ENTRA

Whole-Systems Energy Transparency

ENTRA

318337

Whole-Systems ENergy TRAnsparency

Initial Energy Consumption Analysis (including SW demo)

Deliverable number:	D3.2
Work package:	Analysis and Verification (WP3)
Delivery date:	1 April 2014 (18 months)
Actual date:	1 April 2014
Nature:	Report
Dissemination level:	PU
Lead beneficiary:	IMDEA Software Institute
Partners contributed:	Roskilde University, University of Bristol, IMDEA Software Institute, XMOS Limited

Project funded by the European Union under the Seventh Framework Programme, FP7-ICT-2011-8 FET Proactive call.

Short description:
This deliverable describes a preliminary instantiation of the general resource analysis framework for energy consumption estimation. It includes a prototype implementation demonstration as well. The deliverable is an indicator (together with deliverable D2.2) that Milestone 2 (First Integration of Energy Models with Analysis) has been reached. As planned in the DOW, such milestone has concluded at month 18, with the lower level energy models of WP2 connected to the higher level analysis of WP3, and the first energy consumption estimations for XC programs inferred. An experimental study of such energy consumption estimations has been performed, which is also reported in this deliverable. The deliverable includes the following attachments:

- D3.2.1. *Probabilistic Output Analysis*. Submitted to the 21st International Static Analysis Symposium (SAS 2014).
- D3.2.2: *Tools for Constrained Horn Clause Verification*. Accepted for publication as a technical communication in Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue, On-line Supplement.
- D3.2.3. *Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types*. Accepted for publication as a paper in Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14), Special Issue.
- D3.2.4. *Inferring Energy Consumption at Different Software Levels: ISA vs. LLVM IR*. Technical Report.
- D3.2.5. *Static energy consumption analysis of LLVM IR programs*. Submitted to the International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES 2014).

Contents

1 Introduction **2**

2 Enabling Analysis at a Higher Level via Mapping Techniques **3**

3 Preliminary Instantiations of the General Resource Analysis Framework **5**

3.1 Instantiation based on HC IR Transformation and the CiaoPP Analyzer 6

3.1.1 Improvements to the Resource Usage Analysis of CiaoPP 7

3.2 Instantiation based on Direct Analysis of LLVM IR 7

3.3 Instantiation of the Model-Based Static Analysis Framework 8

4 Probabilistic Output Analysis **10**

5 Towards the Analysis of Concurrent Programs **14**

6 Prototype Implementation Demonstration **15**

6.1 Demonstration of the Prototype Implementation based on HC IR Transformation 15

6.1.1 Getting Started 16

6.1.2 Performing Energy Consumption Analysis of XC Programs 16

6.2 Demonstration of prototype tool for direct static analysis of LLVM IR 21

6.2.1 Prerequisites and installation 21

6.2.2 Running the tool 21

6.2.3 Estimating costs for a range of inputs 23

Attachments **27**

D3.2.1: Probabilistic Output Analysis 29

D3.2.2: Tools for Constrained Horn Clause Verification 46

D3.2.3: Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types 58

D3.2.4: Inferring Energy Consumption at Different Software Levels: ISA vs. LLVM IR 76

D3.2.5: Static energy consumption analysis of LLVM IR programs 89

1 Introduction

This deliverable is an indicator (together with deliverable D2.2) that Milestone 2 (First Integration of Energy Models with Analysis) has been reached. As planned in the DOW, such milestone has concluded at month 18, with the lower level energy models of WP2 connected to the higher level analysis of WP3, and the first energy consumption estimations for XC programs inferred.

Deliverable D3.1 described the general resource analysis framework we developed, and pointed out that there are many ways of instantiating such framework for energy consumption estimation, depending on the implementation of the main components of the framework, such as the analysers and transformations to the internal representation, as well as the levels at which the analysis is performed and the energy models defined.

A preliminary instantiation that used energy models defined at the Instruction Set Architecture level (ISA, or just assembly) and performed the analysis at the ISA level was also reported in deliverable D3.1 and published in [LKS$^+$13]. In this deliverable we focus on the description of preliminary instantiations of the framework that perform the analysis at the LLVM IR level and use energy models defined at the ISA level. First, the mapping tool to propagate the energy model from the ISA level up to the LLVM IR level is described in Section 2. The analysis approaches are described in Section 3. The ISA level energy model used in these instantiations is described in Deliverable D2.2 (Low-Level Energy Models). A preliminary instantiation of an analysis framework for the internal representation is presented in Section 3.3 along with the report in Attachment D3.2.2. Its purpose is to allow the analysis of relationships between program state variables and resource usage, by abstract interpretation of the models of the internal representation language.

A description of our investigation into the foundations of probabilistic resource usage analysis was included in Deliverable D3.1 [LG13] since supporting probabilistic resource usage analysis was a requirement of our general resource analysis framework. In this deliverable we have developed an approach to derive a probability distribution of output values for a program from a probability distribution of its input, which can be used to perform resource usage analysis by instrumenting it with step-counters, or by integrating it in other energy consumption analyses based on size measures [LKS$^+$13]. This is explained in Section 4. In Section 5 we briefly set out the foundations upon which we intend to further develop analysis approaches for concurrent programs. Finally, Section 6.1 provides demonstrations of prototype implementations.

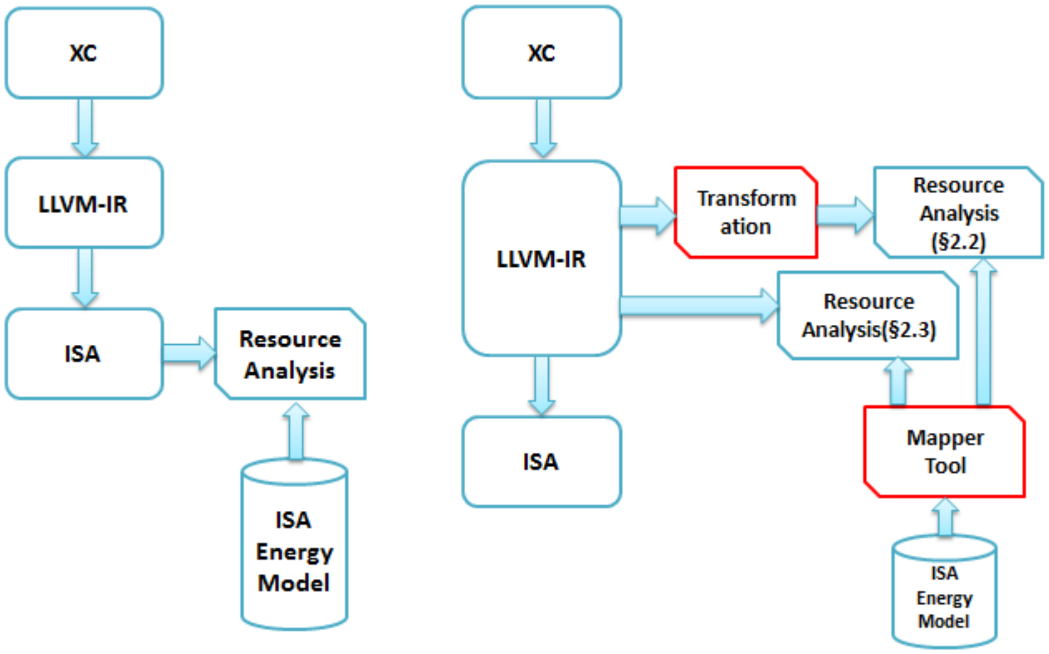


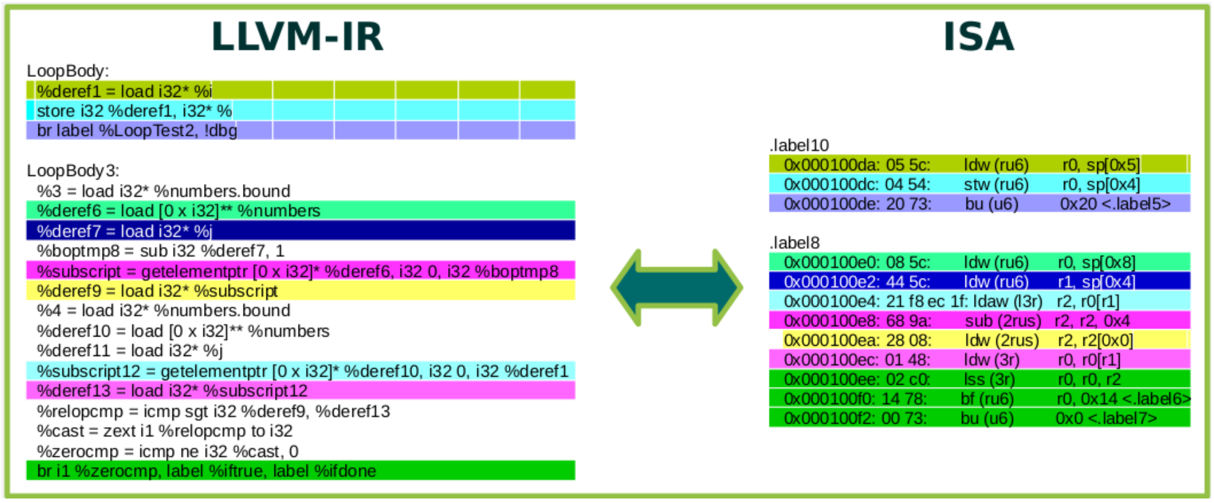
Figure 1: LHS: Analysis on the ISA level (D3.1). RHS: Mapper enables analysis at the LLVM IR level.

2 Enabling Analysis at a Higher Level via Mapping Techniques

Prior to discussing the different approaches of analysis, we introduce here the mapping techniques developed to enable the low level energy model, introduced in Deliverable D2.2, to be used for analysis at a higher level, namely the LLVM IR code. These mapping techniques have been implemented in a mapper tool (an LLVM pass) that can be used to support different analysis. Figure 1 indicates how the mapper tool is used in both of the analysis approaches described in Section 3.1 and Section 3.2, respectively, to associate higher level code segments at the LLVM IR level with values from an energy model established at the ISA level.

To enable analysis at the LLVM IR level, taking an existing energy model at the ISA level as starting point, a mechanism is needed to propagate the ISA-level energy information up to the LLVM IR level. A set of mapping techniques have been developed to serve this purpose. These techniques create 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. An example of this mapping is shown in Figure 2.

Our mapping technique leverages the existing debug mechanism in the XMOS compiler



```
LoopBody:
  %deref1 = load i32* %i
  store i32 %deref1, i32* %
  br label %LoopTest2, !dbg

LoopBody3:
  %3 = load i32* %numbers.bound
  %deref6 = load [0 x i32]** %numbers
  %deref7 = load i32* %j
  %boptmp8 = sub i32 %deref7, 1
  %subscript = getelementptr [0 x i32]* %deref6, i32 0, i32 %boptmp8
  %deref9 = load i32* %subscript
  %4 = load i32* %numbers.bound
  %deref10 = load [0 x i32]** %numbers
  %deref11 = load i32* %j
  %subscript12 = getelementptr [0 x i32]* %deref10, i32 0, i32 %deref1
  %deref13 = load i32* %subscript12
  %relopcmp = icmp sgt i32 %deref9, %deref13
  %cast = zext i1 %relopcmp to i32
  %zerocmp = icmp ne i32 %cast, 0
  br i1 %zerocmp, label %iftrue, label %ifdone
```

```
.label10
  0x000100da: 05 5c:        ldw (ru6)     r0, sp[0x5]
  0x000100dc: 04 54:        stw (ru6)     r0, sp[0x4]
  0x000100de: 20 73:        bu (u6)       0x20 <.label5>

.label8
  0x000100e0: 08 5c:        ldw (ru6)     r0, sp[0x8]
  0x000100e2: 44 5c:        ldw (ru6)     r1, sp[0x4]
  0x000100e4: 21 f8 ec 1f: ldaw (l3r)    r2, r0[r1]
  0x000100e8: 68 9a:        sub (2rus)    r2, r2, 0x4
  0x000100ea: 28 08:        ldw (2rus)    r2, r2[0x0]
  0x000100ec: 01 48:        ldw (3r)      r0, r0[r1]
  0x000100ee: 02 c0:        lss (3r)      r0, r0, r2
  0x000100f0: 14 78:        bf (ru6)      r0, 0x14 <.label6>
  0x000100f2: 00 73:        bu (u6)       0x0 <.label7>
```

Figure 2: An example of mapping ISA to LLVM-IR code, obtained by the mapper tool.

toolchain. This mechanism is originally meant to facilitate the debugging process of an application, particularly when stepping through a program line by line. During the lowering phase of the compilation process, the LLVM IR code is transformed to the specific ISA code by the compiler backend. The debug information (DI) is also stored alongside with the ISA code using the DWARF standard [DWA13], a standardized debugging data format used by many compilers and debuggers to support source level debugging. By tracking this information we can extract an n:m relationship between the two levels, because one source code instruction can be related to many different sequences of LLVM IR instructions and therefore many different sequences of ISA instructions. This n:m relation complicates static analysis.

To address this issue, we created an LLVM pass that traverses the LLVM IR and replaces the *Source Location Information* with LLVM IR location information, right after all the optimization passes and just before emitting the ISA code. In this way, we can extract a 1:m relationship between the mapping of LLVM IR instructions and ISA instructions. Also, by doing it after the LLVM optimization passes the optimized LLVM IR is closer to the ISA code than the unoptimized one, which will go through a series of transformations. There are optimizations that happen during the lowering phase, such as peephole optimizations and some late target specific optimizations that can affect the mapping. However, the effect of these optimizations on the structure of the code is not as profound as those applied to LLVM IR. After a mapping is extracted for a particular program, the associated energy values for the ISA instructions corresponding to a specific LLVM IR instruction are aggregated and then associated with the LLVM IR instruction, and finally to every LLVM IR block.

Although we use the XMOS tool-chain for the mapper tool, the approach is generic and transferable, due to the use of the common LLVM optimizer and code generator, and the use of the DWARF standardized debugging data format, used by many compilers and debuggers to support source layer debugging.

One of the recently published works for LLVM IR energy modeling is [BCF11]. The authors employed a statistical analysis and characterization of LLVM IR code together with instrumentation and execution on the host machine, to estimate performance and energy requirements in embedded software. In their case, porting the LLVM IR energy model to a new platform requires performing the statistical analysis again. Our mapping technique requires only to adjust the LLVM IR mapping pass for the new architecture based on an existing ISA level model for this architecture.

The mapping tool is still a prototype and a more detailed description of the techniques employed in it can be found here [GE14]. The tool will be fully described in deliverable D2.3 (High-level energy models). A brief description of the mapping techniques and tool is also contained in Attachment D3.2.4 and D3.2.5.

3 Preliminary Instantiations of the General Resource Analysis Framework

In this section we report on the different preliminary instantiations of the general analysis framework that we have performed.

Both the prototypes described in Sections 3.1 and 3.2 are based on a well-developed approach in which recursive equations (cost relations) are extracted from a program, representing the cost of running the program in terms of its input [Weg75, Ros89, DLH90, DLGHL97, AAGP11, AAGP09].

These cost relations are converted to *closed-form*, i.e. without recurrences, by means of a solver. The analysis automatically infers an approximate upper (and lower) bound of the energy consumed by programs compiled ISA or to LLVM IR. An energy model defined at the ISA level is used; for LLVM IR analyses the model is propagated to the LLVM IR level via the mapping techniques described in Section 2 for the analysis of XC programs. In Section 3.2 the prototype is also applied to analysing C programs running on an ARM Cortex-M3, using an energy model for LLVM IR for that platform.

3.1 Instantiation based on HC IR Transformation and the CiaoPP Analyzer

This instantiation allows the analysis of XC programs both at the ISA and LLVM IR levels, using an energy model at the ISA level. The analysis of an XC program at the ISA (resp. LLVM IR) level consists of: 1) generating the ISA (resp. LLVM IR) code for such program using the XMOS xcc compiler, 2) transforming the ISA (resp. LLVM IR) into an intermediate block representation based on Horn Clauses (HC IR), 3) using an existing, parametric resource usage analyzer (CiaoPP) to infer energy consumption functions (which depend on input data sizes) for each block in the Horn Clause representation, and 4) mapping the analysis results back to the XC source code.

The instantiation that performs the analysis at the ISA level, using an energy model at the same level, was reported in deliverable D3.1 and published in [LKS+13]. The instantiation that performs the analysis at the LLVM IR level, using the same energy model at the ISA level, together with a mapping tool to propagate the energy model from the ISA level up to the LLVM IR level is fully described in Attachment D3.2.4 in this deliverable. Attachment D3.2.4 also reports on an experimental study of the accuracy and efficiency of this instantiation, comparing it with the one performing the analysis at the ISA level previously mentioned [LKS+13].

These experimental results (described in detail in Attachment D3.2.4, in particular in Tables 1 and 2) show that:

- On average, the analysis performed at either level is reasonably accurate and the relative error between the two analysis at different levels is small.

- ISA-level estimations are slightly more accurate than the ones at the LLVM IR level (3.5% vs. 6.46 % error on average with respect to the actual energy consumption measured on the hardware respectively). This is because the ISA-level analysis uses accurate energy models at the same 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, which causes a slight loss of accuracy.

- The LLVM IR level analysis is more powerful than the one at the ISA level. This is because typing information is preserved at the LLVM IR level, which allows the analysis of programs using data structures (such as arrays) that could not be analyzed at the ISA level, without a significantly more complex representation of memory in the Horn clause representation.

We can conclude that performing the static analysis at the LLVM-IR level is a good 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) 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.

3.1.1 Improvements to the Resource Usage Analysis of CiaoPP

In the instantiation process, we also have developed a new resource usage analyzer within CiaoPP to overcome some important limitations of the existing analyses. This novel general resource analysis for HC IR programs is based on sized types. Sized types are representations that incorporate structural (shape) information and allow expressing both lower and upper bounds on the size of a set of terms and their subterms at any position and depth. They also allow relating the sizes of terms and subterms occurring at different argument positions in logic predicates. Using these sized types, the resource analysis can infer both lower and upper bounds on the resources used by all the procedures in a program as functions on input term (and subterm) sizes, overcoming limitations of existing resource analyses, in particular the resource analyses present in CiaoPP, and enhancing their precision. Our new resource analysis has been developed within the abstract interpretation framework offered by CiaoPP, as an extension of the sized types abstract domain. The abstract domain operations are integrated with the setting up and solving of recurrence equations for inferring both size and resource usage functions. The experimental results for this new analysis are very encouraging, showing that the analysis represents an improvement over the previous one present in CiaoPP and compares well in power to state of the art systems.

A full description of this work has been accepted for publication in [SLGH14], and can be found in attachment D3.2.3 to this document.

3.2 Instantiation based on Direct Analysis of LLVM IR

This instantiation performs energy consumption analysis directly at the LLVM IR level. LLVM IR is an intermediate representation used by modern compilers, including Clang and the XMOS xcc compiler. The analysis approach described in Section 3.1 translates LLVM IR into HC IR in order to benefit from the well developed features of a generic resource analyser, i.e. CiaoPP. Performing the analysis directly on the LLVM IR means that no transformation to a different representation is necessary; instead, the analysis can be applied directly to the LLVM IR blocks. An advantage of this approach is that the analysis tools can be integrated seamlessly into the LLVM toolchain. To illustrate the versatility of this analysis approach we target two different platforms, the XMOS XS1-L and the ARM Cortex-M3, using two different compilers, Clang and xcc.

The approach has been validated using a set of single threaded, open source benchmarks representative for deeply embedded applications as detailed in Section 5 of Attachment D3.2.5, compiled using optimization level `O2` with two compilers. The detailed results for the XMOS XS1-L and the ARM Cortex-M3 processors are shown in Figures 4 and 5 in Appendix D3.2.5, respectively. The graphs show three benchmarks, insertion sort, matrix multiplication and mac, with data series for the static analysis results and actual energy measurements. It can be seen that the static analysis closely fits the empirical results, validating our approach. Table 2 in Attachment D3.2.5 shows the energy consumption formulae and final errors for all seven benchmarks.

Overall, the final error is typically less than 10% and 20% on the XMOS and ARM platforms respectively, showing that the general trend of the static analysis results can be relied upon to give an estimate of the energy consumption accurate enough to inform software design decisions.

Section 6.2 presents the prototype tool for direct static analysis at the LLVM IR level.

3.3 Instantiation of the Model-Based Static Analysis Framework

In this subsection we report on prototype tools developed to analyse CLP programs based on their models, which is one of the techniques for analysis and verification explored in the ENTRA project. The approach was explained in Deliverable 3.1 [LG13]. It was shown how imperative program code can be mapped, via a specification of the operational semantics, to CLP programs. Furthermore it was shown how resource variables could be embedded in the semantics and thus in the resulting CLP programs. The obtained CLP programs typically have run_j predicates whose arguments are divided into two parts.

$$run_j(\ \overbrace{X_1, \ldots, X_{j_n}}^{\text{program state variables}}\ ,\ \overbrace{E_0, \ldots, E_k}^{\text{resource variables}}\ ).$$

Here the predicate run_j represents the program state at some program point j. The program behaviour is represented as a transition system modelled by CLP clauses of the following form.

$$run_j(X_1, \ldots, X_{j_n}, E_0, \ldots, E_k) \leftarrow\ C, run_k(X'_1, \ldots, X'_{j_n}, E'_0, \ldots, E'_k)$$

where C is a constraint relating the variables of the two states j and k.

Analysis of the model of the CLP program yields information about the possible states of the program at each selected program point, including the relationship between program state variables and resource variables. The precise state at each program point is in general not computable and we have to resort to approximations. Let P be a CLP program. An over-approximation of

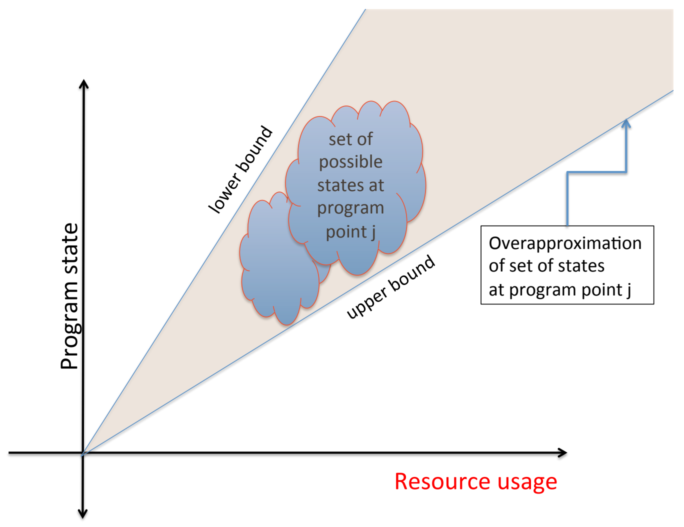

Program state
lower bound
upper bound
set of possible states at program point j
Overapproximation of set of states at program point j
Resource usage

Figure 3: Over-approximation of relationship between program and resource variables

the model of P is a set of variable-free facts M of the form $r_j(v_1, \ldots, v_{j_n}, e_0, \ldots, e_k)$ such that for all facts A, $P \models A$ implies $A \in M$. In other words M contains at least all the atomic logical consequences of the program P, which capture all possible states of the program.

The relationship between resource variables and program variables at specific program points can be obtained from the over-approximation as illustrated in the Figure 3. The diagram shows two dimensions – one program state variable and one resource variable – but more complex relationships can be handled. The cloud-shaped region represents the actual points relating the variables, which in general is an uncomputable set. The analysis computes a convex polyhedron (or more generally a finite set of convex polyhedra) that includes the actual points. From this we can deduce information such as upper and lower bounds of resource usage with respect to the values of program variables. We can also use the approximation to verify invariants on the state, including invariants relating resource variables to program variables. This is the focus of the paper in Attachment D3.2.2 in which we give details of a toolset for the verification of Horn clause properties. The aim in this work is to investigate the use of a combination of off-the-shelf techniques from the literature in analysis and transformation of Constraint Logic Programs (CLPs) to solve Horn clause verification problems. We find that many problems can be solved using a combination of tools based on well-known techniques from abstract interpretation, semantics-preserving transformations, program specialisation and query-answer transformations. This gives insights into the design of automatic, more general verification tools based on a library of components.

Suppose that we wish to verify a property $c(X, R)$ at some program point k, where $c(X, R)$ is a constraint expressing the invariant to be verified. In the assertion language of the ENTRA project [EG13] this is represented in the following form.

```
:- check pred run_j(...,X,...,R,...): c(X,R).
```

In the model-based verification framework the property to be verified is stated as a Horn clause whose head is false. Such a clause is known as an integrity constraint. Then the integrity constraint representing the property is as follows.

$$\mathsf{false} \leftarrow \neg c(X, R), run_j(\ldots, X, \ldots, R, \ldots).$$

The task of the verification tools is in general to show that the integrity constraints are satisfied, or equivalently that the predicate false is not derivable from the Horn clauses.

The full article in Attachment D3.2.2 describes a number of tools. The core is an abstract interpretation procedure for building a convex polyhedral approximation of the model of the program. The other tools are transformations of the input Horn clause program that enhance the efficiency or precision of the analysis while preserving the derivability of false.

4 Probabilistic Output Analysis

The aim of a probabilistic analysis is to derive a probability distribution for output values from a probability distribution for input to a program. We can analyse resource usage in this way by instrumenting programs with step-counters for complexity analysis [Ros89] or energy consumption measures [LKS$^+$13]. Since resource usage is an internal property, we externalise the time or energy consumption so that it is a denotational property of the analysed program.

When analysing energy consumption, probability distributions may provide more useful information than boundaries. Wierman et al. states that *"global energy consumption is affected by the average case, rather than the worst case"* [WAT08]. Also in scheduling *"an accurate measurement of a tasks average-case execution time (ACET) can assist in the calculation of more appropriate deadlines"* [GBMH07]. For a subset of programs a precise average case execution time can be found using static analysis [FSZ91, Sch08, Gao13]. In some cases the probabilistic analysis delivers not only an accurate output average but the more descriptive accurate output distribution. In other cases the probabilistic analysis must over-approximate the probability distribution and the expected value (average case result) will be approximated safely as a range. Another application area for such analyses is in temperature management, where worst-case bounds are important [SBYT12]. Because the analysis return distributions it can be used to calculate the probability energy consumptions above a certain limit, and thereby indicating the risk of over-heating.

When analysing probabilities the main challenge is to maintain the dependencies throughout the program. Schellekens defines this as *Randomness preservation* [Sch08] (or random bag preservation) which in his (and Gao's [Gao13]) case enables tracking of certain data structures and their distributions. They use special data structures as they find these suitable to derive the average number of basic operations. In another approach [Weg75, PCG09], tests in programs has been assumed to be independent of previous history, also known as the Markov property (the probability of true is fixed). As Wegbreit remarked, this is true only for some programs (e.g. linear search for repeating lists) and others, this is not the case (linear search for non-repeating lists). The Markov property is the foundation in Markov decision processes which is used in probabilistic model-checking [FKNP11]. Cousot et al. presents a probabilistic abstraction framework where they divide the program semantics into probabilistic behaviour and (non-)deterministic behaviour. They propose handling of tests when it is possible to assume the Markov property, and handle loops by using a probability function describing the probability of entering the loop in the ith iteration. Monniaux propose another approach for abstracting probabilistic semantics [Mon00]; he first lifts a normal semantics to a probabilistic semantics where random generators are allowed and then uses an abstraction to reach a closed form. Monniaux's semantic approach uses a backward probabilistic semantics operating on measurable functions. This is closely related to the forward probabilistic semantics proposed earlier by Kozen [Koz79, Koz81].

We will here give a short outline of the probabilistic analysis and refer to the paper in Attachment D3.2.1 for further details.

Probabilistic analysis of a first-order functional language

Normal dataflow analysis will propagate abstract descriptions of the set of possible data through the program. A probabilistic analysis uses probability distributions so that we not only describe possible values but also the likelihood of a variable having a specific value. Program analysis requires a method for approximating such a description since the precise analysis is typically undecidable. In a dataflow analysis a standard approach is to over-approximate the set of possible values. In a probabilistic analysis we will use over-approximations of probability distributions such that the sum of probabilities might be greater than one.

Our analysis is based on a first order functional language but as we will discuss in the next section, it can be used to analyse resource usage of XC programs. Our approach is based on program transformation rather than direct abstraction of the individual constructs in the language.

We will here only give an outline of the approach and refer to the full paper in Attachment D3.2.1 for further details.

We assume that the program to be analysed is a mapping f from a vector of input values $\vec{x} \in X$ to a vector of output values $\vec{z}$ and that we have a probability distribution for input values $P_x : X \to \{v \mid 0 \leq x \leq 1\}$.

The output probability distribution P_f can be defined as

$$P_f(\vec{z}) = \sum_{\vec{x}} P_x(\vec{x}) \cdot C(\vec{z} = f_1(\vec{x}))$$

where $C(v) = 1$ if $v = \textit{true},$ otherwise 0.

The aim of the analysis is to obtain a closed form expression from the output probability distribution. The developed method uses a number of stages of transformations

- Unfolding. The probability distributions are propagated through the program by function calls in the program. The aim is to remove functions from the original program in the output probability function.

- Symbolic summation. The probability function may contain recursion that correspond to finite sums and we rewrite the expressions correspondingly.

- Approximation. If the previous stages did not provide a closed form expression we can use various techniques to over-approximate the probability distribution

Example As an example let us consider a recursive addition function. We shall see how the original program is inserted into the probability formula, expanded and reduced to a closed form function expressing the probability distribution for the output. Recall an add-program:

```
add(x,y) = if(x=0) then y else add(x-1,y+1)
```

Let us consider the addition of two independent integer values `x` and `y`, where both input variables `x` and `y` have a uniform distribution from 1 to a number n.

Propagating probability functions through the program by unfolding function calls will give us the following output probability function

$$P_{\texttt{add}}(z) = \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot \\ \sum_{i=0} \prod_{j=0}^{i-1} C(\neg b(h'(j,x,y))) \cdot C(b(h'(i,x,y))) \cdot C(z = g(h'(i,x,y))$$

where

$$\begin{aligned}
&b(x,y) = x = 0\\
&g(x,y) = y\\
&h(\langle x,y\rangle) = \langle x-1, y+1\rangle\\
&h'(i,\langle x,y\rangle) = \textbf{ if } (i=0) \textbf{ then } \langle x,y\rangle \textbf{ else } h'(i-1,\langle x-1,y+1\rangle)\\
&\qquad\quad = \langle x-i, y+i\rangle
\end{aligned}$$

The next stage is to use symbolic summation and algebraic rewriting rules to obtain an expression in closed form.

$$\begin{aligned}
P_{\texttt{add}}(z) &= \sum_x \sum_y P_x(x)\cdot P_y(y)\cdot\\
&\qquad \sum_{i=0} \prod_{j=0}^{i-1} C(\neg(x-j=0))\cdot C(x-i=0)\cdot C(z=y+i)\\
&= \sum_x \sum_y P_x(x)\cdot P_y(y)\cdot \sum_{i=0} C(x=i)\cdot C(z=y+i)\\
&= \sum_x \sum_y P_x(x)\cdot P_y(y)\cdot C(z=y+x)\\
&= \sum_x \sum_y \frac{1}{n}\cdot C(1\le x\le n)\cdot \frac{1}{n}\cdot C(1\le y\le n)\cdot C(z=y+x)\\
&= \sum_y \frac{1}{n}\cdot C(z-n\le y\le z-1)\cdot \frac{1}{n}\cdot C(1\le y\le n))\\
&= \frac{1}{n^2}\cdot \max(\min(n,z-1)-\max(1,z-n)+1,0)\\
&= \frac{1}{n^2}\cdot (C(n<z\le 2n)\cdot(2n-z+1)+C(1\le z\le n)\cdot(z-1))
\end{aligned}$$

The final expression is the output probability distribution for addition program (in a closed form). The distribution has the well-known pyramid shape as expected.

Future work

The approach described here can be extended in a number of ways to improve the analysis of energy consumption in XC programs. Some of the main areas of further research are outlined below.

Approximation techniques At the core of useful program analysis techniques are the approach it uses to approximate values. The transformation method we use is not guaranteed to produce probability functions in closed form and further methods may be used to simplify and approximate expressions. The crude approach is to overapproximate probabilities as one whenever we cannot obtain a closed form expression but techniques using copulas can give more precise results. Copulas uses the theory of comonotonicity [DDG$^+$02] for distributions that may depend on a common (possibly unknown) random variable and makes it possible to give tighter bounds when subexpressions may be dependent but the approximations have removed precise information about the dependencies.

Composite data structures The approach is not restricted to tuples of simple values but can also be used when data is structured as lists and arrays. We are, however, working on techniques for improving the precision of analysis that uses composite data structures.

Implementation and experiments We have a basic implementation of the core concepts in the analysis. We will, however, extend the implementation so as to obtain further results from larger examples and integrate the analysis into a common framework where XC programs can be analysed directly or after translation into CLP form.

5 Towards the Analysis of Concurrent Programs

In deliverable D3.1 [LG13] approaches for the analysis of concurrent programs have been described. The focus of our research over the coming project period is to explore these further and to develop energy consumption analysis techniques for multi-threaded programs.

It is important to note that the energy models (deliverable D2.2 [EKG14]) that underpin our analysis are inherently multi-threaded to reflect the processor architecture that they represent. As such, the currently utilised energy models remain useful as we progress towards energy consumption analysis of multi-threaded programs.

Looking towards the target of analysis, one of the primary languages under inspection, XC, is inherently multi-threaded, with multi-threading and inter-process communication semantics forming first-class components of the language. As such, structural information on the threading and communication of programs written in XC, is available at various stages in compilation for exploitation by our analysis.

Further work is required in the area of modelling communication in order to better support accurate analysis of communicating multi-threaded programs. This is serviced by Task 2.3, which commences as this deliverable is concluded.

Considering targets for analysis, a number of the benchmarks delivered in D5.1 have multi-threaded implementations. A variety of communication methods (synchronised and unsynchronised) as well as numbers of threads and cores utilised are covered in the benchmarks, some of which have multiple implementations, including sequential versions for comparison.

6 Prototype Implementation Demonstration

This section provides demonstrations of the prototype tools developed as part of this work package, contributing to this deliverable. The first demonstration shows a prototype implementation of software analysis based on HC IR transformation, and the second demonstrates static analysis directly applied to LLVM IR.

6.1 Demonstration of the Prototype Implementation based on HC IR Transformation

In this section we demonstrate the prototype implementation described in Section 3.1. It makes use of CiaoPP, a system that offers a framework for performing program analysis, verification and optimization, based on modular, incremental abstract interpretation.

The role of CiaoPP in the ENTRA project is to allow experimentation with energy analysis at different levels of abstraction. During later stages of the project, when experimental studies have stabilised, functionality from the CiaoPP system can be integrated with a compiler tool-chain. In other words, the prototype presented here is intended for use by the ENTRA project developers rather than XC developers.

CiaoPP is the preprocessor of the Ciao program development system [HBC$^+$12]. Ciao is a multi-paradigm programming system, allowing programming in logic, constraint, and functional styles (as well as a particular form of object-oriented programming). At the heart of Ciao is an efficient logic programming-based kernel language. This allows the use of the very large body of approximation domains, inference techniques, and tools for abstract interpretation-based semantic analysis which have been developed to a powerful and mature level in this area. These techniques and systems can approximate at compile-time, always safely, and with a significant degree of precision, a wide range of properties which is much richer than, for example, traditional types. This includes data structure shape (including pointer sharing), independence, storage reuse, bounds on data structure sizes and other operational variable instantiation properties, as well as procedure-level properties such as determinacy, termination, non-failure, and bounds on resource consumption.

CiaoPP [HPBLG05] is a standalone preprocessor to the standard clause-level compiler. It performs source-to-source transformations. The input to CiaoPP are logic programs (optionally with assertions and syntactic extensions). The output are *error/warning messages* plus the *transformed logic program*, with:

- Results of analysis (as assertions).

- Results of static checking of assertions.
- Assertion run-time checking code.
- Optimizations (specialization, parallelization, etc.)

By design, CiaoPP is a generic tool that can be easily customized to different programming systems and dialects and allows the integration of additional analyses in a simple way. As a particularly interesting example, in the ENTRA project, the preprocessor has been adapted to perform energy consumption analysis of XC Programs, allowing rapid prototyping of XC resource analysis tools.

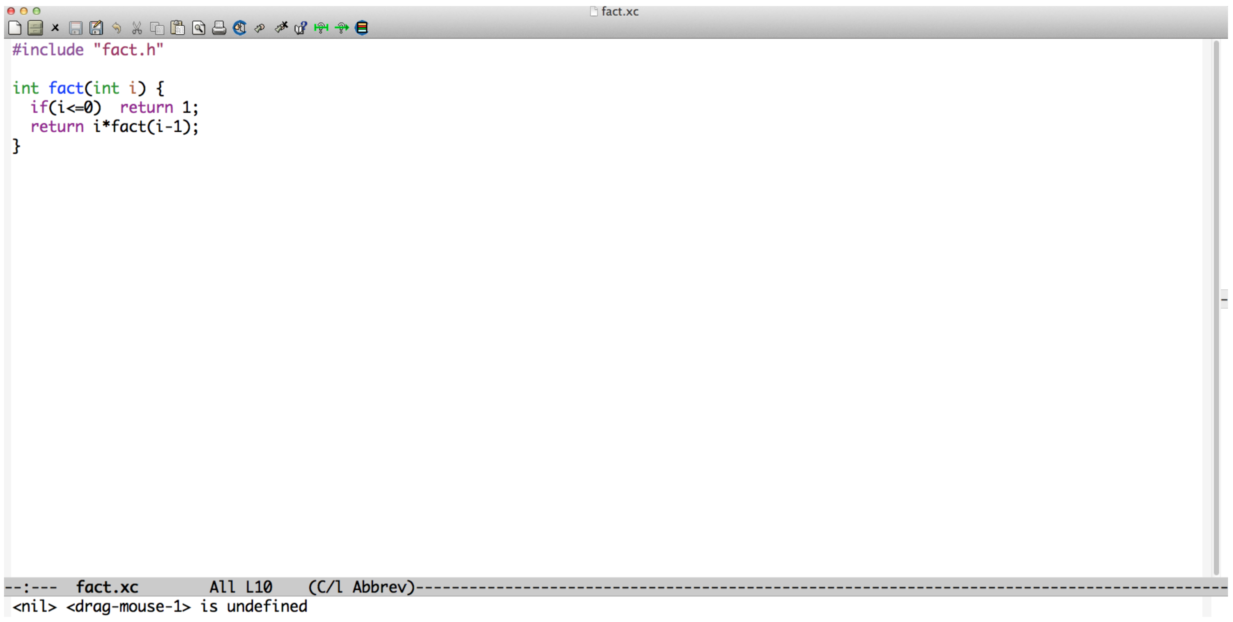
6.1.1 Getting Started

A CiaoPP session consists in the preprocessing of a file. A session using the graphical user interface is governed by a menu, where you can choose the kind of preprocessing you want to be done to your file among several analyses and program transformations available. Clicking on the icon in the buffer containing the file to be preprocessed displays the menu, which will look (depending on the options available in the current CiaoPP version) something like the "Preprocessor Option Browser" shown in Figure 5.

Except for the first and last lines, which refer to loading or saving a menu configuration (a predetermined set of selected values for the different menu options), each line corresponds to an option you can select, each having several possible values. In the `Action Group` line you can select either analysis (`analyze`), assertion checking (`check_assertions`), certificate checking (`check_certificate`) or program optimization (`optimize`), and you can later combine the four kinds of preprocessing. The relevant options for the `Action Group` selected are then shown, together with the relevant flags. In this document we focus on the `analyze` option, since our goal is to perform energy consumption analysis.

6.1.2 Performing Energy Consumption Analysis of XC Programs

In order to analyze an XC program using the CiaoPP graphical interface, we first open it in a buffer, as shown in Figure 4. Then we select the menu options depicted in Figure 6 (marked with a red arrow to ease identification in this document): `analyze`, for `Action Group`, `res_plai`, for `Resource Analysis` (which will select the new analysis we have developed, described in Section 3.1.1 (fully described in [SLGH14], attachment D3.2.3). After clicking on the `Apply` button (marked with a red oval) below the menu options, the analysis is performed,



```
#include "fact.h"

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

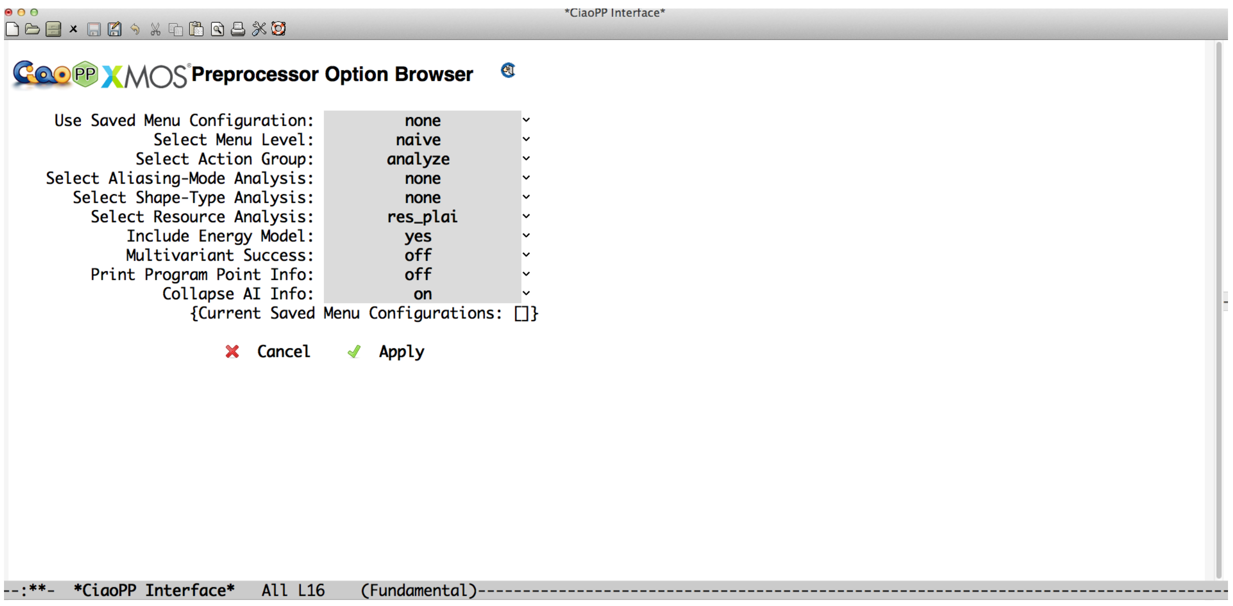
```
--:---  fact.xc        All L10    (C/l Abbrev)------------------------------------
<nil> <drag-mouse-1> is undefined
```

Figure 4: XC source of a factorial program.

producing the results as depicted in Figure 7 (marked with a red arrow). Such results are expressed in the front end aspect of the common assertion language, as explained in Deliverables D2.1 [EG13] and D3.1 [LG13]. We can see that the energy consumption of the factorial program is given as a linear function on the size of the input argument to the program, A, namely $21469718 * A + 16420396\ nJ$.

Besides the graphical user interface that has been illustrated so far, CiaoPP also offers a command line interface to perform the analysis (and other actions) for more advanced users and analysis tool developers.

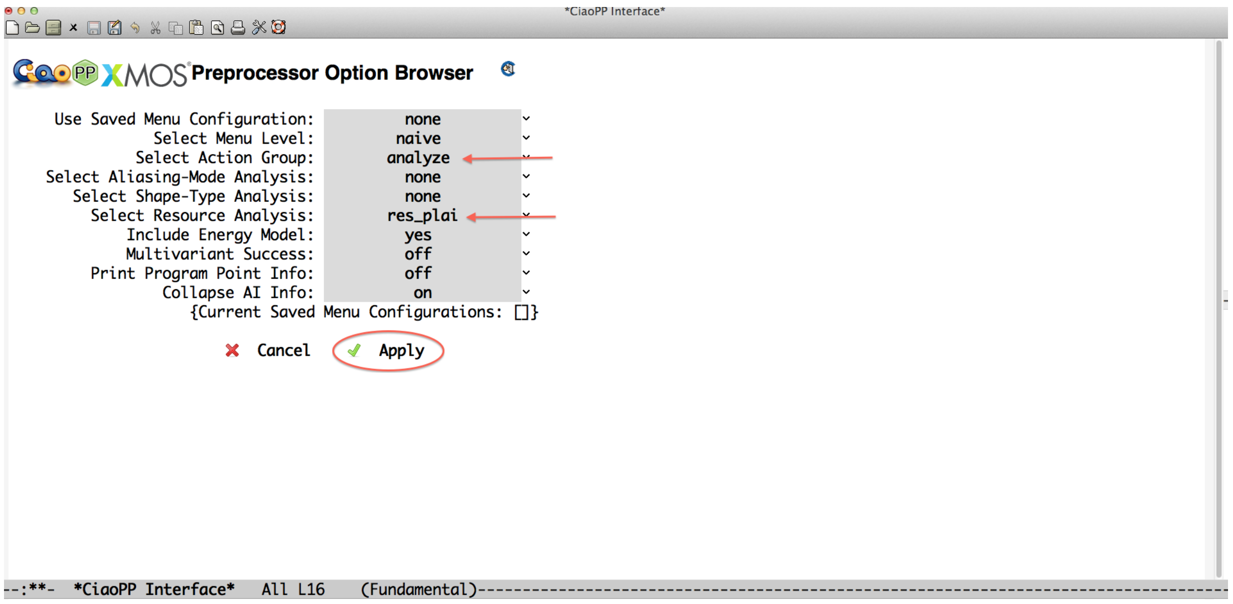
It is also possible to visualize a number of files that contain useful information for the developers of the analysis tools of the ENTRA project. For example, the ISA code generated by the xcc compiler corresponding to the XC source program is shown in Figure 8; the energy model at the ISA level, expressed in the Internal Assertion Language (IAL) [EG13] used by the analyzer is shown in Figure 9; and part of the Horn Clause representation of the ISA code together with an assertion in the IAL expressing analysis results is shown in Figure 10.



```
Use Saved Menu Configuration:    none
           Select Menu Level:    naive
         Select Action Group:    analyze
Select Aliasing-Mode Analysis:   none
   Select Shape-Type Analysis:   none
     Select Resource Analysis:   res_plai
         Include Energy Model:   yes
         Multivariant Success:   off
     Print Program Point Info:   off
             Collapse AI Info:   on
       {Current Saved Menu Configurations: []}
```

CiaoPP Interface
Preprocessor Option Browser
Cancel
Apply
--:**- *CiaoPP Interface*   All L16   (Fundamental)----

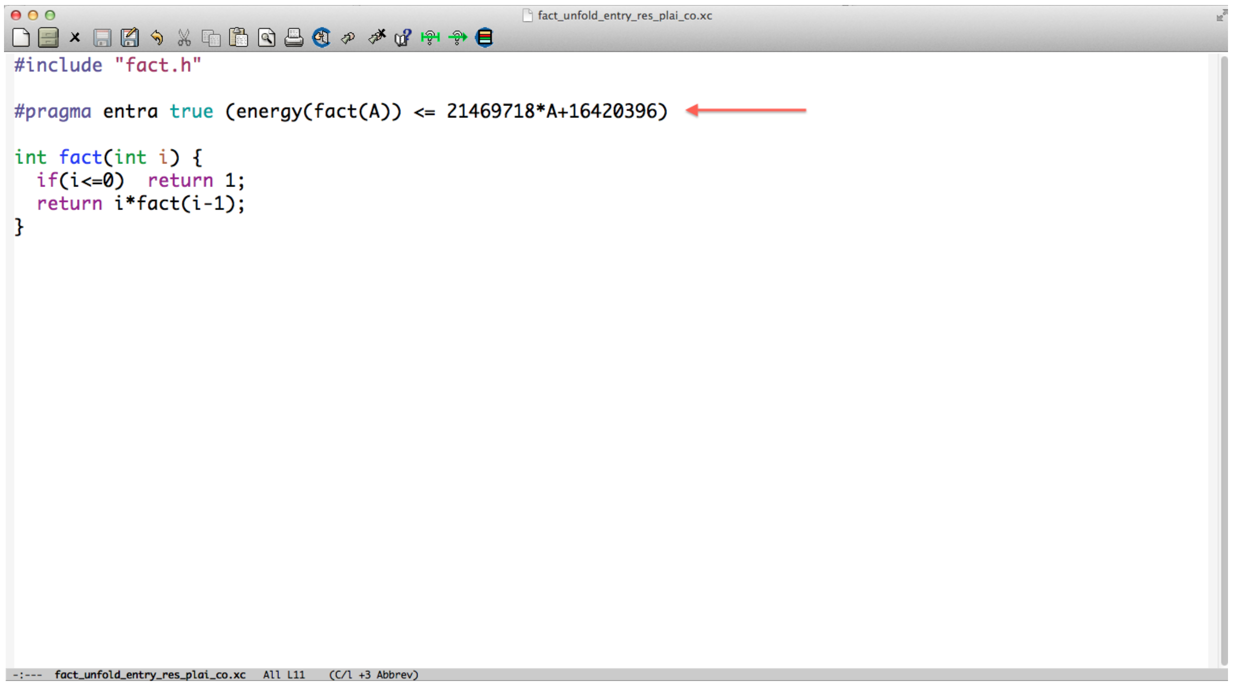
Figure 5: CiaoPP menu.



```
Use Saved Menu Configuration:    none
           Select Menu Level:    naive
         Select Action Group:    analyze
Select Aliasing-Mode Analysis:   none
   Select Shape-Type Analysis:   none
     Select Resource Analysis:   res_plai
         Include Energy Model:   yes
         Multivariant Success:   off
     Print Program Point Info:   off
             Collapse AI Info:   on
       {Current Saved Menu Configurations: []}
```

CiaoPP Interface
Preprocessor Option Browser
Cancel
Apply
--:**- *CiaoPP Interface*   All L16   (Fundamental)----

Figure 6: CiaoPP menu with options selected for resource analysis.

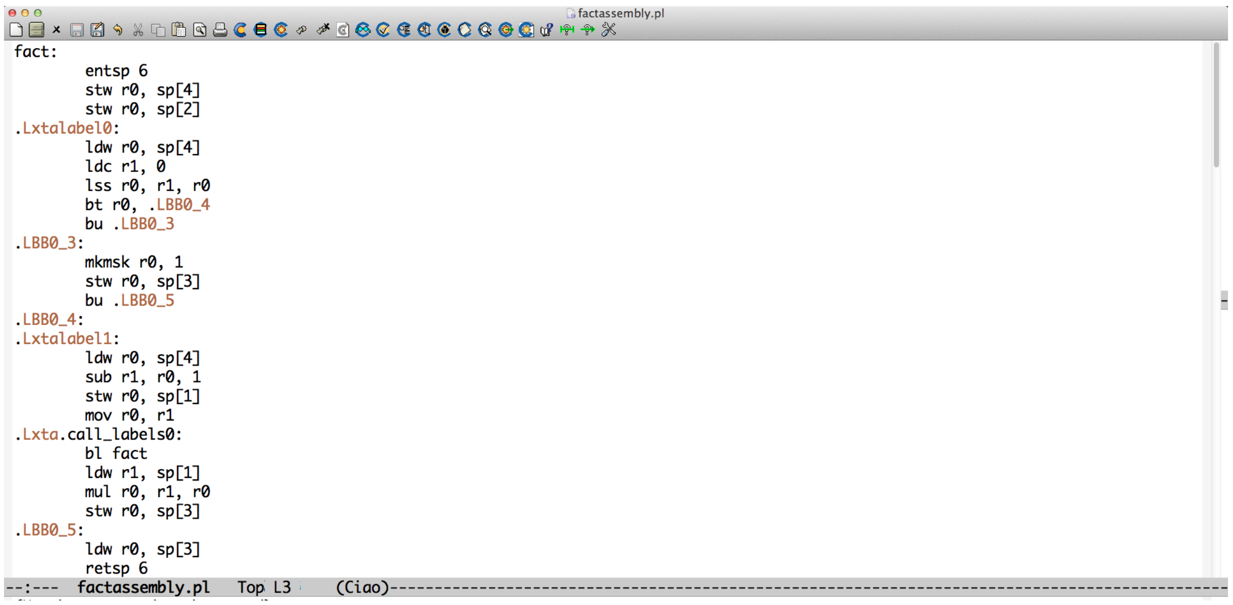


fact_unfold_entry_res_plai_co.xc

```
#include "fact.h"

#pragma entra true (energy(fact(A)) <= 21469718*A+16420396)

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

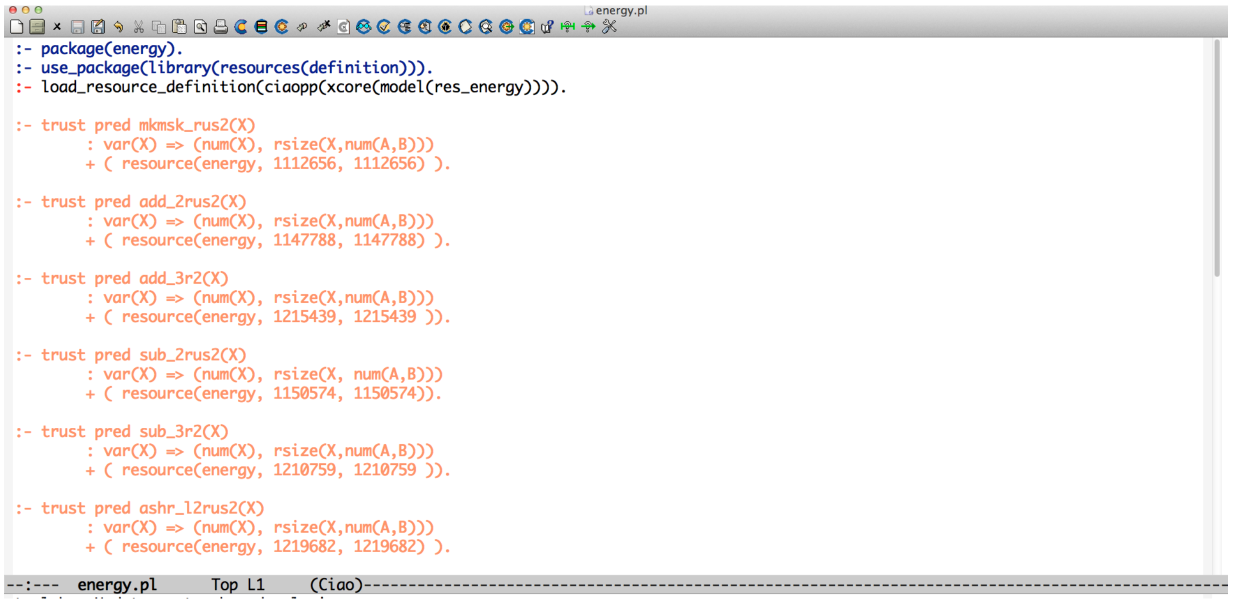
-:--- fact_unfold_entry_res_plai_co.xc   All L11   (C/l +3 Abbrev)

Figure 7: Analysis results (in the front end assertion language).



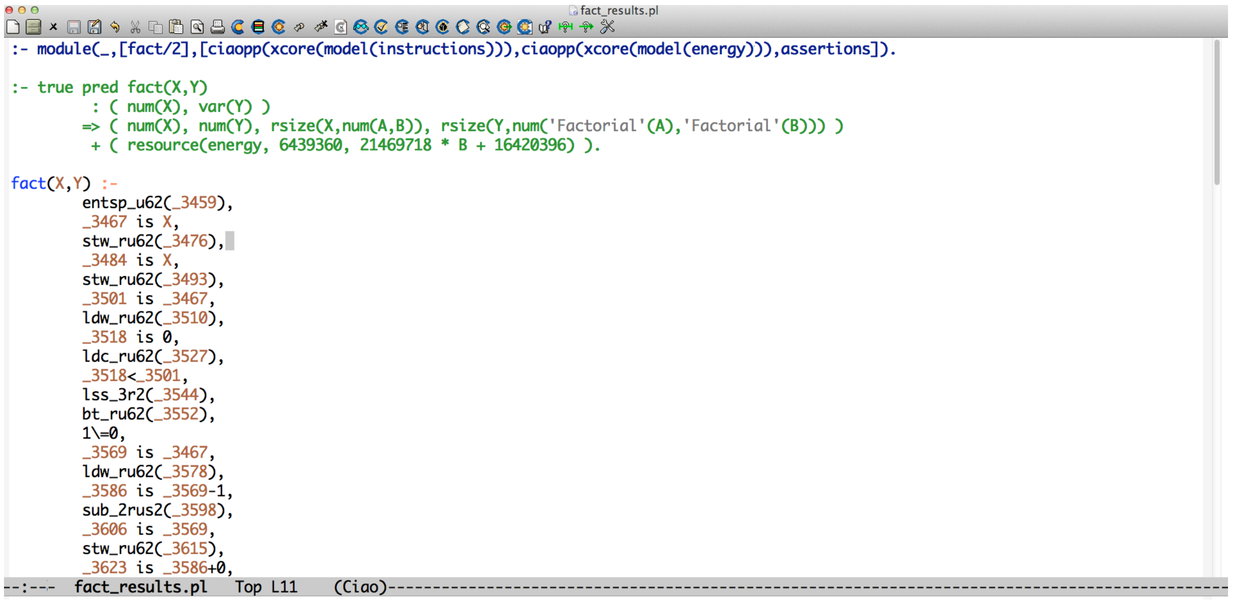
```
fact:
        entsp 6
        stw r0, sp[4]
        stw r0, sp[2]
.Lxtalabel0:
        ldw r0, sp[4]
        ldc r1, 0
        lss r0, r1, r0
        bt r0, .LBB0_4
        bu .LBB0_3
.LBB0_3:
        mkmsk r0, 1
        stw r0, sp[3]
        bu .LBB0_5
.LBB0_4:
.Lxtalabel1:
        ldw r0, sp[4]
        sub r1, r0, 1
        stw r0, sp[1]
        mov r0, r1
.Lxta.call_labels0:
        bl fact
        ldw r1, sp[1]
        mul r0, r1, r0
        stw r0, sp[3]
.LBB0_5:
        ldw r0, sp[3]
        retsp 6
```

Figure 8: Assembly code for the factorial program.



```
:- package(energy).
:- use_package(library(resources(definition))).
:- load_resource_definition(ciaopp(xcore(model(res_energy)))).

:- trust pred mkmsk_rus2(X)
        : var(X) => (num(X), rsize(X,num(A,B)))
        + ( resource(energy, 1112656, 1112656) ).

:- trust pred add_2rus2(X)
        : var(X) => (num(X), rsize(X,num(A,B)))
        + ( resource(energy, 1147788, 1147788) ).

:- trust pred add_3r2(X)
        : var(X) => (num(X), rsize(X,num(A,B)))
        + ( resource(energy, 1215439, 1215439 )).

:- trust pred sub_2rus2(X)
        : var(X) => (num(X), rsize(X, num(A,B)))
        + ( resource(energy, 1150574, 1150574)).

:- trust pred sub_3r2(X)
        : var(X) => (num(X), rsize(X,num(A,B)))
        + ( resource(energy, 1210759, 1210759 )).

:- trust pred ashr_l2rus2(X)
        : var(X) => (num(X), rsize(X,num(A,B)))
        + ( resource(energy, 1219682, 1219682) ).

--:---  energy.pl      Top L1     (Ciao)----------------------------------------
```

Figure 9: Energy model at the ISA level (in the Internal Assertion Language).



```
:- module(_,[fact/2],[ciaopp(xcore(model(instructions))),ciaopp(xcore(model(energy))),assertions]).

:- true pred fact(X,Y)
         : ( num(X), var(Y) )
        => ( num(X), num(Y), rsize(X,num(A,B)), rsize(Y,num('Factorial'(A),'Factorial'(B))) )
         + ( resource(energy, 6439360, 21469718 * B + 16420396) ).

fact(X,Y) :-
        entsp_u62(_3459),
        _3467 is X,
        stw_ru62(_3476),
        _3484 is X,
        stw_ru62(_3493),
        _3501 is _3467,
        ldw_ru62(_3510),
        _3518 is 0,
        ldc_ru62(_3527),
        _3518<_3501,
        lss_3r2(_3544),
        bt_ru62(_3552),
        1\=0,
        _3569 is _3467,
        ldw_ru62(_3578),
        _3586 is _3569-1,
        sub_2rus2(_3598),
        _3606 is _3569,
        stw_ru62(_3615),
        _3623 is _3586+0,
--:---  fact_results.pl    Top L11    (Ciao)----------------------------------------
```

Figure 10: HC IR for factorial and analysis results (in the Internal Assertion Language).

6.2 Demonstration of prototype tool for direct static analysis of LLVM IR

The implementation of this tool, along with the theory and research relating to it, is described in Section 3.2. The workflow of this tool is as follows:

1. Compile source code using LLVM, producing an LLVM bitcode (bc) file.
2. Execute static analysis tool, `inferCR.py`, to produce Cost Relations (CRs), optionally providing a mapped energy model (Section 2), which will provide CRs in terms of energy consumption, or simply instruction counts otherwise.
3. Solve CRs, using a solver such as `PUBS`, in order to produce a bounded resource consumption estimation for the analysed code.

6.2.1 Prerequisites and installation

The tool `inferCR.py` is implemented in Python and utilises various modules that are part of the core Python installation. In addition, the `python-llvm` module, version 3.2, is used.

Installing `llvm-3.2` with RTTI LLVM must be compiled with Run-Time Type Information (RTTI) enabled. This can be done either by obtaining an appropriate build of `llvm-3.2` from a suitable repository, or by building LLVM from source with `REQUIRES_RTTI=1` set in the environment during `make`.

Installing `llvm-py` Once LLVM is installed, `llvm-py` can be installed, which can be obtained from `http://www.llvmpy.org/`, wherein installation instructions can also be found. If the `llvm.test()` suite can be run successfully, as per the `llvm-py` installation instructions, then `inferCR.py` should also run correctly.

6.2.2 Running the tool

The following demonstration shows how the tool can be used, taking a simple function `accumulate` as an example, which produces the result x, which is a sum N, of a list of numbers, A, in the form $x = \sum_i^N \left(A_{i=0} + \sum_{j=0}^N A_j\right)$, implemented as the following code sample:

```
int accumulate(int numbers[], int size) {
  int i, j, temp=0;

  for (i = size-1; i > 0; i--)  {
      temp+=numbers[i];
      for (j = size-1; j > 0; j--)  {
          temp+=numbers[j];
      }
      temp+=numbers[i];
  }
  return temp;
}
```

Note that in this example, the equation and implementation are equivalent in result, but the code implements the solution by iterating over the array of numbers in reverse.

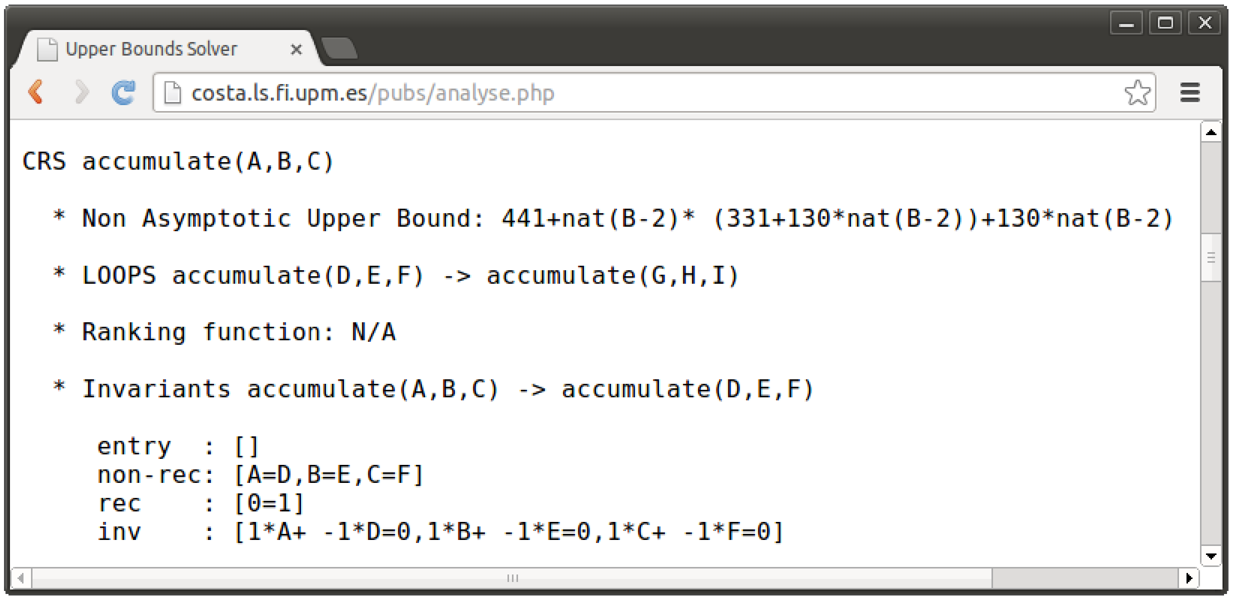
Compilation The code must be compiled to a bitcode file. In the case of this XC source and appropriate build of the XMOS tools can be used as follows:

```
$ xcc -target=XK-1A -emit-llvm -c accumulate.xc -o accumulate.bc
```

Analysis Next, run `inferCR.py` against the bitcode file and a mapping between the LLVM IR and underlying ISA energy model (Section 2):

```
$ inferCR.py accumulate.bc accumulate-mapping.json
eq(accumulate(In0,In1,In2),0,[f0allocas(In0,In1,In2)],[]).
eq(f0allocas(In0,In1,In2),52,[f0ifdone],[]).
eq(f0allocas(In0,In1,In2),52,
    [f0loopbody(In1 + -1,In1 + -1),f0ifdone],[In1 + -1 >= 1]).
eq(f0ifdone,58,[],[]).
eq(f0loopbody(Vboptmp,Vi0),104,[f0ifdone9(Vboptmp,Vi0)],[]).
eq(f0loopbody(Vboptmp,Vi0),104,
    [f0loopbody15(Vboptmp,Vboptmp,Vi0),
    f0ifdone9(Vboptmp,Vi0)],[]).
eq(f0ifdone9(Vboptmp,Vi0),97,[],[]).
eq(f0ifdone9(Vboptmp,Vi0),97,
    [f0loopbody(Vboptmp,Vi0 + -1)],[Vi0 + -1 >= 1]).
eq(f0loopbody15(Vboptmp,Vj0,Vi0),130,[],[]).
eq(f0loopbody15(Vboptmp,Vj0,Vi0),130,
    [f0loopbody15(Vboptmp,Vj0 + -1,Vi0)],[Vj0 + -1 >= 1]).
```

This output can then be used with a solver, for example PUBS, to yield the upper bound of the cost function for the code, in terms of any input parameters. In this example, shown in Figure 11, only parameter B, the size of the list of numbers to accumulate, influences the cost. This is an intuitive example, as the cost of accumulating a list of values will be correlated to the length of that list.



```
CRS accumulate(A,B,C)

  * Non Asymptotic Upper Bound: 441+nat(B-2)* (331+130*nat(B-2))+130*nat(B-2)

  * LOOPS accumulate(D,E,F) -> accumulate(G,H,I)

  * Ranking function: N/A

  * Invariants accumulate(A,B,C) -> accumulate(D,E,F)

     entry  : []
     non-rec: [A=D,B=E,C=F]
     rec    : [0=1]
     inv    : [1*A+ -1*D=0,1*B+ -1*E=0,1*C+ -1*F=0]
```

Figure 11: Upper bound solution for the accumulate example.

6.2.3 Estimating costs for a range of inputs

With the cost function provided by the solver, an estimation of the energy cost of a function over a range of values for the relevant input parameters can be given. This may be a line/curve for single parameter cost functions, or a multidimensional surface for multi-parameter cases. A sample set of results in this form, along with an evaluation of the accuracy of the analysis when combined with energy models for two architectures, is given in Section 3.2.

References

[AAGP09] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Cost relation systems: A language-independent target language for cost analysis. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 248:31–46, August 2009.

[AAGP11] 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.

[BCF11] C. Brandolese, S. Corbetta, and W. Fornaciari. Software energy estimation based on statistical characterization of intermediate compilation code. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pages 333–338, Aug 2011.

[DDG$^+$02] Jan Dhaene, Michel Denuit, Marc J Goovaerts, R Kaas, and David Vyncke. The concept of comonotonicity in actuarial science and finance: Theory. *Insurance, mathematics & economics*, 31(2):133–161, 2002.

[DLGHL97] 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.

[DLH90] 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.

[DWA13] The dwarf debugging standard, October 2013. `http://dwarfstd.org/`.

[EG13] K. Eder and N. Grech, editors. *Common Assertion Language*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013. Deliverable 2.1, http://entraproject.eu.

[EKG14] K. Eder, S. Kerrison, and K. Georgiou, editors. *Low-Level Energy Models*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), May 2014. Deliverable 2.2, http://entraproject.eu.

[FKNP11] Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Automated verification techniques for probabilistic systems. In Marco Bernardo and

Valérie Issarny, editors, *SFM*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.

[FSZ91] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Automatic average-case analysis of algorithm. *Theor. Comput. Sci.*, 79(1):37–109, 1991.

[Gao13] Ang Gao. *Modular average case analysis: Language implementation and extension*. Ph.d. thesis, University College Cork, 2013.

[GBMH07] Xi Guo, Menouer Boubekeur, J. McEnery, and David Hickey. Acet based scheduling of soft real-time systems: An approach to optimise resource budgeting. *International Journal of Computers and Communications*, 1(1):82–86, 2007.

[GE14] K. Georgiou and K. Eder. Mapping an isa energy model to llvm ir. Technical report, University of Bristol, April 2014.

[HBC$^+$12] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012. http://arxiv.org/abs/1102.5497.

[HPBLG05] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2), 2005.

[Koz79] Dexter Kozen. Semantics of probabilistic programs. In *FOCS*, pages 101–114. IEEE Computer Society, 1979.

[Koz81] Dexter Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.

[LG13] P. López-García, editor. *A General Framework for Resource Consumption Analysis and Verification*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013. Deliverable 3.1, http://entraproject.eu.

[LKS$^+$13] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M.V. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Pre-proceedings of LOPSTR*, September 2013.

[Mon00] David Monniaux. Abstract interpretation of probabilistic semantics. In Jens Palsberg, editor, *SAS*, volume 1824 of *LNCS*, pages 322–339. Springer, 2000.

[PCG09] Hector Soza Pollman, Manuel Carro, and Pedro Lopez Garcia. Probabilistic cost analysis of logic programs: A first case study. *INGENIARE - Revista Chilena de Ingeniera*, 17(2):195–204, 2009.

[Ros89] M. Rosendahl. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*. ACM Press, 1989.

[SBYT12] Lars Schor, Iuliana Bacivarov, Hoeseok Yang, and Lothar Thiele. Worst-case temperature guarantees for real-time applications on multi-core systems. In Marco Di Natale, editor, *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 87–96. IEEE, 2012.

[Sch08] Michel P. Schellekens. *A modular calculus for the average cost of data structuring.* Springer, 2008.

[SLGH14] 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*, 2014. To Appear.

[WAT08] Adam Wierman, Lachlan L. H. Andrew, and Ao Tang. Stochastic analysis of power-aware scheduling. In *Proceedings of Allerton Conference on Communication, Control and Computing*. Urbana-Champaign, IL, 2008.

[Weg75] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

Attachments

Attachment D3.2.1

Probabilistic Output Analysis

Submitted to the 21st International Static Analysis Symposium (SAS 2014).

Probabilistic Output Analysis

Mads Rosendahl[⋆] and Maja Kirkeby[⋆⋆]

Computer Science, Roskilde University
Roskilde, Denmark
`madsr@ruc.dk`, `majaht@ruc.dk`

Abstract. The aim of a probabilistic output analysis is to derive a probability distribution of possible output values for a program from a probability distribution of its input. We present a method for performing the output analysis, based on program transformation techniques. It generates a probability function as a possibly uncomputable expression and transforms that into a closed form expression. The probability functions are viewed as programs in a separate language in which they may be analysed, transformed, and approximated. We focus on programs in a deterministic language where the possible input follows a known probability distribution. Tests in programs are not assumed to satisfy the Markov property of having fixed branching probabilities independently of previous history.

1 Introduction

The aim of a probabilistic output analysis (POA) is to derive a probability distribution for output values from a probability distribution for input to a program. Internal properties of a program can also be analysed in this way by instrumenting programs with step-counters for complexity analysis [21] or energy consumption measures [15]. When we analyse programs for this type of property we should externalise the resource usage so that it is a denotational property of the analysed program.

When analysing energy consumption, probability distributions may provide more useful information than boundaries. Wierman et al. states that *"global energy consumption is affected by the average case, rather than the worst case"* [27]. Also in scheduling *"an accurate measurement of a tasks average-case execution time (ACET) can assist in the calculation of more appropriate deadlines"* [9]. For a subset of programs a precise average case execution time can be found using static analysis [6, 23, 8]. In some cases the POA delivers not only an accurate output average but the more descriptive accurate output distribution. In other cases the POA must over-approximate the probability distribution and the

[⋆] The research was partially funded by Danish National Science Foundation: Numeric and Symbolic Abstractions in Software

[⋆⋆] The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 318337, ENTRA - Whole-Systems Energy Transparency.

expected value (average case result) will be approximated safely as a range. Another application area for POA is in temperature management, where worst-case bounds are important [24]. Because POA return distributions it can be used to calculate the probability energy consumptions above a certain limit, and thereby indicating the risk of over-heating.

When analysing probabilities the main challenge is to maintain the dependencies throughout the program. Schellekens defines this as *Randomness preservation* [23] (or random bag preservation) which in his (and Gao's [8]) case enables tracking of certain data structures and their distributions. They use special data structures as they find these suitable to derive the average number of basic operations. In another approach [26, 20], tests in programs has been assumed to be independent of previous history, also known as the Markov property (the probability of true is fixed). As Wegbreit remarked, this is true only for some programs (e.g. linear search for repeating lists) and others, this is not the case (linear search for non-repeating lists). The Markov property is the foundation in Markov decision processes which is used in probabilistic model-checking [7]. Cousot et al. presents a probabilistic abstraction framework where they divide the program semantics into probabilistic behaviour and (non-)deterministic behaviour. They propose handling of tests when it is possible to assume the Markov property, and handle loops by using a probability function describing the probability of entering the loop in the ith iteration. Monniaux propose another approach for abstracting probabilistic semantics [17]; he first lifts a normal semantics to a probabilistic semantics where random generators are allowed and then uses an abstraction to reach a closed form. Monniaux's semantic approach uses a backward probabilistic semantics operating on measurable functions. This is closely related to the forward probabilistic semantics proposed earlier by Kozen [11, 12].

The method in this paper is inspired by the techniques used in automatic complexity analysis. Wegbreit's Metric system [26] laid the ground work for many later systems with an aim of deriving least, worst and average case complexity measures. Later works in this area have focused on worst case complexity [21, 1, 16] with advanced systems that can analyse realistic programs. The approach in this paper uses an approach similar to [21] in that we derive the probability function without approximations but only in the last phase introduce approximations. We transform the original program into a program that computes the probability distribution. The intermediate stage is then a potential subject of further analysis based on abstract interpretation. This program can be analysed, transformed, and approximated. It is thus an alternative to deriving cost relations directly from the program [1, 16] or expressing costs as abstract values in a semantics for the language.

To increase the expressiveness of the analysis, we can handle symbolic input distributions; e.g. it is possible to define the input distribution as uniform distribution from m to n, and where the output distribution is then expressed symbolically in terms of m and n. As a small example let us consider the addition `add` of two independent integer values `x` and `y` evenly distributed from 1 to

n. It is a tail-recursive program where the output distribution is well-known to be a pyramid shaped distribution.

```
add(x,y) = if(x=0) then y else add(x-1,y+1)
```

Our probabilistic output analysis returns a function describing the probability distribution of the output:

$$\mathrm{P}_{\texttt{add}}(z) = \frac{1}{n^2} \cdot \max(\min(n, z-1) - \max(1, z-n) + 1, 0)$$

The analysis can also be used for more complex input distributions and programs but it will not always be able to reduce it to closed form. If this is not possible we will approximate the distribution and thus get an over approximation of the extreme cases and a range for the expected value.

2 Probability distributions

The analysis presented here is based on using a discrete set of values for input and output. The set will be finite or countable and we will use discrete probability distributions. It is also possible to use an uncountable set of values or a combination of discrete and continuous random variables if one uses cumulative probability measures in the analysis. This will be discussed further later in the paper.

We consider the input to a program as a discrete random variable and the input probability distribution is then a probability measure that to an event of input having a given value assigns a value between 0 and 1. This is also often referred to as the *probability mass function* in the discrete case, and in the continuous case one will use a *probability density functions*. We will use the phrase *probability function* to denote mappings from single values (input or output) to a probability or number between 0 and 1, and we will use upper case P letters to denote such functions.

Definition 1 (input probability). *For a countable set X an input probability function is a mapping $P_X : X \to \{r \in \mathbb{R} \mid 0 \leq r \leq 1\}$, where*

$$\sum_{x \in X} P_X(x) = 1$$

We define the output probability distribution for a program `p` in a forward manner. It is the *weight* or sum of all probabilities of input values where the program returns the desired value z as output.

Definition 2 (output probability). *Given a program, $\texttt{p} : X \to Z$ and a probability distribution for the input, P_X, the output probability function, $P_{\texttt{p}}(z)$, is defined as:*

$$P_{\texttt{p}}(z) = \sum_{x \in X \wedge \texttt{p}(x) = z} P_X(x)$$

Note that Kozen also uses a similar forward definition [11, 12], whereas Monniaux constructs the inverse and express the relationship in a backwards style [17].

Lemma 1. *The output probability distribution, $P_{\mathtt{p}}(z)$, satisfies*

$$0 \leq \sum_z P_{\mathtt{p}}(z) \leq 1$$

The program may not terminate for all input and this means that the sum may be less than one. If we expand the domain Z with an element to denote non-termination, $Z_\bot$, the total sum of the output distribution $P_{\mathtt{p}}(z)$ would be 1.

An analysis may not be able to derive the exact output probability for all programs but we can always derive over and under approximations of the distribution.

Definition 3 (over and under approximation). *For a distribution $P_{\mathtt{p}}$ an over and under approximation ($P^\sharp$ and $P^\flat$) of the distribution satisfies the conditions:*

$$P^\sharp_{\mathtt{p}} : \forall z. P_{\mathtt{p}}(z) \leq P^\sharp_{\mathtt{p}}(z) \leq 1 \qquad P^\flat_{\mathtt{p}} : \forall z. 0 \leq P^\flat_{\mathtt{p}}(z) \leq P_{\mathtt{p}}(z)$$

The notation is naturally inspired by Alan Mycroft's strictness analysis. The aim of the output analysis is to derive as tight approximations $P^\flat$ and $P^\sharp$ as possible.

Lemma 2. *Given the definition for over and under approximation they will have boundaries for their total weights as*

$$0 \leq \sum_z P^\sharp_{\mathtt{p}}(z) \leq \infty \qquad 0 \leq \sum_z P^\flat_{\mathtt{p}}(z) \leq 1$$

When $P^\flat_{\mathtt{p}} = P^\sharp_{\mathtt{p}}$ the total weight for each function will be equal to the total weight of $P_{\mathtt{p}}$, according to definition 3. For terminating programs the total weight is 1.

Expected value. Provided that the output from the program is numerical, one may be interested in the average output value of the program. In this context this is *the expected value* of the output distribution. If the program does not terminate for all input it is not clear how to define the expected value because non-termination may indicate a possibly infinite output value. As part of the further analysis we need a guarantee that the program terminates. If the sum of the $P^\flat_{\mathtt{p}}$ is one then we know that the program terminates for all possible input (*i.e.* input with probability greater than zero). We may be able to guarantee termination of the program without being able to derive a tight bound for the under approximation $P^\flat_{\mathtt{p}}$.

Lemma 3. *The under approximation of a probability distribution satisfies*

$$\sum_z P^\flat_{\mathtt{p}}(z) = 1 \Rightarrow \sum_z P_{\mathtt{p}}(z) = 1$$

The expected value of the output distribution is defined as the weighted average of the distribution.

Definition 4 (expected value). *The expected value of the output distribution is defined as*

$$E_{\mathtt{p}} = \sum_{z} z \cdot P_{\mathtt{p}}(z)$$

If we cannot analyse the program precisely, we can use the over approximation to compute an interval for the expected value. Using $P^{\sharp}_{\mathtt{p}}$ we can create two new probability distributions, each with a total weight of 1. One that favours the lower values, and one that favours the higher values. These two can then be used to calculate a lower and an upper bound for the expected values.

Definition 5 (expected value interval). *For an over approximation of a probability distribution $P^{\sharp}_{\mathtt{p}}$ we define an over and under accumulation ($F^{\uparrow}$ and $F^{\downarrow}$) and over and under expected value ($E^{\uparrow}$ and $E^{\downarrow}$).*

$$F^{\uparrow}(z) = \min(\sum_{v \leq z} P^{\sharp}_{\mathtt{p}}(v), 1) \qquad F^{\downarrow}(z) = \max(1 - \sum_{v \geq z} P^{\sharp}_{\mathtt{p}}(v), 0)$$

$$E^{\uparrow} = \sum_{z} z \cdot (F^{\uparrow}(z) - F^{\uparrow}(dec(z))) \qquad E^{\downarrow} = \sum_{z} z \cdot (F^{\downarrow}(z) - F^{\downarrow}(dec(z)))$$

$$dec(z) = \max\{v \in Z \mid v < x\}$$

Lemma 4 (expected value interval). *For a terminating program the expected value can be approximated by an interval from the over approximation of the probability distribution.*

$$E^{\downarrow} \leq E_{\mathtt{p}} \leq E^{\uparrow}$$

Externalise resource usage. The output analysis can be used to analyse internal properties of the program provided these properties are externalised. As in automatic worst case complexity analysis [21], this may be done by instrumenting the program with step counting information. Similarly we might instrument programs with energy consumption based on low level energy models for operations [15] to be able to analyse programs for average energy consumption.

The challenge of approximation. Analysis of probabilistic behaviour introduces some new challenges compared to worst case analysis. It is well known that a function of expected values is not necessarily the same as the expected value of the function. There are a number of other potential pitfalls when making approximations in a probabilistic setting. One might assume that conditions in a program can be assigned a fixed probability of being true independently of previous execution paths in the program. One might also assume that variables have independent probability distributions. An unfortunate effect of using independence as an approximation is that it tends to under approximate the extreme cases. In a throw of two dice the sum of 12 has probability 1/36 if we can assume independence. If (by some magic) they always showed the same the probability increases to 1/6. The situation is well-known in the insurance industry and for financial risk management (valuation of derivatives) where one may want to over approximate the risk of extreme event when events are not guaranteed to be independent. One approach to handle such situations is the use of copulas [2] and comonotonicity of probability measures [5].

3 Transformation Based Analysis

Our analysis is based on a small first order functional language with primitive recursion. The first step of the analysis is to translate programs into a new language of probability distribution programs. We will then use analysis and transformation techniques to transform the probability distributions into closed form. Failing that, we may approximate the distribution with an upper and lower approximation ($P^\sharp$ and $P^\flat$).

Programs have the form of a collection of functions

$$\begin{aligned} &f_1(\vec{x}) = e_1 \\ &\vdots \\ &f_n(\vec{x}) = e_n \end{aligned}$$

The language uses a base set D of values for simple expressions, and functions in a program denotes mappings from tuples of values to tuples of values $D^* \to D^*$. Parameters that represent tuples are written with an arrow symbol $\vec{x}$ or as list of variable names in angled brackets $\langle x_1, \ldots, x_n \rangle$. We do not distinguish between singleton tuples and values and may view a function argument tuple as multiple arguments. The base set of values will not be further restricted here, nor do we specify the exact set of basic operations in the language. The first function in the program is called externally and for that function we have an input probability distribution P_x specified as a symbolic expression e_x.

3.1 Probability distribution program

When constructing the probability distribution program, in its raw form, we use two new language constructs: A sum over the (possibly) infinite set of all input values and a constraint function C. The constraint function eases the handling of boundaries and is defined as

$$C(\mathit{condition}) = \begin{cases} 1 & \text{if } \mathit{condition} = \mathit{true} \\ 0 & \text{otherwise} \end{cases}$$

Given an output value tuple, the distribution program sums the probabilities for all input value tuples that the original program maps to the output value tuple.

The probability distribution program is defined as follows.

$$\begin{aligned}
&P_f(\vec{z}) = \sum_{\vec{x}} P_x(\vec{x}) \cdot C(\vec{z} = f_1(\vec{x})) \\
&P_x(\vec{x}) = e_x \\
&f_1(\vec{x}) = e_1 \\
&\vdots \\
&f_n(\vec{x}) = e_n
\end{aligned}$$

We interpret a probability distribution program as a program that can be transformed and analysed. The second phase is to unfold function calls and the following phase is to try to remove the infinite summations.

3.2 Unfolding

The this phase we unfold function calls in the program. For simplicity we assume functions in the original program to have one of the following five forms:

$$\begin{aligned}
&f(\vec{x}) = e \\
&f(\vec{x}) = \langle e_1, \ldots, e_n \rangle \\
&f(\vec{x}) = g(h(\vec{x})) \\
&f(\vec{x}) = \textbf{ if } (b(x)) \textbf{ then } g(\vec{x}) \textbf{ else } h(\vec{x}) \\
&f(\vec{x}) = \textbf{ if } (b(x)) \textbf{ then } g(\vec{x}) \textbf{ else } f(h(\vec{x}))
\end{aligned}$$

where b, g, and h are functions in the program, and other expressions e are simple expressions that do not contain function calls. The only way recursion can appear is through primitive recursion in the fifth and last form. We assume that b evaluates to a boolean value. In the following we will introduce the central transformation rules for unfolding calls to function in the original program.

Tuples. For tupling expressions we separate the tests into the elements of the tuple. The transformation is

$$\sum_{\vec{x}} P(\vec{x}) \cdot C(\vec{z} = \langle e_1, \ldots, e_n \rangle) = \sum_{\vec{x}} P(\vec{x}) \cdot C(z_1 = e_1) \cdots C(z_n = e_n)$$

where $\vec{z} = \langle z_1, \ldots, z_n \rangle$ and P is a probability function in the program.

Function composition. For function compositions we rewrite the program as follows

$$\sum_{\vec{x}} P(\vec{x}) \cdot C(\vec{z} = g(h(\vec{z}))) = \sum_{\vec{x}} P_{hP}(\vec{x}) \cdot C(\vec{z} = g(\vec{x}))$$

where

$$P_{hP}(\vec{z}) = \sum_{\vec{x}} P(\vec{x}) \cdot C(\vec{z} = h(\vec{x}))$$

This rule extends the program with an extra probability function P_{hP}. As we assume the programs do not have unrestricted recursion we will only generate a bounded number of extra probability functions.

Conditional expressions. For conditional expressions we use the following rule

$$\begin{aligned}&\sum_{\vec{x}} P(\vec{x}) \cdot C(\vec{z} = \textbf{if } (b(\vec{x})) \textbf{ then } g(\vec{x}) \textbf{ else } h(\vec{x}))\\ &= \sum_{\vec{x}} P(\vec{x}) \cdot (P_{bP}(\mathit{true}) \cdot P_{gP}(\vec{x}) + P_{bP}(\mathit{false}) \cdot P_{hP}(\vec{x}))\end{aligned}$$

where P_{bP}, P_{gP}, and P_{hP} are output probability functions for b, g, and h with input probability P. If b, g, and h are all defined as simple expressions e_1, e_2, and e_3, respectively, this may be simplified as

$$\begin{aligned}&\sum_{\vec{x}} P(\vec{x}) \cdot C(\vec{z} = \textbf{if } (e_1) \textbf{ then } e_2 \textbf{ else } e_3)\\ &= \sum_{\vec{x}} P(\vec{x}) \cdot (C(e_1) \cdot C(\vec{z} = e_2) + C(\neg e_1) \cdot C(\vec{z} = e_3))\end{aligned}$$

Unfolding primitive recursion. For primitive recursion we collect the probability of a given result being returned for any number of recursive calls. Following the fifth of the allowed program forms, a primitive recursive function, f, calls itself recursively when the condition is false and it terminates the first time the condition evaluates to true. The condition may never evaluate to true for a certain input (non-termination), and in that situation the sum of output probabilities will be less than 1.

The transformation for the primitive recursion form is

$$\begin{aligned}&\sum_{\vec{x}} P(\vec{x}) \cdot C\,(\vec{z} = \textbf{if } (b(\vec{x})) \textbf{ then } g(\vec{x}) \textbf{ else } f(h(\vec{x})))\\ &= \sum_{\vec{x}} P(\vec{x}) \sum_{i=0}^{\infty} \prod_{j=0}^{i-1} C(\neg b(h'(j, \vec{x}))) \cdot C(b(h'(i, \vec{x}))) \cdot C(\vec{z} = g(h'(i, \vec{x})))\end{aligned}$$

where

$$h'(i, \vec{x}) = \textbf{if } (i = 0) \textbf{ then } \vec{x} \textbf{ else } h'(i-1, h(\vec{x}))$$

In the transformed expression we introduce two variables: i that represents the number of recursive calls, and j that represents all previous recursions for the i under investigation (when i is 0 the term $\prod_{j=0}^{i-1} C(\neg b(h'(j, \vec{x})))$ evaluates to 1). The new function $h'(i, \vec{x})$ relates to $h(\vec{x})$ and corresponds to calling h on itself i times (e.g. $h(2, \vec{x}) = h(h(\vec{x}))$). The result can be given to b and allows us to evaluate the condition for the ith and jth recursive call. Only when the ith condition is true ($C(b(h'(i, \vec{x}))) = 1$) and all previous conditions are false ($\prod_{j=0}^{i-1} C(\neg b(h'(j, \vec{x}))) = 1$), then can the expression evaluate to a probability above 0 (when $z = g(h'(i, \vec{x}))$).

3.3 Symbolic summation

In the previous phase we unfolded calls to functions in the original program. The aim of the next phase use algebraic transformation techniques to remove summations. The methods we use are similar to the transformations used in worst case execution time system for solving recurrence equations [22, 16] or symbolic summation techniques in loop bound computations [10]. Some of the central transformation rules we use in this phase are listed below. In the following transformations the expressions e_1 and e_2 are assumed not to contain the summation variable x.

$$\sum_x C(x = e_1) \cdot f(x) = f(e_1)$$

$$\sum_x C(e_1 \leq x \leq e_2) = (e_2 - e_1 + 1) \cdot C(e_1 \leq e_2)$$

$$\sum_x x \cdot C(e_1 \leq x \leq e_2) = \left(\frac{e_2 \cdot (e_2 + 1)}{2} - \frac{e_1 \cdot (e_1 - 1)}{2} \right) \cdot C(e_1 \leq e_2)$$

One could also use computer algebra systems in the reduction process but some of the rules are quite specific to way we handle the boundaries of summations with the special constraint function.

$$C(e_1 \leq x \leq e_2) \cdot C(e_3 \leq x \leq e_4) = C(\max(e_1, e_3) \leq x \leq \min(e_2, e_4))$$

$$C(\max(e_1, e_2) \leq e_3) = C(e_1 > e_2) \cdot C(e_1 \leq e_3) + C(e_1 \leq e_2) \cdot C(e_2 \leq e_3)$$

There are similar rules for removing the minimum function and for isolating variables.

Max example. As a small example, let us look at the simple non-recursive program `max`, which given two values return the largest. It is chosen because it only makes use of the symbolic summation rules and that the output follows a non-uniform distribution even if the input variables are uniformly distributed. The program is defined as

```
max(x,y) = if (x>y) then x else y
```

The input probabilities are independent, each is a uniform distribution from 1 to n and can be defined as

$$P_x(x) = \frac{1}{n} \cdot C(1 \leq x \leq n) \qquad \text{and} \qquad P_y(y) = \frac{1}{n} \cdot C(1 \leq y \leq n)$$

The following example uses the conditional transformation rule and the symbolic summation rules.

$$
\begin{aligned}
P_{\texttt{max}}(z) &= \sum_{\langle x,y\rangle} P_{\langle x,y\rangle}(\langle x,y\rangle)\cdot C(z=\texttt{max}(x,y))\\
&= \sum_x \sum_y P_x(x)\cdot P_y(y)\cdot C(z=\texttt{if}\ (x>y)\ \texttt{then}\ x\ \texttt{else}\ y)\\
&= \frac{1}{n^2}\cdot\Big(\quad \sum_y \big(C(1\le z\le n)\cdot C(1\le y\le n)\cdot C(y\le (z-1))\big)\\
&\qquad\quad +\sum_x \big(C(1\le x\le n)\cdot C(1\le z\le n)\cdot C(x\le z)\big)\Big)\\
&= \frac{1}{n^2}\cdot(2z-1)\cdot C(1\le z\le n)
\end{aligned}
$$

Add example. The recursive addition function was used as an example in the introduction. We shall see how the original program is inserted into the probability formula, expanded and reduced to a closed form function expressing the probability distribution for the output. Recall the program:

```
add(x,y) = if(x=0) then y else add(x-1,y+1)
```

and that we assume independence between the input variables and for the sake of simplicity we let both input variables `x` and `y` have a uniform distribution from 1 to a number n.

$$
\begin{aligned}
P_{\texttt{add}}(z) &= \sum_x \sum_y P_x(x)\cdot P_y(y)\cdot\\
&\sum_{i=0}\prod_{j=0}^{i-1} C(\neg b(h'(j,x,y)))\cdot C(b(h'(i,x,y)))\cdot C(z=g(h'(i,x,y))
\end{aligned}
$$

where

$$
\begin{aligned}
b(x,y) &= x=0\\
g(x,y) &= y\\
h(\langle x,y\rangle) &= \langle x-1,y+1\rangle\\
h'(i,\langle x,y\rangle) &= \textbf{ if } (i=0) \textbf{ then } \langle x,y\rangle \textbf{ else } h'(i-1,\langle x-1,y+1\rangle)\\
&= \langle x-i,y+i\rangle
\end{aligned}
$$

$$
\begin{aligned}
P_{\mathtt{add}}(z) &= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot \\
&\qquad \sum_{i=0} \prod_{j=0}^{i-1} C(\neg(x-j=0)) \cdot C(x-i=0) \cdot C(z=y+i) \\
&= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot \sum_{i=0} C(x=i) \cdot C(z=y+i) \\
&= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot C(z=y+x) \\
&= \sum_x \sum_y \frac{1}{n} \cdot C(1 \le x \le n) \cdot \frac{1}{n} \cdot C(1 \le y \le n) \cdot C(z=y+x) \\
&= \sum_y \frac{1}{n} \cdot C(z-n \le y \le z-1) \cdot \frac{1}{n} \cdot C(1 \le y \le n)\big) \\
&= \frac{1}{n^2} \cdot \max(\min(n, z-1) - \max(1, z-n) + 1, 0) \\
&= \frac{1}{n^2} \cdot \big(C(n < z \le 2n) \cdot (2n-z+1) + C(1 \le z \le n) \cdot (z-1)\big)
\end{aligned}
$$

3.4 Expected value

If we have derived a probability program, we may also derive an expression that computes the expected value of the distribution.

$$E_p = \sum_x x \cdot P_p(x)$$

For the `add` program this gives

$$E_{\mathtt{add}} = \sum_{z=1}^{n} z \cdot (z-1) + \sum_{z=n+1}^{2n} z \cdot (2n-z+1)$$

which, of course, can be reduced further.

4 Composite Types

In the approach we have presented the base domain is a countable set and not necessarily just numbers. We only need to be able to define a probability distribution for values in the domain.

For lists of length $k > 0$ where elements are uniformly distributed over the interval 1 to n we can use the probability function

$$P_L(L) = \frac{1}{n^k} \cdot C(\mathit{length}(L) = k \wedge \forall j : 0 \le j \le k-1 \wedge 1 \le \mathit{hd}(\mathit{tl}^j(L)) \le n)$$

We assign the probability $1/n^k$ to any list of length k where all elements are in the interval from 1 to n.

If we consider the member function for non-empty lists, it can be written as

```
member(X,L) = if(tl(L)=[] || hd(L)=X) then hd(L)=X
              else member(X,tl(L))
```

The function will follow the pattern of primitive recursion as described earlier and the output probability function for the member function is then

$$P_{\texttt{member}}(z) = \sum_{X} \sum_{L} P_X(X) \cdot P_L(L) \cdot C(z = \texttt{member}(X, L))$$

We can then use the unfolding rules to simplify the expression further.

The lists were here assumed to contain possibly repeating elements in the list. We could also use a different probability measure to restrict lists to non-repeating lists of values. This restriction is made by Wegbreit [26] in his examples, where the probability is derived as $1 - (1 - (1/n)^k)$, which is the correct result for repeating lists of values. His technique is valid for programs where one can safely assume the Markov property (that probability of conditions are fixed). Wegbreit observes that this is not always true even in very simple cases, e.g. in nested conditionals where the outcome of the first condition influences the probability of the outcome for the subsequent condition.

It should be noted that conditions existing inside a recursive structure often invokes dependencies between variables. This occur when there is a *gain of knowledge*: For instance in the `union` function for two repeating lists; if the head of the list is not in the second list, the likelihood of the next element not being in the second list increases slightly.

5 Approximation Techniques

The probability distribution program expresses the probability distribution for output values. Our aim is to transform it into a closed form but this may not always be possible. Failing that, we can instead use approximation techniques to obtain an upper bound for the probability distribution. We have referred to this as the over approximation of the probability distribution, $P^\sharp$. The techniques we may apply here are similar to automatic worst case complexity analysis [21] where the aim is to obtain a closed form expression for the complexity of programs but failing that may obtain an over approximation.

Cumulative distribution functions. Cumulative probabilities will in some cases be more useful and expressive than probability distributions: Cumulative probabilities can be used in both the discrete and the continuous case, and in some cases approximations can be described more precisely using accumulated probabilities than with ordinary distributions.

Definition 6. *Given a program output probability distribution, $P_{\texttt{p}}(z)$, the cumulative program output probability distribution, $F_{\texttt{p}}(z)$, is defined as*

$$F_{\texttt{p}}(z) = \sum_{w \leq z} P_{\texttt{p}}(w)$$

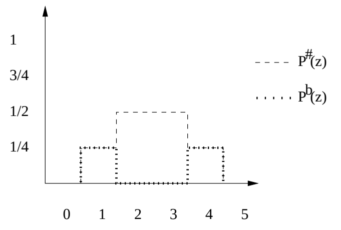
Definition 7 (over and under-approximation). *Given an accumulated output probability of a program* P*,* $F_{\mathtt{P}}$*, the over-approximation,* $F^{\sharp}_{\mathtt{P}}$*, and the under-approximation,* $F^{\flat}_{\mathtt{P}}$*, are defined as*

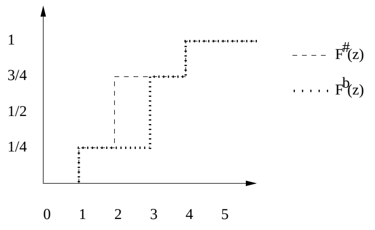
$$F^{\sharp}_{\mathtt{P}} : \forall z. F_{\mathtt{P}}(z) \leq F^{\sharp}_{\mathtt{P}}(z)$$
$$F^{\flat}_{\mathtt{P}} : \forall z. F^{\flat}_{\mathtt{P}}(z) \leq F_{\mathtt{P}}(z)$$

where for each approximation it must always hold that

$$\forall z. 0 \leq F^{\sharp}_{\mathtt{P}}(z) \leq 1$$
$$\forall z. 0 \leq F^{\flat}_{\mathtt{P}}(z) \leq 1$$

When we can deduce that a program may return one of two values, but not which one, then the cumulative probability can be used for a more precise description. Such a program could be `if x = 1 then 1 else (if x = 4 then 4 else (if (unanalysable) then 2 else 3))` and $1 \leq \mathtt{x} \leq 4$ with the probability distribution $P_x(x) = 1/4 \cdot C(1 \leq x \leq 4)$.





Here, the under approximating and over approximating distributions will assign 0 and 1/2 for both 2 and 3, respectively. In contrast, the under-approximating cumulative distribution can express that if the program-output is not 2 it must be 3.

$P^{\flat}_{\mathtt{P}}$ and $P^{\sharp}_{\mathtt{P}}$ can be used to derive $F^{\flat}_{\mathtt{P}}$ and $F^{\sharp}_{\mathtt{P}}$. However, these may not be as precise as cumulative distributions derived directly.

Approximations for accumulated probability functions.

When approximating accumulated probability functions the techniques are different from probability mass functions. Instead one may use copulas [2] to over and under approximate dependencies between subexpressions. Copulas are based on the theory of comonotonicity [5] for distributions that may depend on a common (possibly unknown) random variable.

6 Related Work

Probabilistic analysis is different from analysis of probabilistic programs. Probabilistic analysis is analysis of programs with a normal semantics where the input variables are interpreted over probability distributions. Analysis of probabilistic programs analyse programs with probabilistic semantics where the values of the input variables are unknown (e.g. flow analysis [19]).

In probabilistic analysis it is important to determine how variables depend on each other, but already in 1976 Denning proposed a flow analysis for revealing whether variables depend on each other [4]. This was presented in the field of secure flow analysis. Denning introduced a lattice-based analysis where she, given the name of a variable, that should be kept secret, deducted which other variables those should be kept secret in order to avoid leaking information. In 1996, Denning's method was refined by Volpano et al. into a type system and for the first time, it was proven sound [25].

Reasoning about probabilistic semantics is a closely related area to probabilistic analysis, as they both work with nested probabilistic influence. The probabilistic analysis work on standard semantic and analyse it using input probability distributions, where a probabilistic semantics allow for random assignments and probabilistic choices [11] and is normally analysed using any expanded classical analysis or verification method [3].

Probabilistic model checking is an automated technique for formally verifying quantitative properties for systems with probabilistic behaviours. It is mainly focused on Markov decision processes, which can model both stochastic and nondeterministic behaviour [7, 13, 14]. It differs from probabilistic analysis as it assumes the Markov property.

In 2000, Monniaux applied abstract interpretation to programs with probabilistic semantics and gained safe bounds for worst case analysis [17]. Pierro et al. introduce a linear mapping structure, a Moore-Penrose pseudo-inverse, instead of a Galois connection. They use the linear structures to compare 'closeness' of approximations as an expression using the average approximation error. Pierro et al. further explores using probabilistic abstract interpretation to calculate the average case analysis [18]. In 2012, Cousot and Monerau gave a general probabilistic abstraction framework [3] and stated, in section 5.3, that Pierro et al.'s method and many other abstraction methods can be expressed in this new framework.

7 Conclusion

Probabilistic analysis of program have a renewed interest for analysing programs for energy consumptions. Numerous embedded systems and mobile applications are limited by restricted battery life on the hardware. In this paper we present a technique for extracting a probability distribution for programs from symbolic distributions of the input. It is a transformation based method, where we analyse a first order language with primitive recursion. From the original program we generate an equivalent probability distribution program, and transform this program into closed form. We present the essential transformation rules for unfolding calls to the original program and removing infinite sums. The transformed program may then be analysed and approximated using program analysis and transformation techniques known from automatic complexity analysis. The core elements of the analysis have been implemented in a prototype system with the aim of using it to improve energy efficiency of systems. The central challenges of approximating in a probabilistic setting are discussed and we describe some advantages of using cumulative distributions along with copulas to achieve a tighter approximation.

Acknowledgements. This work has benefitted from numerous discussions with Pedro López-García, Alejandro Serrano Mena and other collegues in Madrid, Bristol and Roskilde.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Cost relation systems: A language-independent target language for cost analysis. Electr. Notes Theor. Comput. Sci. **248** (2009) 31–46
2. Bernat, G., Burns, A., Newby, M.: Probabilistic timing analysis: An approach using copulas. J. Embedded Computing **1**(2) (2005) 179–194
3. Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In Seidl, H., ed.: ESOP. Volume 7211 of LNCS., Springer (2012) 169–193
4. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5) (1976) 236–243
5. Dhaene, J., Denuit, M., Goovaerts, M.J., Kaas, R., Vyncke, D.: The concept of comonotonicity in actuarial science and finance: Theory. Insurance, mathematics & economics **31**(2) (2002) 133–161
6. Flajolet, P., Salvy, B., Zimmermann, P.: Automatic average-case analysis of algorithm. Theor. Comput. Sci. **79**(1) (1991) 37–109
7. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In Bernardo, M., Issarny, V., eds.: SFM. Volume 6659 of LNCS., Springer (2011) 53–113
8. Gao, A.: Modular average case analysis: Language implementation and extension. Ph.d. thesis, University College Cork (2013)
9. Guo, X., Boubekeur, M., McEnery, J., Hickey, D.: Acet based scheduling of soft real-time systems: An approach to optimise resource budgeting. International Journal of Computers and Communications **1**(1) (2007) 82–86
10. Knoop, J., Kovács, L., Zwirchmayr, J.: Symbolic loop bound computation for wcet analysis. In Clarke, E.M., Virbitskaite, I., Voronkov, A., eds.: Ershov Memorial Conference. Volume 7162 of LNCS., Springer (2011) 227–242
11. Kozen, D.: Semantics of probabilistic programs. In: FOCS, IEEE Computer Society (1979) 101–114
12. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. **22**(3) (1981) 328–350
13. Kwiatkowska, M., Norman, G., Parker, D.: Advances and challenges of probabilistic model checking. In: 48th Annual Allerton Conference on Communication, Control, and Computing, IEEE (September 2010) 1691–1698
14. Kwiatkowska, M., Parker, D.: Advances in probabilistic model checking. Software Safety and Security: Tools for Analysis and Verification **33** (2012) 126
15. Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M.V., Eder, K.: Energy consumption analysis of programs based on xmos isa level models. In: 23rd International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR. (2013)

16. López-García, P., Darmawan, L., Bueno, F.: A framework for verification and debugging of resource usage properties: Resource usage verification. In Hermenegildo, M.V., Schaub, T., eds.: ICLP (Technical Communications). Volume 7 of LIPIcs., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010) 104–113
17. Monniaux, D.: Abstract interpretation of probabilistic semantics. In Palsberg, J., ed.: SAS. Volume 1824 of LNCS., Springer (2000) 322–339
18. Pierro, A.D., Hankin, C., Wiklicky, H.: Abstract interpretation for worst and average case analysis. In Reps, T.W., Sagiv, M., Bauer, J., eds.: Program Analysis and Compilation. Volume 4444 of LNCS., Springer (2006) 160–174
19. Pierro, A.D., Wiklicky, H., Puppis, G., Villa, T.: Probabilistic data flow analysis: a linear equational approach. In: Proceedings Fourth International Symposium. Volume 119., Open Publishing Association (2013) 150–165
20. Pollman, H.S., Carro, M., Garcia, P.L.: Probabilistic cost analysis of logic programs: A first case study. INGENIARE - Revista Chilena de Ingeniera **17**(2) (2009) 195–204
21. Rosendahl, M.: Automatic complexity analysis. In: FPCA. (1989) 144–156
22. Rosendahl, M.: Simple driving techniques. In Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H., eds.: The Essence of Computation. Volume 2566 of LNCS., Springer (2002) 404–419
23. Schellekens, M.P.: A modular calculus for the average cost of data structuring. Springer (2008)
24. Schor, L., Bacivarov, I., Yang, H., Thiele, L.: Worst-case temperature guarantees for real-time applications on multi-core systems. In Natale, M.D., ed.: IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE (2012) 87–96
25. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security **4**(2/3) (1996) 167–188
26. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9) (1975) 528–539
27. Wierman, A., Andrew, L.L.H., Tang, A.: Stochastic analysis of power-aware scheduling. In: Proceedings of Allerton Conference on Communication, Control and Computing. Urbana-Champaign, IL (2008)

Attachment D3.2.2

Tools for Constrained Horn Clause Verification

Accepted for publication as a technical communication in Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue, On-line Supplement.

Analysis and Transformation Tools for Constrained Horn Clause Verification∗

John P. Gallagher

Roskilde University, Denmark and IMDEA Software Institute, Madrid, Spain
(*e-mail:* `jpg@ruc.dk`)

Bishoksan Kafle

Roskilde University, Denmark
(*e-mail:* `kafle@ruc.dk`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Several techniques and tools have been developed for verification in Horn clauses with constraints over some background theory (CHC). Current CHC verification tools implement intricate algorithms and are limited to certain subclasses of CHC problems. Our aim in this work is to investigate the use of a combination of off-the-shelf techniques from the literature in analysis and transformation of Constraint Logic Programs (CLPs) to solve challenging CHC verification problems. We find that many problems can be solved using a combination of tools based on well-known techniques from abstract interpretation, semantics-preserving transformations, program specialisation and query-answer transformations. This gives insights into the design of automatic, more general CHC verification tools based on a library of components.

KEYWORDS: Constraint Logic Program, Constrained Horn Clause, Abstract Interpretation, Software Verification.

1 Introduction

CHCs provide a suitable intermediate form for expressing the semantics of a variety of programming languages (imperative, functional, concurrent, *etc.*) and computational models (state machines, transition systems, big- and small-step operational semantics, Petri nets, *etc.*). As a result it has been used as a target language for software verification. Recently there is a growing interest in CHC verification from both the logic programming and software verification communities, and several verification techniques and tools have been developed for CHC.

Pure CLPs are syntactically and semantically the same as CHC. The main difference is that sets of constrained Horn clauses are not necessarily intended for execution, but rather as specifications. From the point of view of verification, we do not distinguish between CHC and pure CLP. Much research has been carried out on the analysis

∗ The research leading to these results has received funding from the European Union 7th Framework Programme under grant agreement no. 318337, ENTRA - Whole-Systems Energy Transparency and the Danish Natural Science Research Council grant NUSA: Numerical and Symbolic Abstractions for Software Model Checking.

and transformation of CLP programs, typically for synthesising efficient programs or for inferring run-time properties of programs for the purpose of debugging, compile-time optimisations or program understanding. In this paper we investigate the application of this research to the CHC verification problem.

In Section 2 we define the CHC verification problem. In Section 3 we define a number of basic transformation and analysis components drawn from or inspired by the CLP literature. Section 4 discusses the role of these components in verification, illustrating them on an example problem. In Section 5 we construct a tool-chain out of these components and test it on a range of CHC verification benchmark problems. The results reported in this section represent one of the main contributions of this work. In Section 6 we propose possible extensions of the basic tool-chain and compare them with related work on CHC verification tool architectures. Finally in Section 7 we summarise the conclusions from this work.

2 Background: The CHC Verification Problem

A CHC is a first order predicate logic formula of the form $\forall(\phi \wedge B_1(X_1) \wedge \ldots \wedge B_k(X_k) \rightarrow H(X))$ $(k \geq 0)$, where ϕ is a conjunction of constraints with respect to some background theory, X_i, X are (possibly empty) vectors of distinct variables, $B_1, \ldots, B_k, H$ are predicate symbols, $H(X)$ is the head of the clause and $\phi \wedge B_1(X_1) \wedge \ldots \wedge B_k(X_k)$ is the body. Sometimes the clause is written $H(X) \leftarrow \phi \wedge B_1(X_1), \ldots, B_k(X_k)$ and in concrete examples it is written in the form `H :-` ϕ`,` $\mathtt{B_1(X_1),\ldots,B_k(X_k)}$. In examples, predicate symbols start with lowercase letters while we use uppercase letters for variables.

We assume here that the constraint theory is closed with respect to negation and that there is a distinguished predicate symbol false which is interpreted as false. In practice the predicate false only occurs in the head of clauses; we call clauses whose head is false *integrity constraints*, following the terminology of deductive databases. Thus the formula $\phi_1 \leftarrow \phi_2 \wedge B_1(X_1), \ldots, B_k(X_k)$ is equivalent to the formula $\mathsf{false} \leftarrow \neg\phi_1 \wedge \phi_2 \wedge B_1(X_1), \ldots, B_k(X_k)$. The latter might not be a CHC but can be converted to an equivalent set of CHCs by transforming the formula $\neg\phi_1$ and distributing any disjunctions that arise over the rest of the body. For example, the formula `X=Y :- p(X,Y)` is equivalent to the set of CHCs `false :- X>Y, p(X,Y)` and `false :- X<Y, p(X,Y)`. Integrity constraints can be viewed as safety properties. If a set of CHCs encodes the behaviour of some system, the bodies of integrity constraints represent unsafe states. Thus proving safety consists of showing that the bodies of integrity constraints are false in all models of the CHC clauses.

The CHC verification problem. To state this more formally, given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. In every interpretation of P, the predicate false is interpreted as false; hence $P \models \mathsf{false}$ if and only if P has no model. Equivalently P has a model if and only if $P \not\models \mathsf{false}$. It is clear that any model of P assigns false to the bodies of integrity constraints.

The verification problem can be formulated deductively rather than model-theoretically. Let the relation $P \vdash A$ denote that A is derivable from P using some proof procedure. If the proof procedure is sound and complete then $P \not\models A$ if and only if $P \not\vdash A$. So the verification problem is to show (using CLP terminology) that the computation of the goal $\leftarrow$ false in program P does not succeed using a complete proof procedure. Although in this work we follow the model-based formulation of the problem, we exploit the equivalence with the deductive formulation, which underlies, for example, the query-answer transformation and specialisation techniques to be presented.

2.1 Representation of Interpretations

An interpretation of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \mathcal{C}$ where A is an atomic formula $p(Z_1, \ldots, Z_n)$ where $Z_1, \ldots, Z_n$ are distinct variables and $\mathcal{C}$ is a constraint over $Z_1, \ldots, Z_n$. If $\mathcal{C}$ is true we write $A \leftarrow$ or just A. The constrained fact $A \leftarrow \mathcal{C}$ is shorthand for the set of variable-free facts $A\theta$ such that $\mathcal{C}\theta$ holds in the constraint theory, and an interpretation M denotes the set of all facts denoted by its elements; M assigns true to exactly those facts. $M_1 \subseteq M_2$ if the set of denoted facts of M_1 is contained in the set of denoted facts of M_2.

Minimal models. A model of a set of CHCs is an interpretation that satisfies each clause. There exists a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$ where P is the set of CHCs. $M[\![P]\!]$ can be computed as the least fixed point (lfp) of an immediate consequences operator, $T_P^{\mathcal{C}}$, which is an extension of the standard T_P operator from logic programming, extended to handle constraints. Furthermore $\mathsf{lfp}(T_P^{\mathcal{C}})$ can be computed as the limit of the ascending sequence of interpretations $\emptyset, T_P^{\mathcal{C}}(\emptyset), T_P^{\mathcal{C}}(T_P^{\mathcal{C}}(\emptyset)), \ldots$. For more details, see (Jaffar and Maher 1994). This sequence provides a basis for abstract interpretation of CHC clauses.

Proof by over-approximation of the minimal model. It is a standard theorem of CLP that the minimal model $M[\![P]\!]$ is equivalent to the set of atomic consequences of P. That is, $P \models p(v_1, \ldots, v_n)$ if and only if $p(v_1, \ldots, v_n) \in M[\![P]\!]$. Therefore, the CHC verification problem for P is equivalent to checking that $\mathsf{false} \notin M[\![P]\!]$. It is sufficient to find a set of constrained facts M' such that $M[\![P]\!] \subseteq M'$, where $\mathsf{false} \notin M'$. This technique is called proof by over-approximation of the minimal model.

3 Relevant tools for CHC Verification

In this section, we give a brief description of some relevant tools borrowed from the literature in analysis and transformation of CLP.

Unfolding. Let P be a set of CHCs and $c_0 \in P$ be $H(X) \leftarrow \mathcal{B}_1, p(Y), \mathcal{B}_2$ where $\mathcal{B}_1, \mathcal{B}_2$ are possibly empty conjunctions of atomic formulas and constraints. Let $\{c_1, \ldots, c_m\}$ be the set of clauses of P that have predicate p in the head, that is, $c_i = p(Z_i) \leftarrow \mathcal{D}_i$, where the variables of these clauses are standardised apart from the variables of c_0 and from each other. Then the result of unfolding c_0 on $p(Y)$ is the set of CHCs $P' = P \setminus \{c_0\} \cup \{c'_1, \ldots, c'_m\}$ where $c'_i = H(X) \leftarrow \mathcal{B}_1, Y = Z_i, \mathcal{D}_i, \mathcal{B}_2$. The equality $Y = Z_i$ stands for the conjunction of the equality of the respective elements of the vectors Y and Z_i. It is a standard result that unfolding a clause in P preserves P's minimal model (Pettorossi and Proietti 1999). In particular, $P \models \mathsf{false} \equiv P' \models \mathsf{false}$.

Specialisation. A set of CHCs P can be specialised with respect to a query. Assume A is an atomic formula; then we can derive a set P_A such that $P \models A \equiv P_A \models A$. P_A could be simpler than P, for instance, parts of P that are irrelevant to A could be omitted in P_A. In particular, the CHC verification problem for P_{false} and P are equivalent. There are many techniques in the CLP literature for deriving a specialised program P_A. Partial evaluation is a well-developed method (Gallagher 1993; Leuschel 1999).

We make use a form of specialisation know as forward slicing, more specifically redundant argument filtering (Leuschel and Sørensen 1996), in which predicate arguments can be removed if they do not affect a computation. Given a set of CHCs P and a query A, denote by P_A^{raf} the program obtained by applying the RAF algorithm from (Leuschel and Sørensen 1996) with respect to the goal A. We have the property that $P \models A \equiv P_A^{\mathsf{raf}} \models A$ and in particular that $P \models \mathsf{false} \equiv P_{\mathsf{false}}^{\mathsf{raf}} \models \mathsf{false}$.

Query-answer transformation. Given a set of CHCs P and an atomic query A, the query-answer transformation of P with respect to A is a set of CHCs which simulates the computation of the goal $\leftarrow A$ in P, using a left-to-right computation rule. Query-answer transformation is a generalisation of the magic set transformations for Datalog. For each predicate p, two new predicates p_{ans} and p_{query} are defined. For an atomic formula A, A_{ans} and A_{query} denote the replacement of A's predicate symbol p by p_{ans} and p_{query} respectively. Given a program P and query A, the idea is to derive a program P_A^{qa} with the following property $P \models A$ iff $P_A^{\mathsf{qa}} \models A_{ans}$. The A_{query} predicates represent calls in the computation tree generated during the execution of the goal. For more details see (Debray and Ramakrishnan 1994; Gallagher and de Waal 1993; Codish and Demoen 1993). In particular, $P_{\mathsf{false}}^{\mathsf{qa}} \models \mathsf{false}_{ans} \equiv P \models \mathsf{false}$, so we can transform a CHC verification problem to an equivalent CHC verification problem on the query-answer program generated with respect to the goal $\leftarrow \mathsf{false}$.

Predicate splitting. Let P be a set of CHCs and let $\{c_1, \ldots, c_m\}$ be the set of clauses in P having some given predicate p in the head, where $c_i = p(X) \leftarrow \mathcal{D}_i$. Let $C_1, \ldots, C_k$ be some partition of $\{c_1, \ldots, c_m\}$, where $C_j = \{c_{j_1}, \ldots, c_{j_{n_j}}\}$. Define k new predicates $p_1 \ldots p_k$, where p_j is defined by the bodies of clauses in partition C_j, namely $D_j = \{p_j(X) \leftarrow \mathcal{D}_{j_1}, \ldots, p_j(X) \leftarrow \mathcal{D}_{j_{n_j}}\}$. Finally, define k clauses $C_p = \{p(X) \leftarrow p_1(X), \ldots, p(X) \leftarrow p_k(X)\}$. Then we define a splitting transformation as follows.

1. Let $P' = P \setminus \{c_1, \ldots, c_m\} \cup C_p \cup D^1 \cup \ldots \cup D^k$.
2. Let P^{split} be the result of unfolding every clause in P' whose body contains $p(Y)$ with the clauses C_p.

In our applications, we use splitting to create separate predicates for clauses for a given predicate whose constraints are mutually exclusive. For example, given the clauses `new3(A,B) :- A=<99, new4(A,B)` and `new3(A,B) :- A>=100, new5(A,B)`, we produce two new predicates, since the constraints `A=<99` and `A>=100` are disjoint. The new predicates are defined by clauses `new3`$_1$`(A,B) :- A=<99, new4(A,B)` and `new3`$_2$`(A,B) :- A>=100, new5(A,B)`, and all calls to `new3` throughout the program are unfolded using these new clauses. Splitting has been used in the CLP literature to improve the precision of program analyses, for example in (Serebrenik and De Schreye 2001). In our case it improves the precision of the convex polyhedron analysis discussed below, since separate polyhedra will be maintained for each of the disjoint cases. The correctness of splitting can be shown using standard transformations that preserve the minimal model of the program (with respect to the predicates of the original program) (Pettorossi and Proietti 1999). Assuming that the predicate false is not split, we have that $P \models \mathsf{false} \equiv P^{\mathsf{split}} \models \mathsf{false}$.

Convex polyhedron approximation. Convex polyhedron analysis (Cousot and Halbwachs 1978) is a program analysis technique based on abstract interpretation (Cousot and Cousot 1977). When applied to a set of CHCs P it constructs an over-approximation M' of the minimal model of P, where M' contains at most one constrained fact $p(X) \leftarrow \mathcal{C}$ for each predicate p. The constraint $\mathcal{C}$ is a conjunction of linear inequalities, representing a convex polyhedron. The first application of convex polyhedron analysis to CLP was by Benoy and King (Benoy and King 1996). Since the domain of convex polyhedra contains infinite increasing chains, the use of a widening operator is needed to ensure convergence of the abstract interpretation. Furthermore much research has been done on improving the precision of widening operators. One techniques is known as widening-upto, or widening with thresholds (Halbwachs et al. 1994). A threshold is an assertion that is combined with a widening operator to improve its precision. Recently, a technique for deriving more effective thresholds was developed (Lakhdar-Chaouch et al. 2011), which we have found to be effective in experimental studies.

4 The role of CLP tools in verification

The techniques discussed in the previous section play various roles. The convex polyhedron analysis, together with the helper tool to derive threshold constraints, constructs an approximation of the minimal model of a set of CHCs. If false (or false_{ans}) is not in the approximate model, then the verification problem is solved. Otherwise the problem is not solved; in effect a "don't know" answer is returned. We have found that polyhedron analysis alone is seldom precise enough to solve non-trivial CHC verification problems; in combination with the other tools, it is very effective.

Unfolding can improve the structure of a program, removing some case of mutual recursion, or propagating constraints upwards towards the integrity constraints, and can improve the precision and performance of convex polyhedron analysis.

Specialisation can remove parts of theories not relevant to the verification problem, and can also propagate constraint downwards from the integrity constraints. Both of these have a beneficial effect on performance and precision of polyhedron analysis.

Analysis of a query-answer program (with respect to false) is in effect the search for a derivation tree for false. Its effectiveness in CHC verification problems is variable. It can sometimes worsen performance since the query-answer transformed program is larger and contains more recursive dependencies than the original. On the other hand, one seldom loses precision and it is often more effective in allowing constraints to be propagated upwards (through the *ans* predicates) and downwards (through the *query* predicates).

4.1 Application of the tools

We illustrate the tools on a running example (Figure 1), one of the benchmark suite of the VeriMAP system (De Angelis et al. 2013). The result of applying unfolding is shown in

```
new6(A,B) :- B=<99.
new5(A,B) :- B>=101.
new5(A,B) :- B=<100, new6(A,B).
new4(A,B) :- C=1+A, A=<49, new3(C,B).
new4(A,B) :- C=1+A, D=1+B, A>=50, new3(C,D).
new3(A,B) :- A=<99, new4(A,B).
new3(A,B) :- A>=100, new5(A,B).
false :- A=0, B=50, new3(A,B).
```

Fig. 1. The example program `MAP-disj.c.map.pl`

```
false :- A=0, B=50, new3(A,B).
new3(A,B) :- A=<99, C = 1+A, A=<49, new3(C,B).
new3(A,B) :- A=<99, C = 1+A, D = 1+B, A>=50, new3(C,D).
new3(A,B) :- A>=100, B>=101.
new3(A,B) :- A>=100, B=<100, B=<99.
```

Fig. 2. Result of unfolding `MAP-disj.c.map.pl`

Figure 2 (omitting the definitions of the unfolded predicates `new4, new5` and `new6`, which are no longer reachable from `false`. The unfolding strategy we adopt is the following: the predicate dependency graph of a program consists of the set of edges (p, q) such that there is clause where p is the predicate of the head and q is a predicate occurring in the body. We perform a depth-first search of the predicate dependency graph, starting from `false`, and identify the backward edges, namely those edges (p, q) where q is an ancestor of p in the depth-first search. We then unfold every body call whose predicate is not at the end of a backward edge. In Figure 1, we thus unfold calls to `new4, new5` and `new6`.

The query-answer transformation is applied to the program in Figure 2, with respect to the goal `false` resulting in the program shown in Figure 3. The model of the predicate `new3_query` corresponds to those calls to `new3` that are reachable from the call in the integrity constraint. Explicit representation of the query predicates permits more effective propagation of constraints from the integrity clauses during model approximation.

The splitting transformation is now applied to the program in Figure 3. We do not show the complete program, which contains 30 clauses. Figure 4 shows the split definition of `new3_query`, which is split since the last two clauses for `new3_query` in Figure 3 have mutually disjoint constraints, when projected onto the head variables.

A convex polyhedron approximation is then computed for the split program, after computing threshold constraints for the predicates. The resulting approximate model is shown in Figure 5 as a set of constrained facts. Since the model does not contain any

```
false_ans :- false_query, A=0, B=50, new3_ans(A,B).
new3_ans(A,B) :- new3_query(A,B), A=<99, C = 1+A, A=<49, new3_ans(C,B).
new3_ans(A,B) :- new3_query(A,B),A=<99,C is 1+A,D is 1+B, A>=50, new3_ans(C,D).
new3_ans(A,B) :- new3_query(A,B), A>=100, B>=101.
new3_ans(A,B) :- new3_query(A,B), A>=100, B=<100, B=<99.
new3_query(A,B) :- false_query, A=0, B=50.
new3_query(A,B) :- new3_query(C,B), C=<99, A = 1+C, C=<49.
new3_query(A,B) :- new3_query(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
false_query.
```

Fig. 3. The query-answer transformed program for program of Fig. 2

```
new3_query___1(A,B) :- false_query___1, A=0, B=50.
new3_query___1(A,B) :- new3_query___1(C,B), C=<99, A = 1+C, C=<49.
new3_query___1(A,B) :- new3_query___2(C,B), C=<99, A = 1+C, C=<49.
new3_query___2(A,B) :- new3_query___1(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
new3_query___2(A,B) :- new3_query___2(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
```

Fig. 4. Part of the split program for the program in Fig. 3

```
false_query___1 :- []
new3_query___1(A,B) :- [1*A>=0,-1*A>= -50,1*B=50]
new3_query___2(A,B) :- [1*A>=51,-1*A>= -100,1*A+ -1*B=0]
```

Fig. 5. The convex polyhedral approximate model for the split program

constrained fact for `false_ans` we conclude that `false_ans` is not a consequence of the split program. Hence, applying the various correctness results for the unfolding, query-answer and splitting transformations, `false` is not a consequence of the original program.

Discussion of the example. Application of the convex polyhedron tool to the original, or the intermediate programs, does not solve the problem; all the transformations are needed in this case, apart from redundant argument filtering, which only affects efficiency.

The model of the query-answer program is finite for this example. However, the problem is essentially the same if the constants are scaled; for instance we could replace 50 by 5000, 49 by 4999, 100 by 10000 and 101 by 10001, and the problem is essentially unchanged. We noted that some CHC verification tools applied to this example solve the problem, but essentially by enumeration of the finite set of values encountered in the search. Such a solution does not scale well. On the other hand the polyhedral abstraction shown above is not an enumeration; an essentially similar polyhedron abstraction is generated for the scaled version of the example, in the same time. The VeriMAP tool (De Angelis et al. 2013) also handles the original and scaled versions of the example in the same time.

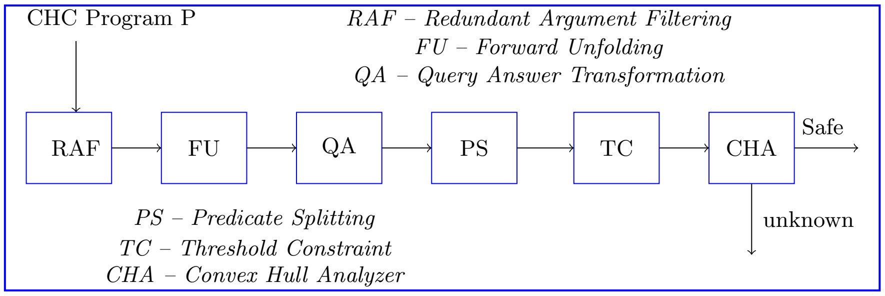

CHC Program P
RAF – Redundant Argument Filtering
FU – Forward Unfolding
QA – Query Answer Transformation
RAF
FU
QA
PS
TC
CHA
Safe
unknown
PS – Predicate Splitting
TC – Threshold Constraint
CHA – Convex Hull Analyzer

Fig. 6. *The basic tool chain for CHC verification.*

5 Combining off-the-shelf tools: Experiments

The motivation for our tool-chain, summarised in Fig. 6, comes from our example program, which is a simple yet challenging program. We applied the same tool-chain to a number of benchmarks from the literature, taken mainly from the repository of Horn clause benchmarks in SMT-LIB2[1] and other sources including (Gange et al. 2013) and some of the VeriMap benchmarks (De Angelis et al. 2013). Many of these problems are considered challenging because they cannot be solved by one or more of the state-of-the-art-verification tools. Programs taken from the SMT-LIB2 repository are first translated to CHC form. The results are summarised in Table 1.

In Table 1, columns Program and Result respectively represent the benchmark program and the results of verification using our tool combination. Problems marked with (*) could not be handled by our tool-chain since they contain numbers which does not fit in 32 bits, the limit of our Ciao Prolog implementation. Whereas problems marked with (**) are solvable by our modified tool-chain. Problems such as systemc-token-ring.01-safeil.c contain complicated loop structure with large strongly connected components in the predicate dependency graph and our convex polyhedron analysis tool is unable to derive required invariant. However overall result shows that our simple tool-chain begins to compete with advanced tools like HSF (Grebenshchikov et al. 2012), VeriMAP (De Angelis et al. 2013), TRACER (Gange et al. 2013), *etc.* We do not report timings, though all these results are obtained in a matter of seconds, since our tool-chain is not at all optimised, relying on file input-output and the individual components are often prototypes.

Table 1. *Experiments results on CHC benchmark program*

SN	Program	Result	SN	Program	Result
1	MAP-disj.c.map.pl	verified	17	MAP-forward.c.map.pl	verified
2	pldi12.pl	verified	18	tridag.smt2	verified
3	t1.pl	verified	19	qrdcmp.smt2	verified
4	t1-a.pl	verified	20	choldc.smt2	verified
5	t2.pl	verified	21	lop.smt2	verified
6	t3.pl	verified	22	pzextr.smt2	verified
7	t4.pl	verified	23	qrsolv.smt2	verified
8	t5.pl	verified	24*	crank.smt2	NOT
9	MAP-disj.c.map-scaled.pl	verified	25*	bandec.smt2	NOT
10	INVGEN-id-build	verified	26**	amebsa.smt2	verified
11	INVGEN-nested5	verified	27	sshsimpl-s3-srvr-1b-safeil.c	NOT
12	INVGEN-nested6	verified	28*	sshsimpl-s3-srvr-1a-safeil.c	NOT
13	INVGEN-nested8	verified	29**	DAGGER-barbr.map.c	verified
14	INVGEN-svd-some-loop	verified	30	INVGEN-apache-escape-absolute	verified
15	INVGEN-svd1	verified	31	systemc-token-ring.01-safeil.c	NOT
16	INVGEN-svd4	verified	32	TRACER-testabs15	verified

[1] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/

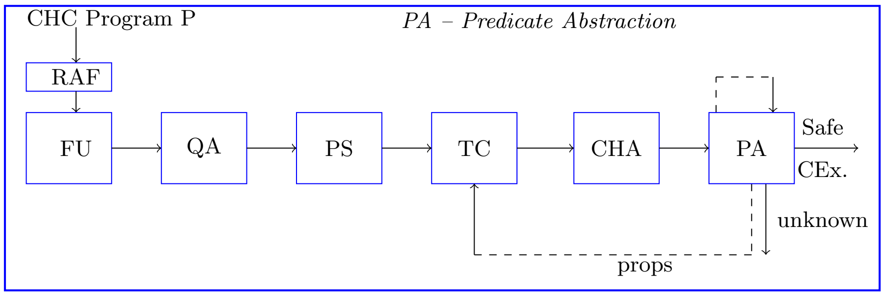


Fig. 7. *Future extension of our toolchain.*

6 Discussion and Related Work

The most similar work to ours is by De Angelis *et al.* (De Angelis et al. 2013) which is also based originally on CLP program transformation and specialisation. They construct a sequence of transformations of P, say, $P, P_1, P_2, \ldots, P_k$; essentially if P_k contains no clause with head false the problem is solved. A proof of unsafety is obtained if P_k contains a clause false $\leftarrow$. In contrast to their approach, we believe that our tool-chain-based approach gives more insight into the role of each transformation. Work by Gange et al. (Gange et al. 2013) is a top-town evaluation of CLP programs which records certain derivations and learns only from failed derivations. This helps to prune further derivations and helps to achieve termination in the presence of infinite executions. HSF(C) (Grebenshchikov et al. 2012) and Duality[2] are examples of the CEGAR approach (Counter-Example-Guided Abstraction Refinement). This approach can be viewed as property-based abstract interpretation based on a set of properties that is refined on each iteration. The refinement of the properties is the key problem in CEGAR; an abstract proof of unsafety is used to generate properties (often using interpolation) that prevents that proof from arising again. Thus more and more abstract counter-examples are successively eliminated. The relatively good performance of our tool-chain, without any refinement step at all, suggests that finding the right invariants is aided by a tool such as the convex polyhedron solver and the pre-processing steps we applied. In any case, property-based abstractions could easily be added to the tool-chain, perhaps using properties generated by the convex polyhedron solver. In Figure 7 we sketch possible extensions of our basic tool-chain, incorporating a refinement loop and property-based abstraction.

It should be noted that the query-answer transformation, predicate splitting and unfolding may all cause an increase in the program size. The convex polyhedron analysis becomes more effective as a result, but in the worst case the program size might restrict the applicability of the analysis. In future work, more sophisticated heuristics controlling these transformations, especially unfolding and splitting, are needed.

[2] http://research.microsoft.com/en-us/projects/duality/

7 Concluding remarks and future work

We have shown that a combination of off-the-shelf tools from CLP transformation and analysis, combined in a sensible way, is surprisingly effective in CHC verification. The component-based approach allowed us to experiment with the tool-chain until we found an effective combination. This experimentation is continuing and we are confident of making improvements by incorporating other standard techniques and by finding better heuristics for applying the tools. Further we would like to investigate the choice of chain suitable for each example since more complicated problems can be handled just by altering the chain. We also suspect from initial experiments that an advanced partial evaluator such as ECCE (Leuschel et al. 2006) will play a useful role. Our results give insights for further development of automatic CHC verification tools. We would like to combine our program transformation techniques with abstraction refinement techniques and experiment with the combination.

References

Benoy, F. and King, A. 1996. Inferring argument size relationships with CLP(R). In *Logic-Based Program Synthesis and Transformation (LOPSTR'96)*, J. P. Gallagher, Ed. Springer-Verlag Lecture Notes in Computer Science, vol. 1207. 204–223.

Codish, M. and Demoen, B. 1993. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In *Proceedings of the 1993 International Symposium on Logic Programming, Vancouver*, D. Miller, Ed. MIT Press.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM, 238–252.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*. 84–96.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2013. Verification of imperative programs by transforming constraint logic programs. In *CILC*, D. Cantone and M. N. Asmundo, Eds. CEUR Workshop Proceedings, vol. 1068. CEUR-WS.org, 83–98.

Debray, S. and Ramakrishnan, R. 1994. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming 18*, 149–176.

Gallagher, J. P. 1993. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, Copenhagen, 88–98.

Gallagher, J. P. and de Waal, D. 1993. Deletion of redundant unary type predicates from logic programs. In *Logic Program Synthesis and Transformation*, K. Lau and T. Clement, Eds. Workshops in Computing. Springer-Verlag, 151–167.

Gange, G., Navas, J. A., Schachte, P., Søndergaard, H., and Stuckey, P. J. 2013. Failure tabled constraint logic programming by interpolation. *TPLP 13,* 4-5, 593–607.

Grebenshchikov, S., Gupta, A., Lopes, N. P., Popeea, C., and Rybalchenko, A. 2012. Hsf(c): A software verifier based on Horn clauses - (competition contribution). In *TACAS*, C. Flanagan and B. König, Eds. LNCS, vol. 7214. Springer, 549–551.

Halbwachs, N., Proy, Y. E., and Raymound, P. 1994. Verification of linear hybrid systems by means of convex approximations. In *Proceedings of the First Symposium on Static Analysis.* LNCS, vol. 864. 223–237.

Jaffar, J. and Maher, M. 1994. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19/20*, 503–581.

Lakhdar-Chaouch, L., Jeannet, B., and Girault, A. 2011. Widening with thresholds for programs with complex control graphs. In *ATVA 2011*, T. Bultan and P.-A. Hsiung, Eds. Lecture Notes in Computer Science, vol. 6996. Springer, 492–502.

Leuschel, M. 1999. Advanced logic program specialisation. In *Partial Evaluation - Practice and Theory*, J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1706. Springer, 271–292.

Leuschel, M., Elphick, D., Varea, M., Craig, S.-J., and Fontaine, M. 2006. The Ecce and Logen partial evaluators and their web interfaces. In *PEPM 2006*, J. Hatcliff and F. Tip, Eds. ACM, 88–94.

Leuschel, M. and Sørensen, M. H. 1996. Redundant argument filtering of logic programs. In *Logic Programming Synthesis and Transformation, 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 28-30, 1996, Proceedings*, J. P. Gallagher, Ed. 83–103.

Pettorossi, A. and Proietti, M. 1999. Synthesis and transformation of logic programs using unfold/fold proofs. *J. Log. Program. 41,* 2-3, 197–230.

Serebrenik, A. and De Schreye, D. 2001. Inference of termination conditions for numerical loops in Prolog. In *LPAR 2001*, R. Nieuwenhuis and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 2250. Springer, 654–668.

Attachment D3.2.3

Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types

Accepted for publication as a paper in Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14), Special Issue.

Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types *

A. SERRANO[1]† P. LOPEZ-GARCIA[2,3] M. V. HERMENEGILDO[3,4]

[1] Dept. of Information and Computing Sciences, Utrecht University
(e-mail: A.SerranoMena@uu.nl)
[2] IMDEA Software Institute
(e-mail: pedro.lopez@imdea.org, manuel.hermenegildo@imdea.org)
[3] Spanish Council for Scientific Research (CSIC)
[4] Technical University of Madrid (UPM)
(e-mail: herme@fi.upm.es)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

We present a novel general resource analysis for logic programs based on sized types. Sized types are representations that incorporate structural (shape) information and allow expressing both lower and upper bounds on the size of a set of terms and their subterms at any position and depth. They also allow relating the sizes of terms and subterms occurring at different argument positions in logic predicates. Using these sized types, the resource analysis can infer both lower and upper bounds on the resources used by all the procedures in a program as functions on input term (and subterm) sizes, overcoming limitations of existing resource analyses and enhancing their precision. Our new resource analysis has been developed within the abstract interpretation framework, as an extension of the sized types abstract domain, and has been integrated into the Ciao preprocessor, CiaoPP. The abstract domain operations are integrated with the setting up and solving of recurrence equations for inferring both size and resource usage functions. We show that the analysis is an improvement over the previous resource analysis present in CiaoPP and compares well in power to state of the art systems.

1 Introduction

Resource usage analysis infers the aggregation of some numerical properties (named *resources*), like memory usage, time spent in computation, or bytes sent over a wire, throughout the execution of a piece of code. The expressions giving the usage of resources are usually given in terms of the sizes of some input arguments to procedures.

Our starting point is the methodology outlined by (Debray et al. 1990; Debray and Lin 1993) and (Debray et al. 1997), characterized by the setting up of recurrence equations. In that methodology, the size analysis is the first of several other analysis steps that include cardinality analysis (that infers lower and upper bounds on the number of solutions computed by a predicate), and which ultimately obtain the resource usage bounds. One

* This research was supported in part by projects EU FP7 318337 *ENTRA*, Spanish MINECO TIN2012-39391 *StrongSoft* and TIN2008-05624 *DOVES*, and Madrid TIC/1465 *PROMETIDOS-CM*.
† A. Serrano performed this work during his former affiliation to the IMDEA Software Institute.

drawback of these proposals, as well as most of their subsequent derivatives, is that they are able to cope with size information about subterms in a very limited way. This is an important limitation, which causes the analysis to infer trivial bounds for a large class of programs. For example, consider a predicate which computes the factorials of a list:

```
% listfact(+L, -FL).                % fact(N, F).
listfact([],    []).                fact(0,1).
listfact([E|R],[F|FR]) :-           fact(N,M) :- N1 is N - 1,
  fact(E, F),                                    fact(N1, M1),
  listfact(R, FR).                               M is N * M1.
```

Intuitively, the best bound for the running time of this program for a list L is $c_1 + \sum_{e \in L} (c_2 + time_{fact}(e))$, where c_1 and c_2 are constants related to unification and calling costs. But with no further information, the upper bound for the elements of L must be ∞ to be on the safe side, and then the returned overall time bound must also be ∞.

In a previous paper (Serrano et al. 2013) we focused on a proposal to improve the size analysis based on *sized types*. These sized types are similar to the ones present in (Vasconcelos and Hammond 2003) for functional programs, but our proposal includes some enhancements to deal with regular types in logic programs, developing solutions to deal with the additional features of logic programming such as non-determinism and backtracking. While in that paper we already hinted at the fact that the application of our sized types in resource analysis could result in considerable improvement, no description was provided of the actual resource analysis.

This paper is complementary and fills this gap by describing a new resource usage analysis with two novel aspects. Firstly, it can *take advantage of the new information contained in sized types*. Furthermore, this resource analysis is *fully based on abstract interpretation*. In the past, only auxiliary analyses were developed using abstract interpretation, whereas the core resource analysis was outside this framework. This allows us to integrate resource analysis within the PLAI abstract interpretation framework (Muthukumar and Hermenegildo 1992; Puebla and Hermenegildo 1996) in the CiaoPP system, which brings in features such as *multivariance*, fixpoints, and assertion-based verification and user interaction for free. We also perform an assessment of the accuracy and efficiency of the resulting global system.

In Section 2 we give a high-level view of the approach. In the following section we review the abstract interpretation approach to size analysis using sized types. Section 4 gets deeper into the resource usage analysis, our main contribution. Experimental results are shown in Section 5. Finally we review some related work and discuss future directions.

2 Overview of the Approach

We give now an overview of our approach to resource usage analysis, and present the main ideas in our proposal using the classical `append/3` predicate as a running example:

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

The process starts by performing the regular type analysis present in the CiaoPP system (Vaucheret and Bueno 2002). 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 set of "list of numbers" is represented by the regular type *listnum*, defined as follows:

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

From this regular type definition, sized type schemas are derived. The sized type schema *listnum-s* is derived from *listnum*. This schema corresponds to a list whose length is between α and β, containing numbers between γ and δ.

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

From now on, in the examples we will use *ln* and *n* instead of *listnum* and *num* for the sake of conciseness. The next phase involves relating the sized types of the different arguments to the `append/3` predicate using recurrence (in)equations. Let $size_X$ denote the sized type schema for argument `X` in a call `append(X, Y, Z)` (from the regular type inferred by a previous analysis). We have that $size_X$ denotes $ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)})$. Similarly, the sized type schema for the output argument `Z` is $ln^{(\alpha_Z,\beta_Z)}(n^{(\gamma_Z,\delta_Z)})$, denoted by $size_Z$. We are interested in expressing bounds on the length of the output list `Z` and the value of its elements as a function of size bounds for the input lists `X` and `Y` (and their elements). For this, we set up a system of inequations. For instance, the inequations that are set up to express a lower bound on the length of the output argument `Z`, denoted α_Z, as a function on the size bounds of the input arguments `X` and `Y`, and their subarguments ($\alpha_X,\ \beta_X,\ \gamma_X,\ \delta_X,\ \alpha_Y,\ \beta_Y,\ \gamma_Y,$ and δ_Y) are:

$$\alpha_Z\begin{pmatrix}\alpha_X,\beta_X,\gamma_X,\delta_X,\\ \alpha_Y,\beta_Y,\gamma_Y,\delta_Y\end{pmatrix} \geq \begin{cases}\alpha_Y & \text{if } \alpha_X = 0\\ 1+\alpha_Z\begin{pmatrix}\alpha_X-1,\beta_X-1,\gamma_X,\delta_X,\\ \alpha_Y,\beta_Y,\gamma_Y,\delta_Y\end{pmatrix} & \text{if } \alpha_X > 0\end{cases}$$

Note that in the recurrence inequation set up for the second clause of `append/3`, the expression $\alpha_X - 1$ (respectively $\beta_X - 1$) represents the size relationship that a lower (respectively upper) bound on the length of the list in the first argument of the recursive call to `append/3` is one unit less than the length of the first argument in the clause head.

As the number of size variables grows, the set of inequations becomes too large. Thus, we propose a compact representation, which allows us to grasp all the relations in one view. The first change in our proposal is to write the parameters to size functions directly as sized types. Now, the parameters to the α_Z function are the sized type schemas corresponding to the arguments `X` and `Y` of the `append/3` predicate:

$$\alpha_Z\begin{pmatrix}ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)})\\ ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\end{pmatrix} \geq \begin{cases}\alpha_Y & \text{if } \alpha_X = 0\\ 1+\alpha_Z\begin{pmatrix}ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)})\\ ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\end{pmatrix} & \text{if } \alpha_X > 0\end{cases}$$

In a second step, we group together all the inequalities of a single sized type. As we always alternate lower and upper bounds, it is always possible to distinguish the type of each inequality. We do not write equalities, so that we do not use the symbol $=$. However, we always write inequalities of both signs ($\geq$ and $\leq$) for each size function, since we compute both lower and upper size bounds. Troughout this paper we use a representation using $\lesseqgtr$ for the symbols $\geq$ and $\leq$ that are always paired. For example, the expression $ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}) \lesseqgtr ln^{(e_1,e_2)}(n^{(e_3,e_4)})$ represents the conjunction of the following size constraints:$\alpha_X \geq e_1,\ \beta_X \leq e_2,\ \gamma_X \geq e_3,\ \delta_X \leq e_4$. In the implementation, constraints for each variable are kept apart and solved separatedly.

After setting up the corresponding system of inequations for the output argument `Z` of `append/3`, and solving it, we obtain the following expression:

$$size_Z\,(size_X, size_Y) \lesseqgtr ln^{(\alpha_X+\alpha_Y,\beta_X+\beta_Y)}(n^{(\min(\gamma_X,\gamma_Y),\max(\delta_X,\delta_Y))})$$

that represents, among others, the relation $\alpha_z \geq \alpha_X + \alpha_Y$ (resp. $\beta_z \leq \beta_X + \beta_Y$), expressing that a lower (resp. upper) bound on the length of the output list `Z`, denoted α_z (resp. β_z), is the addition of the lower (resp. upper) bounds on the lengths of `X` and `Y`. It also represents the relation $\gamma_Z \geq \min(\gamma_X, \gamma_Y)$ (resp. $\delta_Z \leq \max(\delta_X, \delta_Y)$), which expresses that a lower (resp. upper) bound on the size of the elements of the list `Z`, denoted γ_z (resp. δ_z), is the minimum (resp. maximum) of the lower (resp. upper) bounds on the sizes of the elements of the input lists `X` and `Y`.

Resource analysis builds upon the sized type analysis and adds recurrence equations for each resource we want to analyze. Apart from that, when considering logic programs, we have to take into account that they can fail or have multiple solutions when executed, so we need an auxiliary *cardinality analysis* to get correct results.

Let us focus on cardinality analysis. Let s_L and s_U denote lower and upper bounds on the number of solutions for `append/3`. Following the program structure we can infer:

$$\begin{array}{rcl}
s_L\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \geq & 1 \\
s_L\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \geq & s_L\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \\
s_U\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \leq & 1 \\
s_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \leq & s_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right)
\end{array}$$

Since $s_L \leq s_U$, the solution to these inequations must be $(s_L, s_U) = (1, 1)$. Thus, we have inferred that `append/3` has at least (and at most) one solution: it behaves like a function. When setting up the equations, we use the result of the non-failure analysis to see that `append/3` cannot fail when given lists as arguments. If not, the lower bound is 0.

Now we move forward to analyzing the number of resolution steps performed by a call to `append/3` (we will only focus on upper bounds, r_U, for brevity). 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. Thus, the equation reads:

$$\begin{array}{rcl}
r_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \leq & 1 \;+\; s_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \\
& & \quad\times\; r_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \\
& = & 1 \;+\; r_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right)
\end{array}$$

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:

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

3 Sized Types Review

As shown in the `append` example, the variables that we relate in our inequations come from sized types, which are ultimately derived from the regular types previously inferred for the program. Among several representations of regular types used in the literature, we use one based on *regular term grammars*, equivalent to (Dart and Zobel 1992) but with some adaptations. A *type term* is either a *base type* η_i (taken from a finite set), a *type symbol* τ_i (taken from an infinite set), or a term of the form $f(\phi_1, \ldots, \phi_n)$, where f is a n-ary function symbol (taken from an infinite set) and $\phi_1, \ldots, \phi_n$ are *type terms*. A *type rule* has the form $\tau \to \phi$, where τ is a *type symbol* and ϕ a *type term*. A *regular term grammar* Υ is a set of *type rules*.

To devise the abstract domain we focus specifically on the AND-OR trees procedure of (Bruynooghe 1991), with the optimizations of (Muthukumar and Hermenegildo 1992). This procedure is *generic* and goal dependent: it takes as input a pair (L, λ_c) representing a predicate along with an abstraction of the call patterns (in the chosen *abstract domain*) and produces an abstraction λ_o which overapproximates the possible outputs. This procedure is the basis of the PLAI abstract analyzer present in CiaoPP (Hermenegildo et al. 2012), where we have integrated an implementation of the proposed size analysis.

The formal concept of *sized type* is an abstraction of a set of Herbrand terms which are a subset of some regular type τ and meet some lower- and upper-bound size constraints on the number of *type rule applications* needed to generate the terms. A grammar for the new sized types follows:

$$\begin{array}{rcll}
\textit{sized-type} & ::= & \eta^{\textit{bounds}} & \eta \text{ base type} \\
 & | & \tau^{\textit{bounds}}(\textit{sized-args}) & \tau \text{ recursive type symbol} \\
 & | & \tau(\textit{sized-args}) & \tau \text{ non-recursive type symbol} \\
\textit{bounds} & ::= & \textit{nob} \ | \ (n, m) & n, m \in \mathbb{N}, m \geq n \\
\textit{sized-args} & ::= & \epsilon \ | \ \textit{sized-arg}, \ \ \textit{sized-args} & \\
\textit{sized-arg} & ::= & \textit{sized-type}_{\textit{position}} & \\
\textit{position} & ::= & \epsilon \ | \ \langle f, n \rangle & f \text{ functor, } 0 \leq n \leq \text{arity of } f
\end{array}$$

However, in our abstract domain we need to refer to sets of sized types which satisfy certain constraints on their bounds. For that purpose, we introduce *sized type schemas*: a schema is just a sized type with variables in bound positions, i.e., where n and m in the pair (n, m) defining the symbol *bounds* in the grammar above are variables (called bound variables), along with a set of constraints over those variables. We call such variables *bound variables*. We will denote $\textit{sized}(\tau)$ the sized type schema corresponding to a regular type τ where all the bound variables are fresh.

The full abstract domain is an extension of sized type schemas to several predicate variables. Each abstract element is a triple $\langle t, d, r \rangle$ such that:

1. t is a set of $v \to (\textit{sized}(\tau), c)$, where v is a variable, τ its regular type and c is its classification. Subgoal variables can be classified as *output*, *relevant*, or *irrelevant*. Variables appearing in the clause body but not in the head are classified as *clausal*;
2. d (the *domain*) is a set of constraints over the relevant variables;
3. r (the *relations*) is a set of relations among bound variables.

For example, the final abstract elements corresponding to the clauses of the `listfact` example can be found below. The equations have already been normalized into their simplest form, and the variables refer to the predicate arguments in normal form. *listfact* refers implicitly to the solution of the joint equations: it is the recurrence we need to solve. In order to enhance readability, we have dropped the position element $\langle ., 1\rangle$ from ln.

$$\lambda_1' = \left\langle \begin{array}{c} \left\{L \to (ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1,\delta_1)}), rel.), FL \to (ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}), out.)\right\} \\ \{\alpha_1 = 1, \beta_1 = 1\}, \{ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \lesseqgtr ln^{(1,1)}(n^{nob})\} \end{array} \right\rangle$$

$$\lambda_2' = \left\langle \begin{array}{c} \left\{ \begin{array}{c} L \to (ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1,\delta_1)}), rel.), FL \to (ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}), out.), \\ E \to (n^{(\gamma_3,\delta_3)}, cl.), R \to (ln^{(\alpha_4,\beta_4)}(n^{(\gamma_4,\delta_4)}), cl.), \\ F \to (n^{(\gamma_5,\delta_5)}, cl.), FR \to (ln^{(\alpha_6,\beta_6)}(n^{(\gamma_6,\delta_6)}), cl.) \end{array} \right\} \\ \{\alpha_1 > 0, \beta_1 > 0\}, \\ \left\{ \begin{array}{c} ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \lesseqgtr ln^{(\alpha'+1,\beta'+1)}(n^{(\min(\gamma_1!,\gamma'),\max(\delta_1!,\delta'))}) \\ ln^{(\alpha',\beta')}(n^{(\gamma',\delta')}) \lesseqgtr listfact\left(ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)})\right) \end{array} \right\} \end{array} \right\rangle$$

4 The Resources Abstract Domain

We take advantage of the added power of sized types to develop a better resource analysis which infers upper and lower bounds on the amount of resources used by each predicate as a function of the sized type schemas of the input arguments (which encode the sizes of the terms and subterms appearing in such input arguments). For this reason, the novel abstract domain for resource analysis that we have developed is tightly integrated with the sized types abstract domain. Following (Navas et al. 2007), we account for two places where the resource usage can be abstracted:

- When entering a clause: some resources may be needed during unification of the call (subgoal) and the clause head, the preparation of entering that clause, and any work done when all the literals of the clause have been processed. This cost, dependent on the head h, is called *head cost*, $\varphi(h)$.
- Before calling a literal q: some resources may be used to prepare a call to a body literal (e.g., constructing the actual arguments). The amount of these resources is known as *literal cost* and is represented by $\omega(q)$.

We first consider the case of estimating upper bounds on resource usages. For simplicity, assume first that we deal with predicates having a behavior that is close to functional or imperative programs, i.e., that are deterministic and do not fail. Then, we can bound the resource consumption of a clause $C \equiv p(\bar{x})\ :-\ q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)$, denoted $r_{U,clause}$:

$$r_{U,clause}(C) \leq \varphi(p(\bar{x})) + \textstyle\sum_{i=1}^{n} \left(\omega(q_i(\bar{x}_i)) + r_{U,pred}(q_i(\bar{x}_i))\right)$$

As in sized type analysis, the sizes of some input arguments may be explicitly computed, or, otherwise, we express them by using a generic expression, giving rise (in the case of recursive clauses) to a recurrence equation that we need to solve in order to find closed form resource usage functions.

The resource usage of a predicate, $r_{U,pred}$, depending on its input data sizes, is obtained from the resource usage of the clauses defining it, by taking the maximum of the equation expressions that meet the constraints on the input data sizes (i.e., have the same domain).

In addition, we need to deal with two extra features of logic programming:

- We may execute a literal more than once on backtracking. To bound the number of times a literal is executed, we need to know the *number of solutions* each literal (to its left) can generate. Using the information provided by cardinality analysis, the number of times a literal is executed is at most the product of the upper bound on the number of solutions, s_U, of all the previous literals in the clause. We get:

$$\begin{aligned} & r_{U,clause}\,(p(\bar{x}) :- \ q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)) \\ & \quad \leq \varphi(p(\bar{x})) + \textstyle\sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} s_{pred}(q_j(\bar{x}_j)) \right) (\omega(q_i(\bar{x}_i)) + r_{U,pred}(q_i(\bar{x}_i))) \end{aligned}$$

- Also, in logic programming more than one clause may unify with a given subgoal. In that case it is incorrect to take the maximum of the resource usages of each clause when setting up the recurrence equations (whereas this was valid in size analysis). A correct solution is to take the sum of every set of equations with a common domain, but the bound becomes then very rough. Finer-grained possibilities can be considered by using different *aggregation* procedures per resource.

Lower bounds analysis is similar, but needs to take into account the possibility of failure, which stops clause execution and forces backtracking. Basically, no resource usage should be added beyond the point where failure may happen. For this reason, in our implementation we use the non-failure analysis already present in CiaoPP. Also, the aggregation of clauses with a common domain must be different to that used in the upper bounds case. The simplest solution is to just take the minimum of the clauses. However, this again leads to very rough bounds. We will discuss lower bound aggregation later.

Cardinality Analysis. We have already discussed why cardinality analysis (which estimates bounds on the number of solutions) is instrumental in resource analysis of logic programs. We can consider the number of solutions as another resource, but, due to its importance, we treat it separately.

An upper bound on the number of solutions of a single clause could be gathered by multiplying the number of solutions of its body literals:

$$s_{U,clause}\,(p(\bar{x}) :- \ q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)) = \textstyle\prod_{i=1}^{n} s_{U,pred}(q_i(\bar{x_i}))$$

For aggregation we need to add the equations with a common domain, to get a recurrence equation system. These equations will be solved later to get a closed form function giving an upper bound on the number of solutions.

It is important to remark that many improvements can be added to this simple cardinality analysis to make it more precise. Some of them are discussed in (Debray and Lin 1993), like maintaining separate bounds for the relation defined by the predicate and the number of solutions for a particular input, or dealing with mutually exclusive clauses by performing the max operation, instead of the addition operation when aggregating. However, our focus here is the definition of an abstract domain, and see whether a simple definition produces comparable results for the resource usage analysis.

One of the improvements we decided to include is the use of the determinacy analysis present in CiaoPP (López-García et al. 2010). If such analysis infers that a predicate is deterministic, we can safely set the upper bound for the number of solutions to 1.

In the case of lower bounds, we need to know for each clause whether it may fail or not. For that reason we use the non-failure analysis already present in CiaoPP (Bueno et al. 2004). In case of a possible failure, the lower bound on cardinality is set to 0.

The Abstract Elements. Within the PLAI abstract interpretation framework (Muthukumar and Hermenegildo 1992; Puebla and Hermenegildo 1996) an analysis is defined by the abstract elements involved in it and a set of operations. We refer the reader to the Appendix A for an overview of the overall framework. In our case, the abstract elements are derived from sized type analysis by adding some extra components. In particular:

1. The *current variable for solutions*, and *current variable for each resource*.
2. A boolean element for telling whether we have already found a failing literal.
3. An abstract element from the non-failure domain.
4. An abstract element encoding information about determinacy.

We will denote the abstract elements by $\langle (s_L, s_U), v_{resources}, failed?, d, r, nf, det\rangle$ where (s_L, s_U) are the lower and upper bound variables for the number of solutions, $v_{resources}$ is a set of pairs (r_L, r_U) giving the lower and upper bound variables for each resource, $failed?$ is a boolean element (`true` or `false`), d and r are defined as in the sized type abstract domain, and nf and det can take values `not_fails`/`fails` and `non_det`/`is_det` respectively, as explained in (López-García et al. 2010; Bueno et al. 2004). Appendix B gives some more details of the domain.

We assume that we are given the definition of a set of resources, which are fixed throughout the whole analysis process. We assume that for each resource r we have: its head cost, φ_r, which takes a clause head as parameter; its literal cost, ω_r, which takes a literal as parameter; its aggregation procedure, Γ_r, which takes the equations for each of the clauses and creates a new set of recurrence equations from them; and the default upper $\bot_{r,U}$ and lower $\bot_{r,L}$ bound on resource usage.

To better understand how the domain works, we will continue with the analysis of `listfact` that we started in the previous section. We assume that the only resource to be analyzed is the "number of resolution steps," which uses the following parameters:

$$\varphi = 1, \quad \omega = 0, \quad \Gamma_r = +, \quad (\bot_L, \bot_U) = (0, 0)$$

The $\sqsubseteq$, $\sqcup$ Operations and the $\bot$ Element. We do not have a decidable definition for $\sqsubseteq$ or $\sqcup$, because there is no general algorithm for checking the inclusion or union of sets of integers defined by recurrence relations. Instead, for the inequation components we just check whether one is a subset of another one, up to variable renaming, or perform a syntactic union of the inequations. The ordering is finished by taking the product order with the non-failure and determinacy parts. This is enough for having a correct analysis. For the bottom element, $\bot$, we first generate new variables for each of the resources and the solution. Then, we add relations between them and the default cost for each resource. For an unknown predicate, the number of solutions should be $[0, \infty)$ and it may fail. For example, the bottom element for the "number of resolution steps" resource will be:

$$\langle (s_L, s_U), \{(n_L, n_U)\}, \texttt{true}, \emptyset, \{(s_L, s_U) \leqslant (0, \infty), (n_L, n_U) \leqslant (0, 0)\}, \texttt{fails}, \texttt{non\_det}\rangle$$

where `fails` and `non_det` are the bottom elements of their respective domains.

The λ_{call} to β_{entry} Operation. In this operation we need to create the initial structures for handling the bounds on the number of solutions and resources. This implies the generation of fresh variables for each of them, and setting them to their initial values. In the case of the number of solutions, the initial value is 1 (which is the number of solutions generated by a fact). For a resource r, the initial value is exactly φ_r. We will name new fresh variables by adding an integer subscript. For example, $s_{L,1,1}$ will be the first fresh variable related to the *l*ower bound on *s*olutions on *first* clause.

The addition of constraints over sized types when the head arguments are partially instantiated is inherited from the sized types domain. Finally, for the $failed?$ component, we should start with value `false`, as no literal has been executed yet, so it cannot fail.

In the `listfact` example, the entry substitutions are:

$$\beta_{entry,1} = \left\langle \begin{array}{c} (s_{L,1,1}, s_{U,1,1}), \{(n_{L,1,1}, n_{U,1,1})\}, \texttt{false}, \{\alpha_1 = 0, \beta_1 = 0\}, \\ \{(s_{L,1,1}, s_{U,1,1}) \lesseqgtr (1,1), (n_{L,1,1}, n_{U,1,1}) \lesseqgtr (1,1)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

$$\beta_{entry,2} = \left\langle \begin{array}{c} (s_{L,2,1}, s_{U,2,1}), \{(n_{L,2,1}, n_{U,2,1})\}, \texttt{false}, \{\alpha_1 > 0, \beta_1 > 0\}, \\ \{(s_{L,2,1}, s_{U,2,1}) \lesseqgtr (1,1), (n_{L,2,1}, n_{U,2,1}) \lesseqgtr (1,1)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

The Extend Operation. In the *extend* operation we get both the current abstract substitution and the substitution from the literal call. We need to update several components of the abstract element. First of all, we need to include a call to the function giving the number of solutions and the resource usage from the called literal.

Afterwards, we need to generate new variables for the number of solutions and resources, which will hold the bounds for the clause up to that point. New relations must be added to the abstract element to give a value to those new variables:

- For the number of solutions, let $s_{U,c}$ be the new upper bound variable, $s_{U,p}$ the previous variable defining an upper bound on the number of solutions, and $s_{U,\lambda}$ an upper bound on the number of solutions for the subgoal. Then we need to include a constraint: $s_{U,c} \leq s_{U,p} \times s_{U,\lambda}$.
  In the case of lower bound analysis, there are two phases. First of all, we check whether the called literal can fail, looking at the output of the non-failure analysis. If it is possible for it to fail, we update the $failed?$ component of the abstract element to `true`. If after this checking the $failed?$ component is still `false` (meaning that neither this literal nor any of the previous ones may fail) we include a relation similar to the one for the upper bound case: $s_{L,c} \geq s_{L,p} \times s_{L,\lambda}$. Otherwise, we include the relation $s_{L,c} \geq 0$, because failing predicates produce no solutions.
- The approach for resources is similar. Let $r_{U,c}$ be the new upper bound variable, $r_{U,p}$ the previous variable defining an upper bound on that resource and $r_{U,\lambda}$ an upper bound on resources from the analysis of the literal. The relation added in this case is $r_{U,c} \leq r_{U,p} + s_{U,p} \times (\omega + r_{U,\lambda})$.
  For lower bounds, we have already updated the $failed?$ component, so we only have to work in consequence. If the component is still `false`, we add a new relation similar to the one for upper bounds. If it is `true`, it means that failure may happen at some point, so we do not have to add that resource any more. Thus the relation to be included is $r_{L,c} \geq r_{L,p}$.

In our example, consider the extension of `listfact` after performing the analysis of the `fact` literal, whose resource components of the abstract element will be:

$$\left\langle \begin{array}{c} (s_L, s_U), \{(n_L, n_U)\}, \texttt{false}, \{\alpha, \beta \geq 0\} \\ \{(s_L, s_U) \lesseqgtr (1,1), (n_L, n_U) \lesseqgtr (\alpha, \beta)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

This literal is known not to fail, so we do not change the value of *failed*? in our abstract element for the second clause. That means that it is still `false`, so we add complete calls:

$$\beta_{entry,2} = \left\langle \begin{array}{c} (s_{L,2,2}, s_{U,2,2}), \{(n_{L,2,2}, n_{U,2,2})\}, \texttt{false}, \{\dots\} \\ \left\{ \begin{array}{c} \dots, \\ (s_{L,2,2}, s_{U,2,2}) \lesseqgtr (1 \times s_{L,2,1}, 1 \times s_{U,2,1}), \\ (n_{L,2,2}, n_{U,2,2}) \lesseqgtr (\gamma_1 + n_{L,2,1}, \delta_1 + n_{U,2,1}) \end{array} \right\}, \\ \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

The β_{exit} ***to*** λ' ***Operation.*** After all the extend operations, the variables appearing in the number of solutions and resources positions will hold the correct value for their properties. As we did with sized types, we follow now a normalization step, based on (Debray and Lin 1993): replace each variable appearing in an expression with its definition in terms of other variables, in reverse topological order. Following this process, we should reach the variables in the sized types of the input parameters in the head.

Going back to `listfact`, the final substitutions are as follows. s'_L, s'_U, n'_L and n'_U refer to number of solutions and resolution steps from the recursive call to `listfact`.

$$\lambda'_1 = \left\langle \begin{array}{c} (s_{L,1,1}, s_{U,1,1}), \{(n_{L,1,1}, n_{U,1,1})\}, \texttt{false}, \{\alpha_1 = 0, \beta_1 = 0\}, \\ \{(s_{L,1,1}, s_{U,1,1}) \lesseqgtr (1,1), (n_{L,1,1}, n_{U,1,1}) \lesseqgtr (1,1)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

$$\lambda'_{entry,2} = \left\langle \begin{array}{c} (s_{L,2,3}, s_{U,2,3}), \{(n_{L,2,3}, n_{U,2,3})\}, \texttt{false}, \{\alpha_1 > 0, \beta_1 > 0\}, \\ \left\{ \begin{array}{c} s_{L,2,3} \geq 1 \times s'_L(ln^{(\alpha_1 - 1, \beta_1 - 1)}(n^{(\gamma_1, \delta_1)})), \\ s_{U,2,3} \leq 1 \times s'_U(ln^{(\alpha_1 - 1, \beta_1 - 1)}(n^{(\gamma_1, \delta_1)})), \\ n_{L,2,3} \geq \gamma_1 + n'_L(ln^{(\alpha_1 - 1, \beta_1 - 1)}(n^{(\gamma_1, \delta_1)})), \\ n_{U,2,3} \leq \delta_1 + n'_U(ln^{(\alpha_1 - 1, \beta_1 - 1)}(n^{(\gamma_1, \delta_1)})) \end{array} \right\}, \\ \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

The Widening Operator ∇ ***and Closed Forms.*** As mentioned before, in contrast to previous cost analyses, at this point we bring in the possibility of different aggregation operators. Thus, when we have the equations, we need to pass them to each of the corresponding Γ_r per each resource r to get the final equations.

This process can be further refined in the case of solution analysis, using the information from the non-failure and determinacy analyses. If the final output of the non-failure analysis is `fails`, we know that the only correct lower bound is 0. So we can just assign the relation $s_L \geq 0$ without further relations. Conversely, if the final output of the determinacy analysis is `is_det`, we can safely set the relation $s_U \leq 1$, because at most one solution will be produced in each case. Furthermore, we can refine the lower bound on the number of solutions with the minimum between the current bound and 1.

In the example analyzed above there was an implicit assumption while setting up the relations: that the recursive call in the body of `listfact` refers to the same predicate call, so we can set up a recurrence. This fact is implicitly assumed in Hindley-Milner type systems. But in logic programming it is usual for a predicate to be called with different patterns (for example, modes). Fortunately, the CiaoPP framework allows multivariance (support for different call patterns of the same predicate). For the analysis to handle it, we cannot just add calls with the bare name of the predicate, because it will conflate all the versions. The solution is to add a new component to the abstract element: a random name given to the specific instance of the predicate, and generated in the λ_{call} to β_{entry}. In the widening step, all different versions of the same predicate are conflated.

Even though the analysis works with relations, these are not as useful as functions defined without recursion or calls to other functions. First of all, developers will get a better idea of the sizes presented in such a closed form. Second, functions are amenable to comparison as outlined in (López-García et al. 2010), which is essential in verification. There are several packages able to get bounds for recurrence equations: computer algebra systems, such as Mathematica (which has been integrated to get a fully automated analysis) or Maxima; and specialized solvers such as PURRS (Bagnara et al. 2005) or PUBS (Albert et al. 2011). In our implementation we apply this overapproximation operator after each widening. For our example, the final abstract substitution is:

$$\lambda_1' \nabla \lambda_2' = \left\langle \begin{array}{c} (s_L, s_U), \{(n_L, n_U)\}, \texttt{false}, \{\alpha_1, \beta_1 \geq 0\}, \\ \{(s_L, s_U) \lessgtr (1,1), (n_L, n_U) \lessgtr (\alpha_1\gamma_1, \beta_1\delta_1)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

Table 1. *Experimental results.*

Program	***Resource A. (LB)***			***Resource A. (UB)***					***A. Times (s)***	
	New	*Prev.*		*New*	*Prev.*		*RAML*		*New*	*Prev.*
`append`	α	α	=	β	β	=	β	=	0.999	0.530
`appendAll2`	$a_1a_2a_3$	a_1	+	$b_1b_2b_3$	∞	+	$b_1b_2b_3$	=	2.408	0.668
`coupled`	μ	0	+	ν	∞	+	ν	=	1.365	0.644
`dyade`	$\alpha_1\alpha_2$	$\alpha_1\alpha_2$	=	$\beta_1\beta_2$	$\beta_1\beta_2$	=	$\beta_1\beta_2$	=	1.658	0.620
`erathos`	α	α	=	β^2	β^2	=	β^2	=	2.251	0.772
`fib`	ϕ^μ	ϕ^μ	=	ϕ^ν	ϕ^ν	=	infeasible	+	1.064	0.671
`hanoi`	1	0	+	2^ν	∞	+	infeasible	+	0.819	0.603
`isort`	α^2	α^2	=	β^2	β^2	=	β^2	=	1.675	0.617
`isortlist`	a_1^2	a_1^2	=	$b_1^2b_2$	∞	+	$b_1^2b_2$	=	2.546	0.669
`listfact`	$\alpha\gamma$	α	+	$\beta\delta$	∞	+	unknown	?	1.387	0.644
`listnum`	μ	μ	=	ν	ν	=	unknown	?	1.189	0.581
`minsort`	α^2	α	+	β^2	β^2	=	β^2	=	1.938	0.671
`nub`	a_1	a_1	=	$b_1^2b_2$	∞	+	$b_1^2b_2$	=	3.614	0.910
`partition`	α	α	=	β	β	=	β	=	1.698	0.647
`zip3`	$\min(\alpha_i)$	0	+	$\min(\beta_i)$	∞	+	β_3	+	2.484	0.570

5 Experimental results

We have constructed a prototype implementation in Ciao by defining the abstract operations for sized type and resource analysis that we have described and plugging them into CiaoPP's PLAI. Our objective is to assess the gains in precision in resource analysis.

Table 1 shows the results of the comparison between the new lower (***LB***) and upper bound (***UB***) resource analyses implemented in CiaoPP, which also use the new size analysis (columns *New*), and the previous resource analyses in CiaoPP (Debray and Lin 1993; Debray et al. 1997; Navas et al. 2007) (columns *Prev.*). We also compare (for upper bounds) with *RAML* (Hoffmann et al. 2012). Although the new resource analysis and the previous one infer concrete resource usage bound functions, for the sake of conciseness and to make the comparison with RAML meaningful, Table 1 only shows the complexity orders of such functions, e.g., if the analysis infers the resource usage bound function Φ, and $\Phi \in \Theta(\Psi)$, Table 1 shows Ψ. The parameters of such functions are (lower or upper) bounds on input data sizes. The symbols used to name such parameters have been chosen assuming that lists of numbers L_i have size $ln^{(\alpha_i,\beta_i)}(n^{(\gamma_i,\delta_i)})$, lists of lists of lists of numbers have size $llln^{(a_1,b_1)}(lln^{(a_2,b_2)}(ln^{(a_3,b_3)}(n^{(a_4,b_4)})))$, and numbers have size $n^{(\mu,\nu)}$. The calling modes are the usual ones with the last argument as output.

Table 1 includes columns with symbols summarizing whether the new CiaoPP resource analysis improves on the previous one and *RAML*'s: $+$ (resp. $-$) indicates more (resp. less) precise bounds, and $=$ the same. The new resource analysis improves on CiaoPP's previous analysis. Moreover, RAML can only infer polynomial costs, while our approach is able to infer other types of functions, as shown for the divide-and-conquer benchmarks `hanoi` and `fib`, which represent a common class of programs. For predicates with polynomial cost, we get equal or better results than RAML.

The last two columns show the times (in seconds) required by both lower and upper bound analysis together for the new resource analysis, and for the previous resource analysis in CiaoPP (Ciao/CiaoPP version 1.15-2124-ga588643, on an Intel Core i7 2.4 GHz, 8 GB 1333 MHz DDR3 memory, running MAC OS X Lion 10.7.5). These times include also the auxiliary non-determinism and failure analyses. The resulting times are encouraging, despite the currently relatively inefficient implementation of the interface with the Mathematica system which is used for solving recurrence equations.

6 Related work

Several other analyses for resources have been proposed in the literature. Some of them just focus on one particular resource (usually execution or heap consumption), but it seems clear that they could be generalized. We already mentioned RAML (Hoffmann et al. 2012) in Section 5. Their approach differs from ours in the theoretical framework being used: RAML uses a type and effect system, whereas we use abstract interpretation. Another difference is the use of polynomials in RAML, which allows a complete method of resolution but limits the type of closed forms that can be analyzed. In contrast, we use recurrence equations, which have no complete decision procedure, but encompass a much larger class of functions. Type systems are also used to guide inference in (Grobauer 2001) and (Igarashi and Kobayashi 2002). In (Nielson et al. 2002), the authors use sparsity information to infer asymptotic complexities, instead of recurrences. Similarly to CiaoPP's previous analysis, the approach of (Albert et al. 2011) applies the recurrence equation method directly (i.e., not within an abstract interpretation framework). (Rosendahl 1989) shows a complexity analysis based on abstract interpretation over a step-counting version of functional programs. (Giesl et al. 2012) uses symbolic evaluation graphs to derive termination and complexity properties.

7 Conclusions

We have presented a new formulation of resource analysis as a domain within abstract interpretation and which uses as input information the sized types that we developed in (Serrano et al. 2013). Our approach overcomes important limitations of existing resource analyses and enhances their precision. It also benefits from an easier implementation and integration within an abstract interpretation framework such as PLAI/CiaoPP, which brings in useful features such as *multivariance* for free. Finally, the results of our experimental assessment regarding accuracy and efficiency are quite encouraging.

References

Albert, E., Genaim, S., and Masud, A. N. 2011. More Precise yet Widely Applicable Cost Analysis. In *12th Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, R. Jhala and D. Schmidt, Eds. Lecture Notes in Computer Science, vol. 6538. Springer Verlag, 38–53.

Bagnara, R., Pescetti, A., Zaccagnini, A., and Zaffanella, E. 2005. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. Tech. rep. `arXiv:cs/0512056` available from `http://arxiv.org/`.

Bruynooghe, M. 1991. A practical framework for the abstract interpretation of logic programs. *J. Log. Program. 10,* 2, 91–124.

Bueno, F., López-García, P., and Hermenegildo, M. 2004. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*. Number 2998 in LNCS. Springer-Verlag, Heidelberg, Germany, 100–116.

Cousot, P. and Cousot, R. 1992. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming 13,* 2-3, 103–179.

Dart, P. and Zobel, J. 1992. A Regular Type Language for Logic Programs. In *Types in Logic Programming*. MIT Press, 157–187.

Debray, S. K. and Lin, N. W. 1993. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems 15,* 5 (November), 826–875.

Debray, S. K., Lin, N.-W., and Hermenegildo, M. 1990. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*. ACM Press, 174–188.

Debray, S. K., López-García, P., Hermenegildo, M., and Lin, N.-W. 1997. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*. MIT Press, Cambridge, MA, 291–305.

Giesl, J., Ströder, T., Schneider-Kamp, P., Emmes, F., and Fuhs, C. 2012. Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In *PPDP*. ACM, 1–12.

Grobauer, B. 2001. Cost recurrences for DML programs. In *International Conference on Functional Programming*. 253–264.

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

Hoffmann, J., Aehlig, K., and Hofmann, M. 2012. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst. 34,* 3, 14.

Igarashi, A. and Kobayashi, N. 2002. Resource usage analysis. In *Symposium on Principles of Programming Languages*. 331–342.

López-García, P., Bueno, F., and Hermenegildo, M. 2010. Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information. *New Generation Computing 28,* 2, 117–206.

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

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

Navas, J., Mera, E., López-García, P., and Hermenegildo, M. 2007. User-Definable

Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*. Lecture Notes in Computer Science, vol. 4670. Springer.

Nielson, F., Nielson, H. R., and Seidl, H. 2002. Automatic complexity analysis. In *European Symposium on Programming*. 243–261.

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

Rosendahl, M. 1989. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*. ACM Press.

Serrano, A., Lopez-Garcia, P., Bueno, F., and Hermenegildo, M. 2013. Sized Type Analysis for Logic Programs (technical communication). In *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement*, T. Swift and E. Lamma, Eds. Vol. 13. Cambridge U. Press, 1–14.

Vasconcelos, P. B. and Hammond, K. 2003. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *IFL*, P. W. Trinder, G. Michaelson, and R. Pena, Eds. Lecture Notes in Computer Science, vol. 3145. Springer, 86–101.

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

Appendix A The Abstract Interpretation Framework

Abstract interpretation (Cousot and Cousot 1992) is a framework for static analysis. Execution of the program on a concrete domain is simulated in an abstract domain, simpler than the former one. Both domains must be lattices, $\langle \mathcal{P}(\Sigma), \subseteq \rangle$ and $\langle \Delta, \sqsubseteq \rangle$. To go from one to another we use a pair of functions, called *abstraction* $\alpha : \mathcal{P}(\Sigma) \to \Delta$ and *concretization* $\gamma : \Delta \to \mathcal{P}(\Sigma)$, which should form a Galois connection:

$$\langle \mathcal{P}(\Sigma), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \Delta, \sqsubseteq \rangle \text{ if and only if } \alpha(x) \sqsubseteq y \iff x \subseteq \gamma(y)$$

Intuitively $\alpha(\sigma)$ generates the smallest element in Δ that contains all the elements in σ, and $\gamma(\delta)$ computes all the concrete elements represented by δ.

The methodology is very general, so we focus specifically on the generic AND-OR trees procedure of (Bruynooghe 1991), with the optimizations of (Muthukumar and Hermenegildo 1992). This procedure is *generic* in the sense that it separates the abstraction of program execution flow (the and-or trees), from other (mainly data-related) abstractions, which are encoded as one or more *abstract domains*. It is also goal dependent: it takes as input a pair (L, λ_c) representing a predicate along with an abstraction of the call patterns (in the chosen *abstract domain*) and produces an abstraction λ_o which overapproximates the possible outputs, as well as all different call/success pattern pairs for all called predicates in all paths in the program and the corresponding abstract information at all other program points. This procedure is the basis of the PLAI abstract analyzer found in CiaoPP (Hermenegildo et al. 2012), where we have integrated a working implementation of the proposed resource analysis. In PLAI, abstract domains are pluggable units which need to define implementations of $\sqsubseteq$, least upper bound ($\sqcup$), bottom ($\bot$), and a number of other operations related to predicate calls and successes.

For any clause $h :- q_1, \ldots, q_n.$, let λ_i and λ_{i+1} be the abstract substitutions to the left and to the right of literal q_i, and λ_{call_i} and $\lambda_{success\_i}$ their projections onto the variables of q_i respectively. λ_1 and λ_{n+1} are the *entry* and *exit* substitutions of the clause respectively, denoted also as β_{entry} and β_{exit}. We can show this graphically as follows:

$$\begin{tikzcd}
 & & \lambda_{call} & p \arrow[dll, no head] \arrow[drr, no head] & \lambda_{success} & & \\
\beta_{1,entry} & h_1 & \beta_{1,exit} & \cdots & \beta_{m,entry} & h_m & \beta_{m,exit}
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \beta_{entry} & & h \arrow[dll, no head] \arrow[drr, no head] & & \beta_{exit} & \\
\lambda_1 & p_1 & \lambda_2 & \cdots & \lambda_n & p_n & \lambda_{n+1}
\end{tikzcd}$$

To compute $\lambda_{success}$ from λ_{call} of a generic (sub)goal $p(\bar{x})$ with predicate p:

1. Generate a β_{entry_i} from λ_{call} for each of the m clauses C_i defining the predicate p. This transfers the unification of the subgoal and head variables into Δ.
2. For each clause C_i, compute β_{exit_i} from β_{entry_i}, and then project β_{exit_i} back again onto the subgoal variables, obtaining λ'_i.
3. Aggregate all the exit substitutions using the least upper bound, $\lambda_{success} = \bigsqcup_{i=1}^{m} \lambda'_i$.

Computing β_{exit} from β_{entry} is straightforward: set β_{entry} as λ_1. Then, project it onto the variables appearing in the call to the first literal q_1, obtaining λ_{call_1} for q_1, and compute $\lambda_{success\_1}$ from it using the procedure mentioned above. Now λ_1 is integrated with this success substitution, referred to as *extending* λ_1 with $\lambda_{success\_1}$. The result is set as λ_2, for which the same series of steps is performed with respect to the second literal q_2. The process continues until λ_{n+1} is obtained, which is actually β_{exit}.

In the process, more than one call substitution may appear for the same predicate. This is called *multivariance* of predicates. Furthermore, if the predicate is recursive, a fixpoint needs to be computed. To do so, the process above is iterated starting from the bottom element of the lattice, $\perp$. (Muthukumar and Hermenegildo 1992; Puebla and Hermenegildo 1996) describe performant algorithms for this purpose, which are implemented in CiaoPP.

Appendix B The Abstract Elements, Redux

Because of space constraints, in the main part of the paper the concrete and abstract domains have not been described in full. In this section we aim to give a more precise definition of both elements within the framework of abstract interpretation.

In the concrete domain, the *resource usage* of a predicate p with respect to a set of resources r_i is given by a set of triples $(\bar{t}, s, r_{p,i})$, where $\bar{t}$ is a tuple of terms. The interpretation of such set is that for a call to p with arguments bound to $\bar{t}$, the number of solutions is exactly s and the resource usage of each r_i is exactly $r_{p,i}$. Note that s and $r_{p,i}$ are actual values, not equations or recurrences. The resource usage is computed by adding the head cost at the point of entering a clause and the literal cost at the point of calling a literal in the body, using the usual SLD resolution semantics. This definition follows closely the one in (López-García et al. 2010), but extended to support several resources and cardinality.

Let $dom(e)$ be the set of tuples of terms $\bar{t}$ for which a concrete element e has information over its resource usage. We define $e \sqsubseteq_c e'$ if and only if $dom(e) \subseteq dom(e')$ and for each $\bar{t} \in dom(e)$, $(p(\bar{t}), s, r_{U,i}) = (p(\bar{t}), s', r'_{U,i})$. That is, the set of terms of the smaller element must be a subset of the larger one, and the cardinality and resource usage must coincide in the common part of their domains.

This concrete domain is abstracted in three different ways, to get a compound domain. Two of them have already been discussed in the literature: the non-failure and determinacy analyses. Those components of the abstract domain correspond to abstracting the set of elements $\bar{t}$ using a regular type abstract domain and then summarizing for those elements whether $s = 0$ or $s > 0$ (for the non-failure domain) and whether $s = 1$ or $s \neq 1$ (for the determinacy one). The $failed?$ component of the abstract elements follows closely the non-failure analysis, keeping different information during the analysis, but with the same result.

For the recurrences part, we perform several abstractions. First of all, we move from strict values for the number of solutions and resource usage to value bounds. Thus, the elements are sets of triples $(\bar{t}, (s_L, s_U), (r_{L,i}, r_{U,i}))$. The ordering is now given by:

$$\begin{array}{rcl} e \sqsubseteq_1 e' & \iff & dom(e) \subseteq dom(e') \\ & \text{and} & \text{for each } \bar{t} \in dom(e), (s_L, s_U) \subseteq (s'_L, s'_U) \text{ and } (r_{L,i}, r_{U,i}) \subseteq (r'_{L,i}, r'_{U,i}) \end{array}$$

The abstraction function in this case is very simple, we just need to send each value to an interval with it as only point:

$$\alpha_1(\{(\bar{t}, s, r_{p,i})\}_{\bar{t}}) = \{(\bar{t}, (s, s), (r_{p,i}, r_{p,i}))\}_{\bar{t}}$$

The second abstraction involves summarizing the domain of each $\alpha_1(e)$ using the sized types abstract domain. As discussed in (Serrano et al. 2013), a set of terms is described via sized types using sized type schemas along with a domain d which tells which are the values of the bound variables which are covered by the abstract element, and a set of recurrences r which defines the relations that bound variables must satisfy between them. When adding resource usage information, apart from the bounds from sized types we can refer to new variables: s_L and s_U refer to the upper and lower bound in the number of solutions, and $v_{resources}$ contains such variables for each resource in the system.

In this case, it is easier to give the concretization function to move from an abstract element e to one in the intermediate abstract domain:

$$\gamma_2(\langle d, (s_L, s_U), v_{res}, r) \rangle) = \bigcup_{\bar{t} \,\in\, \gamma_{\text{sized types}}(\langle d, r \rangle)} (\bar{t}, bound_{(s_L, s_U)}(\bar{t}, r), bound_{v_{res}}(\bar{t}, r))$$

where $bound_v(\bar{t}, r)$ returns the upper and lower *numerical* bounds for the variables v as given in the recurrences r for the tuple of values $\bar{t}$. In few words, γ_2 takes all the possible tuples of values given by the sized type we refer to, and computes the cardinality and resource usage of each of them as given by the recurrence equations.

The intermediate domain and this concretization function allows us to define an ordering $\sqsubseteq$ in the abstract elements. But, as stated in the main part of the paper, doing so would entail knowing whether some recurrences define a set that is larger or smaller than another one. This is an undecidable problem, and thus we need to resort to other checks which, while being correct, are not complete. In our case, we chose to use a syntactic check.

From α_1 we can obtain the corresponding concretization function γ_1, and from γ_2 we can do the same to obtain an α_2. By composition we obtain the abstraction $\alpha_r = \alpha_2 \cdot \alpha_1$ and concretization $\gamma_r = \gamma_1 \cdot \gamma_2$ functions that define the Galois connection between concrete resource usage triples and the abstract domain of recurrence equations.

As stated before, our complete abstract elements:

$$\langle (s_L, s_U), v_{resources}, failed?, d, r, nf, det \rangle$$

are the combination of that given by $\langle \alpha_r, \gamma_r \rangle$ with those of non-failure (which give the $failed?$ and nf components) and determinism (which gives the det component), which abstract information about s over all possible values. For an abstract element a to be smaller than b, it must be smaller in all of the three domains at the same time.

Attachment D3.2.4

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

Technical Report

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

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

[1]IMDEA Software Institute, Spain
[2]University of Bristol, UK
[3]Spanish Council for Scientific Research (CSIC), Spain
[4]Universidad Politécnica de Madrid (UPM), Spain
[5]Roskilde University, Denmark

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 as functions on the input data sizes of programs. We build on our previously developed analysis which is based on transforming the *assembly code* corresponding to a given program into a Horn Clause representation that is supplied, together with an energy model of individual assembly instructions, to an existing resource analysis tool. In this work we aim at increasing the power of the energy analysis by transforming and analyzing instead the intermediate compiler representation (in particular, the LLVM IR) in order to take advantage of the program information present at this level, often needed by the analysis. To this end, we use the same energy model defined at the assembly level, together with a mapping mechanism for propagating this energy model up to the LLVM IR level. 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 and more powerful than our previous analysis at the assembly level.

1. INTRODUCTION

Energy consumption and the environmental impact of computing technologies have become a major worldwide concern. It is a major 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 applications, 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 applications usually operate at the source code level. 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, which make it difficult or impossible to find precise enough mappings between segments of the source code and segments of the code actually executed. Thus, some mechanism is required to propagate the information about the energy consumed at the hardware level up to the source code level. Moreover, the inference of energy consumption information for lower levels such as the Instruction Set Architecture (ISA) or intermediate compiler representations (such as LLVM IR [17]) is also fundamental for two reasons: 1) It is an intermediate step that allows easy 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. It is also desirable that the static analysis handle (or at least, be easily adaptable to) different programming languages in the same framework.

In this paper we propose a static analysis system that infers energy consumption information at the ISA, LLVM IR, and source code levels, and provides this information in the form of functions on input data sizes. The results of analysis are 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 levels for which energy information will be inferred. As we will show, such selection has an impact on the analysis accuracy and on the class of programs that can be analyzed.

To show the feasibility of our approach, in this paper we focus on the energy analysis of programs written in XC [28] running on the XMOS XS1-L architecture. XC is a high-level C-based programming language that includes extensions for concurrency, communication, input/output opera-

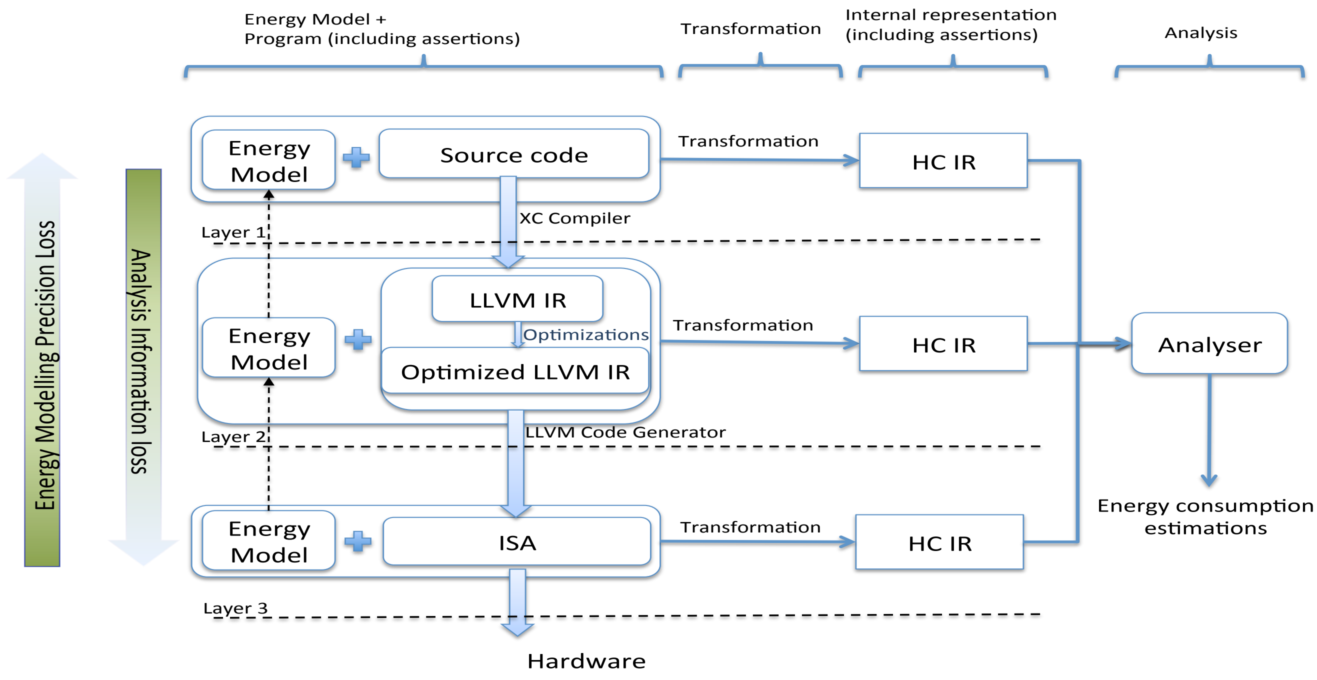


Figure 1: Overview of the analysis/modeling level choices.

tions, and real-time behavior. However, our ideas are general enough to be applied to the analysis of other programming languages (and associated lower level program representations) and architectures as well.

The requirement of supporting different programming languages and different program representations at different levels of compilation (e.g., LLVM IR and ISA) in the same framework leads us to differentiate between the *input language* (which can be XC source, LLVM IR, or ISA) and the *intermediate semantic program representation* that the resource analysis takes 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. The approach we propose is based on performing a transformation from each *input language* into the HC IR and passing it to the CiaoPP [12] resource analyzer. This analyzer deals with the HC IR always in the same way, independently of where it comes from, inferring energy consumption information 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 [19]. For instance, it supports naturally Static Single Assignment (SSA) and recursive forms, as will be explained later. Moreover, many tools are available that use Horn Clauses as intermediate representation for analysis, including the above-mentioned CiaoPP analyzer that we have extended and used in this paper. In fact, currently, there is a trend to use Horn Clause programs as intermediate representation in analysis and verification tools [2].

In our approach the HC IR also includes an *energy model*, represented by means of trust assertions (see [13] and its references for a description of the Ciao assertion language). The role of the energy model is to express the effect, in terms of energy consumption, of the execution of a software segment (e.g., an assembly instruction) on the hardware. Such information is required by the analyzer, which propagates it during the static analysis of a program (expressed in the HC IR) through code segments, conditionals, loops, recursions, etc., in order to infer information (the analysis results) for higher-level entities such as functions or procedures in the program.

In principle, analysis can be performed (and the energy models defined) at any language level, e.g., XC source, LLVM IR, ISA, etc. Performing the analysis of a program at a given level means that it is the representation of the program at that level that is transformed into the HC IR, and it is the execution of the program instructions at that level that the analyzer "mimics" according to the semantics of interest, inferring information (in our case, energy) also for that level. The energy model provides basic information on the energy cost of instructions (in principle, also at the same level) that the analysis will just trust. It is also possible to reflect the analysis results upwards to a higher level after analysis. For example, inferring energy consumption for XC source programs can also be achieved by analyzing the corresponding ISA or LLVM IR programs (as generated using the XC compiler tool, XCC) and mapping the results back to the source code. Furthermore, it is also possible to perform analysis at a given level with an energy model for a lower level. In this case the energy model must somehow be reflected up to the analysis level.

Our hypothesis is that going down through the hierarchy of levels 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

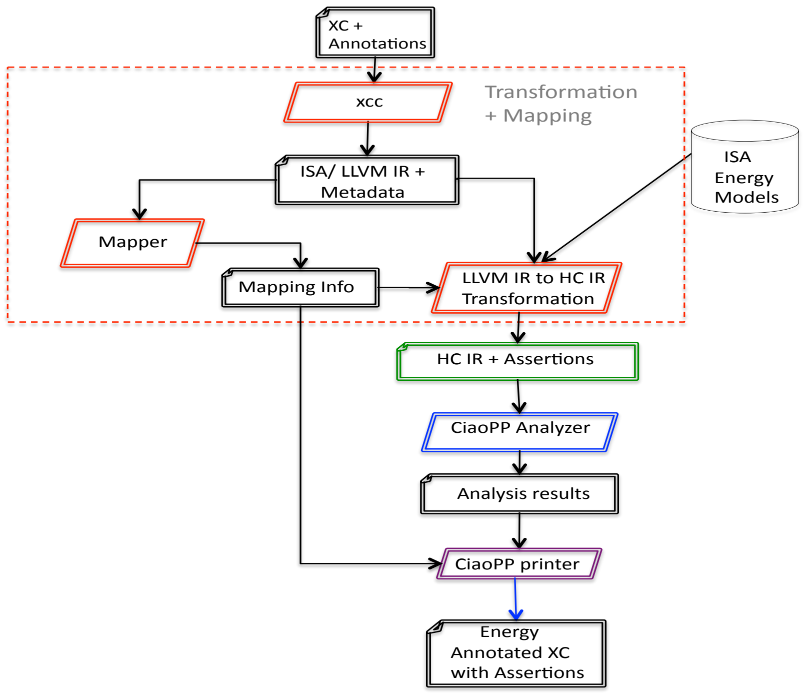


Figure 2: An overview of the analysis at the LLVM IR level using ISA models.

more program structure and data type/shape information is lost due to lower-level representations, and we expect a corresponding loss of accuracy (and restricted scalability) in the analysis. This analysis/modeling level trade-off 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.

In [18], we explored the choice of analysing the generated ISA program using models defined at the ISA level that express the energy consumed by the execution of individual ISA instructions [15] (choice 1). We showed that precise energy models can be constructed at the ISA level (because of its proximity to the hardware), which when used in the static analysis resulted in reasonably accurate energy consumption estimations for some programs. However, we also experienced that due to the loss of information related to program structure and types of arguments at the ISA level, the power of the analysis was limited, identifying some classes of programs that could not be analyzed.

In this paper we explore an alternative approach: the analysis of the generated LLVM IR (which preserves much more of the program information needed by the analyzer) using the same ISA-level energy model [15], together with techniques that map segments of ISA instructions to LLVM IR blocks [10] (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 (Constraint Horn Clauses) and used the CiaoPP resource analyzer. Finally, we have performed an experimental comparison of the two choices above. Our results support the idea that choice 2 can be a good compromise within the level hierarchy, since it broadens the class of programs that can be analyzed without significant loss of accuracy.

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 a source program that can (optionally) contain assertions (used to provide useful hints and information to the analyzer), from which a *Transformation and Mapping* 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. The procedural interpretation of these HC IR programs, coupled with the resource-related information contained in the 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. 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 is in charge of showing 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).

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

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

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, independently of its origin. The transformation ensures that the program information relevant to resource usage is preserved, so that the energy consumption functions of the HC IR programs inferred by the resource analysis are applicable to the original LLVM IR programs.

The transformation also generates on the fly energy models for the LLVM IR level for different programs based on the ISA/LLVM IR mapping information, as explained in Section 4. Furthermore it translates assertions in the XC source code into the HC IR.

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 our previous ISA-level transformation [18]. However, the LLVM IR includes an additional type transformation as well as better memory modeling. The main aspects of this process are the following:

1. Infer input/output parameters to each block, removing the need for ϕ (phi) nodes.
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 block as a literal (S_i).
4. Resolve branching to multiple clauses with the same block name, where each clause denotes one of the blocks the branch may jump to.

These steps will be described further in the following 3 sections. The transformation is implemented as an LLVM pass within the LLVM Pass Framework (LPF), which allows easily plugging such user-defined transformation passes over the LLVM IR. It allows us to reuse some of the existing analysis and transformation passes over the LLVM IR to aid our transformation and also hopefully makes the implementation forward compatible as far as LLVM IR is concerned.

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 instructions (LLVM IR instructions in this case). 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.

The entry block in the LLVM IR is always *alloca*, where the program variables are allocated. The input/output arguments to the corresponding HC IR entry block are same as the input/output arguments to the function under transformation. We define the functions $param_{in}$ and $param_{out}$ which infer input and output parameters to a block. These are recomputed 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 b with a jump instruction, while $gen(k)$ and $kill(k)$ are the read and written variables in a block 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 in the block, respectively.

Note that the LLVM IR is in SSA form at the function level. This 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. Once the input/output parameters are inferred for each block, a post-process removes all the ϕ nodes and modifies the block input arguments such that it receives x directly as an input and an appropriate x_i is passed by the call site.

Consider the example in Figure 5, 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 6, line 4.

3.2 Translating LLVM IR Types into HC IR Types

Since LLVM IR is a typed representation, these types are transformed as well into their counterparts in HC IR. The LLVM type system defines primitive and derived types. The primitive types are the fundamental building blocks of

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

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

Figure 3: Example of types in a XC program to be translated into HC IR.

```
:- use_package(regtypes).

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

:- regtype struct/1.
struct := mystruct(~num,~array2).

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

Figure 4: Regular type definitions in HC IR.

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 XC does not support 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. Translating LLVM IR primitive types into regular types is straightforward. The *integer* and *character* types are represented as *num*, whereas the *label*, *void*, and *metadata* types are represented as *atm* (atoms). LLVM supports several *integer* types of differing bit widths. While it is possible to support such types in HC IR, we currently abstract for simplicity all integers to *num.*

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. For example, the benchmark *sum_facts* (Table 1) computes the sum of the factorials of the numbers in a list. The energy function depends not only on the size of the list (that controls the iteration of the outer loop traversing the list) but also on the sizes of the elements of the list (that control the iteration of the inner loop computing the factorial). The array, vector, and structure types are represented as follows:

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

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 declaration of the function *print* in Figure 3 is:

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

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 is read as 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., 5 elements of $i32$ integer type.[1]

This type is represented in the HC IR using the set of regular types illustrated in Figure 4. 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. *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

Each function's body in LLVM IR is defined by a set of basic blocks which may jump to other blocks. Each function has an entry and an exit block (exit block only for the terminating functions). A basic block is simply a container of LLVM IR instructions that execute sequentially. In order to represent an LLVM IR function by an HC IR function, 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.

Consider the LLVM IR block *looptest* in Figure 5, it has 3 LLVM IR instructions. The first instruction is a phi (ϕ) assignment. The ϕ instruction assigns either *%N* or *%I1* to *%I* based on the predecessor LLVM IR block used to reach the current block, *%alloca* and *%loopbody* respectively. The second instruction at line 5 is a comparison instruction that assigns the result of the test "$\%I \neq 0$" to a boolean variable *%Zcmp.* The last instruction of the block is a conditional jump to one of the two blocks defined at lines 8 and 14 based on the boolean value of the variable *%Zcmp.*

In Figure 6, the block *looptest* is represented as an HC IR block. Each of the LLVM IR instructions is transformed into an equivalent HC IR literal. The ϕ instruction is removed from the HC IR block. The semantics of the ϕ instruction

[1] $[0 \times i32]$ is read as an arbitrary length array of $i32$ integer type elements.

```
 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
 8 loopbody:
 9   %Elm = getelementptr [0xi32]* %Arr,i32
         0,i32 %I
10   ... //process array element 'Elm'
11   %I1 = sub i32 %I, 1
12   br label %looptest
13
14 loopend:
15   ret void
```

Figure 5: LLVM IR blocks example: Array traversal.

```
 1 alloca(N, Arr):-
 2   looptest(N, Arr).
 3
 4 looptest(I, Arr):-
 5   icmp_ne(I, 0, Zcmp),
 6   loopbody_loopend(Zcmp, ...).
 7
 8 icmp_ne(X, Y, 1):- X \= Y.
 9 icmp_ne(X, Y, 0):- X = Y.
10
11 loopbody_loopend(Zcmp, I, Arr):-
12   Zcmp=1,
13   nth(I, Arr, Elm),
14   ... //process list element 'Elm'
15   I1 is I - 1,
16   looptest(I1, Arr).
17
18 loopbody_loopend(Zcmp, I, Arr):-
19   Zcmp=0.
```

Figure 6: HC IR representation of LLVM IR blocks in Figure 5

is preserved on the call sites of the *looptest* (lines 2 and 16). The call sites *alloca* and *loopbody* pass the corresponding value as an argument to *looptest*, which is received in I. The conditional instruction on line 5 in Figure 5 is translated into a literal *icmp_ne/3* in Figure 6. The *icmp_ne/3* predicate is defined during the transformation to represent the semantics of the conditional instruction on lines 8 and 9.

The branching instruction at line 6 in Figure 5, 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. The name of the predicate is the concatenation of the names of the two LLVM IR blocks. The two clauses of the predicate on lines 11 and 18 in Figure 6 represent the LLVM IR blocks *loopbody* and *loopend* respectively. The test on the conditional variable is placed in both clauses to preserve the energy consumption semantics of the conditional jump.

Consider also the memory instruction *getelementptr* on line 9 in Figure 5, which accesses an element of an array *%Arr* indexed by *%I* and assigns it to a variable *%Elm.* Such a memory instruction is represented by a call to the library predicate *nth/3*, when it extracts an element from a list. However if the *getelementptr* instruction extracts an element from a user-defined data type (e.g., a structure) rather than an array, an abstract predicate is generated and used. In any case, the effect of the execution of the instructions represented by those predicates on the energy consumption, as well as the relationships between the sizes of their input and output arguments, are described using trust assertions. These trust assertions are instrumental in the resource usage analysis. For example, the assertion:

```
:- trust pred nth(I, L, Elem)
  :(num(I), list(L,num), var(Elem))
  => ( num(S), list(L,num), num(Elem),
       rsize(S,num(SL,SU)),
       rsize(L, list(C, D, num(E, F))),
       rsize(Elem, num(E, F)) )
       + (resource(avg, energy, 1215439) ).
```

indicates that if the `nth(I, L, Elem)` predicate (representing the *elementptr* LLVM IR instruction) is called with `I` and `L` bound to an integer and a list respectively, and `Elem` an unbound variable (precondition field "`:`"), after the successful completion of the call (postcondition field "`=>`"), `Elem` is an integer number and the upper and lower bounds on its size are equal to the upper and lower 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 upper and lower bounds on the length of the list L are C and D respectively. Similarly, the lower and upper bounds on the elements of the list are E and F respectively, which are specified to be the same for *Elem.*

4. OBTAINING THE ENERGY CONSUMPTION OF LLVM IR BLOCKS

As mentioned before, 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 this end we take as starting point the energy consumption information available from the XS1-L ISA Energy Model (Section 4.1) and use a mapper tool to reflect it up to the LLVM IR level (Section 4.2).

4.1 XS1-L ISA Energy Model

An ISA energy model characterises the energy consumption of a processor based on observation of the sequence of instructions executed by the processor and a representation of how the processor behaves given this sequence. This can be achieved at various levels, from gate or transistor level simulation of the device [5], up to empirical measurement of real hardware, with trade-offs in speed of simulation/measurement, accuracy and access to required processor information. In the case of the XS1-L energy model, the processor is characterised using empirical measurements obtained from a series of tests.

The simplest form of ISA modelling from empirical measurement, demonstrated by [26], attempts to capture three characteristics, the *base* energy cost of executing an instruction, the cost of the *overhead* of switching from one instruction to the next, and any *external effects* such as cache activity. In Equation 1, this is represented by the base cost, B_i, for an ISA instruction, i, counted N_i times, the number of times the instruction is executed. The overhead, $O_{i,j}$, of switching between instructions i and j, and the sum of all k external effects, E_k, are added to the base cost, resulting in an energy estimate for the program, E_{prog} that is represented as a sequence of instructions.

$$E_{\text{prog}} = \sum_{i \in \text{ISA}} (B_i N_i) + \sum_{i,j \in \text{ISA}} (O_{i,j} N_{i,j}) + \sum_{k \in \text{ext}} E_k \quad (1)$$

More complex representations of a processor's behaviour have been demonstrated in [25], where this style of equation is extended with additional terms and a larger number of parameters. For the XS1-L, a similar extension approach is used to consider two key characteristics of the architecture, namely its hardware multi-threading and its event-driven nature. The processor can have idle periods in which no instructions are executed. The number of active threads governs how full the pipeline will be. This, in turn, has an effect on energy consumption beyond simply the sequence of instructions in a program.

Research into these behaviours and their impact on the XS1-L's energy characteristics [15] has produced the model Equation 2. This equation considers a base power, P_{base} that is separated from instruction execution, and thus can be counted during a number of idle cycles, N_{idle}, multiplied by the cycle time, T_{clk}, to give the energy cost of idle time. During instruction execution, sequences of instructions are considered at the various levels of concurrency (or the number of *active threads*), t. Due to multi-threaded interleaved instructions, the overhead, O, is simplified to a constant, as the next instruction in the processor's pipeline may be from a different thread. The instruction power, P_i, is in addition to the base power, P_{base}, and is multiplied by a concurrency scaling term, M_t, which is governed by the number of active threads, t. Finally, as with idle time, this power figure is multiplied by the number of times the instruction is executed at the concurrency level t, $N_{\text{i,t}}$ and the cycle time, T_{clk}. Summing this over all executed instructions and utilised concurrency levels produces an energy estimate, E_{prog}.

$$\begin{aligned} E_{\text{prog}} &= P_{\text{base}} N_{\text{idle}} T_{\text{clk}} \\ &\quad + \sum_{t=1}^{N_t} \sum_{i \in \text{ISA}} ((M_t P_i O + P_{\text{base}}) \times (N_{i,t} T_{\text{clk}})) \end{aligned} \quad (2)$$

In cases where a single-threaded program is executed on the XS1-L, as is the case in this paper, the architecture's multi-threaded pipeline must still be considered, but the equation can be simplified. If there are no idle periods, for example due to not waiting for inter-thread communication, the equation can be simplified further. Equation 3 combines these simplifications, removing the idle component of the previous equation and considering only one concurrency level, $t = 1$.

$$E_{\text{prog}} = \sum_{i \in \text{ISA}} ((M_1 P_i O + P_{\text{base}}) \times (N_i T_{\text{clk}})) \quad (3)$$

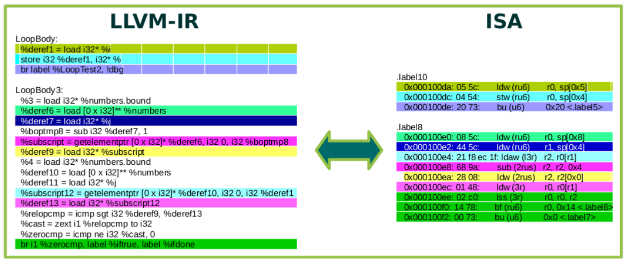
In the experiments performed in this paper, with single-threaded programs, an energy value E_i is assigned to each instruction i in the ISA. Such value is used to obtain energy values for LLVM IR blocks, as explained in Section 4.2, and is related to Equation 3 in the following way:

$$E_i = (M_1 P_i O + P_{\text{base}}) \times T_{\text{clk}} \quad (4)$$

The detailed study of the XS1-L architecture, containing a description of the test and measurement process along with the construction of the model, is performed in [15].

4.2 Propagating the ISA-level energy model up to the LLVM IR Level

To enable the analysis at the LLVM IR level, taking an existing energy model at the ISA level (described in Section 4.2) as starting point, a mechanism is needed to propagate the ISA-level energy information up to the LLVM IR level. 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. An example of this mapping is shown in Figure 7.



```
LoopBody:
 %deref1 = load i32* %i
 store i32 %deref1, i32* %
 br label %LoopTest2, !dbg

LoopBody3:
 %3 = load i32* %numbers.bound
 %deref6 = load [0 x i32]** %numbers
 %deref7 = load i32* %j
 %boptmp8 = sub i32 %deref7, 1
 %subscript = getelementptr [0 x i32]* %deref6, i32 0, i32 %boptmp8
 %deref9 = load i32* %subscript
 %4 = load i32* %numbers.bound
 %deref10 = load [0 x i32]** %numbers
 %deref11 = load i32* %j
 %subscript12 = getelementptr [0 x i32]* %deref10, i32 0, i32 %deref1
 %deref13 = load i32* %subscript12
 %relopcmp = icmp sgt i32 %deref9, %deref13
 %cast = zext i1 %relopcmp to i32
 %zerocmp = icmp ne i32 %cast, 0
 br i1 %zerocmp, label %iftrue, label %ifdone
```

```
.label10
 0x000100da: 05 5c:       ldw (ru6)    r0, sp[0x5]
 0x000100dc: 04 54:       stw (ru6)    r0, sp[0x4]
 0x000100de: 20 73:       bu (u6)      0x20 <.label5>

.label8
 0x000100e0: 08 5c:       ldw (ru6)    r0, sp[0x8]
 0x000100e2: 44 5c:       ldw (ru6)    r1, sp[0x4]
 0x000100e4: 21 f8 ec 1f: ldaw (l3r)   r2, r0[r1]
 0x000100e8: 68 9a:       sub (2rus)   r2, r2, 0x4
 0x000100ea: 28 08:       ldw (2rus)   r2, r2[0x0]
 0x000100ec: 01 48:       ldw (3r)     r0, r0[r1]
 0x000100ee: 02 c0:       lss (3r)     r0, r0, r2
 0x000100f0: 14 78:       bf (ru6)     r0, 0x14 <.label6>
 0x000100f2: 00 73:       bu (u6)      0x0 <.label7>
```

Figure 7: An example of mapping ISA to LLVM IR code, obtained by the mapper tool.

Our mapping technique leverages the existing debugging mechanism in the XMOS compiler tool chain. This mechanism supports the debugging process used by a programmer to investigate and fix a crashing application or any other kind of faulty program behavior. The main idea is to find which ISA instructions correspond to each LLVM IR instruction. During the lowering phase of the compilation, the LLVM IR code is transformed to the specified architecture ISA code by the back-end of the compiler. The debug information is also stored alongside with the ISA code using the DWARF standard format [2]. Tracking this information can result in an n:m relationship between the two levels, because one source code command can be translated to many LLVM IR instructions and therefore to many ISA instructions. This n:m relation is not adequate to provide the static analysis with precise energy values and there is a clear need for a more fine-grained mapping.

To address this issue, we created an LLVM pass that traverses the LLVM IR and replaces the Source Location Information with LLVM IR location information, right after all the optimization passes and just before emitting the ISA code. In this way, we achieved a 1:m relation in the mapping of LLVM IR instructions to ISA instructions. Also, by performing the process after the LLVM optimization passes, the optimized LLVM IR is closer to the ISA code than the un-optimized one, which still has to go through a series of transformations. There are also some optimizations happening during the lowering phase, such as peephole optimizations and some late target-specific optimizations, that can affect the mapping, but not to the same degree as the LLVM optimizations. After the mapping is done for a specific program, the associated energy values for the ISA instructions (value E_i in Equation 4 for ISA instruction i) corresponding to an LLVM IR instruction are aggregated and then associated with the LLVM IR instruction. Afterwards, an energy value is assigned to each LLVM IR block by aggregating the energy values of its LLVM IR instructions by the mapper.

Although we use the XMOS tool-chain for the mapper tool, the approach and techniques employed are generic and transferable, due to the use of the common LLVM optimizer and code generator, and the use of the DWARF standardized debugging data format [1], used by many compilers and debuggers to support source-level debugging. A full description of the mapping techniques is given in [10].

5. RESOURCE ANALYSIS WITH CIAOPP

Finally, in order to perform the global energy consumption analysis, our approach leverages the CiaoPP tool [12], the preprocessor of the Ciao programming environment [13]. CiaoPP includes a global static analyzer which is parametric with respect to resources and type of approximation (lower and upper bounds) [21, 23]. 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 [23].

Consider the example in Figure 6. 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 (which is obtained as described in Section 4.2). Then, the inferred equations for the HC IR blocks in Figure 6 are:

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

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

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

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):

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

Note that using average values in the model implies that the energy function for the whole program inferred by the upper-bound resource analysis is an approximation of the actual upper bound (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. However, such bounds would be very conservative, causing a loss in accuracy that would make the analysis impractical.

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.

6.1 Power Monitoring Setup

Figure 8 demonstrates the power monitoring setup used to run our benchmarks and measure their energy consumption. The most widely used methodology for measuring power consumption, is the use of a shunt resistor. As shown in the figure, the power measurement board places a shunt resistor between the power supply and the monitored board, which runs the benchmarks. Since the resistor value is known, we can easily and accurately calculate the current using Ohm's law, by measuring the voltage drop across the shunt resistor. The power samples are then sent to the control board which processes them and then sends the power data to the host pc. The control board also controls the running of individual benchmarks on the target board by sending control signals for starting and stopping the individual runs and power sampling.

6.2 Results

Table 1 lists the benchmarks we analyzed at the LLVM IR level along with the average error over the input for each benchmark compared to the actual energy consumption measured on the hardware (in the **llvm** column). Also, for comparison purposes the column **isa** provides the same information for the analysis at the ISA level, as obtained in previous work [18]. The last row of the table shows the average error over the number of benchmarks analyzed at each level.

In Table 2, more detailed data are presented. Column **SA energy function** shows the energy consumption functions,

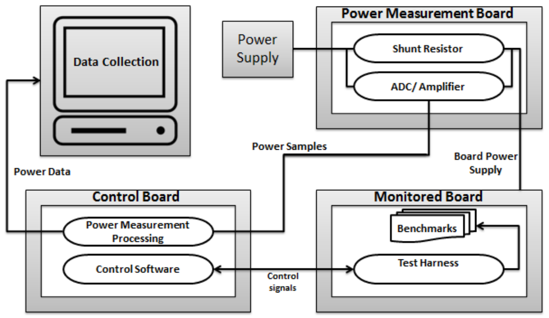


Figure 8: The power monitoring setup.

Program	Error vs. HW		ISA/LLVM
	llvm	isa	
`fact`	4.5%	2.86%	0.94
`fibonacci`	11.94%	5.41%	0.92
`sqr`	9.31%	1.49%	0.91
`power_of_two`	11.15%	4.26%	0.93
`reverse`	2.18%	N/A	N/A
`concat`	8.71%	N/A	N/A
`mat_mult`	1.47%	N/A	N/A
`sum_facts`	2.42%	N/A	N/A
`Average`	**6.46%**	**3.50%**	**0.92**

Table 1: LLVM IR- vs. ISA-level analysis accuracy.

which depend on input data sizes, inferred for each program both at the ISA and LLVM IR levels (denoted with subscripts *isa* and *llvm* respectively). Column **HW** shows the actual energy consumption measured on the hardware corresponding to the execution of the programs with input data of different sizes (shown in column **Input Data Size**). The **Estimated** and **Error vs. HW** columns present the estimated energy consumption inferred by the static analysis at the LLVM IR and ISA levels and their error with respect to the actual energy consumption measured on the hardware, respectively.

The experimental results show that:

- On average, the analysis performed at either level is reasonably accurate and the relative error between the two analysis at different levels is small.
- ISA-level estimations are slightly more accurate than the ones at the LLVM IR level (3.5% vs. 6.46 % 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, which causes a slight loss of accuracy.
- The LLVM IR level analysis is more powerful than the one at the ISA level. This is because typing information is preserved at the LLVM IR level, which allows analyzing programs using data structures (such as arrays) that could not be analyzed at the ISA level, without a significantly more complex representation of memory in the Horn clause representation.

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 [18] based on transforming the corresponding ISA code into a Horn Clause representation that is supplied, together with an ISA-level energy model, to an existing resource analysis tool. 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 at 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. Finally, we study a larger set of benchmarks, obtaining promising results for a good number of them for which [18] was not able to produce useful energy functions. A similar approach was proposed for upper-bound energy analysis of Java bytecode programs in [20], where the Jimple (a typed three-address code) representation of Java bytecode was transformed into Horn Clauses, and a simple energy model at the Java bytecode level [16] was used. However, this work did not compare the results with actual, measured energy consumptions. In all the approaches mentioned above, instantiations for energy consumption of general resource analyzers are used, namely [21] in [20] and [18], and [23] in this report. Such resource analyzers are based on setting up and solving recurrence equations, an approach proposed by Wegbreit [29] that has been developed significantly in subsequent work [22, 7, 8, 27, 21, 3, 23]. 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 [11] and Java (by means of a transformation into Java bytecode) [4]. In [4], cost relations are inferred directly for these bytecode programs, whereas in [20] the bytecode is first transformed into Horn Clauses. The worst-case analysis presented in [14], 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. A number of static analyses are aimed at worst case execution time (WCET), usually for imperative languages in different application domains (see e.g., [9] and its references). However, in contrast to the work presented here, WCET methods do not infer cost functions on input data sizes but rather absolute maximum execution times, and they generally require the manual annotation of loops to express an upper-bound on the number of iterations. A timing analysis based on game-theoretic learning is presented in [24]. 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.

Although substantial effort has been devoted to ISA energy modeling, there is not a lot of work done at higher levels of the software stack. This is mostly because precision decreases as you move further away from the hardware. One of the most recently pertinent works for LLVM IR energy modeling is [6]. The authors, employed a statistical analysis and characterization of LLVM IR code together with instrumentation and execution on the host machine, to estimate performance and energy requirements in embedded software.

SA energy function (nJ)	*Input Data Size*	HW (nJ)	Estimated (nJ)		Error vs. HW		isa/llvm
			llvm	isa	llvm	isa	
$Fact_{isa}(N)$=	$N = 2$	78	75	70	-3.40%	-9.63%	0.94
$26.0\ N + 19.4$	$N = 4$	128	129	121	1.34%	-4.79%	0.94
	$N = 8$	227	237	223	4.59%	-1.48%	0.94
$Fact_{llvm}(N)$=	$N = 16$	426	453	428	6.52%	0.49%	0.94
$28.4\ N + 22.4$	$N = 32$	824	886	836	7.58%	1.57%	0.94
	$N = 64$	1690	1751	1654	3.59%	-2.15%	0.94
$Fibonacci_{isa}(N)$=	$N = 2$	75	74	69	-1.16%	-7.88%	0.93
$31.62 + 36.6 \times 1.62^N + 11.4 \times (-0.62)^N$	$N = 4$	219	241	221	10%	0.93%	0.92
	$N = 8$	1615	1951	1693	14.75%	4.81%	0.91
$Fibonacci_{llvm}(N)$=	$N = 15$	47×10^3	57×10^3	50×10^3	16.47%	6.33%	0.91
$37.53 + 42.3 \times 1.62^N + 11.68 \times (-0.62)^N$	$N = 26$	9.30×10^6	11.5×10^6	9.9×10^6	17.31%	7.09%	0.91
$Sqr_{isa}(N)$=	$N = 9$	1242	1302	1209	4.86%	-2.66%	0.93
$9.02\ N^2 + 51.29\ N + 16.5$	$N = 27$	8135	8734	7979	7.36%	-1.92%	0.91
	$N = 73$	52×10^3	57×10^3	52×10^3	8.51%	-1.58%	0.91
$Sqr_{llvm}(N)$=	$N = 144$	19.7×10^4	21×10^4	19.4×10^4	8.89%	-1.47%	0.90
$10.52\ N^2 + 55.79\ N + 16.5$	$N = 234$	51×10^4	55×10^4	50×10^4	9.61%	-0.91%	0.90
	$N = 360$	11.89×10^5	13×10^5	11.87×10^5	10.49%	-0.17%	0.90
	$N = 400$	14.65×10^5	16×10^5	14.64×10^5	10.58%	-0.10%	0.90
	$N = 3$	326	344	322	5.68%	- 1.10%	0.94
$Poweroftwo_{isa}(N)$=	$N = 6$	2729	2965	2770	6.59%	1.49%	0.93
$10.9 \times 2^{N+2} - 27.29$	$N = 9$	21.9×10^3	23.9×10^3	22.3×10^3	6.61%	1.81%	0.93
$Poweroftwo_{llvm}(N) =$	$N = 12$	17.57×10^4	19.1×10^4	17.9×10^4	6.62%	1.85%	0.93
$49.2 \times 2^N - 31.5$	$N = 15$	13.8×10^5	15.3×10^5	14.3×10^5	6.62%	3.71%	0.93
	$N = 57$	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
$20.50\ N + 72.98$	$N = 320$	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
	$N = 10$	49.77×10^3	49.88×10^3	N/A	0.22%	N/A	N/A
$matmult_{llvm}(N)$=	$N = 15$	15.79×10^4	15.96×10^4	N/A	1.03%	N/A	N/A
$44.71N^3 + 72.47N^2 + 52.52N + 25.49$	$N = 20$	36.29×10^4	36.84×10^4	N/A	1.51%	N/A	N/A
	$N = 25$	69.56×10^4	70.79×10^4	N/A	1.77%	N/A	N/A
	$N = 31$	13.07×10^5	13.78×10^5	N/A	1.98%	N/A	N/A
	$N = 50$; $M = 154$	14.8×10^3	13.5×10^3	N/A	8.67%	N/A	N/A
$concat_{llvm}(N, M)$=	$N = 131$; $M = 69$	14.5×10^3	13.2×10^3	N/A	8.65%	N/A	N/A
$69.14N + 69.14M + 14.12$	$N = 170$; $M = 182$	25.44×10^3	23.25×10^3	N/A	8.60%	N/A	N/A
	$N = 188$; $M = 2$	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
	$N = 15$; $M = 7.6$	4097	4196	N/A	2.40%	N/A	N/A
$sum\_facts_{llvm}(N, M$ †$)$=	$N = 40$; $M = 7.4$	10.7×10^3	10.96×10^3	N/A	2.45%	N/A	N/A
$28.45\ N \times M + 76.71\ N + 22.50$	$N = 80$; $M = 8$	22.7×10^3	23.3×10^3	N/A	2.52%	N/A	N/A
	$N = 160$; $M = 7.8$	44.3×10^3	45.4×10^3	N/A	2.45%	N/A	N/A

Table 2: Comparison of the accuracy of energy analyses at the LLVM IR and ISA levels.

† The argument M refers to the estimated size of an element of the array (which controls the inner loop/recursion).

In their case, retrieving the LLVM IR energy model to a new platform requires performing the statistical analysis again. Our mapping technique requires only to adjust the LLVM IR mapping pass for the new architecture.

8. CONCLUSIONS AND FUTURE WORK

We have presented techniques for adapting 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 existing ISA-level energy model to the LLVM IR level. A second 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 show that performing the static analysis at the LLVM IR level is a good 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.

Regarding future work, the experimental results show that the mapping technique can propagate the existing ISA energy model to the LLVM IR level and that analysis can be done with acceptable error levels. However, we have already identified some cases where the mapping is not as precise as expected, which we are planning to address. The mapping technique currently does not take into consideration the FNOPS introduced in the pipeline under some instruction scheduling scenarios that create inaccuracies. Another source of inaccuracies is due to the code lowered from ϕ (phi) nodes.

Using this mapping technique we plan to obtain an LLVM IR standalone energy model by applying it to a large set of programs and performing a regression analysis to identify energy values for each LLVM IR instruction. Finally, we plan to extend the idea of mapping energy values from the ISA level to the LLVM IR level to also map timing information. This will allow us to incorporate timing analysis in our existing analyses.

LLVM IR is in partial Static Single Assignment (SSA) form (up to the variable names only), which makes it difficult to model memory operations in the presence of derived types (arrays, structures etc.) as well as pointer arithmetic. Particularly in programs where memory locations of a derived type change the control flow of the program and at the same time get updated more than once during program execution. A possible solution to be explored in these scenarios is to convert LLVM IR to hashed SSA.

Finally, the transformation-based analysis approach as well as the mapping techniques showed promising results for sequential programs, however, we plan to extend our techniques to concurrent programs.

Acknowledgements

This research has received funding from the European Union 7th Framework Program agreement no 318337, ENTRA, Spanish MINECO TIN'12-39391 project DOVES, and Madrid TIC/1465 project PROMETIDOS-CM.

9. REFERENCES

[1] The dwarf debugging standard, Oct. 2013. `http://dwarfstd.org/`.

[2] Workshop on Horn Clauses for Verification and Synthesis, July 2014. `http://vsl2014.at/meetings/HCVS-index.html`.

[3] 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.

[4] 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.

[5] A. Bogliolo, L. Benini, G. D. Micheli, and B. Ricc. Gate-Level Power and Current Simulation of CMOS Integrated Circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(4):473–488, 1997.

[6] C. Brandolese, S. Corbetta, and W. Fornaciari. Software energy estimation based on statistical characterization of intermediate compilation code. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pages 333–338, Aug 2011.

[7] 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.

[8] 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.

[9] R. W. et al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[10] K. Georgiou and K. Eder. Mapping an isa energy model to llvm ir. Technical report, University of Bristol, April 2014.

[11] 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.

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

[13] 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.

[14] 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.

[15] S. Kerrison and K. Eder. Energy modelling and

optimisation of software for a hardware multi-threaded embedded microprocessor. Technical report, University of Bristol, June 2013.

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

[17] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, 2004.

[18] 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 *Pre-proceedings of LOPSTR*, September 2013.

[19] 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.

[20] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, April 2008. Extended Abstract.

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

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

[23] 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*, 2014. To Appear.

[24] 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.

[25] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-level Energy Model Supporting Software Optimizations. In *Proceedings of PATMOS*, 2001.

[26] V. Tiwari, S. Malik, and A. Wolfe. *Power analysis of embedded software: a first step towards software power minimization*, pages 222–230. Kluwer Academic Publishers, 1994. 567021.

[27] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proceedings of the Workshop on Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, September 2003.

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

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

Attachment D3.2.5

Static energy consumption analysis of LLVM IR programs.

Submitted to the International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES 2014).

Static energy consumption analysis of LLVM IR programs

Names

Emails

Abstract

Energy models can be constructed by characterizing the energy consumed by executing each instruction in a processor's instruction set. This can be used to determine how much energy is required to execute a sequence of assembly instructions.

However, statically analyzing low-level program structures is hard, and the gap between the high-level program structure and the low-level energy models needs to be bridged. We have developed techniques for performing a static analysis on the intermediate compiler representations of a program. Specifically, we target LLVM IR, a representation used by modern compilers, including Clang. Using these techniques we can automatically infer an approximate upper bound of the energy consumed when running a function under different platforms, using different compilers – without the need of actually running it.

One of the challenges in doing so is that of determining an energy cost of executing LLVM IR program segments, for which we have developed two different approaches. When this information is used in conjunction with our analysis, we are able to infer energy formulae that characterize the energy consumption for a particular program. This approach can be applied to any languages targeting the LLVM toolchain, including C and XC or architectures such as ARM Cortex-M or XMOS xCORE. Our techniques are validated on these platforms by comparing the static analysis results to the physical measurements taken from the hardware. Static energy consumption estimation enables energy-aware software development.

1. Introduction

In embedded systems, low energy consumption is a very important requirement. The software running on these systems has a profound effect on the energy consumed. The design of software and algorithms, the programming language and the compiler together with its optimization level all contribute towards energy consumption of an application. Estimations of energy consumption of programs are very useful to software engineers, so that these can understand the effect of their code on the energy consumption of the final system. Accurate energy consumption and timing analysis of programs involves analyzing low-level machine code representations. However, programs are written in high-level languages with rich abstraction mechanisms, and the relation between the two is often blurred. For instance, optimizations such as dead code elimination, various kinds of code motion, inlining and other clever loop optimization techniques obfuscate the structure of the program and make the resultant code difficult to analyze.

In this paper, we develop a static analyzer that works on the intermediate compiler representation of the program (LLVM IR). Our analysis is based on a well-developed approach in which recursive equations (cost relations) are extracted from a program, representing the cost of running the program in terms of its input [2, 6, 7, 25, 31]. Finally, these cost relations are converted to *closed-form*, i.e. without recurrences, by means of a solver. For example, we can analyze the following program.

```
void proc(int v[], int l)
{
    for (int i = 0; i < l; i++)
      if(v[i] & 1)
        odd();
      else
        even();
}
```

The following CRs are extracted from the program,

$$\begin{array}{lr}
(a)\ C_{\text{proc}}(l) = k_1 + C_{\text{for}}(l,0) & \text{if } l \geq 0 \\
(b)\ C_{\text{for}}(l,i) = k_2 & \text{if } i \geq l \wedge l \geq 0 \\
(c)\ C_{\text{for}}(l,i) = k_3 + C_{\text{odd}}() + C_{\text{for}}(l,i+1) & \text{if } i \leq l \wedge l \geq 0 \\
(d)\ C_{\text{for}}(l,i) = k_4 + C_{\text{even}}() + C_{\text{for}}(l,i+1) & \text{if } i \leq l \wedge l \geq 0
\end{array}$$

where l denotes the length of the array `v`, i stands for the counter of the loop and C_{proc}, C_{odd} and C_{even} approximate, respectively, the costs of executing their corresponding methods. The constraints, denoted on the right hand side of the relations, specify a condition that must be true for the cost relation to be applicable. For instance, relation (a) corresponds to the cost of executing `proc` with an array of length greater than 0 (stated in the condition $l > 0$), where cost k_1 is accumulated to the cost of executing the loop, given by C_{for}. Note that the transition into (c) and (d) is non deterministic. The constants $k_1, \ldots, k_4$ take different values depending on the cost model that one adopts. In this paper, our cost model focuses on energy. These constants are obtained from energy models created at the Instruction Set Architecture (ISA) level [13]. Such models have previously been applied to analysis at the same level [15, 18], and in this paper we propagate this up to the LLVM level.

Many modern compilers such as Clang or XCC are built using the LLVM framework. These internally transform source programs into intermediate compiler representations, which are more amenable to analysis than either source or machine level programs. We show how resource consumption analysis techniques can be adapted and applied to programming languages targeting LLVM IR (such as C or XC [30]) by reusing some of the existing machinery available in the compiler framework (for instance LLVM analysis passes). We show how cost relations can be extracted from programs, such that these can be solved using PUBS (practical upper bounds solver) [2]. Specifically, we focus on *optimized LLVM IR*, that has been compiled with optimization levels `O2` or higher. This ensures that the experiments we perform are realistic and that the techniques can be used for real-world applications.

Time is a significant component of energy consumption, in that a program that computes its result quicker will typically consume less energy by virtue of a shorter run-time. However, the correlation between time and energy varies between architectures, and is related to the complexity of the processor's pipeline [23]. For example, one of the target architectures for this paper exhibits an approximately $2\times$ difference in energy depending on the instructions that are executed, with a similar relationship for the number of threads executed upon it [13]. Analysis of system energy and not just of execution time will therefore garner better information on the energy characteristics of a program.

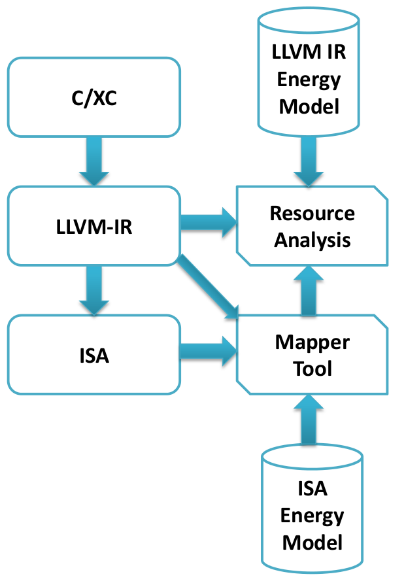


Figure 1. Illustration of the analysis toolchain.

Energy models can be constructed for a processor's instruction set, however this information needs to be constructed, or propagated to a higher level program representation in order to benefit our analysis mechanism. We propose two different techniques (Section 4), for assigning energy to a higher level program representation (LLVM IR). We first propose a mechanism for *mapping* program segments at ISA level to program segments at LLVM IR level. Using this mapping, we can perform a multi level program analysis where we consider the LLVM IR for the structure and semantics of the program and the ISA instructions for the physical effect on the hardware. We also propose an alternative technique, of determining the instruction energy model directly at the LLVM IR level. This is based on empirical data and domain knowledge of the compiler backend and underlying processor. The analysis toolchain is illustrated in Figure 1. The static resource analysis mechanism is described in Section 3. Parts of this mechanism perform a symbolic execution of LLVM IR, which is described in Section 2. The techniques described are built into a tool, which can be integrated into the build process and statically estimates the energy consumption of an embedded program (and its constituent parts, such as procedures and functions) as a function on several parameters of the input data. Our approach is validated in Section 5 on a number of embedded systems benchmarks, on both xCORE and Cortex-M platforms. Finally, we describe related work in Section 6 and conclude in Section 7.

2. Structure and interpretation of LLVM IR

In this section we describe the core language and an important technique we utilize in the resource analysis mechanism (Section 3), which infers energy formulae given an LLVM IR program.

2.1 The LLVM IR language

LLVM IR is a Static Single Assignment (SSA) based representation. This is used in a number of compilers, and is designed to represent high-level languages. For presentation purposes, we first formalize a simple calculus of LLVM IR, based on the following syntax:

$$
\begin{aligned}
inst = \;& \mathsf{br}\; p\; BB_1\; BB_2 \quad \text{(conditional branch)}\\
&\mid x = \mathsf{op}\; a_1..a_n \quad \text{(generic op., no side-effects)}\\
&\mid x = \phi\; \langle BB_1, x_1\rangle..\langle BB_n, x_n\rangle \quad \text{(phi nodes)}\\
&\mid x = \mathsf{call}\; f\; a_1\; ..\; a_n\\
&\mid x = \mathsf{memload} \quad \text{(dynamic memory load)}\\
&\mid \mathsf{memstore} \quad \text{(dynamic memory store)}\\
&\mid \mathsf{ret}\; a
\end{aligned}
$$

We use metavariable names p, f, a, x to describe predicates, function names, generic arguments and variables respectively. The instruction semantics are modeled on the actual LLVM IR semantics [33]. Instruction op represents any side effect free operation such as `icmp` or `add` in LLVM. The ϕ instruction takes a list of pairs as arguments, with one pair for each predecessor basic block of the current block. Each pair contains a reference to the predecessor block together with the variable that is propagated to the current block. The only place where a ϕ instruction can appear is in the beginning of a basic block. Two interesting instructions are memload and memstore. These represent any dynamic memory load and store operation respectively. For instance, `getelementptr` and `load` are some examples of instructions represented by memload. These instructions typically compute pointers dynamically and load data from memory. In our abstract semantics of LLVM IR, we therefore treat variables assigned with values dynamically loaded from memory as unknown (denoted '?').

LLVM IR instructions are arranged in *basic blocks*, labeled with a unique name. A basic block BB over a CFG is a maximal sequence of instructions, $inst_1$ through $inst_n$, such that all instructions up to $inst_{n-1}$ are not branch or return instructions and $inst_n$ is br or ret. The ϕ instructions always appear as the first instructions in a block, as a block can have multiple in-edges. All call instructions are assumed to eventually return.

2.2 Symbolic evaluation of LLVM IR variables

At the core of our resource analysis mechanism of LLVM IR is a symbolic evaluation function $seval$. Given a block of code BB, and a variable x, $seval(BB, x)$ symbolically executes a slice from this block to compute the effect on x. A program slice is a set of instructions that may affect the value of x at some point of interest. During this static analysis phase, we do not simply execute the LLVM IR, but we use a non-standard semantics, which abstracts away dynamic memory reads and writes i.e., memload and memstore. This has the effect of producing simple expressions, which can be handled by the PUBS solver. We proceed by showing examples of actual LLVM IR snippets and showing the effect of this on some variables:

```
LoopBody:
  %i.0 = phi i32 [ %postinc, %LoopIncrement ],
                 [ 0, %allocas ]
  %subscript = getelementptr [0 x i32]* %0, i32 0,
                             i32 %i.0
  %deref6 = load i32* %subscript, align 4
  %not.zerocmp7 = icmp eq i32 %deref6, 0
  br i1 %not.zerocmp7, label %iffalse,
        label %iftrue2
```

In this case, the symbolic evaluation concludes that $\mathit{seval}(BB,$%not.zerocmp7$) = ?$. The evaluation starts at the assignment of %not.zerocmp7, which evaluates to %deref6 == 0. However, since %deref6 is a dynamically loaded value memload, the analyzer concludes that %deref6 is ? and that therefore $\mathit{seval}(BB,$ %not.zerocmp7$) = ?$.

```
iftrue2:
  call void @odd()
  br label %LoopIncrement
```

Sometimes, the code inside a block has no effect on a variable of interest. In this case $\mathit{seval}($%i.0$)$ is %i.0.

```
LoopIncrement:
  %postinc = add i32 %i.0, 1
  %exitcond = icmp eq i32 %postinc, %1
  br i1 %exitcond, label %return, label %LoopBody
```

In this case $\mathit{seval}(BB,$%exitcond$)$ is $($%i.0$+1)$ $==$%1, which is easily found by traversing the structure of the LLVM block backwards.

3. Resource Analysis for LLVM IR

The techniques described here are used to infer cost relations [2]. Cost relations are recursively defined and closely follow the flow of the program. What we actually want to infer is a closed form formula modeling the cost, which is parametric to any relevant input arguments to the program, which requires solving using a cost relation solver. These solvers typically work with simplified control flow graph structures, and therefore we must first perform some simplifications on the control flow graphs, as described in Section 3.3. The analysis then infers block arguments by using symbolic evaluation as described in Section 2.2.

3.1 Inferring block arguments

Block arguments characterize the input data, which flows into the block, and is either consumed (killed) or propagated to another block or function. Unfortunately, solving multi-variate cost relations and recurrence relations automatically is still an open problem, and the fewer arguments each relation has, the easier it is to solve these. For this reason, we designed an analysis algorithm to minimize the block arguments before inferring the cost relations.

The algorithm for inferring block arguments is a data flow analysis algorithm. We use a standard means to describe this algorithm, as in [21]. We define a data flow analysis function gen, which, given a basic block, returns the variables of interest in that block:

$$\mathit{gen}(BB) = \mathit{gen}_{\mathit{blk}}(BB) \cup \mathit{gen}_{\mathit{fn}}(BB)$$

The function $\mathit{gen}_{\mathit{blk}}$ returns the input arguments that affect the branching in a block BB, composed of instructions inst_1 through inst_n, and $\mathit{gen}_{\mathit{fn}}$ returns the variables that affect the input to any external calls in the block. $\mathit{gen}_{\mathit{blk}}$ is defined as follows:

$$\mathit{gen}_{\mathit{blk}}(BB) = \begin{cases} \mathit{ref}(\mathit{seval}(BB,p)) & \text{if } \mathit{inst}_n = [\mathsf{br}\ p\ ..] \\ \emptyset & \text{otherwise} \end{cases}$$

The function ref returns all variables referred to in the symbolically evaluated expression given as argument, for example $\mathit{ref}(x > (y+3))$ returns $\{x, y\}$. We also define function $\mathit{gen}_{\mathit{fn}}$. This returns all the input arguments that affect the parameters given to the function, and is defined as:

$$\mathit{gen}_{\mathit{fn}}(BB) =$$

$$\bigcup_{k=1}^{n} \begin{cases} \bigcup_{i=1}^{m} \mathit{ref}(\mathit{seval}(BB, a_i)) & \text{if } \mathit{inst}_k \text{ is } [x = \mathsf{call}\ f\ a_1\ ..\ a_m] \\ \emptyset & \text{otherwise} \end{cases}$$

The data flow analysis function kill is defined as:

$$\mathit{kill}(BB) = \bigcup_{k=1}^{n} \begin{cases} \{x\} & \text{if } \mathit{inst}_k \text{ is } x = \mathsf{call}\ \ldots \\ \{x\} & \text{if } \mathit{inst}_k \text{ is } x = \mathsf{op}\ \ldots \\ \{x\} & \text{if } \mathit{inst}_k \text{ is } x = \mathsf{memload}\ \ldots \\ \emptyset & \text{otherwise} \end{cases}$$

Finally, we combine gen and kill by utilizing a transfer function, which is inlined into $\mathit{args}_{\mathit{in}}$ and $\mathit{args}_{\mathit{out}}$. These compute the relevant block arguments utilized by our resource analysis. $\mathit{args}_{\mathit{in}}(BB)$ is defined as the function's arguments if BB is the function's first block. In all other cases, $\mathit{args}_{\mathit{in}}$ and $\mathit{args}_{\mathit{out}}$ are defined as:

$$\mathit{args}_{\mathit{in}}(BB) = \bigcup_{BB' \in \mathit{next}(BB)} \mathit{phimap}_{\langle BB, BB' \rangle}(\mathit{args}_{\mathit{out}}(BB'))$$

$$\mathit{args}_{\mathit{out}}(BB) = (\mathit{args}_{\mathit{in}}(BB) - \mathit{kill}(BB)) \cup \mathit{gen}(BB)$$

where phimap maps variables between adjacent blocks BB and BB' based on the ϕ instructions in BB'.

Functions $\mathit{args}_{\mathit{in}}$ and $\mathit{args}_{\mathit{out}}$ are recomputed until their least fixpoint is found. Finally, the block arguments are found in $\mathit{args}_{\mathit{in}}$. The analysis explained in this section is closely related to live variable analysis. A crucial difference, however, is in the function gen. In our case, this returns a smaller subset of variables than live variable analysis i.e., only the ones that may affect control flow.

3.2 Generating and solving cost relations

In order to generate cost relations we need to characterize the energy exerted by executing the instructions in a single block. We also need to model the continuations of each block. Continuations, expressed as calls to other cost relations, arise from either branching at the end of a block, or from function calls in the middle of a block. For instance, consider the following LLVM IR block:

```
LoopIncrement:
  %postinc = add i32 %i.0, 1
  %exitcond = icmp eq i32 %postinc, %1
  br i1 %exitcond, label %return, label %LoopBody
```

This would translate to the following relation:

$$\begin{aligned} C_{\mathrm{LI}}(i) &= C_0 + C_{\mathrm{ret}}(i+1) && \text{if } i+1 = a_1 \\ C_{\mathrm{LI}}(i) &= C_1 + C_{\mathrm{LB}}(i+1) && \text{if } i+1 \neq a_1, \end{aligned}$$

where C_{LI}, C_{ret} and C_{LB} characterize the energy exerted when running the blocks LoopIncrement, return and LoopBody respectively. We therefore refer to C_{ret} and C_{LB} as continuations of C_{LI}. Expressing these calls to other cost relations involves evaluating their arguments, which cannot be done without evaluating the program. Instead, by symbolically executing the block, we can express the arguments of the continuation in terms of the input arguments to the block. In order to do so, we perform symbolic evaluation using the function seval.

The cost relations, extracted from recursive programs using the techniques discussed in this section, can be automatically solved by

PUBS [2] after translating to its proprietary format. In the output, this shows the upper bound obtained, as a formula together with the results of the intermediate steps performed. Internally, PUBS solves these by computing ranking functions and loop invariants. The problem of solving cost relations is composable, i.e., complex functions can be inferred by first inferring simpler ones and composing these together mathematically.

There are cases where the optimized program structures produced by LLVM based compilers prevent the cost relation solvers from finding unique *cover points* in the structure of the cost relations. In order to solve this problem, we need to perform transformations to the call graph upon which we construct our cost relations. This is described in the next section.

3.3 Transformations for control flow graphs

After compilation, nested loop program structures are mangled by compiler optimizations. When the resulting Control Flow Graph (CFG) is directly used to produce CRs, it is usually not possible to infer closed form solutions. For instance PUBS [2] cannot handle complex CFGs, and therefore in order to analyze programs with nested loops, the CFG needs to be simplified. The simplification is actually done at an early stage in the analysis, right after generating an initial CFG, using the following steps:

1. Identify a loop's CFG, A, that has nested loops.
2. Identify the sub-CFG, B, of A that corresponds to the inner loop.
3. Extract B out of A, so that B is a separate CFG. This can be thought of as a new function with multiple return points. Hence B's exit edges are removed.
4. In A, in the place where B used to be, keep the continuation to B. Append a continuation to B's exit targets to B's caller in A.

In order to perform the first two steps, we need to identify the loops in the CFG. While LLVM has specific passes to do so, we had better success when using the algorithm described in [32]. As an example, we show how these steps can be used to transform the CFG of a simple insertion sort, as shown in Listing 1. The original CFG of this program, when compiled using `clang` with optimization level `O2` is shown in Figure 2 (left). In this CFG, the nested loops are identified, which also involves identifying their corresponding entries, re-entries, exit and loop headers. Here, blocks `bb1`, `bb2` and `.backedge` form the inner loop. These blocks are hoisted and the exit edge from `.backedge` (dotted) is eliminated. Instead, `.loopexit` is then called after `bb1` "returns" (Figure 2, right).

The CFG simplifications described in this section preserve the same order of operations when applied to an existing CFG compiled from typical `while` or `for` using `clang` or `xcc`. This means that the program called in the left-side of Figure 2 will consume as much energy as the program in the right-side Figure 2. The only limitation of this approach is when an induction variable of an outer loop is modified in an inner loop. In this case the transformation cannot occur, however we have not encountered real benchmarks where this takes place.

In order to verify the transformation with respect to energy, let us consider a typical `while` or `for` loop and show that the same sequence of blocks is called after the transformation takes place. We can assume that such a loop has a single header, but may have multiple exits or reentries and induction variables of the outer loops are not modified in the inner loops. After the transformation takes place on a nested loop structure (B inside A), B is still called from A, however B's exit edges are now removed. The target of B's exit edges will still be called after B completes. This is because we have appended a continuation in A to this target, in Step 4. Hence all

```
void sort(int numbers[], int size) {
  int i=size, j, temp;
  while(j = i--) {
    while(j--) {
      if(numbers[j] > numbers[i]) {
        temp = numbers[i];
        numbers[i] = numbers[j];
        numbers[j] = temp;
      }
    }
  }
}
```

Listing 1. This insertion sort demonstrates that certain classes of programs require further analysis or transformation.

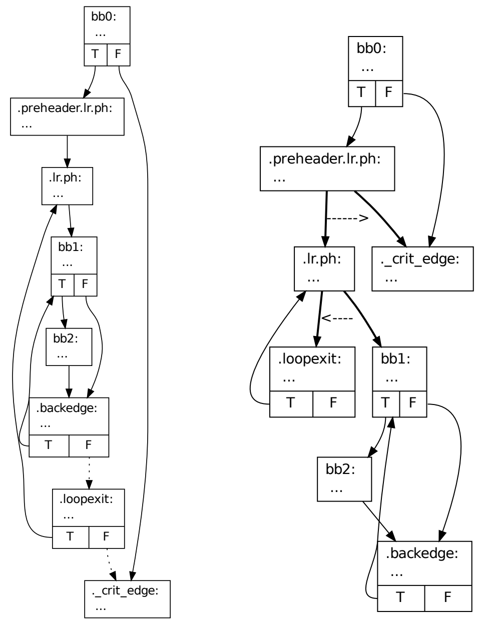


Figure 2. CFG of an insertion sort compiled using `clang` with optimization level `O2` before (left) and after (right) simplification.

blocks will be called in the same sequence. The argument above can be inductively applied to loops with arbitrary nesting levels.

4. Computing energy cost of LLVM IR blocks

The intermediate representation used by LLVM is architecture independent. Any given LLVM IR sequence can be passed to one of many different backends, including ISAs [16]. The exact implementation of the ISA determines the energy consumed by each instruction that is executed. Thus, the conversion to machine code, together with the processor implementation, affects the energy consumption of an instruction at the LLVM IR level.

For static analysis of LLVM IR to produce useful energy formulae for programs, a method of assigning an energy cost to an LLVM IR segment must be used. Two possible methods are demonstrated in this paper:

1. *ISA energy model w/mapping*. LLVM IR is mapped to its corresponding ISA instructions and the energy cost is obtained from

the ISA level cost model. The advantage is that it is simpler to characterize at ISA level, however this requires an additional step to correlate LLVM with ISA instructions.

2. *LLVM energy model.* Attributing costs directly to LLVM IR removes the need for a mapping. However, it necessarily simplifies the energy consumption characteristics, reducing accuracy.

In principle, both methods can be explored for both architectures. This paper utilizes an ISA level model for the XMOS processor. The Cortex-M is modeled at the LLVM IR level directly.

4.1 XMOS XS1-L ISA level modeling

The aim of ISA level modeling is to associate machine instructions with an energy cost. To achieve this, energy consumption samples must be collected and an appropriate representation of the underlying hardware must be used as a basis for the model. A single-threaded model, such as that defined by Tiwari [27] and expressed in Equation 1, describes the energy of a sequence of instructions, or program.

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

The program's energy, E_{prog}, is first formed from the base cost, B_i of all instructions, i, in the ISA, multiplied by the occurrences, N_i, of each instruction. For each transition in a sequence of instructions, the overhead, $O_{i,j}$, of switching from instruction i to instruction j, multiplied by the number of times the combination i, j occurs, $N_{i,j}$. Finally, for a set of k external effects, the cost of each of these effects, E_k is added. For example, these external effects may represent the cache and memory costs, based on the cache hit rate statistics of the program.

The XS1-L architecture implements multi-threading in a hardware pipeline. Even for single-threaded programs, we need to consider the behavior of this multi-threaded pipeline. The power of individual instructions varies by up to $2\times$, with multi-threading introducing up to a $1.6\times$ increase with a $4\times$ performance boost. This means execution time and energy are related in a more complex way than a simpler single-threaded architecture. The model for the XS1-L is built upon existing work of [28] and the more detailed [26], which obtain model data through the energy measurement of specific instruction sequences, and create a representation of some of the processor's internal structure in the model equations. A full description of the XS1-L's energy characteristics and the model is given in [13].

To extend a Tiwari style approach to model the XS1-L processor, two new characteristics must be accounted for: idle time and concurrency. The XS1 ISA has a number of event-driven instructions, which can result in the processor executing no instructions for a period of time, until the event occurs. Furthermore, the multi-threaded pipeline permits only one instruction from a given thread to be present in the pipeline at any one time. These changes are expressed in Equation 2. Here, the energy exerted by running a program depends on a base power, P_{base}, which represents the energy cost when no instructions are executed, multiplied by the number of idle periods, N_{idle}. The clock period of the processor, T_{clk} is also introduced, to allow different clock speeds to be considered. The inter-instruction overhead, previously described in Equation 1 as $O_{i,j}$, is generalized to a constant overhead, O, due to the unpredictability of instruction interaction between threads. For each instruction, the base cost is added to the instruction cost, P_i, which is scaled by the overhead and an additional scaling factor based on the number of active threads, M_t. This is multiplied by the number of occurrences of this instruction at t threads, $N_{i,t}$ and the clock period, T_{clk}. This is done for the varying number of threads, t that may be active in the program over its lifetime.

$$\begin{aligned} E_{\text{prog}} &= P_{\text{base}} N_{\text{idle}} T_{\text{clk}} \\ &+ \sum_{t=1}^{N_t} \sum_{i \in \text{ISA}} ((M_t P_i O + P_{\text{base}}) \times (N_{i,t} T_{\text{clk}})) \end{aligned} \tag{2}$$

The multi-threaded ISA level model for the XS1-L requires that for each level of concurrency, t, the number of instructions executed at that level should be known, or estimated. If a single threaded program is run on its own on the XS1-L and there are no idle periods, then Equation 2 simplifies to Equation 3, where the idle accounting is removed, and only the first threading level, $t = 1$, is considered.

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

The current analysis effort focuses upon single threaded experiments, thus Equation 3 can be used. Multi-threaded analysis is proposed as future work in Section 7.

4.2 XMOS LLVM IR energy characterization by mapping

To enable the analysis at the LLVM IR level we need a mechanism to propagate the existing energy model at the ISA level up to the LLVM level. The mapping technique described in this section serves this purpose by creating a fine grained mapping between segments of ISA instructions to LLVM IR instructions, in order to enable the energy characterization of each LLVM IR instruction in a program. A full description of the mapping techniques is given in [8].

Our mapping technique leverages the existing debug mechanism in the XMOS compiler toolchain. This mechanism is originally meant to facilitate the debugging process of an application, particularly when stepping through a program line by line. During the lowering phase of the compilation process, the LLVM IR code is transformed to the specific ISA code by the backend. The debug information (DI) is also stored alongside with the ISA code using the DWARF standard [1], a standardized debugging data format used by many compilers and debuggers to support source level debugging. By tracking this information we can extract an n:m relationship between the two levels, because one source code instruction can be related to many different sequences LLVM IR instructions and therefore many different sequences of ISA instructions. Because this n:m relation complicates static analysis, there is a need for a more fine grained mapping.

To address this issue, we created an LLVM pass that traverses the LLVM IR and replaces the *Source Location Information* with LLVM IR location information, right after all the optimization passes and just before emitting the ISA code. In this way, we can extract a 1:m relationship between the mapping of LLVM IR instructions and ISA instructions. Also, by doing it after the LLVM optimizations passes the optimized LLVM IR is closer to the ISA code than the unoptimized one, which will go through a series of transformations. There are optimizations that happen during the lowering phase, such as peephole optimizations and some late target specific optimizations that can affect the mapping. However, the effect of these optimizations on the structure of the code is not as profound as those applied to LLVM IR. After a mapping is extracted for a particular program, the associated energy values for the ISA instructions corresponding to a specific LLVM IR instruction are aggregated and then associated with the LLVM IR instruction, and finally to every LLVM IR block.

Although we use the XMOS tool-chain for the mapper tool, the approach is generic and transferable, due to the use of the common LLVM optimizer and code generator, and the use of the DWARF standardized debugging data format, used by many compilers and debuggers to support source layer debugging.

4.3 LLVM IR energy model for ARM

An energy model for ARM Cortex-M series is applied directly at the LLVM IR level, based upon empirical energy measurement data, and knowledge of both the processor architecture and the compiler backend. The Cortex-M3 model is for the most part a simplification of the Tiwari model [27], applied at the LLVM IR level. The simple, embedded nature of this processor removes the need to model external effects such as cache misses, and the effect of the switching cost between instructions is approximated into the actual instruction cost. Through analysis of energy measurements for a large set of the target ISA instructions, it was found that LLVM IR instructions can be segmented into four groups: memory, M, program flow, B, division, D, and all other instructions, G.

The LLVM IR syntax described in Section 2 can be related to these groupings. In particular, br, call and ret can be combined into group B; memload and memstore are members of M; the subset of op relating to division make up group D; and finally, ϕ and all remembering members of op form group G.

This yields a model equation that accumulates the energy of a program based on the number of instructions executed from each group. Equation 4 considers each group, which is assigned an energy cost, which combined give the total program energy, E_{prog}, where E_i is the energy cost of a single instruction in group i, and N_i is the number of instructions executed in that group.

$$E_{\text{prog}} = \sum_{i \in \{M,B,D,G\}} E_i N_i \qquad (4)$$

In addition, there are a number of other factors that affect energy, due to the relation between the LLVM IR and the ISA:

1. **Variadic arguments.** LLVM has instructions with variadic arguments. Typically, the number of arguments in the instruction affects the energy consumed in a linear manner.
2. **Data types.** LLVM operations op can be performed on values of different data types. If the data type is larger than 32 bits, or floating point, this will translate into a larger number of ISA instructions on a Cortex-M with no floating point unit.
3. **Predicated instructions.** The Cortex-M processor is capable of executing predicated instruction sequences. In some cases, short LLVM IR blocks originating from ternary expressions in the original source code are directly translated to a number of predicated instructions in the ARM ISA. Therefore, the number of ISA instructions generated could be less than the instructions in LLVM IR, and the static analysis over-approximates the energy consumption of these blocks.

Factors (1) and (2) can be accounted for by parameterizing the LLVM IR energy model. For instance, consider the following call instruction:

```
%6 = call i32 @min(i32 %boptmp88, i32 %boptmp96)
```

This translates to a single branch instruction in the ARM ISA, with surrounding register moves to ensure the correct calling convention:

```
mov r0, r4     # move arg1 into r0nnn
mov r1, r5     # move arg2 into r1
bl min         # call min
mov r4, r0     # move the result into r4
```

As we can see, the energy consumed by an LLVM call instruction is parametric in the number and types of the arguments and return value.

Benchmark	L	NL	A	B	C
base64	×		×	×	
mac	×		×		
levenshtein	×	×	×	×	
insertion sort	×	×	×		
matrix multiply	×	×	×		
gcd	×			×	×
jpegdct	×	×	×	×	×

Table 1. Benchmark Characteristics.

5. Experimental Evaluation

We have selected a series of benchmarks of core algorithmic functions, particularly from the BEEBS [22] and MDH WCET benchmark [9] suites. These are collections of open source benchmarks for deeply embedded systems, slightly modified to work with our test harness. The benchmarks are single threaded, reflecting the scope of the analysis performed in this paper. Table 1 summarizes the characteristics of the benchmarks and the meaning of the last 5 columns is as follows: (L) contains loops, (NL) contains nested loops, (A) uses arrays and/or matrices, (B) contains bitwise operations, (C) contains loops with complex control flow predicates.

In order to show that our techniques are applicable to multiple languages and platforms, we have ported some of the benchmarks from C to XC. Porting C code to XC typically does not involve rewriting, since the syntax is very similar and they both use the same preprocessors. However, since XC does not provide pointers some changes need to be made to the benchmarks during the porting process. For the benchmarks that run on the xCORE, we have used the XC compiler, version 13. For Cortex-M benchmarks we have used Clang version 3.5. We proceed by describing the benchmarks. In both cases, the benchmarks are compiled under optimization level O2.

GCD. This benchmark is an implementation of the Euclidean algorithm, which computes the greatest common divisor between any two numbers. This is implemented using an iterative style and parameterized with its two input numbers (A and B).

Insertion sort. The code of the main function is shown in Figure 1. The energy exerted by the insertion sort partly depends on how many swaps need to take place, and this is dependent on the actual data present inside the array. Since our analysis infers approximate upper bounds, we will be measuring the energy consumed in sorting a reverse-ordered list, and comparing this to the statically inferred formula. Note that the number of iterations in the inner loop depends on an induction variable in the outer loop. This benchmark is parameterized by the length of the list to be sorted, P.

Matrix multiply (BEEBS/MDH WCET). We slightly modified this so that it can work with matrices of various sizes. The matrices are all square, of size P.

Base64 encode. Computes the base64 encoding[1] as a string, given an input string of length P.

MAC (MDH WCET). Dot product of two vectors together with sum of squares. This is parameterized by the length of the vectors, P.

Jpegdct (MDH WCET). Performs a JPEG discrete cosine transform. Taken from the MDH WCET benchmark suite. This benchmark is not parameterized.

[1] Posted by user2859193 on stackoverflow.com/questions/342409

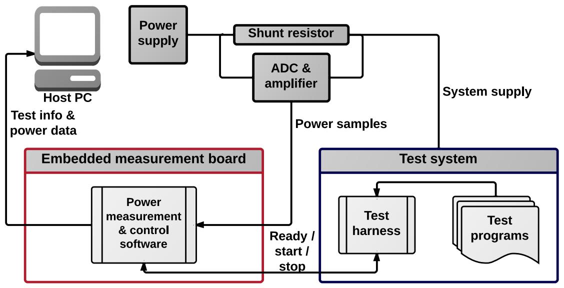


Figure 3. An illustration of our measurement systems for our test platforms.

Levenshtein distance (BEEBS). Computes the minimum number of edits to change one string into another. The lengths of the two strings are parameterized with the variables A and B.

5.1 Experimental Setup

For both ARM and XMOS platforms, power measurement data is collected by using appropriately instrumented power supplies, a power sense chip and an embedded system running control and data acquisition software. The implementations differ, but are structurally very similar. Both of these periodically calculate the power using Equation 5 during a test run by sampling the voltage on either side of a shunt resistor (V_{bus} and V_{shunt}) to determine the supplied current.

$$P = I \times V_{\text{bus}} \quad \text{where} \quad I = \frac{V_{\text{bus}} - V_{\text{shunt}}}{R_{\text{shunt}}} \tag{5}$$

For the Cortex-M processor, the measurements are taken on an ST Microelectronics STM32VLDISCOVERY board while for the xCORE, a custom XMOS board with an XS1-L based XS1-U16A chip is used.

5.2 Results

The results for the XMOS xCORE and ARM Cortex-M processors are shown in Figures 4 and 5, respectively. These graphs show the insertion sort, matrix multiplication and mac benchmarks, with data series for the static analysis results and actual energy measurements. The static analysis closely fits the empirical results, validating our approach. Table 2 shows the formulae and final errors for all benchmarks. Overall, the final error is typically less than 10% and 20% on the XMOS and ARM platforms respectively, showing that the general trend of the static analysis results can be relied upon to give an estimate of the energy consumption. We explain the sources of error in our results below:

Simple LLVM IR energy model (ARM). For the case of Cortex-M the errors in the analysis mostly stem from the greatly simplified model of energy consumption in the Cortex-M. The LLVM energy model used for the Cortex-M assigns an energy cost to each IR instruction. Therefore, when an IR instruction expands to unexpected, or many ISA level instructions, the energy consumption can be inaccurate. In particular, for base64, ternary operators are heavily used inside its main loop. In LLVM IR, this introduces a number of short conditional blocks inside this loop. These multiple basic blocks in LLVM IR are translated to a smaller number of predicated instructions in the ARM ISA by the compiler, so the static analysis will over approximate the energy consumed.

Measurement error. Measurement errors are introduced from environmental factors such as temperature and power supply fluctua-

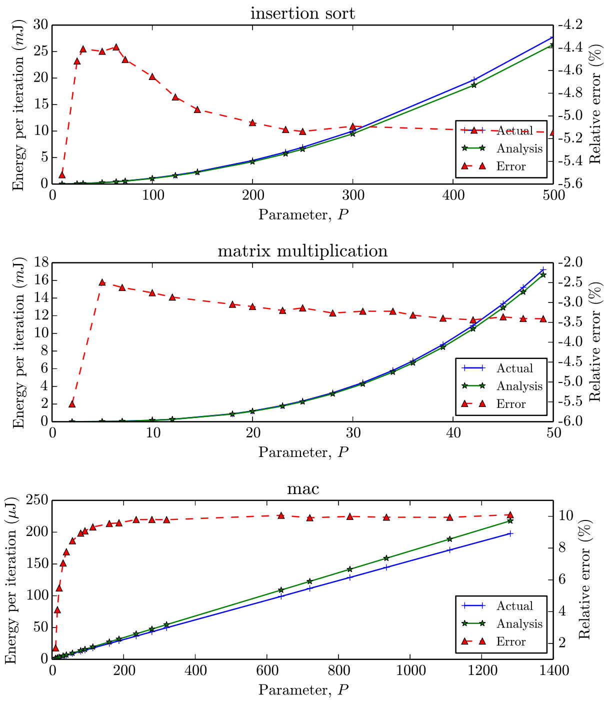


Figure 4. The measurement results and static analysis for the XMOS processor.

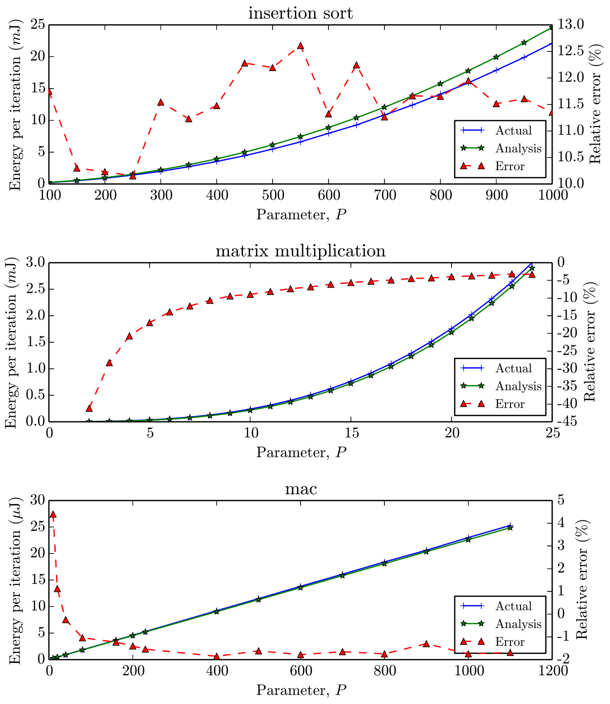


Figure 5. The measurement results and static analysis for the Cortex-M processor.

Benchmark[*]	Formulae		Final error (%)	
	ARM (nJ)	XMOS (nJ)	ARM	XMOS
base64	$158 + 94 \cdot \lfloor \frac{P-1}{3} \rfloor$	$1270 + 734 \cdot \lfloor \frac{P-1}{3} \rfloor$	28.0	1.1
mac	$23P + 14$	$133P + 192$	-1.7	10.1
levenshtein	$47AB + 14A + 31B + 44$	$559AB + 78A + 571 + \max(225B, 180B + 213)$	7.0	0.4
insertion sort	$25P^2 + 11P + 7.1$	$105P^2 + 30P + 75$	11.1	3.0
matrix multiply	$20P^3 + 13P^2 + 97P + 84$	$144P^3 + 200P^2 + 77P + 332$	-3.3	-3.4
gcd	$(22 + 15 \cdot \log_2 A)$ pJ	$(272 + 195 \cdot \log_2 A)$ pJ	N/A[†]	N/A[†]
jpegdct	54 mJ[‡]	463 mJ[‡]	8.5	2.6
Mean relative error			9.9	3.4

Table 2. Formulae and error values for each benchmark.

[†] The error in this benchmark is data dependent; [‡] This benchmark was not parameteric, thus is not parameterised.
[*] Benchmark sources are available from `http://anonymized.for.review.com/`

```c
void function(int A, int B)
{
  int i;

  if(A < B)
    for(i = 2*A; i >= 0; i--)
      ...
  else
    for(i = B; i >= 0; i--)
      ...
}
```

Listing 2. Example of a case where the analysis infers a `max` function.

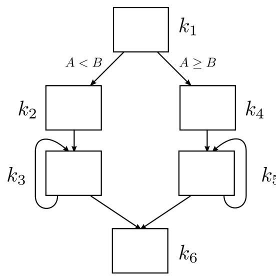


Figure 6. Control flow graph of the function given in Listing 2.

tions. The tolerance of the components is also another factor. Another important factor is the test harness itself, which has to call a function repeatedly in order to get its energy measurements. The loop surrounding this function, together with the act of calling can be a significant overhead when the amount of computation inside the function is low. In fact, we can see that in all cases except GCD, the relative error converges to a single error result. This is expected because in all of the benchmarks the parameter controls the number of iterations performed in one or more loops. As the parameter increases, the difference in the constant energy overhead is minimized, with respect to the energy consumption of the function under test. Measurement runs were run numerous times to ensure consistency of results within the expected error margins described above.

Data flow through the processor's execution units. The energy models for the xCORE and ARM assume a random distribution of operand data. In practice, however, operations such as logical tests, bit-manipulation and instructions performed on shorter data types such as `char` will not use the full bit-range of the data path. In cases such as these, energy consumption will be lower, therefore introducing some estimation inaccuracy.

LLVM IR to ISA mapping (xCORE). In the case of the xCORE, the overall results are better than that of the Cortex-M. This is due to a more accurate assignment of energy values to LLVM-IR instructions, which the mapper can produce for each individual program, as described in Section 4.2. Nevertheless, the mapper introduces analysis error. For instance, the mapper does not consider instruction scheduling on the processor, where an instruction fetch stall can happen in some limited scenarios. This can be addressed by performing a further local analysis on the ISA code to determine the possible locations where this happens, and adjusting the energy accordingly. Another problem arises when mapping LLVM IR `phi` instructions, to the "corresponding" ISA code. This code is sometimes hoisted out of loops at a later compilation phase. Also, multiple ISA instruction sequences that are conditionally executed sometimes map to a single `phi` instruction. This phenomenon was partially addressed by automatically adjusting the energy of `phi` instructions in the mapping in such cases.

Static analysis and data dependence. Programs where the behavior and state depends on complex properties of the actual input data are problematic for static resource analysis. An extreme example of such a program would be an interpreter. The execution time of an interpreter not only depends on the size of the program file it is supplied, but also on the contents of this file. A more typical example would be the euclidean algorithm (GCD), where the number of steps taken to execute depends on a relationship between its parameters A and B. Our static analysis technique, however, still manages to compute an approximate, logarithmic upper bound, which is dependent on only one of the arguments. Part of the reason why we can analyze programs of this type is that symbolic evaluation of modulus between two variables $x \bmod y$ returns an upper bound of $y - 1$, a lower bound of 0 and an approximation of $(y - 1)/2$.

The `levenshtein` cost function for the xCORE processor includes a `max` function, making it a different type of formula to the Cortex-M's cost function. This occurs when a data dependent branch is on the upper bound of the function and the analysis is unable to resolve the branch statically, possibly because the branching is data dependent. An example of this is shown in Listing 2. The analysis cannot statically ascertain the outcome of the $A < B$ expression, so simply returns the cost function as the maximum of the two possible branches:

$$function = k_1 + \max(k_2 + 2 \cdot k_3 \cdot A, k_4 + k_5 \cdot B) + k_6,$$

where $k_1, ..., k_6$ are the costs of executing the respective basic blocks, as seen in Figure 6. The same effect causes `max` to appear in the xCORE's formula — there is a data dependent `if` statement in an inner loop of `levenshtein`.

5.3 Composability

All of the benchmarks so far have consisted of relatively simple code, for which a single function is analyzed. However, the analysis can handle nesting and recursion, in the same way that it can handle functions with multiple basic blocks. In the code in Listing 3, the `levenshtein` and modified `insertion sort` functions are composed into a simple spell checker — for a given string, sort the list of strings by the `sortbysimilarity` to the target string.

In this listing, `dictword_len` is the maximum size of the strings in `dictionary`. Inferring a cost formula for this program does not present any issues as long as it is possible to infer formulas for its

```c
int distances[MAX];

void sortbysimilarity(char *word, int word_len,
      char *dictionary[], int dictword_len,
      int n_strings)
{
    int i = n_strings;

    while(i--) {
      distances[i] = levenshtein(word,
              dictionary[i], word_len,
              dictword_len);
    }

    sort(distances, dictionary, n_strings);
}
```

Listing 3. Sort by similarity function, demonstrating that the analysis can be composed across multiple functions.

constituent parts. Our techniques construct Cost Relations (CRs) from the program that is being analyzed. An important feature of CRs is their compositionality. This allows computing upper bounds of CRs composed of multiple relations by concentrating on one relation at a time. The process starts by computing upper bounds for cost relations which do not depend on any other relations, which we refer to as stand-alone cost relations, and continues by replacing the computed upper bounds on the equations which call such relations. For instance, for the above program `levenshtein_distance` has an associated energy cost of

$$\big(A(53B + 16) + 35B + 31\big)\ \text{nJ}, \tag{6}$$

where A and B are the third and fourth arguments to the function. Our modified string sorting routine has a cost of:

$$\big(37A^2 + 14A + 14\big)\ \text{nJ}. \tag{7}$$

These functions are systematically combined together so that a cost for `sortbysimilary` is computed. In this case it is

$$\big(530ABC + 157AC + 346BC + 366C^2 + 629C + 210\big)\ \text{nJ}, \tag{8}$$

where A is `word_len`, B is `dictword_len` and C is `n_strings`.

6. Related Work

Related work exists in four different areas: energy modeling of processors, mapping low-level program segments to higher level structures, static resource usage analysis and worst-case execution time analysis (WCET).

Energy models of processors for program analysis require energy consumption data in relation to the program's instructions. This data can be collected by simulating the hardware at various levels, including semiconductor [17] and CMOS [4]. Alternatively, higher level representations may be used such as functional block level [26] that reflects the micro-architecture, direct measurement on a per-instruction basis [27], or by profiling the energy consumption of commonly used software blocks [24]. Higher level data collection and modeling efforts are typically quicker to use once the data has been acquired, as there is less computational burden than a low-level simulation. However, the accuracy may be lower, therefore a suitable trade-off must be met.

Although substantial effort has been devoted to ISA energy modeling, there is not a lot of work done for higher level program representations. This is mostly because precision decreases when moving further away from the hardware. One of the most recently pertinent works for LLVM IR energy modeling is [5]. The authors performed statistical analysis and characterization of LLVM IR code, together with instrumentation and execution on the host machine, to estimate the performance and energy requirements in embedded software. In their case, retrieving the LLVM IR energy model to a new platform requires performing the statistical analysis again. Our LLVM IR energy model takes into consideration types and other aspects of the instructions. Furthermore, our mapping technique requires only to adjust the LLVM mapping pass for the new architecture.

Static cost analysis techniques based on setting up and solving recurrence equations date back to Wegbreit's [31] seminal paper, and have been developed significantly in subsequent work [2, 6, 7, 20, 25, 29]. In [18] this approach is applied to inferring statically the energy consumption of Java programs as functions of input data sizes, by specializing a generic resource analyzer [10, 20] to Java bytecode analysis [19]. However, this work did not compare the results with measured energy consumptions. In [15] the approach is applied to the energy analysis of XC programs using ISA-level models [13], and the results are compared to actual hardware measurements. Our analysis continues in this line of work but with a number of important differences. First, analysis is performed at the LLVM-IR level and we propose novel techniques for reflecting the ISA-level energy models at the LLVM-IR level. Also, instead of using a generic resource analyzer (requiring translating blocks to its Horn Clause-based input syntax) and delegating the generation of cost equations to it, we generate the equations directly from the LLVM-IR compiler representation, performing control flow simplifications, and reducing the number of variables modelled by the analysis mechanism. Finally, we study a larger set of benchmarks. There exist other approaches to cost analysis such as those using dependent types [11], SMT solvers [3], or *size change abstraction* [34].

As discussed in Section 1, energy and time are often correlated to some degree. Techniques such as implicit path enumeration [14] are often used in worst-case execution time analysis of programs. In most cases, programs are assumed to be preprocessed such that no loops are present (e.g. using loop unrolling). Some approaches such as in [12] focus on statically predicting cache behavior. WCET analysis is concerned with getting an absolute worst-case timing for hard real-time systems. In practice, for energy consumption analysis we typically are more interested in average cases. Also, most WCET analysis approaches produce absolute timing figures. In our case, we infer energy formulae parameterized by the program's input.

7. Conclusion and Future Work

In this paper we have introduced an approach for estimating the energy consumption of programs based on the LLVM compiler framework. We have shown that this approach can be applied to multiple embedded languages (such as C or XC), compiled using optimization level `O2` with different compilers (such as Clang or XCC). We have also validated this approach for multiple backends, via two target architectures: ARM Cortex M3 and XMOS XS1-L. Our approach is validated by comparing the static analysis to physical measurement taken from the hardware. The results on our benchmarks show that energy estimations using our technique are within 10% and 20% or better in the case of the xCORE and the Cortex-M processors, respectively.

Although the techniques discussed here were initially designed for single threaded programs, these can be adapted to multi-threaded programs. For these programs, we also need to take the synchronization time into consideration. For example, the XC language has explicit constructs for thread communication using channels, and therefore the blocking communication between threads needs to be modeled. In order to do so, we can analyze the communication throughput of individual threads using techniques discussed in this paper. Using this information we can estimate the time between events happening on channels and hence the utilization of the processor. This, coupled with multi-threaded energy models as discussed in Section 4.1, can be used to analyze multi-threaded programs.

An interesting direction is to further develop the assignment of energy to LLVM IR program segments. In particular, an LLVM IR energy model for the xCORE can be implemented by using the information gathered from the mapping technique together with statistical analysis. The mapping technique used for the xCORE can also be adapted for the ARM case. We aim to further develop our techniques so they can be applied and evaluated against other embedded processor architectures, such as MIPS, or other ARM variants.

Finally, the static analysis techniques can be improved further. Currently the biggest limitation is solving the cost relations. Cost relations could also be solved numerically. In some cases this can produce tighter upper bounds and enable us to analyze more complex programs. An implementation of this can be used when actual formulae are not required.

References

[1] The dwarf debugging standard, Oct. 2013. `http://dwarfstd.org/`.

[2] 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.

[3] D. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. 7460:405–421, 2012.

[4] A. Bogliolo, L. Benini, G. D. Micheli, and B. Ricc. Gate-Level Power and Current Simulation of CMOS Integrated Circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(4):473–488, 1997.

[5] C. Brandolese, S. Corbetta, and W. Fornaciari. Software energy estimation based on statistical characterization of intermediate compilation code. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pages 333–338, Aug 2011.

[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 and K. Eder. Mapping an isa energy model to llvm ir. Technical report, University of Bristol, April 2014.

[9] J. Gustafsson. The Mälardalen WCET benchmarkspast, present and future. *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

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

[11] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.

[12] N. D. Jones and M. Müller-Olm, editors. *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, Lecture Notes in Computer Science. Springer, 2009.

[13] S. Kerrison and K. Eder. Energy modelling and optimisation of software for a hardware multi-threaded embedded microprocessor. Technical report, University of Bristol, June 2013.

[14] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 88–98, 1995.

[15] 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 *Pre-proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, September 2013.

[16] LLVM Project. Writing an LLVM backend. `http://llvm.org/docs/WritingAnLLVMBackend.html`, 2014. Accessed: 2014-03-11.

[17] L. W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, EECS Department, University of California, Berkeley, 1975.

[18] 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.

[19] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 6–86. Elsevier - North Holland, March 2009.

[20] 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.

[21] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[22] J. Pallister, S. Hollis, and J. Bennett. BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms. 2013.

[23] J. Pallister, S. J. Hollis, and J. Bennett. Identifying Compiler Options to Minimise Energy Consumption for Embedded Platforms. *Computer Journal*, 2013.

[24] G. Qu, N. Kawabe, K. Usami, and M. Potkonjak. Function-level power estimation methodology for microprocessors, 2000. 337786 810-813.

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

[26] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Hrain Instruction-level Energy Model Supporting Software Optimizations. In *Proceedings of PATMOS*, 2001.

[27] V. Tiwari, S. Malik, and A. Wolfe. *Power analysis of embedded software: a first step towards software power minimization*, pages 222–230. Kluwer Academic Publishers, 1994. 567021.

[28] V. Tiwari, S. Malik, A. Wolfe, and M. T. C. Lee. Instruction level power analysis and optimization of software. In *VLSI Design, 1996. Proceedings., Ninth International Conference on*, pages 326–328, 1996.

[29] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proceedings of the International Workshop on Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, Sept. 2003.

[30] D. Watt. *Programming XC on XMOS Devices*. XMOS Ltd., 2009.

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

[32] T. Wei, J. Mao, W. Zou, and Y. Chen. A new algorithm for identifying loops in decompilation. In *Proceedings of SAS*, pages 170–183, 2007.

[33] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of POPL*, POPL '12, pages 427–440, 2012.

[34] 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.