ENTRA

318337

Whole-Systems ENergy TRAnsparency

A General Framework for Resource Consumption Analysis and Verification

Deliverable number:	D3.1
Work package:	Analysis and Verification (WP3)
Delivery date:	1 October 2013 (12 months)
Actual date:	13 November 2013
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 the general program analysis framework that we have developed, and how it can be instantiated for inferring particular resource usages, such as energy, which is our main motivation. It includes the following attachments:

- D3.1.1. *Energy Consumption Analysis of Programs based on XMOS ISA-Level Models*. Published at the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13).
- D3.1.2. *Sized Type Analysis for Logic Programs*. Published as a technical communication in Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement.
- D3.1.3. *Towards an Abstract Domain for Resource Analysis of Logic Programs Using Sized Types*. Published at the 23rd Workshop on Logic-based Methods in Programming Environments (WLPE 2013).
- D3.1.4. *Genetic Algorithm-based Allocation and Scheduling for Voltage and Frequency Scalable XMOS Chips*. Published at Hybrid Artificial Intelligent Systems (HAIS 2013).
- D3.1.5. *A Coverage Model to Capture the Communication Behaviour of Multi-Threaded Message-Passing Programs*. Submitted to the International Conference on Software Testing (ICST 2014).
- D3.1.6. *Operational Semantics for XC*.

Contents

1 Introduction

This deliverable describes the general program analysis framework we have developed, and how it can be instantiated for inferring particular resource usages. A resource is a very general notion that includes energy, which is our main motivation, execution time, and other non-functional program properties about the whole execution of programs or program segments, such as procedures or functions.

We have designed the framework to meet the following **requirements**:

1. Supporting energy transparency through the system layers. This implies that the framework should allow the relation of higher layer properties which are independent from the underlying hardware, such as the number of times a procedure, loop, or instruction are executed, with lower layer properties such as energy and execution time.

2. The framework should be parametric with respect to resources and cost models. In particular, it should give support to easily:

   - define resources and
   - express how different basic elements of a program affect its resource usage. This implies an easy use of the models for lower system layers developed in WP2.

3. The resource information inferred by the analysis is:

   - given as functions of data sizes and other needed parameters,
   - expressed in the common assertion language described deliverable D2.1 (in particular, in the internal assertion language aspect of the common assertion language, IAL), and
   - used for verification (WP3), optimization (WP4), and to help the system developers to better understand the effect of their designs in energy consumption (WP1).

4. The framework is flexible and powerful enough to support a wide range of resources and other (auxiliary/instrumental) program properties, focusing on those that we foresee to be used in next stages of the project.

5. The framework should support probabilistic resource analysis.

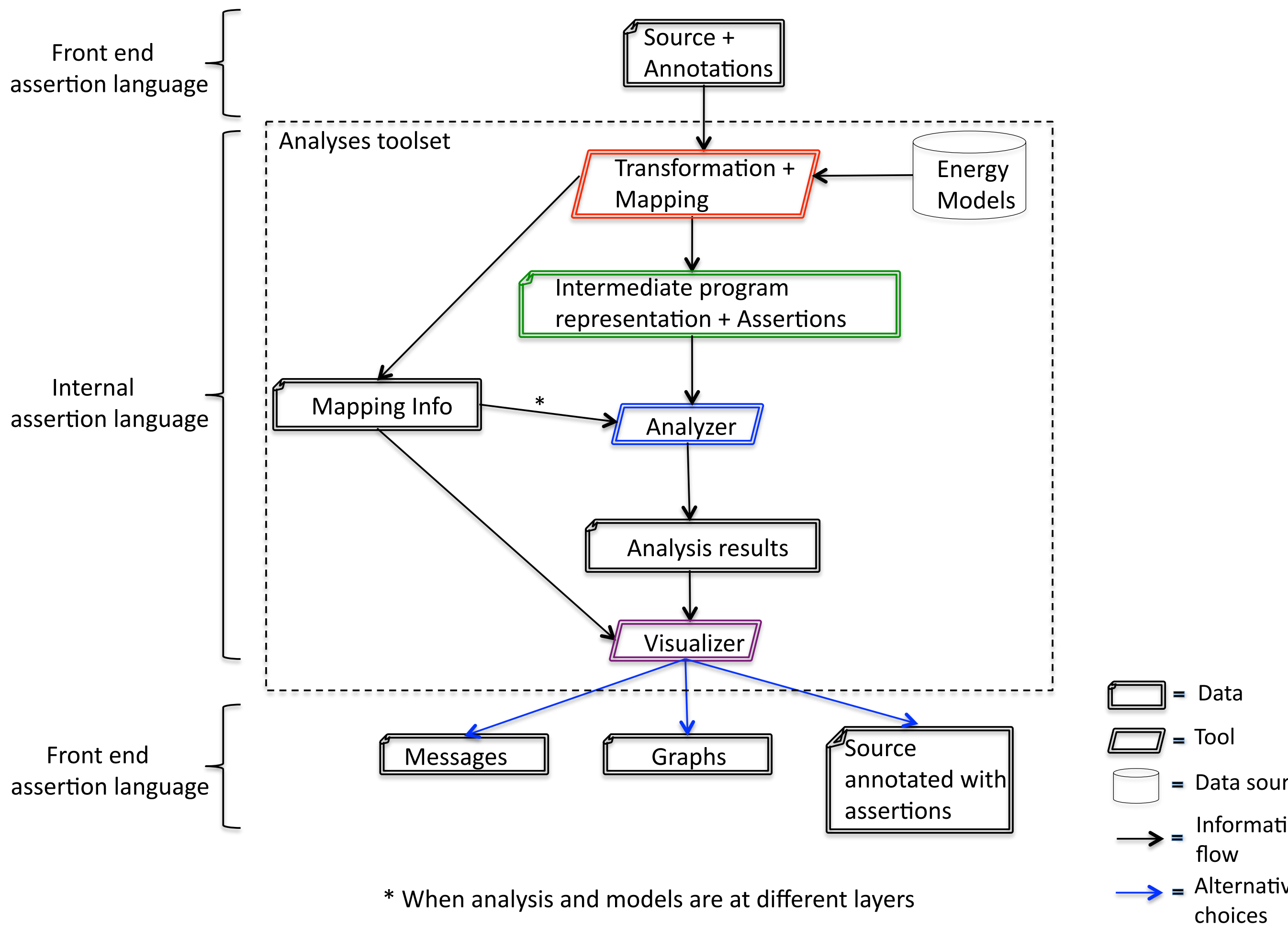
6. The framework should be capable of dealing with concurrent programs, and

7. It should be able to handle different languages in the same framework.

The last requirement (7), leads us to differentiate between the *source language* and the language actually analysed, which we call *intermediate semantic program representation language* (*IR* from now on). As a proof of concept to be developed in the project, we focus on the analysis of programs written in the XC source language. However, our framework is general enough to easily allow the analysis of other source languages as well. Our approach to allow the analysis of different source codes is to perform a transformation from each source code into the IR. Then, the analyser deals with the IR always in the same way, independently of where it comes from.

We have decided to use (Constraint) Horn Clauses, or equivalently, (Constraint) Logic Programs (CLP from now on), as the IR of a given source code. The main reason is that CLP offers a good number of features that make it very convenient as the IR for analysis. For instance, CLP programs are in Static Single Assignment (SSA) form, as it will be explained later. Moreover, there are available tools (such as CiaoPP [HPBLG05]) that use a CLP intermediate representation for analysis and that we can use and extend. In fact, currently, there is a trend to use CLP as intermediate representation in analysis and verification tools. The CLP representation is described in Section 2.

Figure 1 shows a high-level view of the general analysis framework that we propose, illustrating the ideas described above. The process starts with a source program that may contain assertions (expressed in the *front end* part of the common assertion language, and used to provide useful hints and information to the analyser, as described in deliverable D2.1), from which the *Transformation* tool (red box) generates its associated IR (green box). We assume that the *Transformation* box includes the compilation process. Thus, the idea is that the transformation box takes the source code as input and produces the program to be analysed. The transformation process is also in charge of translating the (front end) assertions (or annotations) present in the source code into assertions written in the internal assertion language (IAL) described in deliverable D2.1 already mentioned. Moreover, the transformation process takes an *energy model* as input, and translates it into assertions expressed in the IAL. The role of the energy models is to express the effect, in terms of energy consumption or other resources that depend on the underlying hardware, of the execution of a software segment (e.g., an assembly instruction) on the hardware. Such information is required by the analyser, which propagates it during the static analysis of a program (expressed in the IR) in order to infer information (the analysis results) for higher-level entities such as functions or procedures in the program. This is also illustrated in Figure 1, where the *Analyser* (blue box) takes the IR, together with the IAL assertions, expressing the energy models and possibly useful (trusted) information, and processes them, producing the analysis results, expressed also in the IAL. The analysis results include energy consumption (or, in general, resource usage) information expressed as functions on data sizes for the whole program and parts of it, such as procedures and functions, as mentioned before. Such results, are then processed by a *Visualiser* (pink box) which is in charge of showing the information to the users (system developers) in an appropriate format, in order to help them to better understand the effect of their designs on the energy consumption early on during the software development process, and make more informed design decisions (e.g., using the appropriate data structures), even when there are parts not developed yet.



Front end assertion language
Source + Annotations
Analyses toolset
Transformation + Mapping
Energy Models
Intermediate program representation + Assertions
Internal assertion language
Mapping Info
*
Analyzer
Analysis results
Visualizer
Front end assertion language
Messages
Graphs
Source annotated with assertions
= Data
= Tool
= Data sour
= Informati flow
= Alternativ choices
* When analysis and models are at different layers

Figure 1: High-level view of the general analysis framework architecture.

There can be different ways of showing such information to the user, for example, by annotating the source code with assertions expressing energy consumption functions for the procedures and functions in it; by colouring the source code according to how much energy it consumes (e.g., red for the energy-expensive code and green for the "cool" parts); by means of messages;

or by using data structures such as graphs. These alternative choices are represented in Figure 1 by using blue arrows.

The *Visualiser* also takes as input the *mapping information*, which includes the information needed for mapping the analysis results back to the procedures and functions in the source code. Such *mapping information* is produced by the transformation process (red box).

We have explored different realizations of the general analysis framework shown in Figure 1 by choosing different combinations of realizations for the *Analyser* and *Transformation*. These will be explained in detail later.

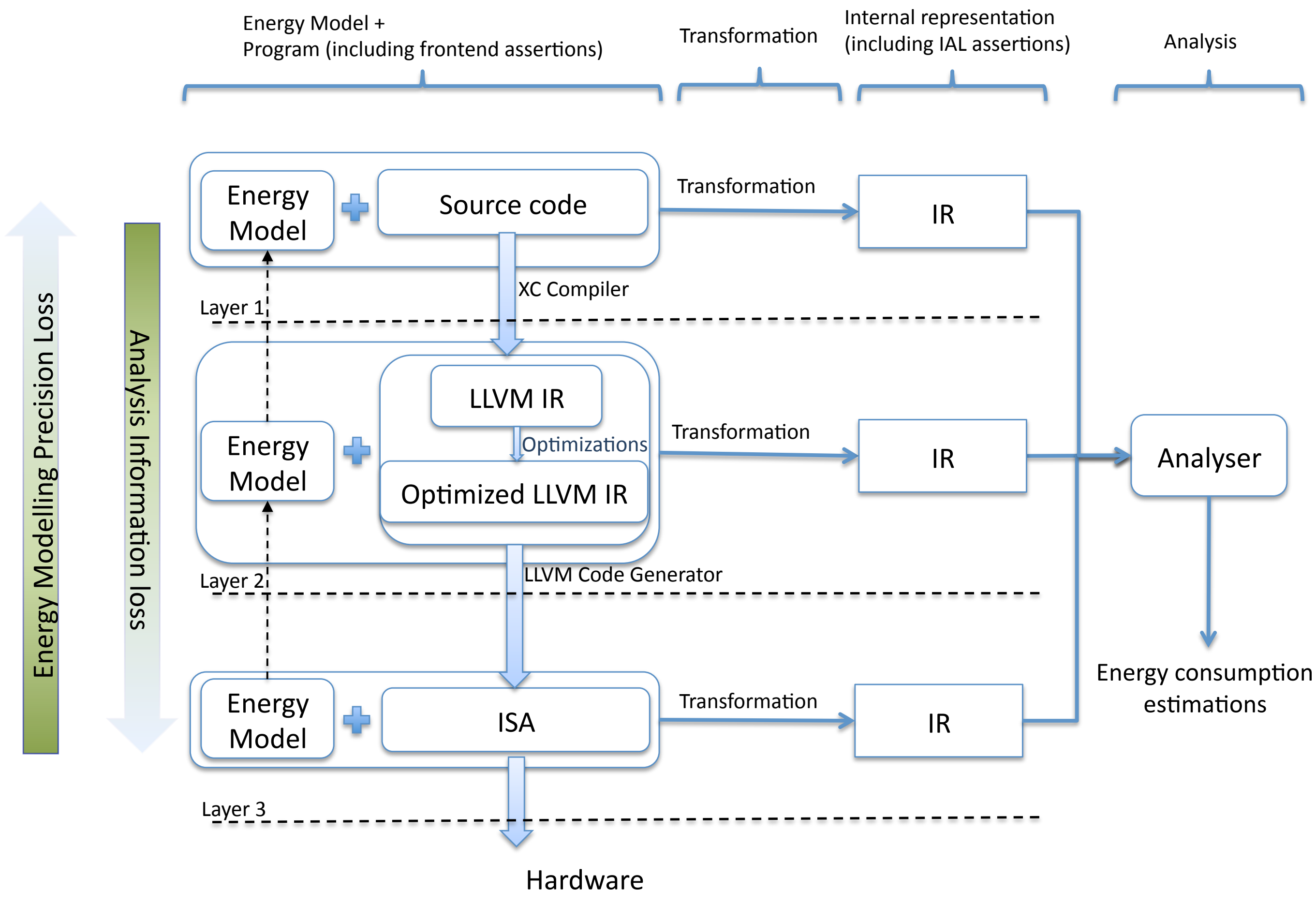


Figure 2: Overview of the analysis/modeling layer trade-off.

Requirement 1, supporting energy transparency through the system layers, implies that energy consumption at hardware layer should be immediately visible at the layer at which software

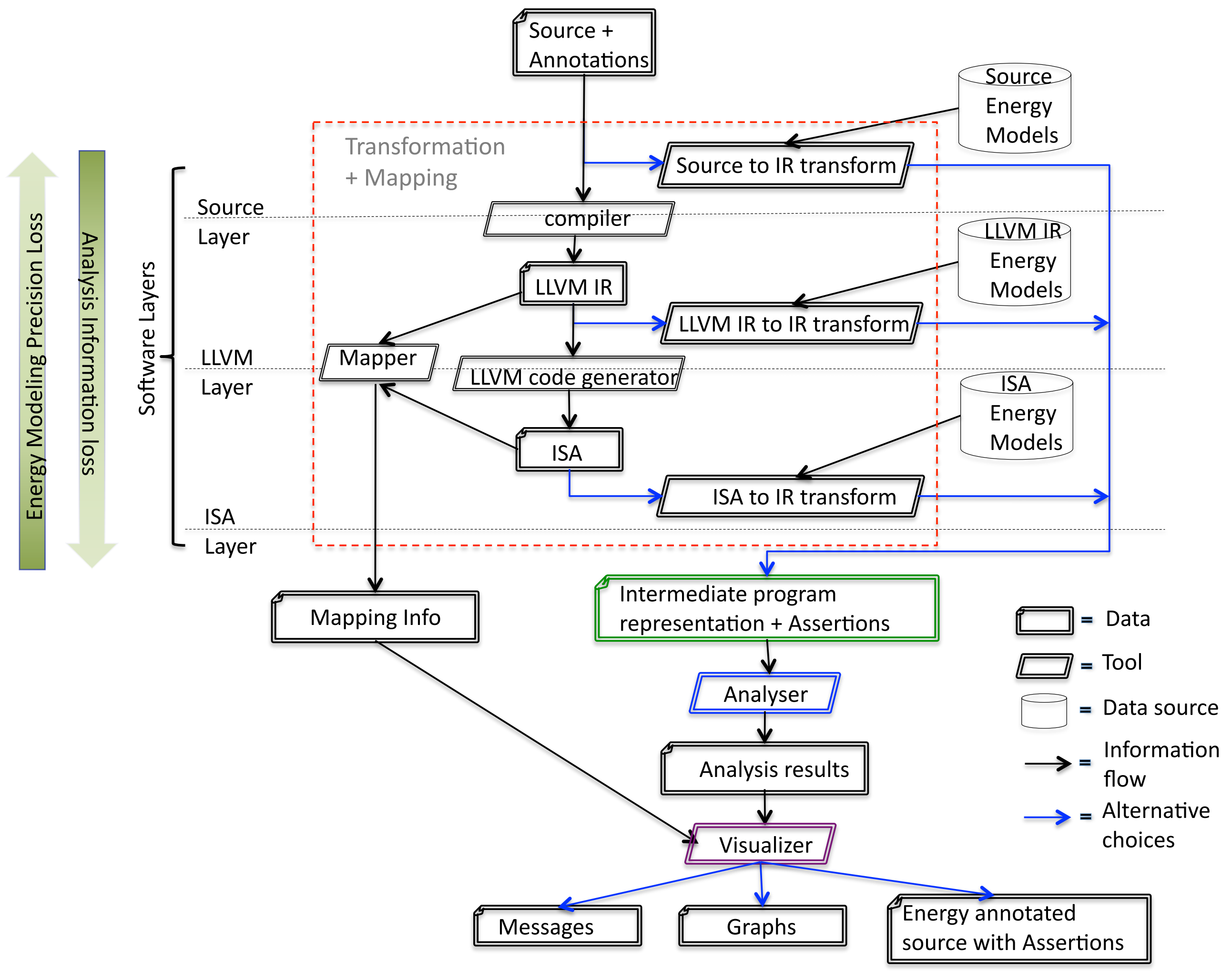


Figure 3: Detailed view of the analysis/modeling layers within the analysis framework.

is designed or used (normally at source code layer). Since the execution of high-level source code on the hardware is achieved through layers of compilation or interpretation, our approach to meet the energy transparency requirement is to perform an analysis in combination with an energy model. In our initial study, we have considered three software layers: XC source code, LLVM IR [LA04], and Instruction Set Architecture (ISA, or just assembly). Since our goal is to investigate all possibilities, we assume that the analysis can be performed, and the models defined at any one of these layers (with the obvious constraint that the analysis should be performed at the same or at another upper layer the model is defined). Performing the analysis at a given layer means that the analyser "mimics" the execution of the program (or an internal representation of it) at such layer according to the semantics of interest, inferring energy information for such layer. The energy model determines the bottom layer at which the analysis will propagate energy consumption information up to the layer at which the analysis is performed, and thus provides basic information that the analysis will just trust.

Figures 2 and 3 illustrate the idea that the analysis can be performed at the three layers mentioned before. Figure 3 provides a more detailed view of the analysis/modeling layers within the analysis framework shown in Figure 1. For simplicity, Figure 3 shows the models defined at the same layer the analysis is performed. At any of these layers, a transformation into the IR is carried out, and the IR is then passed to the analyser. These alternative choices for the "Transformation box" are represented in the figure by using blue arrows. Note that in order to map the analysis information inferred at one layer to an upper layer (typically to the source code layer) the (program dependent) *mapping information*, produced also by the transformation process, is used. However, this mapping process does not perform any analysis in principle. Such mapping information is different from the (program dependent) mapping information that can be used by the analyser to propagate the energy model information defined at one layer up to the (different) layer at which the analysis is performed.

Our research on the ideas described above led us to face an interesting problem: deciding at which layer the analysis should be performed and the model defined. Basically, the problem arises because going down through the mentioned layers has opposite effects on the accuracy of models and on the precision of the analysis: energy models at lower layers (e.g., at the ISA layer) are more precise than at higher layers (e.g., XC source code), since the closer to the hardware, the easier it is to determine the effect of the execution of the program on the hardware. However, the program structure is lost and types are erased at lower layer representations, resulting in an accuracy loss in the analysis. This analysis/modeling trade-off is also illustrated in Figures 2 and 3. The possible choices are classified into two groups: those that analyse and model at the same layer, and the ones that analyse and model at different layers. For the latter, the problem we face is that we need to find good mappings between software segments from the layer the model is defined up to the layer the analysis is performed, in order not to lose accuracy in the energy information. A good compromise could be performing the analysis and defining the energy model at the LLVM IR layer. However, our plan is to explore all choices.

We first have explored the choice of both analysing and modelling at the ISA layer, and published our results in [LKS$^+$13], which is included in this document as attachment D3.1.1. In such work, we have realized the general analysis framework in Figure 1 by using the CiaoPP analyser, which uses a CLP based intermediate representation (Ciao IR), and performing a transformation that produces a CLP program representing the ISA program associated to the XC source program. This is illustrated in Figure 4, as a realization of Figure 1.

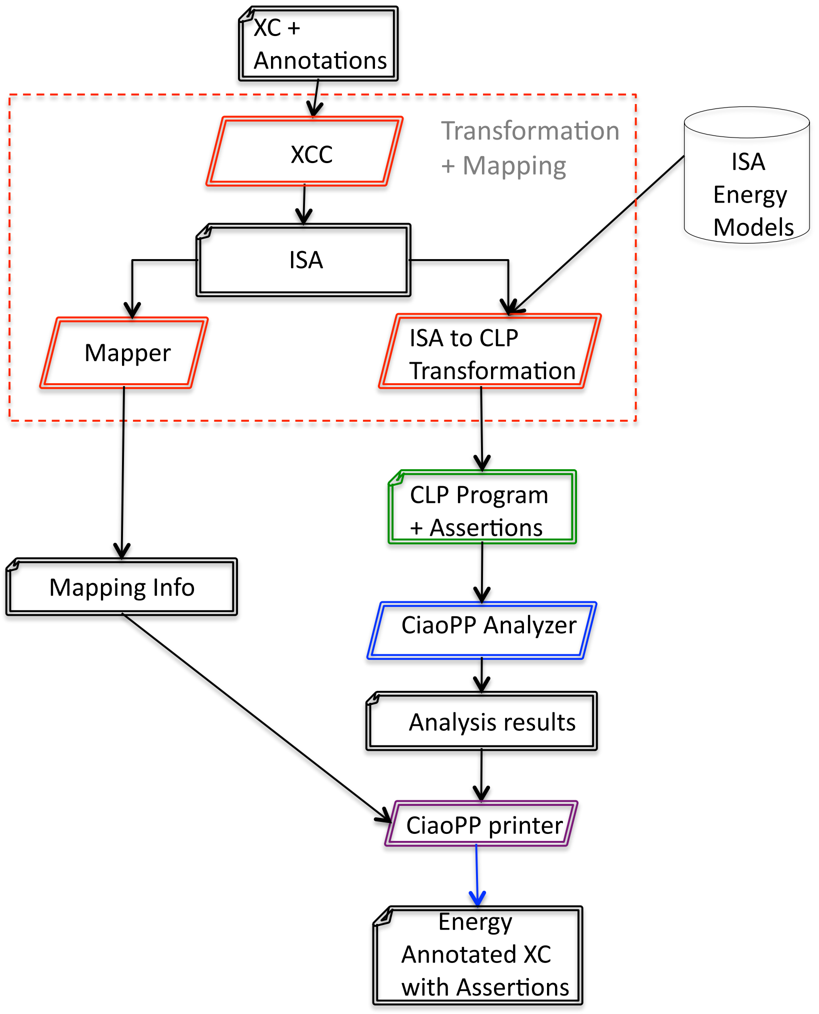


Figure 4: A realization of the general analysis framework using models and performing the analysis at the ISA layer.

To give another example, a different realization of the general analysis framework in Figure 1 that we have explored is presented in Figure 5. This and other realizations will be described in detail later as well.

In rest of this document, Section 2 explains fundamental concepts and results that our framework is based on, such as semantics and intermediate program representation. Section 3 describes different approaches to produce the program for the analysis, which will be in the IR form, i.e., different realizations for the "transformation" box in Figure 1. Then, Section 6 describes different choices for the "analyser" box in Figure 1, including the CiaoPP analyser, and

Section 7 comments on possible realizations for the "visualiser" box in Figure 1.

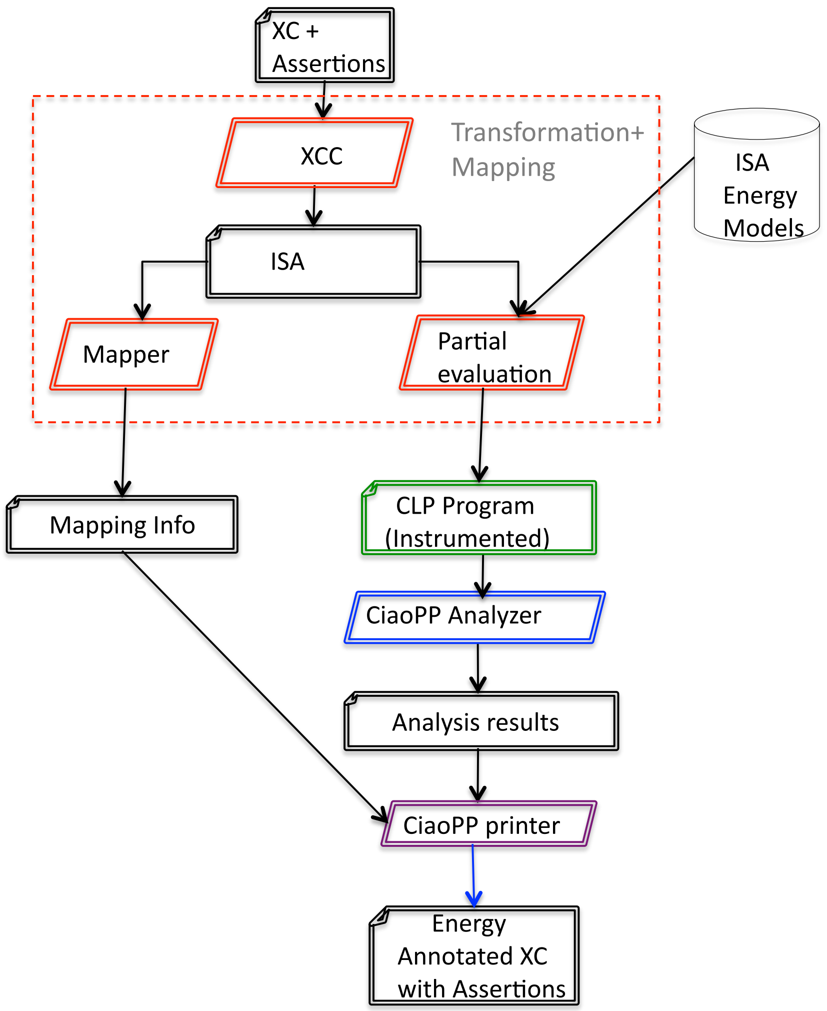


Figure 5: Realization of the general analysis framework based on partial evaluation.

Regarding requirements 5 and 6, Section 8 is devoted to the investigation of the foundations of probabilistic resource usage analysis, and Section 9 to the investigation of the formal basis of the analysis of concurrent programs.

Finally, in relation to requirement 4, the properties that the analysis should infer depend on:

- what properties are interesting to the software engineer to talk about in program specifications, and
- what properties are needed to perform the optimizations of WP4.

For this reason, we have performed an initial study of power-aware software optimization techniques to identify the most promising techniques, and the properties that the analysis should supply to them. The properties used by the known energy optimization techniques are summarized in Section 10.

2 Essentials of the Analysis Framework

This section gives a summary of essentials of the analysis framework, such as the *Intermediate Semantic Program Representation for Analysis* (*IR*) and translations from source languages to *IR*.

Intermediate Semantic Program Representation for Analysis (IR). As already said, the key to handle different languages in the same framework is to use an *Intermediate Semantic Program Representation for Analysis*, *IR*, and perform a transformation from each source code into the IR. This representation consists of a sequence of *blocks*. Each block is represented as a *Horn clause*:

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

which has an entry point, that we call the *head* of the block (to the left of the $:-$ symbol), including a number of parameters $< \ params \ >$, and a sequence of steps (the *body*, to the right of the $:-$ symbol), each of which is either, (the representation of) an ISA or LLVM IR *instruction* (depending on the layer at which the program is analysed), or a *call* to another (or the same) block. Be it LLVM IR, ISA, or any other program representation, the analyser deals with the IR always in the same way, independently of its origin.

Translations into IR. We have explored different approaches to produce the program for the analysis, which will be in the IR form. An approach is to perform a direct transformation into IR, which is described in detail in Section 3.1, in particular, the transformation from (XS1) ISA and LLVM IR programs into IR are described in Sections 3.1.1 and 3.1.2 respectively. The resulting programs are analysed with the CiaoPP tool.

¡¡¡¡¡¡¡ .mine Another approach consist on producing the IR by applying partial ======= Another approach consists of producing the IR by applying partial ¿¿¿¿¿¿¿ .r925 evaluation techniques to instrumented interpreters that directly implement the semantics of the language to be analysed. Information relevant to energy usage can be encoded in the semantics and thus embedded in the resulting IR (CLP) programs. CLP static analysis tools, such as CiaoPP, can then be applied to the CLP programs to derive assertions relating energy usage to other program variables. This is described in detail in Section 3.2.

Analysers. In order to analyse the IR representing programs at the XC, ISA, or LLVM IR layers, we have experimented with the CiaoPP analyser, which uses the CLP based intermediate representation that we call Ciao IR. In addition, we have improved and extended CiaoPP to integrate it in different realizations of the general analysis framework. This is described in detail in Section 6.1. For the particular case of analysing at the LLVM IR layer, we have also experimented and proposed a tool for the direct analysis of LLVM IR, which is described in Section 6.2.

Mapping Information. Mapping information also plays an important role in our analysis framework. There are two types of mapping information. The first one is the information needed for mapping the analysis results back to the procedures and functions in the source code, and showing such results to the user in the right place. Such information is a mapping between the different intermediate code representations, ISA and the source code. The second type of mapping information is useful when performing the analysis at one layer using energy models defined at a lower layer. Such mapping is used to propagate the energy model information defined at one layer up to the layer at which the analysis is performed. We have developed a tool that produces both types of mapping information, which is described in detail in Section 5.

3 Producing the Program for Analysis

In this section we describe the different approaches that we have explored to generate the IR to be analysed, i.e., different realizations for the "transformation" box in Figure 1. In particular, Sections 3.1 and 3.2 explain different approaches to produce the IR for analysis of LLVM IR and ISA programs respectively.

3.1 Producing Ciao IR (CLP) by Direct Transformation

In this section we describe the transformations from programs at both the ISA and LLVM layers into a CLP program written in the Ciao intermediate representation (Ciao IR). Such representation has already been described in Section 2. The transformation ensures that the program information relevant to resource usage is preserved, so that the energy consumption functions of the Ciao IR programs inferred by the resource analysis are applicable to the original programs.

3.1.1 ISA to Ciao IR Transformation

In this section we comment on the transformation used in the realization of the general analysis framework depicted in Figure 4. ISA programs are expressed using the XS1 instruction set [May13]. The transformation framework currently works on a subset of this instruction set. The ISA program is parsed and a control flow analysis is carried out, yielding an inter-procedural control flow graph (CFG). This process starts by identifying control transfer instructions such as branch or call instructions. Basic blocks are then constructed, for which the input/output arguments are inferred. These blocks are transformed into Static Single Assignment (SSA) form, which, finally yields the target Ciao IR (i.e., Horn clauses). The details of this transformation can be found in [LKS$^+$13] (attachment D3.1.1).

3.1.2 LLVM to Ciao IR Transformation

In this section we comment about the LLVM to Ciao IR transformation that will be used in the realization of the general analysis framework depicted in Figure 6. The way to generate the programs at the LLVM IR layer will be described in Section 4.1. 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 instructions that are guaranteed to be executed in the same sequence.

Each block either ends in a branching instruction or return. In order to represent each of the basic blocks of the LLVM IR in the Ciao IR, we follow a similar approach as in the case of the the ISA transformation:

1. Represent each LLVM instruction as a literal.
2. Infer input/output parameters to each block, removing the need for ϕ (phi) nodes.
3. Transform LLVM types into Ciao regular types.
4. Resolve branching to the predicate with multiple clauses, where each clause denotes one of the blocks the branch may jump to.

Inferring Block Arguments. As described before a *block* in the Ciao IR has an entry point called head of the block with input/output parameters, and a body containing a sequence of instructions (LLVM instructions in this case). Since the scope of the variables in LLVM blocks is at the function level, the blocks are not required to pass parameters while making jumps to other blocks. In order to represent LLVM blocks as Ciao IR blocks, we need to infer input/output parameters to each block.

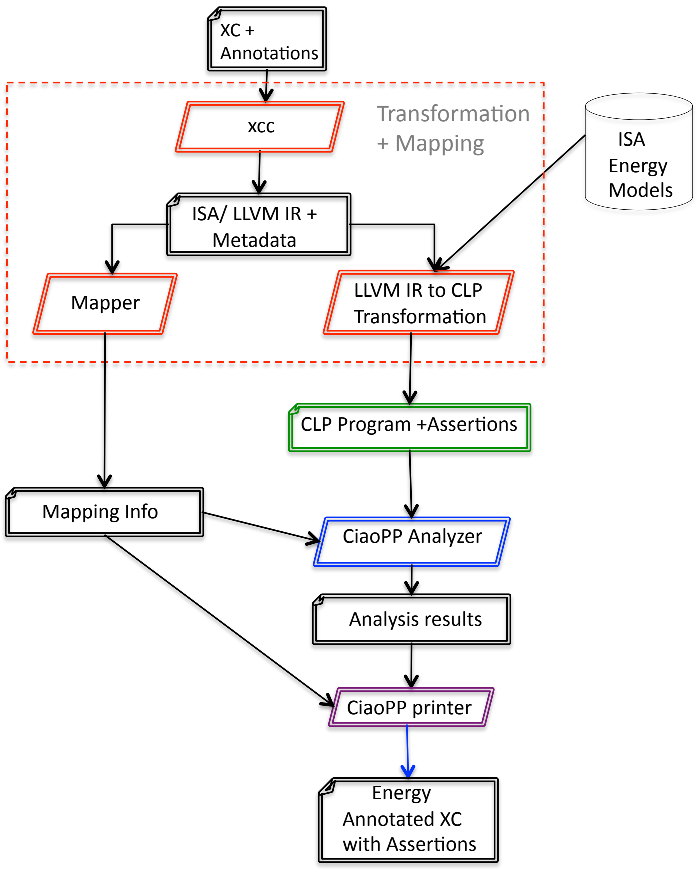


Figure 6: A realization of the general analysis framework performing the analysis at the LLVM IR layer and using ISA models.

The entry block in the LLVM IR is always *alloca* where the program variables are allocated. The input/output arguments to the corresponding Ciao 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 the 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:

$$kill(b) = \bigcup_{k=1}^{n} def(k)$$

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

and $def(k)$ 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 to SSA form. The 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 on 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.

Translating LLVM Types into Ciao Regular Types. The LLVM type system defines primitive and derived types. The primitive types are the fundamental building blocks of the LLVM type system. The primitive types include *label, void, integer, character, floating point, x86mmx, 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, these include *array, function, pointer, structure, vector, opaque*. Since XC does not support pointers nor floating point data types, the LLVM IR generated from XC program uses only a subset of the LLVM types.

Translating primitive types to regular types is straightforward. The *integer* and *character* types are represented as *num*, whereas the *label*, *void* and *metadata* types are represented as atoms. However, LLVM allows to define varying bit-number *integer* types which are translated to a *num* whose number of bits is not bounded.

Derived types are translated to compound terms. The array, vector, and structure types are represented as follows:

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

Both the *array* and *vector* types are represented by the *list* type in Ciao which is a special case of compound term. The elements of the list are again one of the primitive or derived types. The *structure* type is represented by a compound term which is composed of an atom called *functor* and a number of *arguments*, which are again either primitive or derived types. LLVM also introduces pointer types in the intermediate representation even though the front end language (e.g. XC) does not use one. This is usually in the pass-by-reference arguments, memory allocations in *allocas* block, and memory load and store operations. The types of these pointer variables in the Ciao IR are treated the same as the types these pointers point to. Consider the example below written in XC:

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

```
void print(struct mystruct s[7])
{
  ...
}
```

The type of argument s of the function $print$ 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 $print$ is:

define void @print($[7 \times \{i32, [5 \times i32]\}]*$ *noalias nocapture) nounwind*

The array type $[7 \times \{i32, [5 \times i32]\}$ is read as an array of 7 elements of $\{i32, [5 \times i32]\}$ structure type which is composed of a $i32$ integer type and a $[5 \times i32]$ array type. The array type $[5 \times i32]$ is of 5 elements of $i32$ integer type. This type is represented in the Ciao IR as follows using the $regtype/1$ construct of Ciao.

```
:- regtype array1/1.
array1([]).
array1([Elem|T]):-
        struct1(Elem),
        array1(T).
```

```
:- regtype struct1/1.

struct1('mystruct1'(X,Y)):-
        num(X),
        array2(Y).
```

```
:- regtype array2/1.
array2([]).
array2([Elem|T]):-
        num(Elem),
        array2(T).
```

$array1$ type is a list of $struct1$ elements. Each $struct1$ type element is represented as a functor $mystruct/2$ where the first argument is a num and the second is another list type $array2$. $array2$ is defined to be a list of num.

The implementation will be an LLVM pass in the LLVM Pass Framework (LPF) which is an important part of LLVM system. LPF allows plugging user defined transformation passes over the LLVM IR. It allows us to re-use existing analysis and transformation passes over the LLVM IR to aid our transformation.

We plan to use the LLVM IR to Ciao IR transformation described here for a realization of the general analysis framework that performs the analysis at the LLVM IR layer (using the CiaoPP tool) and uses the available models at the ISA level, together with the mapping information produced by the mapping tool described in Section 5.3. Such a realization is illustrated in Figure 6. The next step will be to use standalone models at the LLVM IR level.

3.2 Producing CLP by Partial Evaluation of Instrumented Interpreters

3.2.1 Introduction

In this section we present a method for the translation of a variety of source programs into the internal constraint logic program (CLP) representation. It has been implemented in the project for substantial subsets of the XC language [Wat09] and the assembly language for XCore [May13]. The method is based on specialising a CLP interpreter that directly implements the semantics of the language. Information relevant to energy usage can be encoded in the semantics and thus embedded in the resulting CLP programs. CLP static analysis tools can then be applied to the CLP programs to derive assertions relating energy usage to other program variables.

Section 3.2.2 describes the main aspects of operational semantics. Most features of imperative languages, including features for parallel execution, can be described using small-step operational semantics. Section 3.2.4 explains how this interpreter can be specialised with respect to a given source program abstract syntax tree, yielding a CLP program whose structure and behaviour mirror those of the original program. Section 6.3 outlines methods and tools for analysis of CLP programs and the form of the results of analysis.

In Section 9.1, an operational semantics for XC is presented, which provides the basis for translating multi-threaded XC programs into CLP using the techniques to be described in this section.

3.2.2 Operational Semantics

The main features of typical small-step operational semantics are the following:

- A *configuration* is a pair $\langle S, \sigma \rangle$ where S is a statement in the abstract syntax and σ is a store. A store σ is a finite mapping from a set of variable names $\mathcal{N}$ to a set of values $\mathcal{V}$, written as $\{x_1{=}V_1, \cdots, x_n{=}V_n\}$.

- A *transition* has the general form $\langle S_1, \sigma_1 \rangle \xrightarrow{\mathcal{L}} \langle S_2, \sigma_2 \rangle$, where $\mathcal{L}$ is a set of labels (for example indicating which communications to the external environment of S_1 are being made by the transition). A transition represents one computation step. In the semantics for parallel threads, one transition encodes one parallel step in which several threads may progress simultaneously and the label might contain several communications.

- A *computation* is a finite sequence of zero or more *synchronized* transitions, which are those in which the labelling $\mathcal{L}$ is empty. Note that it is also possible to represent asynchronous communication with a small-step semantics, by explicitly modelling buffers or similar constructs in the configurations.

- *Transition rules* are of the form:

$$\frac{\phi_1 \quad \cdots \quad \phi_m}{\langle S_1, \sigma_1 \rangle \xrightarrow{\mathcal{L}} \langle S_2, \sigma_2 \rangle}$$

  where $\phi_1 \cdots \phi_m$ ($m \geq 0$) are the premises of the rule, which involve transitions on constituent parts of S_1 as well as subsidiary functions, and $\langle S_1, \sigma_1 \rangle \xrightarrow{\mathcal{L}} \langle S_2, \sigma_2 \rangle$ is the small-step transition that is established if the premises hold. There is at least one transition rule for each statement type except possibly the skip or terminal statement; that is, each non-skip statement in a program matches S_1 in at least one transition rule.

- *Computations* (multi-step transitions) are defined by the following two rules:

$$\frac{}{\langle \mathsf{skip}, \sigma \rangle \longrightarrow^* \sigma} \tag{1}$$

$$\frac{\langle S_0, \sigma_0 \rangle \xrightarrow{\epsilon} \langle S_1, \sigma_1 \rangle \quad \langle S_1, \sigma_1 \rangle \longrightarrow^* \sigma_2}{\langle S_0, \sigma_0 \rangle \longrightarrow^* \sigma_2} \tag{2}$$

  A computation with initial configuration $\langle S_1, \sigma_1 \rangle$ and final configuration $\langle S_2, \sigma_2 \rangle$ is represented as $\langle S_1, \sigma_1 \rangle \longrightarrow^* \sigma_2$. S_2 is some terminal statement.

The transition semantics can also be used to define the reachable states of a computation.

3.2.3 Semantics-Based Interpreter

A CLP program is obtained systematically from the operational semantics.

- Data structures are chosen to express the basic components of the semantics.
  - A store is represented as a list of pairs; that is, $\{x_1{=}V_1, \cdots, x_n{=}V_n\}$ is represented as $[(x_1, V_1), \ldots, (x_n, V_n)]$. Operations such as lookup and update of the store are implemented straightforwardly using CLP clauses. For example the *find* relation is defined such that *find*$(St0, X, V, St1)$ is true iff X represents the variable x, $St0$ represents the store $\sigma{\oplus}\{x{=}V\}$ (where $\oplus$ is the disjoint union of stores) and $St1$ represents the store σ. Its definition is given by the following clauses:

$$\begin{array}{l} \textit{find}([(X,N)|St], X, N, St). \\ \textit{find}([(Y,M)|St], X, N, [(Y,M)|St1]) \text{ :-} \\ \quad X \backslash{==}\ Y, \textit{find}(St, X, N, St1). \end{array}$$

  - Suitable functors are chosen to represent the abstract syntax constructs, so that an AST is a CLP term. Values are encoded along with their type. For instance the statement $\mathsf{while}\,(x)\ \mathsf{send}\, c\,(y+1)$ could be represented as the term:

$$\mathit{while}(\mathit{var}(x), \mathit{send}(\mathit{const}(\mathit{chan}(c)), \mathit{var}(y) + \mathit{const}(\mathit{int}(1))))$$

  - Suitable terms are chosen to represent the transition labels.

- A single-step transition $\langle S_1, \sigma_1\rangle \xrightarrow{\mathcal{L}} \langle S_2, \sigma_2\rangle$ is represented by a predicate $exec(S1, St1, L, S2, St2)$ and a multi-step transition $\langle S_1, \sigma_1\rangle \longrightarrow^* \sigma_2$ is represented by a predicate $run(S1, St1, St2)$ (where the predicate arguments are the CLP translations of the respective semantic expressions).

- Each transition rule of the form:

$$\frac{\phi_1 \quad \cdots \quad \phi_m}{\phi_0}$$

  is translated to a CLP clause of the form:

$$A_0 \text{ :-} A_1, \ldots, A_n.$$

  where A_i is the translation of the premise or conclusion ϕ_i $(1 \leq i \leq n)$ or ϕ_0 respectively.

If P is a CLP program and A is a closed formula, $P \models A$ states that A is a logical consequence of P, assuming that constraint predicates and functions are given their standard interpretations.

Given a source program C, let I_C be the CLP program representing the operational semantics (possibly together with clauses representing the global definitions comprising C). Let S_0 be an initial statement for C (such as a call to the main function) and σ_0 be some initial store containing values for (at least) the free variables of S_0; suppose $S0$ is the CLP representation of S_0, $St0$ is the representation of σ_0 and $St1$ represents some other store σ_1. Then the following proposition expresses the correctness of the translation.

Proposition. $\langle S_0, \sigma_0 \rangle \longrightarrow^* \sigma_1$ iff $I_C \models run(S0, St0, St1)$

The program I_C can be run as an interpreter for program C using a CLP system in the following way. Given an initial statement $S0$ and an initial store $St0$ the CLP query

$$?\!-\ run(S0, St0, Stn)$$

computes the final store Stn, which will contain the final values assigned to the free variables in $S0$.

However the main goal of translating the semantics into CLP is not to run source programs, but rather to exploit static analysis tools for CLP. This will be discussed in Section 6.3.

3.2.4 Partial Evaluation of the Semantics-Based Interpreter

Let I_C be the CLP program resulting from the operational semantics together with source program C and let S_0 be an initial statement. If the initial state $St0$ is unknown then the query $run(S0, St0, St1)$ potentially has many (perhaps an infinite number of) solutions. However we can *partially evaluate* I_C with respect to $run(S0, St0, St1)$ with unknown $St0$, obtaining a *specialised* interpreter.

Partial evaluation is a program transformation that specialises a program with respect to a partially known input or goal. In CLP, the key property of partial evaluation is the following.

Proposition. *Let P be a CLP program and let P_G be a partial evaluation of P with respect to a goal G. Then for all instances $G\theta$ of G, $P \models G\theta$* **iff** *$P_G \models G\theta$.*

In other words P_G gives exactly the same results for (instances of) goal G as P does. Of course, we expect P_G to be more efficient than P when computing the solutions of G.

The idea of partial evaluating I_C with respect to $run(S0, St0, St1)$ is to perform as much as possible of the computation of the goal $run(S0, St0, St1)$ that is determined by the known argument $S0$. We call the specialised program C_I since it can be seen as a translation of the source program C into CLP, via the interpreter I_C. There is a well known connection between partial evaluation of an interpreter I with respect to a source program C and the *compilation* of C into the language in which I is written. For more details see Jones *et al.* [JGS93].

Partial evaluation is not needed, strictly speaking, since it would in principle be possible to analyse the program I_C directly. In practice direct analysis of I_C is very hard and would require specialised CLP analysis tools, whereas analysis of the partially evaluated programs can be done by standard CLP analysis tools.

General Form of the Specialised Program. Many possible specialisations are possible (including returning the program unchanged), depending on the strategy in the partial evaluator for deciding which parts of the computation can be executed and which cannot.

The key requirement for a useful specialisation of I_C is that sufficient computation is performed to discover the control flow of the program C. When evaluating the small-step semantics of a program with initial statement S_0 and initial store σ_0, a sequence of configurations $\langle S_0, \sigma_0\rangle, \langle S_1, \sigma_1\rangle, \langle S_2, \sigma_2\rangle, \ldots$ is encountered, such that $\langle S_i, \sigma_i\rangle \xrightarrow{\epsilon} \langle S_{i+i}, \sigma_{i+i}\rangle$ for all $i \geq 0$. The expectation of partial evaluation is that, given the initial statement S_0 but unknown initial store, it will be able to construct the possible sequences of the form $S_0, S_1, S_2, \ldots$, except possibly for some constants in each S_i; for instance S_i could contain an expression containing an unknown value V. Note that since the initial store is unknown, there could be many possible such sequences of statements even if the semantics is deterministic. The ideal specialisation of I_C should contain code capable of handling only those sequence which could arise from S_0 with *some* initial store.

The following clauses for the run predicate are derived from the transition rules whose conclusion is of the form $\langle S_0, \sigma_0\rangle \longrightarrow^* \sigma_1$ shown above:

$$\begin{array}{l} run(skip, St, St). \\ run(S0, St0, St2) :- \\ \quad exec(S0, St0, L, S1, St1), \\ \quad emptyLabel(L), \\ \quad run(S1, St1, St2). \end{array}$$

Given a call $run(S0, St1, St2)$ in which the structure of $S0$ is known, except possibly for the values of constant expressions, only one of the two clauses for run can succeed. Furthermore, if it matches the second clause, then the call to $exec(S0, St0, L, S1, St1)$ can be completely unfolded apart for some operations on the store, say $C_1, \ldots, C_k$, and the call $emptyLabel(L)$ can also be unfolded. After such unfolding the value of $S1$ will be completely known, except possibly for the values of constants expressions within $S1$.

To summarise, the partial evaluation of a call $run(S0, St1, St2)$ in which the structure of $S0$ is known gives rise either to:

$$run(skip, St, St).$$

in the case that $S = \mathsf{skip}$, or to zero or more clauses of the form:

$$run(S0, St0, St2) \mathbin{:\!-} C_1, \ldots, C_k, \;\; run(S1, St1, St2).$$

in which $S1$ is known except possibly for the values of some constants in $S1$, and $C_1, \ldots, C_k$ are constraints or calls to simple predicates defined only using constraints. Zero clauses could be produced if, for example, there was no transition from $S0$ with an empty label, and thus the computation of $S0$ was blocked; more than one clause could be produced if $S0$ was a branching statement. The transition rules are defined so that for all (non-blocked) statements other than skip and branching statements, exactly one transition rule is applicable.

Partial evaluation can then continue with the goal $run(S1, St1, St2)$ producing further clause(s) of the same form. Leaving aside for a moment the issue of termination of partial evaluation, we obtain a set of clauses, such as the following:

$$\begin{array}{l}
run(S0, St0, St2) \mathbin{:\!-} C_{0,1}, \ldots, C_{0,k_0}, \;\; run(S1, St1, St2).\\
run(S1, St0, St2) \mathbin{:\!-} C_{1,1}, \ldots, C_{1,k_1}, \;\; run(S2, St1, St2).\\
run(S2, St0, St2) \mathbin{:\!-} C_{2,1}, \ldots, C_{2,k_2}, \;\; run(S3, St1, St2).\\
run(S2, St0, St2) \mathbin{:\!-} C_{3,1}, \ldots, C_{3,k_3}, \;\; run(S4, St1, St2).\\
run(S3, St0, St2) \mathbin{:\!-} C_{4,1}, \ldots, C_{4,k_4}, \;\; run(S5, St1, St2).\\
run(S4, St0, St2) \mathbin{:\!-} C_{5,1}, \ldots, C_{5,k_5}, \;\; run(S6, St1, St2).\\
\ldots\\
run(skip, St, St).
\end{array}$$

(Here $S2$ is an example of a branching statement, giving rise to two clauses for run).

Termination of Partial Evaluation. If we generate a clause such as the following:

$$run(Sj, St0, St2) \mathbin{:\!-} C_{j,1}, \ldots, C_{j,k_j}, \;\; run(Si, St1, St2).$$

where Si is a statement that has previously been encountered (modulo the values of some unknown constants in Si) then Si is not (re-)partially evaluated. The process thus terminates if the number of distinct statements generated is finite.

Termination of partial evaluation has to be established for each set of transition rules. For languages without recursive functions or procedures it is usually straightforward to show termination. For programs with recursion, a small modification of the transition rules for function calls can be made, which is sufficient to ensure partial evaluation. Essentially a big-step semantics rule is used for evaluating (recursive) function and procedure calls. The specialised clauses resulting from a recursive procedure call followed by some other statement Sk will have the form:

$$\begin{aligned} run((p(y)\,;Sk), St0, St3) \;\; :- \;& C_{j,1}, \ldots, C_{j,k_j}, \\ & run(p(y), St1, St2), \\ & run(Sk, St2, St3). \end{aligned}$$

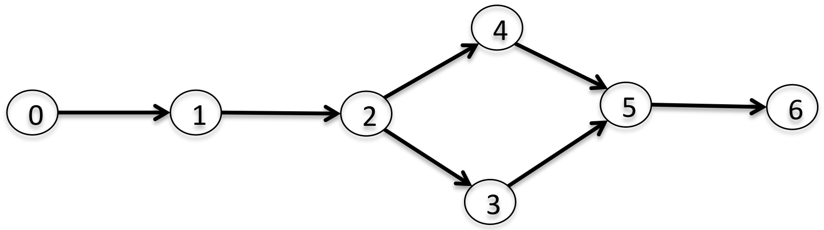
Renaming and Flattening of run Clauses. A standard CLP transformation can be used to simplify the partially evaluated program [Gal93] and make it more amenable to analysis. For each distinct call $run(Sj, St0, St2)$ generated during partial evaluation (including the initial goal) we define a new predicate

$$run_j(X_1, \ldots, X_n) :- run(Sj, St0, St2).$$

where $X_1, \ldots, X_n$ are the distinct variables in $run(Sj, St0, St2)$ and run_j is a fresh predicate name. By a standard fold-unfold transformation for CLP [PP99] the run predicates in the program can be replaced by the appropriate run_j predicates. The following program is an example of the resulting form.

$$\begin{array}{l} run_0(X_1, \ldots, X_{n_0}) :- C_{0,1}, \ldots, C_{0,k_0}, \;\; run_1(Y_0, \ldots, Y_{n_1}). \\ run_1(X_1, \ldots, X_{n_1}) :- C_{1,1}, \ldots, C_{1,k_1}, \;\; run_2(Y_0, \ldots, Y_{n_2}). \\ run_2(X_1, \ldots, X_{n_2}) :- C_{2,1}, \ldots, C_{2,k_2}, \;\; run_3(Y_0, \ldots, Y_{n_3}). \\ run_2(X_1, \ldots, X_{n_2}) :- C_{3,1}, \ldots, C_{3,k_3}, \;\; run_4(Y_0, \ldots, Y_{n_4}). \\ run_3(X_1, \ldots, X_{n_3}) :- C_{4,1}, \ldots, C_{4,k_4}, \;\; run_5(Y_0, \ldots, Y_{n_5}). \\ run_4(X_1, \ldots, X_{n_4}) :- C_{5,1}, \ldots, C_{5,k_5}, \;\; run_5(Y_0, \ldots, Y_{n_5}). \\ run_5(X_1, \ldots, X_{n_5}) :- C_{6,1}, \ldots, C_{6,k_6}, \;\; run_6(Y_0, \ldots, Y_{n_6}). \\ \ldots \\ run_z(X_0, \ldots, X_{n_z}). \end{array}$$

It captures a control flow graph of the following form.



Finally a clause defining the initial goal is added.

$$run(S0, St0, St2) :- run_0(X_1, \ldots, X_{n_0}).$$

where $X_1, \ldots, X_n$ are the free variables in $S0, St0, St2$.

The arguments of the run_j predicates range only over source program values, not over statements, stores, or other artifacts of the semantics.

Partial Evaluation with LOGEN**.** The online tool LOGEN [LJ99] provides a very convenient interface and efficient implementation of a partial evaluator suitable for handling the semantics-based interpreter. The program to be partially evaluated is first *annotated* to indicate which calls are to be unfolded, and which are kept in the specialised program. Furthermore LOGEN provides the possibility of defining *filters* that control the flattening and renaming process described above.

LOGEN can either generate the specialised program directly from the annotated interpreter and an initial goal, or else it can produce a so-called *generating extension* that is a specialised (much faster) version of the partial evaluator itself.

3.2.5 Instrumented Semantics for Resource Usage Analysis

The semantics can be instrumented; that is, extra information can be added to the configurations and transitions to capture non-functional aspects of the computation. Information added to configurations could include for example clocks, energy meters, memory meters and communication counts.

Here we consider two basic techniques for instrumenting semantics that can be used in analysis of energy usage.

- A global "energy counter" which records the total energy consumed, based on an energy model for individual statements.

- A vector of counters recording the number of times that each statement is executed. The values of these counters could be used in a variety of specific resource analyses such as time or energy usage analysis, and give information relative to specific program points rather than a global result.

The total energy cost of running a program was expressed by Tiwari *et al.* [TMWL96] using the following formula.

$$E_P = \Sigma_i(B_i \times N_i) + \Sigma_{i,j}(O_{i,j} \times N_{i,j}) + \Sigma_k E_k$$

Here N_i is the number of times instruction i is executed, and $N_{i,j}$ is the number of times that instruction i is followed immediately by instruction j.

The *energy model* for a programming language executed on a given hardware platform is given by the values B_i (the energy consumed when executing instruction i), the matrix of values $O_{i,j}$ (the transition energy of moving from instruction i to instruction j) and a set of values E_k representing assorted other energy overheads. Thus an analysis that yields the values N_i and $N_{i,j}$ for a given program execution can be combined with the energy model to yield the total energy. When combined with a timing analysis, energy usage analysis can be used to derive a power dissipation analysis (an analysis of the rate of use of energy).

As will be seen in Section 6.3, analysis procedures would normally return constraints on N_i and $N_{i,j}$, or express them as functions of some other variables, rather than precise values. Hence the total energy consumed by a program will also be expressed in terms of constraints or parametrised by other variables in the program's state.

Analysis of a Global Energy Counter. A global energy counter is considered as an extra component of configurations in multi-step transitions. It can be modelled as an extra variable e in the store, where e is a variable name not occurring elsewhere. That is, an instrumented configuration is $\langle S, \sigma \oplus \{e{=}V_e\} \rangle$, where V_e is the total amount of energy used so far in the computation. The initial store σ_0 is extended by initialising the energy counter, i.e. $\sigma_0 \oplus \{e{=}0\}$.

Each individual transition then updates the energy counter, according to the energy model for that statement. The label on transitions is extended to return the energy used.

Example: Consider the following transition for the statement $\mathsf{send}\, c\, E$ which models the behaviour of the XC statement that sends a value on a channel.

$$\frac{}{\langle \mathsf{send}\, \alpha\, V, \sigma \rangle \xrightarrow{\alpha!V} \langle \mathsf{skip}, \sigma \rangle} \tag{3}$$

This is modified to return the energy n_e consumed by this step, which could be dynamically computed depending on (for example) the size of the data or the destination of the channel. This is represented by a function $F_{\mathsf{send}}(\alpha, V)$ in the transition.

$$\frac{F_{\mathsf{send}}(\alpha, V) = n_e}{\langle \mathsf{send}\, \alpha\, V, \sigma \rangle \xrightarrow{(\alpha!V, n_e)} \langle \mathsf{skip}, \sigma \rangle} \tag{4}$$

The energy model for XC would be encoded in the function F_s for each statement type s. Note that the energy consumed in evaluating the expression to be sent and the channel name would be computed by other transitions.

The multi-step transitions are then modified to accumulate the individual transition energies into the energy counter, returning in the final state the total energy used, V_{tot}.

$$\frac{}{\langle \mathsf{skip}, \sigma \oplus \{e{=}V_{tot}\}\rangle \longrightarrow^{*} \sigma \oplus \{e{=}V_{tot}\}} \tag{5}$$

$$\frac{\langle S_0, \sigma_0 \rangle \overset{(\epsilon, n_e)}{\longrightarrow} \langle S_1, \sigma_1 \rangle \quad \langle S_1, \sigma_1 \oplus \{e{=}V_e + n_e\}\rangle \longrightarrow^{*} \sigma_2 \oplus \{e{=}V_{tot}\}}{\langle S_0, \sigma_1 \oplus \{e{=}V_e\}\rangle \longrightarrow^{*} \sigma_2 \oplus \{e{=}V_{tot}\}} \tag{6}$$

We will see in Section 3.2.6 how the energy counter will appear in the partially evaluated version of the instrumented semantics.

Analysis of Program Point Execution Frequency. We now describe how counters can be added to the semantics, and how they appear in the resulting CLP program. Let us suppose that we identify a number of program points whose execution we wish to analyse. (These could be chosen automatically, say the entry of each procedure and loop). Let these points be identified in the AST by labels $\{\mathfrak{l}_1, \ldots, \mathfrak{l}_k\}$. It is assumed that the AST representation is extended with such labels. E.g. we take as an example the following abstract syntax for XC programs.

$$\begin{aligned} \text{(Statements)} \quad S ::= {} & \mathsf{skip}^{\mathfrak{l}} \mid \mathsf{return}^{\mathfrak{l}}\, E \mid x :=^{\mathfrak{l}} E \mid S_1 \,;^{\mathfrak{l}}\, S_2 \mid S_1 \parallel^{\mathfrak{l}} S_2 \mid \\ & \mathsf{if}^{\mathfrak{l}}\,(n)\, S_1 \,\mathsf{else}\, S_2 \mid \mathsf{while}^{\mathfrak{l}}\,(n)\, S_1 \mid \mathsf{let}^{\mathfrak{l}}\, x = E \,\mathsf{in}\, S \\ & \mathsf{send}^{\mathfrak{l}}\, E_1\, E_2 \mid x := \mathsf{get}^{\mathfrak{l}}\, E \end{aligned}$$

A statement label can have a null value $\mathfrak{o}$, which means that the statement is not one of those whose execution is counted.

Recording Statement Counters in Transitions. We then extend transition rules so that each transition records (as an extra label on the transition) a set of statement labels, namely of those statements (with non-null statement labels) that have been encountered during the transition. The set could be empty if no non-null-labelled statement has been executed, or could contain several labels if several threads in a parallel statement have simultaneously been executed.

Example: Consider the following transition for the statement $\mathsf{while}\,(n)\, S$.

$$\frac{}{\langle \mathsf{while}\,(E)\, S, \sigma\rangle \overset{\epsilon}{\longrightarrow} \langle \mathsf{if}\,(e)\, S \,;\, \mathsf{while}\,(E)\, S \,\mathsf{else\,skip}, \sigma\rangle} \tag{7}$$

This is modified to handle labelled statements and record the label of the while statement on the transition arrow.

$$\frac{}{\langle \mathsf{while}^{\mathfrak{l}}\,(E)\, S, \sigma\rangle \overset{(\epsilon, \{\mathfrak{l}\})}{\longrightarrow} \langle \mathsf{if}^{\mathfrak{o}}\,(e)\, S ;^{\mathfrak{o}} \mathsf{while}^{\mathfrak{l}}\,(E)\, S \,\mathsf{else\,skip}^{\mathfrak{o}}, \sigma\rangle} \tag{8}$$

Note that the statements introduced by the transition itself are labelled with null ($\mathfrak{o}$) labels, whereas the label on the while statement is propagated, so that the next time that while statement is executed it will be recorded.

Configurations for Multi-step Transitions. Configurations for the multi-step transition relation $\longrightarrow^*$ are extended by adding k statement counters corresponding to the labels of the statements whose execution we wish to count. That is, an instrumented configuration is $\langle S, \sigma, \bar{\mathcal{V}} \rangle$, where $\bar{\mathcal{V}} = \{\mathfrak{l}_1 = v_1, \ldots, \mathfrak{l}_k = v_k\}$ representing the number of times that $\mathfrak{l}_1, \ldots, \mathfrak{l}_k$ respectively have been encountered so far. The initial configuration with initial statement S_0 and initial store σ_0 is $\langle S_0, \sigma_0, \bar{\mathcal{V}}_0 \rangle$ with $\bar{\mathcal{V}}_0 = \{\mathfrak{l}_1 = 0, \ldots, \mathfrak{l}_k = 0\}$.

Updating Statement Counters in Multi-step Transitions. The transitions for multi-step computations are modified to update the configuration by incrementing the counters for the labels returned by the small-step transitions. Let $\bar{\mathcal{V}}$ be a tuple of statement counters and $\mathcal{C}$ be a set of statement labels. Let the function incr be defined such that $\mathsf{incr}(\bar{\mathcal{V}} \oplus \{\mathfrak{l}{=}v\}, \mathfrak{l}) = \bar{\mathcal{V}} \oplus \{\mathfrak{l}{=}v+1\}$. Define $\bar{\mathcal{V}} \bigtriangleup \mathcal{C}$ to be $\{\mathsf{incr}(\bar{\mathcal{V}}, \mathfrak{l}) \mid \mathfrak{l} \in \mathcal{C}\}$.

The multi-step transitions are then modified as follows.

$$\frac{}{\langle \mathsf{skip}^{\mathfrak{l}}, (\sigma, \bar{\mathcal{V}}) \rangle \longrightarrow^* (\sigma, \bar{\mathcal{V}})} \tag{9}$$

$$\frac{\langle S_0, \sigma_0 \rangle \xrightarrow{(\epsilon, \mathcal{C})} \langle S_1, \sigma_1 \rangle \quad \langle S_1, (\sigma_1, \bar{\mathcal{V}} \bigtriangleup \mathcal{C}) \rangle \longrightarrow^* (\sigma_2, \bar{\mathcal{V}}')}{\langle S_0, (\sigma_0, \bar{\mathcal{V}}) \rangle \longrightarrow^* (\sigma_2, \bar{\mathcal{V}}')} \tag{10}$$

3.2.6 Partial Evaluation of the Instrumented Semantics

Semantics with Global Energy Counter. The result of partially evaluating the extended semantics with statement counters, after renaming and flattening the specialised predicates, is to add two arguments to the predicate run with a constraint in the clause bodies performing the incrementation of the energy counter with the transition energy (some constant n_e in the clauses below). The resulting specialised programs have the following form.

$$
\begin{array}{ll}
run_0(X_1,\ldots,X_{n_0}, & E_1, E_3) :- \\
 & C_{0,1},\ldots,C_{0,k_0}, \\
 & E_2 = E_1 + n_e, \\
 & run_1(Y_0,\ldots,Y_{n_1},E_2,E_3). \\
run_1(X_1,\ldots,X_{n_1}, & E_1, E_3) :- \\
 & C_{1,1},\ldots,C_{1,k_1}, \\
 & E_2 = E_1 + n_e, \\
 & run_2(Y_0,\ldots,Y_{n_2},E_2,E_3). \\
\ldots & \\
run_z(X_0,\ldots,X_{n_z}, & E, E).
\end{array}
$$

Finally a clause defining the initial goal is added, initialising the energy counter to zero.

$$run(S0, St0, St2, E) :- run_0(X_1,\ldots,X_{n_0},0,E).$$

Semantics with Statement Counters. Similarly, the result of partially evaluating the extended semantics with statement counters is to add $2k$ arguments to the predicate run, two variables for each of the k statement counters, with constraints in the clause bodies performing the incrementation of the appropriate counter(s).

$$
\begin{array}{ll}
run_0(X_1,\ldots,X_{n_0}, & V_1,\ldots,V_k,V_1'',\ldots,V_k'') :- \\
 & C_{0,1},\ldots,C_{0,k_0}, \\
 & V_1' = V_1,\ldots,V_j' = V_j + 1,\ldots,V_k' = V_k, \\
 & run_1(Y_0,\ldots,Y_{n_1},V_1',\ldots,V_k',V_1'',\ldots,V_k''). \\
run_1(X_1,\ldots,X_{n_1}, & V_1,\ldots,V_k,V_1'',\ldots,V_k'') :- \\
 & C_{1,1},\ldots,C_{1,k_1}, \\
 & V_1' = V_1,\ldots,V_m' = V_m + 1,\ldots,V_k' = V_k, \\
 & run_2(Y_0,\ldots,Y_{n_2},V_1',\ldots,V_k',V_1'',\ldots,V_k''). \\
\ldots & \\
run_z(X_0,\ldots,X_{n_z}, & V_1,\ldots,V_k,V_1,\ldots,V_k).
\end{array}
$$

Finally a clause defining the initial goal is added, initialising the counters to zero.

$$run(S0, St0, St2, [V_0,\ldots,V_k]) :- run_0(X_1,\ldots,X_{n_0},0,\ldots,0,V_0,\ldots,V_k).$$

More generally, we can see that the partially evaluated program contains run_j predicates whose arguments are divided into two parts.

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

Although logically there is no distinction between these arguments, they might be treated differently by the analysis tools, and analysis results relating to these variables could be reported in different ways. For example, recurrence equations for the resource variables could be extracted from the specialised program, or relational abstract domains could derive invariants for each run_j predicate giving constraints relating the program variables and the resource variables.

4 LLVM and its Role in Analysis

The LLVM open source project is a collection of modular and reusable compiler and tool-chain technologies. The LLVM Core libraries provide a modern source and target-independent optimizer, along with code generation support for many CPUs. These libraries are built around a well specified code representation known as the LLVM intermediate representation (LLVM IR) [BW11].

LLVM supports the three-phase compiler design as demonstrated in figure 7. This makes it easier for a single compiler to support multiple source languages or target architectures. The strength of the whole design comes from the LLVM IR common code representation in its optimizer. For each source language only a front end that compiles to the LLVM IR is needed, and then the existing LLVM target-independent optimizer and back-end can be reused.

Due to the retargetability enabled by the three-phase design, all source languages and target architectures can get advantage of the continuous enhancements and improvements to the compiler by the open source community. Also there is a large number of useful tools working directly on the LLVM IR.

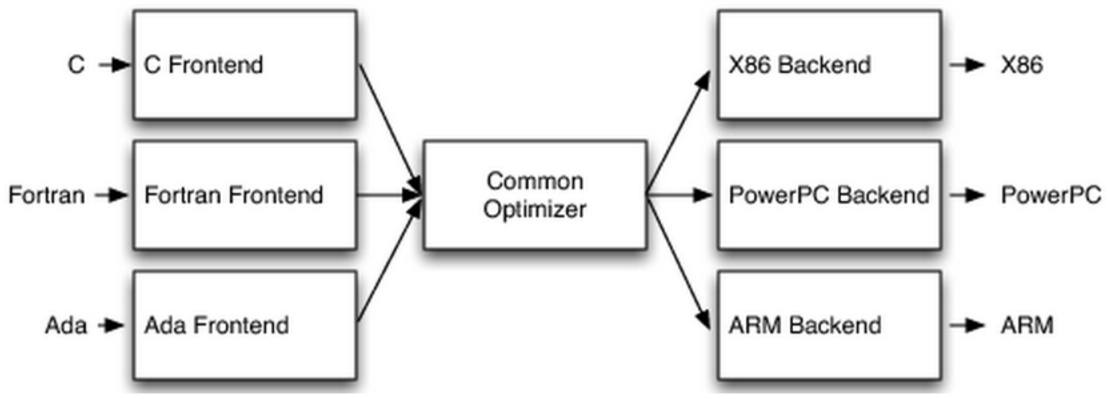


Figure 7: Three-phase compiler design using LLVM core libraries [BW11]

4.1 Accessing the LLVM IR Produced by the XMOS xcc Compiler

The XMOS xcc compiler uses both the LLVM optimizer and the code generator framework. The xcc front end emits LLVM IR code which then is passed to the LLVM optimizer. The LLVM optimizer runs a series of optimization and analysis passes on it depending on the specified optimization level and compiler flags. Our resource analysis must be done on the optimized LLVM IR code, after all the optimizations passes are completed. This will ensure that the intermediate code will be the one closer to the latter assembly emitted code, which will be executed.

There are three possible ways to access to the optimized LLVM IR code in order to perform our resource analysis. Firstly, using the xcc compiler flag “-emit-llvm” the LLVM IR code can be emitted and then an external tool can parse it and perform all the analysis. The second way is to implement an LLVM analysis pass to traverse the structure of the LLVM IR code and do the analysis. The existing framework of creating LLVM passes allows to create new passes rapidly with new functionality that can be used by many architectures. Finally, LLVM bindings exist for many languages such Java and Python which can provide interfaces to the LLVM optimizer. Probably for the scope of this project a combination of the methods above will be used.

4.2 LLVM vs. ISA: Advantages and Disadvantages of Analysing at these Layers

As explained before, the LLVM IR code produced the xcc front end compiler goes through a series of optimizations to finally end up in the LLVM *code generator*, the back-end of the compiler, which lowers the LLVM IR to the target ISA representation. The ISA representation layer, being closer to the hardware than the source or LLVM IR layer, makes it easier to characterize and profile power consumption and build an energy model for ISA than the other two layers. However due to the lowering, the ISA representation becomes unstructured and untyped, which hinders the capability of the analysers to infer bounds on energy consumption for programs with complicated structure and complex data types (i.e., structures and arrays). This analysis and modeling information trade-off was mentioned in Section 1 and depicted in Figures 2 and 3.

For the reasons discussed above, the analysis at the LLVM IR layer could be reasonably precise, since it is neither too close to hardware to suffer the loss of important program information (e.g., typing information), nor too close to the source language to suffer the power modeling and characterization problem. Theoretically, analysing LLVM IR will enable us to analyse a bigger class of programs since it preserves more program information for the analysis phase. However the profiling to obtain the energy models would not be as precise at the LLVM layer, which is farther away from the hardware than the ISA layer.

With the above considerations in mind, we plan to implement prototype tools to perform an experimental study of the analysis at the LLVM IR layer, which preserves the structure and semantics of the source program, using different options for the energy models. For example, we plan to use the already available energy models at the ISA layer, and use the information produced by the mapping tool that will be described in detail in Section 5.3 to propagate the ISA energy model information up to the LLVM layer in which the analysis is performed. We also plan to use (standalone) energy models at the LLVM IR layer, either, obtained directly by profiling, or obtained from the ISA models by using the mapping information that relates the LLVM IR and ISA layers, which is generated by the mentioned mapping tool.

5 Producing the Mapping Information

In this section we describe the mapper tool we have developed. This tool serves two main functionalities, as demonstrated in Figure 1. Firstly, to create mapping information between the different intermediate code representations, ISA and the source code. This enables the mapping of the analysis results back to the procedures and functions in the source code. The second functionality of the mapper tool, is to propagate the energy model information defined at one layer up to the (different) layer at which the analysis is performed. It is important to state that this mapping tool does not perform any analysis in principle, but it can be seen as the glue that brings together all the pieces of our framework for resource consumption analysis and verification, enabling energy transparency.

5.1 Mapping through Debug Information

After investigating our possible options for creating mapping information, we decided that the best approach is to take advantage of the existing debug mechanism in the XMOS tool-chain. This mechanism serves the debugging process used by a programmer to investigate and fix potential bugs in a crashing application or any other kind of faulty program behaviour. The main idea of this mechanism is the creation of debug symbols during the compilation of the source code and the propagation of them to all the different intermediate code layers and down to the ISA code. A Debug Symbol (DS) is information that expresses which programming-language constructs generated a specific piece of machine code in a given executable module. In our case the Debug Symbols are generated by the front end of the XMOS compiler and then they are transformed to LLVM metadata information attached to the LLVM IR. LLVM metadata is a mechanism to tag information for a custom code generator, or pass through information to link time optimization. LLVM 2.7 and upwards, provides first-class support for this, and has switched debug information over to use it. The main benefit of using the Debug Information (DI) is that we get advantage of the significant effort the LLVM open source community put in to preserving those metadata nodes through the different layers of optimizations and transformations taking place in the LLVM optimizer. The higher the optimization level applied during the compilation of a program, the less accurate DI associated to the resulted executable code. This is due to the fact that a lot of the initial code is either discarded or merged during optimization and transformation phases.

Although we use the XMOS tool-chain for the mapper tool, the approaches and techniques employed are generally transferable, due to the use of the common LLVM optimizer and code generator. The LLVM code generator is using the DI to produce DWARF information [dwa13], a standardized debugging data format used by many compilers and debuggers to support source layer debugging. DWARF is architecture independent and applicable to any processor or operating system.

5.2 Mapping the Analysis Results Back to the Source Code

As debugging information communicates Source Location Information (SLI), type information and variable information to the debugger, it is easy to manipulate them for the case of mapping the analysis results back to the procedures and functions in the source code. The idea was to create a mapping between source code, LLVM IR and ISA code using the location information of the source code stored in the DI. To achieve this, firstly an LLVM pass was created. This pass traverses the LLVM IR code and extracts the SLI stored in the metadata of each LLVM IR instruction. Then a data structure is created, a list of tuples of LLVM IR instruction and their corresponding SLI. Similarly, a list of tuples of ISA instructions and their corresponding SLI is created. We build this list by disassembling the executable of a program and then extracting the DI related to each ISA instruction. Finally a third data structure is created by parsing the source code and creating tuples of each instruction in the source code and its SLI. Having these three data structures, enables the mapping between the three layers. These data structures can be used as inputs to any resource analysis that need to correlate its results back to the source code.

5.3 Propagating Energy Models to a Different Layer Through Mapping

In the case of propagating the energy model information defined at one layer up to a different layer, the mapper is again using the mechanism of DI but by modifying the DI of a program at the LLVM IR layer to gain a more fine grained mapping. Since we currently have a low-level energy model, at ISA layer, we need to map this model up to the LLVM IR layer in order to perform the resource analysis at this layer. The idea here is to find which ISA instructions correspond to which 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 DI is also stored alongside with the ISA code using the DWARF standard as explained earlier. Using the same methodology described earlier we can extract a mapping between the LLVM IR and ISA. This will give as an n:m relationship between the two layers, because one source code instruction can be translated to many LLVM IR instructions and therefore many ISA instructions. In the case of communicating results from the analysis back to the source code this is enough, but for propagating energy values to the LLVM IR layer we need something more fine grained.

To address this issue, we created an LLVM pass which traverses the LLVM IR and replaces the SLI with LLVM IR location information, right after all the optimization passes and just before emitting the ISA code. In this way, we achieved to have an 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 more close to the ISA code than the unoptimized one, which will 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 in the degree of the LLVM optimizations. After the mapping is done for a specific program, the associated energy values for the ISA instructions corresponding to an LLVM IR instruction, are aggregated and then associated to the LLVM IR instruction. Although this mapping is more fine grained, we are expecting to have a deviation from the ISA energy model, due to the optimizations happening after the translation to ISA. This deviation will be estimated by using the analysis on the LLVM IR layer proposed in this document on our benchmarks and, depending on the results, the mapping will be further optimized.

Using this fine grained mapping, we plan to obtain an LLVM IR stand alone energy model. We will first get the mappings of a large set of programs using the methodology described above. Then we will perform a regression analysis on these mappings to identify energy values for each LLVM IR instruction. An experimental evaluation will also give us the deviation of this model.

6 Analysers

As mentioned in the introduction and illustrated by Figure 1, the analyser takes as input the analysable form of the program, the IR, together with the (IAL) assertions, expressing the energy models and possibly useful (trusted) information, and process them, producing the analysis results, expressed also in the IAL.

Energy transparency relies on the analysis results, and enables energy optimizations (WP4) and energy-aware software engineering (WP1). As illustrated in Figure 3, the analyser can infer information at different software system layers (e.g., XC source code, LLVM IR, and ISA) and can use energy models defined at the same layer or at a lower layer (provided there are mappings between software segments at different layers in the latter case). Thus, the IR represents a program at the layer the analysis is performed. The analyser deals with the IR always in the same way, independently of the layer of the program the IR represents. This implies that the analysis results enable (automatic) optimizations at different layers, e.g., at the LLVM IR layer, not only at the source code layer.

The analysis information at the source code layer can be used to help software developers to better understand the effect of their designs on the energy consumption, and make more informed design decisions. Such information can include the energy consumption expressed as functions on data sizes for the whole program and parts of it, such as procedures and functions.

In this section we describe different choices for the realization of the "analyser" box in Figure 1. The first one, described in Section 6.1, is based on the techniques present in the CiaoPP analysis tool [HPBLG05], which have been refined, enhanced and integrated in the general pipeline of energy transparency. The second choice, described in Section 6.2, is applicable to the analysis at the LLVM IR layer, and propose a direct process of LLVM IR by the analyser. Finally, Section 6.3 describes a model-based static analysis of CLP Programs.

6.1 Analysis with the CiaoPP Tool

CiaoPP [HPBLG05], the preprocessor of the Ciao programming environment [HBC$^+$12a], is a powerful tool for abstract interpretation-based analysis, optimization and verification of CLP programs. This tool has served as a ground for the development of a first prototype of an energy analysis (see attachment D3.1.1 for details).

Three main features of CiaoPP make it a good choice for our analysis framework:

- The CiaoPP analyser already uses the IR that we propose for our framework, CLP, as the analysable language.
- CiaoPP includes a generic abstract interpretation engine, PLAI that allows developing new analyses, which can be combined, ensuring correctness, as we will see later.
- In particular, CiaoPP includes a generic analysis for resource usage that can be instantiated to infer bounds on resources of interest, energy consumption in our case. The instantiation of such analysis for energy consumption (or any other resource) is done by means of an assertion language that allows the user to define resources and express the resource usage

of elementary program operations, certain program constructs, and library procedures. The details on how to perform such instantiation can be found in attachment D3.1.1.

- CiaoPP already implements the idea of using its assertion language to provide information to the analyser and output analysis results. This eases the communication between the tools in the pipeline.

6.1.1 Abstract Interpretation

One of the most important points for analysis is that it should be easily extended and changed, as we delve more into the complexities of energy consumption analysis. This implies that, rather than a single energy analysis, we would have to develop an analysis framework that allows changing or plugging new analyses for inferring other properties in the future. This section delves into the choice of *abstract interpretation* as a good ground for developing these analyses, including a brief introduction to its theory.

As mentioned above, CiaoPP includes a generic abstract interpreter called PLAI. This tool gives us the engineering and extensible side, and they build upon the basis of abstract interpretation. Furthermore, the ENTRA team includes researchers which have participated in the development of CiaoPP and which are experts in abstract interpretation of programs. As we will see below, some work force has been devoted to reengineering some analysis as abstract domains to take advantages of all of its features.

As a motivating example, in the current stage we use an energy model which assigns a constant amount of energy to each instruction. But in the near future we plan to enlarge the model with dependencies on the frequency, voltage, accuracy, and other state of the underlying hardware (the environment). This new feature needs changes in two sides:

- First of all, a new analysis for frequency, voltage, or any of the other processor characteristics must be developed. Writing this new analysis from scratch is quite time-consuming. Furthermore, it may be the case that we only need a prototype of the analysis to check whether a new idea for the energy model seems correct;

- The old analysis for energy must be re-engineered to communicate with the new analysis of processor characteristics. This requires some amount of work. Furthermore, these changes may put on risk the correctness of an analysis that was working until that point.

In conclusion, we are looking for an analysis framework that helps in the engineering process, while having enough theoretical power to express the analyses we need to perform and check their correctness. As mentioned before, the chosen analysis framework is *abstract interpretation*.

Introduced in 1977 by Patrick and Radhia Cousot [CC77b], abstract interpretation is a general methodology for developing static analyses of any kind of programs. The main idea is that the execution of the program is simulated by running it in an easier state space, usually called an abstract domain. By designing this abstract domain in the correct way (the formal definition of how the abstract domain should relate to the original program state), we are able to abstractly execute the program, and get as result an element of the abstract domain.

As an example, assume a simple programming language that works with values coming from a certain domain Σ. We would like to know statically at each point in the program which are the possible values for a variable: that is, each variable will have an assignment into an element of $\mathcal{P}(\Sigma)$. However, this analysis is undecidable in the general case (we may even run into infinite loops), so that a solution is choosing a smaller abstract domain and using the techniques of abstract interpretation to derive a static analysis. This abstract domain could be, for example, a set of types of variables.

The formal definition of an abstract domain requires both the initial set of possible states and the simpler one to be lattices, that is, sets with a partial ordering $\sqsubseteq$, and two operations called join ($\sqcup$) and meet ($\sqcap$) which must satisfy some laws with respect to the ordering. In this work our initial state space will always be the powerset of some domain of values, so we will show it as $\mathcal{P}(\Sigma)$ with the lattice structure given by set containment.

Up to this point, we are given a concrete domain $\langle \mathcal{P}(\Sigma), \subseteq \rangle$ and an abstract one $\langle \Delta, \sqsubseteq \rangle$. 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 $\alpha(x)$ computes the element in Δ that is the best (most precise) representation of x; while $\gamma(y)$ computes the element in $\mathcal{P}(\Sigma)$ that is represented by y.

From this Galois connection, there are several methodologies to calculate an analysis using the desired abstract domain (some of them are presented in [Cou99]). The main idea is that the analysis will over-approximate the result of the computation by choosing an element in Δ that includes at least all the possible elements from $\mathcal{P}(\Sigma)$. Using the particular properties of Galois connections, these analyses can also be proven correct. Furthermore, abstract domains can be combined in several ways [CC79], one of the key features that we wanted for our analysis framework.

Abstract interpretation is a very general methodology, which can encompass any possible computation schema. On the other hand, our translation produces programs with a very specific shape, so that it makes sense to use an instantiation of abstract interpretation specially tailored to these uses. In particular, we focus on the generic AND-OR trees procedure of [Bru91], with the optimizations of [MH92]. 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 or function along with an abstraction of the call patterns, that is, the information about the input state and arguments, 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 analyser present in CiaoPP [HBC$^+$12b]. In PLAI, abstract domains are pluggable units which need to define implementations of $\sqsubseteq$, least upper bound ($\sqcup$), bottom ($\perp$), and a number of other operations related to calls.

6.1.2 Resources

In order to accommodate several types of analysis, such as energy, time or accuracy, we build upon the concept of resource. This concept was already present in the resource analysis found in CiaoPP prior to the start of the ENTRA project [DLGHL97, LGDB10], but we have greatly enhanced and adapted it to the framework of abstract interpretation in the project.

In short, a *resource* is any numerical property that changes through the course of a computation. Examples of resources are memory usage, time spent on a piece of code, bytes sent over a network, or, central in our case, the energy consumed by the computation. Resource analysis is the inference of the aggregated value of any of these properties through the execution of a piece of code.

Usually, the amount of resources needed by some computation is not fixed, but rather depends on some environment properties or the inputs to the computation itself. This is obvious in the case of energy: its consumption depends on the frequency of the processor and the temperature of the environment, and also on the size of the data to be processed. For that reason, we are interested in *parametric resource analysis*, not just worst-case resource analysis.

6.1.3 Sized Types Analysis

Size analysis is one of the steps that is needed for producing a precise resource consumption analysis. This section motivates the need of a powerful size analysis, and presents *sized types* as a solution to this problem. The work on this area has been presented as a technical communication in ICLP 2013 [SLGBH13], and is attached to this deliverable as attachment D3.1.2.

As mentioned above, our starting point is the methodology outlined in [DLGHL97]. This approach shows several important features:

- It is based on setting up recurrence equations from the possible executions of the code. Solving those equations gives the developer a closed function, parametrized by input parameters or any other quantity of interest, giving the consumption of the resource;
- These equations are generated in three steps: first it performs an analysis of the size of data structures involved in the computation, then infers the number of solutions of predicates, and finally generates the recurrence equations for resource usage.

One drawback of that previous work is its very limited ability to cope with size information about subterms. However, dealing fully with subterms is useful not only for the system developer, but in fact is a key issue in the resource analysis of realistic programs. Without it, the analysis can only infer trivial bounds for a large class of programs. For example, consider a predicate which computes the factorials of a list:

```
listfact([],     []).
listfact([E|R], [F|FR]) :- fact(E, F), listfact(R, FR).

fact(0, 1).
fact(N, M) :- N1 is N - 1, fact(N1, M1), M is N * M1.
```

Intuitively, the best bound for the running time over a list L is $\alpha + \sum_{e \in L} (\beta + time_{fact}(e))$ where α and β are constants related to the 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 ∞.

Our solution is based on *sized types*, a representation that incorporates structural (shape) information and allows expressing both lower and upper bounds on the size of a set of terms and their subterms at any position and depth. Furthermore, we have developed the analysis using abstract interpretation and integrated in into PLAI and CiaoPP.

Sized types are derived automatically from the types of the variables present in the code. These types could be taken directly from the programmer input (for example, from XC code which includes function prototypes), or inferred by another analysis. We have chosen this second option in our implementation, integrating the abstract domain of regular types described in [VB02]. As we mentioned in the previous section, using this framework allowed us to integrate other analysis posed as abstract domains.

Using regular types, the type of "list of numbers" is represented as:

```
listnum -> []
listnum -> .(num, listnum)
```

where `num` is a built-in type for numbers. This notation tells us that we have two choices for building a value of the `listnum` type: either we create an empty list using `[]/0` or we create a cons-cell with the functor `./2`.

From this regular type definition, a sized type schema is derived. In our case, the sized type schema $listnum\text{-}s$ is derived from `listnum`. This schema corresponds to a list that contains a number of elements between α and β, and where each element is between the bounds γ and δ. It is defined as:

$$listnum\text{-}s \rightarrow listnum^{(\alpha,\beta)}(num^{(\gamma,\delta)}_{\langle .,1\rangle})$$

The $\langle ., 1\rangle$ below num expresses that this inner size description applies to subterms occurring at the first parameter of the `./2` functor.

In a subsequent phase, these sized type schemas are put into relation, giving raise to a system of recurrences where output argument sizes are expressed as function of input argument sizes.

Our experiments show that sized types improve by a large margin the amount of information given to the developer. This information is also used by the resource analysis, which results in much more precise resource consumption information, as it is shown in the next section.

6.1.4 Cardinality and Resource Usage Analysis

Our resource usage analysis builds upon the sized type analysis and adds recurrence equations for each resource we want to analyse. As mentioned above, our main aim for extending the power of size analysis was indeed getting more precise results in resource analysis. This section introduces the work done on enhancing and formalizing resource analysis as an abstract domain. This work has been presented at WLPE 2013 [SLGH13], and can be found in attachment D3.1.3 to this document.

In order to infer the resource usage, some description of the resource must be given to the analysis as input. Following [NMLGH07], we account for two places where resources can be modified:

- When entering a clause: some resources may be needed during unification of the call (subgoal) and the clause head, in 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, is called *head cost* and is represented by β,

- Before calling a literal: 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 δ.

Since our analysis is developed for general CLP programs, we need to take care of some features of logic programs:

- We may execute a literal more than once on backtracking.
- more than one clause may unify with a given subgoal.
- When estimating lower bounds on resource usage, in the case of a possibly failing literal, no resource usage should be added beyond the point where failure may happen.

All these features make a case for a cardinality analysis, which estimate bounds on the number of solutions of a certain predicate, and also for a non-failure analysis, which allows us to know at which point in a clause body we can find a possibly failing literal, which has an effect on lower bound resource analysis. Luckily, CiaoPP already integrates a non-failure analysis based on abstract interpretation [BLGH04] which we can directly integrate in our analysis.

The experimental results for this new analysis are very encouraging. In particular, we conducted an assessment comparing it with the resource analysis available in CiaoPP prior to the start of the ENTRA Project, and also comparing our new analysis with the Resource Aware ML system (RAML) [HAH12], which is the closest related work.

In the first case we get better results with the new resources abstract domain 40% of the times when inferring lower bounds, and 46% of the times for upper bounds. In the comparison with RAML we get better results 26% of the times. Furthermore, RAML is tied to polynomial resource usage functions, whereas our approach using recurrence relations supports a wider range of behaviours, such as exponential or logarithmic.

In conclusion, the sized type, cardinality and resource usage analyses we have developed for ENTRA are comparable to state-of-the-art systems in these fields. In the next section we will see how these analyses have been used in the specific case of energy consumption estimation the XMOS XS-1 architecture.

6.1.5 Energy Analysis of XC Programs with CiaoPP

As a proof of concept, in the ENTRA project, we are developing an energy consumption analysis for the XS1 processors designed by XMOS. This architecture is usually programmed using the XC language. In this section we introduce the work done in order to instantiate the CiaoPP analyser for energy consumption estimation, and use it to realize the general resource analysis as illustrated by Figure 1. Part of it is included in the paper accepted for publication at LOPSTR 2013 [LKS+13] and attached to this document with the identifier D3.1.1.

Defining Resources. As mentioned before, the resource analysis needs a description of the resource to be analysed. We start by defining the identifier ("counter") associated to the energy consumption resource, through the following Ciao declaration:

```
resource(energy).
```

Expressing the Energy Model. We connect the energy models developed in WP2 with the analyser, using the Common Assertion Language described in deliverable D2.1. Since we are using CiaoPP as the analysis tools, the assertions will be expressed in the specific syntax of CiaoPP.

For expressing the energy model, we use a predicate for each possible instruction and its associated formats. These are the predicates which are called in the transformation from low-level code to a CLP program. All of them have a similar format: there is some logic code abstracting the operation performed on the processor, which is needed by the analyses to derive correct equations; and after that a call to a predicate which is annotated with the resource consumption.

For example, the `lss` operation that performs the "less than" comparison on two registers is written as:

```
lss_3r(1,N,M) :- N < M,  lss_3r_cost(_).
lss_3r(0,N,M) :- N >= M, lss_3r_cost(_).
```

And the annotated cost predicate is:

```
:- trust pred lss_3r2(_)
+ ( is_det, not_fails, cardinality(1,1)
          , resource(energy,1141148,1141148) ).
```

Notice the inclusion of the information about cardinality, non-failure and determinism (as explained in attachment D3.1.3) and finally the information about resource consumption.

In this case, no energy consumption is associated to head unification or call to literals in program clauses, which are the means in which we express the low-level flow in a higher-level logic language. For that reason, the β and δ parameters to the analysis should be 0. This is expressed by the following Ciao assertions:

```
head_cost(energy,_,_,0).
literal_cost(energy,_,_,0).
```

Analysis of Jump-Based Control Languages. As mentioned in Section 3, the programming style in languages whose control is based on jumps is quite different from the style in more structured languages. In this stage of analysis, we find further mismatch between the low-level description and the description based on Horn clauses.

In logic programs, the tests to input variables are usually found in the head of the clauses. For example, this is how one would express the factorial function in Prolog:

```
fact(0,1) :- !.
fact(N,M) :- N1 is N-1, fact(N1,M1), M is N*M1.
```

However, the translation from low-level code to Prolog produces instead code similar to the following:

```
fact(N,M) :- less_than(B,N,1),fact_aux(B,N,M).

fact_aux(1,N,M) :- store(M,1).
fact_aux(0,N,M) :- subtract(N1,N,1),fact(N1,M1),multiply(M,N,M1).

% Built-ins described here for convenience
store(X,Y) :- var(X), X = Y, st_cost.
less_than(1,X,Y) :- X < Y, lt_cost.
less_than(0,X,Y) :- X >= Y, lt_cost.
subtract(X,Y,Z) :- X is Y-Z, sub_cost.
multiply(X,Y,Z) :- X is Y*Z, mul_cost.
```

After the unfolding step, the logic code to analyse is:

```
fact(N,M) :- N < 1, lt_cost,
             M = 1, st_cost.
fact(N,M) :- N >= 1, lt_cost,
             N1 is N-1, sub_cost,
             fact(N1,M1),
             M is N*M1, mul_cost.
```

In this code, the tests are now explicit in the body, instead of being done in the heads. If we follow the approach described in D3.1.2, the recurrence relations that will result, for example, for the energy resource would be:

$$fact_e(N) = \begin{cases} c_{lt} + c_{st} \\ fact_e(N-1) + c_{lt} + c_{sub} + c_{mul} \end{cases}$$

In those equations there is no mention to which case should be taken. Thus, it is an incorrect equation for which we cannot find any closed form solution.

The solution has been to enhance the sized type and resource analyses with special cases for those built-ins which introduce tests. From the description in attachment D3.2.1, the changes are as follows:

- The domain component is not only extended in the abstract unification (λ_{call} to β_{entry}) step, also when one of the built-in operations $=$, $>$, $\geq$, $<$ and $\leq$ is found in the analysis, during the $extend$ step;

- The β_{exit} to λ' step must perform an extra task: deriving the larger possible set of relations between bound positions for variables in the head. The algorithm here is very simple: apply the transitivity of the operators until no more relations can be discovered, and then retain only those whose both sides refer to those bound positions;

- The $\uparrow$ operator for obtaining closed form bounds must take care also of these new tests.

6.2 Direct Analysis of LLVM IR

As already said, modern compilers, such as XMOS xcc and those built using the LLVM framework, internally transform source programs into intermediate compiler representations, which are more amenable to analysis than either source or machine layer programs. The advantages ¡¡¡¡¡¡¡ .mine of analyzing at the LLVM IR layer have been already discussed in ======= of analysing at the LLVM IR layer have been already discussed in ¿¿¿¿¿¿¿ .r925 Section 4.2.

In this section we show how resource consumption analysis techniques can be adapted and applied to programming languages targeting LLVM IR (such as C or XC) by reusing some of the existing machinery available in the compiler framework (for instance LLVM analysis passes). We describe a tool for performing an analysis at the LLVM IR level. The analyser, together with the toolchain surrounding it can be seen in Figure 8. We discuss generic techniques, which can be used to extract cost relations from existing LLVM IR code. These can then be solved using existing solvers [AAGP11].

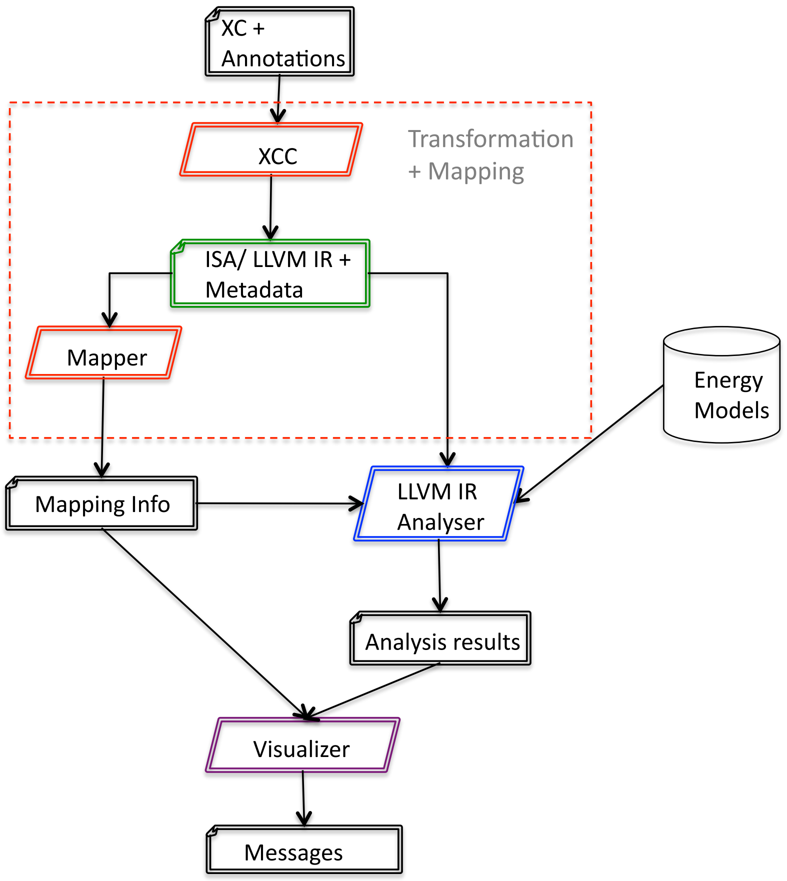


Figure 8: Realization of the general framework by analysing LLVM IR directly.

6.2.1 LLVM IR

For presentation purposes, we formalise a simple calculus of LLVM IR, based on the following syntax:

$inst = \mathsf{br}\ p\ BB_1\ BB_2$	conditional branch instruction
$\mid x = \mathsf{op}\ a_1..a_n$	generic side effect-free operation
$\mid x = \phi\ \langle BB_1, x_1\rangle..\langle BB_n, x_n\rangle$	phi nodes
$\mid x = \mathsf{call}\ f\ a_1\ ..\ a_n$	
$\mid x = \mathsf{memload}$	dynamic memory load operation
$\mid \mathsf{memstore}$	dynamic memory store operation
$\mid \mathsf{ret}\ a$	

We use metavariable names *p,f,a,x* to describe predicates, function names, generic arguments and variables respectively. The instruction semantics are modelled on the actual LLVM IR semantics [ZNMZ12]. 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 a predecessor block together with the variable which is propagated from this predecessor block to the current 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. We therefore choose not to statically model these instructions. In our abstract semantics of LLVM IR, we therefore treat variables assigned with values dynamically loaded from memory as unknown, which we represent using the symbol $?$.

LLVM IR instructions are arranged in *basic blocks*, labelled with a unique name. A basic block BB over a CFG is a maximal sequence of distinct instructions, $inst_1$ through $inst_n$, such that all instructions up to $inst_{n-1}$ are not branch or return instructions. The last instruction in a basic block is always a br or ret instruction. The only place where a ϕ instruction can appear is in the beginning of a basic block. All call instructions are assumed to eventually return.

6.2.2 Symbolic evaluation of LLVM IR variables

For every LLVM IR block, and for every output variable for the block, we aim to infer expressions that express output arguments in terms of input arguments. At the core of our resource analysis mechanism is a symbolic evaluation function $seval$. Given a block of code, and a variable symbolically executes a slice from this block to produce a result. A program slice, for a variable x, is a set of instructions that may affect the computation 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 the effect of dynamic memory reads and writes, i.e., memload and memstore. By doing so, we can produce simple expressions, which can be handled by existing solvers. We proceed by showing some example 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 interesting case, the symbolic evaluator concludes that $seval(BB,$`%not.zerocmp7`$) =$?. It does so by first evaluating `%not.zerocmp7`. This evaluates to `%deref6 == 0`. However, since `%deref6` is a dynamically loaded value memload, the analyser concludes that `%deref6` is ? and that therefore $seval(BB,$`%not.zerocmp7`$) =$?.

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

A lot of times, the code inside a block has no effect on a variable of interest. Therefore $seval($`%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 $seval(BB,$`%exitcond`$)$ is $($`%i.0`$+1) =$`%1` which is easily found by traversing the structure of the LLVM block backwards.

6.2.3 Inferring block arguments

Cost relations (CRs) [AAGP11] characterise the cost of running a program. These are ¡¡¡¡¡¡¡ .mine recursively defined and closely follow the control flow structure of the program. What we actually want to infer is a closed form formula modelling theInferring ======= recursively defined and closely follow the control flow structure of the program. What we actually want to infer is a closed form formula modelling the ¿¿¿¿¿¿¿ .r925 cost, which is parametric to any relevant input arguments to the program, which requires CRs. Unfortunately, solving multivariate cost relations and recurrence relations automatically is still an open problem, and the less arguments each relation has, the easier it is to solve these. Block arguments characterise the input data, which flows into the block, and is either consumed (killed) or propagated to another block or function. Due to the reasons outlined above, we designed an analysis algorithm to minimise the block arguments before inferring the CRs.

The algorithm for inferring block arguments is a data flow analysis algorithm. We will use a standard means to describe this algorithm, as in [NNH99]. We define a data flow analysis function gen, which given a basic block returns the variables of interest in that block:

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

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

$$gen_{blk}(BB) = \begin{cases} ref(seval(BB, p)) & \text{if } inst_n = [\mathsf{br}\ p\ \ldots] \\ \emptyset & \text{otherwise} \end{cases}$$

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

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

The data flow analysis function $kill$ is defined as:

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

Finally, we combine gen and $kill$ by utilising a transfer function, which is inlined into $args_{in}$ and $args_{out}$. These compute the relevant block arguments utilised by our resource analysis.

$$args_{in}(BB) = \begin{cases} \text{functions's args} & \text{if } BB \text{ is function's entry} \\ \bigcup_{BB' \in next(BB)} phimap_{\langle BB, BB' \rangle}(args_{out}(BB')) & \text{otherwise} \end{cases}$$

$$args_{out}(BB) = (args_{in}(BB) - kill(BB)) \cup gen(BB)$$

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

Functions $args_{in}$ and $args_{out}$ are recomputed until their least fixpoint is found. Our block arguments are found by evaluating $args_{in}$.

6.2.4 Generating Cost Relations

In order to generate cost relations, we have to perform two things. Firstly, we need to characterise the energy exerted by executing the instructions in a single block. Crucially, 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:

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

Where C_{LI}, C_{ret} and C_{LB} characterise 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 we cannot do without actually 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 produced using the techniques discussed in this section can be automatically solved by cost relation solvers after "massaging" these to a format accepted by the specific solver. Experimentally, we have discovered that there are cases where the optimised program structures produced by LLVM based compilers prevents the cost relation solvers from finding unique "cover points" in the structure of the cost relations. In order to solve this problem we have to perform simple transformations to the LLVM IR representation of the program that would move induction variables out of nested loops. We anticipate that the machinery for doing so is already available in the LLVM toolkit.

6.3 Model-based Static Analysis of CLP Programs

Analysis of CLP programs can be based either on the model semantics or the proof semantics of CLP. The latter approach underlies the CiaoPP tools described in Section 6.1. In this section we focus on the model semantics. The essential task is to compute over-approximations of the minimal model of a given CLP program. Thereby invariants of the model can be inferred or checked.

For instance given a CLP program representing an XC or XS1 source program, suppose that the predicate $run_j(X_1, \ldots, X_n, E_1, \ldots, E_k)$ represents the state of computation at some program point j. Then the model of the program might imply facts of the form

$$run_j(X_1, \ldots, X_n, E_1, \ldots, E_k) \rightarrow \mathcal{C}$$

where $\mathcal{C}$ is some invariant on the variables of run_j. Note that these could relate program variables and resource variables at program point j as explained in Section 3.2.4. That is, the arguments of $run_j(X_1, \ldots, X_n, E_1, \ldots, E_k)$ may represent a mixture of program variables $X_1, \ldots, X_n$ and resource variables $E_1, \ldots, E_k$, and thus the derived invariant $\mathcal{C}$ can capture relationships between them.

We note that computing models for CLP programs provides the basis for several state-of-the-art software model checkers and verification tools that perform well in software verification competitions [1].

A model 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 $Z_1, \ldots, Z_n$. The constrained fact $A \leftarrow \mathcal{C}$ is equivalent to the closed formula $\forall(\mathcal{C} \rightarrow A)$.

A set of constrained facts M denotes the set of true facts $\{p(v_1, \ldots, v_n) \mid M \models p(v_1, \ldots, v_n)\}$, where $v_1, \ldots, v_n$ are objects in the domain of the constraint theory. Depending on context, a model M may stand for a set of constrained facts or the denoted set of true facts.

The "immediate consequence" operator for a CLP program (a generalisation of the standard T_P function for logic programs) is given as follows.

$$T_P^{\mathcal{C}}(I) = \left\{ A \leftarrow \mathcal{C} \;\middle|\; \begin{array}{l} A \leftarrow B_1, \ldots, B_n, D \in P \\ \{A_1 \leftarrow \mathcal{C}_1, \ldots, A_n \leftarrow \mathcal{C}_n\} \in I \\ \exists\, \theta \text{ such that} \\ mgu((B_1, \ldots, B_n), (A_1, \ldots, A_n)) = \theta \\ \mathcal{C}' = \bigcup\limits_{i=1,\ldots,n} \{\mathcal{C}_i\theta\} \cup D \\ \mathsf{SAT}(C') \\ \mathcal{C} = \mathsf{proj}_{Var(A)}(\mathcal{C}') \end{array} \right\}$$

$$M^{\mathcal{C}}[\![P]\!] = \mathsf{lfp}(T_P^{\mathcal{C}})$$

[1]See `http://sv-comp.sosy-lab.org/2014/`

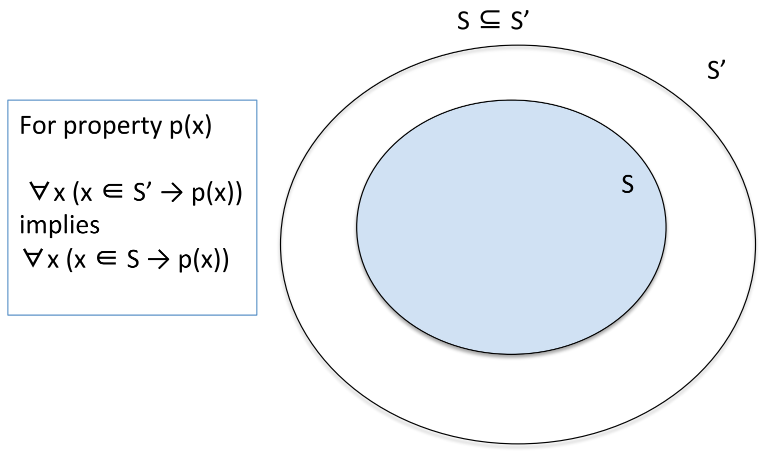


Figure 9: Proof by approximation

Here the SAT operator is a satisfiability check, and we assume that the constraint theory underlying the CLP language has a satisfiability procedure, such as those built in to powerful SMT solvers [DdM06, CGSS13, dMB08].

$M^{\mathcal{C}}[\![P]\!]$ is the minimal model of P over the constraint theory underlying the program.

6.3.1 Checking program properties

The minimal model is equivalent to the set of facts derivable from the program. So given an atomic formula A we can check whether $P \models A$ by checking whether $A \in M[\![P]\!]$. This is an undecidable task for CLP programs over most constraint theories, even those theories that have a decidable satisfiability test, such as linear arithmetic. Hence we focus on computing decidable *over-approximations* of $M[\![P]\!]$ (see Figure 9).

A typical case of proof by approximation is to show that some property holds for all elements of the minimal model of a program (an invariant). It is sufficient to show that the invariant holds in an over-approximation of the model. One common case of an invariant for programs is a *safety property*, stating that certain states are not reachable. If such states are not reachable in a program representing an over-approximation of the reachable states, then the safety property is established.

Computing Fixpoints. The minimal model is computed as the least fixed point of the immediate consequence function $T_P^{\mathcal{C}}$. This is the limit of the Kleene sequence $\emptyset, T_P^{\mathcal{C}}(\emptyset), T_P^{\mathcal{C}}(T_P^{\mathcal{C}}(\emptyset)), \ldots$. In general this is not a finite sequence – hence *approximation is required*.

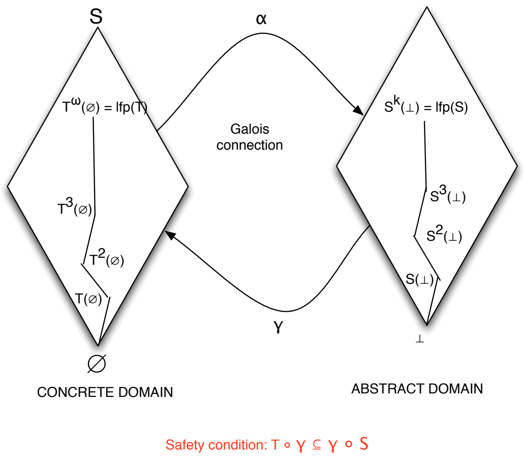


Figure 10: Abstract interpretation of CLP in one picture

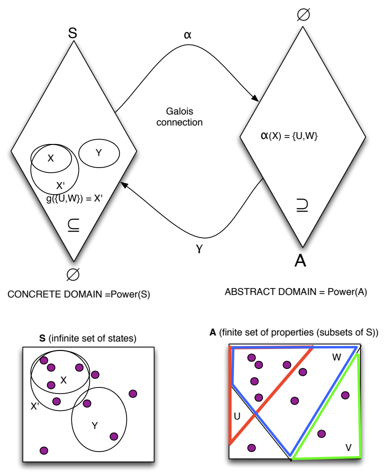

S
α
∅
Galois connection
α(X) = {U,W}
X
Y
X'
g({U,W}) = X'
⊇
⊆
γ
A
∅
CONCRETE DOMAIN =Power(S)
ABSTRACT DOMAIN = Power(A)
S (infinite set of states)
A (finite set of properties (subsets of S))
X
X'
Y
W
U
V

Figure 11: Predicate Abstraction

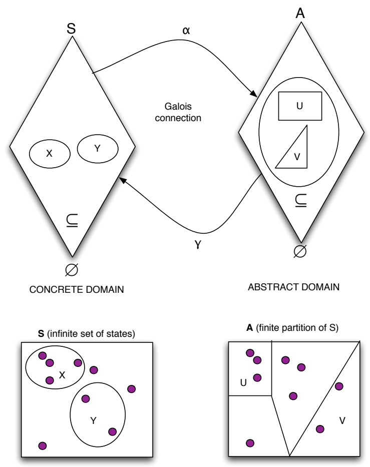


Figure 12: Partition-based Abstraction

6.3.2 Abstract interpretation of fixpoint semantics

The framework of abstract interpretation [CC77a] gives a systematic and practical approach to proving program properties by safe approximation of the semantics. The essential elements of abstract interpretation are concrete and abstract domains for expressing concrete and abstract semantics respectively, with two monotonic functions α and γ called the abstraction and concretisation functions respectively. The generic form of abstract interpretations for CLP is shown in Figure 10. Here, the function T is shorthand for the concrete function $T_P^{\mathcal{C}}$ over the domain 2^{S} where S is the set of all atoms of the CLP program. We construct a function S over some other domain whose elements describe sets of atoms. The least fixed point of S (or some post-fixpoint of S) describes an over-approximation of the least fixed point of $T_P^{\mathcal{C}}$. A generic correctness condition relating T and S is given, that can be used to establish the soundness of particular abstractions.

We briefly review three kinds of abstract domain widely applicable to abstract interpretation of CLP programs. Two of them employ finite domains, while the other uses an infinite domain together with a convergence accelerator (a so-called *widening* operator) to ensure termination of the Kleene sequence.

Constraint Domains: Property-based Abstractions. A property-based abstraction is an abstract interpretation, illustrated in Figure 11. Let A be some finite set of constraints. Let S be a set of atomic formulas. Then the abstraction function α_1 yields some subset of A, namely $\alpha_1(S) = \{q \in A \mid \forall s \in S.(q \rightarrow s)\}$. That is, $\alpha_1(S)$ is the set of constraints that hold for all elements of S. The concretisation function γ_1 is defined on the subsets of A. Let $A' \subseteq A$: $\gamma_1(A') = \{s \mid \forall q \in A'.(q \rightarrow s)\}$, that is, the set of atoms that satisfy all the constraints in A'.

Then we construct a function $T_P^{\alpha_1} : 2^A \rightarrow 2^A$, defined as $\alpha_1 \circ T_P^{\mathcal{C}} \circ \gamma_1$.

An abstract interpretation of a program P with respect to a set of properties A is then defined as $\mathsf{lfp}(T_P^{\alpha_1})$. This is the limit of the sequence $A, T_P^{\alpha_1}(A), T_P^{\alpha_1}(T_P^{\alpha_1}(A)), \ldots$, which converges after a finite number of steps. The framework of abstract interpretation establishes that $\mathsf{lfp}(T_P^{\alpha_1})$ represents an over-approximation of the minimal model of P, i.e. that $\mathsf{lfp}(T_P^{\mathcal{C}}) \subseteq \gamma_1(\mathsf{lfp}(T_P^{\alpha_1}))$.

Constraint Domains: Partition-based Abstractions. A related abstract interpretation is based on a set of properties that forms a partition of the state-space of the program (that is, every atom A satisfies exactly one property). This abstract interpretation is illustrated in Figure 12.

Given a partition A (as a set of constraints), the abstraction function on a set of atoms S is the set of elements of the partition that S overlaps with. That is, $\alpha_2(S) = \{q \in A \mid \exists s \in S.\mathsf{SAT}(q \wedge s)\}$, while γ_2 yields the set of all atoms that could satisfy the a set of constraints;

$\gamma_2(A') = \{s \mid \exists q \in A'.SAT(q \wedge s)\}$.

An abstract interpretation is then constructed using the same pattern as for property-based abstractions. Note that the ordering on the abstract domain is opposite in the two domains: in property-based abstractions a larger set of properties describes a smaller set of atoms than a smaller set of properties, whereas a larger set of partition elements describes a larger set of elements than a smaller set of elements.

Constraint Domains: Convex Polyhedral Abstractions. Both of the two abstract domains described above are based on a given fixed set of properties. By contrast, in the domain of convex polyhedra no properties are supplied - the purpose of the analysis is to discover them. We do not give details of this domain here: it was invented by Cousot and Halbwachs [CH78] and implemented for logic programs by Benoy and King [BK96] and by Henriksen *et al.* [HBG07] among others. The PPL library [BHZ08] provides a robust and efficient implementation of operations on convex polyhedra and related objects. Special cases of convex polyhedra that involve less complex analyses include intervals, octagons and difference-bound matrices.

For a CLP program, a convex polyhedral abstraction returns a set of constrained atoms of the form $p(X_1, \ldots, X_n) \leftarrow \mathcal{C}$, one such constrained atom for each predicate p of the program. The constraint $\mathcal{C}$ represents a convex polyhedron as a conjunction of linear constraints. This constraint $\mathcal{C}$ is an invariant discovered by the analysis.

Note that the Kleene sequence for the convex polyhedral abstraction does not usually terminate. It is necessary to use a widening operator to force termination, at the price of losing precision.

Convex polyhedra are a useful and expressive domain; however our intention is to supplement them with predicate-based abstractions and specialisation algorithms in the context of an iterative abstraction-refinement algorithm (see below).

SMT Solvers in Constraint-based Abstractions. The domains described above are all heavily reliant on constraint solvers to compute the abstract semantics. In particular, constraint satisfiability algorithms play a critical role in the abstraction functions for property-based abstractions and partition-based abstractions. In experiments to date we made use of the powerful SMT (Satisfiability Modulo Theories) solver Yices [DdM06]. Others that can be used are Z3 [dMB08] and MathSAT [CGSS13]. These are applied as back-end solvers for the analysis tools interfaced to Prolog (the implementation language for our analysis tools) as external libraries.

Solvers for linear arithmetic constraints are the basic tool employed so far, but the SMT solvers cited above contain satisfiability procedures for a range of theories including equality and uninterpreted functions, bit-vectors, and arrays. These theories (especially arrays) will be essential to the analysis of realistic programs.

Abstraction-refinement Techniques Combined with Abstract Interpretation. The abstract interpretations outlined above can be embedded in a family of iterative verification algorithms called abstraction-refinement. For example, the choice of properties with which to construct a property-based or partition-based abstraction is not easy to invent directly. The central idea of abstraction-refinement is to start with a coarse abstraction and successively refine it, using information gained from previous unsuccessful attempts. Often, refinements are drawn from counter-examples; if the required property cannot be proved in a given abstraction then the failure of the proof can guide the construction of the next refinement. This family of verification algorithms is known as CEGAR (Counterexample-Guided Abstraction and Refinement) [CGJ$^+$00]. An example of a state-of-the art CLP-based approach for software verification using the CEGAR approach is presented by Grebenshchikov *et al.* [GLPR12].

7 Showing the Analysis Information to the User

In this section we comment on possible realizations for the "visualizer" box in Figure 1. The results from the various parts of the resource consumption analysis will be used either for transformation of programs into more efficient forms or relayed back to the programmer as suggestions for improvements. The general approach of ENTRA is based on transformation from source code to an internal representation for analysis. This presents a challenge, namely that results for the analysed program has to be directly related back to the source program level.

This topic is dealt with in more detail in Deliverable 2.1 (Common Assertion Language) but it also concerns the analysis tools directly, since source code position information is retained throughout the analysis process. In order to explore some of the practical issues we have constructed a prototype version of an editor/analyser framework. Key elements of the prototype are as follows:

- The source code parser will store source file positions such as line and column numbers for all variables, expressions and statements.
- When programs are translated to intermediate representations the source file positions are kept, either as comments or as extra values in representation.
- The source file information are merged into the analysis results as assertions (since the assertion language will have the ability to specify scope).

- The editor can display results from the analysis, for example as highlighting, mouse-over pop-up boxes or textual annotations added to the code.

The overall goal of the analysis tools is to inform the developer of analysis results, such that the programmer can make immediate use of it in developing energy efficient programs. We intend to continue experimentation with the prototype editor/analyser to investigate the challenges raised. These include the following:

- The level of detail can be critical: we may be able to extract fine-grained information about a program's behaviour but very detailed information may confuse the programmer and clutter the development process.

- We also need to avoid any negative impact on the performance of analysis tools by the extra source code information carried into the internal representation.

8 Probabilistic Resource Usage Analysis

Analysis of upper and lower bounds facilitates many optimizations to be performed with respect to time and energy. For example, we can decide whether to split some code in two parallel tasks or not by taking into consideration the time and energy spent in both the sequential and parallel case and composing the results (parallel execution may take less time, but may consume more energy because it needs to wake up another processor). Tight bounds on the resource consumption also allows us to perform a better scheduling of tasks in the processor.

However, interval-based resource analysis is too restricted. It will give safe bounds for the best and worst case, without any information about how many times those cases are to be reached. So, if we perform a subsequent optimization based on intervals, this will not always give the ideal results.

For example:

- Error branches are often very expensive. However, the resource analysis takes into consideration all possible paths of execution, including error branches. Given that a developer typically would want to optimize resource usage for non-error execution, optimizations for error traces may not be appropriate;

- To perform an optimization that guarantees to give better results in 90 percent of the cases, may only be interesting if the gain is much larger than the losses in the other 10 percent.

The previous limitations make the case for a more powerful resource analysis, which takes into consideration not only the bounds in which the resource consumption will appear, but also the frequency with which time or energy value will happen in the measurements. We have called this extension *probabilistic resource usage analysis*, because our aim is to model these scenarios with the help of probability distributions over the variables.

Once we start using probabilities, there is even a larger range of applications in the ENTRA project. For example, we have used a simple model which assigns a constant cost to each ISA instruction. If we use a simple Gaussian distribution for each instruction instead, we could get information about the variability of energy consumption of the whole program. Since an energy model is inherently uncertain, a more realistic model which includes probability distributions for each instruction seems a natural step to take.

8.1 The Need for Dependence Analysis

Modelling resources using only upper and lower bounds facilitates simplicity and gives us the advantage of using a rough approximation. For example, we can be sure that if an upper energy usage bound for program function p is E_p and similarly the bound for q is E_q, a correct upper bound for the sequential composition of the program functions is $E_p + E_q$, plus possibly some constant related to the cost of preparing a call.

However, when dealing with probabilistic resources, things are more complicated. If we know the distribution of energy spent in each of the task executions p and q, namely E_p and E_q, we cannot directly deduce the distribution of its sequential or parallel composition $E_{p,q}$, due to a possible correlation between them.

As an example, consider variables X and Y discrete and uniform over the values $\{1, 2, 3, 4, 5, 6\}$, modelling a dice roll. The distribution of $X + Y$ can differ as described in the following (and illustrated in Figure 13):

- When X and Y are independent, the distribution of its sum is given *convolution* of X and Y:

  $$P(X + Y = z) = \sum_x P(X = x)P(Y = z - x)$$

  The more dice that are summed together, the more the shape will look like a bell-like shape that is found in most maths textbooks.

- When X and Y are strongly dependent on each other, in particular, the outcome of X and Y are always the same, the distribution of their sum is:

  $$P(X + Y = z) = \begin{cases} \frac{1}{6} & z \text{ even} \\ 0 & z \text{ odd} \end{cases}$$

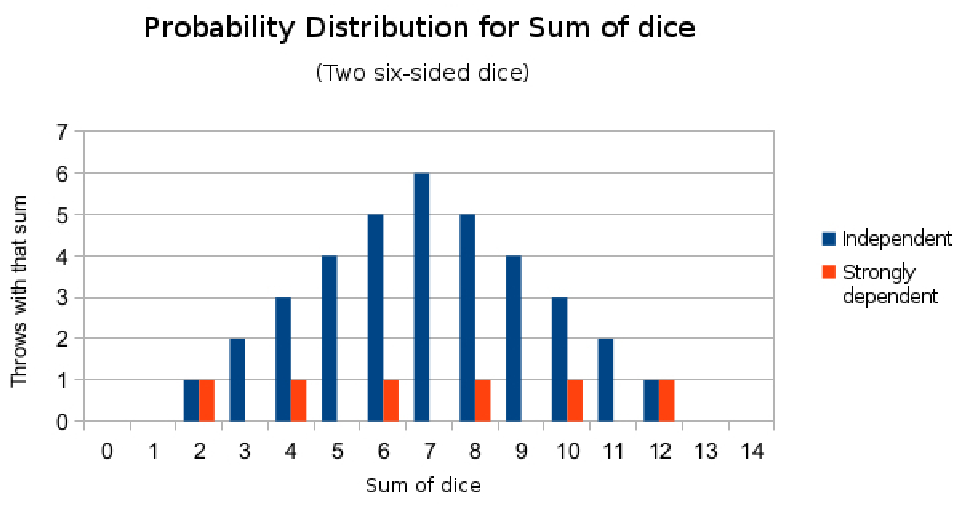


Figure 13: The probability distribution for the result differs whether the variables are independent or strongly dependent.

An example is that the sum 7 can occur 1 in 6 times when the dice are independent and occurs 0 times when the dice are strongly dependent. This example shows that, when dealing with probabilities, one must be very careful with dependencies between random variables.

In probability theory we know that dependence between variables is relevant. The same is true for variables in programs: Variables are often dependent on each other, as any assignment gives rice to dependency (e.g. in `a := b + c` the value of `a` is dependent on the value of `b` and `c`) and also any condition can introduce dependence between variables (e.g. in the program `if(x > y) then x else y;` the if-condition `(x > y)` will introduce a dependency between y and x because only some pairs of x and y values can occur in a branch (have a probability above 0). We can generalize this to resources: We have two sequential program parts each with its own distribution of estimated resource usage. If we assume independence of the two program parts we sum the distributions of the energy estimates (equivalent to the independent dice). Independence is not safe in the general case as it may be such that if one program part is slow then the other program part is also slow, and if one program part is fast then the other is as well. In other words, the resource usage for each program part may depend on each other, and then the sum will behave like the strongly dependent dice.

A general observation is that the assumption of independence has a tendency to under-approximate the worst and best cases and over-approximate the rest (equivalent to the dice graph, where for the independent dice the probability for the sum is equal to 12 is 1:36 and for the strongly dependent dice the probability is 1:6.

The goal is to decide the exact dependencies between variables. However, that is undecidable when programs contain non-analysable parts as this implies that the probability distribution is undecidable. Therefore we can only approximate the dependencies, and our work is focused on producing a safe and sound probabilistic analysis which gives precise and usable probability distributions, even in those non-analysable cases. Therefore we seek the most precise and still decidable approximation of the result.

In the next sections we will introduce our research on two different approaches, which both seek a good representation of variable dependency. In the first section we will present *copulas*, a mathematical framework for describing dependencies between random variables without having to know the distribution of the variables themselves, which allow us to avoid joint distributions. This representation seems to fit into abstract interpretation because the copulas naturally forms a lattice. The second section presents a transformation-based approach using a special constraint function to describe the variable dependencies. This approach uses probability distributions and transforms the program into a distribution function describing the probability distribution for the output of the program.

We will explore some of the definitions and results that should be used in a future implementation of such an analysis, and sketch the challenges and future work to be done in this area.

8.2 Copula-Based Analysis

In this section we will be showing definitions and results mostly from the monograph on copulas by Nelsen [Nel03]. Formally, a (2-)*copula* is a function $C : [0,1] \times [0,1] \to [0,1]$ such that:

1. Boundary conditions: $C(x,0) = C(0,y) = 0, C(x,1) = x, C(1,y) = y \; \forall\, x, y \in [0,1]$

2. Monotonicity or 2-increasing: for a, b, c, $d \in [0,1]$ with $a \leq b$ and $c \leq d$,

$$V_C([a,b] \times [c,d]) = C(b,d) - C(a,d) - C(b,c) + C(a,c) \geq 0$$

   V_C is called the C-volume of the rectangle.

Copulas become very interesting when we look at Sklar's Theorem (1959): *Let H be a two-dimensional distribution function with marginal distribution functions F and G. Then there exists a copula C such that $H(x,y) = C(F(x), G(y))$. This copula is given by:*

$$C(x,y) = H(F^{-1}(x), G^{-1}(y))$$

Conversely, for any distribution functions F and G and any copula C, the function H defined above is a two-dimensional distribution function with marginals F and G. Furthermore, if F and G are continuous, C is unique.

The distribution of $F(x)$ and $G(y)$ is always uniform. So, in essence, Sklar's Theorem allows us to partition a joint distribution into the marginal distribution of each variable in isolation, and the copula C with has all the information about dependence.

Examples of copulas arise in many contexts:

- The *product copula* $\Pi(u, v) = u \cdot v$ corresponds to the case of random variables being independent,
- The *comonotonic copula* $M(u, v) = \min\{u, v\}$ models a perfectly positive correspondence between variables. It is used in several financial models to derive the maximum risk of a set of products, which occurs when it is assumed that all of them are correlated. That is, if one defaults, the other ones have high probability of doing so.
- The *countermonotonic copula* $W(u, v) = \max\{u + v - 1, 0\}$ which models maximal negative correlation.

There is very strong connection between copulas and dependence measures [OOS13]. We can partially order the copulas using *concordance* [Nel03] a copula C is *less concordant* than C', $C \preceq C'$ iff:

$$C(u, v) \leq C'(u, v); \forall\, u, v \in [0, 1]$$

An important result of the theory of copulas is that for every copula C we have:

$$W(u, v) = \max\{u + v - 1, 0\} \leq C(u, v) \leq \min\{u, v\} = M(u, v)$$

The functions W and M are called in this context the *Fréchet-Hoeffding bounds*. Indeed, the copulas in 2 dimensions form a lattice structure where W and M are the bottom and top element, respectively. Such a structure is very appealing for an analysis using abstract interpretation.

8.2.1 Application to Resource Usage Analysis

Copulas were previously been used for timing analysis [GBN05]. Once we have the dependence information for all points of the program, we can use it to perform a powerful probabilistic resource analysis, and also to guide some of the optimizations to be performed.

In general, suppose we have a dependency between $X_1, \ldots, X_n$ given by copula C, and a non-decreasing continuous function $\phi(x_1, \ldots, x_n)$. Results from the copula theory makes it possible to derive bounds for the expected value of $\phi(X_1, \ldots, X_n)$ from the copula and the function ϕ. In our analysis, these non-decreasing functions are very interesting, because two scenarios related to resources are modelled this way:

- The consumption of energy by two pieces of code (either sequential or parallel) and the time taken by two sequential program elements use addition, a non-decreasing function;
- The time taken by executing two parts of the program in parallel is roughly the maximum of the two execution times, again a non-decreasing function.

Thus, we can use this to derive bounds on the mean of both time and energy consumed by a piece of code. This information is instrumental for many optimizations.

8.2.2 Nested Archimedean Copulas

Within the set of copulas one can define various classes that may form the basis of abstractions. The Archimedean copulas is a such a class where one can define domains of copulas with finite height and with the Fréchet-Hoeffding lower and upper bounds as extremal values.

The Archimedean copulas are symmetric so that $C(u, v) = C(v, u)$. To overcome this limitation, *nested* or *hierarchical* Archimedean copulas (HACs) are used. These are n-dimensional Archimedean copulas where the parameters of some generator can be replace by another nested copula, and so on recursively, in a tree-like structure.

8.3 Transformation based Probability Analysis

Another approach to deduct probability distribution for the output is based on transformations. In program analysis dependence between variables has been exemplified in two ways: data dependency, based on assignments, and control dependency, based on whether a condition evaluates to true. However, when working with probabilities there are further complications; when two variables have been compared in a condition, then knowledge about one of the variables will allow us to deduce something more precise about the other. For instance, if x and y are independent, and there is condition (x=y), then the x and y will be fully dependent on each other in the true branch.

To keep this information through the rest of the program we propose to use a constraint-based approach, where we use probability summation in combination with a constraint function C.

Definition (constraint function)**.** $C(condition)$ *is defined as*

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

A condition could for instance be $x < y$, $z = f(x, y)$, or $1 \leq x \leq n$.

For a program function f, we would like to find the probability distribution for the output, P_f. The probability distribution for the output is based on the probability distributions for the input variables. Given a program function f with input parameters x and y, $f(x,y)$, with the distributions P_x and P_y, respectively, we can define a new random variable z describing the result of the function $z = f(x,y)$. Then we can describe the distribution of the output of f as a function on z, $P_f(z)$:

$$P_f(z) = \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot C(z = f(x,y)) \tag{11}$$

We instantiate this expression with the program function and perform a little algebra, which turns it into a closed form expression describing the distribution of the output. We have found that using mechanical transformations (assuming that the input is uniformly distributed) we can transform the following example program into a probability distribution for `max`, $P_{\mathtt{max}}$.

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

Given that P_x and P_y are independent uniform distributions, where all numbers from 1 to n have an equal large probability, namely, $\frac{1}{n}$:

$$P_x : \mathsf{unif}(1,n) \qquad\qquad P_y : \mathsf{unif}(1,n)$$

At first, we insert the program function into the formula (11)

$$\begin{aligned}
P_f(z) &= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot C(z = \mathtt{max}(x,y)) \\
&= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot C(z = \mathtt{if}\ \ (x > y)\ \ \mathtt{then}\ \ x\ \ \mathtt{else}\ \ y) \\
&= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot (\, C(x > y) \cdot C(z = x) + C(x \leq y) \cdot C(z = y)\,)
\end{aligned}$$

We use mechanical rewritings to transform this into a probability distribution describing $P_{\mathtt{max}}(z)$:

$$P_{\mathtt{max}}(z) = \frac{1}{n^2} \cdot (2z - 1) \cdot C(1 \leq z \leq n)$$

where $P_{max}(z)$ is sound for all z.

8.4 Summary and Future Directions for Probabilistic Analysis

In the previous sections we describe why probabilistic analysis is important for extracting resource usage information from programs. The main challenge in this analysis is to handle dependencies between variables and programs parts. As with any powerful program analysis we need a safe and sound approach to approximate information in those situations where program parts cannot be analysed. We have outlined two approaches to perform probabilistic analysis, one based on copulas and one using program transformation techniques.

8.4.1 Copula Analysis Using Abstract Interpretation

The idea is to devise an analysis of dependence between numerical variables using *abstract interpretation*. The concrete domain will be the one of copulas, which would be abstracted by nested Archimedean copulas containing the Fréchet-Hoeffding lower and upper bounds.

So far, we have designed an initial probabilistic analysis based on one way to abstract sets of copulas but further work is needed to assess various ways to make abstractions and ways to use dependency information in subsequent analysis.

8.4.2 Transformation Based Probabilistic Analysis

The transformational approach to probabilistic analysis uses rewriting techniques known from computer algebra systems to obtain simplified or closed form expressions that describe probability distributions for resource usage. In general programs will contains parts that cannot be analysed and where we need to develop techniques to over and under approximate probability distributions.

If transformation techniques cannot simplify descriptions of probability distributions we may use approximation techniques to obtain simpler and safe limits to distributions. One approach we will explore is to extend probability distributions with constraints on possible relationship between variables.

8.4.3 Probabilistic Information in Program Development

The aim is to incorporate probabilistic analysis into the analysis framework. The various techniques needs to be explored for the ease of implementation, efficiency, precision and how the information can be used in program transformations or relayed to a programmer in a way that can guide the development into more energy efficient solutions.

9 Analysis of Concurrent Programs

The various analysis techniques discussed in previous sections are applicable to both sequential and concurrent programs, provided that they can be translated to the internal semantic representation, which consists of constraint Horn clauses (CLP). In Section 9.1 we describe current work in translating XC programs into this form, using the approach described in Section 3.2.4. That is, the semantics of XC is formulated as small-step operational semantics, from which an interpreter in CLP is systematically developed. This is partially evaluated to yield a "pure" CLP program whose variables range over the values underlying the XC program.

This work proceeded in the following stages:

- Modelling semantics of XC programs including aspects concerning concurrency and communication. The purpose was to provide a reference semantics, though not necessarily the semantics that is partially evaluated. The reference semantics gives a classical interleaving semantics to parallel threads.

- Using the reference semantics as a starting point, a deterministic "parallel-step" semantics was developed. This semantics is designed with partial evaluation in mind, in that it is possible to partially evaluate the semantics with respect to an input program without the blow-up in program size that the interleaving semantics would give. On the other hand, the deterministic semantics does not yield information about all possible states that could arise, but only captures the states that arise at the start and end of each parallel statement. This is expected to be sufficient for energy usage analysis.

- The deterministic semantics was instrumented with statement counters (see Section 3.2.5) and partially evaluated using Logen.

In Section 9.2 we introduce another possible approach towards the analysis of multi-threaded programs using event interaction graphs.

9.1 XC Semantics

In this section we introduce the main features of the XC semantics. The parallel-step semantics for XC is shown in detail in attachment D3.1.6.

XC is "an imperative programming language with a computational framework based on C" [Wat09]. (An important difference from C is that there are no pointers.) XC provides features for creating concurrent threads that can communicate and synchronise with each other using channels.

The most distinctive semantic features of XC relate to concurrency. One property is that parallel threads cannot make any updates to shared variables; another is that communication channels have two ends owned by exactly one thread each, so there is no race on being the first to send or get on a given channel.

9.1.1 Parsing and Abstract Syntax

A parser analyses the structure of an XC program according to the XC grammar, and outputs an abstract syntax tree (AST) that represents the essential structure of the program. The grammar of XC is not shown here; a parser is assumed that returns the AST. It is also assumed that the parser checks types while generating the AST as there is no type checking in the semantics.

Abstract Syntax. The abstract syntax of XC statements S for the present purpose is as follows. We let x (variable names) range over a set $\mathcal{N}$ of names.

$$\begin{aligned}
\text{(Values)} \quad & V ::= n \mid \alpha \\
\text{(Expressions)} \quad & E ::= V \mid x \mid E_1 + E_2 \mid f(E) \mid \{S\} \\
\text{(Statements)} \quad & S ::= \mathsf{skip} \mid \mathsf{return}\, E \mid x := E \mid S_1 \,;S_2 \mid S_1 \parallel S_2 \mid \\
& \qquad \mathsf{if}\,(n)\, S_1 \,\mathsf{else}\, S_2 \mid \mathsf{while}\,(n)\, S_1 \mid \mathsf{let}\, x = E \,\mathsf{in}\, S \\
& \qquad \mathsf{send}\, E_1\, E_2 \mid x := \mathsf{get}\, E
\end{aligned}$$

A program consists of a set of function definitions of the form $f(x)\{S\}$. Functions are assumed for convenience to have one argument; it is easy to extend to the general case.

The statements $\mathsf{send}\, E_1\, E_2$ and $x := \mathsf{get}\, E$ are the XC instructions for sending and receiving data on channels, while the statement $S_1 \parallel S_2$ represents the XC *par* statement for creating concurrent threads. Input and output from and to the external environment (through port I/O for example) are not explicitly considered in the semantics but again it is straightforward to add them. The main restriction is that only integer variables are considered; clearly modelling of arrays will be essential and that is ongoing work.

Note that the statements in the abstract syntax do not directly correspond to the statements found in the XC Programmer's Guide. For instance there are no skip or let commands in XC and there are various assignment operators other than $:=$ such as $=+$, $=-$, and so on. The intention is to keep the semantics as simple as possible and it is assumed that the parser transforms XC statements into the statement forms in the abstract syntax. A detailed justification for the transformations is not given here and it is assumed that any such transformations have negligible impact on energy usage[2].

[2]If it turns out that specific XC statements are compiled to more energy-efficient code than the more generic statements of the abstract syntax, then the abstract syntax and the operational semantics will be extended to handle such statements explicitly. For example, it is possible that $x \;=+E$ can be compiled to more efficient code than $x := x + E$. If so, then the assignment operator $=+$ will be handled explicitly.

9.1.2 Extra Information in the AST

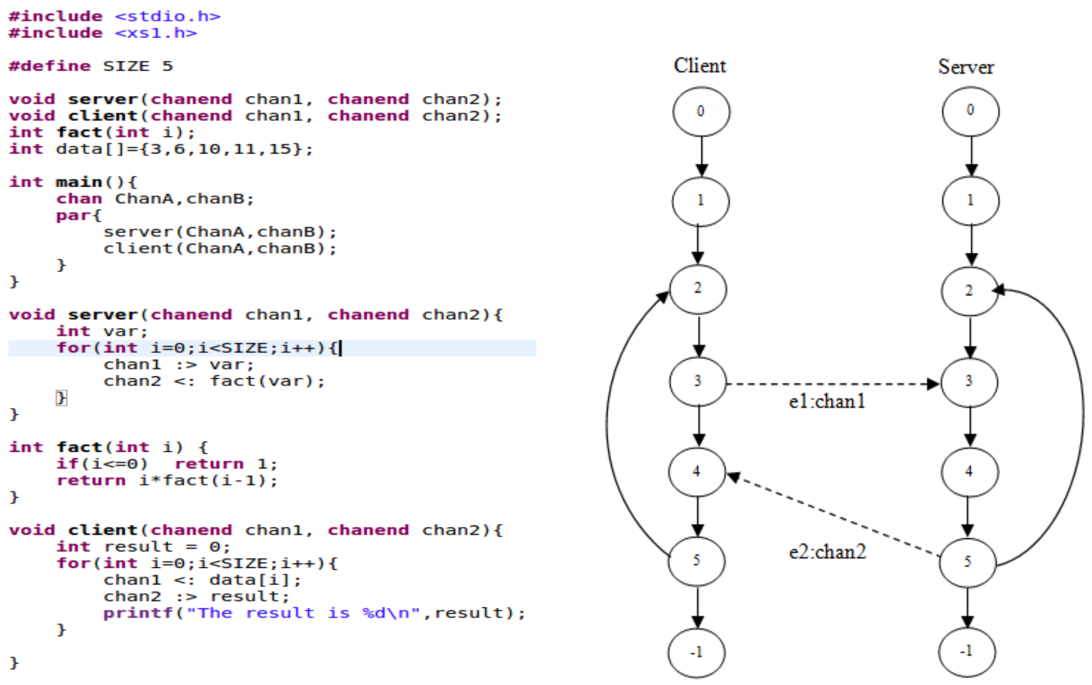
The AST is sufficient for the purposes of program analysis but some information about the concrete syntax, such as line and column numbers, needs to be preserved in order to link analysis results with the source program. It is also necessary to attach labels to some nodes in the AST, for example to identify statements whose execution tally is being kept. In this section this extra information is ignored.

9.1.3 The XCore Thread Execution Model

XC is designed to be executed on XCore processors [Wat09]. The way concurrency is handled in XCore processors is important for the semantics. Each XCore has hardware support for executing a number of concurrent threads. This includes registers for each thread, and a thread scheduler. The processor is implemented using a short pipeline and provides deterministic execution of multiple threads. The threads are executed in a round-robin fashion. The scheduling method used "allows any number of threads to share a single unified memory system and input-output system whilst guaranteeing that with n threads able to execute, each will get at least $1/n$ processor cycles. In fact, it is useful to think of a thread cycle as being n processor cycles. From a software design standpoint, this means that the minimum performance of a thread can be calculated by counting the number of concurrent threads at a specific point in the program" [May13].

Deterministic Thread Semantics. The semantics for thread execution is deterministic, even though the computation of a parallel program may evolve in many different execution orders depending on hardware (for example the same multi-threaded program could be run both on single- and multi-core processors) and the external environment. In this semantics a fixed execution order for threads is chosen. Due to the properties of XC mentioned above, this choice will always give the same final state as any other execution order for a terminating program, and for looping or blocking programs each thread will extend its execution as far as possible. (Note that programs are often perpetual processes; the term "final state" means the state when exiting from a parallel statement).

It would be possible to incorporate a scheduler in the semantics, permitting programs to execute in other orders depending on an extended configuration incorporating (for example) timing information and more detailed modelling of thread behaviour on processor cores. However our working assumption is that the energy consumption of a computation can be reasonably well approximated using the deterministic semantics. The total energy consumed by a set of parallel threads is assumed to be closely related to the sum of the energy consumed by individual threads, regardless of their interleaving. The semantics correctly captures synchronisation; thus informa-



```
#include <stdio.h>
#include <xs1.h>

#define SIZE 5

void server(chanend chan1, chanend chan2);
void client(chanend chan1, chanend chan2);
int fact(int i);
int data[]={3,6,10,11,15};

int main(){
    chan ChanA,chanB;
    par{
        server(ChanA,chanB);
        client(ChanA,chanB);
    }
}

void server(chanend chan1, chanend chan2){
    int var;
    for(int i=0;i<SIZE;i++){
        chan1 :> var;
        chan2 <: fact(var);
    }
}

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

void client(chanend chan1, chanend chan2){
    int result = 0;
    for(int i=0;i<SIZE;i++){
        chan1 <: data[i];
        chan2 :> result;
        printf("The result is %d\n",result);
    }
}
```

Figure 14: An example of an EIAG for the factorial server-client program.

tion can be extracted related to the number of active threads and the patterns of communication among threads, information that can also contribute to more accurate energy estimation.

9.2 Approaches Towards the Analysis of Multi-Threaded Programs

One of the possible approaches for analysing multi-threaded programs is by the use of Event InterAction Graphs (EIAG). EIAG were introduced by Katayama [KFU96], in the context of modelling and analysing multi-threaded, message-passing programs for testing. An EIAG consists of Event Graphs and interactions, where an Event Graph (EG) is a control flow graph of a program unit in a concurrent program (threads in the XCore case), and the interactions represent interactions between the program units. Nodes in the EG, denote concurrent event statements and flow control statements , and edges indicate the transfer of control between nodes. An example of EIAG for the factorial client server program is shown in Figure 14, where the client request the factorial of a number from the server, the server calculates it and then sends back the result to the client. In this graph, the dotted arrows represent the interactions between the server and client thread.

Moving from single-threaded programs to multi-threaded, analysing and modelling gets an extra level of complexity due to the interactions happening between threads. These interactions in the case of XC, constitute the communication of programs on channels. The energy consumption of a program is mainly affected by its communication by two factors. Firstly, energy consumption for the communication of two threads depends on the physical distance between them. Secondly, the synchronization of the communication over a channel will affect the idle time for threads waiting on a communication point, and therefore the overall energy consumption. By companying timing analysis using the EIAG and modelling the pipeline behaviour of the XCore architecture, we believe that it will be possible to statically capture the synchronization behaviour of a communication in most of the cases. Combining this with our energy models and resource analysis framework, possibly energy consumption estimations for multi-thread programs can be retrieved.

During the last months we have developed a method to model and analyse multi-threaded code of XC programs with EIAG. This work is currently under consideration for publication at the International Conference on Software Testing (ICST), as a paper entitled *A Coverage Model to Capture the Communication Behaviour of Multi-Threaded Message-Passing Programs*, and is included in this document as attachment D3.1.5. The techniques developed in this paper can be employed for energy budget verification and energy optimizations in work package 4.

10 Properties used by the State of the Art Energy Optimization Techniques

The analysis framework is flexible and powerful enough to infer a wide range of resources and other (auxiliary/instrumental) program properties. Since we want to focus on those that we foresee to be used in next stages, we have performed an initial study of power-aware software optimization techniques to identify the most promising techniques, and the properties that the analysis should supply to them. Such techniques have been classified into three main groups: compiler level, OS/middleware level and algorithm level. In this section, the techniques that need information from static analyses, as well as these static analyses, are summarized.

10.1 Compiler Optimizations for Low Power

Research on power-aware compiler optimizations is not very extensive, mostly due to the lack of reliable and effective evaluation methods. Thus, an important starting point is to provide ways to evaluate optimization effects.

10.1.1 Power/Energy vs. Performance Optimizations

It has been widely accepted that existing compiler optimizations for improving performance also achieve lower energy consumption given that higher execution time usually means higher energy consumption. This is true for some of them, such as dead code elimination, common subexpression elimination, and in general all those that decrease the amount of workload to be performed. However, there are optimizations where the conclusion about the decrease in energy consumption is not so straightforward.

The work of Kandermir et al al. [KVI02] explores the effect of loop optimization techniques on energy. One possibility presented is to combine loop optimizations, in particular loop fission, with the possibilities offered by the hardware, in this case the existence of different memory banks that have different power modes. In a similar fashion, compilers can help operating systems to decide when to turn on a particular power saving mode by inferring the time during which particular modules are inactive, so they can be turned off: if the operating system knows when particular module is to be used, it can start its activation and deactivation in a timely manner. Example are given in [HPH$^+$02, SCO$^+$pt].

Similarly, when applying Dynamic Voltage and Frequency Scaling (DVFS) optimizations, the compiler can infer the parts of the code for which the processor can be slowed down with negligible performance loss [HK03]. Another example is presented in [SKLil], where the compiler provides an estimation of the execution time of each block. Hence, the static analysis necessary in this case is timing analysis.

10.1.2 Energy Efficient Compiler-based Task and/or Data Parallelization

Nowadays, having multiprocessors or multiple cores on the same chip in practically the standard, which provides the possibility of both task and data parallelization. Along with the possibility of voltage and frequency scaling, as well as turning off unused components, it can bring significant energy savings. Apart form multicore systems, parallelism is also supported in Very Long Instruction Word (VLIW) architectures through Instruction Level Parallelism (ILP) [Azeec], or in Digital Signal Processors (DSP) through ILP or Single Instruction Multiple Data (SIMD) [LMD$^+$04] instruction format.

The connection between task parallelization, voltage and frequency scaling and selective turning off of different components on a multicore chip is investigated in the work of Cho and Melhem [CM10], which derives fundamental formulas to describe the connection between parallelizing an application, its performance and energy consumption.

10.1.3 Summary of Used Static Analysis Techniques

We will now list all the static analyses used by the above mentioned compilers as inputs to the optimization step, or identified as a necessity:

- Energy accounting, i.e. providing insight into the amount of energy a piece of code spends, in order to enable the evaluation of different optimization techniques
- Inferring the time (starting and ending point) the components (e.g. disks) are not active, which can be used by the OS to turn them on and off in a timely manner
- Identify parts of the code the processor can be slowed down with no performance loss, e.g. memory access, as an enabler for voltage and frequency scaling
- Load imbalance analysis, as a special case of the previous item
- Execution time estimation, as another enabler for voltage and frequency scaling
- If there is more than one resource for a certain action, find the shortest path to it (in the terms of energy)

However, none of the cited works mention the usefulness of the independence analysis, which is very important when parallelizing a task. One example of the efficient use of abstract interpretation in automatic parallelization is given in [BGdlBH99], and is implemented in CiaoPP.

10.2 Algorithm and Code Energy-aware Optimizations

10.2.1 Re-computation vs. Communication

Technology scaling has been more beneficial to transistors than to wires. For this reason, communication has become the limiting factor of both power and performance [MG08], especially having in mind its growing level in today's multi-core era. Even the introduction of the Network-on-Chip (NoC) paradigm [BWM$^+$ch] does not solve the problem completely, given the growing trend of communication requirements.

A solution to this problem is to perform re-computation of a code, rather than fetching it from a remote place [MG08]. An important enabler for this approach is to develop greater understanding of algorithms and data structures in order to better manage data movement in systems. Furthermore, it is important to be able to estimate the cost of both computation and communication in order to be able to decide which one is more beneficial in particular cases. The authors believe that the significant work done in VLSI domain in characterizing and predicting interconnections can be helpful in understanding communications in multi-core processors.

10.2.2 Precision (QoS) - Energy Trade-off

In the recent past, researchers have studied energy-accuracy trade-offs [LYeb]. The main conclusion is that a significant part of the energy is spent on providing correctness, while there are many applications that are resilient to errors [LYeb, MSHR10].

This idea is exploited in the design of EnerJ [SDF$^+$11], an extension to Java which provides a solution for isolating precise parts of the program from the ones that can be approximated by introducing approximate types, i.e., type qualifiers declaring that the data can be used in approximate computations. Important contributions to this line of research has been provided by the Computer Science and Artificial Intelligence Laboratory from MIT [MRR11a, MRR11b, ZMKR12, MSHR10, RHMS10].

10.2.3 Summary of Used Static Analysis Techniques

The enabler for all mentioned techniques is again the estimation of the consumed energy. Other static analysis techniques used to enable the code transformation optimization are the following:

- Understanding communication patterns in order to enable communication - re-computation trade-off

- Static energy-bounded scalability analysis, which optimizes performance of parallel algorithms given an energy bound

- Static verification of approximate and precise code computation, in order to enable EnerJ-like applications

- Probabilistic reasoning which justifies that the applied code transformations change the result within given accuracy bounds, in order to enable approximate computations

10.3 OS/Middleware Level Energy Optimizations

Operating Systems (OS) have many aspects that can provide energy savings. For example, Dynamic Power Management (DPM), whose most logical implementation is at OS level since OS has the ultimate control of all computational, storage and I/O system operations. Other important OS aspect is task scheduling, coupled with the hardware possibility of voltage and frequency scaling, which can minimize energy consumption taking advantage of lower requirements of particular implementation. In the context of the ENTRA project, we have studied DVFS techniques and the feasibility and effectiveness of their application to the XMOS architecture. This work has been published in [BLG13], and is included in this document as attachment D3.1.4. The necessary information for these techniques has already been mentioned in Section 10.1.3.

References

[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.

[Azeec] Naeem Zafar Azeemi. Exploiting parallelism for energy efficient source code high performance computing. In *Industrial Technology, 2006. ICIT 2006. IEEE International Conference on*, pages 2741–2746, Dec.

[BGdlBH99] Francisco Bueno, María García de la Banda, and Manuel Hermenegildo. Effectivness of abstract interpretation in automatic parallelization: a case study in logic programming. *ACM Trans. Program. Lang. Syst.*, 21(2):189–239, March 1999.

[BHZ08] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[BK96] F. Benoy and A. King. Inferring argument size relationships with CLP(R). In John P. Gallagher, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'96)*, volume 1207, pages 204–223, August 1996.

[BLG13] Z. Banković and P. López-García. Genetic Algorithm-based Allocation and Scheduling for Voltage and Frequency Scalable XMOS Chips. In Jeng-Shyang Pan, MariosM. Polycarpou, Micha Woniak, AndrC.P.L.F. Carvalho, Hctor Quintin, and Emilio Corchado, editors, *Hybrid Artificial Intelligent Systems*, volume 8073 of *Lecture Notes in Computer Science*, pages 401–410. Springer, 2013.

[BLGH04] F. Bueno, P. López-García, and M. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[Bru91] Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *J. Log. Program.*, 10(2):91–124, 1991.

[BW11] A.E. Brown and G. Wilson. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. The Achrictecture of Open Source Applications. CreativeCommons, 2011.

[BWM$^+$ch] A. Banerjee, P.T. Wolkotte, R.D. Mullins, S.W. Moore, and G. J M Smit. An energy and performance exploration of network-on-chip architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(3):319–329, March.

[CC77a] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *POPL*, pages 238–252. ACM, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *POPL*, pages 269–282. ACM Press, 1979.

[CGJ$^+$00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[CM10] Sangyeun Cho and Rami G. Melhem. On the interplay of parallelization, program performance, and energy consumption. *IEEE Trans. Parallel Distrib. Syst.*, 21(3):342–353, March 2010.

[Cou99] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[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.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[dwa13] org dwarfstd. The dwarf debugging standard, October 2013.

[Gal93] J. P. Gallagher. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, June 1993. ACM Press.

[GBN05] Alan Burns Guillem Bernat and Martin Newby. Probabilistic timing analysis: An approach using copulas. *J. Embedded Computing*, 1(2):179–194, 2005.

[GLPR12] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, pages 405–416. ACM, 2012.

[HAH12] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.

[HBC$^+$12a] 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.

[HBC$^+$12b] 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. *The-*

ory and Practice of Logic Programming, 12(1–2):219–252, January 2012. http://arxiv.org/abs/1102.5497.

[HBG07] Kim S. Henriksen, Gourinath Banda, and John P. Gallagher. Experiments with a convex polyhedral analysis tool for logic programs. In *Workshop on Logic Programming Environments, Porto, 2007*, 2007.

[HK03] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. *SIGPLAN Not.*, 38(5):38–48, May 2003.

[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.

[HPH$^+$02] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini. Application transformations for energy and performance-aware device management. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 121–130, 2002.

[JGS93] N. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Software Generation*. Prentice Hall, 1993.

[KFU96] T. Katayama, Z. Furukawa, and K. Ushijima. A method for structural testing of Ada concurrent programs using the event interactions graph. In *Proceedings of the Third Asia-Pacific Software Engineering Conference*, pages 335 – 364, 1996.

[KVI02] Mahmut Kandemir, N. Vijaykrishnan, and Mary Jane Irwin. Compiler optimizations for low power systems. In Robert Graybill and Rami Melhem, editors, *Power aware computing*, pages 191–210. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[LA04] C. Lattner and V.S. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, 2004.

[LGDB10] P. López-García, L. Darmawan, and F. Bueno. A Framework for Verification and Debugging of Resource Usage Properties. In M. Hermenegildo and T. Schaub, editors, *Technical Communications of the 26th Int'l. Conference on Logic Programming (ICLP'10)*, volume 7 of *Leibniz International Proceedings in Informat-*

ics (LIPIcs), pages 104–113, Dagstuhl, Germany, July 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[LJ99] M. Leuschel and J. Jørgensen. Efficient specialisation in Prolog using the hand-written compiler generator LOGEN. *Elec. Notes Theor. Comp. Sci.*, 30(2), 1999.

[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 the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, September 2013.

[LMD$^+$04] Markus Lorenz, Peter Marwedel, Thorsten Dräger, Gerhard Fettweis, and Rainer Leupers. Compiler based exploration of dsp energy savings by simd operations. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 838–841, Piscataway, NJ, USA, 2004. IEEE Press.

[LYeb] Xuanhua Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 181–192, Feb.

[May13] D. May. The XMOS XS1 architecture. available online: http://www.xmos.com/published/xmos-xs1-architecture, 2013.

[MG08] Simon Moore and Daniel Greenfield. The next resource war: computation vs. communication. In *Proceedings of the 2008 international workshop on System level interconnect prediction*, SLIP '08, pages 81–86, New York, NY, USA, 2008. ACM.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[MRR11a] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistic and statistical analysis of perforated patterns. Technical Report MIT-CSAIL-TR-2011-003, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 2011.

[MRR11b] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistically accurate program transformations. In *Proceedings of the 18th international conference* *on Static analysis*, SAS'11, pages 316–333, Berlin, Heidelberg, 2011. Springer-Verlag.

[MSHR10] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 25–34, New York, NY, USA, 2010. ACM.

[Nel03] Roger B. Nelsen. Properties and applications of copulas: A brief survey. In *First Brazilian Conference on Statistical Modelling in Insurance and Finance*, pages 10–28, 2003.

[NMLGH07] 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.

[NNH99] F. Nielson, HR. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[OOS13] Yarema Okhrin Ostap Okhrin and Wolfgang Schmid. Properties of hierarchical archimedean copulas. *Statistics & Risk Modeling*, 30(1):21–54, 2013.

[PP99] Alberto Pettorossi and Maurizio Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *J. Log. Program.*, 41(2-3):197–230, 1999.

[RHMS10] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. *SIGPLAN Not.*, 45(10):806–821, October 2010.

[SCO$^+$pt] Seung Woo Son, Guangyu Chen, O. Ozturk, M. Kandemir, and A. Choudhary. Compiler-directed energy optimization for parallel disk based systems. *Parallel and Distributed Systems, IEEE Transactions on*, 18(9):1241–1257, Sept.

[SDF$^+$11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. *SIGPLAN Not.*, 46(6):164–174, June 2011.

[SKLil] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *Design Test of Computers, IEEE*, 18(2):20–30, March-April.

[SLGBH13] A. Serrano, P. Lopez-Garcia, F. Bueno, and M. Hermenegildo. Sized Type Analysis for Logic Programs (technical communication). *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement*, August 2013. To Appear.

[SLGH13] Alejandro Serrano, Pedro López-García, and Manuel V. Hermenegildo. Towards an abstract domain for resource analysis of logic programs using sized types. *CoRR*, abs/1308.3940, 2013.

[TMWL96] 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.

[VB02] C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.

[Wat09] Douglas Watt. *Programming XC on XMOS Devices*. XMOS Ltd., 2009.

[ZMKR12] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. *SIGPLAN Not.*, 47(1):441–454, January 2012.

[ZNMZ12] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 427–440, New York, NY, USA, 2012. ACM.

Attachments

Attachment D3.1.1

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

Published at the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)

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

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

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

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

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

1 Introduction

Energy consumption and the environmental impact of computing technologies are a major focus. Despite advances in power-efficient hardware, more energy savings can be achieved by improving the way current software technologies make use of such hardware. Many optimization techniques that can be used for producing energy-efficient software need estimations of the energy consumption of software segments prior to their execution, in order to make decisions about the optimal way of executing them. These a priori estimations are also very useful to software engineers to better understand the effect of their designs on the energy consumption early on during the software development process, and make more informed design decisions (e.g., using the appropriate data structures), even when there are parts not developed yet.

In this paper we combine static analysis and low level energy modelling techniques to implement a tool capable of estimating the energy consumption of an embedded program (and its constituent parts, such as procedures and functions) as a function on several parameters of the input data (e.g., sizes), and the hardware platform where they are executed (e.g., clock frequency and voltage). We show the feasibility of our proposal with a concrete case study: analysis of ISA (Instruction Set Architecture) code compiled from XC [17]. XC is a high-level C-based programming language that includes extensions for concurrency, communication, input/output operations, and real-time behavior. XC libraries share a common API with standard C libraries and therefore C code can commingle with XC code in a single application.

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

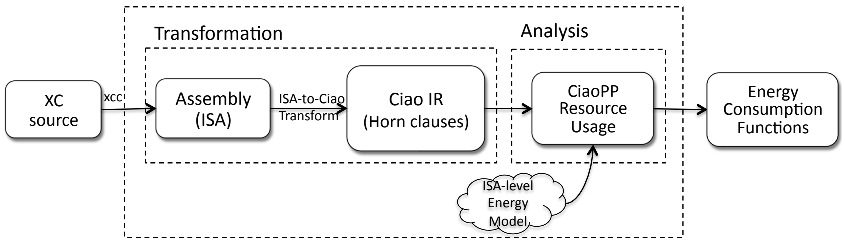


Fig. 1. Overview of the analysis framework for XC programs.

Our approach leverages the CiaoPP tool [4], the preprocessor of the Ciao programming environment [5]. CiaoPP includes a parametric analysis framework for resource usage that can be instantiated to infer bounds on resources of interest, energy consumption in our case. 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 express the resource usage of elementary program operations, certain program constructs, and library procedures. Based on this information, the analyser can infer upper and lower bounds on the resource usage of the whole program. This CiaoPP analysis works on an intermediate block-based representation language, which we call the Ciao IR. Each block is represented as a Horn clause, so that, in essence, the Ciao IR is a logic programming subset of the Ciao language. To this end we propose a transformation of the ISA program into Ciao IR (containing Horn clauses and assertions), which allows us to analyse the transformed program with CiaoPP. The procedural interpretation of these Ciao IR programs, coupled with resource-related information contained in the assertions (such as the energy consumption models at the ISA level), allow the resource analysis to infer static bounds on the energy consumption of the Ciao programs that are applicable to the original ISA programs.

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

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

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

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

1. Production of a low-level energy consumption model at the ISA level for our case study architecture (XMOS XS1-L) that is also appropriate for the high-level cost analysis tools.

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



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

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


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

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

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

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

Fig. 3. An ISA (factorial) program (left side) and its Ciao IR (right side).

2. Design and implementation of a translation from ISA programs into a Horn clause representation (Ciao IR).
3. Instantiation of the CiaoPP general resource analysis framework to infer energy consumption using the low-level energy consumption model.
4. Overall design and implementation of a fully automatic system that statically estimates the energy consumption of functions and procedures written in a high-level C-based programming language, giving the results as functions on input data sizes.
5. Experimental assessment of the developed energy consumption static analyzer.

Point 4 above may look simple at first sight, given that we have taken advantage of a number of existing tools, mainly the CiaoPP general resource analyzer. However, in practice the implementation has required the implementation of new modules and functionalities, as well as interfaces between these existing tools, all of which posed some design and implementation challenges and problems that we have successfully solved. For instance, we have improved and extended CiaoPP to deal with new types of Ciao IR programs coming from the translation of the generated ISA programs.

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

2 Energy Characterization and Modelling

The assertion-based model utilises power consumption data collected during hardware measurement. We have developed an ISA-level model that provides software energy consumption estimates based on Instruction Set Simulation (ISS) statistics. The hardware, the measurement process, as well as the construction of the ISS-driven model, are detailed in [8], with the key components relevant to this paper explained in the rest of this section.

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

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

2.1 Energy Profiling Framework and Strategy

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

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

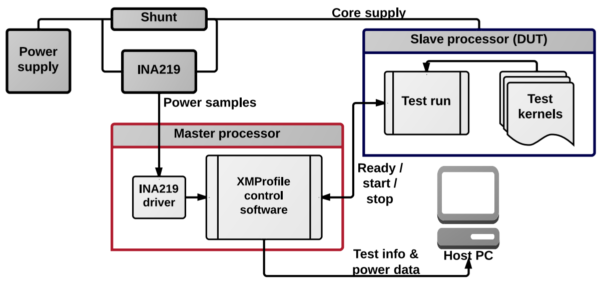


Fig. 4. Overview of test harness hardware and software structure, with a slave processor executing test kernels and a master processor collecting power samples.

2.2 ISA-level Model

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

Our model is based on that devised by Tiwari [16]. Tiwari's approach is shown in Equation 1. The energy of an ISA program, E_p, is characterised as the sum of base energy cost, B_i, for all ISA instructions, i, multiplied by the number of executions of each instruction, N_i. An inter-instruction overhead energy, $O_{i,j}$, is then accounted for by enumerating for all instruction combinations i, j and their frequency, $N_{i,j}$. Finally, additional contributions to program energy can be accounted for by k external effects, E_k, which may include externally modelled behaviours such as cache memory.

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

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

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

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

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

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

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

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

3 Transforming ISA Programs into Ciao IR Programs

In this section we describe the transformation from ISA programs into the Ciao intermediate representation (Ciao IR) mentioned in Such representation consists of a sequence of *blocks* (as in Figure 3). Each block is represented as a *Horn clause*:

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

which has an entry point, that we call the *head* of the block (to the left of the $:-$ symbol), including a number of parameters $< params >$, and a sequence of steps (the *body*, to the right of the $:-$ symbol), each of which is either, (the representation of) an ISA *instruction*, or a *call* to another (or the same) block. The analyzer deals with the Ciao 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 Ciao IR programs inferred by the resource analysis are applicable to the original ISA programs.

ISA programs are expressed using the XS1 instruction set [10]. The transformation framework currently works on a subset of this instruction set. The ISA program is parsed and a control flow analysis is carried out, yielding an inter-procedural control flow graph (CFG). This process starts by identifying control transfer instructions such as branch or call instructions. Basic blocks are then constructed, which are annotated with input/output arguments and transformed into Static Single Assignment (SSA) form. Finally, the target Ciao IR (i.e., Horn clauses) is emitted.

A basic block over a CFG is a maximal sequence of distinct instructions, S_1 through S_n, such that all instructions $S_k, 1 < k < n$ have exactly one in-edge and one out-edge (excluding call/return edges), S_1 has one out-edge, and S_n has one in-edge. A basic block therefore has exactly one entry point at S_1 and one exit point at S_n. All call instructions are assumed to eventually return. Using the basic block definition a block control flow graph is constructed by the analyser, where each node represents a block. Edges between the blocks are derived from calls/jumps between blocks. This process involves iterating through the CFG of the ISA program and marking block boundaries, which are instructions that either begin or end a basic block.

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

$$\begin{aligned} params_{out}(b) &= kill(b) \cup \textstyle\bigcup_{b' \in next(b)} params_{out}(b') \\ params_{in}(b) &= gen(b) \cup \textstyle\bigcup_{b' \in next(b)} params_{in}(b') \end{aligned}$$

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

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

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

Our approach here is closely related to that of the live variable analysis (LVA) [13] used in compilers, and in dead code elimination in particular. A variable is live at a program point if it may get referenced later in the program (which is decided by considering the whole CFG of the program). In LVA, for each program point, a set of live variables is computed using functions similar to our *kill* and *gen* functions with data flow equations. In our approach however, instead of computing liveness information for each program point, we compute a least fixpoint of our $params_{out}$ and $params_{in}$ functions over the program's block control flow graph. This is an efficient solution that safely over-approximates the set of input/output arguments to each block, so that the extra arguments inferred for block heads due to such over-approximation do not affect the energy consumption estimations, since they are not used in the analysis of procedures corresponding to the original XC code.

Resolving Branching to Multiple Blocks. In the XS1 instruction set, conditional branch instructions (e.g., `bt`, `bf`) jump to one of the two target blocks based on the value of the branching variable. For example, in Figure 3, at line 7 the `bf` instruction (branch if fail) will jump to address `008` if $r0 = 0$, otherwise to address `007`. In the Ciao IR this branch needs to be a call to one of the two blocks.

We use a similar approach to the one described in [11] to resolve branches to multiple blocks. The multiple target blocks of a jump instruction are assigned the same head, which essentially are clauses of the same Ciao IR predicate. This is achieved by merging the heads of the target clauses so that each clause has the same head. The algorithm is trivial, since we have already inferred the input/output parameters to each block's head. The input/output parameters to the new head of the clauses are the union of the input/output parameters of all the clauses along with the branching variable. This enables preservation of the branching semantics of the original ISA program in Ciao IR form.

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

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

In compilers, the SSA form is generated at the function level (e.g., at LLVM [9] level) where a function might consist of multiple basic blocks. However, we follow the approach of generating SSA form at the block level, and therefore we do not need to generate ϕ nodes. A ϕ node is an instruction used to select a version of the variable depending on the predecessor of the current block. Since each block is already annotated with input/output arguments, any predecessor block will pass the appropriate values as input parameters when making a call to the target block.

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

Analysis on low level (ISA) representations, in general, suffers from the problem of extracting a precise control flow graph in the presence of indirect jumps and calls. The current implementation of our transformation is restricted to direct jumps and calls. We plan to integrate other techniques into the transformation tool to resolve such problems including recognizing code patterns used by compilers and performing static program analysis (see [19] and its references).

4 General Analysis Framework

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

CiaoPP includes a global static analyser which is parametric with respect to resources and type of approximation (lower and upper bounds) [12]. The user can define the parameters of the analysis for a particular resource by means of assertions that associate basic cost functions with elementary operations of the base language and procedures in libraries, thus expressing how they affect the usage of a particular resource. The global static analysis can then infer bounds on the resource usage of all the procedures in the program, as functions of input data sizes. Examples of resources that can be analysed by instantiating the CiaoPP general framework include execution steps, execution time, number of accesses to a database, number of bytes sent or received through a socket, etc.

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

4.1 Instantiating the General Framework

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

```
:- resource energy.
```

Expressing Energy Models and Resource Usage of Library Functions. The resource usage of Ciao library predicates is expressed using "trust" assertions (see [5] and its references for a description of the Ciao assertion language). For example, we can write assertions for each Ciao predicate that represents an ISA instruction; these constitute the energy models. The following assertions (for the `add` and `sub` instructions) are part of the simple energy model that we used in the static analysis, which assigns a constant energy amount to each ISA instruction:

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

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

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

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

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

4.2 Performing the Analysis

Once the parameters of the general resource analysis framework have been defined, and assertions for library predicates (including the ones representing energy models) have been provided, the CiaoPP global static analysis can infer the resource usage of all the procedures in the program (as functions of input data sizes). A full description of how this is done can be found in [12].

Mode Analysis. In general, mode analysis determines, for each argument in each predicate in the block representation (Ciao IR), whether it acts as an input or an output argument. In our context mode analysis is not needed for any predicate. The modes of some predicates can be extracted from the XC source code that the Ciao IR is originated from. This is possible because mode (and type) information is statically known at the XC language level and is propagated to the Ciao IR (and hence to the mode analyzer) using (trust) assertions. This means that the analyzer will just trust such mode information and use it without performing any inference process for the predicates that have an associated assertion containing modes. The rest of the predicates are new predicates generated by the transformation from ISA programs into Ciao IR (described in Section 3), and originated from conditional branching, which cannot be directly related to the XC source code, and, thus, do not have any associated assertion. For such predicates, CiaoPP uses the results from the transformation phase where the input/output arguments are inferred for each predicate.

Size Measure Analysis. CiaoPP uses type information to decide which metric to use to express data sizes from a set of predefined metrics (e.g., the value of an integer, *int*, or the depth of a term, *depth*).

Type analysis is needed because most of that information is lost in the conversion to assembly. CiaoPP is able to reconstruct almost all that information. Analysis of the Ciao IR program in Figure 3 (right hand side) infers that `fact` will be called with $R0$ bound to an integer and $R0\_3$ a free variable, and will succeed with $R0\_3$ bound to an integer. Also, `fact_aux` will be called with the first two arguments bound to integers, and the rest free, and, upon success, all of them will be bound to integers. Given that information, the chosen metric for all the arguments will be *int*.

Size Analysis. It determines the relative sizes of variable bindings at different program points. The size analysis (as well as the resource usage analysis) is performed for each strongly-connected component of the control flow graph of the program in reverse topological order. For each clause, size relations are propagated to express each output data size as a function of input data sizes. For recursive functions this is done symbolically, creating a set of recurrence relations that will be solved to get a closed form function.

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

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

These inferred recurrence relations/equations are then fed into a computer algebra system (Mathematica, in the implementation developed in this paper) that gives the following closed form function for it: $fact_{R0\_3}(R0) = R0!$

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

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

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

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

Note also that using average values in the model implies that the energy function for the whole program inferred by the upper-bound resource analysis is an approximation of the actual upper bound that can possibly be below it. To ensure that the analysis infers a strict upper bound, we would need to use strict upper bounds as well in the energy models. However, with the current models such bounds would be very conservative, causing a loss in accuracy that would make the analysis not useful in practice. Thus, the current approach is a practical compromise.

5 Benchmarks, Results and Evaluation

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

Benchmarks. For this type of evaluation we use as benchmarks mainly small mathematical functions. The structure of these programs is either iterative or recursive, with their cost depending on the function argument. For such programs state of the art solvers can easily provide the cost functions, by solving the system of recurrence relations provided by the SRA framework. Table 5 shows the benchmarks used in this comparison, their execution behaviour in relation to each function's parameters, and the method by which their cost function was calculated. Also, some hand-solved examples have been provided in addition to those that were solved using SRA, both to compare a manual solution to SRA and to provide an additional set of data points that will, in the future, be solved automatically.

Experimental method. Hardware energy readings were obtained by repeatedly executing a benchmark function over a 0.5 second period, T, collecting a set of power samples, P, whilst counting the number of executions, N_{fn}. From this, the energy of a single function call, $E_{\text{fn}} = \frac{\text{mean}(P) \times T}{N_{\text{fn}}}$ is calculated. This

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

Function name	Description	Energy function	Calculation
`fact(N)`	Calculates $N!$	$26.0\ N + 19.4$	SRA
`fibonacci(N)`	Nth Fibonacci no.	$30.1 + 35.6\ \phi^N + 11.0\ (1-\phi)^N$	
`sqr(N)`	Computes N^2	$103.0\ N^2 + 205.8\ N + 188.32$	
`poweroftwo(N)`	Calculates 2^N	$62.4 \cdot 2^N - 312.3$	
`sumofdigits(N)`	Adds all digits in N	$84.4\lceil\log_{10} N\rceil - 78.7$	By hand
`isprime(N)`	Checks if N is prime	$58.6\ N - 35.5$	
`power(base,exp)`	Calculates $base^{exp}$	$6.3\ (\log_2 exp + 1) + 6.5$	

was performed using a similar method to the collection of energy model data described in Section 2, but was performed on separate hardware so as to de-couple modelling from testing.

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

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

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

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

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

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

the error begins to increase markedly. The `power` function behaves in a similar way and demonstrates the relationship between multiple input arguments.

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

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

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

6 Related Work

Static analysis of energy consumption is still an emerging research field. The worst-case analysis presented in [7] distinguishes instruction-specific (not proportional to time, but to data) from pipeline-specific (roughly proportional to time) energy consumption.

A timing analysis based on game-theoretic learning is presented in [15]. The apprach 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.

The approach we have followed in this paper, as well as the one in [1], based on recurrence relations, derives from the seminal works on time analysis of [18] and [2]. A general framework to deal with user-defined resources was proposed later in [12]. A different approach is based on the potential method, such as in [6],

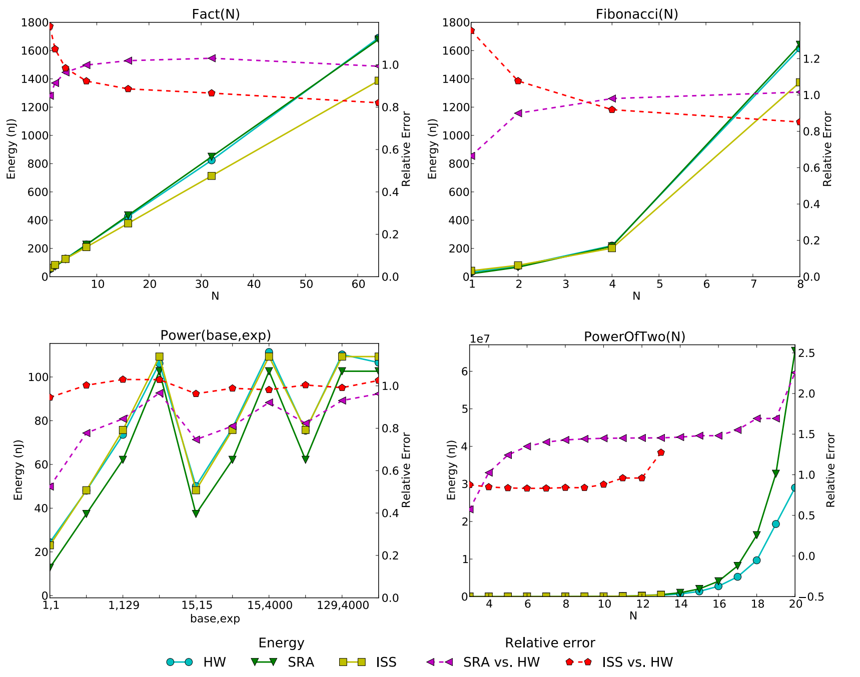


Fig. 5. Hardware energy, estimations and relative errors for (starting top-left, moving clock-wise) `fact`, `fibonacci`, `poweroftwo` and `power`.

which is based on a type-and-effect system. However, it is limited to polynomial bounds, which do not allow expressing some non-polynomial energy functions, as the ones we show in the experimental results table.

Transformation-based frameworks for program analysis that analyse low level microprocessor code [3] and Java source and bytecode [1, 11] have been proposed. In [1], cost relations are inferred directly for the bytecode programs, whereas in [11] the bytecode is first transformed into a Ciao program. Our transformation framework is closely related to [11] where the Jimple (a typed three-address code) representation of Java bytecode is transformed into Ciao. However, unlike Jimple, we employ transformations at lower level (XS1-ISA), irrespective of source language in general, where much of the program structure and typing information is trimmed away. Our transformation employs analysis techniques to reverse engineer ISA programs, which requires control flow graph reconstruction and transformation into an equivalent Ciao IR that safely approximates the semantics of the original ISA program.

Instruction Set Simulation can be used to estimate the energy of a program running on a suitably profiled hardware platform. Simple models for single-threaded architectures have been demonstrated [16]. These have then been expanded upon, leading to models capable of modelling more complex hardware such as that used in this paper, which comprise a multi-threaded architecture [8].

7 Conclusions and Future Work

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

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

We also plan to produce and deal with energy models that take into account the switching cost among pairs of ISA instructions (i.e., the energy consumed by bit flipping), since our analysis framework allows it. The improvement in accuracy from this approach can vary between architectures, for example research such as [14], shows that a simple model can be sufficient in some cases, due to bit flipping effects averaging out over time. Thus, the impact in the context of any target architectures must therefore be considered in this future work, in order to establish whether the increased complexity of analysis delivers a worthwhile gain in accuracy.

Our analysis accuracy can also be further improved by propagating high-level program information such as types to the lower-level representations. We also intend to improve upon the energy measurements of commonly used instructions, which involves more complex techniques such as linear regression. This technique can also be used to construct energy models of intermediate compiler representations such as LLVM IR [9], which would enable us to apply our analysis techniques to more structured program representations. Another method for analysing LLVM IR would involve mapping low-level program instruction segments to LLVM IR segments and reusing the energy models at ISA level.

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

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.
2. S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
3. 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.
4. 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.
5. 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.
6. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.
7. 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.
8. 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.
9. C. Lattner and V.S. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, 2004.
10. D. May. The xmos xs1 architecture. available online: http://www.xmos.com/published/xmos-xs1-architecture.
11. M. Mendez-Lojo, J.A. Navas, and M.V. Hermenegildo. A flexible, (c)lp-based approach to the analysis of object-oriented programs. In *LOPSTR*, pages 154–168, 2007.
12. 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.
13. F. Nielson, HR. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
14. J. T. Russell and M. F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *ICCD*, pages 328–333, 1998.
15. S. A. Seshia and J. Kotker. Gametime: A toolkit for timing analysis of software. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 388–392. Springer, 2011.
16. 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.
17. D. Watt. *Programming XC on XMOS Devices*. XMOS Limited, 2009.
18. B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.
19. L. Xu, F. Sun, and Z. Su. Constructing Precise Control Flow Graphs from Binaries. *University of California, Davis, Tech. Rep*, 2009.

Attachment D3.1.2

Sized Type Analysis for Logic Programs

Published as a technical communication in Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13), Special Issue, On-line Supplement.

Sized Type Analysis for Logic Programs

A. SERRANO[1] P. LOPEZ-GARCIA[1,2] F. BUENO[3] M. V. HERMENEGILDO[1,3] *

[1] *IMDEA Software Institute*
(*e-mail:* alejandro.serrano@imdea.org, pedro.lopez@imdea.org, manuel.hermenegildo@imdea.org)
[2] *Spanish Council for Scientific Research (CSIC)*
[3] *Universidad Politécnica de Madrid (UPM)*
(*e-mail:* bueno@fi.upm.es, herme@fi.upm.es)

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

Abstract

We present a novel analysis for relating the sizes of terms and subterms occurring at different argument positions in logic predicates. We extend and enrich the concept of *sized type* as a representation that incorporates structural (shape) information and allows expressing both lower and upper bounds on the size of a set of terms and their subterms at any position and depth. For example, expressing bounds on the length of lists of numbers, together with bounds on the values of all of their elements. The analysis is developed using abstract interpretation and the novel abstract operations are based on setting up and solving recurrence relations between sized types. It has been integrated, together with novel resource usage and cardinality analyses, in the abstract interpretation framework in the Ciao preprocessor, CiaoPP, in order to assess both the accuracy of the new size analysis and its usefulness in the resource usage estimation application. We show that the proposed sized types are a substantial improvement over the previous size analyses present in CiaoPP, and also benefit the resource analysis considerably, allowing the inference of equal or better bounds than comparable state of the art systems.

1 Introduction

Size analysis is the process of assigning numerical metrics to terms appearing in a program and estimating bounds for these metrics. Such analysis is useful on its own as a source of information for the developer, and it is also often instrumental to other analyses. For example, the consumption of resources, such as memory or time, by a program is usually expressed in terms of the sizes of its arguments. In this paper we focus on the size analysis of Prolog terms. 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. There, the size analysis is the first of several other analysis steps ultimately arriving at cost bounds. An important limitation of that analysis is that it is only able to cope with size information about subterms in a very limited way. However, dealing fully with subterms is in fact a key issue in the cost analysis of realistic programs. For example, consider a predicate which computes the factorials of a list:

* 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*.

```
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 over a list L is $\alpha + \sum_{e \in L} (\beta + \mathit{time}_{fact}(e))$ where α and β are constants related to the unification and calling costs. However, 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 ∞.

Several authors have worked to overcome this limitation. In (Hoffmann et al. 2012) a system is proposed which is able to analyze `listfact`. This is done within the framework of amortized analysis with the potential method, enriched with fixed polynomials relating the cost with sizes of contained elements. However, polynomials are not enough for expressing some kinds of bounds, especially exponential ones.

In (Vasconcelos and Hammond 2003) the authors introduce the idea of *sized types* to directly represent information about the upper bounds on sizes within a Hindley-Milner type system, for functional programs. Our proposal of *sized types* is related to this idea, but differs from it in several significant ways:

- We incorporate structural (shape) information expressing *both lower and upper bounds* on the sizes of a set of terms and their subterms, *at any depth*.
- We focus on *logic programming*, which includes features such as non-determinism and creation of terms without previously having to define the constructors involved.
- Instead of a Hindley-Milner type system, we use *regular types* (Dart and Zobel 1992) as the base for sized types. Regular types are structural instead of nominal, and there is a notion of subtyping based on inclusion, important differences that the analysis must handle. Furthermore, the sized types are *automatically derived*.
- We develop the analysis as an *abstract interpretation* instead of a type and effect system. To our knowledge, this is the first time a recurrence-based analysis is developed entirely as an abstract domain. Using abstract interpretation enables us to integrate the analysis in a standard engine (in our case PLAI within the CiaoPP analysis framework), which brings in features such as *multivariance*, accelerated fixpoint computation, and assertion-based verification and user interaction for free.
- (Vasconcelos and Hammond 2003) allows assigning costs to higher-order functions based on the cost of other functions. Our system does not yet allow this, but we believe the extension is not complex.

2 Overview of the Approach

We show the different ideas in our proposal using the classical `append` predicate:

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

In a first phase we infer types for the predicate arguments by using an existing analysis for regular types (Vaucheret and Bueno 2002). This analysis infers for instance that if we call `append(X, Y, Z)` with `X` and `Y` bound to lists of numbers and `Z` a free variable, then `Z` gets bound to a list of numbers upon success.

Even more importantly, the definition of the *inferred* regular type is the following:

```
listnum -> [] | .(num, listnum)
```

From this inferred definition, or any other expressed as a regular type, we derive the *schema* of the corresponding sized type. Such sized types represent the size of a particular term, i.e., in our case, the sized type *listnum-s*:

$$listnum\text{-}s \rightarrow listnum^{(\alpha,\beta)}(num^{(\gamma,\delta)}_{\langle .,1\rangle})$$

represents that the list has between α and β elements which are numbers between γ and δ. The $\langle .,1\rangle$ below *num* expresses that this inner size description applies to subterms occurring at the first parameter of the `./2` functor.

The next phase involves relating the sized types of the different arguments to the predicate using recurrences. Let[1] $size_X = ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}_{\langle .,1\rangle})$ be the sized type of a list X of numbers. Assume a call `append(X, Y, Z)`. The inequations for the lower bound on the length of the output argument Z, denoted α_Z, as a function on input data sizes are:

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

The whole set of inequations defining all bounds on a sized type is too large. Thus, we aim for a more compact representation. Our proposal is to write parameters directly as sized types and group all inequalities (both upper and lower bounds) on a single type in a expression. We decided to use the symbol $\lesseqgtr$ to mean that both types of inequalities are represented. For example:

$$ln^{(a_1,b_1)}(n^{(c_1,d_1)}) \lesseqgtr ln^{(a_2,b_2)}(n^{(c_2,d_2)}) \iff a_1 \geq a_2, b_1 \leq b_2, c_1 \geq c_2, d_1 \leq d_2$$

Using this syntax, the tightest bounds on the entire recurrence relation are:

$$size_Z\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}_{\langle .,1\rangle}), ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)}_{\langle .,1\rangle})\right) \lesseqgtr ln^{(\alpha_X+\alpha_Y,\beta_X+\beta_Y)}(n^{(\min(\gamma_X,\gamma_Y),\max(\delta_X,\delta_Y))}_{\langle .,1\rangle})$$

3 Sized Types

As shown in the `append` example, the variables we relate in our inequations come from sized types. *Sized types* are representations for summarizing the size of a set of terms, close to those found in (Hughes et al. 1996) for functional languages. In our approach, *sized types* schemas are automatically built from automatically inferred regular types by analyses present in the CiaoPP system (Vaucheret and Bueno 2002). 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* α_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 $\tau \rightarrow \phi$, where τ is a *type symbol* and ϕ a *type term.* A *regular term grammar* Υ is a set of *type rules.*

In this paper, we introduce the concept of *sized type* as an abstraction of a set of Herbrand terms that: 1) are a subset of a set abstracted by some regular type τ, and 2) meet

[1] In the examples we will use *ln* and *n* instead of *listnum* and *num* for the sake of conciseness.

$$
\begin{aligned}
\gamma\left(num^{(\alpha,\beta)}\right) &= \{n \in \mathbb{Z} : \alpha \le n \le \beta\} \\
\gamma\left(\tau^{(\alpha,\beta)}(\bar{x})\right) &= \bigcup_{\alpha \le s \le \beta} \gamma_{exact}\left(\tau^{s}(\bar{x})\right), \qquad \text{if } \tau \text{ is recursive} \\
\gamma\left(\tau(\bar{x})\right) &= \gamma_{exact}\left(\tau^{1}(\bar{x})\right), \qquad \text{if } \tau \text{ is not recursive} \\
\gamma\left(\tau^{nob}(\bar{x})\right) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\gamma_{exact}\left(\tau^{0}(\bar{x})\right) &= \emptyset \\
\gamma_{exact}\left(\tau^{s}(\bar{x})\right) &= \bigcup_{\tau \to \phi \in \Phi} \gamma_{rule}(\phi, \bar{x}, \tau^{s}), \quad s > 0
\end{aligned}
$$

$$
\begin{aligned}
\gamma_{rule}\left(\sigma, d, \tau^{s}\right) &= \gamma(d), \qquad \text{if } \sigma \text{ is a type symbol} \\
\gamma_{rule}\left(f(\sigma_1, \dots, \sigma_n), \bar{x}, \tau^{s}\right) &= \left\{f(y_1, \dots, y_n) : \textstyle\sum a_i = s - 1\right\}, \qquad f \text{ functor} \\
\text{where} \qquad y_i &= \begin{cases} \gamma(d), & \sigma_i \ne \tau \text{ and } d_{\langle f,i\rangle} \in \bar{x} \\ \gamma_{exact}\left(\tau^{a_i}(\bar{x})\right), & \sigma_i = \tau \end{cases}
\end{aligned}
$$

Fig. 1. Concretization function γ for sized types.

some lower- and upper-bound size constraints on the number of *type rule applications* (or other metrics for base types). A grammar for the these sized types follows:

sized-type	::=	α^{bounds}	α base type
	$\mid$	$\tau^{bounds}(\textit{sized-args})$	τ recursive type symbol
	$\mid$	$\tau(\textit{sized-args})$	τ non recursive type symbol
bounds	::=	*nob* $\mid$ (n, m)	$n, m \in \mathbb{N}, m \ge n$
sized-args	::=	ϵ $\mid$ *sized-arg*, *sized-args*	
sized-arg	::=	$\textit{sized-type}_{position}$	
position	::=	ϵ $\mid$ $\langle f, n\rangle$	f functor, $0 \le n \le$ arity of f

We say that n and m appearing in the *bounds* element of this grammar are in *bound positions*. The concretization function γ given in Figure 1 takes a *sized type* and returns the set of terms defined by it. Note that for each type appearing in the right hand side of a type rule, we include its sized type along with the position (functor and place) where it appears. In the case of top level types we use ϵ. We use *nob* as a value for *bounds* to prevent the application of a specific type rule.

Other approaches, e.g., the one proposed for CASLOG (Debray et al. 1990; Debray and Lin 1993) and previous CiaoPP analyses (López-García et al. 1996; Navas et al. 2007), use a predefined set of size metrics, such as the actual value of a number, the length of a list, or term depth. In addition, the developer can create new metrics. We only use type rule applications to bound compound terms. This is not a limitation, since most useful metrics can be expressed or bounded as sized types. For example, the size of a list of between a and b elements is $list^{(a+1,b+1)}$ (we have to include the extra `[]`) and the depth of a term is bounded by the sum of all numbers appearing in the bound positions.

Sized Type Schemas In our abstract domain, we need to refer to sets of sized types which satisfy certain conditions on their bounds. For that purpose, we introduce *sized type schemas*: a schema is just a sized type with variables in bound positions, along with

$$\begin{aligned}
\mathit{sized}(\mathit{num}) &= \mathit{num}^{(\alpha,\beta)}, && \alpha \text{ and } \beta \text{ fresh}\\
\mathit{sized}(\tau) &= \tau^{(\alpha,\beta)}\left(\mathit{sized\text{-}args}(\tau)\right), && \tau \text{ recursive}, \alpha \text{ and } \beta \text{ fresh}\\
\mathit{sized}(\tau) &= \tau\left(\mathit{sized\text{-}args}(\tau)\right), && \tau \text{ not recursive}
\end{aligned}$$

$$\mathit{sized\text{-}args}(\tau) = \bigcup_{\tau\to\phi\in\Phi} \mathit{sized\text{-}rule}(\phi,\tau)$$

$$\begin{aligned}
\mathit{sized\text{-}rule}(\sigma,\tau) &= \emptyset, && \sigma \sqsubseteq \tau\\
\mathit{sized\text{-}rule}(\sigma,\tau) &= \{d_\epsilon : d = \mathit{sized}(\sigma)\}, && \sigma \not\sqsubseteq \tau\\
\mathit{sized\text{-}rule}(f(\sigma_1,\ldots,\sigma_n),\tau) &= \bigcup\{d_{\langle f,i\rangle} : d \in \mathit{sized}(\sigma_i), \sigma_i \not\sqsubseteq \tau\}
\end{aligned}$$

Fig. 2. Sized type schema $\mathit{sized}(\tau)$ for a regular type τ.

a set of constraints over those variables. We call such variables *bound variables*. Given a schema s_i, the set of bound variables appearing in it is denoted $\mathit{vars}(s_i)$.

For each regular type, we can compute a sized type schema representing the same set of terms: basically a schema without any constraints on the variables. The algorithm in Figure 2 generates the mentioned schema for a type τ. Basically, it traverses the set of rules while keeping track of the last type seen in order to detect where recursion appears in type rules. If we apply it to the type:

```
nonemptylistnum -> .(num, listnum)
listnum -> [] | .(num, listnum)
```

we get as sized type schema: $\mathit{nonemptylistnum}\left(\mathit{num}^{(\alpha,\beta)}_{\langle\cdot,1\rangle}, \mathit{listnum}^{(\gamma,\delta)}_{\langle\cdot,2\rangle}(\mathit{num}^{(\mu,\nu)}_{\langle\cdot,1\rangle})\right)$

4 The Abstract Domain

To devise the abstract domain we focus specifically on the geneic 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 and 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 analysis.

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

1. t is a set of $v \to (\mathit{sized}(\tau), c)$, 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 bound variables of relevant variables;
3. r (the *relations*) is a set of relations among bound variables.

The analysis will try to infer a functional relation for the size of output variables in terms of the sizes of relevant variables *output variable* $\leqslant f(\mathit{relevant\ variables})$.

The concretization $\tilde{\gamma}$ of the abstract elements comes from that of sized types: we just need to select the subset of the terms for which domain constraints and relations hold.

$$\tilde{\gamma}(\langle\{v_i \to (s_i, c_i)\}, d, r\rangle) = \left\{ \bigcup\{v_i \to t_i\} \;\middle|\; \begin{array}{l} t_i \in \gamma(s_i(\bar{m}_i)), \bar{v}_i = vars(t_i), \\ \bar{v} = (\bar{v}_1, \ldots, \bar{v}_n), \bar{m} = (\bar{m}_1, \ldots, \bar{m}_n), \\ \bar{v} = \bar{m} \models d(\bar{v}) \wedge r(\bar{v}) \end{array} \right\}$$

As mentioned before, the analysis comprises two stages. The first stage involves running a regular (and moded) type analysis over the program, done in our implementation using (Vaucheret and Bueno 2002). In a second stage we feed this information to the proposed size analysis, which takes such types as fixed, and computes an overapproximation of the least fixpoint for the set of domains and relations. We will now look at each of the operations that define this second stage as an abstract domain in CiaoPP's setting. At the same time we will analyze our initial "list of factorials" example.

4.1 $\sqsubseteq$*,* $\sqcup$ *and* $\bot$

As mentioned before, these three operations are needed to define the abstract domain correctly as a join-semilattice and for the computation of fixpoints in the analysis (Cousot and Cousot 1992). One important remark here is that we do not have a complete definition for $\sqsubseteq$, because there is no general algorithm for checking the inclusion of sets of integers defined by recurrence relations. Instead, we simply check whether one set of inequations is a subset of another one, up to variable renaming (we will denote this syntactic inclusion as $\subseteq^s$). This check is enough to achieve correctness. Recall that in the size analysis we see the types as being fixed by a previous type analysis. We define $\sqsubseteq$ as follows:

$$\langle t, d, r\rangle \sqsubseteq \langle t, d', r'\rangle \iff t = t' \wedge d \subseteq^s d' \wedge r \subseteq^s r'$$

$\sqcup$ and $\bot$ are defined according to this definition of $\sqsubseteq$. For $\bot$ we need to know the types of the variables being referred to as extra parameters. Union is done syntactically again, taking care of renaming variables in inequations to refer to the same terms.

$$\langle t, d, r\rangle \sqcup \langle t, d', r'\rangle = \langle t, d \cup^s d', r \cup^s r'\rangle \qquad \bot_t = \langle t, \emptyset, \emptyset\rangle$$

4.2 λ_{call} *to* β_{entry}

This operation abstracts the unification of the subgoal variables (λ_{call}) onto head variables for each clause C_i (β_{entry_i}) defining a predicate. This is done in four steps. The first one is classification of variables, using mode information (also provided by type analysis): if a variable is unbound at the predicate call and bound to some non-variable term upon success, the variable is clasified as *output*. Otherwise, it is classified as *relevant*.[2]

The next step is generating the sized type schemas of subgoal variables by applying the function *sized* in Figure 2 on their corresponding regular types. Then, we set up constraints for the domain of the relevant variables in these sized types. For this purpose, we check if in the clause head the variable is bound to a ground term, in which case

[2] As future work, we plan to extend this classifier for the discovery of irrelevant variables which play no role in the size of outputs.

unify-sizes takes as input a list of unification equations of the form $X = Y$, where X is a subgoal variable and Y a term in the head, and produces a list of assignments. For each variable X, we denote τ_X and s_X its regular and sized type, respectively.

$$unify\text{-}sizes(R) = \bigcup_{X=Y \in R} \left(\bigcup_{(V,p) \in paths(Y,[])} \{ s_V \lesseqgtr unify\text{-}path(p, \tau_X, s_X) \} \right)$$

$$\begin{array}{rcll} paths(Y, L) & = & \{(Y, L)\}, & Y \text{ variable} \\ paths(f(Y_1, \ldots, Y_n), L) & = & \bigcup paths(Y_i, L +\!\!+ [\langle f, i\rangle]), & f \text{ functor} \end{array}$$

$$\begin{array}{rcl} unify\text{-}path([], \tau, s) & = & s \\ unify\text{-}path\left([\langle f, i\rangle | r], \tau, \tau^{(\alpha,\beta)}(\bar{x})\right) & = & unify\text{-}path(r, \sigma_i, s') \\ \text{where} & & \tau \to f(\sigma_1, \ldots, \sigma_n) \in \Phi \\ & & s' = \begin{cases} \tau^{(\alpha-1,\beta-1)}(\bar{x}), & \sigma_i = \tau \\ d, & \sigma_i \neq \tau, d_{\langle f,i\rangle} \in \bar{x} \end{cases} \\ unify\text{-}path\left([\langle f, i\rangle | r], \tau, \tau(\bar{x})\right) & = & unify\text{-}path(r, \sigma_i, s'), \quad s'_{\langle f,i\rangle} \in \bar{x} \end{array}$$

Fig. 3. Size unification of subgoal and head variables.

we constrain size variables to be bound to concrete numbers, according to the term. Otherwise, we just impose the constraint that size variables must be positive.

Finally, we need to perform the unification between sizes of subgoal input variables and sizes of head variables. This is performed using the algorithm in Figure 3. In the following steps, when a new variable is not found in the first component of the abstract substitution, a new sized type for it is generated, and it is added as *clausal*.

In our `listfact(L, FL)` example, from previous regular type analysis we know that at call time `L` is bound to a list of numbers and `FL` is a free variable, and on success `FL` is also bound to a list of numbers. Thus, we classify `FL` as *output* and `L` as *relevant*. Then, we generate the sized types for them. So far the procedure is the same for both clauses.

From now on, we will focus on the second clause. In β_{entry_2} we have unifications between relevant subgoal variables and head variables: $L = [E|R]$. Following the algorithm for *unify-sizes*$([L = [E|R]])$, given in Figure 3, we need to call *paths*$([E|R], [])$. We get as output $[E = [\langle ., 1\rangle], R = [\langle ., 2\rangle]]$. In both calls to *unify-path* we will go to the rule $listnum \to .(num, listnum)$. Since the type of the variable E is not *listnum*, we just take the sized type referred to by the *position* $\langle ., 1\rangle$, in this case $n^{(\gamma_1,\delta_1)}$. For R, s' is similar to the initial sized type for L, but with one rule application less.

$$\beta_{entry\_2} = \left\langle \begin{array}{c} \left\{ \begin{array}{c} L \to (ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1,\delta_1)}), relevant), FL \to (ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}), output), \\ E \to (n^{(\gamma_3,\delta_3)}, clausal), R \to (ln^{(\alpha_4,\beta_4)}(n^{(\gamma_4,\delta_4)}), clausal) \end{array} \right\}, \\ \{\alpha_1 > 0, \beta_1 > 0\}, \left\{ \begin{array}{c} n^{(\gamma_3,\delta_3)} \lesseqgtr n^{(\gamma_1,\delta_1)} \\ ln^{(\alpha_4,\beta_4)}(n^{(\gamma_4,\delta_4)}) \lesseqgtr ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)}) \end{array} \right\} \end{array} \right\rangle$$

4.3 The Extend Operation

This operation is responsible for extending the current abstract element with the information of a call to a literal. The operation is very simple: include the sized types of any

$$unify\text{-}back(R) = \bigcup_{X=Y \in R} \{s_X \lesseqgtr unify\text{-}back'(Y, \tau_Y)\}$$

$$\begin{array}{rcll}
unify\text{-}back'(t,\tau) & = & ground\text{-}size(t,\tau), & t \text{ ground} \\
unify\text{-}back'(X,\tau) & = & s_X, & X \text{ variable} \\
unify\text{-}back'(f(t_1,\ldots,t_n),\tau) & = & none\text{-}but(\tau, f(d_1,\ldots,d_n)) & \\
\text{where} & & d_i = unify\text{-}back'(t_i,\sigma_i) \ and & \\
 & & \tau \to f(\sigma_1,\ldots,\sigma_n) \in \Phi &
\end{array}$$

Fig. 4. Backwards size unification of subgoal and head variables.

variable which was not yet in the first component of the abstract element and add a call to the equation for the clause referencing the literal.

In our example, we will need to extend $\beta_{entry,2}$ (which will be the first λ in the second clause) with a call to $fact$. To do so, we add the sized type schema for F (we already have information for E) and the call, so the abstract substitution is now:[3]

$$\lambda_{2,2} = \left\langle \ \Big\{\ldots, F \to (n^{(\gamma_5,\delta_5)}, clausal)\Big\}, \{\ldots\}, \{\ \ldots, n^{(\gamma_5,\delta_5)} \lesseqgtr fact(n^{(\gamma_3,\delta_3)})\ \ \} \ \right\rangle$$

4.4 β_{exit} to λ'

This operation abstracts the unification of the execution of an entire clause back with the subgoal variables. Thus the algorithm needs to do the opposite of λ_{call} to β_{entry}: deriving the size of a variable from the sizes of its constituent elements. To do so we use the *unify-back* algorithm outlined in Figure 4.[4] After this point we have a complete set of relations defining the output parameters.

For the second clause of `listfact` we have to call $unify\text{-}back'([F|FR], listnum)$. We need to recursively call $unify\text{-}back'$ with the components F and FR. The final substitution for the second clause will be:

$$\lambda'_2 = \left\langle \begin{array}{c} \Big\{\ldots, FR \to (ln^{(\alpha_6,\beta_6)}(n^{(\gamma_6,\delta_6)}), clausal)\Big\}, \{\ldots\}, \\ \left\{ \begin{array}{c} \ldots \\ ln^{(\alpha_6,\beta_6)}(n^{(\gamma_6,\delta_6)}) \lesseqgtr listfact\left(ln^{(\alpha_4,\beta_4)}(n^{(\gamma_4,\delta_4)})\right) \\ ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \lesseqgtr ln^{(\alpha_6+1,\beta_6+1)}(n^{(\min(\gamma_5,\gamma_6),\max(\delta_5,\delta_6))}) \end{array} \right\} \end{array} \right\rangle$$

4.5 Closed Forms

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 if presented in this closed form. Second, functions are amenable to comparison as outlined in (López-García et al. 2010), essential for example in verification.

[3] Only additions to the elements will be shown. Three dots (...) will replace previous information.
[4] The function *none-but* returns a sized type restricted to a particular type rule.

The $\uparrow$ operator will try to replace relations with a closed form bound. We can see this operator as overapproximating an abstract element, $x \sqsubseteq \uparrow x$. In our experiments we have integrated Mathematica as recurrence solver, and $\uparrow$ is applied at every $\sqcup$ step.

In our example we obtain the following abstract substitution for the first clause:

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

Then, we can bound the joint inequations to get a closed solution:

$$\uparrow(\lambda_1' \sqcup \lambda_2') = \langle\ \{\dots\}, \{\alpha_1, \beta_1 > 0\}, \{ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \lessgtr ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1!,\delta_1!)})\}\ \rangle$$

5 Refinements in the Analysis

Multivariance and Widenings In the example analysed above there is an implicit assumption while setting the relations: the recursive call in the body of `listfact` refers to the same predicate call, so we set up a recurrence equation. 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 (such as different modes or even types).

The CiaoPP framework allows multivariance (support for different call patterns of the same predicate) in the analysis. But to do so we cannot just add calls with the bare name of the predicate, because it will conflate all the existing versions. The proposed solution adds a new component to the abstract element: a random name given to the specific instance of the predicate we are analyzing, that is generated in the λ_{call} to β_{entry}. In the computation of the fixpoint, the $\sqcup$ operator is changed to a widening ∇ which conflates all different versions of the same predicate. In this way we obtain the recurrences.

Structural Subtyping Another problem that may arise is that a predicate returns a subtype of the type we were looking for. For example, in:

```
n_to_zero(0, [0]).
n_to_zero(N, [N|R]) :- N1 is N - 1, n_to_zero(N1, R).
```

the regular type inferred is *nonemptylistnum* for the second argument. In this case, in the backwards unification we have variables of type *num* and *nonemptylistnum* but the rule for the `./2` functor reads `.(num, listnum)`. However, since *nonemptylistnum* $\sqsubseteq$ *listnum* we can view the size description as an instance of a description of its supertype.

To do so, we have developed an extended version of the Dart-Zobel algorithm for type inclusion (Dart and Zobel 1992) which can be found in Appendix B.

6 Cardinality and Resource Analysis

In order to assess the usefulness of the new size analysis in the resource usage estimation application (which is our main goal), we have also developed an integrated into the CiaoPP abstract interpretation framework a resource usage analysis and a cardinality analysis. The latter infers lower and upper bounds on the numbers of solutions produced by a predicate. We provide below a sketch of these analyses (the full details are beyond the scope of this paper).

Cardinality has a multiplicative behavior: if we know the number of solutions of every literal in a clause, we can bound the number of solutions contributed by it using $S_{clause}\,(p(\bar{x})\ :-\ q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)) \leq \prod_{i=1}^{n} S_{pred}(q_i(\bar{x}_i))$. Here we are implicitly using the previously discussed size analysis. The number of solutions of the whole predicate S_{pred} can be calculated by gathering all the equations and solving the resulting system.

Regarding resources, following (Navas et al. 2007) each *resource* is defined by its *head cost* β, which quantifies the amount of resource used in the unification between a subgoal and a clause head, and its *literal cost* δ, which quantifies the amount of resource needed for preparing a call to a predicate. Apart from that, the user can attach directly some resource usage functions to particular predicates. Using these parameters, we can get a formula for upper bounding the resource usage of a clause $C \equiv p(\bar{x}) :- q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)$:

$$RU_{clause}\,(C) \leq \beta(p(\bar{x})) + \textstyle\sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} S_{pred}(q_j(\bar{x}_j))\right) (\delta(q_i(\bar{x}_i)) + RU_{pred}(q_i(\bar{x}_i)))$$

The resource usage of a predicate RU_{pred} is calculated in a similar way to S_{pred}.

As we have seen, cardinality and resource analyses are tightly related to size analysis. Consequently, our implementation of these analyses is via an extension of the previously defined sized types abstract domain:

- The upper and lower bounds, both for the number of solutions and for each resource, is represented by a pair of bound variables (S_l, S_u) and (RU_l, RU_u) respectively, similarly to those used by the analysis in sized type schemas.
- These variables are initialized in the λ_{call} to β_{entry} step: S_l and S_u to 1 (the cardinality before any literal is called), and RU_l and RU_u to the corresponding resource head cost β.
- In each *extend* step, we need to update the bound variables with new values, given the cardinality (S'_l, S'_u) and resource usage (RU'_l, RU'_u) of the called literal:
  - For upper bounds, the cardinality is updated by the product of the previous cardinality and the one from the called literal, $S_u \times S'_u$. For resource usage, the formula is very similar, $RU_u + S_u \times (\delta + RU'_u)$.
  - The methodology is similar for lower bounds, but we have to take into account the possibility of failure, as explained in (Debray et al. 1997).
- As a result of threading the variables through all *extend* steps, in β_{exit} the values of the bound variables for cardinality and resources will be equal to the ones obtained by the formulas we have previously presented.

7 Experimental Results and Conclusions

We have constructed a prototype implementation in Ciao by defining the abstract operations for sized type analysis that we have described in this paper and plugging them into CiaoPP's PLAI implementation. While full benchmarking is beyond the scope of the paper, we provide preliminary results on two aspects: (a) comparison of the new size analysis to the existing CiaoPP size analyses (Debray and Lin 1993; Debray et al. 1997; Navas et al. 2007), and (b) effect of using the new size analysis in the resource usage analysis application.

Regarding (a), the main advantage of our technique is the richer information about the size of terms that is inferred by the analysis. As an illustrative example, consider the predicate `insert` used in insertion sort of a list of lists. The code we used for analysis is a direct translation to Prolog of the one in (Hoffmann et al. 2012):

```
insert(X,[],[X]).
insert(X,[Y|Ys],[X,Y|Ys]) :- leq(X,Y), !.
insert(X,[Y|Ys],[Y|Zs]) :- insert(X,Ys,Zs).

leq([],_).
leq([X|Xs],[Y|Ys]) :- X =< Y, leq(Xs,Ys).
```

Given input arguments[5] $\langle X \rightarrow ln^{(c,d)}(n^{(e,f)}), L \rightarrow lln^{(g,h)}(ln^{(i,j)}(n^{(k,l)}))\rangle$, we get as sized type relation for the third argument `I`[6]:

$$I \quad \rightarrow \quad nelln\begin{pmatrix} ln^{(\min(i,c),\max(d,j))}_{\langle\cdot,1\rangle}(n^{(\min(k,e),\max(f,l))}) \\ lln^{(g,h)}_{\langle\cdot,2\rangle}(ln^{(\min(i,c),\max(d,j))}(n^{(\min(k,e),\max(f,l))})) \end{pmatrix}$$

We see that the analysis has correctly inferred that the result will be a non-empty list and the bounds for all inner elements. For example, the first element of the list of lists will be either the list `X` or one list in `L`, so the bound at that position will be the largest.

Our results show that the new analysis improves on the previous one in 86% (13/15) of a set of benchmarks and produces the same results in the other 14%.

Regarding (b), we have compared the new CiaoPP lower and upper bound resource analyses using the new size analysis with the previous CiaoPP analyses (Debray and Lin 1993; Debray et al. 1997; Navas et al. 2007), and also (upper bounds) with *RAML*'s analysis (Hoffmann et al. 2012). The new analyses improve on CiaoPP's previous resource analysis and in most cases, and are equal in the rest. RAML only infers polynomial costs, while our new approach can infer exponential costs and many other types of cost functions. For predicates with polynomial cost, we get equal or better results than RAML.

8 Other Related Work

Apart from recurrence equations, there are other approaches to size analysis. One popular one is the use of CLP($\mathbb{R}$) and convex hulls, such as (Benoy and King 1997). In this case, the analysis infers a set of linear inequations between sizes of terms. The main advantage of this proposal is the possibility of relating sizes of several arguments. However, these approaches are usually limited in the mathematical domain used for abstraction (for example, linear inequations), whereas recurrence relations allow much richer expressions.

As mentioned in the introduction, (Hoffmann et al. 2012) shows another approach to size analysis, based on the potential method. Although it allows some costs that we cannot express in our system (for example, sums over all the elements in a list), it is limited to polynomial expressions. In our case, not being tied to polynomial bounds is important, since problems such as the number of solutions usually have exponential behavior.

Inference of norms for termination analysis is also related to size analysis. For example, (Decorte et al. 1994) or (Bruynooghe et al. 2007) use semi-linear norms to prove termination. These norms define the size of a term as the sum of some of its components,

[5] We write *lln* for *listlistnum*, the type of lists of lists of numbers, and *nelln* for its non-empty variant.
[6] We are using a condensed version of the abstract element, where we write the results of inequations directly inside the sized type.

which are later related by linear inequations. This approach summarizes all information in one number, so it is less convenient for the developer and less useful for other analyses.

References

Benoy, F. and King, A. 1997. Inferring Argument Size Relationships with CLP(R). In *Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'97).* Lecture Notes in Computer Science, vol. 1207. Springer, 204–223.

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

Bruynooghe, M., Codish, M., J. P. Gallagher, Genaim, S., and Vanhoof, W. 2007. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems 29,* 2.

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. PLDI'90.* ACM, 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.

Decorte, S., Schreye, D. D., and Fabris, M. 1994. Exploiting the power of typed norms in automatic inference of interargument relations. Tech. rep., TR 246, Dpt CS, , K.U.Leuven.

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.

Hughes, J., Pareto, L., and Sabry, A. 1996. Proving the correctness of reactive systems using sized types. In *POPL.* 410–423.

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

López-García, P., Hermenegildo, M., and Debray, S. K. 1996. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation 21*, 715–734.

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.

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.

$$
\begin{array}{rcl}
\mathit{merge}\left(\tau^{(\alpha,\beta)}(\bar{x}), \tau^{(\gamma,\delta)}(\bar{y})\right) & = & \tau^{(\alpha+\gamma,\beta+\delta)}(\mathit{merge\text{-}args}(\bar{x},\bar{y}))
\end{array}
$$

$$
\mathit{merge\text{-}args}(\bar{x},\bar{y}) \;=\; \{\mathit{merge\text{-}arg}(x_i,y_i)_{\langle f,p\rangle} : x_i \text{ and } y_i \text{ have subscript } \langle f,p\rangle\}
$$

$$
\begin{array}{rcl}
\mathit{merge\text{-}arg}(\sigma^{nob}, y) & = & y \\
\mathit{merge\text{-}arg}(x, \sigma^{nob}) & = & x \\
\mathit{merge\text{-}arg}\left(\sigma^{(\alpha,\beta)}(\bar{z}), \sigma^{(\gamma,\delta)}(\bar{w})\right) & = & \sigma^{(\max(\alpha,\gamma),\min(\beta,\delta))}(\mathit{merge\text{-}args}(\bar{z},\bar{w}))
\end{array}
$$

$$
\mathit{none}(\tau) \;=\; \tau^{nob}\left(\bigcup_{\tau\to\phi\in\Phi} \mathit{none\text{-}rule}(\phi,\tau)\right)
$$

$$
\begin{array}{rcll}
\mathit{none\text{-}rule}(f(\sigma_1,\ldots,\sigma_n),\tau) & = & \{\sigma^{nob}_{i,\langle f,i\rangle} : \sigma_i \not\sqsubseteq \tau\} & \\
\mathit{none\text{-}rule}(\sigma,\tau) & = & \{\sigma^{nob}_{\epsilon}\}, & \sigma \not\sqsubseteq \tau \\
\mathit{none\text{-}rule}(\sigma,\tau) & = & \emptyset, & \sigma \sqsubseteq \tau
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{none\text{-}but}(\tau, f(d_1,\ldots,d_n)) & = & \mathit{merge}(\tau^{(1,1)}(A), \text{fold } \mathit{merge} \text{ over } S) \\
\text{where} & & \langle A,S\rangle = \bigcup_{\tau\to\phi\in\Phi} \mathit{none\text{-}but\text{-}rule}(\tau,\phi,f(d_1,\ldots,d_n))
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{none\text{-}but\text{-}rule}(\tau, f(\sigma_1,\ldots,\sigma_n), f(d_1,\ldots,d_n)) & = & \langle\{d_{i,\langle f,i\rangle} : \sigma_i \not\sqsubseteq \tau\}, \{d_{i,\langle f,i\rangle} : \sigma_i \not\sqsubseteq \tau\}\rangle \\
\mathit{none\text{-}but\text{-}rule}(\tau, \phi, f(d_1,\ldots,d_n)) & = & \langle\emptyset,\emptyset\rangle, \quad \phi \text{ does not start with } f
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ground\text{-}size}(n, \mathit{num}) & = & \mathit{num}^{(n,n)} \\
\mathit{ground\text{-}size}(f(t_1,\ldots,t_n),\tau) & = & \mathit{none\text{-}but}(\tau, f(d_1,\ldots,d_n)) \\
\text{where} & & d_i = \mathit{ground\text{-}size}(t_i,\sigma_i) \\
 & & \tau \to f(\sigma_1,\ldots,\sigma_n) \in \Phi
\end{array}
$$

Fig. A 1. Sized types auxiliary functions.

Appendix A Auxiliary Algorithms over Sized Types

In Figure A 1 we describe the auxiliary algorithms used in the operations in the sized types abstract domain. These algorithms are very similar to the derivation of sized type definitions. For simplicity, we only give the algorithms for recursive types, the non-recursive case just does not compute *bounds* for the number of rule applications.

Appendix B Extended Type Inclusion with Sizes

We do not include the definition of the "plural" functions, which just apply a "singular" function over a list (for example *opens* just collects the results of *open* over every element of a list). The auxiliary functions can be found in Figure B 1 and the main *subset* algorithm is in Figure B 2.

We assume *head* and *tail* functions giving the first and rest elements of a list, respectively, and a ++ list concatenation operator.

$$expand(\psi) = \begin{cases} \{\psi\}, & \tau \text{ not a type symbol} \\ \{[\langle \phi, expand\text{-}size(\phi, s)\rangle] \mathbin{+\!\!+} tail(\psi) \ : \ \tau \to \phi \in \Phi\}, & \tau \text{ a type symbol} \end{cases}$$
$$\text{where } \langle \tau, s\rangle = head(\psi)$$

$$\begin{array}{rcll} expand\text{-}size(\sigma, \tau^{(\alpha,\beta)}(\bar{x})) & = & \sigma^{s'}(\bar{y}), & \text{if } \sigma_\epsilon^{s'}(\bar{y}) \in \bar{x} \\ expand\text{-}size(f(\sigma_1, \dots, \sigma_n), \tau^{(\alpha,\beta)}(\bar{x})) & = & f(d_1, \dots, d_n) & \\ \text{where} & & d_i = \begin{cases} \tau^{(\alpha-1,\beta-1)}(\bar{x}), & \sigma_i = \tau \\ s_i, & \sigma_i \neq \tau, s_{i,\langle f,i\rangle} \in \bar{x} \end{cases} & \end{array}$$

$$selects(\tau, \Psi) = \begin{cases} \{\psi \in \Psi : head(\psi) = \langle \top, s\rangle \vee head(\psi) = \langle \tau, s\rangle\}, & \tau \text{ a type symbol} \\ \{\psi \in \Psi : head(\psi) = \langle \top, s\rangle \vee head(\psi) = \langle f(\omega_1, \dots, \omega_n), s\rangle\}, & \tau = f(\sigma_1, \dots, \sigma_n), n > 0 \end{cases}$$

$$open(\langle \tau, s\rangle, \psi) = \begin{cases} tail(\psi), & \tau \text{ is } \top \text{ or a base symbol} \\ [\langle \top, \top\rangle, \dots, \langle \top, \top\rangle] \mathbin{+\!\!+} tail(\psi), & \tau = f(\omega_1, \dots, \omega_n), head(\psi) = \langle \top, s'\rangle \\ [\langle \sigma_1, s_1\rangle, \dots, \langle \sigma_n, s_n\rangle] \mathbin{+\!\!+} tail(\psi), & \tau = f(\omega_1, \dots, \omega_n), \\ & head(\psi) = \langle f(\sigma_1, \dots, \sigma_n), f(s_1, \dots, s_n)\rangle \end{cases}$$

Fig. B 1. Extended type inclusion, auxiliary functions.

$$\begin{array}{rcl} subset(\langle \bot, s\rangle, \langle \tau, s'\rangle) & = & \langle \text{true}, \emptyset\rangle \\ subset(\langle \sigma, s\rangle, \langle \bot, s'\rangle) & = & \langle \text{false}, \emptyset\rangle \\ subset(\langle \sigma, s\rangle, \langle \tau, s'\rangle) & = & \langle b, postprocess(r)\rangle \\ \text{where} & & \langle b, r\rangle = subsetv([\langle \sigma, s\rangle], \{[\langle \tau, s'\rangle]\}, \emptyset) \end{array}$$

$$\begin{array}{rcl} subsetv(\psi, \emptyset, C) & = & \langle \text{false}, \emptyset\rangle \\ subsetv([], \Psi, C) & = & \langle \text{true}, \emptyset\rangle \\ subsetv(\psi, \Psi, C) & = & subsetv(tail(\psi), tails(\Psi), C) \\ & \text{if} & \langle head(\psi), \Theta\rangle \in C \wedge heads(\Psi) \subseteq \Theta \\ subsetv(\psi, \Psi, C) & = & subsetvs(expand(\psi), \Psi, C \cup \{\langle head(\psi), heads(\Psi)\rangle\}) \\ & \text{if} & head(\psi) = \langle \tau, s\rangle, \tau \text{ a type symbol} \\ subsetv(\psi, \Psi, C) & = & \langle b_R, R \cup r_R\rangle \\ & \text{if} & head(\psi) = \langle \tau, s\rangle, \tau \text{ is } \top \text{ or } f(\omega_1, \dots, \omega_n), n > 0 \\ & & \Sigma = selects(\tau, expands(\Psi)) \\ & & R = \bigcup_{S \in \Sigma} unify(s, head(S)) \\ & & \langle b_R, r_R\rangle = subsetv(open(\langle \tau, s\rangle, \psi), opens(\langle \tau, s\rangle, \Sigma), C) \end{array}$$

$$\begin{array}{rcl} subsetvs([], \Psi, C) & = & \langle \text{true}, \emptyset\rangle \\ subsetvs([\psi | R], \Psi, C) & = & \langle b_\psi \wedge b_R, r_\psi \cup r_R\rangle \\ \text{where} & & \langle b_\psi, r_\psi\rangle = subsetv(\psi, \Psi, C) \\ & & \langle b_R, r_R\rangle = subsetvs(R, \Psi, C) \end{array}$$

postprocess(R) gathers all the unifications in R over the same variable X, and generates the maximum or minimum expression of it, depending on whether the variable is in an upper or lower bound position.

Fig. B 2. Extended type inclusion, main *subset* function.

Attachment D3.1.3

Towards an Abstract Domain for Resource Analysis of Logic Programs Using Sized Types

Published at the 23rd Workshop on Logic-based Methods in Programming Environments (WLPE 2013).

Towards an Abstract Domain for Resource Analysis of Logic Programs Using Sized Types

Alejandro Serrano[1], Pedro López-García[1,2] and Manuel Hermenegildo[1,3] ⋆

[1] IMDEA Software Institute
[2] Spanish Council for Scientific Research (CSIC)
[3] Universidad Politécnica de Madrid (UPM)

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 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 both, infering 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, like memory usage, time spent in computation, or bytes sent over a wire, throughout the execution of a piece of code. Such numerical properties are known as *resources*. 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 [7, 6] and [8], 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 pedicate), and which ultimately obtain the resource usage bounds. One drawback of these proposals, as well as most of their subsequent derivations, is that they are

⋆ 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 and the Spanish TIN2012-39391-C04-01 *STRONGSOFT* project.

only 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([],    []).          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 $\alpha + \sum_{e \in L} (\beta + time_{fact}(e))$, where α and β are constants related to the 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 [20] we focused on a proposal to improve the size analysis based on *sized types*. These sized types are similar to the ones present in [21] 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*, i.e., not just the auxiliary analyses but also the resource analysis itself. This allows us to integrate resource analysis within the PLAI abstract interpretation framework [15, 18] in the CiaoPP system, which brings in features such as *multivariance*, fixpoints, and assertion-based verification and user interaction for free. We also perform a performance assessment 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 of our resource analysis work.

2 Overwiew 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 [22]. 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. In our case, the sized type schema *listnum-s* is derived from *listnum*. This schema corresponds to a list that contains a number of elements between α and β, and each element is between the bounds γ and δ. It is defined as:

$$listnum\text{-}s \rightarrow listnum^{(\alpha,\beta)}(num^{(\gamma,\delta)}_{\langle \cdot,1\rangle})$$

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 corresponding to argument `X` in a call `append(X, Y, Z)` (created from the regular type inferred by a previous analysis). We have that $size_X$ denotes $ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}_{\langle \cdot,1\rangle})$. Similarly, the sized type schema for the output argument `Z` is $ln^{(\alpha_Z,\beta_Z)}(n^{(\gamma_Z,\delta_Z)}_{\langle \cdot,1\rangle})$, denoted by $size_Z$. Now, 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 purpose, 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. 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:

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

In a second step, we group together all the inequalities of a single sized type. As we always intercalate 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. Thus, we use a compact representation $\lesseqgtr$ for the symbols $\geq$ and $\leq$ that are always paired. For example, the expression:

$$ln^{(\alpha_X,\beta_X)}(n_{\langle \cdot,1\rangle}^{(\gamma_X,\delta_X)}) \lesseqgtr ln^{(e_1,e_2)}(n_{\langle \cdot,1\rangle}^{(e_3,e_4)})$$

represents the conjunction of the following size constraints:

$$\alpha_X \geq e_1,\ \beta_X \leq e_2,\ \gamma_X \geq e_3,\ \delta_X \leq e_4$$

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

$$size_Z\,(size_X, size_Y) \lesseqgtr ln^{(\alpha_X+\alpha_Y,\beta_X+\beta_Y)}(n_{\langle \cdot,1\rangle}^{(\min(\gamma_X,\gamma_Y),\max(\delta_X,\delta_Y))})$$

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. $\delta_Z \geq \max(\delta_X, \delta_Y)$), 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 now on cardinality analysis. Let s_L and s_U denote lower and upper bounds on the number of solutions respectively that predicate `append/3` can generate. Following the program structure we can infer that:

$$\begin{aligned}
s_L\left(ln^{(0,0)}(n_{\langle \cdot,1\rangle}^{(\gamma_X,\delta_X)}), size_Y\right) &\geq 1\\
s_L\left(ln^{(\alpha_X,\beta_X)}(n_{\langle \cdot,1\rangle}^{(\gamma_X,\delta_X)}), size_Y\right) &\geq s_L\left(ln^{(\alpha_X-1,\beta_X-1)}(n_{\langle \cdot,1\rangle}^{(\gamma_X,\delta_X)}), size_Y\right)
\end{aligned}$$

$$\begin{aligned}
s_U\left(ln^{(0,0)}(n_{\langle \cdot,1\rangle}^{(\gamma_X,\delta_X)}), size_Y\right) &\leq 1\\
s_U\left(ln^{(\alpha_X,\beta_X)}(n_{\langle \cdot,1\rangle}^{(\gamma_X,\delta_X)}), size_Y\right) &\leq s_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n_{\langle \cdot,1\rangle}^{(\gamma_X,\delta_X)}), size_Y\right)
\end{aligned}$$

The solution to these inequations is $(s_L, s_U) = (1, 1)$, so we have inferred that `append/3` generates at least (and at most) one solution. Thus, it behaves like a function. When setting up the equations, we have used our knowledge that `append/3` cannot fail when given lists as arguments. If not, the lower bound in the number of solutions would be 0.

Now we move forward to analyzing the number of resolution steps preformed 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:

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

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 of possible backtrackings is bounded by the number of solutions of the predicate. So the equation reads:

$$\begin{aligned} r_U\left(ln^{(\alpha_X,\beta_X)}(n_{\langle .,1\rangle}^{(\gamma_X,\delta_X)}), size_Y\right) &\leq 1 + s_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n_{\langle .,1\rangle}^{(\gamma_X,\delta_X)}), size_Y\right) \\ &\quad \times r_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n_{\langle .,1\rangle}^{(\gamma_X,\delta_X)}), size_Y\right) \\ &= 1 + r_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n_{\langle .,1\rangle}^{(\gamma_X,\delta_X)}), size_Y\right) \end{aligned}$$

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:

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

3 Sized Types Review

As shown in the `append` example, the (bound) 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 [5] but with some adaptations. A *type term* is either a *base type* α_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 $\tau \rightarrow \phi$, 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 abstact domain we focus specifically on the generic AND-OR trees procedure of [3], with the optimizations of [15]. 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 [10], 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.* A grammar for the new sized types follows:

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

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, 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 for conciseness:

$$
\lambda'_1 = \left\langle \begin{array}{c} \{L \to (ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1,\delta_1)}), rel.), FL \to (ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}), out.)\} \\ \{\alpha_1 = 1, \beta_1 = 1\}, \{ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \lesseqgtr ln^{(1,1)}(n^{nob})\} \end{array} \right\rangle
$$

$$
\lambda'_2 = \left\langle \begin{array}{c} \left\{ \begin{array}{c} L \to (ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1,\delta_1)}), rel.), FL \to (ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}), out.), \\ E \to (n^{(\gamma_3,\delta_3)}, cl.), R \to (ln^{(\alpha_4,\beta_4)}(n^{(\gamma_4,\delta_4)}), cl.), \\ F \to (n^{(\gamma_5,\delta_5)}, cl.), FR \to (ln^{(\alpha_6,\beta_6)}(n^{(\gamma_6,\delta_6)}), cl.) \end{array} \right\} \\ \{\alpha_1 > 0, \beta_1 > 0\}, \\ \left\{ \begin{array}{c} ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \lesseqgtr ln^{(\alpha'+1,\beta'+1)}(n^{(\min(\gamma_1!,\gamma'),\max(\delta_1!,\delta'))}) \\ ln^{(\alpha',\beta'}(n^{(\gamma',\delta')}) \lesseqgtr \textit{factlist}\left(ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)})\right) \end{array} \right\} \end{array} \right\rangle
$$

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 [16], 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, is called *head cost*, β.
- Before calling a literal: 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 δ.

We first consider the case of estimating upper bounds on resource usages. For simplicity, assume also 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 $r_{U,clause}$ using the formula:

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

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 equations that meet the contraints on the input data sizes (i.e., have the same domain).

However, in logic programming we have two extra features to take care of:

- 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 that information, 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 then the relation:

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

- 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 clauses when setting up the recurrence equations. 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 of the abstract domain 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.

4.1 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 all possible clauses:

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

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 [6], 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 [13]. If such analysis infers that a predicate is deterministic, we can safely set the upper bound for the number of solutions to 1, avoiding the setting up of recurrence equations.

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 [4]. In case of a possible failure, the lower bound on the number of solutions is set to 0.

4.2 The Abstract Elements

The abstract elements are derived from sized type analysis by adding some extra components. In particular, we include four new elements:

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. Information about non-failure analysis, coming from its abstract domain.
4. Information about determinacy analysis, coming from its abstract domain.

We will denote the abstract elements by

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

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, $\mathit{failed?}$ is a boolean element (either `true` or `false`), d and r are defined as in the sized type abstract domain, and nf and det can take the values `not_fails/fails` and `non_det/is_det` respectively.

In this analysis we assume that we are given the definition of a set of resources, which are fixed throughout the whole analysis process. We have already mentioned three operations, but we need an extra one for having a complete algorithm. For each resource r we have:

– Its head cost, β_r, which takes a clause head as parameter;
– Its literal cost, δ_r, which takes a literal as parameter;
– Its aggregation procedure, $\mathit{\Gamma}_r$, which takes the equations for each of the clauses and creates a new set of recurrence equations from them;
– 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 the `listfact` predicate that we started in the previous section. We assume that the only resource to be analyzed is the "number of steps," so that we use the following values for the parameters of the resource analyis:

$$\beta = 1, \quad \delta = 0, \quad \mathit{\Gamma}_r = +, \quad (\bot_L, \bot_U) = (0, 0)$$

4.3 $\sqsubseteq$, $\sqcup$ and $\bot$

We do not have a complete 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, we just check whether one set of inequations is a subset of another one, up to variable renaming, or perform a syntactic union of the inequations. This is enough for having a correct analysis.

For $\bot$ 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 could be any natural number, so we take it as $[0, \infty)$. We also assume that the predicate may fail.

As mentioned before, the components for non-failure and determinacy come from the abstract domains for those analyses.

For example, the bottom element for the "number of steps" resource will be (where $\bot_{nf}$ and $\bot_{det}$ are the bottom elements in the non-failure and determinacy domains respectively):

$$\langle (s_L, s_U), \{(n_L, n_U)\}, \texttt{true}, \emptyset, \{(s_L, s_U) \lesseqgtr (0, \infty), (n_L, n_U) \lesseqgtr (0, 0)\}, \bot_{nf}, \bot_{det} \rangle$$

4.4 λ_{call} to β_{entry}

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, and also the neutral element of the product which appears in the corresponding formula). For a resource r, the initial value is exactly β_r.

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:

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

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

4.5 The Extend Operation

In the *extend* operation we get both the current abstract substitution and the abstract substitution coming 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 an assignment: $s_{U,c} \leq s_{U,p} \times s_{U,\lambda}$.
  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 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 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 $r_{U,c} \leq r_{U,p} + s_{U,p} \times (\delta + r_{U,\lambda})$.

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 would be $r_{L,c} \geq r_{L,p}$.

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

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

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

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

4.6 β_{exit} to λ'

After performing all the extend operations, the variables appearing in the number of solutions and resources positions will hold the correct value for their respective numerical properties. As we did with sized types, we follow now a normalization step, based on the algorithm described in [6]: we replace each variable appearing in a expression with its definition in terms of other variables, in reverse topological order, starting from the desired variables. Following this process, we should reach the variables in the sized types of the input parameters in the clause head.

Going back to our `listfact` example, the final substitutions would be:

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

$$\lambda'_{entry,2} = \left\langle \begin{array}{c} (s_{L,2,3}, s_{U,2,3}), \{(n_{L,2,3}, n_{U,2,3})\}, \texttt{false}, \{\alpha_1 > 0, \beta_1 > 0\}, \\ \left\{ \begin{array}{c} s_{L,2,3} \geq 1 \times listfact_{sol.,L}(ln^{(\alpha_1 - 1, \beta_1 - 1)}(n^{(\gamma_1, \delta_1)})), \\ s_{U,2,3} \leq 1 \times listfact_{sol.,U}(ln^{(\alpha_1 - 1, \beta_1 - 1)}(n^{(\gamma_1, \delta_1)})), \\ n_{L,2,3} \geq \gamma_1 + listfact_{no.\ steps,L}(ln^{(\alpha_1 - 1, \beta_1 - 1)}(n^{(\gamma_1, \delta_1)})), \\ n_{U,2,3} \leq \delta_1 + listfact_{no.\ steps,L}(ln^{(\alpha_1 - 1, \beta_1 - 1)}(n^{(\gamma_1, \delta_1)})) \end{array} \right\}, \\ \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

4.7 Widening ∇ and Closed Forms

As mentioned before, in contrast to previous cost analyses, at this point we bring in the possibility of allowing 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. This describes how we 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 recurrence relation setting.
– 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 equation. This fact is implicitly assumed in Hindley-Milner type systems, where each expression and function receives only one type. But in logic programming it is usual for a predicate to be called with different patterns (such as different modes). Fortunatelly, the CiaoPP framework allows multivariance (support for different call patterns of the same predicate) in the analysis. For the analysis to handle it, we cannot just add calls with the bare name of the predicate, because it will conflate all the existing versions. The solution that we propose adds a new component to the abstract element: a random name given to the specific instance of the predicate we are analyzing, that is generated in the λ_{call} to β_{entry}. Then, in the widening step, all different versions of the same predicate name 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 if presented in such a closed form. Second, functions are amenable to comparison as outlined in [14], which is essential for example in resource usage verification. There are several software packages that are able to get bounds for recurrence equations: computer algebra systems, such as Mathematica (the one used in our experiments) or Maxima; and specialized solvers such as PURRS [2] or PUBS [1]. In our implementation we apply this overapproximation operator after each widening step. For our example, the final abstract substitution is:

$$\lambda'_1 \nabla \lambda'_2 = \left\langle \begin{array}{c} (s_L, s_U), \{(n_L, n_U)\}, \mathtt{false}, \{\alpha_1, \beta_1 \geq 0\}, \\ \{(s_L, s_U) \lesseqgtr (1,1), (n_L, n_U) \lesseqgtr (\alpha_1\gamma_1, \beta_1\delta_1)\}, \mathtt{not\_fails}, \mathtt{is\_det} \end{array} \right\rangle$$

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 in this paper and plugging them into CiaoPP's PLAI implementation. Our objective is to assess the gains in precision in resource consumption analysis.

Table 1. Experimental results.

Program	*Resource Analysis (LB)*			*Resource Analysis (UB)*				
	New	*Previous*		*New*	*Previous*		*RAML*	
`append`	α	α	$=$	β	β	$=$	β	$=$
`appendAll2`	$a_1a_2a_3$	a_1	$+$	$b_1b_2b_3$	∞	$+$	$b_1b_2b_3$	$=$
`coupled`	μ	0	$+$	ν	∞	$+$	ν	$=$
`dyade`	$\alpha_1\alpha_2$	$\alpha_1\alpha_2$	$=$	$\beta_1\beta_2$	$\beta_1\beta_2$	$=$	$\beta_1\beta_2$	$=$
`erathos`	α	α	$=$	β^2	β^2	$=$	β^2	$=$
`fib`	ϕ^μ	ϕ^μ	$=$	ϕ^ν	ϕ^ν	$=$	infeasible	$+$
`hanoi`	1	0	$+$	2^ν	∞	$+$	infeasible	$+$
`isort`	α^2	α^2	$=$	β^2	β^2	$=$	β^2	$=$
`isortlist`	a_1^2	a_1^2	$=$	$b_1^2b_2$	∞	$+$	$b_1^2b_2$	$=$
`listfact`	$\alpha\gamma$	α	$+$	$\beta\delta$	∞	$+$	unknown	?
`listnum`	μ	μ	$=$	ν	ν	$=$	unknown	?
`minsort`	α^2	α	$+$	β^2	β^2	$=$	β^2	$=$
`nub`	a_1	a_1	$=$	$b_1^2b_2$	∞	$+$	$b_1^2b_2$	$=$
`partition`	α	α	$=$	β	β	$=$	β	$=$
`zip3`	$\min(\alpha_i)$	0	$+$	$\min(\beta_i)$	∞	$+$	β_3	$+$

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 [6, 8, 16] (columns *Previous*). We also compare (for upper bounds) with *RAML*'s analysis [11] (column *RAML*).

Although the new resource analysis and the previous one infer concrete resource usage bound functions (as the ones in [16]), 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 $\Phi \in \bigcirc(\Psi)$, 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 $n^{(\mu,\nu)}$. Table 1 also includes columns with symbols summarizing whether the new CiaoPP resource analysis improves on the previous one and/or *RAML*'s: $+$ (resp. $-$) indicates more (resp. less) precise bounds, and $=$ the same bounds. The new size analysis improves on CiaoPP's previous resource analysis in most cases. Moreover, RAML can only infer polynomial costs, while our approach is able to infer exponential costs (as well as many other types of cost functions), as is shown for the divide-and-conquer benchmarks `hanoi` and `fib`, which represent a large and common class of programs. For predicates with polynomial cost, we get equal or better results than RAML.

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 time or heap consumption), but it seems clear that those analyses could be generalized.

We already mentioned RAML [11] in Section 5. Their approach differs from ours in the theoretical framework being used: RAML uses a type and effect system, whereas our system uses abstract interpretation. Another important 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 [9] and [12].

In [17], the authors use sparsity information to infer asymptotic complexities. In contrast, we only get closed forms. Similarly to CiaoPP's previous analysis, the approach of [1] applies the recurrence equation method directly (i.e., not within an abstract interpretation framework). [19] shows a complexity analysis based on abstract interpretation over a step-counting version of functional programs.

7 Conclusions and Future Work

In this paper 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 [20]. We have seen how this approach offers benefits both in the quality of the bounds inferred by the analysis, and in the ease of implementation and integration within a framework such as PLAI/CiaoPP.

In the future, we would like to study the generalization of this framework to allow the analysis of resources with slightly different behaviors regarding aggregation. For example, when running tasks in parallel, the total time is basically the maximum of both tasks, but memory usage is bounded by the sum of them. Another future direction is the integration of more of the analyses present in the CiaoPP analysis system, in order to obtain more precise results. Also, since we use sized types as a basis, any new research that improves such analysis will directly benefit the resource analysis. Finally, another planned enhancement is the use of mutual exclusion analysis (already present in CiaoPP) to aggregate recurrence equations in a better way.

References

1. E. Albert, S. Genaim, and A. N. Masud. More Precise yet Widely Applicable Cost Analysis. In *Proc. of VMCAI'11*, volume 6538 of *LNCS*, pages 38–53. Springer, 2011.
2. R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. Technical report, 2005. `arXiv:cs/0512056` available from `http://arxiv.org/`.
3. Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *J. Log. Program.*, 10(2):91–124, 1991.
4. F. Bueno, P. López-García, and M. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.
5. P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

6. S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
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. Bernd Grobauer. Cost recurrences for DML programs. In *International Conference on Functional Programming*, pages 253–264, 2001.
10. 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.
11. Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.
12. Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Symposium on Principles of Programming Languages*, pages 331–342, 2002.
13. P. López-García, F. Bueno, and M. Hermenegildo. Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information. *New Generation Computing*, 28(2):117–206, 2010.
14. P. López-García, L. Darmawan, and F. Bueno. A Framework for Verification and Debugging of Resource Usage Properties. In *Technical Communications of ICLP*, volume 7 of *LIPIcs*, pages 104–113. Schloss Dagstuhl, July 2010.
15. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.
16. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.
17. Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. Automatic complexity analysis. In *European Symposium on Programming*, pages 243–261, 2002.
18. G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium (SAS 1996)*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.
19. M. Rosendahl. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*. ACM Press, 1989.
20. A. Serrano, P. Lopez-Garcia, F. Bueno, and M. Hermenegildo. Sized Type Analysis for Logic Programs (technical communication). *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement*, August 2013. To Appear.
21. Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In Philip W. Trinder, Greg Michaelson, and Ricardo Pena, editors, *IFL*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2003.
22. C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.

Attachment D3.1.4

Genetic Algorithm-based Allocation and Scheduling for Voltage and Frequency Scalable XMOS Chips

Published at Hybrid Artificial Intelligent Systems (HAIS 2013).

Genetic Algorithm-based Allocation and Scheduling for Voltage and Frequency Scalable XMOS Chips

Zorana Banković[1] and Pedro López-García[1,2]

[1] IMDEA Software Institute, Madrid, Spain
[2] Spanish Council for Scientific Research (CSIC), Spain
{zorana.bankovic,pedro.lopez}@imdea.org

Abstract. In this work we present a novel approach, based on genetic algorithms, for automatic scheduling and allocation of tasks in a multi-processor multi-threaded architecture, together with an assignment of the appropriate voltage and frequency of each processor in a way the overall energy consumption is optimized and all task deadlines are met. The approach deals with scheduling, allocation and voltage and frequency assignment at the same time, and provides good solutions in a very short time. As far as we know, this is the first approach that supports two levels of parallelism: multi-processor and multi-thread.

1 Introduction

Dynamic power consumption due to switching activity in digital CMOS circuits can be expressed with the following formula: $P = \alpha C_{eff} V^2 f$, where C_{eff} is the effective capacitance, V is the operating voltage, f is the operating frequency, and α the switching factor. If we can decrease the voltage supply and the operating frequency, the dynamic power will decrease significantly. On the other hand, static power which is the result of the leakage currents also decreases quadratically with voltage [7]. Thus, voltage decrease can achieve significant power and energy savings. This process is known as Dynamic Voltage and Frequency Scaling (DVFS). However, it slows down the operation of the circuit, and has to be applied in a way the required deadlines are still fulfilled. Furthermore, the process introduces additional latencies, so we have to develop a set of requirements that define the applicability of this approach.

The objective of this work is to optimize energy consumption through optimal scheduling and allocation for a set of tasks on XMOS chips, which are multiprocessor and multithreaded voltage and frequency scalable architecture. In XMOS chips threads are pipelined in a four-stage pipeline, where in each stage one instruction from different thread is executed, so in essence we can say that the threads also run in parallel. Thus, we deal here with two levels of parallelism. We assume that different processors can have different (V, f) setting, while the threads running on the same processor at the same time must have the same (V, f) setting.

Given a set of tasks and their corresponding deadlines, the objective is to provide a scheduling and allocation, and also assign voltage and frequency for each processor that would optimize the energy, while respecting the deadlines. The tasks are heterogeneous, and they in general have different starting time and deadline. We assume that there is no precedence between tasks, and no preemption. In order to solve the problem, we need to have safe estimates of power consumption of each task, as well as its execution time. Since this work falls into ENTRA project [1], whose main task is to provide the programmer the estimation of the energy consumption of his/hers program at compile time, we assume that there exists an analysis that would give us this information, as necessary input. On the other hand, there is a great body of work about time analysis, so we assume that the analyzer will provide us this information as well.

The general problem of scheduling and allocation is NP-hard. In order to solve it, different heuristic algorithms have been developed since they are capable of obtaining sub-optimal solutions in real time. Many of them use genetic algorithm (GA) [3, 4, 9] due to its fast exploration of the search space, which allows it to quickly find acceptable solutions. For this reason, our scheduler will also be based on GA. We will provide appropriate solution representation that captures the two levels of parallelism, i.e. at both processor and thread level, and in the same run performs allocation and scheduling and identifies appropriate (V, f) setting in real time. As far as we know, this is the only solution for this type of problems.

The rest of the work is organized as follows. Section 2 details the sources of power consumption, while Section 3 explains the problem that is being solved and draws the constraints that are the basis for generating the solution. Section 4 details the implemented solution, while Section 5 explains the experimentation environment and presents the most significant results. Section 6 presents the most relevant related work, and finally, Section 7 draws the most important conclusions.

2 CPU Power Consumption

The energy required to complete a (set of) task(s) in time T on one processor, given the frequency f and voltage V is defined by:

$$E_{cpu,f,V} = \int_{t_0}^{t_0+T} P_{cpu,f,V}(t)\mathrm{d}t \tag{1}$$

where $P_{cpu,f,V}$ is the time varying XCore power at *(V,f)* setting. This power can be calculated as:

$$P_{cpu,f,V}(t) = P^{fix}_{cpu,V} + P_{idle,f,V} + P^{act}_{cpu,f,V}(t) \tag{2}$$

where P^{fix}_{cpu} is the portion of the power that includes PLL and leakage [7], which is the part that only depends on voltage, not on frequency. $P_{idle,f,V}$ is the power spent when the processor is not executing any application. For a certain fixed *(V,f)* setting, the sum of these two does not change in time, so in the further text we will call it standing power consumption, $P^{std}_{cpu,V,f}$. This power can be easily obtained by measuring the CPU power when there are no running applications for each *(V,f)* setting. On the other hand, $P^{act}_{cpu,f,V}(t)$ is the active power spent on switching activity during the execution of the application(s). Finally, we can write:

$$P_{cpu,f,V}(t) = P^{std}_{cpu,f,V} + P^{act}_{cpu,f,V}(t) \tag{3}$$

which put in 1 gives the energy consumed during time T :

$$E_{cpu,f,V} = P^{std}_{cpu,f,V}T + \sum_{i=1}^{M} P_{i,f,V}T_i \tag{4}$$

where $P_{i,f,V}$ is the power spent by the application i, which is executed during time of T_i, and M is the number of threads, i.e. the maximal number of applications that can be executed on one processor at certain moment. In the cases when the threads can finish more than one application within time T, formula 4 would have the following form:

$$E_{cpu,f,V} = P^{std}_{cpu,f,V}T + \sum_{i=1}^{M}\sum_{j=1}^{K} P_{ij,f,V}T_{ij} \tag{5}$$

where K is the maximal number of applications a thread can execute in time T.

3 Problem Description

Problem Definition
Given a set of concrete tasks, provide optimal scheduling and *(V,f)* pair(s) for each processor in order to optimize energy consumption.

Input
- Set of tasks with their corresponding deadlines.
- Set of possible *(V,f)* pairs.
- Available hardware: n - number of processors, m - number of threads per processor.

Output
Viable scheduling and allocation that optimizes energy.

In the following text we assume the notation where variables are expressed using upper case letters, while constants are expressed using lower case letters.

3.1 Timing Constraint

In general case, for each new frequency $F_{new,i}$ of each processor i, the following should remain valid:

$$\forall i \in [1,n], \forall j \in [1,m],\ T_{oh,i} + \frac{C_{ij}}{F_{new,i}} \leq D_{ij} \tag{6}$$

where T_{oh} is the time overhead introduced by DVFS and C_{ij} is the number of clock cycles needed to execute the application j on processor i, giving its execution time to be $\frac{C_{ij}}{F_{new,i}}$. This is reasonable to assume, given that in XMOS there are no pipeline stalls, nor cache misses, since there is no cache memory. We further have:

$$\forall i \in [1,n],\ T_{oh,i} = t_{oh_V} + T_{oh_{f,i}} \approx t_{oh_V} + \frac{10}{F_{old,i}} + \frac{2}{F_{new,i}} \tag{7}$$

where t_{oh_V} is the time overhead of performing voltage scaling (assumed to be constant), while T_{oh_f} if the time overhead of performing frequency scaling, which takes 10 clock cycles at most of the old clock, and two cycles of the new clock [6]. Finally, from 6 and 7 we get the timing constraints set:

$$\forall i \in [1,n], \forall j \in [1,m],\ F_{new,i} \cdot (C_{ij}+2) \leq D_{ij} - t_{oh_V} - 10/F_{old,i} \tag{8}$$

where we consider that we know t_{oh_V}, and both F_{old} and F_{new} can take one value from the finite set of the pre-established values *(V,f)*.

3.2 Energy Minimization Constraint

The second set of requirements is derived from the condition of reducing the total energy during some known time t, high enough so that it permits the termination of all the applications. This implies the following condition:

$$\forall i \in [1,n], \forall j \in [1,m],\ t \geq \max_{i,j} D_{ij} \tag{9}$$

Thus, for each processor, we have:

$$\begin{aligned}&\sum_{i=1}^{n} E_{old} \geq \sum_{i=1}^{n} E_{new} \Rightarrow \\ &\sum_{i=1}^{n} p^{std}_{i,cpu,F_{old,i},V_{old,i}} \cdot t + \sum_{i=1}^{n}\sum_{j=1}^{m} p_{ij,V_{old,i},F_{old,i}} \cdot \frac{C_{ij}}{F_{old,i}} \geq \\ &\sum_{i=1}^{n} e_{oh} + \sum_{i=1}^{n} p^{std}_{i,cpu,F_{new,i},V_{new,i}} \cdot t + \sum_{i=1}^{n}\sum_{j=1}^{m} p_{ij,V_{new,i},F_{new,i}} \cdot \frac{C_{ij}}{F_{new,i}}\end{aligned} \tag{10}$$

where e_{oh} is the energy spent on voltage and frequency scaling, $p_{ij,V_{old,i},F_{old,i}}$ and $p_{ij,V_{new,i},F_{new,i}}$ are estimated total power consumptions of the application j on XCore i in the different (V,f) settings, while p^{std}_{cpu} is the standing power explained in Section 2 in different settings. Finally, from 10, we get:

$$\begin{aligned}&\sum_{i=1}^{n}\sum_{j=1}^{m}\left(\frac{p_{ij,V_{new,i},F_{new,i}}}{F_{new,i}} - \frac{p_{ij,V_{old,i},F_{old,i}}}{F_{old,i}}\right) \cdot C_{ij} \leq \\ &t \cdot \sum_{i=1}^{n}\left(p^{std}_{i,cpu,F_{old,i},V_{old,i}} - p^{std}_{i,cpu,F_{new,i},V_{new,i}}\right) - n \cdot e_{oh}\end{aligned} \tag{11}$$

where the only unknown parameters are C_{ij}.

4 Proposed Solution

Our solution for optimal scheduling and allocation is based on GA. We have used steady-state GA, where the number of individuals of the population is the same in every generation and in every generation 60% of the population with lowest objective values is replaced with newly created individuals. Custom roulette wheel selector is used for the selection process. In the following text we will explain other important aspects of its implementation in more detail.

Individual. The starting point, and one of the most important parts, in designing a GA-based solution is always a representation of a solution, i.e. individual. In our case, the solution contains information about temporal and spatial allocation of each task. In other words, for each processor and each of its threads we should have an ordered (in time) set of tasks. However, since in this work we deal with DVFS, we have to add the information about the (V, f) state of each processor. All the threads on the same processor have the same (V, f) setting in the same moment, but we assume that different processors can have different (V, f) setting, in order to solve the most general problem.

We can look at a solution to the scheduling problem as a permutation of the task identifiers, where their order also stands for the order of their temporal execution, assuming that each task has a unique identifier. On the other hand, in order to solve the allocation problem, i.e. on which thread (and which processor) each task is executed, we can add delimiters to the permutation of the task IDs that would define where the tasks are being executed, i.e. processor, thread and (V, f) setting (the tasks between two delimiters are executed on the right-side one). In order to be able to distinguish delimiters from the task, they are used as negative three-digit numbers, where the first digit stands for the processor, the second for the thread on that processor, and the third for the processor (V, f) setting (there is a finite number of settings). Part of a solution is depicted in Fig. 1, where tasks with IDs 1, 2, 5 and 7 are executed in that order on the thread 4 of the core 2, with the (V, f) setting marked as 4. In the most general case, the order of delimiters is random. However, if two consecutive delimiters that belong to the same processor have different settings, this means that they are not being executed in parallel, since the state has to be changed. Representing solution in the described way has provided us with a relatively simple solution, which will not introduce great overhead when executing GA.

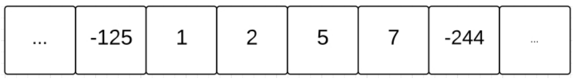


Fig. 1. Solution Representation

Population Initialization. We have used a heuristics when initializing the population in order to provide some good quality individuals from the beginning. According to it, the task is added to the thread in the way the total resulting energy up to the moment is minimal. However, the total energy is calculated for the time equal to the farthest deadline for each thread. In this way, more weight is given to the static power overhead. Thus, the objective of this heuristic is to promote delaying the execution of each task towards its deadline through minimizing the energy overhead. However, since in general GAs benefits from great variety of solutions, we also introduce random solutions. During the initialization process, each individual randomly chooses between heuristics and a randomly generated solution, where the heuristics has slightly bigger possibility to be chosen (0.6).

Solution Perturbations. Given that all the tasks and all the delimiters are different, different solutions are always a permutation of a set of tasks and the set of delimiters. This gives us the opportunity to use some of the permutation-based crossover operator, and in this case we are using the partial match crossover, since it performed better in the terms of objective function than the cycle crossover, and slightly better than order crossover in the terms of objective function and execution time. Since the order of delimiters is not important in the most general case, this operator provides at the same time variety in consecutive changes of (V, f) settings, as well as moving tasks from one thread to another. Regarding mutation, it is implemented in the way that two random threads exchange two random tasks with a small probability.

Objective Function. Since the aim is to the minimize total energy, the objective function is the total energy consumed for executing the given set of tasks and it is calculated as presented in Section 2. However, the execution time for each thread is taken as the farthest deadline of its tasks, in order to take full advantage of the DVFS possibility. Furthermore, we have to be sure that the solution is viable, i.e. that all given deadlines are met. We deal with this problem through the penalization of the inviable solutions by multiplying their energy by 10. In this way, viable solutions will always have higher objective function and thus higher probability of surviving to the next generation.

5 Experimental Evaluation

5.1 Testing Environment

XMOS Chips. The target architecture for this work are XMOS chips. However, the same approach can be followed for any kind of DVFS-enabled multi-processor architectures. In the case of XMOS chip, both voltage and frequency scaling are possible and both introduce time overhead. All their chips provide the possibility of frequency scaling due to the existence of a programable frequency divider. The time overhead introduced when changing frequency is not more than 10 cycles of the old clock, plus two more cycles of the new clock.

On the other hand, only XS1-SU01A-FB96 [8] chip provides the possibility of voltage scaling due to the existence of two DC-DC converters whose output voltage can belongs to the range (0.6V, 1.3V). In order to apply DVSF, we should have a list of Voltage-Frequency *(V,f)* pairs or ranges that provide correct chip functioning. The latency in this case is not controllable, and can be estimated in the following way. Since the switching frequency of the converter is 1MHz, and assuming we have linear control, the bandwidth should be 1/10 to 1/7 of it, i.e. 150kHz in the best case. Thus, the time it takes for the output voltage to stabilize is 1/150kHz, which is around 6μs.

We have experimentally concluded that the XMOS chips can function properly in six (V, f) settings given in the first two columns of Table 1. In order to include the possibility of shutdown, we could include the state (0(V),0(MHz)) and take the wake-up time as the latency of changing the state, and proceed in the same way. For the purpose of this experiment, we assume that we have four different processors, where each processor has eight different threads.

Task Set For the purpose of this initial experiment we have used the tasks from the well known SPECCPU2006 [2] benchmark. The input dataset is composed of 200 tasks randomly chosen, where each is one from the benchmarks. Each task is independent. The same reasonable deadline is assigned to each task, so it provides the possibility of applying DVFS. Their execution time is measured on a general purpose computer, and the execution time on an XMOS chip is estimated to be $T_{measured} \cdot \frac{f_{XMOS}}{f_{gp}}$. This estimation is based on the assumption that the total number of execution cycles is the same in both cases, and that it is representative of the total execution time. While this is true for the XMOS, in the general purpose computer this is not the case due to cache misses, pipeline stalls, etc. Thus, in the future we would have to profile the tasks on the XMOS chips, or use static analysis. It is important to point out that the duration of the tasks, as well as their energy, are much bigger than both time and energy overhead of DVFS scaling, so in this experiment the overhead will not be a limiting factor.

Table 1. Typical power consumption for each processor state

$V(V)$	$f(MHz)$	$P_{dyn}(mW)$	$P_{st}(mW)$
0.95	500	117.325	18.05
0.87	400	78.7176	15.138
0.8	300	49.92	12.8
0.8	150	24.96	12.8
0.75	100	14.625	11.25
0.7	50	6.37	9.8

Since this work represents an initial study of the approach, we have taken that the power consumption of each task is typical XMOS power consumption given in [7]. The estimations for different (V, f) settings are estimated by scaling with voltage and frequency in the case of dynamic power, while the static power is scaled with voltage, i.e. $P_{dyn} = \frac{P_{dyn}^{base} \cdot f_{new} \cdot V_{new}^2}{f_{base} \cdot V_{base}^2}$ and $P_{st} = \frac{P_{st}^{base} \cdot V_{new}^2}{V_{base}^2}$. These values are given in Table 1 for each (V, f) setting. However, it is assumed that in the future the analyzer will give us an estimation of power consumption of each task.

5.2 Obtained Results and Discussion

Genetic algorithm is executed on 500 individuals, during 100 generations. Greater number of individuals does not provide significantly better solution. In Fig. 2 we can see that the best objective value does not significantly change during the last iterations. The objective value is given in Wh. Under these settings, the total execution time of the algorithm is around six minutes on an Intel Dual Core machine, with 2.4GHz clock.

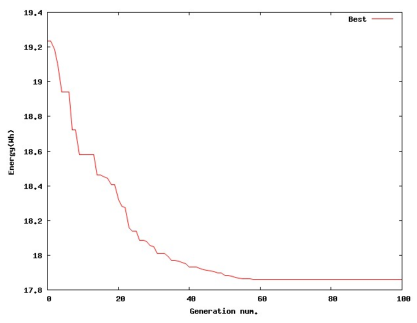


Fig. 2. Evolution of the best, the average and the maximum objective value

The average savings achieved in this way are 33.94%, with standard deviation of 0.56%, compared to the same scheduling and allocation without the DVFS. Speaking in the terms of statistical significance, we can be 95% sure that the obtained savings will belong to the interval (33.02%, 34.86%). A typical solution is presented in Fig. 3. Separate (V, f) settings are distinguished with different colors, where the settings 1-6 correspond to the ones given in Table 1, and one time unit corresponds to one task. As we can observe, the majority of the tasks are executed in low power settings 4, 5 and 6.

6 Related Work

Since DVFS can provide significant energy savings, its optimal usage has been extensively studied. Some examples divide scheduling and allocation in two sep-

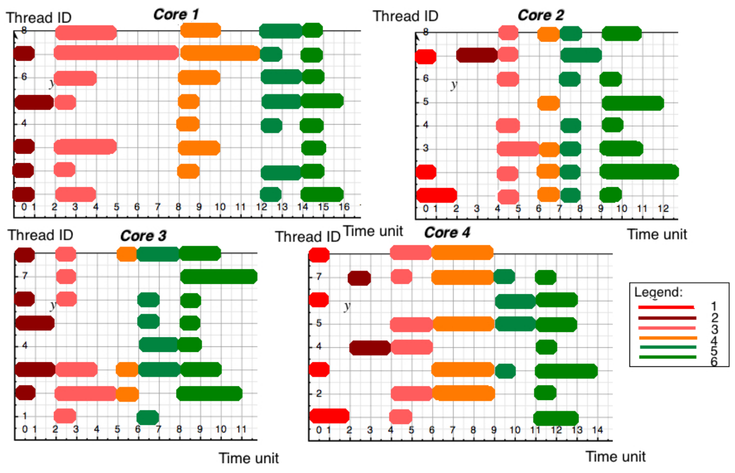


Fig. 3. A Scheduling and Allocation Solution per Core with Assigned DVFS setting

arate tasks, such as the one given in [11], where in the first step the allocation problem is solved using Linear Programming, while in the second the scheduling problem is solved for separate processors using Bin Packing. Another solution [3] solves the scheduling problem using GA, while it integrates DVFS in the fitness function. However, we believe that more optimal solutions could be achieved when solving scheduling and allocation at the same time, while also accounting for the DVFS. There is one example of GA-based scheduling [4] that combines scheduling, allocation and power management in one task. However, it deals only with voltage scaling.

There is also a significant group of publications on using GAs for optimal scheduling and allocation in multiprocessor systems with DVFS possibility. An example given in [9] treats the problem as bi-objective, where they want to minimize both energy and make span. The same objective is solved in another work [10], but using the island model of parallel GA populations. Yet, in this work our aim is to optimize the energy while meeting the deadlines, but our approach can easily be adapted to work as multi-objective. Another solution [5] treats the problem from two opposite points of view: in the first one, optimizes the energy given the scheduler length, while in the other optimizes the scheduling length given the energy bound. Finally, none of the solutions does not include the possibility of two levels of parallelism, where each processor can have a number of different threads executing in parallel.

7 Conclusions

In this work we have presented a solution for optimizing energy consumption for a multiprocessor and multithreaded architecture. The solution performs scheduling and allocation as one task, and deals with two levels of parallelism, which is the only solution of this kind as far as we know.

The solution will form part of the tool developed within the ENTRA project, when we will be able to include the power and time estimates provided by the ENTRA static analyzer. There are also possibilities to further improve the performances of the solution. For example, XMOS chips have the possibility to automatically reduce the frequency of the processor if all of its threads are waiting for an event, and in this way decrease the energy consumption even further. This feature will be included into future versions of our scheduler, as well as the possibility of shutting off separate components while they are not active.

Acknowledgements. The research leading to these results has been supported by the European FP7/2007-2013 318337 *ENTRA* project, and the Spanish TIN2012-39391-C04-01 *STRONGSOFT* project.

References

1. Entra project. `http://entraproject.eu/`, 2013.
2. Speccpu2006. `http://www.spec.org/cpu2006/`, 2013.
3. Ying Chang-tian and Yu Jiong. Energy-aware genetic algorithms for task scheduling in cloud computing. In *ChinaGrid Annual Conference (ChinaGrid), 2012 Seventh*, pages 43–48, 2012.
4. V. Kianzad, S.S. Bhattacharyya, and Gang Qu. Casper: an integrated energy-driven approach for task graph scheduling on distributed embedded systems. In *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on*, pages 191–197, 2005.
5. P.R. Kumar and S. Palani. A dynamic voltage scaling with single power supply and varying speed factor for multiprocessor system using genetic algorithm. In *Pattern Recognition, Informatics and Medical Engineering (PRIME), 2012 International Conference on*, pages 342–346, 2012.
6. XMos Ltd. Xs1-l active energy conservation, april 2010.
7. XMos Ltd. Estimating power consumption for xs1-l devices, may 2012.
8. XMos Ltd. Xs1-su01a-fb96 datasheet, november 2012.
9. M. Mezmaz, Young Choon Lee, N. Melab, E. Talbi, and A.Y. Zomaya. A bi-objective hybrid genetic algorithm to minimize energy consumption and makespan for precedence-constrained applications using dynamic voltage scaling. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8, 2010.
10. M.-S. Mezmaz, Y. Kessaci, Y.C. Lee, N. Melab, E.-G. Talbi, A.Y. Zomaya, and D. Tuyttens. A parallel island-based hybrid genetic algorithm for precedence-constrained applications to minimize energy consumption and makespan. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 274–281, 2010.
11. F. Paterna, A. Acquaviva, A. Caprara, F. Papariello, G. Desoli, and L. Benini. An efficient on-line task allocation algorithm for qos and energy efficiency in multicore multimedia platforms. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March.

Attachment D3.1.5

A Coverage Model to Capture the Communication Behaviour of Multi-Threaded Message-Passing Programs

Submitted to International Conference on Software Testing (ICST 2014).

A Coverage Model to Capture the Communication Behaviour of Multi-Threaded Message-Passing Programs

Kyriakos Georgiou
Department of Computer Science
University of Bristol
Bristol,UK
Email: KyriakosGeorgiou@bristol.ac.uk

Mike Bartley
Test and Verification Solutions
Bristol, UK
Email: mike@testandverification.com

Kerstin Eder
Department of Computer Science
University of Bristol
Bristol,UK
Email: kerstin.eder@bristol.ac.uk

***Abstract*—Concurrency related bugs are reportedly the most difficult to catch during testing of multi-threaded programs. It is therefore important to ensure that the tests run during verification fully exercise the scenarios where threads interact. Traditional code coverage models such as statement coverage are inherently weak for this purpose. This paper presents a new coverage model that complements existing ones. It is designed specifically to capture the communication between threads in message-passing programs. Static code analysis is used to extract the coverage model from a multi-threaded program. Our model is more fine grained than traditional code coverage models because the coverage criteria incrementally capture all aspects of the communication in a program. The model has strongly defined mathematical properties which allow the detection of communication bugs already during static model extraction, leaving specific protocol-related bugs for dynamic detection. The added value of this coverage model is demonstrated in a case study using a prototype tool for the multi-threaded programming language XC.**

I. Introduction

The significant shift towards multi-core designs during the last decade has lead to a surge of attention towards multi-threaded software, in order to exploit the processing power available on these systems. Introducing concurrency in software increases its complexity and the responsibility on developers to manage it. Concurrency introduces an extra source of bugs that are known to be the most difficult to catch, therefore effective testing for multi-threaded programs plays an important role in the development life cycle of multi-threaded programs.

Code and structural coverage tools are provided for both software and hardware languages. Such tools analyze the code and automatically generate coverage tasks to be achieved for a target coverage model. For example, a code coverage tool generates coverage tasks for statements or branches in the code, while a structural coverage tool may generate coverage tasks related to Finite State Machines. Coverage data is collected at runtime and coverage reports summarize the coverage achieved during testing. Coverage data informs test engineers of the diversity of the tests and gives an indication of the degree of completeness of the testing. The process of coverage closure can expose bugs. Achieving full coverage, *i.e.* 100% coverage, requires considerable effort; especially for the more complex coverage models. In practice, engineers compromise by settling for less than 100% coverage to meet their schedule. This leaves code unexercised and can leave bugs undetected.

In a study conducted on message-passing programs [11], communication related bugs were the most common to cause runtime errors, with a share of almost 25%. Analysis revealed that stray and missing messages resulted in communication protocol violations which in turn caused deadlocks, livelocks and other erroneous program behaviour. Considering that such a large proportion of bugs is caused by inter-thread communication, and that conventional code coverage models are inherently weak at exposing concurrency scenarios, priority during testing of multi-threaded programs should be given to exercising these, striving for 100% coverage of these critical parts of the code.

A significant amount of work has been done in defining new concurrent coverage metrics to improve the testing effectiveness without increasing the test suite size in comparison to traditional approaches [5]. Although for most of them their effectiveness still heavily depends on generating large test suites or applying constraints to the program structure to gain testability.

This paper presents a new coverage model that complements existing ones, but focuses on a set of bugs that can be easily captured statically prior to executing any tests, due to its expressiveness. It is designed specifically to capture the communication between threads in message-passing programs. The new coverage model is structured over several layers, increasing in expressive power and complexity. It starts with covering the basic communication channels and builds up towards full communication protocols at the top layer. Static code analysis is used to extract the coverage model from a multi-threaded program prior to running tests. Our model is more fine grained than traditional coverage models, in the sense that the coverage criteria are incrementally capturing all the components of the communication in a program and it has strongly defined mathematical properties that allows the capture of bugs both statically and dynamically.

A prototype tool has been implemented for XC, the concurrent and real-time programming language designed for the XMOS XCore processor [15]. The new coverage model has been evaluated on XC programs containing communication bugs. The evaluation provides evidence that the new coverage model adds value to the verification process in practice; a set of bugs were captured statically by just extracting the coverage tasks. The effectiveness of the model over existing conventional coverage models is also demonstrated. While evaluation has been performed on the XC language, the proposed new coverage model and the static analysis techniques employed to extract the model from the source code are both generally transferable to other languages that support message-passing.

This paper is organized as follows. Section II discusses the background and related work on coverage models for concurent programs, focusing on the message passing programming model. Section III describes the proposed coverage model alongside with the 7 coverage criteria and their hierarchy and the methodologies proposed to extract the coverage tasks. Section IV describes the experimental evaluation of the proposed coverage model. A case study is used to demonstrate the bug exposure capabilities of the model. Section V presents our conclusion and future work.

II. Background and related work

A. Coverage-Based Testing

Testing is still dominant in industry, even though it is often ad hoc, erratically effective and extremely expensive [3]. Studies estimate that 50%, and in some cases even more, of the total software development costs are spent on testing [2]. The definition of measurable criteria is critical to assess the quality and the completeness of the testing. A variety of conceptually very similar coverage models are used during hardware [13] and software [3] development. In general, the most tricky bugs to find often reside in highly concurrent code parts. It is therefore important to ensure these are covered during testing. Coverage models based on source code, such as statement, branch and expression coverage, are inherently weak for this purpose. Implicit structures in the code, such as flow graphs or state machines can be captured in structural coverage models; these are more effective than pure code coverage for concurrent programs. To complement existing coverage models, a coverage model that is increasingly popular in hardware verification is cross-product functional coverage [14]. These coverage models are Cartesian products over strategically selected design signal domains defined by experienced verification engineers. The target coverage space is represented by individual tuples, each of which is a coverage task. It is the engineer's responsibility to apply constraints to this coverage space to identify only those tuples that are intended to be feasible; these constitute the valid coverage tasks. Cross-product functional coverage is also adopted as part of our model as explained later in Section III.

B. Coverage in the Context of Concurrent Programs

In recognition of the lack of coverage models specifically targetting concurrent or multi-threaded software, four requirements have been identified [2] for such coverage models to gain acceptance as follows:

1) The model should be created statically from the code prior to the testing, and each task must be well understood by the user [4].
2) Almost all coverage tasks must be coverable and, for the few that are not, due to practical limitations of testing, a review process should be used.
3) Every uncovered task should yield an action item: either a bug in a program that needs to be fixed, or missing tests that need to be written.
4) Some action is taken upon reaching 100% coverage for the model (*e.g.* the testing phase is complete).

The synchronization coverage model proposed in [2] captures the synchronization primitives in shared memory concurrent programs, with 100% synchronization coverage giving developers confidence in the correctness of thread interaction. The new coverage model presented here complements this model, as it is designed specifically to capture the communication between threads in multi-threaded message-passing programs. It also satisfies all four acceptance criteria.

C. Related work on Coverage for Message-Passing Programs

The most popular programming models for concurrency, the shared-memory and the message-passing models [7], both enjoy popularity today. The latter uses message passing for communication of threads and is typically associated with distributed-memory applications. Message passing programs comprise of a collection of communication objects which synchronize and communicate with each other by sending and receiving messages.

The main difference compared to a sequential program that determines the overall result of the computation is that the final output does not only depend on the program code and the external input but also on the thread interactions. Specifically, in a message-passing program, race conditions may occur due to unintended process scheduling and unpredictable variations in message delays [8]. These race conditions can cause different synchronization sequences even when executing the program with the same input, and may lead to the production of different results, and undesirable program behaviour such as deadlocks or livelocks. Due to this non-deterministic behavior of a message-passing program, testing becomes significantly more difficult than that of a sequential program.

A widely used strategy to test sequences of thread interactions is based on synchronization sequences (SYN-sequence) [1]. A SYN-sequence refers to the execution of a sequence of totally or partially ordered synchronized events; it captures the execution of a parallel program with given input. In the context of message-passing programs, a synchronization event (SYN-event) is defined by a pair that consists of a send event and its corresponding receive event. The set of feasible SYN-sequences consitute the coverage tasks in such a coverage model. SYN-sequences can be classified into feasible and valid SYN-sequences [9], where all SYN-sequences that can be executed are termed feasible, and all SYN-sequences that are intended according to the specification of the program are termed valid.

A framework for non-deterministic testing of message-passing programs based on a SYN-event coverage model is introduced in [8]. SYN-sequences and SYN-events are recorded; testing is stopped when coverage closure has been achieved on all SYN-events and all send/receive statements have been covered.

Atomic SYN-event testing [9] relies on linearization of SYN-events within SYN-sequences, thus each SYN-sequence consists only of serial atomic SYN-events. This ensures that only one feasible SYN-sequence exists for each program input. The number of feasible SYN-sequences is thus drastically reduced at the cost of severely restricting the level of concurrency in the program, while gaining testability.

One way to extract SYN-sequences is based on traversing a reachability graph of a concurrent program [17], where each node in the graph represents a statement. To reduce complexity, the concurrency graph, which is a simplified reachability graph containing only synchronization related statements, can be traversed.

An alternative, simpler technique is based on Event InterActions Graphs (EIAGs) [6]. An EIAG is a combination of Event Graphs (EG), which are control flow graphs of program units such as threads, with InterActions between these program units. The EIAG model was used to direct test generation for structural testing of concurrent Ada programs. The set of SYN-sequences is defined in terms of copaths, where each copath is generated by combining selected paths from individual threads based on the program's EIAG.

Although much effort has been invested towards finding a strategy for generating effective sets of SYN-sequences for testing on parallel programs, there is still a lot of work to been done to achieve a complete, practical solution.

We identified three key points that need to be addressed towards that solution as follows:

1) How to define a coherent, incremental coverage model of coverage criteria that gives control to the programmer in terms of the level of complexity of the testing?
2) How to automatically extract the coverage model from a message-passing program without limiting the programmer to restricted forms of concurrency?
3) How to automatically generate coverage tasks from the source code?

Note that all the techniques reviewed so far require running tests prior starting capturing any bugs. They do not take advantage of the fact that some concurrency bugs in syntactically correct programs can be identified statically, during coverage model extraction. This is also a weakness in the structure and expressiveness of the existing coverage models. Our model is more fine grained in the sense that the coverage criteria are incrementally capturing all the components of the communication in a program and they have strongly defined mathematical properties that allow the capture of bugs both statically and dynamicaly. Extracting the communication of the program under test also provides the ability to compare it to the initial communication specifications.

III. The Proposed Coverage Model

A. The Seven Coverage Criteria

A criterion contains two parts, program properties, like program statements and branches, and a property satisfaction function which is used to choose proper test cases needed to exercise a certain program property [10]. A criterion measures the quality of testing by checking how many program properties are satisfied. In order to have complete testing under a criterion, the testing must achieve complete coverage by satisfying all program properties.

The two metrics to evaluate a coverage criterion are its cost and bug-exposing capability. The cost of a coverage criterion depends on the amount of test cases needed in order to satisfy all the program properties [16]. Bug-exposing capabilities clearly depend on what set of program properties a coverage criterion aims to test. Achieving a good balance between both metrics is the critical part of setting a criterion. Hierarchical families of coverage criteria offer increasingly more complex coverage criteria and hence offer choice to the test engineer.

In this section we propose a hierarchy of seven coverage criteria. In addition to this we also express our criteria using mathematical properties. This will enable the capture of bugs statically when extracting the coverage model, prior to running any test, as is demonstrated in Section IV.

Let $\mathbb{P}$ be a multi-threaded program consisting of a set of threads $\mathbb{T} = \{T_1, \ldots, T_k\}$. The set of communication channels that are declared in $\mathbb{P}$ is defined as

$$\begin{aligned} C &= \{\ c \mid c \textit{ is a communication channel declared in } \mathbb{P}\ \} \\ &= \{c_1, \ldots, c_l\}. \end{aligned} \tag{1}$$

Send and receive statements on a channel c_j are denoted by ${}_{c_j}snd_x$ and ${}_{c_j}rcv_x$, respectively, where x represents a unique identifier such as the location of the statement in a thread in terms of a line number or statement index.

Criterion 1, Channel: The most basic criterion captures channel usage between threads. A channel c_j has been exercised when at least one send and one receive on c_j have been executed. Full coverage is achieved when all channels $c_1, \ldots, c_l$ in C have been covered.

Criterion 2, snd-statement: This criterion captures all send statements in $\mathbb{T}$. For a channel c_j, the set of all send statements is denoted by:

$$\begin{aligned} S_{c_j} &= \{\ {}^{T}_{c_j}snd_x \mid {}_{c_j}snd_x \textit{ is a send statement in } T \in \mathbb{T}\ \} \\ &= \{\ {}^{T}_{c_j}snd_x\ :\ T \in \mathbb{T}\ \} \end{aligned} \tag{2}$$

The number of send statements on channel c_j is $|S_{c_j}| = n_{c_j}$. Full coverage is achieved when for all c_j in C all n_{c_j} send statements have been covered.

Criterion 3, rcv-statement: This criterion captures all receive statements in $\mathbb{T}$. For a channel c_j, the set of all receive statements is denoted by:

$$\begin{aligned} R_{c_j} &= \{\ {}^{T}_{c_j}rcv_x \mid {}_{c_j}rcv_x \textit{ is a receive statement in } T \in \mathbb{T}\ \} \\ &= \{\ {}^{T}_{c_j}rcv_x\ :\ T \in \mathbb{T}\ \} \end{aligned} \tag{3}$$

The number of receive statements on channel c_j is $|R_{c_j}| = m_{c_j}$. Full coverage is achieved when for all c_j in C all m_{c_j} receive statements have been covered.

Criterion 4, SYN-event: In analogy to [12], [1] and [8], SYN-events are ordered pairs of send and receive statements per channel, each representing a potential thread interaction. For a channel c_j, the set of all SYN-events is defined as the Cartesian product over the send statements S_{c_j} and receive statements R_{c_j} as follows:

$$\begin{aligned} &SYN\!-\!event_{c_j} = S_{c_j} \times R_{c_j} \\ &= \{\ ({}^{T'}_{c_j}snd_x, {}^{T''}_{c_j}rcv_y) \mid {}^{T'}_{c_j}snd_x \in S_{c_j} \ \wedge\ {}^{T''}_{c_j}rcv_y \in R_{c_j}\ \} \end{aligned} \tag{4}$$

Note that for XC programs, where channels are uni-directional and shared between two threads only, for each channel c_j we establish the property that the SYN-events $({}^{T'}_{c_j}snd_x, {}^{T''}_{c_j}rcv_y)$ in $SYN\!-\!event_{c_j}$ relatate to exactly two threads T' and T'' such that $T' \neq T''$ and each T' and T'' is used for either send or receive only. This property will be used to capture potential bugs, as demostrated in Section IV-A

The total number of SYN-events on a channel c_j is $\left|SYN\!-\!event_{c_j}\right| = n_{c_j} m_{c_j}$. Full coverage is achieved when for all c_j in C all valid SYN-events have been covered. Note that, as a consequence of program control flow, not all SYN-events in each $SYN\!-\!event_{c_j}$ may be feasible in practice; likewise, not all may be valid, *i.e.* intended. In a bug free program, all valid SYN-events should be feasible, while invalid SYN-events should be infeasible. Figure III-A visualize the SYN-event conditions for a bug free program. Same conditions must apply for the SYN-sequence criterion.

	Valid	Invalid
Feasible	✓	×
Infeasible	×	✓

Fig. 1. SYN-event and SYN-sequence conditions for a bug free program.

Criterion 5, SYN-sequence: As first proposed in [12], SYN-sequences capture traces of synchronization events between threads. Let $SYN\!-\!E$ be the set of all SYN-events in $\mathbb{P}$. A SYN-sequence is formally denoted by $\langle e_1, \ldots, e_l \rangle$ where $e_i \in SYN\!-\!E$. Feasible SYN-sequences can be extracted from $\mathbb{P}$ using static analysis as described in Section III-C. The valid SYN-sequences represent intended communication *protocols* between threads in $\mathbb{P}$. Full coverage is achieved when all valid SYN-sequences have been covered.

An interesting property that applies to the SYN-sequence is the prefix closedness property. For a SYN-sequence coverage task $\langle e_1, \ldots, e_l \rangle$, all prefixes $\langle e_1 \rangle, \langle e_1, e_2 \rangle$, etc. are also coverage targets. A coverage tool can use this to track progress along the SYN-sequence.

Criterion 6, Internal-Order (IO) SYN-event: To increase scrutiny, thread interleaving could be taken into account when covering thread interactions at the level of SYN-events. Thus, for a channel c_j, the IO SYN-events are defined as follows:

$$IO\ SYN\!-\!event_{c_j} \quad = \quad S_{c_j} \times R_{c_j} \cup R_{c_j} \times S_{c_j} \tag{5}$$

IO SYN-events capture the order in which threads arrive at matching send and receive statements. The total number of IO SYN-events on a channel c_j is thus $2\,n_{c_j} m_{c_j}$.

Criterion 7, Internal-Order Based (IOB) SYN-sequence: In analogy to Criterion 5, SYN-sequence, IOB SYN-sequences are traces of synchronization events between threads that distinguish the order in which treads arrive at matching send and receive statements. Thus, let $IOSYN\!-\!E$ be the set of all IO SYN-events in $\mathbb{P}$. An IOB SYN-sequence is formally denoted by $\langle e_1, \ldots, e_l \rangle$ where $e_i \in IOSYN\!-\!E$. This criterion combines Criterion 5 with Criterion 6.

In the scope of this paper coverage analysis is performed up to and including Criterion 5. The XC language supports synchronous message-passing, where threads block to wait for each other before engaging in communication. Recording the order of reaching the send and receive statements adds more value when asynchronous communication is used. Criteria 6 and 7 are therefore shown in gray in Figure 2.

B. The Coverage Model Hierarchy

The seven criteria for coverage models introduced in the previous section form a hierarchy of coverage models as depicted in Figure 2. Solid arrows indicate inclusion of a lower-level criterion in an upper-level criterion in terms of coverage achieved, *i.e.* if full SYN-event coverage (Criterion 4) is achieved, then full coverage is also guaranteed for all criteria below Criterion 4, likewise, full IO SYN-event coverage (Criterion 6) implies full SYN-event coverage (Criterion 4) etc. Dotted arrows indicate that the upper-level criterion is defined in terms of the Cartesian product over several lower-level criteria. The higher the level of a criterion in the hierarchy, the larger the size of the coverage space. Criteria on the higher levels are therefore stronger in terms of the testing thoroughness required to close them.

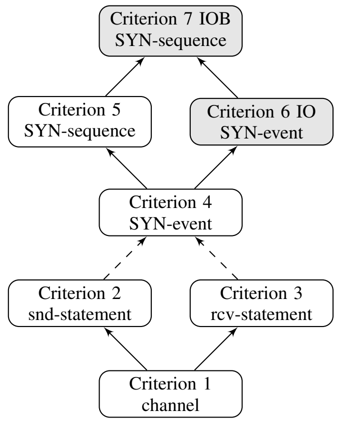


Fig. 2. The hierarchy of coverage models

C. Proposed Coverage Tasks Generation

During the testing stage, a coverage report is created addressing missing coverage tasks, where a coverage task denotes a boolean function on a test. The outcome of a boolean function on a test is the result of the function. For example, in statement coverage, "statement 5" is a coverage task and "statement 5 was not executed" is an outcome of the coverage task "statement 5". To denote missing coverage, is a comparison between coverage tasks in the coverage list and the actually exercised coverage tasks after test runs, is performed. The coverage list containing all valid coverage tasks can be created manually or automated by the use of a specialised tool. Finally when coverage holes are found, new test cases are generated in order to cover them. If the need for quality of a program increases, a new coverage model or criterion can be applied and the whole procedure run again.

In this section we introduce the two phases adopted for coverage tasks generation. The first phase was based on code coverage method and the second one on the structural coverage method. Both of them use static analysis.

1) Phase 1 - Extracting Coverage Tasks for Code Coverage Criterion: This approach was introduced to produce coverage tasks of channel, rcv-statement and snd-statement criterion. They belong to code coverage criteria because they relate to statements in the source code and they can be extracted from it directly. The main idea is to traverse the abstract syntax tree of programs and search for particular nodes, including the corresponding statements for each one of the three criteria. After snd-statements and rcv-statements have been extracted, SYN-events are obtained for free as they form the Cartesian product of the previous two.

2) Phase 2 - Extracting Coverage Tasks for Structural Coverage Criterion: Our method for generating tasks of SYN-sequence criterion is based on EIAG program model proposed by Katayama [6]. Katayama used this model for Ada concurrent programs, which apply to message-passing communication. This makes the model easily adaptable for use by any other message-passing parallel programs, written in other programming languages. An EIAG consists of EGs and interactions between threads. An EG is a control flow graph of a programming unit in a concurrent program, where program units can be procedures, functions and task-types. Nodes in the EG denote concurrent event statements and flow control statements which include the concurrent event statements. Edges in the EG indicate the transfer of control between nodes. The formal steps of creating an EIAG can be found in Katayama [6].

Figure 3(a) shows an example of an EIAG. As Katayama [6] describes their model, node with the number "0" is the start node and node with the number "-1" is the end node of a thread. Moreover the solid line denotes the control flow in a thread, and the dotted line represents an interaction between any two threads. In our work, an interaction is a SYN-event (Criterion 4). The SYN-events in the graph are numbered from the top most left EG to the down most right. Direction of the dotted arrows indicates the direction of messages passed. SYN-events are also distinguished in two cases:

a) ***Straightforward SYN-events on an EIAG :*** SYN-events do not have alternatives, which are due to multiple paths of a thread. Figure 3(a) demonstrates a simple EIAG with straightforward SYN-events.

b) ***Alternative SYN-events on an EIAG :*** Result from multiple paths of threads. Figure 3(b) demonstrates an EIAG with alternative SYN-events.

In phase 1, the set of SYN-events created might contain a number of infeasible SYN-events. By creating the EIAG of a program, some of the infeasible SYN-events are eliminated using data flow and control flow analysis. This can be done up to a certain point, due to the lack of the program semantics. User input will be required to eliminate any remaining infeasible SYN-events. The same applies for generating feasible SYN-sequences. Data flow and control flow analysis is applied on the EIAG to find all the possible combinations of SYN-events that might happen. Then again the user will determine the feasible SYN-sequences. Using the EIAG of Figure 3(a) the only feasible SYN-sequence extracted by the static analysis is $\langle e_3, e_1, e_2 \rangle$. Thus, the coverage task of SYN-sequence criterion in this case is `SYN-seq(1)=`$\langle e_3, e_1, e_2 \rangle$ only.

In the case of Figure 3(b), an EIAG with alternative SYN-events, e_1, e_3 are alternatives because they cannot occur during the same execution, both $\langle e_2, e_1 \rangle$ and $\langle e_2, e_3 \rangle$ sequences can be taken as feasible. Thus, the coverage task of SYN-sequence criterion in this case is `SYN-seq(1)=`$\langle e_2, e_1 \rangle$ and `SYN-seq(2)=`$\langle e_2, e_3 \rangle$.

Finally, after eliminating the infeasible SYN-sequences, we have to eliminate all the remaining feasible ones which are invalid. This is highly depended on the programs semantics (i.e., its control and data flow) [8] which to some extent can be extracted by using static analysis, but user input is also needed in most cases to recognize all the feasible valid ones acording to the specifications of the program.

IV. Experimental Evaluation

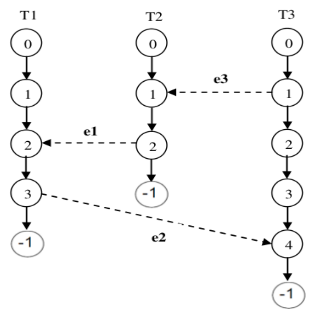
A prototype tool was developed to implement the two methods proposed in the previous section, for the extraction of the coverage tasks. The tool was able to automatically create up to the 4th stage of coverage tasks (SYN-event). As previously stated, user input is required for the creation of the coverage tasks for valid SYN-sequences due to the lack of semantic information in the source code. These valid sequences are defined from the protocol the programmer creates to specify the communication that is allowed to happen between the parallel threads in his program. The tool provides on the same screen all the information needed (SYN-Events and the Event Graph of the program under examination) for the user to create the valid SYN-sequences.

A. Case Study

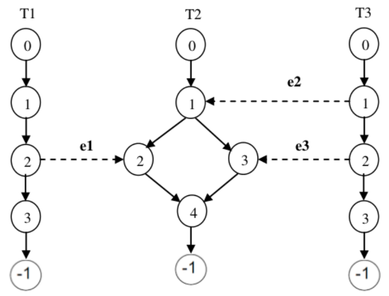
This section illustrates step by step how the new coverage criteria can be applied to a small multi-threaded XC program with communication bugs.

XC is a concurrent and real-time programming language intended for targeting XMOS multi-threaded real-time processors. XC applies the message-passing paradigm for communication between concurrent threads running on the same or different processors. Communication is achieved over channels where each channel provides a synchronous, point-to-point connection between two threads over which data may be exchanged.

Table I summarizes the the concurrency and the communication statements of XC programming language, that are relevant to the scope of this work. A channel is declared using the keyword `chan`. A channel is used in two threads and each use implicitly refers to one of its two channel ends. Data is output to a channel using the output operator <: and input from a channel using the input operator :>. Channels are lossless, in the sense that data output in one thread is guaranteed to be delivered for input by another thread. This stalls a program if one output in one thread does not have a corresponding input in another, or if the amount of data output on one end of the



(a)



(b)

Fig. 3. Example (a) of EIAG with Straightforward SYN-events and (b) with alternative SYN-events

TABLE I. STATEMENTS OF XC PROGRAMMING LANGUAGE, PROVIDING THE CONCURRENCY AND THE COMMUNICATION BETWEEN THREADS, NEEDED TO BE EXTRACTED FROM THE AST

Functionality	Expression	Description
Declaration of channel	`channel C;`	`C` is the name of the channel
Pass channel parameter to a function	`void Function(chanend C)`	`C` is the name of the channel
Sending data through a channel	`cout <: X;`	`cout` is the name of the channel, `X` can be a variable or constant value
Receiving data through a channel	`cin :> X;`	`cin` is the name of the channel, `X` can be a variable or a data type (e.g. int)

channel, does not equal the amount of data consumed at the input end [15].

In this test case, our aim is to demonstrate the full bug exposure capability of the proposed coverage metric using the prototype tool. Another contribution of this example is to show the added value of this coverage metric over the existing conventional ones. A bug free program written in XC is introduced. The specifications for the communication between the threads of the program will be given. Then we introduce bugs into the program to change its communication protocol. Then the program will be given as an input to our prototype tool and the coverage tasks will be collected. Based on the collected coverage tasks, test cases will be created to capture the introduced bugs and also check for communication protocol compliance.

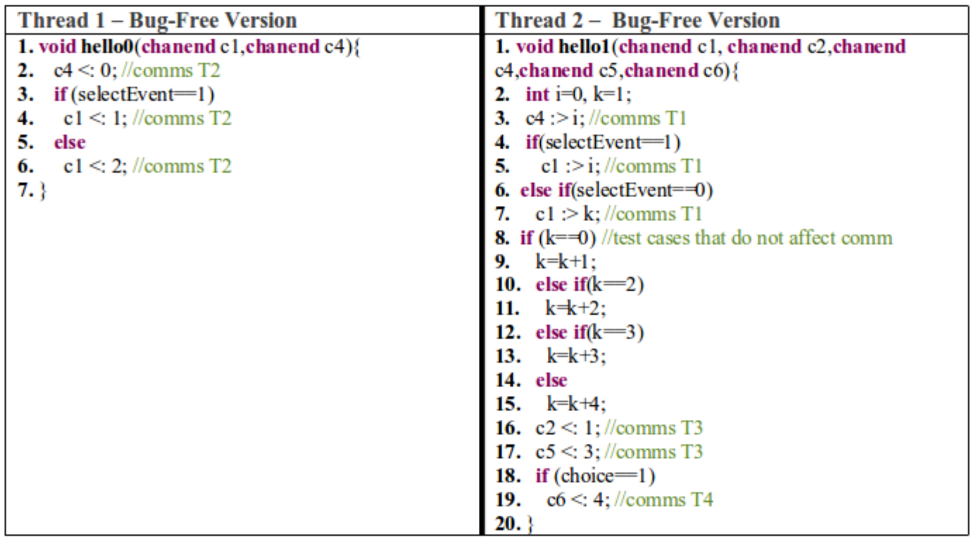
Four threads are used in this test case and two global variables `choice` and `selectEvent` to control all the if branches in the program. The values of these two global variables determine the control flow during execution time and hence the SYN-sequences executed each time.

Figure 4 shows the initial versions of `Thread 1` and `Thread 2`. No bugs were indroduced in these threads.

`Thread 1` uses only two channels to communicate with `Thread 2`, by sending data to it. There are three sending events and one branch (line 3–6) which will affect the SYN-sequences of the program. `Thread 2` uses five channels. Channels c_1 and c_4 are used to receive data from thread1, channels c_2 and c_5 are used to send data to `Thread 3` and finally channel c_6 is used to send data to `Thread 4`. The if statement on lines 4–7 affects the communication and the SYN-sequences of the program. Similarly this happens with the if statement on lines 18–19. The if statement at lines 8–15 is not affecting the communication of the program but it adds more test cases when validating the program using conventional coverage metrics, such as statement coverage. This will be used later to compare existing conventional criteria with our new coverage metric.

Figure 5 compares the initial and the buggy version of `Thread 3`. `Thread 3` uses 5 channels. In the initial version, channels c_2 and c_5 are used to receive data from `Thread 2` and channels c_3, c_7 and c_8 are used to send data to `Thread 4`. The if statement in lines 8–9 affects the communication of the program and thus the SYN-sequences that can be executed. In the buggy version a bug is introduced at line 9. One of the most common sources of errors in programming is mistyping and sometimes it is difficult to locate such errors. Here we change c_8 <: s`;` to c_7 `:>` `s;`. This change leaves the receive statement on channel c_8 in `Thread 4`, without a corresponding send statement, so if that receive statement is reached the input of the channel will be waiting for data that will never come. This will finally crash the whole program execution. To understand the effect of this change we have to examine the initial and the buggy version of `Thread 4` also.

Figure 6 compares the initial and the buggy version of the `Thread 4`. `Thread 4` uses 4 channels. In the initial version, channels c_3, c_7 and c_8 are used to receive data from `Thread 3` and channel c_6 is used to receive data from `Thread 2`. The if statements in lines 6–9 affects the communication and the SYN-sequences of the program. According to the value of the variable `choice`, either receive event c_6 or c_8 will be executed. In the buggy version of the program we change the receive statement c_6 :> int`;` to a send statement c_6 <: 4`;`, and the structure of the if statements as shown in lines 6–13 in order to have both events regarding channels c_6 and c_8 in both branches of the if statement. The first change introduces a logical error because the send event in `Thread 2` on channel c_6, will be left without a corresponding receive statement. This will also crash the program when this send statement is reached. The second change on the structure of the if statement will introduce a functional error because the program no longer complies with the initial communication protocol. In other words feasible invalid SYN-sequences will be introduced.



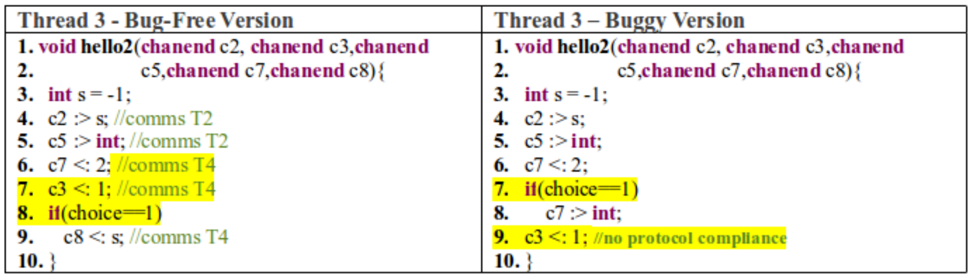
Thread 1 – Bug-Free Version

```
void hello0(chanend c1,chanend c4){
  c4 <: 0; //comms T2
  if (selectEvent==1)
    c1 <: 1; //comms T2
  else
    c1 <: 2; //comms T2
}
```

Thread 2 – Bug-Free Version

```
void hello1(chanend c1, chanend c2,chanend
c4,chanend c5,chanend c6){
 int i=0, k=1;
 c4 :> i; //comms T1
 if(selectEvent==1)
   c1 :> i; //comms T1
 else if(selectEvent==0)
  c1 :> k; //comms T1
 if (k==0) //test cases that do not affect comm
  k=k+1;
 else if(k==2)
   k=k+2;
 else if(k==3)
   k=k+3;
 else
   k=k+4;
 c2 <: 1; //comms T3
 c5 <: 3; //comms T3
 if (choice==1)
   c6 <: 4; //comms T4
}
```

Fig. 4. The initial versions of the thread 1 and 2 of the program under test.



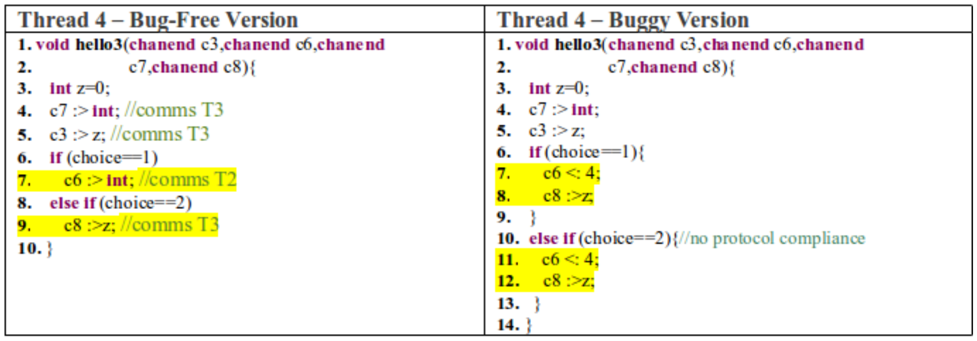
Thread 3 - Bug-Free Version

```
void hello2(chanend c2, chanend c3,chanend
              c5,chanend c7,chanend c8){
int s = -1;
c2 :> s; //comms T2
c5 :> int; //comms T2
c7 <: 2; //comms T4
c3 <: 1; //comms T4
if(choice==1)
   c8 <: s; //comms T4
}
```

Thread 3 – Buggy Version

```
void hello2(chanend c2, chanend c3,chanend
              c5,chanend c7,chanend c8){
int s = -1;
c2 :> s;
c5 :> int;
c7 <: 2;
if(choice==1)
   c7 :> int;
c3 <: 1; //no protocol compliance
}
```

Fig. 5. The initial and the buggy version of the thread3 of the program under test.



Thread 4 – Bug-Free Version

```
void hello3(chanend c3,chanend c6,chanend
              c7,chanend c8){
 int z=0;
 c7 :> int; //comms T3
 c3 :> z; //comms T3
 if (choice==1)
   c6 :> int; //comms T2
 else if (choice==2)
   c8 :>z; //comms T3
}
```

Thread 4 – Buggy Version

```
void hello3(chanend c3,chanend c6,chanend
              c7,chanend c8){
 int z=0;
 c7 :> int;
 c3 :> z;
 if (choice==1){
   c6 <: 4;
   c8 :>z;
 }
 else if (choice==2){//no protocol compliance
   c6 <: 4;
   c8 :>z;
 }
}
```

Fig. 6. The initial and the buggy version of the `Thread 3` of the program under test.

The EIAG of the initial version of the program is demonstrated in Figure 7. The valid SYN-sequences of the program was extracted by contacting control flow analysis on the EIAG as described at section III-C and shown in table II.

B. Coverage Tasks collection of the buggy version of the Program under test

The buggy version of the program under test was first compiled using our tool (customized version of the proprietary XC compiler) to extract the information needed for the generation of the coverage tasks. The collected coverage tasks were checked against the coverage criteria properties, defined in Section III-A, to assess their validity. Table III shows all the extracted coverage tasks. For the convenience of presentation the notation of coverage tasks is linearised e.g. $T^{z}_{c_j} snd_x$ will

TABLE II. THE SYN-SEQUENCES SPECIFIED BY THE COMMUNICATION PROTOCOL OF THE INITIAL VERSION OF THE PROGRAM UNDER TEST.

	SYN-sequences
S_1	$\langle e_1, e_2, e_4, e_5, e_7, e_8 \rangle$
S_2	$\langle e_1, e_3, e_4, e_5, e_7, e_8 \rangle$
S_3	$\langle e_1, e_2, e_4, e_5, e_7, e_8, e_6 \rangle$
S_4	$\langle e_1, e_2, e_4, e_5, e_7, e_8, e_9 \rangle$
S_5	$\langle e_1, e_3, e_4, e_5, e_7, e_8, e_6 \rangle$
S_6	$\langle e_1, e_3, e_4, e_5, e_7, e_8, e_9 \rangle$

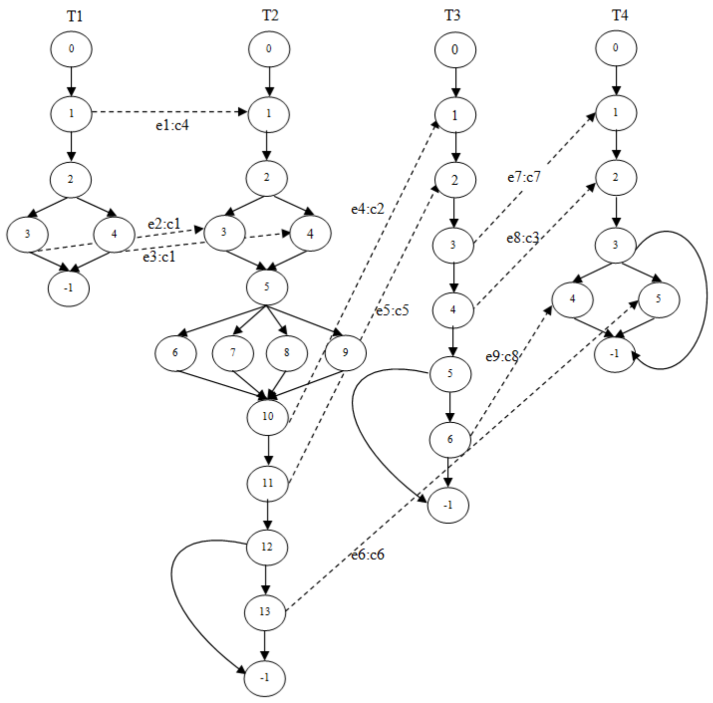


Fig. 7. The EIAG of the initial version of the program.

now be $\texttt{cj(snd}_x^{Tz}\texttt{)}$. The cells coloured in light grey indicate that those coverage tasks are not conforming to their criteria properties and they possibly reveal potential bugs in the code.

First we start looking at coverage tasks generated from the criteria higher in the hierarchy and then move to the lower levels, until clarifying the source of the problem for each case. Starting with the SYN-event, the Cartesian product generated for channel c_6 is empty. This indicates either a channel declared and never used (dead code) either a channel that only one of its endpoints is used (either receive or corresponding send statement is missing). Moving to Criteria 2 and 3 for channel 6, there is an empty set of rcv-statements and a non-empty set of send-statements, indicating a clear case of faulty communication. These can cause the program to stall if one of the unmatched send-statements is executed. Using the same approach the unmatched receive statements on channel 8 ($c_8(rcv0), c_8(rcv1)$) are captured.

In the case of SYN-event e'_7 and using the property of SYN-events that an event must happen in two different threads, a potential bug is captured, since its send and corresponding receive statement are in the same thread. Although the compiler allows this case, is usually meaningless because there is no reason to send data from a thread and receive that same data in the same thread through a channel and typically indicates a typing mistake.

The two events e'_3, e'_4 coloured in dark gray color should not be valid according to the program specifications. This is dependent upon the program semantics and only the user can exclude from the coverage tasks as not feasible and invalid events.

It is important to note that the buggy version of the program is executed exactly the same as the initial version without any defects caused by the existence of the bugs. The reason is that the bugs are guarded by the if statements and they will only be executed when their if branch is reached. Consider a huge program composed of hundreds of lines of code, and somewhere in the middle is an if statement guarding an unmatched send or receive statement. The code of the if statement will only be executed when a variable takes a specific value with a probability of 1/100. Using conventional coverage criteria will be very difficult to detect and will need to check all the 100 possible values of the variable. On the other hand using our coverage metric and our tool this bug can be captured by just collecting the coverage tasks and before even running any test cases.

TABLE III. THE COVERAGE TASKS COLLECTED BY THE CREATED PROTOTYPE TOOL FOR THE PROGRAM UNDER TEST.

Channel	snd-statements	rcv-statements	SYN-events
c_4	$c_4(snd_0^{T1})$	$c_4(rcv_0^{T2})$	$e'_0 : c_4(snd_0^{T1}, rcv_0^{T2})$
c_1	$c_1(snd_0^{T1})$	$c_1(rcv_0^{T2})$	$e'_1 : c_1(snd_0^{T1}, rcv_0^{T2})$
			$e'_2 : c_1(snd_0^{T1}, rcv_1^{T2})$
	$c_1(snd_1^{T1})$	$c_1(rcv_1^{T2})$	$e'_3 : c_1(snd_1^{T1}, rcv_0^{T2})$
			$e'_4 : c_1(snd_1^{T1}, rcv_1^{T2})$
c_2	$c_2(snd_0^{T2})$	$c_2(rcv_0^{T3})$	$e'_5 : c_2(snd_0^{T2}, rcv_0^{T3})$
c_5	$c_5(snd_0^{T2})$	$c_5(rcv_0^{T3})$	$e'_6 : c_5(snd_0^{T2}, rcv_0^{T3})$
c_6	$c_6(snd_0^{T2})$ $c_6(snd_1^{T4})$ $c_6(snd_2^{T4})$		
c_7	$c_7(snd_0^{T3})$	$c_7(rcv_0^{T3})$	$e'_7 : c_5(snd_0^{T3}, rcv_0^{T3})$
		$c_7(rcv_1^{T4})$	$e'_8 : c_5(snd_0^{T3}, rcv_1^{T4})$
c_3	$c_3(snd_0^{T3})$	$c_3(rcv_0^{T4})$	$e'_9 : c_4(snd_0^{T3}, rcv_0^{T4})$
c_8		$c_8(rcv_0^{T4})$ $c_8(rcv_1^{T4})$	

C. *Correction of the potential bugs and checking the protocol compliance*

Up to this point the tool was able to create all the coverage tasks up to Criterion 4 automatically and also capture a number of potential bugs. In order to be able to move forward by collecting the coverage tasks for the last criterion of SYN-sequence, the user has to correct all the bugs identified by applying the earlier criteria. Then all the feasible SYN-sequence coverage tasks can be collected by the tool. SYN-sequences are related to the communication protocol of a program. By collecting them the user is able to compare them to those intended by the program specifications, which are feasible and valid. The process followed by the user is:

1) Create the set V of all the user intented SYN-sequences using the specifications for the program under test. This must be the set of feasible, valid in the program written by the programmer.
2) Extract the set F of all the feasible SYN-sequences from the implemented program, as demonstrated in Section III-C.
3) Compare the two sets. If the program is bug free, then those that should be feasible are the same as those that are feasible in the actual code and so $V = F$. If the program is buggy, then that is not the case and one of the following cases applies:
   - $V \cap F = \{\}$, V and F have no elements in common thus all of the SYN-sequences are invalid.
   - $V \subset F$, Feasible, invalid SYN-sequences exist in the implementation.
   - $F \subset V$, Some valid SYN-sequences are missing from the implementation.
   - $V \cap F \neq \{\}$, Some valid SYN-sequences are also feasible. The SYN-sequences outside the intersection of V with F need to be further investigated.

This process requires the user input due to a lack of program semantics (i.e. its control and data flow). The user will also repeat the above process as many times as needed until the two sets are equal. During this process, a number of EIAGs will be produced for each of the intermediate revised program versions, to be used by static analysis for the feasible SYN-sequences extraction and also to assist the user in comparing the two sets V and F. Test generation is outside the scope of this paper. The user may utilize existing test generation techniques to target the coverage tasks in this process.

In our test case, after correcting the initial bugs captured statically by the extraction of the coverage tasks up to criterion SYN-event in the hierarchy, the EIAG is created as given in Figure 8. Comparing this graph with the one of the correct version of the program in Figure 7 the changes introduced in the original code to introduce communication protocol incompliance can be seen. Events e_7, e_10 and e_11 in Figure 8, do not conform to the initial communication protocol. Moreover, e_10 can cause a deadlock if the branch on node 4 of thread $T3$ is taken. This is a case of out of order, crossing events creating cyclic dependencies on their execution. An EIAG has the property of exposing whenever this occurs and is an indication for the either existance of deadlocks or a dead code in the program if the code is not reachable. Static analysis on the the graph can reveal these cases.

D. *Comparison of the new coverage metric proposed with the existence conventional coverage metrics*

`Thread 2` of the program under test introduces an if statement (lines 8–15) with four branches. The purpose of this if statement is to demonstrate some of the added value of our coverage metric over the old conventional ones. Our coverage model specifically extracts the communication of a message-passing multi-threaded program and allows the user to check the correct communication by using only the test cases affecting the communication. In our case, there is no need to examine the 4 extra branches under that if statement to fully test the communication of the program. Using the conventional statement coverage metric a larger set of test cases will be needed to achieve 100% testing for the program communication than our coverage model. While the example used to illustrate the expressive power of the proposed coverage criteria is clearly small, this difference can be large for complex programs, significantly increasing testing complexity.

V. Conclusion

In this work, we have proposed a coverage model that complements existing ones, but is designed to capture the communication behaviour of multi-threaded message-passing programs. The new coverage model is structured over several layers, increasing in expressive power and complexity.

A prototype tool was developed, implementing the new coverage model, targeted on the XMOS XC language, to evaluate its bug capturing capabilities. The prototype tool offers 100% extraction of the communication of a multi-threaded message-passing program and at the same time the coverage tasks for the first 5 criteria in the hierarchy can be automatically extracted. By just extracting the coverage tasks before running any test cases on the program, some bugs are captured, such as unmatched send or receive statements or SYN-events happening in the same thread. Finally communication protocol compliance testing can be conducted by the user, by comparing the feasible SYN-sequences extracted by the tool with the valid SYN-sequences stated in the program specifications. This process might be repeated until the program is 100% bug free. The use of the EIAG graph also enables the detection of the deadlocks in the program. Moreover, an added value over the conventional existing coverage metrics, such as statement coverage, is that this coverage metric is specific for the communication in a program. Achieving 100% communication coverage requires fewer tests than 100% conventional code coverage.

The coverage model is transferable to any programming language that uses message passing. As future work we will try to apply this model to a different message-passing language such as MPI. An extension to the current implementation is also needed to cover the full spectrum of the XC language, e.g. considering loops.

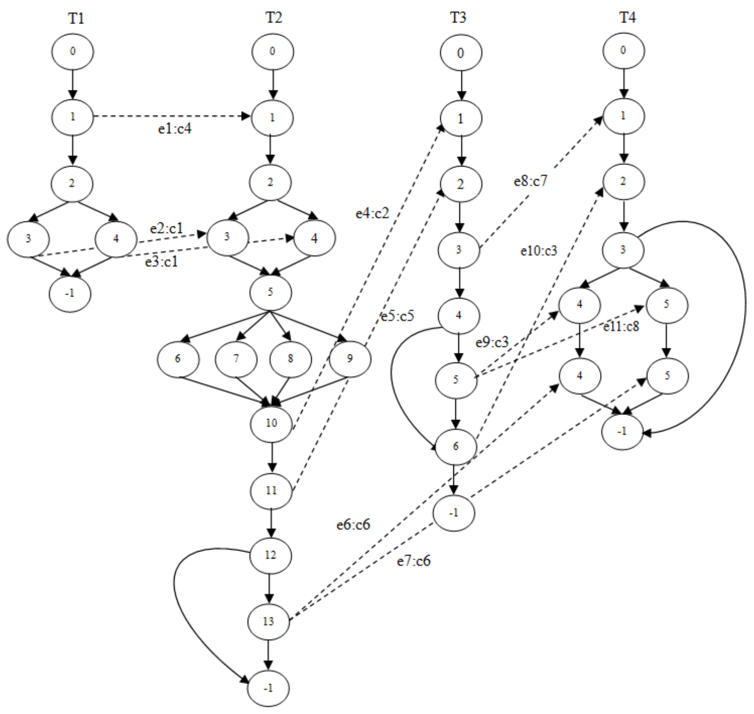


Fig. 8. The EIAG after correcting the statically captured bugs up to SYN-event crterion.

Acknowledgments

The research leading to these results has received funding from the European Union 7th Framework Programme under grant agreement 318337, ENTRA - Whole-Systems Energy Transparency. The authors would also like to thank Xiaorong Yu for her initial work leading to the establishment of the coverage models as presented in this paper.

References

[1] B. Alessio, C. John, and A. Cosimo. A tool for testing of parallel and distributed programs in message-passing environments. In *Proceedings of 9th IEEE Mediterranean Electrotechnical Conference*, pages 1308 – 1312. MELECON, 1998.

[2] B. Arkady, F. Eitan, M. Yonit, N. Yarden, and U. Shmuel. Applications of synchronization coverage. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 206 – 212. ACM, 2005.

[3] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold, 1990.

[4] H. Chockler, O. Kupferman, and M. Vardi. Coverage metrics for formal verication. In Springer-Verlag, editor, *In 12th Advanced Research Working Conference on Correct Hardware Design and Verication Methods*, volume 2860, pages 111 – 125, 2003.

[5] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel. The impact of concurrent coverage metrics on testing effectiveness. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:232 – 241, 2013.

[6] T. Katayama, Z. Furukawa, and K. Ushijima. A method for structural testing of Ada concurrent programs using the event interactions graph. In *Proceedings of the Third Asia-Pacific Software Engineering Conference*, pages 335 – 364, 1996.

[7] T. LeBlanc and E. Markatos. Shared memory vs. message passing in shared-memory multiprocessors. In *Proceedings of the Fourth IEEE Symposium/Parallel and Distributed Processing*, pages 254 – 263, New York, NY, USA, 1992. IEEE.

[8] Y. Lei, E. Wong, and A. Novel. Framework for non-deterministic testing of message-passing programs. In *Proceedings of the Ninth IEEE International Symposium on High-Assurance Systems Engineering*, pages 66 – 75, New York, NY, USA, 2005. IEEE.

[9] Y. Liang, S. Li, H., C. Zhang, and H. Chengde. Timing-sequence testing of parallel programs. *Journal of Computer Science and Technology*, 15(1):84 – 95, 2002.

[10] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *In FSE*, pages 533 – 536, 2007.

[11] J. Pedersen. Classification of programming errors in parallel message passing systems. In *Proceedings of Communicating Process Architectures*, pages 363 – 376. IOS Press, 2006.

[12] K. Tai and R. Carver. *Testing of Distributed Programs. Parallel and Distributed Computing Handbook*. Van Nostrand Reinhold, 1996.

[13] S. Tasiran and K. Keutzer. Coverage metrics forx functional validation of hardware designs. *IEEE Design and Test of Computers*, page 36 45, 2001.

[14] S. Ur and A. Ziv. Off-the-shelf vs. custom made coverage models, which is the one for you? In *In proceedings of STAR98: the 7th international*, 1998.

[15] D. Watt. *Programming XC on XCore XS1Devices*. XMOS, 1st edition, 2009.

[16] E. J. Weyuker. More experience with data flow testing. *IEEE Trans. Softw. Eng.*, 19(9):912 – 919, Sept. 1993.

[17] W. Wong, Y. Lei, and X. Ma. Effective generation of test sequences for structural testing of concurrent programs. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 539 – 548, 2005.

Attachment D3.1.6

Operational Semantics for XC.

Technical Report.

Operational Semantics for XC

Technical Report

Nina Bohr
Roskilde University

1 Operational Semantics for XC

The intension in this semantics is that in each transition step (ϵ-transition) for the program we progress all possible threads. All threads which can in themselves make a transition step (an ϵ-transition) will do that and at the same time all communications where both send and get are offered on the same channel will do a communicating step.

1.1 Syntax

This semantics is closely tied to the XC language as it builds on the properties that in XC 1) parallel threads cannot make any updates to shared variables and 2) in (at least the subset of XC we look at) communication channels have two ends owned by exactly one thread each, so there is no race on being the first to send/get on a given channel.

A parallel program may evolve in many different execution orders depending on hardware and program external properties. In this semantics a given execution order is chosen building on the not very realistic assumption that each basic statement takes the same time. Due to the properties of XC mentioned above, this choice will always give the same final state as any other execution order for a terminating program, and for looping or blocking programs each thread will extend its execution as far as possible (OBS needs to be proven – if it is what we want to prove). It is possible to change the semantics to execute in other orders by adding timing to basic steps maybe depending on an extended description of state.

We let x (variable names) range over a set $\mathcal{N}$ of names.

$$\begin{array}{rl}
\text{(Values)} & V ::= n \mid \alpha \\
\text{(Expressions)} & E ::= V \mid x \mid E_1 + E_2 \mid f(E) \mid \{S\} \\
\text{(Statements)} & S ::= \mathsf{skip} \mid \mathsf{return}\, E \mid x := E \mid S_1 \,;S_2 \mid S_1 \parallel S_2 \mid \\
& \quad\;\; \mathsf{let}\, x = E \,\mathsf{in}\, S \mid \mathsf{send}\, E_1\, E_2 \mid x := \mathsf{get}\, E \\
\text{(Labels)} & L ::= V\alpha! \mid \alpha? \\
\text{(Label-List)} & \mathcal{L} ::= [\,]^p \mid L : \mathcal{L} \\
\text{(MLabels)} & L^m ::= \alpha!V \mid \alpha?V \text{ (understood as match labels)} \\
\text{(MLabel-Lists)} & \mathcal{L}^m ::= [\,]^m \mid L^m : \mathcal{L}^m \\
\text{(Transition-Possibility-Marker)} & t ::= 0 \mid 1 \\
\text{(Transition-labelling)} & \mathfrak{L} ::= \epsilon \mid L^m \;\; \text{(obs: the same as } \epsilon \mid \alpha!V \mid \alpha?V \text{, not a list)}
\end{array}$$

Label-lists are intended to express all communication-offerings to the statement-external surroundings present in a given statement at a given time in execution. Transition-possibility-markers are intended to indicate by 0/1 whether the statement is capable of making an ϵ-transition, that is a transition not communicating with the outside surroundings. Two mlabel-lists for two sides in a given par-statement are intended to contain all communications possible between the two sides at a given time of execution. Hence, they will always have the same length.

For the XC Language there can never be more than two labels using the same channel in a label-list. In erroneous programs it is possible to have two gets or two sends on the same channel, this will block further progress of the involved two threads. In an mlabel-list there cannot be more than one mlabels using the same channel.

For mlabels $L_1 = \alpha?V$ and $L_2 = \alpha!V$ we define an erasure function by overbar to derive associated labels $\bar{L_1} = \alpha?$ and $\bar{L_2} = V\alpha!$. This extends to mlabel-lists $\bar{\mathcal{L}}^m$.

For two label-lists we define a function $Match$ which will find all matching sends and gets on the same channel and add the value from the send-label

to the get-mlabel. $Match(\mathcal{L}_1, \mathcal{L}_2) = (\mathcal{L}_1', \mathcal{L}_2', \mathcal{L}_1^m, \mathcal{L}_2^m)$ such that ($\approx$ meaning contains the same labels):
$(\mathcal{L}_1 \approx \mathcal{L}_1' \uplus \bar{\mathcal{L}}_1^m)$ and $(\mathcal{L}_2 \approx \mathcal{L}_2' \uplus \bar{\mathcal{L}}_2^m)$ and
$(V\alpha! \in \mathcal{L}_1 \wedge \alpha? \in \mathcal{L}_2 \Rightarrow \alpha!V \in \mathcal{L}_1^m \wedge \alpha?V \in \mathcal{L}_2^m)$ and
$(V\alpha! \in \mathcal{L}_2 \wedge \alpha? \in \mathcal{L}_1 \Rightarrow \alpha!V \in \mathcal{L}_2^m \wedge \alpha?V \in \mathcal{L}_1^m)$ and
$(V\alpha! \in \mathcal{L}_1 \wedge \alpha? \notin \mathcal{L}_2 \Rightarrow \alpha V! \in \mathcal{L}_1')$ and
$(V\alpha! \in \mathcal{L}_2 \wedge \alpha? \notin \mathcal{L}_1 \Rightarrow \alpha V! \in \mathcal{L}_2')$ and
$(\alpha? \in \mathcal{L}_1 \wedge V\alpha! \notin \mathcal{L}_2 \Rightarrow \alpha? \in \mathcal{L}_1')$ and
$(\alpha? \in \mathcal{L}_2 \wedge V\alpha! \notin \mathcal{L}_1 \Rightarrow \alpha? \in \mathcal{L}_2')$ and
the $Match$ function must derive the list in an order-specific way to ensure uniqueness (e.g. left-to-right, the actual choice of order is irrelevant).

We let $\mathcal{V}$ denote the set of values. A store σ is a finite mapping from $\mathcal{N}$ to $\mathcal{V}$, written as $\{x_1{=}V_1, \cdots, x_n{=}V_n\}$. We let $\oplus$ denote the disjoint union on stores.

We let $V_\perp$ range over $\mathcal{V} \cup \{\perp\}$. We write (in rule ex-let-2-step)

$$\sigma = \sigma' \oplus \{x{=}V_\perp\}$$

when either (1) $x \notin \text{dom}(\sigma)$ and $V_\perp = \perp$ or (2) $\sigma = \sigma' \oplus \{x{=}V\}$ and $V_\perp = V$ for some σ' and V.

In this version we apply $\mathcal{L}^m$ mlabel-list from left to right when we derive the result on statement and store. This is however arbitrary, as any order will give the same result. An order is chosen to ensure it is done in only one way.

$\mathcal{R}(E, \sigma, \mathcal{L}, t)$ relates expressions, stores, label-lists and transition-possibility-markers and $\mathcal{R}(S, \sigma, \mathcal{L}, t)$ relates statements, stores, label-lists and transition-possibility-markers. The meaning of the $\mathcal{R}$ relation is intended to be that a given statement or expression in a given store is offering all and only the communications in $\mathcal{L}$ to the surroundings and t express whether the statement is capable of making any ϵ-transition. Defined below.

$(\sigma_1 \otimes \sigma_2 \mid \sigma)$ is used in rules where there are two parallel threads, σ is the original store before a transition, σ_1 is a modification of σ by one thread and σ_2 is the modification of σ by the other thread. For a wellformed XC-program the first three implications below are enough as parallel threads cannot make updates to shared variables. The last two rules are added for completeness, the last rule gives more than one possibility for the value of x in the derived store. It is the let-rule than ensures, that σ, σ_1 and σ_2 all have the same domain. By the let-rule local variables are not visible to the surroundings. That allocation always will choose a fresh location is implicit.

$(\sigma_1 \otimes \sigma_2 \mid \sigma)$ is defined by:
$\mathit{domain}((\sigma_1 \otimes \sigma_2 \mid \sigma)) = \mathit{domain}(\sigma) = \mathit{domain}(\sigma_1) = \mathit{domain}(\sigma_2)$
$(\{x{=}V\} \in \sigma \wedge \{x{=}V\} \in \sigma_1 \wedge \{x{=}V\} \in \sigma_2) \Rightarrow \{x{=}V\} \in (\sigma_1 \otimes \sigma_2 \mid \sigma)$
$(\{x{=}V\} \in \sigma \wedge \{x{=}V_1\} \in \sigma_1 \wedge \{x{=}V\} \in \sigma_2) \Rightarrow \{x{=}V_1\} \in (\sigma_1 \otimes \sigma_2 \mid \sigma)$
$(\{x{=}V\} \in \sigma \wedge \{x{=}V\} \in \sigma_1 \wedge \{x{=}V_2\} \in \sigma_2) \Rightarrow \{x{=}V_2\} \in (\sigma_1 \otimes \sigma_2 \mid \sigma)$

$(\{x{=}V\} \in \sigma \wedge \{x{=}V'\} \in \sigma_1 \wedge \{x{=}V'\} \in \sigma_2) \Rightarrow \{x{=}V'\} \in (\sigma_1 \otimes \sigma_2 \mid \sigma)$
$(\{x{=}V\} \in \sigma \wedge \{x{=}V_1\} \in \sigma_1 \wedge \{x{=}V_2\} \in \sigma_2) \Rightarrow (\{x{=}V_1\} \in (\sigma_1 \otimes \sigma_2 \mid \sigma) \vee \{x{=}V_2\} \in (\sigma_1 \otimes \sigma_2 \mid \sigma))$

1.2 Small-step semantics

1.2.1 Transition rules

Comment: In many of the following rules transitions are label with $\mathfrak{L}$ which can by either ϵ or one of the mlabels $\alpha!V$ and $\alpha?V$. The transitions with mlabels are only used to derive the result of transitions over a list of mlabels $\mathcal{L}^m$ in rules (ex-sync-11), (ex-sync-10) and (ex-sync-01) where we search for a derivation of one label at a time (rule ex-$\mathcal{L}^m$). The R relation takes care of most of finding out the basis for which transitions are possible.

Comment: A value is not related by $\mathcal{R}$, *skip* is not related by $\mathcal{R}$.

$$\frac{}{\langle x, \sigma \oplus \{x{=}V\}\rangle \xrightarrow{\epsilon} \langle V, \sigma\rangle} \tag{ev-var}$$

$$\frac{\langle E_1, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle E_1', \sigma'\rangle}{\langle E_1 + E_2, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle E_1' + E_2, \sigma'\rangle} \tag{ev-add-1-step}$$

$$\frac{\langle E_2, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle E_2', \sigma'\rangle}{\langle n_1 + E_2, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle n_1 + E_2', \sigma'\rangle} \tag{ev-add-2-step}$$

$$\frac{}{\langle n_1 \mathbin{\mathsf{+}} n_2, \sigma\rangle \xrightarrow{\epsilon} \langle n_1 + n_2, \sigma\rangle} \qquad \text{(ev-add)}$$

$$\frac{f(x)\{S\} \text{ is a global function}}{\langle f(E), \sigma\rangle \xrightarrow{\epsilon} \langle \{\mathsf{let}\, x = E \,\mathsf{in}\, S\}, \sigma\rangle} \qquad \text{(ev-call)}$$

$$\frac{}{\langle \{\mathsf{return}\, V\}, \sigma\rangle \xrightarrow{\epsilon} \langle V, \sigma\rangle} \qquad \text{(ev-stm-return)}$$

$$\frac{\langle S, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle S', \sigma'\rangle}{\langle \{S\}, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle \{S'\}, \sigma'\rangle} \qquad \text{(ev-stm-step)}$$

$$\frac{\langle E, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle E', \sigma'\rangle}{\langle \mathsf{return}\, E, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle \mathsf{return}\, E', \sigma'\rangle} \qquad \text{(ex-return-1-step)}$$

$$\frac{\langle S_1, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle S_1', \sigma'\rangle}{\langle S_1 \,;\, S_2, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle S_1' \,;\, S_2, \sigma'\rangle} \qquad \text{(ex-seq-1-step)}$$

$$\frac{}{\langle \mathsf{skip} \,;\, S_2, \sigma\rangle \xrightarrow{\epsilon} \langle S_2, \sigma\rangle} \qquad \text{(ex-seq-1-skip)}$$

$$\frac{}{\langle \mathsf{return}\, V \,;\, S_2, \sigma\rangle \xrightarrow{\epsilon} \langle \mathsf{return}\, V, \sigma\rangle} \qquad \text{(ex-seq-1-return)}$$

$$\frac{\langle E, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle E', \sigma'\rangle}{\langle x := E, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle x := E', \sigma'\rangle} \qquad \text{(ex-asg-1-step)}$$

$$\frac{}{\langle x := V, \sigma \oplus \{x{=}V'\}\rangle \xrightarrow{\epsilon} \langle \mathsf{skip}, \sigma \oplus \{x{=}V\}\rangle} \qquad \text{(ex-asg)}$$

$$\frac{\langle E, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle E', \sigma'\rangle}{\langle \mathsf{let}\, x = E \,\mathsf{in}\, S, \sigma\rangle \xrightarrow{\mathfrak{L}} \langle \mathsf{let}\, x = E' \,\mathsf{in}\, S, \sigma'\rangle} \qquad \text{(ex-let-1-step)}$$

$$\frac{\langle S, \sigma \oplus \{x{=}V\}\rangle \xrightarrow{\mathfrak{L}} \langle S', \sigma' \oplus \{x{=}V'\}\rangle}{\langle \mathsf{let}\, x = V \,\mathsf{in}\, S, \sigma \oplus \{x{=}V_\perp\}\rangle \xrightarrow{\mathfrak{L}} \langle \mathsf{let}\, x = V' \,\mathsf{in}\, S', \sigma' \oplus \{x{=}V_\perp\}\rangle}$$

(ex-let-2-step)

$$\frac{}{\langle \mathsf{let}\, x = V \,\mathsf{in}\, \mathsf{skip}, \sigma\rangle \xrightarrow{\epsilon} \langle \mathsf{skip}, \sigma\rangle} \qquad \text{(ex-let-2-skip)}$$

$$\frac{}{\langle \mathsf{let}\, x = V \,\mathsf{in}\, \mathsf{return}\, V', \sigma\rangle \xrightarrow{\epsilon} \langle \mathsf{return}\, V', \sigma\rangle} \qquad \text{(ex-let-2-return)}$$

$$\frac{}{\langle \mathsf{skip} \parallel \mathsf{skip}, \sigma\rangle \xrightarrow{\epsilon} \langle \mathsf{skip}, \sigma\rangle} \qquad \text{(ex-par-skip-skip)}$$

$$\frac{\langle S_2, \sigma\rangle \xrightarrow{\epsilon} \langle S_2', \sigma'\rangle}{\langle \mathsf{skip} \parallel S_2, \sigma\rangle \xrightarrow{\epsilon} \langle \mathsf{skip} \parallel S_2', \sigma'\rangle} \qquad \text{(ex-par-skip-left)}$$

$$\frac{\langle S_1, \sigma\rangle \xrightarrow{\epsilon} \langle S_1', \sigma'\rangle}{\langle S_1 \parallel \mathsf{skip}, \sigma\rangle \xrightarrow{\epsilon} \langle S_1' \parallel \mathsf{skip}, \sigma'\rangle} \qquad \text{(ex-par-skip-right)}$$

Notice in the rules below that a transition labeled with a match-label-LIST can possibly be doing nothing, if the list is empty (rules (ex-$[\,]^m$) and (ex-$\mathcal{L}^m$))

The following rule has the side condition: $Match(\mathcal{L}_1, \mathcal{L}_2) = (\mathcal{L}_1', \mathcal{L}_2', \mathcal{L}_1^m, \mathcal{L}_2^m)$

$$\frac{\begin{array}{c}\mathcal{R}(S_1, \sigma, \mathcal{L}_1, 1) \quad \mathcal{R}(S_2, \sigma, \mathcal{L}_2, 1) \quad \langle S_1, \sigma\rangle \xrightarrow{\epsilon} \langle S_1', \sigma_1'\rangle \quad \langle S_2, \sigma\rangle \xrightarrow{\epsilon} \langle S_2', \sigma_2'\rangle \\ \langle S_1', \sigma_1'\rangle \xrightarrow{\mathcal{L}_1^m} \langle S_1'', \sigma_1''\rangle \quad \langle S_2', \sigma_2'\rangle \xrightarrow{\mathcal{L}_2^m} \langle S_2'', \sigma_2''\rangle\end{array}}{\langle S_1 \parallel S_2, \sigma\rangle \xrightarrow{\epsilon} \langle S_1'' \parallel S_2'', (\sigma_1'' \otimes \sigma_2'' \mid \sigma)\rangle}$$

(ex-sync-11)

The following rule has the side condition: $Match(\mathcal{L}_1, \mathcal{L}_2) = (\mathcal{L}_1', \mathcal{L}_2', \mathcal{L}_1^m, \mathcal{L}_2^m)$

$$\frac{\begin{array}{c}\mathcal{R}(S_1, \sigma, \mathcal{L}_1, 1) \quad \mathcal{R}(S_2, \sigma, \mathcal{L}_2, 0) \quad \langle S_1, \sigma\rangle \xrightarrow{\epsilon} \langle S_1', \sigma_1'\rangle \\ \langle S_1', \sigma_1'\rangle \xrightarrow{\mathcal{L}_1^m} \langle S_1'', \sigma_1''\rangle \quad \langle S_2, \sigma_1'\rangle \xrightarrow{\mathcal{L}_2^m} \langle S_2'', \sigma_2''\rangle\end{array}}{\langle S_1 \parallel S_2, \sigma\rangle \xrightarrow{\epsilon} \langle S_1'' \parallel S_2'', (\sigma_1'' \otimes \sigma_2'' \mid \sigma)\rangle} \qquad \text{(ex-sync-10)}$$

The following rule has the side condition: $Match(\mathcal{L}_1, \mathcal{L}_2) = (\mathcal{L}_1', \mathcal{L}_2', \mathcal{L}_1^m, \mathcal{L}_2^m)$

$$\frac{\begin{array}{c}\mathcal{R}(S_1, \sigma, \mathcal{L}_1, 0) \quad \mathcal{R}(S_2, \sigma, \mathcal{L}_2, 1) \quad \langle S_2, \sigma\rangle \xrightarrow{\epsilon} \langle S_2', \sigma_2'\rangle \\ \langle S_1, \sigma_2'\rangle \xrightarrow{\mathcal{L}_1^m} \langle S_1'', \sigma_1''\rangle \quad \langle S_2', \sigma_2'\rangle \xrightarrow{\mathcal{L}_2^m} \langle S_2'', \sigma_2''\rangle\end{array}}{\langle S_1 \parallel S_2, \sigma\rangle \xrightarrow{\epsilon} \langle S_1'' \parallel S_2'', (\sigma_1'' \otimes \sigma_2'' \mid \sigma)\rangle} \qquad \text{(ex-sync-01)}$$

$$\frac{\begin{array}{c}\mathcal{R}(S_1,\sigma,\mathcal{L}_1,0) \quad \mathcal{R}(S_2,\sigma,\mathcal{L}_2,0)\\ \langle S_1,\sigma\rangle \xrightarrow{\mathcal{L}_1^m} \langle S_1',\sigma_1'\rangle \quad \langle S_2,\sigma\rangle \xrightarrow{\mathcal{L}_2^m} \langle S_2',\sigma_2'\rangle\end{array}}{\langle S_1 \parallel S_2,\sigma\rangle \xrightarrow{\epsilon} \langle S_1' \parallel S_2', (\sigma_1' \otimes \sigma_2' \mid \sigma)\rangle} Match(\mathcal{L}_1,\mathcal{L}_2) = (\mathcal{L}_1',\mathcal{L}_2',\mathcal{L}_1^m,\mathcal{L}_2^m) \wedge \mathcal{L}_1^m \neq [\,]$$

(ex-sync-00)

$$\frac{\langle E_1,\sigma\rangle \xrightarrow{\mathfrak{L}} \langle E_1',\sigma'\rangle}{\langle \mathsf{send}\, E_1\, E_2,\sigma\rangle \xrightarrow{\mathfrak{L}} \langle \mathsf{send}\, E_1'\, E_2,\sigma'\rangle} \qquad \text{(ex-send-1-step)}$$

$$\frac{\langle E_2,\sigma\rangle \xrightarrow{\mathfrak{L}} \langle E_2',\sigma'\rangle}{\langle \mathsf{send}\, c\, E_2,\sigma\rangle \xrightarrow{\mathfrak{L}} \langle \mathsf{send}\, c\, E_2',\sigma'\rangle} \qquad \text{(ex-send-2-step)}$$

Comment: Recall $\alpha!V$ and $\alpha?V$ are mlabels

$$\frac{}{\langle \mathsf{send}\, \alpha\, V,\sigma\rangle \xrightarrow{\alpha!V} \langle \mathsf{skip},\sigma\rangle} \qquad \text{(ex-send-m)}$$

$$\frac{\langle E_1,\sigma\rangle \xrightarrow{\mathfrak{L}} \langle E_1',\sigma'\rangle}{\langle x := \mathsf{get}\, E_1,\sigma\rangle \xrightarrow{\mathfrak{L}} \langle x := \mathsf{get}\, E_1',\sigma'\rangle} \qquad \text{(ex-get-1-step)}$$

$$\frac{}{\langle x := \mathsf{get}\, \alpha, \sigma \oplus \{x{=}V'\}\rangle \xrightarrow{\alpha?V} \langle \mathsf{skip}, \sigma \oplus \{x{=}V\}\rangle} \qquad \text{(ex-get-m)}$$

$$\frac{\langle E,\sigma\rangle \xrightarrow{\mathfrak{L}} \langle E',\sigma'\rangle}{\langle \mathsf{if}\ (E)\ S_1\ \mathsf{else}\ S_2,\sigma\rangle \xrightarrow{\mathfrak{L}} \langle \mathsf{if}\ (E')\ S_1\ \mathsf{else}\ S_2,\sigma'\rangle} \qquad \text{(ex-ifnz-1-step)}$$

$$\frac{}{\langle \mathsf{if}\ (n)\ S_1\ \mathsf{else}\ S_2,\sigma\rangle \xrightarrow{\epsilon} \langle S_1,\sigma\rangle} n \neq 0 \qquad \text{(ex-ifnz-true)}$$

$$\frac{}{\langle \mathsf{if}\ (0)\ S_1\ \mathsf{else}\ S_2,\sigma\rangle \xrightarrow{\epsilon} \langle S_2,\sigma\rangle} \qquad \text{(ex-ifnz-false)}$$

$$\frac{}{\langle \mathsf{while}\ (E)\ S,\sigma\rangle \xrightarrow{\epsilon} \langle \mathsf{if}\ (e)\ S\ ; \mathsf{while}\ (E)\ S\ \mathsf{else}\,\mathsf{skip},\sigma\rangle} \qquad \text{(ex-while)}$$

Comment. Recall L^m is $\alpha!V$ or $\alpha?V$. The following two rules are only used to find the result of transition over an $\mathcal{L}^m$ and should not be the reason for any nondeterminism as it is only used in the rules (ex-sync-11), (ex-sync-10) and (ex-sync-01) where we are searching for a derivation for one L^m at a time (rule ex-$\mathcal{L}^m$).

$$\frac{\langle S_1, \sigma\rangle \xrightarrow{L^m} \langle S_1', \sigma'\rangle}{\langle S_1 \parallel S_2, \sigma\rangle \xrightarrow{L^m} \langle S_1' \parallel S_2, \sigma'\rangle} \qquad \text{(ex-par-1-m)}$$

$$\frac{\langle S_2, \sigma\rangle \xrightarrow{L^m} \langle S_2', \sigma'\rangle}{\langle S_1 \parallel S_2, \sigma\rangle \xrightarrow{L^m} \langle S_1 \parallel S_2', \sigma'\rangle} \qquad \text{(ex-par-2-m)}$$

$$\frac{}{\langle S, \sigma\rangle \xrightarrow{[\,]^m} \langle S, \sigma\rangle} \qquad \text{(ex-}[\,]^m\text{)}$$

$$\frac{\langle S, \sigma\rangle \xrightarrow{L_1^m} \langle S_1, \sigma_1\rangle \quad \langle S_1, \sigma_1\rangle \xrightarrow{\mathcal{L}^m} \langle S_2, \sigma_2\rangle}{\langle S, \sigma\rangle \xrightarrow{L_1^m : \mathcal{L}^m} \langle S_2, \sigma_2\rangle} \qquad \text{(ex-}\mathcal{L}^m\text{)}$$

1.2.2 $\mathcal{R}$ rules

Comment: A value is not related by $\mathcal{R}$, *skip* is not related by $\mathcal{R}$

The following two rules express that for each ϵ-transition-rule with empty premiss, there is a corresponding instance of the $\mathcal{R}$-relation. The two rules may be replaced by a larger number of $\mathcal{R}$-rules with empty premiss.

$$\frac{\left(\dfrac{}{\langle E, \sigma\rangle \xrightarrow{\epsilon} \langle E', \sigma'\rangle}\right)}{\mathcal{R}(E, \sigma, [\,]^p, 1)} \qquad \text{(R-e-step)}$$

$$\frac{\left(\dfrac{}{\langle S, \sigma\rangle \xrightarrow{\epsilon} \langle S', \sigma'\rangle}\right)}{\mathcal{R}(S, \sigma, [\,]^p, 1)} \qquad \text{(R-s-step)}$$

$$\frac{\mathcal{R}(E_1, \sigma, \mathcal{L}, t)}{\mathcal{R}(E_1 + E_2, \sigma, \mathcal{L}, t)} \qquad \text{(R-add-1-step)}$$

$$\frac{\mathcal{R}(E_2, \sigma, \mathcal{L}, t)}{\mathcal{R}(n_1 \mathbin{\texttt{+}} E_2, \sigma, \mathcal{L}, t)} \qquad \text{(R-add-2-step)}$$

$$\frac{f(x)\{S\} \text{ is a global function}}{\mathcal{R}(f(E), \sigma, [\,]^p, 1)} \qquad \text{(R-ev-call)}$$

$$\frac{\mathcal{R}(S, \sigma, \mathcal{L}, t)}{\mathcal{R}(\{S\}, \sigma, \mathcal{L}, t)} \qquad \text{(R-stm-step)}$$

$$\frac{\mathcal{R}(E, \sigma, \mathcal{L}, t)}{\mathcal{R}(\mathsf{return}\, E, \sigma, \mathcal{L}, t)} \qquad \text{(R-return-1-step)}$$

$$\frac{\mathcal{R}(S_1, \sigma, \mathcal{L}, t)}{\mathcal{R}(S_1\,; S_2, \sigma, \mathcal{L}, t)} \qquad \text{(R-seq-1-step)}$$

$$\frac{\mathcal{R}(E, \sigma, \mathcal{L}, t)}{\mathcal{R}(x := E, \sigma, \mathcal{L}, t)} \qquad \text{(R-asg-1-step)}$$

$$\frac{\mathcal{R}(E, st, \mathcal{L}, t)}{\mathcal{R}(\mathsf{let}\, x = E\, \mathsf{in}\, S, \sigma, \mathcal{L}, t)} \qquad \text{(R-let-1-step)}$$

$$\frac{\mathcal{R}(S, \sigma \oplus \{x{=}V\}, \mathcal{L}, t)}{\mathcal{R}(\mathsf{let}\, x = V\, \mathsf{in}\, S, \sigma \oplus \{x{=}V_\perp\}, \mathcal{L}, t} \qquad \text{(R-let-2-step)}$$

$$\frac{\mathcal{R}(S_2, \sigma, \mathcal{L}, t)}{\mathcal{R}((\mathsf{skip} \parallel S_2), \sigma, \mathcal{L}, t)} \qquad \text{(R-par-skip-left)}$$

$$\frac{\mathcal{R}(S_1, \sigma, \mathcal{L}, t)}{\mathcal{R}((S_1 \parallel \mathsf{skip}), \sigma, \mathcal{L}, t)} \qquad \text{(R-par-skip-right)}$$

$$\frac{\mathcal{R}(S_1, \sigma, \mathcal{L}_1, 0) \quad \mathcal{R}(S_2, \sigma, \mathcal{L}_2, 0)}{\mathcal{R}((S_1 \parallel S_2), \sigma, \mathcal{L}_1 \uplus \mathcal{L}_2, 0)} Match(\mathcal{L}_1, \mathcal{L}_2) = (\mathcal{L}_1, \mathcal{L}_2, [\,], [\,]) \qquad \text{(R-par-0)}$$

$$\frac{\begin{array}{c}\mathcal{R}(S_1, \sigma, \mathcal{L}_1, t_1) \quad \mathcal{R}(S_2, \sigma, \mathcal{L}_2, t_2) \\ (t_1 = 1 \vee t_2 = 1 \vee \mathcal{L}^m \neq [\,])\end{array}}{\mathcal{R}((S_1 \parallel S_2), \sigma, \mathcal{L}'_1 \uplus \mathcal{L}'_2, 1)} Match(\mathcal{L}_1, \mathcal{L}_2) = (\mathcal{L}'_1, \mathcal{L}'_2, \mathcal{L}^m_1, \mathcal{L}^m_2)$$

(R-par-1)

$$\frac{\mathcal{R}(E_1, \sigma, \mathcal{L}, t)}{\mathcal{R}(\textsf{send}\, E_1\, E_2, \sigma, \mathcal{L}, t)} \qquad \text{(R-send-1-step)}$$

$$\frac{\mathcal{R}(E_2, \sigma, \mathcal{L}, t)}{\mathcal{R}(\textsf{send}\, c\, E_2, \sigma, \mathcal{L}, t)} \qquad \text{(R-send-2-step)}$$

Comment: Recall $V\alpha!$ and $\alpha?$ are labels

$$\frac{}{\mathcal{R}(\textsf{send}\, \alpha\, V, \sigma, \{V\alpha!\}, 0)} \qquad \text{(R-send)}$$

$$\frac{\mathcal{R}(E_1, \sigma, \mathcal{L}, t)}{\mathcal{R}(x := \textsf{get}\, E_1, \sigma, \mathcal{L}, t} \qquad \text{(R-get-1-step)}$$

$$\frac{}{\mathcal{R}(x := \textsf{get}\, \alpha, \sigma \oplus \{x{=}V'\}, \{\alpha?\}, 0)} \qquad \text{(R-get)}$$

$$\frac{\mathcal{R}(E, \sigma, \mathcal{L}, t)}{\mathcal{R}(\textsf{if}\ (E)\ S_1\ \textsf{else}\ S_2, \sigma, \mathcal{L}, t)} \qquad \text{(R-ifnz-1-step)}$$

1.2.3 Multi-step reduction

$$\frac{}{\langle \textsf{skip}, \sigma \rangle \longrightarrow^* \langle \textsf{skip}, \sigma \rangle / 0} \qquad \text{(step-skip)}$$

$$\frac{\langle S_0, \sigma_0 \rangle \xrightarrow{\epsilon} \langle S_1, \sigma_1 \rangle \quad \langle S_1, \sigma_1 \rangle \longrightarrow^* \langle S_2, \sigma_2 \rangle / r_2}{\langle S_0, \sigma_0 \rangle \longrightarrow^* \langle S_2, \sigma_2 \rangle / r_1 + r_2} \qquad \text{(step-trans)}$$