



ENTRA
318337
Whole-Systems ENergy TRAnsparency

Preliminary Report and Demo on Energy-Aware Tools

Deliverable number:	D1.1
Work package:	Energy-Aware Software Engineering (WP1)
Delivery date:	1 October 2014 (24 months)
Actual date:	3 December 2014
Nature:	Report
Dissemination level:	PU
Lead beneficiary:	IMDEA Software Institute
Partners contributed:	Roskilde University, University of Bristol, IMDEA Software Institute, XMOS Limited

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

Short description:

This deliverable describes our current view about the kind of tools needed for supporting energy transparency. It is a point of view influenced by the project's first two years of research, and the emerging understanding of the needs of an energy-aware software development tool chain. It first gives an overview of the knowledge relevant to an energy-aware software engineer about some of the key aspects affecting a program's energy consumption. The purpose is to identify the kinds of information that is needed and the tool support that could provide such information in order to assist in energy optimization. It does not yet attempt to present a detailed process or method, but rather to provide a broad context for the more detailed descriptions, given later, of the initial prototype tools for energy-aware software engineering, integrating the implementations of the energy analysis, verification and optimization techniques developed in work packages 2, 3 and 4. It also includes a prototype implementation demonstration and an experimental study of such tools.

Contents

1	Introduction	3
2	Key Elements and Tool Support for Energy-aware Software Engineering	5
2.1	Developing Low Energy Systems	5
2.2	Tools Providing Information about Factors Affecting Energy Consumption	7
2.3	Tools for Static Energy Profiling	7
2.4	Tools for Analysis of Concurrency and Communication in Multi-threaded Applications	8
2.5	Tools Supporting Compiler-based Energy Optimizations	9
2.6	Tools Supporting Trade-offs of Energy Against Quality	9
3	Energy-aware Software Development Tools	11
3.1	Multi-level Energy Analysis and Verification Tool based on HC IR Transformation	11
3.1.1	Usage and Interface	13
3.1.2	Methodology and Scenarios for Using the Tool	17
3.1.3	Architecture of the Tool	21
3.1.4	Evaluation and Experimental Results	24
3.1.5	Demonstration of the Tool	28
3.2	Multi-level Mapper Tool	32
3.2.1	Using the <code>--analysis</code> Option	32
3.2.2	Using the <code>--ISAFunCFG</code> Option	42
3.3	A Tool for Flow and Synchronization Analysis of Multi-threaded XC Programs .	42
3.3.1	Usage and Interface	44
3.3.2	Generation of a <i>Run</i> Relation	44
3.3.3	Analysis of Threads and Communication Structure	47
3.3.4	Generating the <i>Reach</i> Relation	47
3.4	Optimization via Dynamic Voltage and Frequency Scaling (DVFS) and Task Scheduling	49
3.4.1	Usage and Interface	50
3.4.2	Methodology and Scenarios for Using the Tool	52
3.4.3	Architecture of the Tool	53
3.4.4	Evaluation and Experimental Results	53

4	Work in Progress	55
4.1	Worst Case Energy Consumption Using Implicit Path Enumeration	55
4.1.1	Integer Linear Programming (ILP) Formulation	55
4.1.2	Structural Constraints	56
4.1.3	Functionality Constraints	56
4.2	Tools for Horn Clause Verification	57
4.2.1	Combining Off-the-shelf Tools: Experiments	61

1 Introduction

This document presents a preliminary overview of the ENTRA tools supporting energy transparency, in the context of the project’s emerging understanding, after two years, of the needs of an energy-aware software development tool chain. In Section 2 we give a broad view of some of the key aspects of a program’s behaviour that an energy-aware software engineer needs to know. We do not yet attempt to present a detailed process or method, but focus on the kinds of information that is needed. We then summarise the tool support that could provide such information and assist in energy optimization.

Section 2 is intended to provide the context for the more detailed descriptions of tools in Section 3, where we present four tool components that are the result of work performed in work packages WP2, WP3 and WP4:

- A multi-level energy analysis and verification tool based on a transformation into a block based *intermediate program representation* (“HC IR” from now on), where each block is written as a Horn Clause.
- A multi-level mapper tool for propagating energy model information defined at one level upwards to other levels, and for creating a mapping of code locations between different intermediate code representations, assembly code (ISA-level code) and source code.
- A tool for flow and synchronization analysis of multi-threaded XC programs.
- A tool for optimization via Dynamic Voltage and Frequency Scaling (DVFS) and task scheduling.

The first two of these, as the names suggest, focus on the analysis and modelling challenges in performing analyses at different levels, in particular intermediate code (LLVM IR) and assembly code (ISA-level code). They are described in Sections 3.1 and 3.2 respectively. The third tool, described in Section 3.3, deals with analysis of source code, for which an energy model has not yet been developed, but which can nonetheless yield information relevant to energy-awareness. The fourth tool, described in Section 3.4, is the first optimization tool under development in the project.

These tools are components, used in the project for experimental studies. We are also considering tool development and integration strategies. It is a research challenge in itself, quite apart from the tool functionalities, to consider how they might be integrated into an energy-aware tool chain. The ENTRA project has two streams of work:

1. Integration in the XC tool chain, which constitutes the *proof of concept* of the project, where different components, such as the tools (or parts of them) presented in Section 3 can be used, once they are in a mature and stable enough state. Such tools exploit the existing infrastructure of the compiler, intermediate code, and other development tools.
2. Stand-alone tool development, which allows a more general investigation and the study of other application areas. This development stream allows the invention of new approaches, the experimentation and evaluation of already developed components, the identification of new components and the investigation of a wider range of scenarios.

Tool components from the second stream could in the longer term migrate to the XC tool chain when they are in a mature enough state.

Sections 3 and 4 give further details about the tools under development, adding information about the individual building blocks, providing information relevant to developers of energy-aware software development tools.

2 Key Elements and Tool Support for Energy-aware Software Engineering

The ENTRA Description of Work states that “the project is built around the central concept of *energy transparency* at every stage of the software lifecycle.” In this section we discuss how, after two years of the project, this overall requirement is understood in terms of key aspects of energy-aware software engineering. As explained in the Introduction, this document does not intend to define methods, procedures or guidelines; these will emerge after experience with using energy-aware tools, and the final deliverable of the ENTRA project is expected to contain an outline of an “energy-aware software engineering method”.

This document discusses the tools needed to enable engineers to understand and quantify the impact of design decisions on energy, and to guide optimizations that are either manual or automatic. Section 3 contains more detailed descriptions and demonstrations of ENTRA tools.

2.1 Developing Low Energy Systems

Although physical energy is only consumed by hardware, it is the software driving the hardware that governs how much energy will be consumed by the hardware whilst completing a task. Previous deliverables (D6.1 and D5.1) have detailed how energy can be saved. In order to make the software low energy, various actors therefore have to work together.

Starting with the programmer; their primary task is to write code efficiently. For example, if a choice of algorithms is available, then the algorithm with the lowest computational complexity is probably a good starting point. Almost all good programming practices such as avoiding replication of computation are helpful.

But once the right algorithms are picked, there are different implementation strategies that each have their unique energy profile. Picking the best of these strategies can have a large impact on the energy consumption. Indeed, it is not implausible that an algorithm that is slightly less time efficient can be made more energy efficient.

In order to achieve this task the programmer will use a toolbox similar to the toolbox designed for designing programs that perform fast enough. The traditional parts of this toolbox are the tools that physically measure power consumption whilst the program is running and the compiler that optimises for energy usage. Akin to the profiler and optimiser used when designing programs.

Less traditional is the tool that statically predicts energy usage. Tools like that are common place in hardware design, where energy usage and timing closure of a new chip are analysed before the chip is physically produced; the software equivalent of static timing closure exists in

the X MOS tool chain (XTA - X MOS Timing Analyser), this project aims to develop tools that statically predict energy usage.

This matters, as such a tool will be able to reason about energy usage over a variety of conditions, and not just the one condition under which energy is actually measured. For example, different input data may cause different energy usages, and finding a worst case is useful to be able to reason about sizing power supplies and batteries.

Such a tool is not trivial, and in particular it requires analysis of concurrency and communication patterns in multi-threaded programs. The project aim is to provide methods and prototype tools that will enable a static energy profiler to be developed.

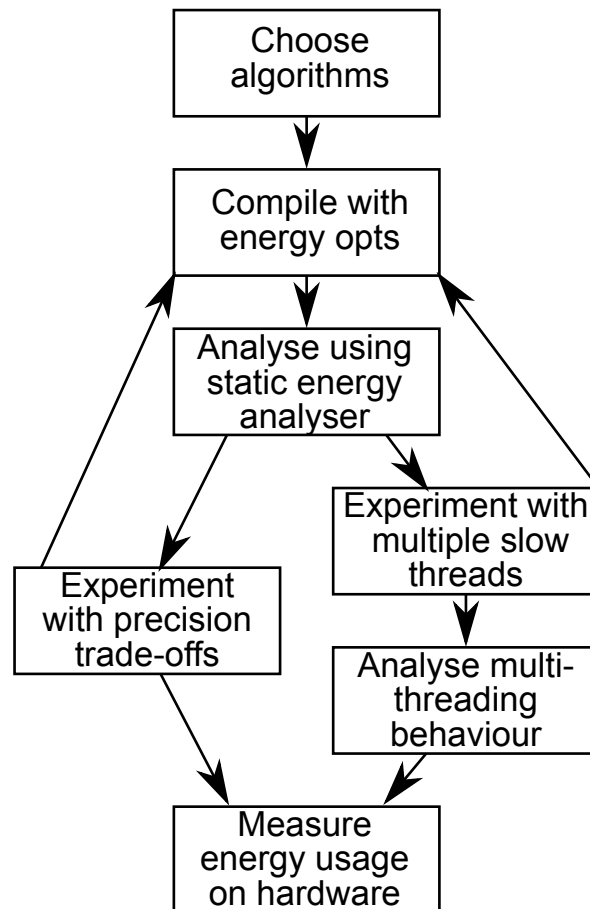


Figure 1: Steps through using the tools

The process that the software engineer will flow through is sketched in Figure 1. The general flow is from top to bottom, but inevitably there will be iterations through several steps. However, the flow is organised in such a way that the iterations are local, and fast; i.e., compilation and static energy analysis are fast steps; that can be repeated often at little cost. Experimentations

with using multiple threads that can run at a lower frequency (and hence a lower voltage), and experimentations with reduced precision all requires iterations through the tools, but this is a quick iteration. The following subsections describe the tools in more detail.

2.2 Tools Providing Information about Factors Affecting Energy Consumption

The high-level objective of energy-aware software engineering is to permit energy efficiency to be a first-class design goal of the software engineer. That is, design and implementation decisions can be made with energy efficiency in mind, already in the early phases of the development process. The alternative to energy-awareness is to test energy consumption of the implemented system and then re-engineer the system (if it is possible to do so) if the energy target is not met.

In order to make energy-related design and implementation decisions, the software engineer needs understanding of the factors affecting the energy consumption of programs. These are typically hidden from the programmer in high-level languages, where software engineering abstractions focus on functionality rather than resource usage. Moreover, even at assembly code level, few programmers have detailed expertise about what causes high or low energy consumption.

This problem has always existed in embedded systems development, since determining the power requirements of a system is a key part of the overall design process. In recent years, energy consumption of individual applications running on general purpose computers has become a widespread concern. Every user of a smartphone or tablet knows that certain apps drain the battery faster than others. Designing low-power apps is thus becoming increasingly important, but energy-aware development tools are lacking.

2.3 Tools for Static Energy Profiling

We define *static energy profiling* as the inference of information about energy usage of a program, which plays a central role in energy-aware software engineering. The tools under development in ENTRA aim to provide information such as the following.

- The total energy consumption of a complete program;
- the total energy consumption of each part of a program, at selected levels of granularity such as functions, blocks or statements;
- the energy consumed by a program fragment between given events such as inter-thread communication or external I/O operations.

All of these may be given

- in parameterized form, where the parameter might be the size of input data or some hardware setting;
- at different levels of the software stack, such as source code, intermediate code (LLVM IR) or assembly code.
- as worst, best or average case estimates.

The purpose of static energy profiling include enabling energy-aware design tasks such as

- verifying that energy usage of an application meets given energy specifications, such as maximum energy;
- identifying energy “hot spots” – program parts that consume the most energy;
- identifying energy distribution over time, over different cores, over threads etc.

When combined with timing information, energy profiling will yield information on the power dissipation profile. (The incorporation of timing models in energy profiling has not yet been tackled in the project).

2.4 Tools for Analysis of Concurrency and Communication in Multi-threaded Applications

Investigation of ENTRA benchmarks and case studies has shown that crucial energy savings can be obtained by understanding the behaviour of a multi-threaded program’s threads, especially with respect to communication and I/O operations. A typical question that the energy-aware software engineer asks is “How slow can a given program or thread be run, while still meeting its internal and external communication deadlines?”, since lowering the clock speed is an important energy-saving measure. Analysis of load balance and data dependencies between threads gives the software engineer useful insights that can be used to design more energy efficient multi-threaded code.

Tools supporting such analyses include the following.

- Dependency analysis with respect to I/O operations. The purpose of such an analysis is to find the instructions on which time-critical operations such as an I/O operation depend. Instructions that are not urgently needed for time-critical operations can sometimes be delayed until after the time-critical point, allowing the other dependent instructions to be run more slowly.

- Analysis of thread load. Uneven distribution of computation over threads can lead to energy inefficiencies, since energy is spent while threads are idling.
- Analysis of data dependencies within thread computations, where the relative load on threads depends on data values passed between them.

2.5 Tools Supporting Compiler-based Energy Optimizations

As with other resources such as time and memory, energy efficiency can often be improved by compiler optimizations, taking advantage of hardware-specific features that are not visible to the software engineer. Supporting tools can assist the compiler, providing information needed to take advantage of available opportunities for optimization.

The ENTRA approach allows the software engineer to add assertions to programs, and these can be passed to the compiler. In addition, or in combination with assertions, tools for dataflow analyses such as the following can be employed.

- Precision or range analysis of numeric expressions allowing the compiler to select energy-optimal representations;
- constant propagation and other forms of specialization, which can lead to both time and energy savings;
- dataflow analysis across modules allowing specialization of interface calls and link-time optimization.

2.6 Tools Supporting Trade-offs of Energy Against Quality

The choice of saving energy at the cost of quality will be an increasingly important decision for the energy-aware software engineer. Some applications (such as media processing, interactive games, or non-safety-critical control system) might be capable of several “energy modes”. In a low-energy mode, the quality (such as image or sound fidelity, game response time, or precision in control) might be lower, whereas if desired a more high-energy mode can increase quality at the expense of higher energy consumption.

All the tools discussed above in principle contribute to enabling such choices to be taken into account during design. The software engineer, having access to precise energy profiling information, can decide how much energy can be saved by decreasing quality. In addition, tools supporting analysis of quality are needed, for example estimating error accumulation using lower-precision numbers, or response-time delays by lowering clock speeds.

3 Energy-aware Software Development Tools

This section describes prototype tools corresponding to the development streams mentioned in Section 1. These tools are intended to be used by (embedded) application developers or energy model constructors.

There is a subsection for each tool, describing the functionality it provides and the input and output. There is also an explanation of interfaces offered by the tool (GUI, command line, or both), and how the user (e.g., the application developer) interacts with them in order to perform tasks like analyzing, verifying or optimizing programs in different scenarios. Information on the architecture of the tool is also provided. Such information is mainly for tool developers and includes an overall diagram of the tool, showing its different components and information flow between them.

The tools (or parts of them) described in this section could potentially be integrated into existing production tool chains, such as the XC development environment, once they are in a mature and stable enough state. For example, the prototype tool presented in Section 3.1 allows experimentation with energy analysis at different levels of abstraction. During later stages of the project, when experimental studies have stabilised, functional components used in such tool can be integrated with a compiler tool chain (e.g., for XC). In other words, the prototype presented in Section 3.1 is intended for ENTRA project partners (for experimentation, research and development) rather than XC developers.

3.1 Multi-level Energy Analysis and Verification Tool based on HC IR Transformation

In this section we describe an experimental prototype tool for analysis and verification of energy, execution time and general resource usage properties.

The user (typically the energy-efficient software developer) can interact with the tool through both, a GUI and a command line interface. Both interfaces are described in Section 3.1.1.

Main functionality Currently, the user can select to perform two main kinds of actions:

- *Analysis*: This action is used to estimate the energy consumed and time spent by the execution of XC programs and each of its procedures (even when there are parts not developed yet.). Such information is given in general as functions on some properties of the input data (e.g., range of integers or length of arrays) and can be used by developers of energy-efficient software to make informed design decisions (e.g., redesigning the most energy

consuming parts of the programs, using alternative data structures, ...) or optimizing the XC programs, either manually or using a (semi-)automatic optimization tool.

- *Verification*: This action is used to prove whether resource usage specifications are met or not, or to infer conditions under which such specifications are met.

Input to the tool The input to the tool is a file with a program encoded in any of the four following languages: XC source, Instruction Set Architecture (ISA), or HC IR. These are recognized by their respective extensions: `.xc` for XC, `.asm` for ISA, `.ll` for LLVM IR, or `.pl` for HC IR.

For an input file in HC IR format, it is the responsibility of the programmer to include assertions in Internal Assertion Language (IAL) format describing the models for particular resources used for the analysis / verification (see [EG13] for a detailed description of the IAL). Nevertheless, CiaoPP provides some packages for predefined resource models in IAL format. For example, the user can include the package `ciaopp(xcore/model/energy)` to use the energy model described in D2.2 [EKG14], or the package `ciaopp(xcore/model/time)` to use a timing model. For a file in a format different from HC IR (i.e., XC, LLVM IR or ISA), the tool automatically uses the energy and timing models defined by the packages above.

Output of the tool The outcome of the analysis (or the verification) is subsequently included as assertions in the output file in one of the two following formats. For XC, LLVM IR and ISA, the results are formatted in the front end aspect of the Common Assertion Language described in deliverable D2.1 [EG13]. For HC IR the results are formatted in the IAL, i.e., the internal aspect of the Common Assertion Language.

Main features: multi-level analysis, experimental The tool integrates two different instantiations of the general resource analysis framework described in deliverable D1.1. Both instantiations use energy models defined at the ISA level (see deliverable D2.2 [EKG14]), but one of them performs the analysis at the ISA level (see deliverable D3.1 and [LKS⁺13]) and the other one performs the analysis at the LLVM IR level (see deliverable D3.2 and Attachment D3.2.4.).

In this sense, the tool is a multi-level analysis and verification tool, and the user can select at which level (LLVM IR or ISA) the analysis is to be performed. In order to perform the analysis at the LLVM IR (resp. ISA) level, the LLVM IR (resp. ISA) corresponding to the input XC file is first generated (by using the standard `xcc` compiler), and then transformed into HC IR using the LLVM IR (resp. ISA) HC IR translation. The HC IR is then analyzed by the analysis engine. Technical details about such translations can be found in deliverable D3.2, Attachment D3.2.4.


The selection of the analysis level has an impact on the accuracy of the results and on the class of programs that can be analyzed. Thus, the tool allows experimentation with energy (and time) analysis at different levels of abstraction. It is intended for ENTRA project partners, rather than XC developers.

The tool has been integrated in an existing tool chain for experimentation, the CiaoPP system, leveraging the environment for program analysis, verification and optimization offered by it, which uses HC IR as internal program representation and is based on modular, incremental abstract interpretation. During later stages of the project, when experimental studies have stabilised, functional components from the CiaoPP system could potentially be integrated with a compiler tool chain.

3.1.1 Usage and Interface

The user can interact with the tool through a Graphical User Interface (GUI) or through a Unix command line. The GUI is mainly used in the project to facilitate experimentation by tool developers and to experiment with aspects of user interaction such as source assertions.

Graphical User Interface (GUI) The CiaoPP Graphical User Interface is based on GNU Emacs. Assuming the CiaoPP system has been properly installed (as described in the Ciao/CiaoPP reference manual [BCH⁺11]), the GUI is automatically started up as soon as the file loaded into the current Emacs buffer is recognized by the tool (e.g., XC, ISA or HC IR files are recognized by their respective extensions `.xc`, `.asm`, `.ll`, or `.pl`).

Within the GUI, the processing of files is governed by a menu, where the user can select different options. Clicking on the icon  in the buffer displays that menu, which looks (depending on the options available in the current CiaoPP version) like the “Preprocessor Option Browser” shown in Figure 2. Except for the first and last lines, which refer to loading or saving a menu configuration (a predetermined set of selected values for the different menu options), each line corresponds to an option the user can select. In the following, we describe the options relevant for the analysis and verification of XC programs.

- The `Action Group` determines which action to perform on the input file (i.e., the file loaded in the Emacs buffer). The user can set the option to two different values:
 - When option is set to the `analyze` value, the tool performs the *analysis* previously described on the input file.
 - When the option is set to the `verify` value, the tool performs the *verification* action previously described, by checking the assertions present in the input file. In this case

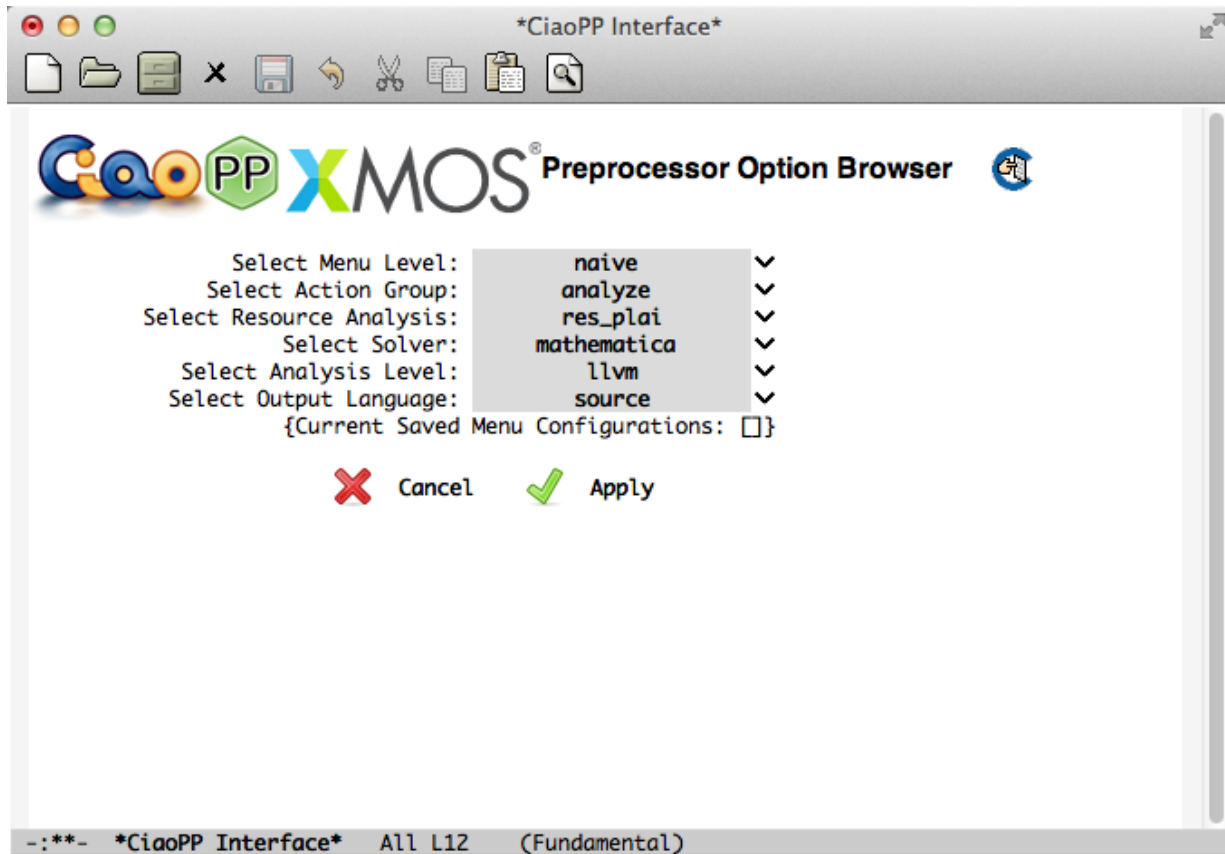


Figure 2: CiaoPP menu showing different options.

an analysis is first performed – as it would be the case if the option were assigned to the `analyze` value – then the results are compared with the input assertions to be checked.

- The `Resource Analysis` determines which resource analysis engine should be used to run the resource analysis. The user can set the option to two different values:
 - When the option is set to the `resources` value, the tool uses the legacy version of the CiaoPP resource analysis engine. Such version is based on [NMLGH07], where the last steps of the analysis (setting up and solving recurrence equations) are not implemented as an abstract domain, and hence offers a more limited number of features than the new resource analysis, explained below.
 - When the option is set to the `res_plai` value, the tool uses a newer version of the CiaoPP resource analysis engine. This analysis has been developed within the scope of ENTRa and integrated in the CiaoPP system (see [SLGH14] or attachment D3.2.3

of D3.2 for a detailed description). Unlike the previous version, the `res_plai` engine is fully defined as an abstract domain in PLAI, the abstract interpretation framework of CiaoPP. This makes the analysis able to infer separate resource information for different calling patterns of the same procedure and handle *sized types*, among other features. *Sized types* are representations that incorporate structural (shape) information and allow expressing both lower and upper bounds on the size of a set of (recursive) data structures and their sub-structures at any position and depth (e.g., the length of a list and the size of its elements). They are fully described in [SLGBH13] or attachment D3.1.2 of deliverable D3.1.

It is recommended to use the `res_plai` resource analysis engine as it turns out to be more general and more efficient than the previous analyzer `resources`. The later can be used for the sake of comparison.

- The `Solver` option determines which recurrence equation solver must be used by the resource analysis engine. Currently, the user can set the option to two different values:
 - When the option is set to the `builtin` value (default value), the resource analysis engine uses the *builtin solver*. This solver is directly incorporated into the CiaoPP analyzer and consequently does not require the installation of any external tool. However, currently, the solver is less powerful than the external solver and therefore can lead to more imprecise analysis results.
 - The `mathematica` value forces the use of Wolfram Mathematica. In general, Mathematica is a more powerful recurrence equation solver than the builtin solver, however being an external component, it has to be installed on the machine separately from CiaoPP.
- The `Analysis Level` option allows the user to select at which level (LLVM IR or ISA) the analysis will be performed, as explained previously. For this, the user can set the option to two different values: `LLVM` or `ISA` respectively.
- The `Output Language` option specifies in which language the output file with the analysis / verification results must be written. The user can set the option to two different values:
 - When the option is set to the `source` value, the results are output in the same language as the input file.

- When the option is set to the `internal` representation value, the results are output in HC IR (with assertions in IAL).

Command line interface Besides the graphical user interface, CiaoPP also offers a specialized command line interface to perform analysis (and other actions) on XC, LLVM IR and ISA files. This command line interface provides the same functionality as the GUI described previously, and can be used by advanced users to bypass it, or by analysis tool developers to ease CiaoPP integration into a heterogeneous tool chain.

The name of the command line executable is `ciaopp_entra` and can be used as follows:

```
$ ciaopp_entra [Options] <InputFilename>
```

where `<InputFilename>`, the last argument, is the path of the input file that contains the program to be processed – as for the GUI the format of this input file is determined by the file name extension – and `[Options]` is a space separated sequence of the following possible options:

- The `--analyze` option is used to perform the resource usage analysis of the input file (equivalently to setting the `Action Group` option of the GUI to the value `analyze`.)
- The `--verify` option is used to perform the resource usage *verification* of the input file (equivalently to setting the `Action Group` option of the GUI to the value `verify`.)

In case neither the `--analyze` nor the `--verify` option is specified, no actions (analysis nor verification) are performed on the input file. However, an output file is generated. This behaviour may be useful to generate the HC IR representing the ISA or LLVM IR code of the input program.

- The `-o <OutputFileName>` option specifies that `<OutputFileName>` is the path of the target output file to be written.
- The `--oformat=<OutputFormat>` option specifies in which language the output should be written. There are two options for “`<OutputFormat>`”:
 - `HC IR`: the analysis / verification results are written in HC IR.
 - `source`: the analysis / verification results are written in the source language.
- The `--level=<level>` option determines at which level (LLVM IR or ISA) the analysis is to be performed (similarly to the `Analysis Level` option of the GUI previously explained).

- The `--req-solver=<solver>` option specifies which recurrence equation solver must be used by the resource analysis engine, similarly to the `Solver` option of the GUI previously explained. Thus, there are two different options for `<solver>`: `builtin` and `mathematica`.
- The `--help` option displays description of the command line usage including the different options described above.

3.1.2 Methodology and Scenarios for Using the Tool

Scenario 1. Deciding values for program parameters that meet an energy budget As illustrative example in this scenario we consider the development of an equaliser (XC) program using a biquad filter similar to the one presented in deliverable D5.1 (Section 2). The purpose of an equaliser is to take a signal, and to attenuate / amplify different frequency bands. This will, for example in the case of an audio signal, correct for a speaker or microphone frequency response. The energy consumed by such a program directly depends on several parameters, such as the sample rate of the signal, and the number of banks (typically between 3 and 30 for an audio equaliser). A higher number of banks enables the designer to create more precise frequency response curves.

Assume that the developer has to decide how many banks to use in order to meet an energy budget while maximizing the precision of frequency response curves at the same time. For simplicity, we assume that the rest of parameters affecting energy consumption are fixed. In this scenario, the developer writes an XC program where the number of banks is a variable, say N . Assume also that the energy constraint to be met is that an application of the biquad program should consume less than 1,300 nJ (nano Joules). Since our purpose in this section is to describe the scenarios conceptually, we do not pay too much attention to the syntax in which such energy specification is written. It could be expressed for example in the (front end) assertion language as follows:

```
#pragma entra check biquad(n) : (energy < 1300)
```

Then the developer makes use of the tool (for example via the graphical interface), which infers an energy consumption function for the program that depends on the number of banks N , namely $E_{biquad}(N) = 165.3N + 54.45$. Moreover, the tool also infers that in order to meet the energy budget of 1,300 nJ, i.e., $165.3N + 54.45 < 1,300$, the number of banks N should be at most 7. This can be expressed for example with the following assertion, which is generated by the tool indicating that the original assertion holds subject to a precondition on the parameter N :

```
#pragma entra checked biquad(n) :
    (0 <= n && n <= 7) ==> (energy < 1300)
```

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

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

Scenario 2. Using analysis information to guide design decisions In this scenario, the user will provide an (XC) program to the analysis tool, which will statically estimate the energy consumed and time spent by its whole execution, and by each of its procedures and functions. This way, the user can identify the more energy consuming procedures / functions in the program in order to perform a better design and encoding of it, choosing better data structures, etc.

Scenario 3. Checking and verifying energy / timing specifications In this scenario, the user (application developer) interacts with the tool to prove whether resource usage (energy / timing) specifications are met or not, or to infer conditions (e.g., intervals for the input data) for which the assertion is correct or incorrect.

The user provides a source program together with energy / timing specifications to be verified by the tool. Specifications are written in the form of assertions with *status check*, which means that the verification process will be performed on such assertions. As a result of the verification process, in the output of the tool, either:

1. The assertion is included with *status checked* (resp. *false*), meaning that the assertion is correct (resp. incorrect) for all input data meeting the precondition of the assertion,
2. the assertion is “splitted” into two or three assertions with different status (*checked*, *false*, or *check*) whose preconditions include a conjunct expressing that the size of the input data belongs to the interval(s) for which the assertion is correct (*status checked*), incorrect (*status false*), or the tool is not able to determine whether the assertion is correct nor incorrect (*status check.*), or

3. in the worse case, the assertion is included with status `check`, meaning that the tool is not able to infer any verification information about it.

Consider for example a (naive reverse) program, and assume that the user has included the following assertion to be checked by the tool:

```
#pragma entra check nrev(n) :
    (length(n) <= energy && energy <= 10*length(n))
```

where `length(n)` represents the length of the input list `n`. In other words, the user wants to check whether the energy consumed by the program for an input `n` is in the interval $[\text{length}(n), 10 \times \text{length}(n)]$. The energy units are not specified, since they are not important for illustration purposes.

The outcome of the verification of the previous assertion is the following set of assertions:

```
#pragma entra checked nrev(n) :
    (1 <= length(n) && length(n) <= 16) ==>
    (length(n) <= energy && energy <= 10*length(n))
#pragma entra false nrev(n) :
    (length(n) == 0 || length(n) >= 17) ==>
    (length(n) <= energy && energy <= 10*length(n))
```

meaning that the assertion is true for values of $\text{length}(n)$ belonging to the interval $[1, 16]$, and false for values of $\text{length}(n)$ in the interval $[0, 0] \cup [17, \infty]$. In order to produce that outcome, the resource analysis infers both upper and lower bounds functions for the energy consumed by the program, then those bounds are compared against the specification. In this particular case, the upper and lower bounds inferred by the analysis are the same, namely the function $0.5 \times \text{length}(A)^2 + 1.5 \times \text{length}(A) + 1$ (which implies that this is the exact energy consumption function for the program).

As we can see in Figure 3, the energy consumption function inferred by the analysis lies in the interval expressed by the specification, namely:

$[\text{length}(n), 10 \times \text{length}(n)]$, for $\text{length}(n)$ belonging to the data size interval $[1, 16]$. Therefore, the tool says that the assertion is *checked* in that data size interval. However for $\text{length}(n) = 0$ or $\text{length}(n) \in [17, \infty]$, the assertion is *false*. This is because the energy consumption interval inferred by the analysis is disjoint with the one expressed in the specification. This is determined by the fact that the lower bound energy consumption function inferred by the analysis is greater than the upper bound energy consumption function expressed in the specification.

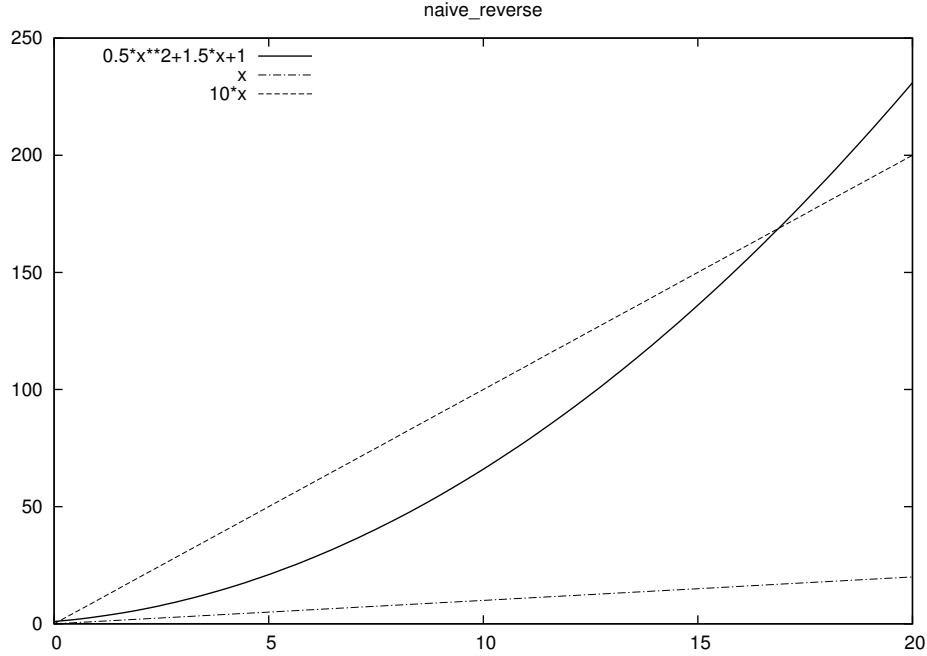


Figure 3: Energy consumption functions for a (naive reverse) program.

We consider now the case where the assertions to be checked by the tool include preconditions expressing data size intervals, i.e., the applicability of such assertions is restricted to certain intervals of input data sizes. This is a common case, since often in a system the possible input data belong to certain value ranges. Such preconditions can be useful to reduce false negative errors during static checking which may be caused by input values that actually never occur.

For example, consider the previous program, and assume now that the possible length of the input list is in interval $[1, 10]$. In this case, we can add a precondition to the specification expressing an interval for the input data size as follows:

```
#pragma entra check nrev(n) : (1 <= length(n) && length(n) <= 10)
==>      (length(n) <= energy && energy <= 10*length(n));
```

As we can see in Figure 3, this assertion is true because for any input value n such that $length(n) \in [1, 10]$, the energy consumption function of the program inferred by analysis lies in the specified energy interval $[length(n), 10 \times length(n)]$. In general, the outcome of the static checking of an assertion with a precondition expressing an interval for the input data size can be different for different subintervals of the one expressed in the precondition.

Scenario 4. Using trusted assertions for unknown (or not available) code In this scenario the code of one procedure (or in general several procedures) called in the main program is not

available or is not implemented yet. The developer wants to know the impact on energy consumption of such a procedure in the main program.

For the procedures that are not developed yet or (for which the code is not available), the user can write *trusted* assertions (in the Common Assertion Language described in deliverable D2.1 [EG13]) providing upper-bound energy (and timing) information for them. Such assertions are assumed to be true by the analysis tool, and the energy information they provide are propagated by the analysis to estimate an upper bound on the energy consumed by the whole program. If such global energy estimate is acceptable, then the user can develop an implementation of the procedure (or use an existing one) whose energy consumption is bounded by the one expressed by the trusted assertion. Otherwise, a more efficient implementation of the procedure (or other parts of the program) should be used in order to reduce the global energy consumed.

Note that *trusted* assertions can be also provided for parts of a system for which the analysis infers inaccurate information, in order to improve its accuracy.

As a future work, trusted assertions will allow to express the energy used by a procedure as a parameter, so that the global energy estimated for the main program will be given in terms of such parameter. This will allow to find values for the parameter for which a global energy budget is met.

3.1.3 Architecture of the Tool

An overview diagram of the architecture of the prototype tool is depicted in Figure 4. The diagram shows its different components (blue boxes), data (pink square boxes) and information flow (arrows).

The tool takes as input an XC source program that can (optionally) contain (front end) assertions (see deliverable D2.1 for details about the common assertion language). Such assertions are used to express energy or timing specifications that the tool will try to prove or disprove, but they can also express trusted information such as the energy usage of procedures that are not developed yet, or useful hints and information to the tool. Since the user can choose between performing the analysis at the ISA or LLVM IR levels (or both), the associated ISA and/or LLVM IR representations of the XC program are generated using the xcc compiler. Such representations include useful metadata. The **HC IR transformation** component (described in deliverable D3.2, Attachment D3.2.4), produces the internal representation used by the tool, HC IR, which includes the program (Horn Clauses) and possibly specifications and/or trusted information (expressed in the IAL). It performs several tasks:

1. Transforming the ISA and/or LLVM IR into the internal representation used by the tool, HC IR. Such transformation preserves the resource consumption semantics, in the sense

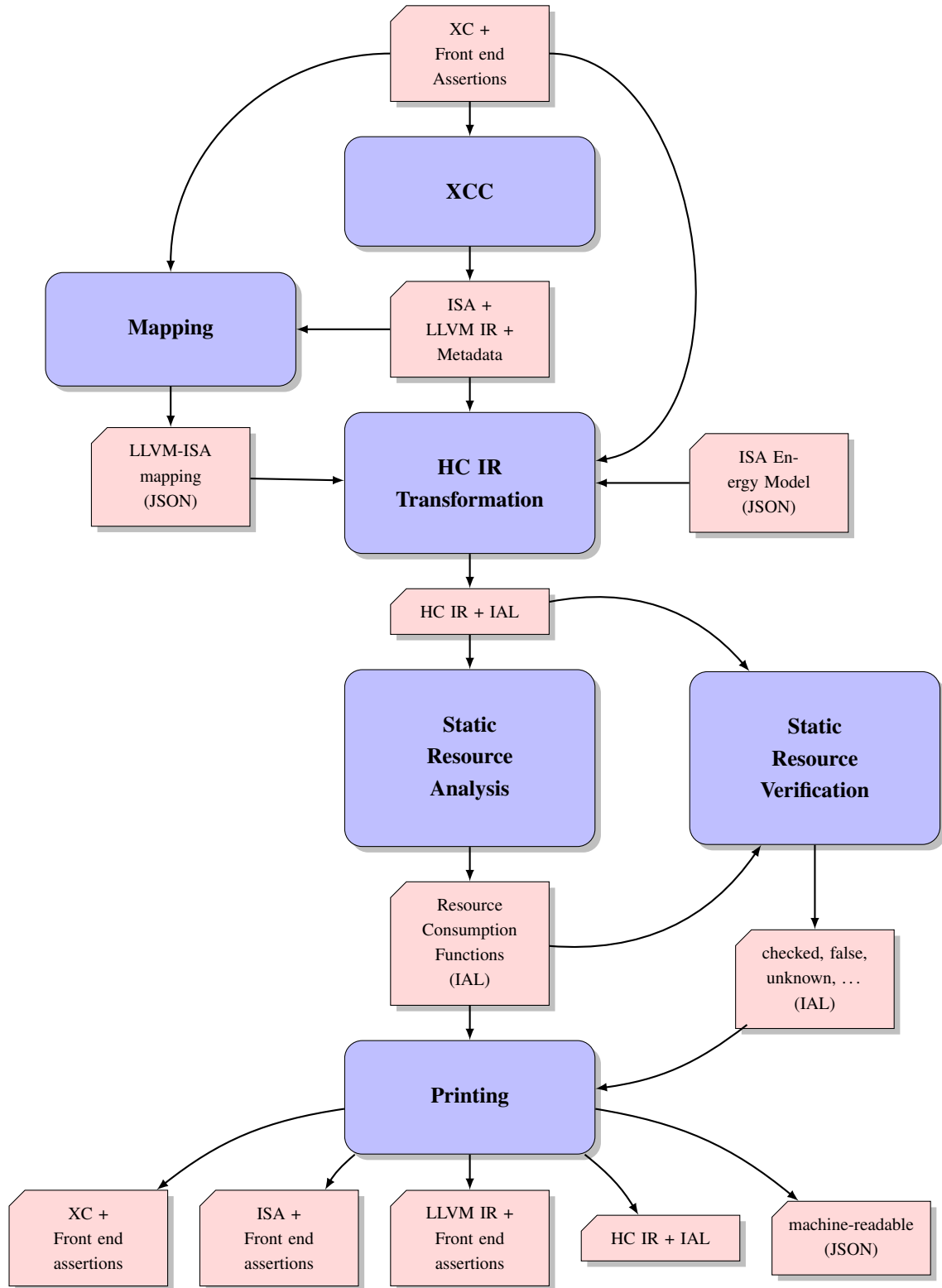


Figure 4: Energy consumption analysis & verification tool using HC IR and CiaoPP.

that the resource usage information inferred by the tool is applicable to the original XC program.

2. Transforming specifications (and trusted information) written as front end assertions into the IAL.
3. Transforming the energy model at the ISA level (described in D2.2 [EKG14]), expressed in JSON format, into the Internal Assertion Language (IAL, described in D2.1 [EG13]). Such IAL assertions express the energy consumed by individual ISA instruction representations.
4. In the case the analysis is performed at the LLVM IR level, the **HC IR transformation** component produces a set of IAL assertions expressing the energy consumption corresponding to LLVM IR block representations in HC IR. Such information is produced from the mapping of LLVM IR instructions with sequences of ISA instructions produced by the **mapping component** and the ISA level energy model.

Then, the parametric static resource usage analyzer (developed in WP3 and described in D3.1, D3.2, Attachment D3.2.4, and [LKS⁺13]) takes the HC IR, together with the assertions which express the energy consumed by LLVM IR blocks and/or individual ISA instructions, and possibly some additional (trusted) information, and processes them, producing the analysis results, which are expressed also using IAL assertions. Such results include resource usage functions (which depend on input data sizes) for each block in the HC IR (i.e., for the whole program and for all the procedures and functions in it.). The procedural interpretation of the HC IR programs, coupled with the resource-related information contained in the (IAL) assertions, together allow the resource analysis to infer static bounds on the energy consumption of the HC IR programs that are applicable to the original LLVM IR and, hence, to their corresponding XC programs.

In case the user wants the system to check energy or timing specifications provided to the tool (by including them in the XC source), the resource usage verification is done by a specialized component which compares the energy / timing specifications with the (safe) approximated information inferred by the static resource analysis.

Finally, the results produced by the static analysis and verification components are then processed by the **printing** component, which is in charge of showing the information to the program developer, referred to the original XC source program, in a user-friendly format. It uses mapping information for this purpose.

3.1.4 Evaluation and Experimental Results

We have performed an experimental evaluation of our prototype tool on a number of selected benchmarks. Power measurement data were collected for the XCore platform by using appropriately instrumented power supplies, a power-sense chip, and an embedded system for controlling the measurements and collecting the power data. The main goal of our experiments was to shed light on the trade-offs implied by performing the analysis at the ISA level (without using complex mechanisms for propagating type information and representing memory) and at the LLVM level using models defined at the ISA level together with a mapping mechanism.

There are two groups of benchmarks that we have used in our experimental study. The first group is composed by four small recursive numerical programs that have a variety of user defined functions, arguments, and calling patterns. These benchmarks only operate over primitive data types and do not involve any structured types. Both the ISA and LLVM IR analyses are able to infer energy functions for them.

- `fact (N)` : Calculates the factorial of the integer `N`.
- `fibonacci (N)` : Calculates the `N`th Fibonacci number.
- `sqr (N)` : Computes N^2 using just additions.
- `power_of_two (N)` : Calculates 2^N using no multiplication.

The second group of benchmarks differs from the first group in the sense that they all involve structured types. The programs are recursive or iterative.

- `reverse (N, M)` : Iterative program that reverses an array.
- `concat (N, M)` : Recursive program that concatenates two arrays.
- `mat_mult (N, M)` : Iterative program that performs matrix multiplication.
- `sum_facts (N, M)` : Recursive program that computes the sum of the factorials of the numbers in an array.
- `fir (N)` : Iterative Finite Impulse Response (FIR) filter. The `fir (N)` benchmark computes the inner-product of two vectors: a vector of input samples, and a vector of coefficients. The more the coefficients, the higher the fidelity, and the lower the frequencies that can be filtered. The cost depends on the number of taps `N` (size of the coefficients).

- `biquad(N)`: This benchmark is an equaliser. An equaliser takes a signal and attenuates / amplifies different frequency bands. This will, in the case of an audio signal, correct for a speaker or microphone its frequency response. This equaliser benchmark uses a cascade of Biquad filters where each filter attenuates or amplifies one specific frequency range. The energy consumed depends on the number of banks N , typically between 3 and 30 for an audio equaliser. A higher number of banks enables a designer to create more precise frequency response curves.

The ISA level analysis used in our experiments is not able to infer useful energy functions for the second group of benchmarks. This is due to the fact that significant program and data type / shape information is lost due to lower-level representations, which sometimes makes the analysis at the ISA level difficult or impossible. In order to overcome this limitation and improve analysis accuracy, a significantly more complex representation of memory in the HC IR would be needed.

Table 1 shows detailed results of our experiments. Column **SA energy function** shows the energy consumption functions, which depend on input data sizes, inferred for each program by the static analyses performed at the ISA and LLVM IR levels (denoted with subscripts *isa* and *llvm* respectively). The argument M in the $sum_facts_{llvm}(N, M)$ function refers to the estimated size of an element of the array (which controls the inner loop / recursion). We can see that the analysis is able to infer different kinds of functions (polynomial, exponential, etc.). Column **HW** shows the actual energy consumption in nano Joules (nJ) measured on the hardware corresponding to the execution of the programs with input data of different sizes (shown in column **Input Data Size**). Both the measurements and inferred estimations are without using any optimization (using -O0) at compile time. **Estimated** presents the energy consumption estimated by static analysis. This is obtained by evaluating the functions in column **SA energy function** for the input data sizes in column **Input Data Size**. The value N/A in such column means that the analysis has not been able to infer any energy consumption function and, thus, no estimated value can be obtained. Column **Error vs. HW** shows the error of the values estimated by the static analysis with respect to the actual energy consumption measured on the hardware calculated as follows: **Error vs. HW** = $(\frac{LLVM(or\ ISA)-HW}{HW} \times 100)\%$ Finally, the last column shows the ratio between the estimations of the analysis at the ISA and LLVM IR levels.

Table 2 shows a summary of results. The first two columns show the name and short description of the benchmarks. The columns under **Error vs. HW** show the average error of the energy consumption estimated by the static analysis (performed at both the ISA and LLVM IR levels) compared to the actual energy consumption measured on the hardware. The average has been obtained by using different values for the input data, by evaluating the energy functions

SA energy function (nJ)	Input Data Size	HW (nJ)	Estimated (nJ)		Error vs. HW %		isa/ llvm
			llvm	isa	llvm	isa	
$Fact_{isa}(N) = 26.0N + 19.4$ $Fact_{llvm}(N) = 28.4N + 22.4$	N=2	78	75	70	-3.40	-9.63	0.94
	N=4	128	129	121	1.34	-4.79	0.94
	N=8	227	237	223	4.59	-1.48	0.94
	N=16	426	453	428	6.52	0.49	0.94
	N=32	824	886	836	7.58	1.57	0.94
	N=64	1690	1751	1654	3.59	-2.15	0.94
$Fibonacci_{isa}(N) = 31.62 + 36.6 \times 1.62^N + 11.4 \times (-0.62)^N$ $Fibonacci_{llvm}(N) = 37.53 + 42.3 \times 1.62^N + 11.68 \times (-0.62)^N$	N=2	75	74	69	-1.16	-7.88	0.93
	N=4	219	241	221	10	0.93	0.92
	N=8	1615	1951	1693	14.75	4.81	0.91
	N=15	47×10^3	57×10^3	50×10^3	16.47	6.33	0.91
	N=26	9.30×10^6	11.5×10^6	9.9×10^6	17.31	7.09	0.91
$Sqr_{isa}(N) = 9.02N^2 + 51.29N + 16.5$ $Sqr_{llvm}(N) = 10.52N^2 + 55.79N + 16.5$	N=9	1242	1302	1209	4.86	-2.66	0.93
	N=27	8135	8734	7979	7.36	-1.92	0.91
	N=73	52×10^3	57×10^3	52×10^3	8.51	-1.58	0.91
	N=144	19.7×10^4	21×10^4	19.4×10^4	8.89	-1.47	0.90
	N=234	51×10^4	55×10^4	50×10^4	9.61	-0.91	0.90
	N=360	11.89×10^5	13×10^5	11.9×10^5	10.49	-0.17	0.90
$Poweroftwo_{isa}(N) = 10.9 \times 2^{N+2} - 27.29$ $Poweroftwo_{llvm}(N) = 49.2 \times 2^N - 31.5$	N=3	326	344	322	5.68	-1.10	0.94
	N=6	2729	2965	2770	6.59	1.49	0.93
	N=9	21.9×10^3	23.9×10^3	22.3×10^3	6.61	1.81	0.93
	N=12	17.57×10^4	19.1×10^4	17.9×10^4	6.62	1.85	0.93
	N=15	13.8×10^5	15.3×10^5	14.3×10^5	6.62	3.71	0.93
$reverse_{llvm}(N) = 20.50N + 72.98$	N=57	1138	1179	N/A	3.60	N/A	N/A
	N=160	3125	3185	N/A	1.91	N/A	N/A
	N=320	6189	6301	N/A	1.82	N/A	N/A
	N=720	13848	14092	N/A	1.76	N/A	N/A
	N=1280	24634	24998	N/A	1.48	N/A	N/A
$matmult_{llvm}(N) = 44.71N^3 + 72.47N^2 + 52.52N + 25.49$	N=10	49.77×10^3	49.9×10^3	N/A	0.22	N/A	N/A
	N=15	15.79×10^4	15.9×10^4	N/A	1.03	N/A	N/A
	N=20	36.29×10^4	36.8×10^4	N/A	1.51	N/A	N/A
	N=25	69.56×10^4	70.8×10^4	N/A	1.77	N/A	N/A
	N=31	13.07×10^5	13.8×10^5	N/A	1.98	N/A	N/A
$concat_{llvm}(N, M) = 69.14N + 69.14M + 14.12$	N=50; M=154	14.8×10^3	13.5×10^3	N/A	8.67	N/A	N/A
	N=131; M=69	14.5×10^3	13.2×10^3	N/A	8.65	N/A	N/A
	N=170; M=182	25.44×10^3	23.3×10^3	N/A	8.60	N/A	N/A
	N=188; M=2	13.8×10^3	12.6×10^3	N/A	8.59	N/A	N/A
	N=13; M=134	10.7×10^3	9.79×10^3	N/A	8.74	N/A	N/A
$sum-facts_{llvm}(N, M) = 28.45N \times M + 76.71N + 22.50$	N=15; M=7.6	4097	4196	N/A	2.40	N/A	N/A
	N=40; M=7.4	10.7×10^3	11×10^3	N/A	2.45	N/A	N/A
	N=80; M=8	22.7×10^3	23.3×10^3	N/A	2.52	N/A	N/A
	N=160; M=7.8	44.3×10^3	45.4×10^3	N/A	2.45	N/A	N/A
$biquad_{llvm}(N) = 165.3N + 54.45$	N=5	871	880	N/A	1.04	N/A	N/A
	N=7	1187	1211	N/A	2.05	N/A	N/A
	N=10	1660	1707	N/A	2.83	N/A	N/A
	N=14	2290	2368	N/A	3.42	N/A	N/A
$fir_{llvm}(N) = 33.47N + 141.6$	N=85	2999	2984	N/A	0.48	N/A	N/A
	N=97	3404	3386	N/A	0.53	N/A	N/A
	N=109	3812	3788	N/A	0.63	N/A	N/A
	N=121	4227	4189	N/A	0.88	N/A	N/A

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

inferred by the analysis and comparing the results with the actual energy of the execution of the programs for such input data. The last row of the table shows the average error over the number of benchmarks analyzed at each level.

Program	Description	Error vs. HW		ISA/LLVM
		llvm	isa	
fact (N)	Calculates $N!$	4.5%	2.86%	0.94
fibonacci (N)	Nth Fibonacci no.	11.94%	5.41%	0.92
sqr (N)	Computes N^2	9.31%	1.49%	0.91
power_of_two (N)	Calculates 2^N	11.15%	4.26%	0.93
reverse (N, M)	Reverses an array	2.18%	N/A	N/A
concat (N, M)	Concatenation of arrays	8.71%	N/A	N/A
mat_mult (N, M)	Matrix multiplication	1.47%	N/A	N/A
sum_facts (N, M)	Sum of factorials in an array	2.42%	N/A	N/A
fir (N)	Finite Impulse Response filter	0.63%	N/A	N/A
biquad (N)	biquad filter	2.34%	N/A	N/A
Average		5.48%	3.50%	0.92

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

The experimental results show that:

- On average, the analysis performed at either level is reasonably accurate and the relative error between the two analysis at different levels is small.
- ISA-level estimations are slightly more accurate than the ones at the LLVM IR level (3.5% vs. 5.48% error on average with respect to the actual energy consumption measured on the hardware respectively). This is because the ISA-level analysis uses very accurate energy models, obtained from measuring directly at the ISA level, whereas at the LLVM IR level, such ISA-level model needs to be propagated up to the LLVM IR level using (approximated) mapping information. This causes a slight loss of accuracy.
- The LLVM IR level analysis is more powerful than the one at the ISA level. This is because type information is preserved at the LLVM IR level, which allows analyzing programs using data structures (such as arrays) that could not be analyzed at the ISA level, without a significantly more complex representation of memory in the Horn clause representation.

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



```
#include "fact.h"

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

Figure 5: XC source of a factorial program.

program information needed by the static analysis, and 2) the LLVM IR is close enough to the ISA level to allow the propagation of the ISA energy model up to the LLVM IR level without significant loss of accuracy for the examples studied. Our experiments are based on single-threaded programs. It remains to be seen whether the results would carry over to other classes of programs, such as multi-threaded programs and programs where timing is more important. In this sense our results are preliminary, yet are promising enough to continue research in analysis of LLVM IR and ISA-LLVM IR energy mapping techniques for a wider class of programs, especially multi-threaded programs.

LLVM IR is in partial Static Single Assignment (SSA) form (up to the variable names only), which makes it difficult to model aliases and indirect memory operations in SSA form in HC IR, particularly for derived types (arrays, structures etc.) as well as pointer arithmetic. A possible solution is to use hashed or array SSA representation of LLVM IR that models aliases and indirect memory operations in SSA for derived types.

3.1.5 Demonstration of the Tool

This section provides a demonstration of the use of the implemented prototype in two typical scenarios: Analyzing the energy consumed by an XC program and verifying energy related specifications.

Analyzing the energy consumed by an XC program In order to analyze an XC program using the CiaoPP graphical interface, we first open it in a buffer, as shown in Figure 5. Then we select the menu options depicted in Figure 2: `analyze`, for `Action Group`, `res_plai`, for `Resource Analysis`, `isa`, for `Select Analysis Level` (which will tell the analysis


```

fact_unfold_entry_res_plai_co.xc
#include "fact.h"

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

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

fact_unfold_entry_res_plai_co.xc  All L11 (C/1 +3 Abbrev)

```

Figure 6: Analysis results (expressed as front end assertions).

to take the ISA option by compiling the source code into ISA and transform into HC IR for analysis) and finally source, for Select Output Language (the language in which the analysis results are shown). After clicking on the Apply button below the menu options, the analysis is performed, producing the results as depicted in Figure 6 (marked with a red arrow). Such results are expressed in the front end aspect of the common assertion language, as explained in Deliverables D2.1 [EG13] and D3.1 [LG13]. We can see that the energy consumption of the factorial program is given as a linear function on the size of the input argument to the program, A , namely $21469718 * A + 16420396 \text{ nJ}$.

Verification of the energy budget for an XC program In order to verify that an energy budget can be met by a given XC program and to find the optimal values for the inputs for which such budget is respected, we first open the program in a using the CiaoPP graphical interface buffer, as shown in Figure 7. Line 5 in the program specifies the energy budget (to be less than equal to 122009721 nj) with a *check* assertion. Then we select the following menu options: *check_assertions*, for Action Group, *res_plai*, for Resource Analysis, *llvm*, for Select Analysis Level (which will tell the analysis to take the LLVM IR route by compiling the source code into LLVM IR and transform into HC IR for analysis) and finally *source*, for Select Output Language (the language in which the analysis results are shown). After clicking on the Apply button below the menu options, the analysis is performed, producing the results as depicted in Figure 8 (on lines 5 and 7). Such results are expressed in the front end aspect of the common assertion language, as explained in Deliverables D2.1 [EG13] and D3.1 [LG13]. We can see that on line 9 the energy consumption of the biquad program is given as a linear function on the size of the input argument to the program, C , namely $16652087 * C + 5445103 \text{ nJ}$. On line 5 the assertion with status *false* indicate that energy budget

```

1 #include "main.h"
2 #include "biquad.h"
3 #include <xs1.h>
4
5 #pragma entra check biquadCascade(n1, n2, n3) : (1 <= n3) ==> (energy <= 122009721)
6
7 #pragma unsafe arrays
8 int biquadCascade(biquadState &state, int xn, int BANKS1) {
9     unsigned int ynl;
10    int ynh;
11
12    for(int k=BANKS1; k>0; k--)
13    {
14        int j = BANKS1-k;
15        ynl = (1<<(FRACTIONALBITS-1));
16        ynh = 0;
17        {ynh, ynl} = macs( biquads[j].b0, xn, ynh, ynl);
18        {ynh, ynl} = macs( biquads[j].b1, state.b[j].xn1, ynh, ynl);
19        {ynh, ynl} = macs( biquads[j].b2, state.b[j].xn2, ynh, ynl);
20        {ynh, ynl} = macs( biquads[j].a1, state.b[j+1].xn1, ynh, ynl);
21        {ynh, ynl} = macs( biquads[j].a2, state.b[j+1].xn2, ynh, ynl);
22        if (sext(ynh, FRACTIONALBITS) == ynh) {
23            ynh = (ynh << (32-FRACTIONALBITS)) | (ynl >> FRACTIONALBITS);
24        } else if (ynh < 0) {
25            ynh = 0x80000000;
26        } else {
27            ynh = 0x7fffffff;
28        }
29        state.b[j].xn2 = state.b[j].xn1;
30        state.b[j].xn1 = xn;
31
32        xn = ynh;
33    }
34    state.b[BANKS1].xn2 = state.b[BANKS1].xn1;
35    state.b[BANKS1].xn1 = ynh;
36    return xn;
37 }

```

--- biquad.xc All L26 Git:new-resources (C/l Abbrev)

Figure 7: XC source of a biquad program.

(specified with a *check* assertion on line 11) is not met for $C \geq 8$. Similarly, on line 7 the assertion with the status *checked* indicate the interval of the argument C ($1 \leq C \leq 7$) for which the budget is met.

```

1 #include "main.h"
2 #include "biquad.h"
3 #include <xs1.h>
4
5 #pragma entra false biquadCascade(A,B,C) : (8 <= C) ==> (energy <= 122009721)
6
7 #pragma entra checked biquadCascade(A,B,C) : (1 <= C && C <= 7) ==> (energy <= 122009721)
8
9 #pragma entra true biquadCascade(A,B,C) : (16502087*C+5445103 <= energy && energy <= 16652087*C+5445103)
10
11 #pragma entra check biquadCascade(n1, n2, n3) : (1 <= n3) ==> (energy <= 122009721)
12 #pragma unsafe arrays
13 int biquadCascade(biquadState &state, int xn, int BANKS1) {
14     unsigned int ynl;
15     int ynh;
16
17     for(int k=BANKS1; k>0; k--)
18     {
19         int j = BANKS1-k;
20         ynl = (1<<(FRACTIONALBITS-1));
21         ynh = 0;
22         {ynh, ynl} = macs( biquads[j].b0, xn, ynh, ynl);
23         {ynh, ynl} = macs( biquads[j].b1, state.b[j].xn1, ynh, ynl);
24         {ynh, ynl} = macs( biquads[j].b2, state.b[j].xn2, ynh, ynl);
25         {ynh, ynl} = macs( biquads[j].a1, state.b[j+1].xn1, ynh, ynl);
26         {ynh, ynl} = macs( biquads[j].a2, state.b[j+1].xn2, ynh, ynl);
27         if (sext(ynh,FRACTIONALBITS) == ynh) {
28             ynh = (ynh << (32-FRACTIONALBITS)) | (ynl >> FRACTIONALBITS);
29         } else if (ynh < 0) {
30             ynh = 0x80000000;
31         } else {
32             ynh = 0x7fffffff;
33         }
34         state.b[j].xn2 = state.b[j].xn1;
35         state.b[j].xn1 = xn;
36
37         xn = ynh;
38     }
39     state.b[BANKS1].xn2 = state.b[BANKS1].xn1;
40     state.b[BANKS1].xn1 = ynh;
41     return xn;
42 }

```

~:*** biquad_unfold_entry_res_plai_co.xc All L16 (C/l Abbrev)

Figure 8: Verification results (expressed as front end assertions).

3.2 Multi-level Mapper Tool

The ENTRA mapper tool serves two purposes. Firstly, to create a mapping of code locations between different intermediate code representations, ISA and source code. This allows analysis results to be mapped back to the procedures and functions in the source code. Secondly, the mapper tool allows energy model information defined at one level to be propagated upwards to other levels. Analysis that can only be performed at these higher levels can then utilise this raised energy model information.

The mapper tool can be seen as the glue that holds together several pieces of our framework for resource consumption analysis and verification, enabling energy transparency. For more details about the techniques utilized to create this tool, see Section 5 from deliverable D3.1.

The mapper tool has been developed using the Java language, for two primary reasons. Firstly, Java is portable to any operating system, including Windows, Linux and Mac OS. Secondly, this makes potential integration into the XMOS tools environment easier, because the tools environment is built around the Eclipse software development suite, which is itself predominantly Java based.

Currently the prototype tool is accessible through a Linux shell script as a normal command line tool. Figure 9 shows the contents of the `ReadMe.txt` file included in the prototype package. This describes the tool version, the contents of the tool folder, the dependencies and the expected output of the tool.

Figure 10 shows the output of executing `mapper --help` on a Linux terminal after the tool is installed. The output describes the usage of the tool and all the available options. In the following subsections we will give a brief description of each option and their results using some screenshots on real examples.

3.2.1 Using the `--analysis` Option

This option performs the analysis of the LLVM IR and the corresponding ISA code, creating a Control Flow Graph (CFG) at both code levels. It then creates a mapping between the two levels, and then using an ISA level energy model, provides energy costs for each LLVM IR instruction and block. This mapping information is saved in a JavaScript Object Notation (JSON) file, a format which is both human readable and widely supported in many languages, making it straightforward for other analysis tools to parse it.

Using a real example from our benchmarks, the `mac` function which can be found in Code Sample 1, the following command will analyze the code to create the CFGs and the mapping info:

```

XC_ISA - LLVMIR mapper (Linux Version)
=====

To use:

1. Download Stable Released XMOS Development Tools, Version: 12.2.0 (XtimeComposer)
2. Extract the folder "entra-utils" into xmos tools folder
3. Make sure you have installed packages::

    1. java version "1.7.0_25"
    2. dwarfdump
    3. graphviz

4. Make sure you give executable rights to user on files::

    1. mapper
    2. map.jar
    3. xcc1llvm
    4. opt
    5. llc

5. Run the mapper using the command::

    ./mapper [options] <path_to_input.xe>

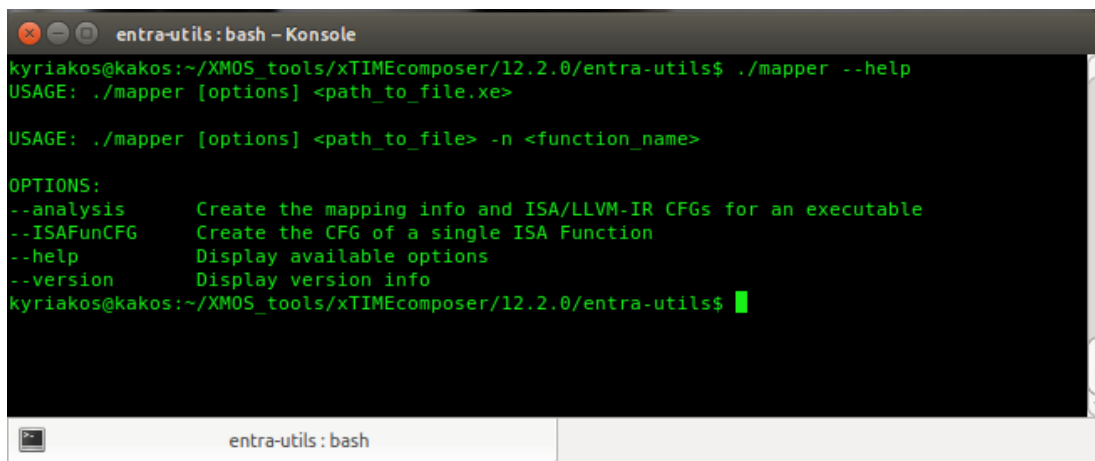
6. Program output::

    1. input.json - the map between LLVMIR and XC_ISA instructions in a json format
    2. input_ISA.ps - a CFG of the ISA code
    3. input_LLVMIR.ps - a CFG of the LLVM IR code
    4. Both graphs have the blocks related instructions and "| custom debug line"
       attached at the end of each instruction

7. For more help on the tool usage run "./mapper --help"

```

Figure 9: The Mapper prototype tool ReadMe.txt installation guide.



```

entra-utils: bash - Konsole
kyriakos@kakos:~/XMOS_tools/xTIMEcomposer/12.2.0/entra-utils$ ./mapper --help
USAGE: ./mapper [options] <path_to_file.xe>

USAGE: ./mapper [options] <path_to_file> -n <function_name>

OPTIONS:
--analysis      Create the mapping info and ISA/LLVM-IR CFGs for an executable
--ISAFuncCFG    Create the CFG of a single ISA Function
--help          Display available options
--version       Display version info
kyriakos@kakos:~/XMOS_tools/xTIMEcomposer/12.2.0/entra-utils$

```

Figure 10: The mapper prototype tool output of the --help command option.

```
$ mapper --analysis benchmarks/mac/mac.xe
```

In this case, `mac.xe` is the executable created ahead by using the XMOS compiler. Figure 11 shows the created CFG for the ISA code. The CFG shows the control flow between each basic block (BB), and the instructions in each block. Each instruction has an `id` number at the end of the line, in square brackets. Every ISA instruction with a given `id` number is derived from an LLVM IR instruction with the same `id` during code lowering. Combined with the CFG in Figure 12, code locations in the ISA CFG can be mapped to locations in LLVM IR using the `id` numbers, and vice versa.

Code Sample 1: The `mac` benchmark source code

```
int macCustom(const short a[], const short b[], short len, int sqr, int &sum)
{
    int i;
    int dotp = sum;

    while (len--) {
        dotp += b[len] * a[len];
        sqr += b[len] * b[len];
    }
    sum = dotp;
    return sqr;
}
```

In addition to providing code location mappings, the `--analysis` option also infers the existence of “fetch, no-operation” (FNOP) instructions, such as in Figure 11, `id 24`.

These are implicit instructions, introduced by the scheduler logic under certain conditions at runtime. An FNOP is needed when the instruction buffer for a thread does not contain the next instruction. This can happen in two cases [May13]. In the first case, due to the fact that the last stage of the pipeline is shared between memory operations and the fetch operation, when multiple memory instructions happen in a row the buffer might be left empty and then an FNOP will be introduced in order to fetch new instructions. In the second case, a branch instruction is executed and flushes the instruction buffer, which may require an FNOP if the branch target is unaligned and the target instruction is long (4 bytes).

The FNOP has no effect on the actual result of the program, but has an affect on the time and energy. To account for this, the mapper tool performs a static analysis on the ISA CFGs to recreate the behaviour of the instruction buffer logic and reason where the FNOPs will be introduced.

The mapper tool output in Figure 12 also identifies several pieces of energy consumption

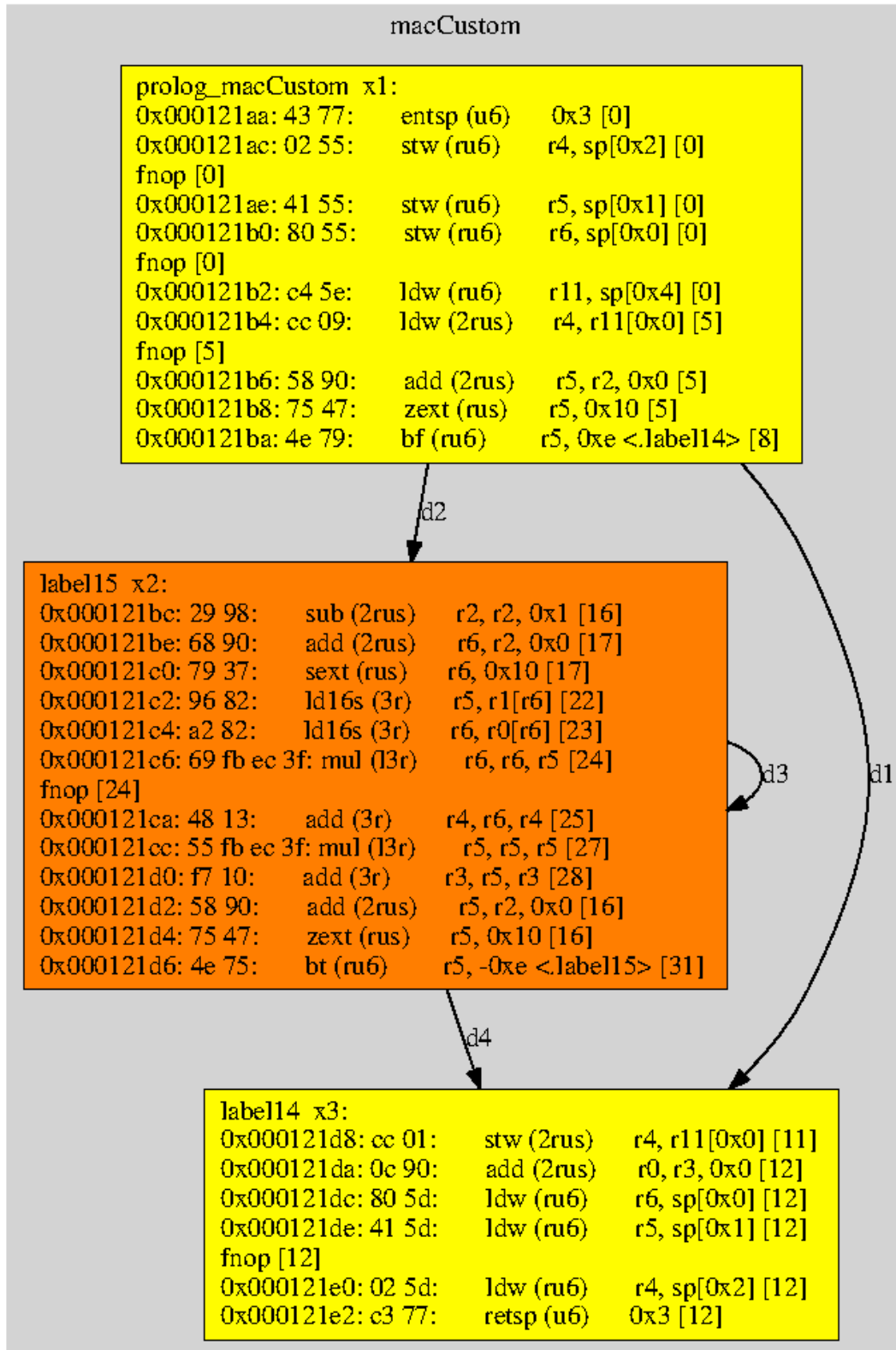


Figure 11: The CFG of the ISA code for the mac benchmark.

data to help developers understand the energy consumption of their code. In each BB the LLVM IR instruction which costs the most energy is identified, and prefixed with +SOS+. This could potentially lead to some energy aware instruction selection. Furthermore, at the end of each BB the total energy cost, whether the block is recursively called, and the loop depth of the block are provided. This information is significant to a programmer because a deeply nested or recursive block will consume more energy than others, and may deliver the greatest energy savings in the program if optimized. Finally, BBs in loops are denoted with orange color in both ISA and LLVM IR CFGs.

All of the information included in the CFGs is also written to a JSON file to be used by other tools. The JSON file for our example is shown in Code Sample 2. The nested structure of this file is as follows. Each function in the program contains a list of BBs. Each BB contains an list of LLVM IR instructions, and each such instruction contains a list of ISA instructions that it maps to. Each BB also has the total energy cost in pico Joules (pJ). Finally for each BB the `start` and `end` fields record the line numbers in the source code that it was compiled from. This is particularly useful for linking analysis results back to the source code.

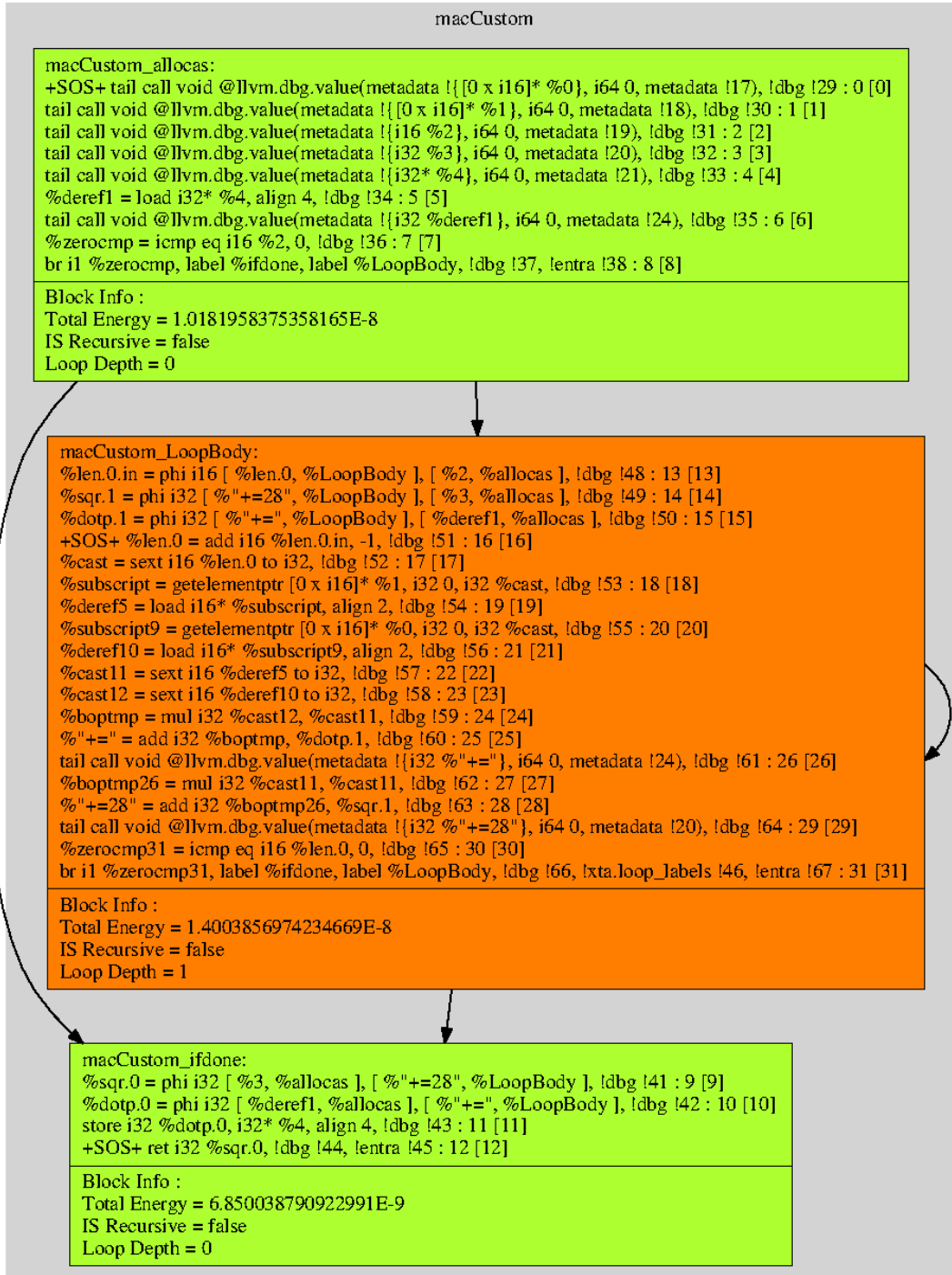


Figure 12: The CFG of the LLVM IR code for the `mac` benchmark.

Code Sample 2: The *json* file emitted for the `mac.xc` with the mapping info

```

1 {"program": {
2   "functions": [{
3     "lineEnds": " ",
4     "name": "macCustom",
5     "blocks": [
6       {
7         "lineEnds": 14,
8         "name": "macCustom_alloca",
9         "mapping": [
10          {
11            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata ![[0 x i16
12              ]* %0]",
13            "ISAMap": [
14              "entsp_u6",
15              "stwsp_ru6",
16              "stwsp_ru6",
17              "stwsp_ru6",
18              "ldwsp_ru6"
19            ]
20          },
21          {
22            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata ![[0 x i16
23              ]* %1]",
24            "ISAMap": []
25          },
26          {
27            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata ![[i16 %2]"
28              ,
29            "ISAMap": []
30          },
31          {
32            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata ![[i32 %3]"
33              ,
34            "ISAMap": []
35          },
36          {
37            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata ![[i32* %4]"
38              ,
39            "ISAMap": []
40          },
41          {
42            "LLVMIRIns": "%deref1 = load i32* %4",
43            "ISAMap": []
44          }
45        ]
46      }
47    ]
48  }
49 }

```

```

38         "ISAMap": [
39             "ldw_2rus",
40             "add_2rus",
41             "zext_rus"
42         ]
43     },
44     {
45         "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{i32 %
46             deref1})",
47         "ISAMap": []
48     },
49     {
50         "LLVMIRIns": "%zerocmp = icmp eq i16 %2",
51         "ISAMap": []
52     },
53     {
54         "LLVMIRIns": "br i1 %zerocmp",
55         "ISAMap": ["brff_ru6"]
56     }
57 ],
58 "lineStarts": 10,
59 "energy": 1.0181958375358165E-8
60 },
61 {
62     "lineEnds": 19,
63     "name": "macCustom_ifdone",
64     "mapping": [
65         {
66             "LLVMIRIns": "%sqr.0 = phi i32 [ %3",
67             "ISAMap": []
68         },
69         {
70             "LLVMIRIns": "%dotp.0 = phi i32 [ %deref1",
71             "ISAMap": []
72         },
73         {
74             "LLVMIRIns": "store i32 %dotp.0",
75             "ISAMap": ["stw_2rus"]
76         },
77         {
78             "LLVMIRIns": "ret i32 %sqr.0",
79             "ISAMap": [

```

```

80         "ldwsp_ru6",
81         "ldwsp_ru6",
82         "ldwsp_ru6",
83         "retsp_u6"
84     ]
85 }
86 ],
87 "lineStarts": 19,
88 "energy": 6.850038790922991E-9
89 },
90 {
91     "lineEnds": 14,
92     "name": "macCustom_LoopBody",
93     "mapping": [
94         {
95             "LLVMIRIns": "%len.0.in = phi i16 [ %len.0",
96             "ISAMap": []
97         },
98         {
99             "LLVMIRIns": "%sqr.1 = phi i32 [ %\ "+=28\"",
100             "ISAMap": []
101         },
102         {
103             "LLVMIRIns": "%dotp.1 = phi i32 [ %\ "+=\\"",
104             "ISAMap": []
105         },
106         {
107             "LLVMIRIns": "%len.0 = add i16 %len.0.in",
108             "ISAMap": [
109                 "sub_2rus",
110                 "add_2rus",
111                 "zext_rus"
112             ]
113         },
114         {
115             "LLVMIRIns": "%cast = sext i16 %len.0 to i32",
116             "ISAMap": [
117                 "add_2rus",
118                 "sext_rus"
119             ]
120         },
121         {
122             "LLVMIRIns": "%subscript = getelementptr [0 x i16]* %1",

```

```

123     "ISAMap": []
124 },
125 {
126     "LLVMIRIns": "%deref5 = load i16* %subscript",
127     "ISAMap": []
128 },
129 {
130     "LLVMIRIns": "%subscript9 = getelementptr [0 x i16]* %0",
131     "ISAMap": []
132 },
133 {
134     "LLVMIRIns": "%deref10 = load i16* %subscript9",
135     "ISAMap": []
136 },
137 {
138     "LLVMIRIns": "%cast11 = sext i16 %deref5 to i32",
139     "ISAMap": ["ld16s_3r"]
140 },
141 {
142     "LLVMIRIns": "%cast12 = sext i16 %deref10 to i32",
143     "ISAMap": ["ld16s_3r"]
144 },
145 {
146     "LLVMIRIns": "%boptmp = mul i32 %cast12",
147     "ISAMap": ["mul_l3r"]
148 },
149 {
150     "LLVMIRIns": "%\ "+=\ " = add i32 %boptmp",
151     "ISAMap": ["add_3r"]
152 },
153 {
154     "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{i32
        %\ "+=\ "},
155     "ISAMap": []
156 },
157 {
158     "LLVMIRIns": "%boptmp26 = mul i32 %cast11",
159     "ISAMap": ["mul_l3r"]
160 },
161 {
162     "LLVMIRIns": "%\ "+=28\ " = add i32 %boptmp26",
163     "ISAMap": ["add_3r"]
164 },

```

```

165     {
166         "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{i32
           %\"+=28\"},
167         "ISAMap": []
168     },
169     {
170         "LLVMIRIns": "%zerocmp31 = icmp eq i16 %len.0",
171         "ISAMap": []
172     },
173     {
174         "LLVMIRIns": "br i1 %zerocmp31",
175         "ISAMap": ["brbt_ru6"]
176     }
177 ],
178 "lineStarts": 14,
179 "energy": 1.4003856974234669E-8
180 }
181 ]
182 }],
183 "fileLocation": "Benchmarks/mac/mac.xc",
184 "name": "mac.xc"
185 }}

```

3.2.2 Using the `--ISAFunCFG` Option

Some of the functions in the ISA code for a program might not correspond to any source code. This is due to the compiler invoking internal functions to perform necessary operations. In this case it may be necessary to analyze these functions in order to reason about their contribution on the time and the energy consumption of a program. Figure 13 is an example of this. The `memset` function is introduced in the ISA code and called by user defined functions in order to initialize some local variables. To create the CFG of the `memset` function the following mapper command is invoked:

```
$ mapper --ISAFunCFG bencharks/mac/ -n memset
```

3.3 A Tool for Flow and Synchronization Analysis of Multi-threaded XC Programs

We outline a tool that is under construction for analysing source code XC programs, whose goal is to provide energy-relevant information to the developer such as:

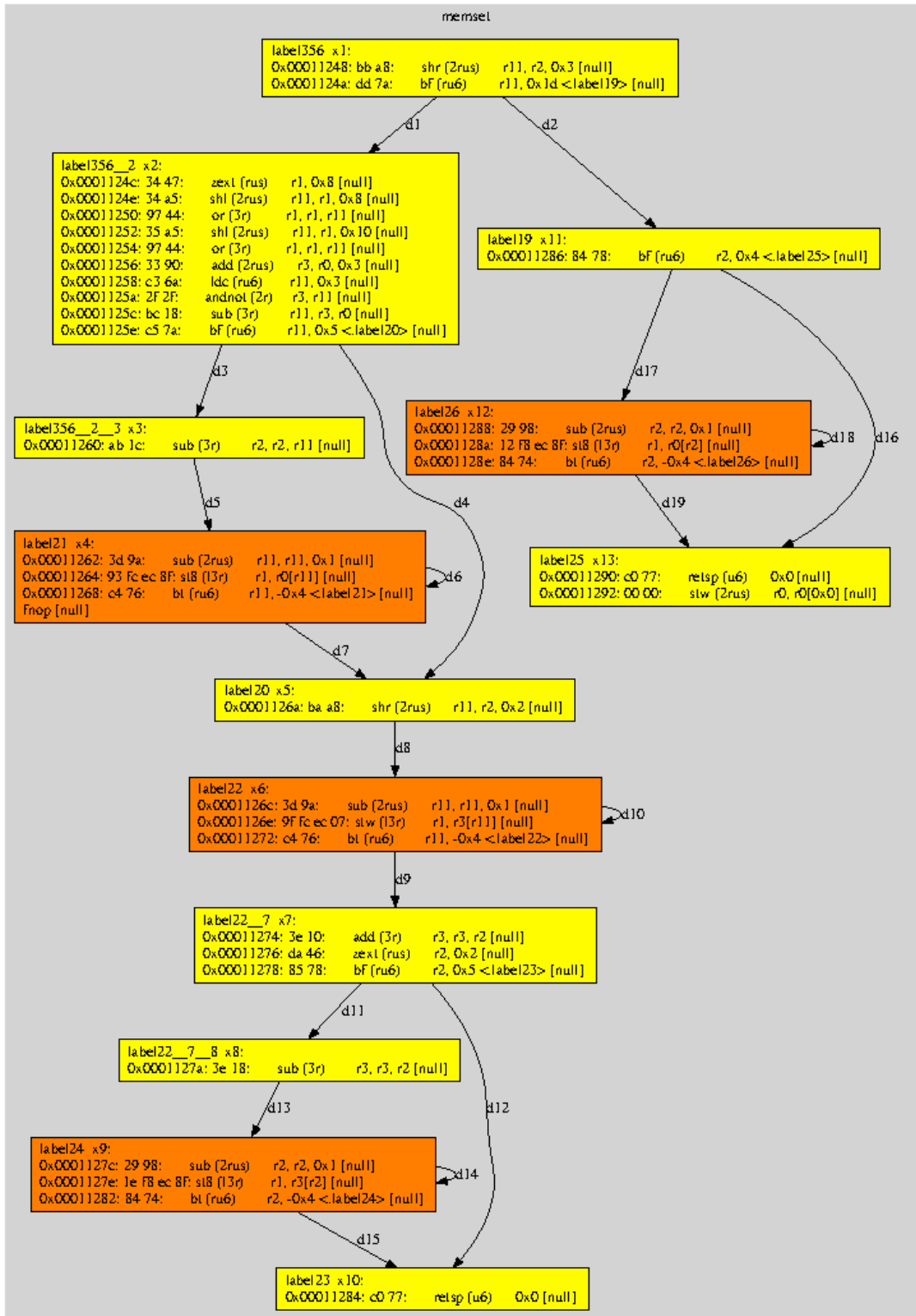


Figure 13: The CFG of the ISA code for the `memset` function.

- What happens between synchronisation points in each thread?
 - max-min time/energy of each non-synchronised code section
- What code blocks can (possibly) run in parallel with each other?
 - how many threads can be active simultaneously?
- How slow can blocks run and meet port deadlines?

The ENTRa tool architecture generally involves a trade-off – greater accuracy at lower level versus better analyses at the higher level. Direct analysis of source code means that we do not compile it to intermediate code or assembly code, but instead model its behaviour (semantics) directly. Despite the reduction in energy modelling precision of source code, some things get simpler at a higher level. For example precise analysis of dataflow between the program’s threads, determining which bits of code are most heavily used and which parts of the program can run concurrently is easier when the details of the machine are abstracted away. Such aspects of program behaviour are relevant for energy consumption; even an approximate energy profile for a high-level implementation of a system can help to make decisions that reduce power consumption later.

3.3.1 Usage and Interface

The tool takes an XC program and yields an HC IR representation of its thread and communication structure as explained below. It is run from the command line by a script named `xcreach.sh`. The command

```
xcreach.sh Tests/ProducerConsumer/main.xc
```

generates an output folder containing the output files, including one called (in this example) `main.synch.pl` which contains the `run` and `reach` relations outlined below, represented as Horn clauses.

3.3.2 Generation of a *Run* Relation

As presented before (Deliverable D3.1) our approach to analysis of source code is based on the following steps.

1. Small-step operational semantics for XC.
2. A CLP program implementing the operational semantics as an “interpreter”.

3. Partial evaluation of the interpreter (using Logen) with respect to (the abstract syntax tree of) an XC program.
4. Some further CLP transformations to “clean up” the resulting specialised interpreter.

The output of the steps above is a set of CLP clauses of the form

$$run_i(X_1, \dots, X_{n_i}) :- c(X_1, \dots, X_{n_i}, Y_1, \dots, Y_{n_j}), \quad run_j(Y_1, \dots, Y_{n_j}).$$

Here the predicates run_i and run_j represent the state of the computation at two program points labelled i and j respectively. The clause expresses a transition stating that if the computation is in state $run_i(X_1, \dots, X_{n_i})$ then there is a transition to state $run_j(Y_1, \dots, Y_{n_j})$, where $c(X_1, \dots, X_{n_i}, Y_1, \dots, Y_{n_j})$ is a constraint relating the state variables in the two states. The variables $X_1, \dots, X_{n_i}, Y_1, \dots, Y_{n_j}$ represent values of XC program variables, labels such as source line numbers, or instrumentation such as counters for energy or time.

A thread-modular approach. In single-threaded code we typically obtain one run predicate per program point. Applying this approach to multi-threaded programs presents several difficulties compared to analysis of single-threaded programs. A naive approach to modelling the operational semantics of multi-threaded code is not feasible due to the combinatorial explosion of possible states arising by concurrent interleaving of threads. To follow a similar approach with multi-threaded code, we would need to consider one run predicate for each *tuple of program points* from the program’s threads, representing the execution state of each thread.

Even leaving aside the explosion of possible tuples due to interleaving and assuming a deterministic thread scheduler, the number of possible code states explodes due to choice-points which are not resolved at analysis time. For example, considering two threads, each of which contains an if-then-else statement, and assuming that at analysis time we do not know which branch will be taken, we have to consider four possibilities, namely then-then, then-else, else-then or else-else branches that might run concurrently when running the threads. With just a few loops and branch expressions in the threads, the number of combinations of program points rises rapidly.

In a *thread-modular* approach the state of each thread is initially modelled separately as a single-threaded execution. The possible interaction of threads is considered as extra constraints on the behaviour of a single thread (sometimes called *interferences*). We now explain how we formalise and implement this approach for multi-threaded XC programs with synchronous channel communication.

Modelling XC threads and synchronization The interpreter is based on the semantics presented in Deliverable D3.1, but with some changes to reflect the thread-modular approach.

- The treatment of the `par` construct expresses the fact that each thread starts a run independently, and ignores synchronization. A multi-step relation (see next item) is used to express the start of the threads, each of which runs to completion. The final state σ' after executing $S_1 \parallel S_2$ is unknown. Note that if one or other of the threads does not terminate, then neither does the `par` statement. We return to this later.

$$\frac{\langle S_1 ; \text{join}(L')^{L'}, \sigma \rangle \longrightarrow^* \quad \langle S_2 ; \text{join}(L'')^{L''}, \sigma \rangle \longrightarrow^*}{\langle S_1 \parallel S_2, \sigma \rangle \xrightarrow{\text{fork}(L', L'')} \langle \text{skip}, \sigma' \rangle} \quad (\text{thread-start})$$

We also add an extra statement called `join(L)` to the end of each thread, where L is the label of the respective thread. This is a marker, whose operational behaviour is the same as `skip`, whose purpose is to allow the detection of the end of a thread's execution.

- The multi-step transition \longrightarrow^* appearing in the above transition is a simplified version of the multi-step transition introduced in the earlier semantics, ignoring the final state, but adding a label. Previously there was a relation $\langle S_0, \sigma_0 \rangle \longrightarrow^* \langle S_1, \sigma_1 \rangle$ expressing a multi-step “run” from $\langle S_0, \sigma_0 \rangle$ to $\langle S_1, \sigma_1 \rangle$. Ignoring the final configuration and adding a label we get a relation $\langle S_0^L, \sigma_0 \rangle \longrightarrow^*$. The label L gives information about the program point at the start of the run. The construct $\langle S_0^L, \sigma_0 \rangle \longrightarrow^*$ can be read as “there is a run starting at configuration $\langle S_0^L, \sigma_0 \rangle$ ”. It is defined as follows.

$$\overline{\langle \text{skip}^{end}, \sigma \rangle \longrightarrow^*} \quad (\text{step-skip})$$

$$\frac{\langle S_0, \sigma_0 \rangle \xrightarrow{L} \langle S_1, \sigma_1 \rangle \quad \langle S_1^{L'}, \sigma_1 \rangle \longrightarrow^*}{\langle S_0^L, \sigma_0 \rangle \longrightarrow^*} \quad (\text{step-trans})$$

The reason for ignoring the final state is that many of the programs we consider are perpetual processes that do not have any final state; we focus instead on which states are reachable from the initial state.

- The treatment of the `send x c` and `$x := \text{get } c$` commands is modified to ignore synchronization. The transition for `send x c` sends the value of x on the channel c without checking that the value is received; the transition for `$x := \text{get } c$` reads an unknown value from the channel c .

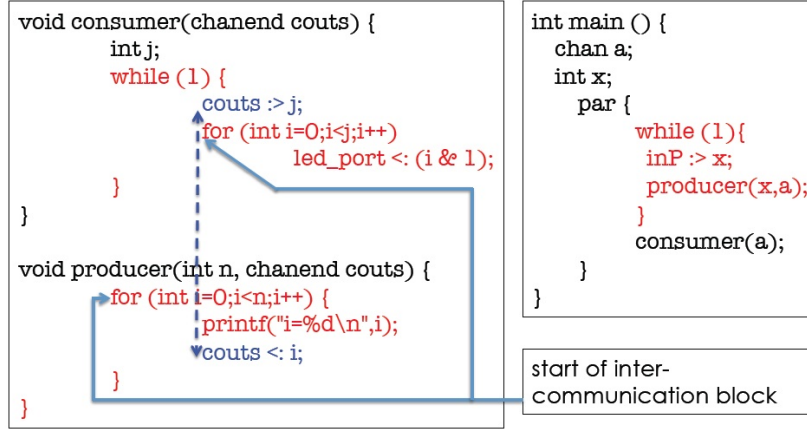


Figure 14: Simple Producer-Consumer XC program

3.3.3 Analysis of Threads and Communication Structure

In Figure 14 we show an example of a simple multi-threaded XC program. In the program there are two threads (the producer and the consumer), which share a channel. The producer writes values onto the channel, which are then processed by the consumer.

The semantics-based translation initially generates four sections of single-threaded CLP clauses, representing the following parts of the program.

- From the entry point (call to *main* procedure) up to the *par* statement.
- The producer thread.
- The consumer thread.
- The code following the *par* statement (which happens to be empty in this program).

3.3.4 Generating the *Reach* Relation

A reachability transition relation is generated directly from the *run* clauses shown above, by reversing the transition rules. That is, for each CLP clause of the form

$$run_i(X_1, \dots, X_{n_i}) :- c(X_1, \dots, X_{n_i}, Y_1, \dots, Y_{n_j}), \quad run_j(Y_1, \dots, Y_{n_j}).$$

there is a corresponding reachability transition of the form

$$reach_j(Y_1, \dots, Y_{n_j}) :- c(X_1, \dots, X_{n_i}, Y_1, \dots, Y_{n_j}), \quad reach_i(X_1, \dots, X_{n_i}).$$

that states that point *j* is reachable if point *i* is reachable and the transition constraint is satisfied. We also add a fact stating that the entry point is reachable.

Adding synchronization constraints to the reachability relation The semantics as shown above omits any modelling of parallelism or synchronization. The synchronization information is now added to the reachability relation as extra constraints. We add extra clauses to the reachability clauses, expressing the following.

- The start state of each thread is reachable, if the par statement containing the thread is reachable.
- The state immediately after the par statement is reachable if the join statements of both threads is reachable.
- The point immediately following a send $x\ c$ statement is reachable, if the send $x\ c$ statement is reachable *and* a corresponding statement $x := \text{get } c$ is reachable.
- The point immediately following a $x := \text{get } c$ statement is reachable, if the $x := \text{get } c$ statement is reachable *and* a corresponding statement send $x\ c$ is reachable.

In order to generate the last two rules, the CLP clauses are searched for matching channel communication points. The last two rules are over-approximations of the actual synchronization, in that different reachable states of the same get and send statements are not distinguished. Nonetheless it results in a safe over-approximation of the reachable states of each thread.

Analysis of the combined *run/reach* relations

- Analysis of the reachability relation permits dataflow analysis between threads, using well-established techniques from analysis of CLP. For example, in Figure 14, we can show that the values received by the consumer at the statement `couts :> j` range from 0 to $x-1$, where x is the value sampled from the port in the statement `inP :> x`. Although in a small program such dependencies are easy to track, such dataflow is not obvious in more complex code.
- Secondly, we can isolate the sections of code in the *run* relation that are not synchronized. This allows us to estimate the resources used by each thread between communications, and estimate the load balance between threads. For instance, analysis of the relation shows that if a value of 0 is sampled from the port in the statement `inP :> x`, the consumer is not invoked at all. More generally, between each channel send operation, the producer performs a constant amount of work, but the consumer does work proportional to the value sent. Thus the load imbalance increases with the value sent on the channel.

- Furthermore, analysis of the *run/reach* relations shows which sections of code can run in parallel. Threads started at the same *par* statement can run in parallel, upto the next synchronization point, and code in threads immediately following a channel communication can run in parallel, upto the next synchronization point.

Figure 14 illustrates how the tool could highlight the channel communication (in blue) and sections of code that run in parallel in each thread between communications (in red).

3.4 Optimization via Dynamic Voltage and Frequency Scaling (DVFS) and Task Scheduling

An optimal task scheduling can provide significant energy savings in XMOs chips as well as in the great majority of today's computing systems that support parallelism. Additional energy savings can be obtained by performing Dynamic Voltage and Frequency Scaling (DVFS), a feature offered by many modern devices. DVFS enables changes to voltage and frequency "on-the-fly" and allows obtaining energy savings while meeting task deadlines.

In this section we describe an optimization tool that we have developed for solving the following general problem: given a multicore system able to execute multiple threads per core and offering the possibility of performing DVFS per core, a set of tasks, each represented by its release time, deadline, and estimated power consumption (if available), find an *optimal task-core (thread) assignment*, so that all deadlines are met and the consumed energy is minimised. This definition includes task sets where the tasks can be related or unrelated. If there are dependencies among the tasks, any task cannot be released before all the tasks it depends on are finished. The dependency is expressed by the release time. If there are periodic tasks, the beginning of the periods can be considered as the task release time and the end of the periods as the task deadline.

Special cases can include one or more relaxations of the previous definition:

- The underlying hardware can be either multicore or multithreaded.
- It does not offer the DVFS feature.
- The tasks can all start at the same time.
- The deadlines do not have to exist. Note however that in this case it is not beneficial to scale down voltage and frequency indefinitely, since static power consumption becomes more significant than dynamic power consumption. Thus, in this case, minimal possible voltage and frequency do not necessarily imply minimal energy consumption.

3.4.1 Usage and Interface

The tool can be used standalone through the command line, but it can also be integrated into a more general tool, e.g., CiaoPP.

The input to the tool are the following files:

- A file that contains the necessary information about each task. There is a line per task with the following fields (given in this order):
 - *Task ID* - unique for each task.
 - *Release time*: the moment when the task becomes available.
 - *Estimated execution time*.
 - *Estimated power consumption* (if available).
 - *Deadline*: the moment until the task has to be finished; if it does not exist, the field should be left empty.
- A file which describes *the architecture*:
 - *Number of cores*.
 - *Number of threads per core*; if there are no multiple threads per core, it should contain 0.
 - Information about voltage and frequency scaling: if applicable, it should provide possible (V, f) pairs, if not, it should be empty.

The execution time can be set from the beginning by the programmer, or can be estimated by using some analysis tool. Also, the power consumption can be estimated via an analysis tool.

Some examples of tasks are the following:

```
Task 0 with release time: 7 deadline: 8 number of cycles: 21
Task 1 with release time: 9 deadline: 10 number of cycles: 33
Task 2 with release time: 1 deadline: 2 number of cycles: 23
...
```

Also, the underlying architecture can be specified in the following way:

```
Cores: 1
Threads per core: 8
Voltage: 0.95 Frequency: 500000000
Voltage: 0.87 Frequency: 400000000
Voltage: 0.80 Frequency: 300000000
```

Voltage: 0.80 Frequency: 150000000

Voltage: 0.75 Frequency: 100000000

Voltage: 0.70 Frequency: 50000000

where voltage and frequency are given in their corresponding basic units: Volts (V) and Hertz (Hz).

The output of the scheduler is a file with the following information for each task:

- The exact moment when its execution starts.
- The core (thread) on which the task is executed.
- (V, f) state of the core where the task is being executed.

An example of the output of the tool for a single core system is the following:

Scheduling of tasks on core: 1

Task 9 with release time: 1 deadline: 5 number of cycles: 40
frequency: 5e+07 scheduled at: 1 until 5

Task 18 with release time: 0 deadline: 2 number of cycles: 8
frequency: 5e+07 scheduled at: 0 until 1

Task 21 with release time: 1 deadline: 10 number of cycles: 19
frequency: 5e+07 scheduled at: 5 until 10

...

Scheduling of tasks on core: 8

Task 22 with release time: 5 deadline: 7 number of cycles: 24
frequency: 5e+07

scheduled at: 5 until 7

Task 4 with release time: 2 deadline: 3 number of cycles: 10
frequency: 5e+07

scheduled at: 2 until 3

Task 6 with release time: 4 deadline: 6 number of cycles: 9
frequency: 5e+07 scheduled at: 4 until 5

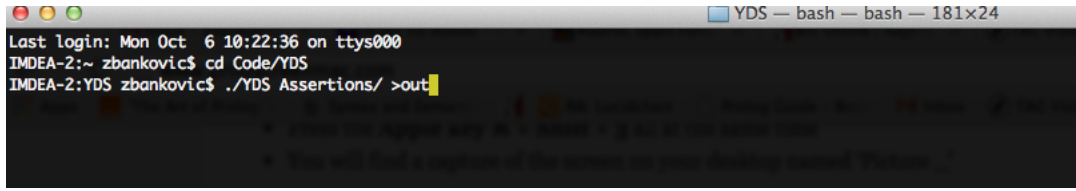
Task 23 with release time: 1 deadline: 9 number of cycles: 24
frequency: 5e+07 scheduled at: 1 until 2

scheduled at: 3 until 4

scheduled at: 7 until 9

Freq: 5e+07 assigned at 5

In order to use the scheduler tool as a standalone tool, we should provide two input files (the extension is not important since they are parsed as text files), namely *tasks* and *architecture* in the formats explained above. They are put in the same folder, and the tool distinguishes them by their format. Figure 15 shows a typical usage of the tool in the command line, where we use the YDS algorithm for scheduling (to be explained later). The input files are given in the folder `Assertions`, and the output is written in the file `out` in the output format explained above.



```

Last login: Mon Oct 6 10:22:36 on ttys000
IMDEA-2:~ zbankovic$ cd Code/YDS
IMDEA-2:YDS zbankovic$ ./YDS Assertions/ >out

```

Figure 15: Command line for using the YDS scheduling algorithm.

We have developed two implementations of the tool (scheduler), depending on the way the execution time and power consumption are estimated:

- *deterministic*, where those values are estimated as concrete (deterministic) values, and
- *stochastic*, where the values are treated as random variables, whose probability density function, as well as interdependence, is estimated using a probabilistic analysis.

3.4.2 Methodology and Scenarios for Using the Tool

Given that the scheduling tool has been designed to solve a quite general problem, it can be specialized and be applied in many different scenarios. Some typical scenarios are the following:

- *Embedded systems that support parallelism*: since these systems usually execute certain application, it can be possible to split the application into tasks that run in parallel. The scheduling tool is then applied on these subprograms. The parallelization of the application can be performed by a parallelizing compiler, or can be done manually by the programmer.
- *Small to large scale parallel computing systems*: these systems usually run a set of unrelated tasks, so the application of the scheduler is straightforward. However, if the level of parallelism they support is bigger than the number of tasks, the most extensive tasks can be further parallelized for a more efficient usage of the underlying system. In this case, the scheduler is applied in the same way as in the previous case.

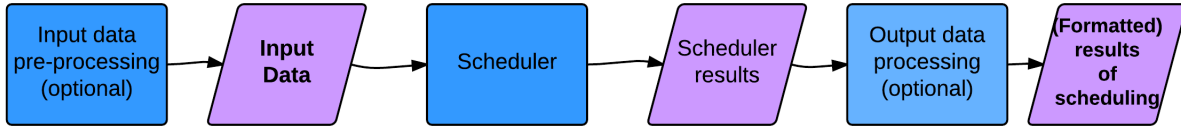


Figure 16: General Flowchart of the Scheduling Tool.

3.4.3 Architecture of the Tool

An overview of the general architecture of the tool is presented in Figure 16. If the scheduler is integrated in a tool chain that requires a different format for the input and output data, we would add modules for translating the input from the tool chain to the internal format of the scheduler, as well as for translating the output of the scheduler to the format required by the tool chain.

The core part of our tool, i.e., the scheduler, can be implemented using different algorithms. Currently, we have implemented a custom genetic algorithm based scheduler for both stochastic and deterministic scheduling [BLG14]. In addition, we have adapted the well-known YDS algorithm [YDS95], initially designed for DVFS-enabled single core, to a multicore environment using two different options for the initial assignment of tasks to cores:

1. The task is assigned to the core with the least load (at the moment of assigning the task).
2. The task is assigned to the core with the least density. The density is the total number of cycles divided by the interval duration, in the interval the task is supposed to be executed, i.e., the interval whose end points are the task release time and the task deadline.

3.4.4 Evaluation and Experimental Results

At the moment, the tool has been evaluated using synthetic data and/or typical power consumption of XMOs chips. The different experiments that we have performed confirm that:

- The application of DVFS can significantly reduce energy consumption: in our experiments, using the typical power consumption of XMOs chips, the savings can go up to 34% [BLG13].
- The stochastic scheduler can significantly improve the results of scheduling in the situations where real values deviate from the estimated ones used to create the deterministic scheduler [BLG14]. In our experiments, additional savings of up to 15.4% can be achieved.

- In the case of stochastic scheduling, if there is dependency among different inputs (execution time and power of different tasks), and with a proper model of it, additional savings can be obtained. We have performed experiments by controlling task dependency modeled with Gumbel copulas [Nel03], and we have achieved average improvements of around 15%, up to 20%, in comparison to the case where the dependency is not modeled, i.e., assuming task independence.

Although we still have not tested the schedulers on real data, the obtained rates confirm the potential of energy-aware scheduling coupled together with DVFS for optimising energy consumption.

4 Work in Progress

This section reports on work in progress towards developing a tool for Worst Case Energy Consumption (WCEC) based on the implicit path enumeration technique, and another tool for Horn clause verification.

4.1 Worst Case Energy Consumption Using Implicit Path Enumeration

A current area of research that is being investigated by the project is Worst Case Energy Consumption (WCEC) using Implicit Path Enumeration (IPE) techniques. This technique is frequently used to estimate Worst Case Execution Time (WCET) of a program and was firstly adopted in [JML06] for inferring the WCEC of a program. We adopt the same methodology using our parametric energy model. Currently a prototype tool is under development utilizing this analysis. In the next subsections a description of the techniques employed is provided along with sample outputs from an example benchmark using the tools that are currently under development.

4.1.1 Integer Linear Programming (ILP) Formulation

In our case we use the same formulation used in [LM97], but we replace the time cost of a CFG basic block with its energy cost provided by our energy model in a similar approach to [JML06]. The formulation adopted is as follows.

Let x_i be the number of times the basic block B_i is executed when the longest path of the program CFG is taken. Let c_i be the energy cost assign to the B_i block using our energy model. The energy cost of a CFG block is calculated by aggregating the energy cost of all ISA instructions included in the block. The energy cost of each ISA instruction is provided by the parametric ISA energy model as described in Deliverable D2.2. Putting them all together the energy consumption of a program with N basic blocks is given by the expression:

$$\sum_{i=1}^N c_i x_i \quad (1)$$

Considering that Equation (1) represents the energy consumption of our program, the number of values the x_i 's can take are constrained by the program structure and functionality and therefore by the data inputs. Solving this equation to get the upper bounds of energy consumption requires maximizing it and taking into account the restrictions driven by the program structure and functionality. Stating those restrictions in the form of linear constraints enables the use of ILP to maximize the Equation (1).

4.1.2 Structural Constraints

To extract the structural constraints of a program, the CFG is created and annotated. The edges are annotated with d_i 's and the basic blocks with x_i 's variables, which represents the number of times the edges or blocks are exercised during the program execution respectively. In order to have a program that normally executes and returns the number of times you enter a basic block must be equal to the number of times you exit the basic block. Hence the x_i of a basic block is equal to the sum of d_i s of edges entering the block and the sum of d_i s of edges exiting the block. For clarity the annotated CFG of `jpegDct`, one of the benchmarks analyzed, is given in Figure 17 alongside with the retrieved structural constraints in linear expressions. The edges d_0 and d_{10} , representing the start and the exit of the CFG respectively, are set to one as they can be executed only once. The source code for this example is given in Code Sample 3

Code Sample 3: The `jpegDct` benchmark source code

```
void jpegdct(short d[], short r[])
{
    long int t[12];
    int v=0;
    short i, j, k, m, n, p, ic, ik;
    for (ik=2; ik; ik--) {
        for (i = 8; i; i--, v+=8) {
            for (j = 3; j>=0; j--) {
                // some code
            }
            // some code
        }
    }
}
```

In the case of function calls, f -edges are used to connect the caller function to the start of the CFG of the callee function. Then the f -edge is treated in the same way as a d -variable to construct the structural constraints in the caller function.

4.1.3 Functionality Constraints

Functionality constraints are used to characterize anything that can affect the program functionality. This includes loop bounds and path information, and usually can be only specified by the programmer as static analysis cannot extract this kind of information in most of the cases. The minimum requirement for user input to enable the bounding of the problem, is the loop bounds. This is standard also in timing analysis. Providing this kind of information is usually easy for

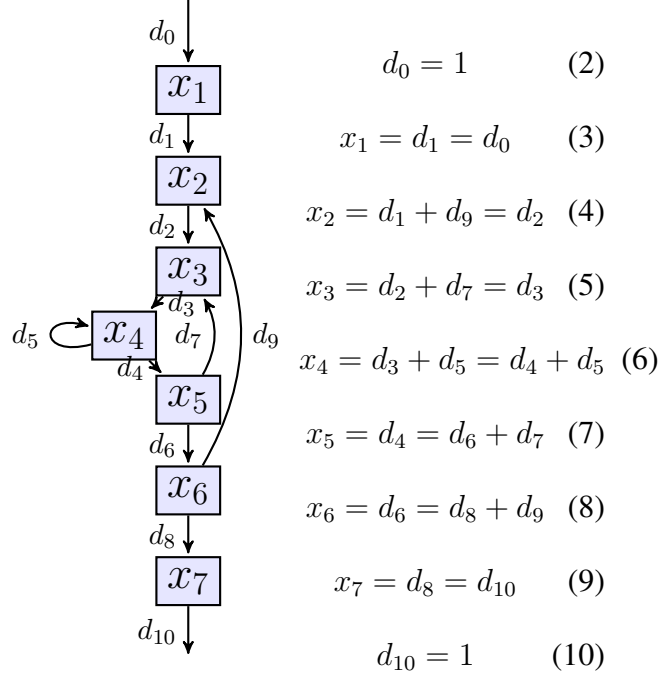


Figure 17: jpegDct benchmark annotated CFG and it's corresponding Structural Constraints formed as linear expressions

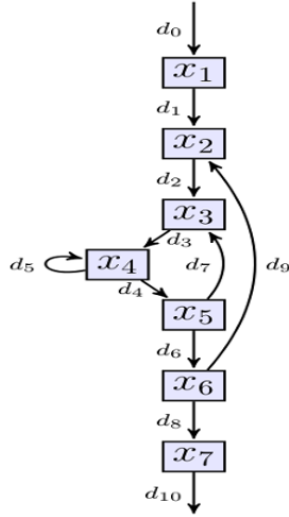
the programmer, as the bounds of the loops usually are associated by the programmer. Figure 18 demonstrates the loops bounds given by the user for the jpegDct benchmark to our prototype tool.

Finally in Equation (17) the ILP formulated function, extracted by our prototype tool, for the jpegDct benchmark is given. This function is passed to one of the many ILP open source solvers, namely the `lpsolver`, to be maximized and retrieve the upper bound on the energy consumption for this program. Figure 19 shows the output of the solver. In the output the energy cost of the worst case execution path is given in (nano joules) and also the times each BB and edge will be executed using this path.

$$\max : b_1 \times x_1 + b_2 \times x_2 + b_3 \times x_3 + b_4 \times x_4 + b_5 \times x_5 + b_6 \times x_6 + b_7 \times x_7 \quad (17)$$

4.2 Tools for Horn Clause Verification

Constrained Horn clauses (CHCs) provide a suitable intermediate form for expressing the semantics of a variety of programming languages (imperative, functional, concurrent, etc.) and computational models (state machines, transition systems, big- and small-step operational semantics, Petri nets, etc.) [PGS98, BG09, GLPR12]. As a result it has been used as a target



$$1x1 \leq x2 \quad (11)$$

$$x2 \leq 2x1 \quad (12)$$

$$1x2 \leq x3 \quad (13)$$

$$x3 \leq 8x2 \quad (14)$$

$$1x3 \leq x4 \quad (15)$$

$$x4 \leq 4x3 \quad (16)$$

Figure 18: jpegDct benchmark annotated CFG and it's corresponding Functional Constraints formed as linear expressions

```

singleThread : bash - Konsole
kyriakos@kakos:~/Measurements/BristolLLVMAnalysisPaperBenchmarks/XCbench/jpegdct/singleThread$ lp_solve FinalISA.lp
Value of objective function: 4.19133e+06
Actual values of the variables:
x1          1
x2          2
x3         16
x4         64
x5         16
x6          2
x7          1
d1          1
d9          1
d2          2
d7         14
d3         16
d5         48
d4         16
d6          2
d8          1
kyriakos@kakos:~/Measurements/BristolLLVMAnalysisPaperBenchmarks/XCbench/jpegdct/singleThread$

```

Figure 19: jpegDct ILP formulated function

language for software verification and has been chosen as the main internal representation for the ENTRA project tools. Recently there is growing interest in CHC verification from both the logic programming and software verification communities, and several verification techniques and tools have been developed for CHC [GGL⁺12, dMB08, DAFPP14, HKG⁺12]. Pure constraint logic programs (CLP) are semantically and syntactically identical to CHCs

We have constructed a toolkit based on established CLP analysis and transformation techniques, whose purpose is verification of assertions in programs as well as discovery of useful invariants. Its main components are the following.

Unfolding. Let P be a set of CHCs and $c_0 \in P$ be $H(X) \leftarrow \mathcal{B}_1, p(Y), \mathcal{B}_2$ where $\mathcal{B}_1, \mathcal{B}_2$ are possibly empty conjunctions of atomic formulas and constraints. Let $\{c_1, \dots, c_m\}$ be the set of clauses of P that have predicate p in the head, that is, $c_i = p(Z_i) \leftarrow \mathcal{D}_i$, where the variables of these clauses are standardised apart from the variables of c_0 and from each other. Then the result of unfolding c_0 on $p(Y)$ is the set of CHCs $P' = P \setminus \{c_0\} \cup \{c'_1, \dots, c'_m\}$ where $c'_i = H(X) \leftarrow \mathcal{B}_1, Y = Z_i, \mathcal{D}_i, \mathcal{B}_2$. The equality $Y = Z_i$ stands for the conjunction of the equality of the respective elements of the vectors Y and Z_i . It is a standard result that unfolding a clause in P preserves P 's minimal model [PP99]. In particular, $P \models \text{false} \equiv P' \models \text{false}$.

Specialisation. A set of CHCs P can be specialised with respect to a query. Assume A is an atomic formula; then we can derive a set P_A such that $P \models A \equiv P_A \models A$. P_A could be simpler than P , for instance, parts of P that are irrelevant to A could be omitted in P_A . In particular, the CHC verification problem for P_{false} and P are equivalent. There are many techniques in the CLP literature for deriving a specialised program P_A . Partial evaluation is a well-developed method [Gal93, Leu99].

We make use of a form of specialisation known as forward slicing, more specifically redundant argument filtering [LS96], in which predicate arguments can be removed if they do not affect a computation. Given a set of CHCs P and a query A , denote by P_A^{raf} the program obtained by applying the RAF algorithm from [LS96] with respect to the goal A . We have the property that $P \models A \equiv P_A^{\text{raf}} \models A$ and in particular that $P \models \text{false} \equiv P_{\text{false}}^{\text{raf}} \models \text{false}$.

Query-answer transformation. Given a set of CHCs P and an atomic query A , the query-answer transformation of P with respect to A is a set of CHCs which simulates the computation of the goal $\leftarrow A$ in P , using a left-to-right computation rule. Query-answer transformation is a generalisation of the magic set transformations for Datalog. For each predicate p , two new predicates p_{ans} and p_{query} are defined. For an atomic formula A , A_{ans} and A_{query} denote the replacement of A 's predicate symbol p by p_{ans} and p_{query} respectively. Given a program P and

query A , the idea is to derive a program P_A^{qa} with the following property $P \models A$ iff $P_A^{\text{qa}} \models A_{\text{ans}}$. The A_{query} predicates represent calls in the computation tree generated during the execution of the goal. For more details see [DR94, GdW93, CD95]. In particular, $P_{\text{false}}^{\text{qa}} \models \text{false}_{\text{ans}} \equiv P \models \text{false}$, so we can transform a CHC verification problem to an equivalent CHC verification problem on the query-answer program generated with respect to the goal $\leftarrow \text{false}$.

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

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

In our applications, we use splitting to create separate predicates for clauses for a given predicate whose constraints are mutually exclusive. For example, given the clauses $\text{new3}(A, B) :- A < 99$, $\text{new4}(A, B)$ and $\text{new3}(A, B) :- A \geq 100$, $\text{new5}(A, B)$, we produce two new predicates, since the constraints $A < 99$ and $A \geq 100$ are disjoint. The new predicates are defined by clauses $\text{new3}_1(A, B) :- A < 99$, $\text{new4}(A, B)$ and $\text{new3}_2(A, B) :- A \geq 100$, $\text{new5}(A, B)$, and all calls to new3 throughout the program are unfolded using these new clauses. Splitting has been used in the CLP literature to improve the precision of program analyses, for example in [SDS01]. In our case it improves the precision of the convex polyhedron analysis discussed below, since separate polyhedra will be maintained for each of the disjoint cases. The correctness of splitting can be shown using standard transformations that preserve the minimal model of the program (with respect to the predicates of the original program) [PP99]. Assuming that the predicate false is not split, we have that $P \models \text{false} \equiv P^{\text{split}} \models \text{false}$.

Convex polyhedron approximation. Convex polyhedron analysis [CH78] is a program analysis technique based on abstract interpretation [CC77]. When applied to a set of CHCs P it constructs an over-approximation M' of the minimal model of P , where M' contains at most one constrained fact $p(X) \leftarrow \mathcal{C}$ for each predicate p . The constraint \mathcal{C} is a conjunction of linear inequalities, representing a convex polyhedron. The first application of convex polyhedron analysis to CLP was by [BK96]. Since the domain of convex polyhedra contains infinite increasing

chains, the use of a widening operator is needed to ensure convergence of the abstract interpretation. Furthermore much research has been done on improving the precision of widening operators. One technique is known as widening-upto, or widening with thresholds [HPR94].

Recently, a technique for deriving more effective thresholds was developed [LCJG11], which we have adapted and found to be effective in experimental studies. The thresholds are computed by the following method. Let T_P^C be the standard immediate consequence operator for CHCs, that is, $T_P^C(I)$ is the set of constrained facts that can be derived in one step from a set of constrained facts I . Given a constrained fact $p(\bar{Z}) \leftarrow C$, define $\text{atomconstraints}(p(\bar{Z}) \leftarrow C)$ to be the set of constrained facts $\{p(\bar{Z}) \leftarrow C_i \mid C = C_1 \wedge \dots \wedge C_k, 1 \leq i \leq k\}$. The function atomconstraints is extended to interpretations by $\text{atomconstraints}(I) = \bigcup_{p(\bar{Z}) \leftarrow C \in I} \{\text{atomconstraints}(p(\bar{Z}) \leftarrow C)\}$.

Let I_\top be the interpretation consisting of the set of constrained facts $p(\bar{Z}) \leftarrow \text{true}$ for each predicate p . We perform three iterations of T_P^C starting with I_\top (the first three elements of a “top-down” Kleene sequence) and then extract the atomic constraints. That is, thresholds is defined as follows.

$$\text{thresholds}(P) = \text{atomconstraints}(T_P^{C(3)}(I_\top))$$

A difference from the method in [LCJG11] is that we use the concrete semantic function T_P^C rather than the abstract semantic function when computing thresholds. The set of threshold constraints represents an attempt to find useful predicate properties and when widening they help to preserve invariants that might otherwise be lost during widening. See [LCJG11] for further details. Threshold constraints that are not invariants are simply discarded during widening.

4.2.1 Combining Off-the-shelf Tools: Experiments

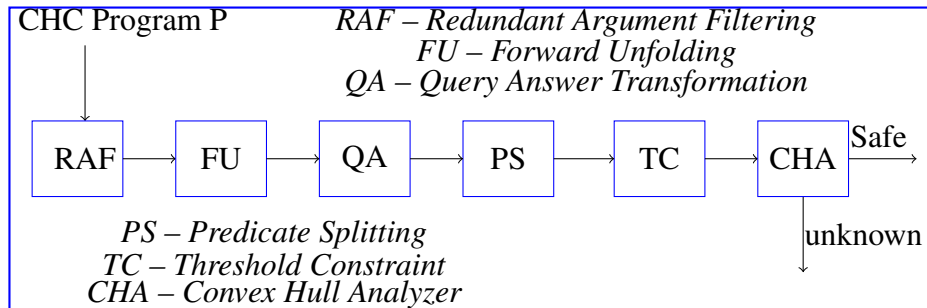


Figure 20: The basic tool chain for CHC verification.

The motivation for our tool chain, summarised in Figure 20, comes from our example program, which is a simple yet challenging program. We applied the tool chain to a number of

benchmarks from the literature, taken mainly from the repository of Horn clause benchmarks in SMT-LIB2 (<https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/>) and other sources including [GNS⁺13] and some of the VeriMap benchmarks [DAFPP14]. We selected these examples because many of them are considered challenging since they cannot be solved by one or more of the state-of-the-art-verification tools discussed below. Programs taken from the SMT-LIB2 repository are first translated to CHC form. The results are summarised in Table 3.

In Table 3, columns Program and Result respectively represent the benchmark program and the results of verification using our tool combination. Problems marked with (*) could not be handled by our tool chain since they contain numbers which do not fit in 32 bits, the limit of our Ciao Prolog implementation, whereas problems marked with (**) are solvable by simple ad hoc modification of the tool chain.

Problems such as `systemc-token-ring.01-safeil.c` contain complicated loop structure with large strongly connected components in the predicate dependency graph. As a result, our convex polyhedron analysis tool is unable to derive the required invariant. However overall results show that our simple tool chain begins to compete with advanced tools like HSF [GGL⁺12], VeriMAP [DAFPP14], TRACER [JMNS12], *etc.* We do not report timings, though all these results are obtained in a matter of seconds, since our tool chain is not at all optimised, relying on file input-output and the individual components are often prototypes.

Table 3: Experiments results on CHC benchmark program

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

References

- [BCH⁺11] F. Bueno, M. Carro, M. Hermenegildo, R. Haemmerlé, P. López-García, E. Mera, J.F. Morales, and G. Puebla-(Eds.). The Ciao System. Ref. Manual (v1.14). Technical report, July 2011. Available at <http://ciao-lang.org>.
- [BG09] Gourinath Banda and John P. Gallagher. Analysis of Linear Hybrid Systems in CLP. In Michael Hanus, editor, *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17-18, 2008*, volume 5438 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2009.
- [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 of *LNCS*, pages 204–223, August 1996.
- [BLG13] Z. Banković and P. López-García. Genetic Algorithm-based Allocation and Scheduling for Voltage and Frequency Scalable X MOS Chips. In Jeng-Shyang Pan, Marios M. Polycarpou, Micha Woniak, Andr C.P.L.F. Carvalho, Hector Quintin, and Emilio Corchado, editors, *Hybrid Artificial Intelligent Systems*, volume 8073 of *Lecture Notes in Computer Science*, pages 401–410. Springer, 2013.
- [BLG14] Z. Banković and P. Lopez-Garcia. Stochastic vs. Deterministic Evolutionary Algorithm-based Allocation and Scheduling for X MOS Chips. *Neurocomputing*, pages 82–89, 2014.
- [CC77] 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.
- [CD95] Michael Codish and Bart Demoen. Analyzing logic programs using "PROP"-ositional logic programs and a magic wand. *J. Log. Program.*, 25(3):249–274, 1995.
- [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.

- [DAFPP14] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verimap: A tool for verifying programs through transformations. In Erika Ábrahám and Klaus Havelund, editors, *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 568–574. Springer, 2014.
- [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.
- [DR94] S. Debray and R. Ramakrishnan. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming*, 18:149–176, 1994.
- [EG13] K. Eder and N. Grech, editors. *Common Assertion Language*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013. Deliverable 2.1, <http://entraproject.eu>.
- [EKG14] K. Eder, S. Kerrison, and K. Georgiou, editors. *Low-Level Energy Models*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), May 2014. Deliverable 2.2, <http://entraproject.eu>.
- [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.
- [GdW93] J. P. Gallagher and D.A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation, Workshops in Computing*, pages 151–167. Springer-Verlag, 1993.
- [GGL⁺12] Sergey Grebenschikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. HSF(C): A software verifier based on Horn clauses - (competition contribution). In Cormac Flanagan and Barbara König, editors, *TACAS*, volume 7214 of *LNCS*, pages 549–551. Springer, 2012.
- [GLPR12] Sergey Grebenschikov, 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, Beijing, China - June 11 - 16, 2012*, pages 405–416. ACM, 2012.

- [GNS⁺13] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Failure tabled constraint logic programming by interpolation. *TPLP*, 13(4-5):593–607, 2013.
- [HKG⁺12] Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. A verification toolkit for numerical transition systems - tool paper. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 247–251. Springer, 2012.
- [HPR94] N. Halbwachs, Y. E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *Proceedings of the First Symposium on Static Analysis*, volume 864 of *LNCIS*, pages 223–237, September 1994.
- [JML06] R. Jayaseelan, T. Mitra, and Xianfeng Li. Estimating the worst-case energy consumption of embedded software. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 81–90, April 2006.
- [JMNS12] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. Tracer: A symbolic execution tool for verification. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 758–766. Springer, 2012.
- [LCJG11] Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. Widening with thresholds for programs with complex control graphs. In Tefik Bultan and Pao-Ann Hsiung, editors, *ATVA 2011*, volume 6996 of *Lecture Notes in Computer Science*, pages 492–502. Springer, 2011.
- [Leu99] Michael Leuschel. Advanced logic program specialisation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation - Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 271–292. Springer, 1999.

- [LG13] P. López-García, editor. *A General Framework for Resource Consumption Analysis and Verification*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013. Deliverable 3.1, <http://entraproject.eu>.
- [LKS⁺13] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M.V. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Pre-proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, September 2013.
- [LM97] Y.T.-S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(12):1477–1487, Dec 1997.
- [LS96] Michael Leuschel and Morten Heine Sørensen. Redundant argument filtering of logic programs. In John P. Gallagher, editor, *Logic Programming Synthesis and Transformation, 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 28-30, 1996, Proceedings*, pages 83–103, 1996.
- [May13] D. May. The XMOS XS1 architecture. available online: <http://www.xmos.com/published/xmos-xs1-architecture>, 2013.
- [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.
- [PGS98] J.C. Peralta, J. P. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In G. Levi, editor, *Static Analysis. 5th International Symposium, SAS'98, Pisa*, volume 1503 of *LNCS*, pages 246–261, 1998.
- [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.

- [SDS01] Alexander Serebrenik and Danny De Schreye. Inference of termination conditions for numerical loops in Prolog. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR 2001*, volume 2250 of *Lecture Notes in Computer Science*, pages 654–668. Springer, 2001.
- [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*, 13(4-5):1–14, August 2013.
- [SLGH14] A. Serrano, P. Lopez-Garcia, and M. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming, 30th Int’l. Conference on Logic Programming (ICLP’14) Special Issue*, 14(4-5):739–754, 2014.
- [YDS95] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0:374, 1995.