}) \leq \Phi^u(\bar{n}))$.

[1]Also called *predicate* in the HC IR.

$input_p$ is a function that takes a tuple of data $\bar{t}$ and returns a tuple with the input arguments to p. This function can be inferred by using the existing mode analysis or be given by the user by means of assertions. $size_p(\bar{t})$ is a function that takes a tuple of terms $\bar{t}$ and returns a tuple with the sizes of those data under the size metric described in Section 5.

In order to make the presentation simpler, we will omit the $input_p$ and $size_p$ functions in abstract tuples, with the understanding that they are present in all such tuples.

Intended meaning.

The intended approximated meaning I_α of a program is an abstract semantic object with the same kind of tuples: $(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$, which is represented by using Ciao assertions (which are part of the HC IR) of the form:

`:- check` *Pred* [`:` *Precond*] `+` *ResUsage*`.`

where $p(\bar{v}) : c(\bar{v})$ is defined by *Pred* and *Precond*, and Φ is defined by *ResUsage*. The information about $input_p$ and $size_p$ is implicit in *Precond* and *ResUsage*. The concretization of I_α, $\gamma(I_\alpha)$, is the set of all pairs $(p(\bar{t}), r)$ such that $\bar{t}$ is a tuple of terms and $p(\bar{t})$ is an instance of *Pred* that meets precondition *Precond*, and r is a number that meets the condition expressed by *ResUsage* (i.e., r lies in the interval defined by *ResUsage*) for some assertion.

EXAMPLE 6.1. Consider the following HC IR program that computes the factorial of an integer.

```
fact(N,Fact) :- N=<0, Fact=1.
fact(N,Fact) :- N>0, N1 is N-1,
        fact(N1,Fact1), Fact is N*Fact1.
```

One could use the assertion:

```
:- check pred fact(N,F)
   : (num(N), var(F))
  => (num(N), num(F),
      rsize(N, num(Nmin, Nmax)),
      rsize(F, num(Fmin, Fmax)))
   + resource(steps, Nmin+1, Nmax+1).
```

to express that for any call to `fact(N,F)` with the first argument bound to a number and the second one a free variable, the number of resolution (execution) steps performed by the computation is always between `Nmin`$+1$ and `Nmax`$+1$, where `Nmin` and `Nmax` respectively stand for a lower and an upper bound of `N`. In this concrete example, the lower and upper bounds are the same, i.e., the number of resolution steps is exactly `N` $+ 1$, but note that they could be different. □

EXAMPLE 6.2. The assertion in Example 6.1 captures the following concrete semantic tuples:

```
( fact(0, Y), 1 )     ( fact(8, Y), 9 )
```

but it does not capture the following ones:

```
( fact(N, Y), 1 )     ( fact(1, Y), 35 )
```

the left one in the first line above because it is outside the scope of the assertion (i.e., N being a variable, it does

not meet the precondition *Precond*), and the right one because it violates the assertion (i.e., it meets the precondition *Precond*, but does not meet the condition expressed by *ResUsage*). □

Partial correctness: comparing to the abstract semantics.

Given a program p and an intended resource usage semantics I, where $I : \mathcal{C}_p \mapsto \mathcal{R}$, we say that p is partially correct w.r.t. I if for all $p(\bar{t}) \in \mathcal{C}_p$ we have that $(p(\bar{t}), r) \in I$, where r is precisely the amount of resource usage of the computation of the call $p(\bar{t})$. We say that p is partially correct with respect to a tuple of the form $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ if for all $p(\bar{t}) \in \mathcal{C}_p$ such that r is the amount of resource usage of the computation of the call $p(\bar{t})$, it holds that: if $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$ then $r \in \Phi_I(\bar{s})$, where $\bar{s} = size_p(input_p(\bar{t}))$. Finally, we say that p is partially correct with respect to I_α if:

- For all $p(\bar{t}) \in \mathcal{C}_p$, there is a tuple $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ in I_α such that $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$, and
- p is partially correct with respect to every tuple in I_α.

Let $(p(\bar{v}) : c(\bar{v}), \Phi)$ and $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ be tuples expressing an abstract semantics $[\![P]\!]_\alpha$ inferred by analysis and an intended abstract semantics I_α, respectively, such that $c_I(\bar{v}) \sqsubseteq c(\bar{v})$,[2] and for all $\bar{n} \in S$ ($S \subseteq \mathcal{R}^k$), $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$ and $\Phi_I(\bar{n}) = [\Phi_I^l(\bar{n}), \Phi_I^u(\bar{n})]$. We have that:

(1) If for all $\bar{n} \in S$, $\Phi_I^l(\bar{n}) \leq \Phi^l(\bar{n})$ and $\Phi^u(\bar{n}) \leq \Phi_I^u(\bar{n})$, then p is partially correct with respect to $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

(2) If for all $\bar{n} \in S$ $\Phi^u(\bar{n}) < \Phi_I^l(\bar{n})$ or $\Phi_I^u(\bar{n}) < \Phi^l(\bar{n})$, then p is incorrect with respect to $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

Checking the two conditions above requires the comparison of resource usage bound functions.

Resource Usage Bound Function Comparison.

Since the resource analysis we use is able to infer different types of functions (e.g., polynomial, exponential, and logarithmic), it is also desirable to be able to compare all of these functions.

For simplicity of exposition, consider first the case where resource usage bound functions depend on one argument. Given two resource usage bound functions (one of them inferred by the static analysis and the other one given in an assertion/specification present in the program), $\Psi_1(n)$ and $\Psi_2(n)$, $n \in \mathcal{R}$ the objective of the comparison operation is to determine intervals for n in which $\Psi_1(n) > \Psi_2(n)$, $\Psi_1(n) = \Psi_2(n)$, or $\Psi_1(n) < \Psi_2(n)$. For this, we define $f(n) = \Psi_1(n) - \Psi_2(n)$ and find the roots of the equation $f(n) = 0$. Assume that the equation has m roots, $n_1, \ldots, n_m$. These roots are intersection points of $\Psi_1(n)$ and $\Psi_2(n)$. We consider the intervals $S_1 = [0, n_1)$, $S_2 = (n_1, n_2)$, $S_m = \ldots (n_{m-1}, n_m)$, $S_{m+1} = (n_m, \infty)$. For each interval S_i, $1 \leq i \leq m$, we select a value v_i in the interval. If $f(v_i) > 0$ (respectively $f(v_i) < 0$), then $\Psi_1(n) > \Psi_2(n)$ (respectively $\Psi_1(n) < \Psi_2(n)$) for all $n \in S_i$.

There exist powerful algorithms for obtaining roots of polynomial functions. In our implementation we have used the GNU Scientific Library [10], which offers a specific polynomial function library that uses analytical methods for finding roots of polynomials up to order four, and uses numerical methods for higher order polynomials.

We approximate exponential and logarithmic resource usage functions using Taylor series. In particular, for exponential functions we use the following formulae:

$$e^x \approx \Sigma_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \qquad for\ all\ x$$

$$a^x = e^{x\ ln\ a} \approx 1 + x\ ln\ a + \frac{(x\ ln\ a)^2}{2!} + \frac{(x\ ln\ a)^3}{3!} + \ldots$$

In our implementation these series are limited up to order 8. This decision has been taken based on experiments we have carried out that show that higher orders do not bring a significant difference in practice. Also, in our implementation, the computation of the factorials is done separately and the results are kept in a table in order to reuse them.

Dealing with logarithmic functions is more complex, as Taylor series for such functions can only be defined for the interval $(-1, 1)$.

For resource usage functions depending on more than one variable, the comparison is performed using constraint solving techniques.

Safety of the Approximations.

When the roots obtained for function comparison are approximations of the actual roots, we must guarantee that their values are safe, i.e., that they can be used for verification purposes, in particular, for safely checking the conditions presented above. In other words, we should guarantee that the error falls on the safe side when comparing the corresponding resource usage bound functions. For this purpose we developed an algorithm for detecting whether the approximated root falls on the safe side or not, and in the case it does not fall on the safe side, performing an iterative process to increment (or decrement) it by a small value until the approximated root falls on the safe side.

7. USING THE TOOL: EXAMPLE

As an illustrative example of a scenario where the embedded software developer has to decide values for program parameters that meet an energy budget, we consider the development of an equaliser (XC) program using a biquad filter. In Figure 2 we can see what the graphical user interface of our prototype looks like, with the code of this biquad example ready to be verified. The purpose of an equaliser is to take a signal, and to attenuate / amplify different frequency bands. For example, in the case of an audio signal, this can be used to correct for a speaker or microphone frequency response. The energy consumed by such a program directly depends on several parameters, such as the sample rate of the signal, and the number of banks (typically between 3 and 30 for an audio equaliser). A higher number of banks enables the designer to create more precise frequency response curves.

Assume that the developer has to decide how many banks to use in order to meet an energy budget while maximizing the precision of frequency response curves at the same time. In this example, the developer writes an XC program where the number of banks is a variable, say `N`. Assume also that the energy constraint to be met is that an application of the

[2]Note that the condition $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ can be checked using the CiaoPP capabilities for comparing program state properties such as types.

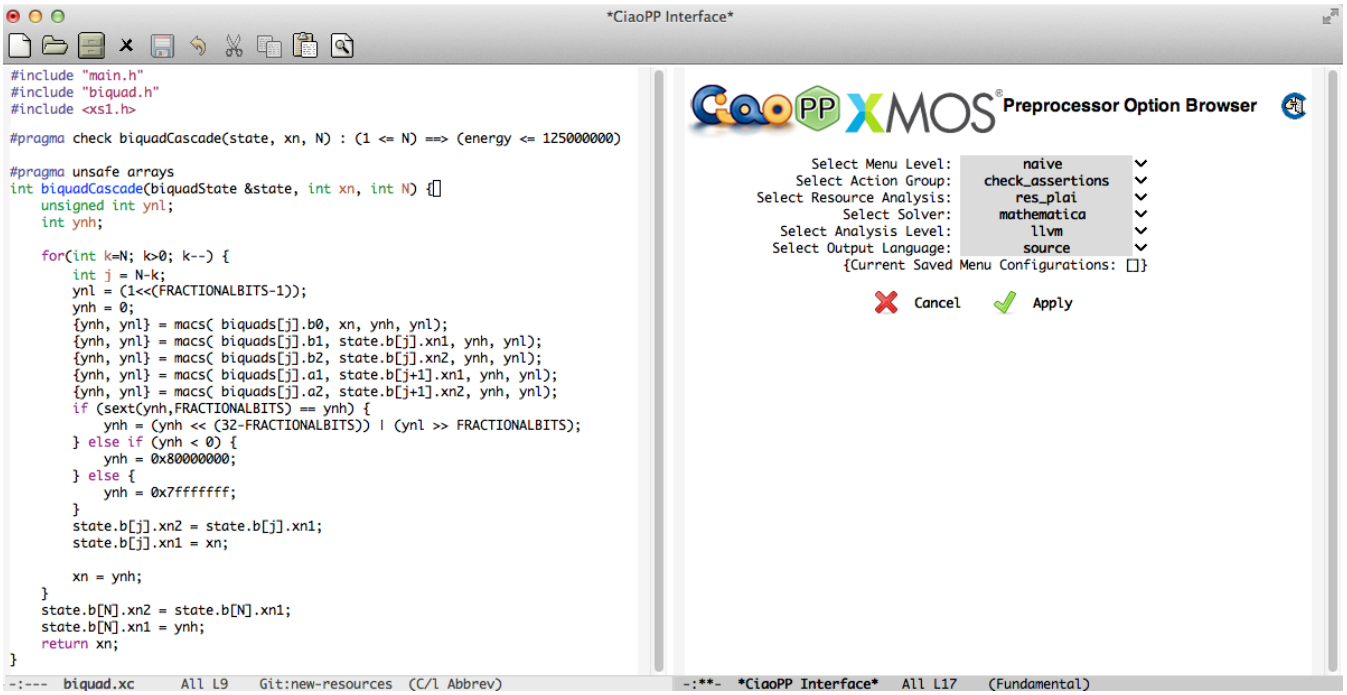

Figure 2: Graphical User Interface of the prototype with the XC biquad program.

biquad program should consume less than 125 millijoules (i.e., 125000000 nanojoules). This constraint is expressed by the following check assertion (specification):

```
#pragma check biquadCascade(state,xn,N) :
   (1 <= N) ==> (energy <= 125000000)
```

where the precondition `1 <= N` in the assertion (left hand side of `==>`) expresses that the number of banks should be at least 1.

Then, the developer makes use of the tool, by selecting the following menu options, as shown in the right hand side of Figure 2: `check_assertions`, for `Action Group`, `res_plai`, for `Resource Analysis`, `mathematica`, for `Solver`, `llvm`, for `Analysis Level` (which will tell the analysis to take the LLVM IR option by compiling the source code into LLVM IR and transform into HC IR for analysis) and finally `source`, for `Output Language` (the language in which the analysis / verification results are shown). After clicking on the `Apply` button below the menu options, the analysis is performed, which infers a lower and an upper bound function for the consumption of the program. Concretely those bounds are represented by the following assertion, which is included in the output of the tool:

```
#pragma true biquadCascade(state,xn,N) :
   (16502087*N + 5445103 <= energy &&
    energy <= 16502087*N + 5445103)
```

In this particular case, both bounds are identical. In other words, the energy consumed by the program is exactly characterized by the following function, depending on `N` only:

$$E_{\text{biquad}}(\texttt{N}) = 16502087 \times \texttt{N} + 5445103 \text{ nJ}$$

Then, the verification of the specification (check assertion) is performed by comparing the energy bound functions above with the upper bound expressed in the specification, i.e., 125000000, a constant value in this case. As a result, the two following assertions are produced (and included in the output file of the tool):

```
#pragma checked biquadCascade(state,xn,N) :
   (1 <= N && N <= 7)
       ==> (energy <= 125000000)
#pragma false biquadCascade(state,xn,N) :
   (8 <= N)
       ==> (energy <= 125000000)
```

The first one expresses that the original assertion holds subject to a precondition on the parameter `N`, i.e., in order to meet the energy budget of 125 millijoules, the number of banks `N` should be a natural number in the interval [1, 7] (precondition `1 <= N && N <= 7`). The second one expresses that the original specification is not met (status `false`) if the number of banks is greater or equal to 8.

Since the goal is to maximize the precision of frequency response curves and to meet the energy budget at the same time, the number of banks should be set to 7. The developer could also be interested in meeting an energy budget but this time ensuring a lower bound on the precision of frequency response curves. For example by ensuring that $\texttt{N} \geq 3$, the acceptable values for `N` would be in the range [3, 7].

In the more general case where the energy function inferred by the tool depends on more than one parameter, the determination of the values for such parameters is reduced to a constraint solving problem. The advantage of this approach is that the parameters can be determined analytically at the program development phase, without the need of determining them experimentally by measuring the energy of expensive program runs with different input parameters.

8. RELATED WORK

As mentioned before, this work adds verification capabilities to our previous work on energy consumption analysis for XC/XS1-L [21], which builds on of our general framework for resource usage analysis [31, 28, 35, 14, 26] and its support for resource verification [22, 24], and the energy models of [16].

Regarding the support for verification of properties expressed as functions, the closest related work we are aware of presents a method for comparison of cost functions inferred by the COSTA system for Java bytecode [1]. The method proves whether a cost function is smaller than another one *for all the values* of a given initial set of input data sizes. The result of this comparison is a Boolean value. However, as mentioned before, in our approach [22, 24] the result is in general a set of subsets (intervals) in which the initial set of input data sizes is partitioned, so that the result of the comparison is different for each subset. Also, [1] differs in that comparison is syntactic, using a method similar to what was already being done in the CiaoPP system: performing a function normalization and then using some syntactic comparison rules. Our technique goes beyond these syntactic comparison rules. Moreover, [1] only covers (generic) cost function comparisons while we have addressed the whole process for the case of energy consumption *verification*. Note also that, although we have presented our work applied to XC programs, the CiaoPP system can also deal with other high- and low-level languages, including, e.g., Java bytecode [29, 26].

In a more general context, using abstract interpretation in debugging and/or verification tasks has now become well established. To cite some early work, abstractions were used in the context of algorithmic debugging in [18]. Abstract interpretation has been applied by Bourdoncle [4] to debugging of imperative programs and by Comini et al. to the algorithmic debugging of logic programs [7] (making use of partial specifications in [6]), and by P. Cousot [8] to verification, among others. The CiaoPP framework [5, 12, 14] was pioneering in many aspects, offering an integrated approach combining abstraction-based verification, debugging, and run-time checking with an assertion language.

9. CONCLUSIONS

We have specialized an existing general framework for resource usage verification for verifying energy consumption specifications of embedded programs. These specifications can include both lower and upper bounds on energy usage, expressed as intervals within which the energy usage is supposed to be included, the bounds (end points of the intervals) being expressed as functions on input data sizes. Our tool can deal with different types of energy functions (e.g., polynomial, exponential or logarithmic functions), in the sense that the analysis can infer them, and the specifications can involve them. We have shown through an example, and using the prototype implementation of our approach within the Ciao/CiaoPP system and for the XC language and XS1-L architecture, how our verification system can prove whether such energy usage specifications are met or not, or infer particular conditions under which the specifications hold. These conditions are expressed as intervals of input data sizes such that a given specification can be proved for some intervals but disproved for others. The specifications themselves can also include preconditions expressing intervals for input data sizes. We have illustrated through this example how embedded software developers can use this tool, and in particular for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service.

10. 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 all the participants of the ENTRA project team, and in particular John P. Gallagher, Henk Muller, Kyriakos Georgiou, Steve Kerrison, and Kerstin Eder for useful and fruitful discussions. Henk Muller (XMOS Ltd.) also provided benchmarks (e.g., the biquad program) that we used to test our tool.

11. ADDITIONAL AUTHORS

Additional authors: John Smith (The Thørväld Group, email: `jsmith@affiliation.org`) and Julius P. Kumquat (The Kumquat Consortium, email: `jpkumquat@consortium.net`).

12. REFERENCES

[1] E. Albert, P. Arenas, S. Genaim, I. Herraiz, and G. Puebla. Comparing cost functions in resource analysis. In *1st International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*, volume 6234 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2010.

[2] N. Bjørner, F. Fioravanti, A. Rybalchenko, and V. Senni, editors. *Workshop on Horn Clauses for Verification and Synthesis*, July 2014. To appear in Electronic Proceedings in Theoretical Computer Science.

[3] N. Bohr, K. Eder, J. P. Gallagher, K. Georgiou, R. Haemmerlé, M. V. Hermenegildo, B. Kafle, S. Kerrison, M. Kirkeby, X. Li, U. Liqat, P. Lopez-Garcia, H. Muller, M. Rhiger, and M. Rosendahl. Initial Energy Consumption Analysis. Technical report, FET 318337 ENTRA Project, April 2014. Deliverable 3.2, `http://entraproject.eu`.

[4] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.

[5] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[6] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.

[7] M. Comini, G. Levi, and G. Vitiello. Declarative diagnosis revisited. In *1995 International Logic Programming Symposium*, pages 275–287, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.

[8] P. Cousot. Automatic Verification by Abstract Interpretation, Invited Tutorial. In *Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, number 2575 in LNCS, pages 20–24. Springer, January 2003.

[9] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL'77)*. ACM Press, 1977.

[10] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual*. Network Theory Ltd, 2009. Available at `http://www.gnu.org/software/gsl/`.

[11] K. Georgiou, S. Kerrison, and K. Eder. A Multi-level Worst Case Energy Consumption Static Analysis for Single and Multi-threaded Embedded Programs. Technical Report CSTR-14-003, University of Bristol, December 2014.

[12] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[13] M. Hermenegildo, G. Puebla, F. Bueno, and P. L. García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2), 2005.

[14] 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.

[15] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *TPLP*, 12(1–2):219–252, 2012. http://arxiv.org/abs/1102.5497.

[16] S. Kerrison and K. Eder. Energy modelling of software for a hardware multi-threaded embedded microprocessor. *ACM Transactions on Embedded Computing Systems (TECS)*, 2015. To appear.

[17] 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.

[18] Y. Lichtenstein and E. Y. Shapiro. Abstract algorithmic debugging. In R. A. Kowalski and K. A. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 512–531, Seattle, Washington, August 1988. MIT.

[19] 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, ENTRA Project, April 2014. Appendix D3.2.4 of Deliverable D3.2. Available at `http://entraproject.eu`.

[20] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Proceedings of LOPSTR'13*, 2014.

[21] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, 2014.

[22] P. López-García, L. Darmawan, and F. Bueno. A Framework for Verification and Debugging of Resource Usage Properties. In *Technical Communications of ICLP*, volume 7 of *LIPIcs*, pages 104–113. Schloss Dagstuhl, July 2010.

[23] P. Lopez-Garcia, L. Darmawan, F. Bueno, and M. Hermenegildo. Interval-Based Resource Usage Verification: Formalization and Prototype. In R. P. na, M. Eekelen, and O. Shkaravska, editors, *Foundational and Practical Aspects of Resource Analysis*, volume 7177 of *LNCS*, pages 54–71. Springer-Verlag, 2012.

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

[25] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *LOPSTR 2007*, number 4915 in LNCS, pages 154–168. Springer-Verlag, August 2007.

[26] 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.

[27] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[28] 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.

[29] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *BYTECODE'09*, volume 253 of *ENTCS*, pages 6–86. Elsevier, March 2009.

[30] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds

Analysis for Logic Programs. In *Proc. of ICLP'07*, volume 4670 of *LNCS*, pages 348–363. Springer, 2007.

[31] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.

[32] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, 2000.

[33] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium (SAS 1996)*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[34] A. Serrano, P. Lopez-Garcia, and M. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *TPLP, ICLP'14 Special Issue*, 14(4-5):739–754, 2014.

[35] 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.

[36] C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.

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