ENTRA

318337

Whole-Systems ENergy TRAnsparency

Energy-Aware Software Development Methods and Tools

Deliverable number:	D1.2
Work package:	Energy-Aware Software Engineering (WP1)
Delivery date:	31 December 2015 (39 months)
Actual date:	1 March 2016
Nature:	Prototype
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 reviews the field of energy-aware software development and describes how the areas most closely related to the ENTRA project have advanced during the execution of the project. This establishes the context for a description of different activities and scenarios for energy-aware software development. Then, it demonstrates (by using a tools front end developed in the project) how the final energy-aware prototype tools for analysis, verification and optimisation can be integrated into tool-chains for energy-aware software development processes and life-cycle. Finally, such prototype tools, which are components providing the functionality supporting the activities of energy-aware software development, are described. Links to repositories containing the prototype tools and other software developed are available at the ENTRA project website[1].

The deliverable includes the following attachments:

- D1.2.1 [LGK+16]: Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR. Foundational and Practical Aspects of Resource Analysis. Fourth International Workshop FOPARA 2015, Revised Selected Papers, Lecture Notes in Computer Science, Springer. To appear.

- D1.2.2 [HK16]: *Swallow: Building an Energy-Transparent Many-Core Embedded Real-Time System*. International Conference on Design, Automation and Test in Europe (DATE 2016), Dresden, Germany, IEEE, March 2016.

- D1.2.3 [HLGL+16]: *A Transformational Approach to Parametric Accumulated-cost Static Profiling*. Thirteenth International Symposium on Functional and Logic Programming (FLOPS 2016), LNCS, Springer, March 2016.

- D1.2.4: *Integrating energy modelling into the development process: A Makefile approach*. Technical brief, December 2015.

[1] `http://entraproject.eu/software-and-tools`.

Contents

1 Introduction

Deliverable D1.1 gave 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. It included a broad description of the kinds of information that energy-aware software engineers needs to know, as well as a summary of the tool support that could provide such information and assist in energy optimization.

In this deliverable we start by reviewing the field of energy-aware software development, identifying the areas most closely related to the ENTRA project, and describing how the field has advanced during the lifetime of the ENTRA project (Section 2).

Then, we describe different activities and scenarios for energy-aware software development (Section 3). In order to demonstrate how the prototype tools and components for analysis, verification and optimisation can be integrated into a tool-chain for energy-aware software development, we have implemented a front-end that provides a friendly Graphical User Interface, that we call the ENTRA tools front end (Section 4). Such components, which provide functionality supporting the activities of energy-aware software development, are described in Section 5.

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. In the ENTRA project we have followed 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 5 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.

The strategy is to migrate tool components from the second stream to the XC tool-chain once they reach a mature enough state (in the longer term). The ENTRA tools front end allows us to experiment with integration of prototype tools with components from the mainstream tool-chain such as the XC compiler and the LLVM framework.

Links to repositories containing the prototype tools and other software developed as part of this deliverable are available at the ENTRA project website[2].

[2] `http://entraproject.eu/software-and-tools`.

2 Overview of the Field of Energy-aware Software Development

In this section we review the field of energy-aware software development, and identify the areas most closely related to the ENTRA project. We also look at how the field has advanced during the lifetime of the ENTRA project.

2.1 Green IT

There has been a growth of interest in the field of *Green IT* [KCWCW10, NKD13, CFS12, NKD13, MA13] since approximately 2010; for example the conference series International Green And Sustainable Computing Conference[3] started in 2011 and the IEEE technical area of Green Computing[4] was launched in 2010. The Energy Aware COmputing workshop series[5] was initiated in Bristol in 2011. Since the start of the ENTRA project in 2012 dedicated workshops such as GREENS[6] and SMARTGREENS[7] have been launched.

Green IT covers energy aspects of the complete life-cycle and context of ICT systems, including software and hardware, development energy costs, maintenance and deployment energy costs, cooling costs, the energy costs of communication infrastructure, raw materials and disposal costs and a host of other energy costs and environmental effects associated directly or indirectly with software systems.

2.2 Energy-aware software development

Energy-aware software development is only one aspect of Green IT; it is only concerned with the energy efficiency of software, that is, the energy costs directly attributable to execution of programs. The energy-aware software engineer cannot in general be aware of the whole Green IT field, which involves complex dependencies and tradeoffs.

In short, energy-aware software development concerns the use of tools and methods to allow *energy consumption as a first-class software design goal*. The goal could be to increase energy efficiency or to achieve stated energy targets, for a given ICT application running on a given hardware platform. Energy-awareness for software development thus requires an understanding of the implications for energy consumption of design decisions in the software.

[3] `http://igsc.eecs.wsu.edu/` (formerly International Green Computing Conference (IGCC))
[4] `http://sameekhan.org/tagc/`
[5] `http://www.cs.bris.ac.uk/Research/Micro/eaco.jsp`
[6] `http://greens.cs.vu.nl/`
[7] `http://www.smartgreens.org/`

Very few programmers at present have much idea of how much energy their programs consume, or which parts of the program use the most energy. The development of software tools and techniques for resource-awareness (including energy-awareness) is now attracting more attention, but the field remains somewhat fragmented. Resource analysis is the topic of dedicated workshops such as FOPARA[8]. However, energy modelling is mainly studied in different application contexts such as embedded systems, high-performance systems, mobile systems and so on, rather than as a coherent set of techniques applicable to any software-based system.

The ENTRA project marks itself out from the current state of the art in three main respects.

- It takes a generic approach, not driven by any particular class of applications, platforms or programming languages.
- It combines energy analysis and energy modelling and the interaction between them.
- It considers the tools needed to support energy-aware software development.

2.3 Motivation for energy-aware software development

Environmental impact. The energy consumed by ICT is growing both in absolute terms and as a proportion of the global energy consumption and thus plays an important role in meeting the targets of the Europe 2020 Agenda, which includes a goal to reduce greenhouse gas emissions by at least 20% compared to 1990 levels. Every device, from autonomous sensor systems operating at the mW level to high performance computing (HPC) systems and data centres requiring tens of MWs for operation, consumes a certain amount of energy which results in the emission of CO_2.

Although energy is ultimately consumed by physical processes in the hardware, the software controls the hardware and indeed typically causes a great deal of energy waste by inefficient use of the hardware. This waste cannot be recovered by relying on the development of more energy-efficient hardware – increasing the energy efficiency of the software is the most effective approach to reducing overall energy consumption.

In many cases the energy efficiency of software has a direct positive effect on the efficiency of other energy-related aspects of systems. Obvious case are cooling costs and battery costs – cooling requirements for data centres are directly related to the power dissipated by the computations, while for mobile systems the number of battery replacements or recharges is similarly reduced if software is more energy-efficient.

[8] `http://resourceanalysis.cs.ru.nl/fopara/`

Strategic impact. The energy efficiency of ICT systems plays a critical role in exploiting the massive amounts of information available in data centres, and the full vision of the so-called Internet of Things. The power requirement of a data centre is typically measured in tens of MW, including cooling costs, while the Internet of Things generates increasing demand for a huge number of very low-power devices. The dream of "wireless sensors everywhere" is accompanied by the nightmare of battery replacement and disposal unless the energy requirements of devices can be lowered to enable them to be powered by energy harvesters or RF power sources.

Development costs of energy-efficient software. In the current state of the art, development costs for energy-efficient systems are higher than for energy-wasteful systems due to the extra effort required to take energy consumption into account. This is a significant barrier to making energy efficiency a first-class design goal.

The motivations for ENTRA research in energy-aware software development can thus be summarised as follows.

1. To lower the energy costs directly attributable to software execution, helping to reduce the environmental impact of ICT and to enable the next generation of ambient low-power devices.

2. To lower energy costs indirectly caused by software, such as the cost of cooling, power supplies, battery replacement and recharging.

3. To reduce the costs of the process of developing energy-efficient systems, by developing tools and techniques to assist the energy-aware developer.

2.4 Techniques for application software energy efficiency

One of the first works to stress the general importance of software energy efficiency, and identify aspects of software that affect energy consumption, was by Roy and Johnson [RJ97]. Since then the topic has been addressed mostly in the context of specialised application areas such as high-performance computing, embedded systems and mobile systems. There has been some more generic research in the past five years in the context of green computing [Goe13, Dew14, dSCM+12]. However the mainstream view remains that energy efficiency is a concern for hardware designers. Low-power architecture, for example, is still a very active area of research.

ENTRA Deliverable D4.1 [Gal14] summarised a number of software-based approaches to achieving lower energy consumption. Here we briefly review these along with other techniques described in [Lar11, SA12].

2.4.1 Computational efficiency

Firstly, there is a strong correlation between time and energy consumption for a given platform running a single computation thread. There are two reasons for this: less time means fewer instructions and secondly when the task is finished the processor can revert to a lower-power state for the excess time that a less efficient algorithm would use. The latter is called the "race to idle" in [SA12]. The correlation between time and energy is especially strong when asymptotic complexity is considered. It is highly likely, for example, that a single-threaded task that has $O(n^2)$ time consumption also has $O(n^2)$ energy consumption. Thus one of the main concerns of the energy-aware programmer, even with no knowledge of the energy consumption of the hardware, is to find computationally efficient algorithms and data structures suited to the task at hand.

2.4.2 Low-level or intermediate code optimisation

Deliverable D4.1 reviewed a range of techniques for low-level code energy optimisations, which could in principle be carried out by a compiler. These range from register allocation policies to avoid overheating a few intensively-used registers, use of VLIW (Very Long Instruction Word) instructions and vectorisation, to exploitation of low-power processor states using frequency and voltage scaling (DVFS). Note that such optimisations, in contrast to computational efficiency, are highly platform-dependent and rely on a platform energy model expressed at the level of low-level code. Computational efficiency as described in Section 2.4.1 is also important in that low-level code optimisations are most effective when applied to frequently executed sections of code, such as tight inner loops, where a small savings in energy can make a significant different to the overall computation.

It was also noted in Deliverable D4.1 that some energy optimisations rely on advanced compile-time (i.e. static) analysis. For example, knowledge of thread load imbalance and knowledge of predictable idle periods when processors can be put into low-power states are difficult to apply in the current compiler state of the art, since the analyses providing this knowledge are still emerging research areas.

2.4.3 Parallelism

The relationship between computational efficiency, time consumption and energy consumption is more complex for parallel than for sequential code. A multithreaded solution using multiple cores is often more energy-efficient than a single-threaded solution, even though the total amount of work done is greater for the multithreaded code, due to the extra instructions needed for

communication and synchronisation. The savings are mainly due to the fact that the overall task time is reduced, and so the processor(s) can revert sooner to a low-power state (the "race to idle" mentioned earlier).

Secondly, there can be energy savings if one or more cores can be run more slowly and still achieve the same overall task time as the sequential code. This is because power (P), frequency (f) and voltage (V) are related by the equation $P = cV^2f$ where c is a constant. Thus slowing down the processor (reducing f) saves power but not overall energy since the computation time is increased proportionally. However, a lower frequency is typically accompanied by a lower voltage, and the power/energy savings are quadratic in relation to voltage reduction.

2.4.4 Data and communication efficiency

Energy can be saved by minimising data movement. This can be achieved by writing software that reduces data movement by using appropriate data structures, by understanding and exploiting the underlying system's memory hierarchy and by designing multithreaded code that reduces the cost of communication among threads.

For example the size of blocks read and written to memory and external storage can have a major impact on energy efficiency, while memory layout of compound data structures should match the intended usage in the algorithm, so that consecutively referenced data items are stored adjacently if possible. In multithreaded code, consolidating all read-writes to or from disk to a single thread can reduce disk contention and consequent disk-head thrashing [SA12]. Furthermore, knowledge of the relative communication distances for inter-core communication can be used to place frequently communicating threads close to each other [HK16] thus reducing communication energy costs.

2.5 Tool support for energy-aware software development

Given the potential energy optimisations in described above, we identify various classes of tool support for energy-aware software development.

- Tools for energy modelling and transparency through the layers.
  - Tool role: to make visible energy consumption associated with programs at different levels.
  - Energy mappings showing relation between energy of ISA blocks, intermediate code blocks and source statement blocks.

- Energy consumption analysis.
  - Tool role: to show how energy models can be used by static analysis to analyse energy consumption.
  - Parametric energy expressions derived and displayed (as functions, or as graphs).

    Static profiling showing distribution of energy in a program, e.g. highlighting hot blocks.
- Energy simulation.
  - Tool role: similar to energy analysis tools, but using simulations based on an energy model.
  - Components: Simulation-based energy profiling of code, with suitable display of results.
- Energy verification.
  - Tool role: to show that energy specifications can be checked, and constraints derived giving conditions under which specs are satisfied.
  - Display of specifications, and results and interpretation of verification.
- Energy optimization and design space exploration.
  - Tool role: to use energy information (from any of the above tools) in manual or automatic (compiler-based) energy optimisation.
  - Energy optimising compiler.

    Exploration of thread and communication behaviour with implications for improved design with respect to energy consumption.

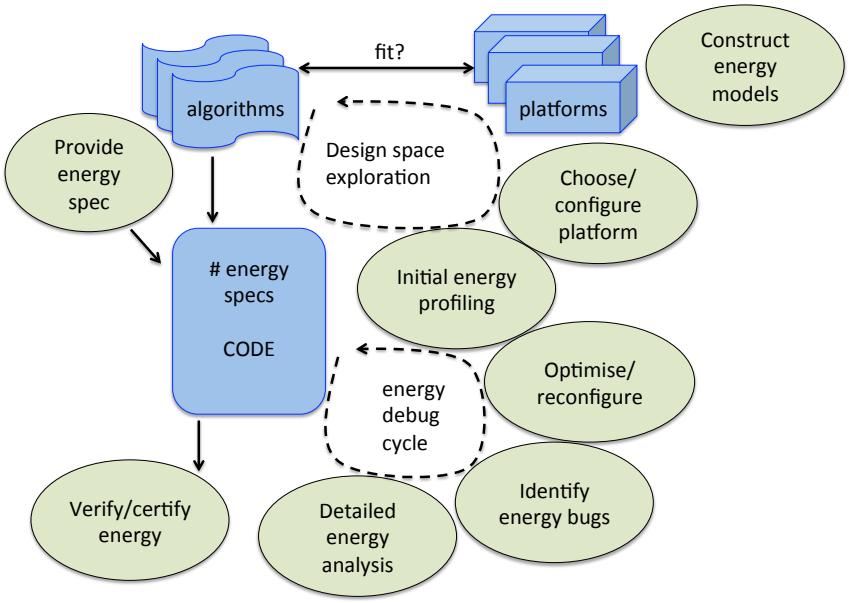


Figure 1: Energy-aware software engineering activities.

3 Software Engineering Activities and Scenarios

We now look at energy-aware software from the designer's and developer's point of view. What are the activities that distinguish energy-aware design and development from standard approaches in which energy is considered at the end of the development process, if at all? In Section 3.1 we identify a number of generic activities that play an important role in energy-aware software engineering. In Section 3.2 we make the discussion a little more concrete by sketching scenarios in which these activities are applied.

3.1 Energy-aware software engineering activities

In this section we describe the most important activities involved in energy-aware software engineering. Some of these activities are extensions or modifications of conventional software engineering practices; others are new activities that only exist when energy efficiency is a design goal. Figure 1 shows a number of activities and (some of) the inter-dependencies that arise in the context of different scenarios.

3.1.1 Specify application, including energy

The process of developing application software starts with a requirements specification that expresses not only *functional* properties, as in the classical approach, but also *non-functional* properties, including energy consumption and other resources. Classical methods for requirements specification need to be extended to allow non-functional specifications to be expressed.

Satisfying functional properties (in the sense of the classical concept of correctness with respect to a test suite or a formal input-output specification) is as important as doing so for non-functional properties: an application that makes a device run out of batteries before a task is completed is as erroneous and useless as an application that does not compute the right result.

3.1.2 Construction of energy models

Creating energy models for different combinations of hardware platforms and programming languages is a part of the energy-aware development process. At one end of the spectrum, one might expect future hardware manufacturers to deliver an energy model for their instruction set architecture and thus the model would be available "off the shelf". At the other end, some projects might require the construction of an energy model specific to that project, perhaps because the hardware or software environment was not standard. In between these two extremes, energy modelling for energy-aware software development is becoming a more well understood process.

3.1.3 Resource model of deployment platform

If energy efficiency is a design goal, we need to obtain an *energy model of the platform* on which the system is to be deployed (even though the software might be developed on a different platform).

Thus obtaining the appropriate energy model is a vital task in energy-aware software engineering. Not only should an appropriate platform be selected, but its energy model should be available during software development to support other activities (see for example Sections 3.1.6, 3.1.7, 3.1.8 and 3.1.9). We note also that several different energy models for a given platform might be selected, at different levels of abstraction suitable for different activities. For instance, high-level approximate models might be suitable for design space exploration (Section 3.1.6) and initial energy profiling (Section 3.1.7), while more precise low-level models are needed for detailed energy analysis (Section 3.1.8) and optimisation (Section 3.1.10).

3.1.4 Selection of deployment platform

The choice of deployment platform itself might depend on its resource-usage model; thus this activity and Section 3.1.3 are interdependent. By "platform" here is meant both the hardware and the software platform; thus the model should be capable of predicting the energy usage of software (in a given language and with a given runtime environment) being executed on a given piece of hardware.

3.1.5 Configure platform

Some platforms allow configuration that can have implications for energy consumption. Among such settings are clock frequency and voltage, the number of cores and the communication paths among them. At the software level, operating system settings can also be considered, such as the settings for power saving and the resolution of OS timer processes that can send interrupts to other processes.

3.1.6 Design space exploration

Choices taken early on in the design process can have a profound effect on the energy efficiency of the final result. *Design space exploration* as an energy-aware software development activity refers to the process of estimating energy implications of different possible design solutions, before they are implemented. It may involve especially activities such as Selection of deployment platform (Section 3.1.4), Platform configuration (Section 3.1.5) and Initial energy profiling (Section 3.1.7). This involves energy modelling and analysis tools as in some other activities, but with the difference that one is likely to be more satisfied with approximate models and thus rougher estimates of energy consumption rather than precise predictions.

3.1.7 Initial energy profiling

At early stages of energy aware software design and implementation, tools are needed to perform an *initial energy analysis*. The purpose of this is to produce statically an *energy profile* that identifies the overall complexity of the energy consumption of the software and how energy consumption is distributed over the parts of the program. It could also at this stage identify energy bugs (parts of the application software that do not meet their energy consumption specification).

Initial energy analysis requires an an *energy model* of the deployment platform at an appropriate level of abstraction. At early stages, parts of the software may be missing and it might not be possible to compile it to machine instructions; thus an approximate model based on a model of source code might have to suffice.

3.1.8 Detailed energy analysis

During more advanced stages of energy aware software implementation, detailed energy analyses at finer levels of granularity are needed. These are provided by tools containing more precise low-level energy models of the platform, able to give precise estimates of the energy consumption of critical parts of the code, which could be targets for energy optimisation.

3.1.9 Identify energy bugs

Energy bugs occur when software does not conform to an energy specification. The specification might state some overall resource requirement in which energy consumption is implicit, for example on the length of battery life. The bug in such a case could be some energy-consuming process that is more expensive than necessary, a service that is not switched off when required, threads that synchronise badly and spend too much time waiting, and so on.

3.1.10 Energy optimisation or reconfiguring

The broad concept of energy optimisation is applied throughout the whole software engineering process, and starts right at the beginning with design space exploration and selection of appropriate platform, algorithms and data structures.

The specific energy optimisation performed in this activity is driven by the detailed energy analysis and the energy model of the platform. Both manual and automatic optimisations can be applied; the energy analysis should point to the sections of code that use the most energy, either because they involve costly energy operations, or because they are frequently executed (e.g., tight inner loops). This activity also includes application of energy-optimising compilers and such generic optimisers.

3.1.11 Verify or certify energy consumption

Energy-critical applications need to be certified with respect to an energy specification. Tools combining detailed energy models and precise energy analysis are required in order to compare the inferred energy consumption with the specification, either verifying conformance or certifying that it holds within some specified limits of behaviour such as input ranges.

3.2 Energy-aware software engineering scenarios

In this section we sketch scenarios in which the activities described in the previous section are applied.

3.2.1 Embedded system development on xCORE

The ENTRA project case studies focussed on embedded systems implemented in the XC language and deployed on the xCORE multicore architecture. An energy-aware software development strategy for such applications might involve the following energy-aware activities.

- Energy specification by writing pragma comments in the XC source code. Such pragmas could express energy constraints derived from customer requirements on the power supply.

- Platform selection and configuration. The xCORE architecture is highly configurable both in terms of the number of cores and their interconnection. The choice and configuration is guided by an energy model applied to proposed solutions, taking into account thread communication energy costs in a given configuration, as described in more detail in Attachment D1.2.2 (Swallow: Building an Energy-Transparent Many-Core Embedded Real-Time System [HK16]).

- Detailed program-independent energy models of the platform at ISA level, are available. Program-dependent energy models are obtained for XC and LLVM IR code for the application from the ISA model, and used to perform more precise and detailed energy analysis of the application.

- Optimisations of expensive or frequently executed code is performed on the basis of the energy analysis.

- The energy optimising compiler for XC is applied to the application.

- Pragmas in the code are verified using comparison of the energy consumption predicted by the analysis with the constraints in the specification.

3.2.2 Android app development

A case study on Android app energy optimisation was carried out [LG16] (see Attachment D4.2.7 in Deliverable D4.2). The study involved energy modelling and optimisation of applications based on an established game-engine. An energy specification was not given; the aim of the study was to use a source-code-level energy model to identify the most energy-intensive parts of the code in a number of typical use-cases, and then apply manual optimisations, reducing energy usage directly and thus prolonging battery life.

Energy-aware software engineering activities included:

- Building a fine-grained source code energy model by regression analysis from energy measurements on the target hardware and Android software platform of a set of test cases exercising the functions of the underlying game engine.
- Dynamic profiling of the code, which provided an energy profile that allowed the most energy-expensive basic blocks to be identified.
- Manual refactoring of the source code, targeted at the most expensive blocks, which succeeded in increasing energy efficient by a factor of 6% to 50% in various use-case scenarios.

4 The ENTRA tools front end

In order to clarify and demonstrate some of the functionality required to support the energy-aware software development activities described in Section 3.1, we implemented a Java front-end that provides a graphical user interface (the ENTRA tools front end) for some of the prototype tools developed in the ENTRA project. The use of Java allowed tools running on different platforms and operating systems to be provided with a common interface, and prototype tools to be rapidly integrated into the common interface to emphasise their functionality. Note that the ENTRA tools front end does not provide a suitable interface for all the tools, particularly those that are already more closely integrated in the XMOS tool-chain.

A secondary goal of the ENTRA tools front end is to give hints on how the ENTRA prototype tools and components for analysis, verification and optimisation could be integrated into an energy-aware software development tool-chain. The ENTRA tools front end contains tabs corresponding to different tool functions; these can be manually combined to demonstrate tool-chains providing the required functionality supporting energy-aware software engineering activities and scenarios (Section 3).

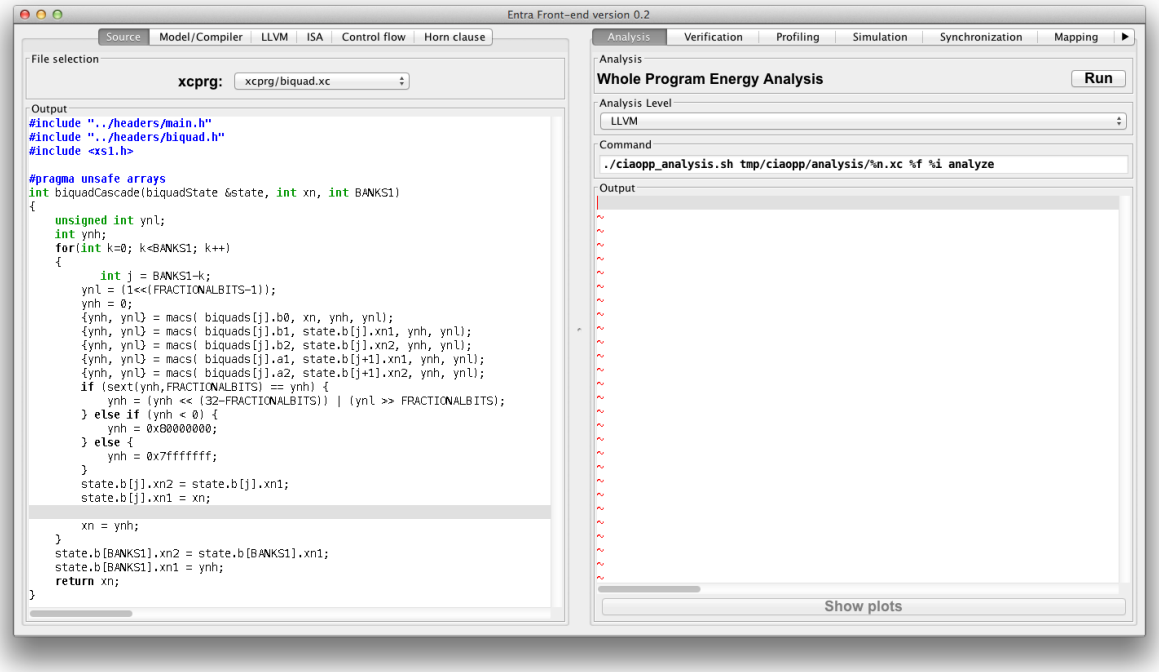


Figure 2: The ENTRA tools front end.

Figure 2 shows a screenshot of the ENTRA tools front end, which has two main sections on

the left and the right. The source file is loaded into the buffer on the left section under the *Source* tab. For demo purposes the user can load one of the benchmarks from the *xcprog* drop down.

The tabs *Analysis* and *Verification* (within the section on the right) are merely indicative of a wide range of functions to which the front end can be adapted. In the two example tabs shown, the user can select the *Analysis Level* to be either *LLVM* or *ISA*, which specify the level (LLVM IR or ISA) to which the XC source is compiled and then analyzed by the underlying tool (e.g., CiaoPP). Once the user presses the button *Run*, the output of the analysis/verification is loaded into the buffer on the right. The underlying command used to invoke the required functionality (e.g., analysis, verification) is also shown under *Command*. The button *Show plots* allows the user to see the output plotted graphically. It is shown using the gnuplot utility.

A short summary of the procedure for integrating a new tool into the ENTRA tools front end is given in Appendix A. The underlying assumption for integration is that the tool with its parameters can be run from the command line. The command line string is incorporated into the Java code and the input and output directed to the appropriate areas of the ENTRA tools front end.

5 Energy-aware Software Development Tools: Description and Demos

In this section we describe the final versions of the energy-aware prototype tools (components) for analysis, modelling, verification and optimization developed in the project, that can be integrated to build tool-chains providing the required functionality for the activities and scenarios described in Section 3.

5.1 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.

Main functionality. The tool can 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), LLVM Intermediate Representation (LLVM IR), 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, the tool 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. The tool also generates graphics of the energy consumption functions involved in the process and the input values for which energy specifications are met, which facilitates the interpretation of the analysis and verification results.

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+14]) and the other one performs the analysis at the LLVM IR level [LGK+16] (included in this document as Attachment D1.2.1).

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 [LGK+16] (Attachment D1.2.1).

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.

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. Functional components from the CiaoPP system can potentially be integrated with a compiler tool-chain, although it may require an implementation effort to make it robust to the real life environment.

5.1.1 Usage and interface

The user can interact with the tool through a Graphical User Interface (GUI) or through a Unix command line. The GUI has mainly used in the project to facilitate experimentation by tool developers and to experiment with aspects of user interaction such as source assertions. We focus here on the command line interface, which has been used for integration in the ENTRA tools front end, in order to demonstrate that it can also be integrated in other tool-chains as well.

The command line interface can be used to perform analysis (and other actions) on XC, LLVM IR and ISA files. 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. The format of this input file is determined by the file name extension (e.g., XC, ISA or HC IR files are recognized by their respective extensions `.xc`, `.asm`, `.ll`, or `.pl`.). – 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.
- The `--verify` option is used to perform the resource usage *verification* of the input file, by checking the assertions present in the input file. In this case an analysis is first performed – as it would be the case if the option were `--analyze` – then the results are compared with the input assertions to be checked.

  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 is useful for generating 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. For this, the user can set the option to two different values: `LLVM` or `ISA` respectively.

- The `--req-solver=<solver>` option specifies which recurrence equation solver must be used by the resource analysis engine. Currently, the user can set the option to three 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 [Mat]. 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 `chain` value forces to use the *chain* strategy from the modular solver implemented in CiaoPP, which basically tries to solve a recurrence relation by calling in sequence each available back-end solver. These back-end solvers are `mathematica`, `builtin`, or a specialized solver for recurrences with increasing arguments (that uses `ppl`). In any case, the first solution found is the one that is returned, obtained from one of the back-end solvers.

- The `--math-system=<cas>` option specifies which algebraic system must be used by the resource analysis engine to solve operations like simplification of expressions, variable isolation, expression comparison, etc. Currently, the user can set the option to `mathematica` or `builtin` (default value).

- The `--res-analysis=<analysis>` option specifies which analyzer must be used for perform the resource analysis. Currently the options are:

  - `resources`, which forces to use the legacy version of the resource analysis present in CiaoPP.
  - `res_plai`, indicating that the CiaoPP's abstract interpretation-based resource analysis must be used (default value).

- The `--help` option displays description of the command line usage including the different options described above.

5.2 Multi-level mapper tool

In deliverable D3.1, we introduced a novel mapping technique to lift our ISA-level energy model to a higher level, the intermediate representation of the compiler, namely LLVM IR [LA04], implemented within the LLVM tool chain [LLV14]. In deliverable D1.1 under Section 3.2, the mapping tool was introduced, together with a use case example. The mapping techniques implemented by the tool, were evaluated further on both single- and multi-threaded benchmarks, and the results are reported in deliverable D2.3 under Section 2. The evaluation was done using Energy Consumption Static Analysis (ECSA) based on an Implicit Path Enumeration Technique (IPET) [LM95], which we introduced in deliverable D1.1 under Section 4, *Work in Progress*. Our results show that the mapping technique allowed for energy consumption transparency at the LLVM IR level, with accuracy keeping within 1% of ISA-level estimations in most cases. The mapper tool makes energy consumption information accessible directly to the optimizer, and therefore creates new opportunities towards energy specific compiler optimizations.

A paper detailing the mapping techniques together with their evaluation, the ECSA static analysis used and ECSA practical applications at both the ISA and the LLVM IR levels, is attached as D2.3.3 to D2.3.

5.3 Performing parametric static profiling of energy consumption

The standard or classical notion of cost (given in terms of different resources, e.g., energy or execution time) inferred by the ENTRA analysers only partially fulfils typical requirements of some energy-aware software development activities. For example, the software developer may want to know which parts of a program are the most resource-consuming and which procedures or functions should be optimised first, because of their greater impact on the overall energy consumption of the main program. Procedures/functions with the highest (standard) costs may not need to be optimised first, but perhaps, procedures/functions with lower costs but which are called many times. In this context, what is really needed, is information about how much of the total cost of a program is each procedure/function responsible for, i.e., the *distribution* of energy consumption over the parts of the program. Such information is provided by the parametric static profiling tool described in [HLGL$^+$16] (also included in this document as Attachment D1.2.3).

The tool analyses a program and produces energy consumption functions giving the *accumulated cost* in selected parts of the program (named *cost centers*) as a function of input data sizes.

Such parameterised information allows to know how the distribution of energy consumption grows depending on the variation of the input, unlike the non-parametric information provided by dynamic profilers. Moreover, the information inferred by the static profiler tool is valid for all input values to the program, in contrast to the information provided by dynamic profilers, which is only valid for particular inputs and execution traces, and hence, may give an incomplete view about the distribution of the energy consumption in the program.

The (accumulated cost) information provided by our static profiling tool can be a more valuable aid for resource-aware software development than standard/classical resource usage analysis as it helps identify parts that should be optimised first. It can provide a ranking of the procedures of the program according to its accumulated cost to guide program optimisation. In addition, such accumulated cost information can be used in combination with functions indicating the number of times each procedure is called (which depend on input data sizes to the main program). These functions can be inferred by specialising the general resource analysis developed in WP3 by defining explicitly a *resource* for the number of calls performed by each cost center procedure. A big complexity order in the number of calls to a procedure (in relation to that of a single call) might give hints to reduce the number of calls to such procedure in order to effectively reduce its impact on the overall energy of the program.

Other situation where the static profiler is useful is when the overall resource complexity of a program might not be obtainable. For instance, some parts of it might be too complex for analysis or else because the code for some parts is not available and the cost cannot even be reasonably estimated. In this case useful information could still be obtained by excluding such parts from the analysis, obtaining information about the resource usage for the rest of the program. We refer to deliverable D3.3 for other aspects of the static profiling tool, such as an overview of the technique behind the tool and other motivations to develop it.

5.4 The Swallow platform

Swallow is an experimental many-core system consisting of XMOS XS1 processors, intended for use as a research tool and to aid the development and testing of software tools and programming techniques in an energy-aware, multi-core context. The system is composed of *slices*, where each slice is a board with sixteen XS1-L cores as a grid arrangement of dual-core chips. Slices can be arranged into a larger grid-like structure, allowing hundreds of cores to be used in a single system.

Although this is a significant leap in the number of cores examined, and goes beyond the limits of the chip vendors development tools, the Swallow system has proved useful in the collection of data for ENTRA related tools. In particular, the rich connectivity of Swallow has aided the

construction of the multi-core model parameters, particularly with respect to the communication costs.

Attachment D1.2.2 is a paper detailing the Swallow platform, including the energy consumption characteristics. This paper will appear at the Design Automation and Test in Europe (DATE) conference 2016 in March. Data from this work has been used in tools from ENTRA, in combination with modelling techniques described in attachment D2.3.1 of Deliverable D2.3.

5.5 Optimization via Dynamic Voltage and Frequency Scaling (DVFS) and task scheduling

In deliverable D1.1, we introduced an optimization tool that we have developed for solving a general problem of optimal task scheduling in XMOS chips. These XMOS chips are DVFS-enabled multicore systems able to execute multiple threads per core. The set of tasks to be scheduled is represented by its release time, deadline, and estimated power consumption (if available). The tool schedules the tasks in order to find an optimal task-core (thread) assignment, so that all the deadlines are met and the energy consumption is minimised. Based on the way the execution time and power consumption are estimated, a custom genetic algorithm based scheduler is implemented for both stochastic and deterministic scheduling [BLG15]. In addition, we have adapted the well-known YDS algorithm [YDS95], initially designed for DVFS-enabled single core, to a multicore environment for the initial assignment of tasks to cores.

The evaluation of the tool have also been presented in D1.1 under Section 3.4.4 using synthetic data and/or typical power consumption of XMOS chips. The results obtained (reported in [BLG13, BLG15]) confirm the potential of energy-aware scheduling coupled with DVFS for optimising energy consumption.

Further advancements have been reported in [**?**] and [**?**]. In [**?**], an approach to scheduling problems based on a custom evolutionary algorithm (EA) is described. The algorithm is fed with information provided by the CiaoPP static analyzer about predictions of the energy consumed by tasks. The speed ups gained using static analysis predictions solve the time inefficiency problem faced by EAs. In cases when our custom EA fails to produce a feasible solution, our approach resort to a modified YDS algorithm, which is an adaptation to multicore environments and to situations when the static power becomes the predominant part. This combined approach (custom EA algorithm with modified YDS) produces an energy efficient scheduling in reasonable time, that always finds a viable solution. Our approach has been tested on multicore XMOS chips in different scenarios, and the experimental results show that the modified YDS algorithm improves the original one up to 20%, while the custom EA can save 55 - 90% more energy on average than the modified YDS.

In [**?**], a trade-off between accuracy and energy is studied for the problem of energy efficient scheduling and allocation of tasks in multicore environments, where the tasks can permit certain loss in accuracy of either final or intermediate results while still providing proper functionality. Such situations allow the application of techniques that decrease the computational load, which result in significant energy savings but also in certain accuracy loss. In particular, we have applied the loop perforation [**?**] technique that transforms loops to execute a subset of their iterations. The experiments conducted on a case study in different scenarios show that our new scheduler enhanced with loop perforation improves the previous one, achieving significant energy savings (31% on average) for acceptable levels of accuracy loss.

5.6 XMOS tools: Supporting low power design in XC

The XMOS tools 14.1 include a global optimiser and an implementation of interfaces; the latter (described in deliverable D4.2) to enable modular development of software in an energy efficient matter, the former, described in this section, to generally increase efficiency of applications.

The XMOS programming environment, centered around the XC programming language, requires a program to be split into tasks that communicate over channels.

Although this model can naturally support low energy programs by enabling the programmer to spread their load over devices that run at similar low frequencies, the process of splitting tasks and balancing them is one that can be at odds with normal software engineering practices, where software is split in modules that are based on their function, not on their load balancing properties.

To this effect, we have developed an extension to the XC language, the *interface*, that enables software to be split functionally, yet still achieve load balancing as is needed to develop a low-energy design.

5.6.1 Introduction to the interface

The idea behind the interface is to merge the traditional concept of a software module in an imperative language, with the concept of threads communicating over channels.

When programming using an interface, there are at least two actors: a *server* and one or more *clients*. The server implements the interface, and the client makes calls to the interface. Notionally, the server runs in a separate thread to the client, and a channel connects the two. This model is similar to that of an RPC (Remote Procedure Call), or to forms of distributed Object Oriented programming models.

An example interface is shown below:

```
typedef enum { I2C_NACK, I2C_ACK  } i2c_res_t;

typedef interface i2c_master_if {
  i2c_res_t write(uint8_t device_addr, uint8_t buf[n], size_t n,
                  size_t &num_bytes_sent, int send_stop_bit);
  i2c_res_t read(uint8_t device_addr, uint8_t buf[n], size_t n,
                 int send_stop_bit);
  void send_stop_bit(void);
  void shutdown(void);
} i2c_master_if;
```

This definition states that there are four calls that a client can make to a server: `write`, `read`, `send_stop_bit`, and `shutdown`. The server side of the interface implements those four calls, and the client side to the interface can make those calls. The calls themselves happen, under the bonnet, over channels.

For a server with a single client the server side of the interface is implemented as follows:

```
void i2c_master(server interface i2c_master_if c,
                port p_scl, port p_sda, unsigned kb_per_sec) {
    unsigned bit_time = (XS1_TIMER_MHZ * 1000) / kb_per_sec;
    p_scl :> void;
    p_sda :> void;
    while (1) {
        select {
        ...
        case c.read(uint8_t device, uint8_t buf[m], size_t m,
                    int send_stop_bit) -> i2c_res_t result:
            ...
            result = (ack == 0) ? I2C_ACK : I2C_NACK;
            break;

        case c.send_stop_bit(void):
            ...
            break;
        ...
        }
    }
```

```
}
```

The client can make calls `i2c_server.send_stop_bit()` akin to an object oriented programming model. Multiple clients are also supported by using an array of channels. In this particular example, a client can choose to make an atomic sequence of calls, which leads to a rather complex case statement:

```
void i2c_master(server interface i2c_master_if c[n], size_t n,
                port p_scl, port p_sda, unsigned kb_per_sec) {
    unsigned bit_time = (XS1_TIMER_MHZ * 1000) / kb_per_sec;
    unsigned locked_client = -1;
    p_scl :> void;
    p_sda :> void;
    while (1) {
        select {
        case (size_t i =0; i < n; i++)
          (n==1 || locked_client == -1 || i == locked_client) =>
            c[i].read(uint8_t device, uint8_t buf[m], size_t m,
                      int send_stop_bit) -> i2c_res_t result:
            ...
            locked_client = send_stop_bit ? -1 : i;
            result = (ack == 0) ? I2C_ACK : I2C_NACK;
            break;

        case c[int i].send_stop_bit(void):
            ...
            locked_client = -1;
            break;
        ...
        }
    }
}
```

Here, `locked_client` is a variable that stores the client that is currently in an atomic sequence of reads and or writes. There are two ways that the sequence can end: by a call to `read()` with `send_stop_bit` set, or by a call to `send_stop_bit()`.

Analysing the first case statement reveals how the multiple interfaces work:

```
case (size_t i =0; i < n; i++)
  (n==1 || locked_client == -1 || i == locked_client) =>
    c[i].read(...
```

That is short-hand for a series of `n` case statements on `c[0]...c[n-1]` with each case having a guard that the case statement can only occur if at least one of three conditions is met:

1. `n==1` (there is only one client - nobody else could have started an atomic sequence that is being interrupted) or
2. `locked_client == -1` (no atomic sequence has started) or
3. `i == locked_client` (it is this client who is running an atomic sequence)

This description is hence a complete generic description, that works for any number of clients needing access to this single interface.

If there is only a single client, it may seem that the former implementation of the server is more efficient; however, the global optimiser (developed in WP4), actually makes the implementations as efficient. If only a single client is used, then the call to `i2c_master` will use the value `1` for `n`, and as there will only be one call the function `i2c_master` will be specialised for the case where `n` equals `1`. This specialisation will throw away all the guards (as the condition `n==1` evaluates to true), and will subsequently throw away `locked_client` since it is no longer used.

5.6.2 Combinable

One common problem is that often tasks can be described as individual threads, but implementing them as a single thread is energy inefficient, as the thread will be under-utilised: best energy efficiency is obtained by balancing all threads.

For this purpose, we have defined the `[[combinable]]` attribute. Functions marked `[[combinable]]` can be merged together and implemented in a single thread by the toolchain. When a function is marked as `[[combinable]]` it must be implemented using the following template:

```
[[combinable]]
void ...(server interface c, ...) {
    ...
    while (1) {
        select {
```

```
        case c... :
            ...
            break;
        ...
        }
    }
}
```

The programmer using the module can decide on whether and how to combine different combinable servers, by placing all the servers in a single thread. This loads that one thread more heavily, but reduces the total number of threads required, reducing energy load.

5.6.3 Distributable

The opposite of a `[[combinable]]` interface is a `[[distributable]]` interface; that is an interface where the server is so simple that it can be assimilated in the client(s) side. This is another way to balance load and thereby reduce energy.

5.6.4 Status and use

Interfaces have been released during the project in version 13 of the tools; and subsequent optimisations in version 14 of the tools.

Where appropriate, software libraries have been rewritten to make use of interfaces, simplifying modularisation and software development.

Compared to the pre-interface status; modularised software engineering principles can now be applied to a programming model that supports energy efficient design.

5.7 Implicit path enumeration ECSA applications

In this section we provide a set of ECSA applications. Software developers, compiler engineers, development tools and RTOS can get advantage of these applications for making energy aware decisions. ECSA using the Implicit Path Enumeration Technique (IPET) was introduced in D1.1 under section 4.1 and used together with the high level energy model as described in D2.3 under Section 2. A paper detailing the ECSA techniques together with its practical applications is attached as D2.3.3 to D2.3.

5.7.1 Design-space exploration for multi-threaded programs using ECSA

ECSA is applied to a set of multi-threaded programs for the first time to our knowledge. This is a significant step beyond existing work that examines single-thread programs, because such an analysis can provide significant guidance for time-energy design space exploration between different numbers of threads and cores.

The first class of parallel programs to which ECSA was applied is the class of replicated non-communicating threads. The user can make energy aware decisions on the number of threads to use, with respect to time and energy estimations retrieved by our analysis. For example, take four independent matrix multiplications on four pairs of equally sized matrices (28×28). Our analysis will show that a single thread will have an execution time of 4x the time needed to execute one matrix multiplication. However, two threads will half the execution time and decrease the energy by 54%. Four threads which will half the execution time again, and decrease the energy by 41% compared to the two-thread version. Using more threads increases the power dissipation, but the reduction in execution time saves energy on the platform under investigation. Although there is a different estimation error between different numbers of active threads, the error range of 6% is small enough to allow comparison between these different versions.

The second class of parallel programs that our ECSA was applied to was streaming pipelines of communicating threads. There is a choice in how to spread the computation across threads to maximize throughput and therefore minimize execution time or lower the necessary device operating frequency. Having a number of available threads, a number of cores and the ability to apply voltage and frequency scaling, provides a wide range of configuration options in the design phase, with multiple optimization targets. This can range from optimizing for quality of service, time and energy, or a combination of all three. Our ECSA can take advantage of the fact that the energy model used can be parametric to voltage and frequency, to statically identify the most energy efficient configuration of the same program, among a number of different options that deliver the same required performance. The first step of analyzing the pipelined versions of industrial filter applications has been made. We are currently working on extending our ECSA to automatically exploit the possible different configurations and provide the optimal solution, within the user's constraints.

5.7.2 Other ECSA applications

Figure 3 shows the ISA energy consumption upper and lower bounds retrieved by ECSA for the `Radix4Div` and `B.Radix4Div` benchmarks. `Radix4Div` benchmark is a radix-4 software divider and `B.Radix4Div` is a less efficient version which is added for comparison. This version omits an early return when the dividend is greater than 255. A consequence of excluding this

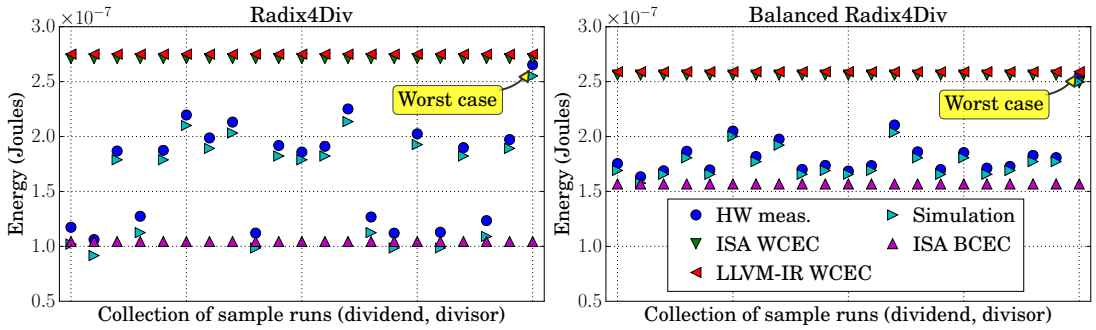


Figure 3: Radix4 division benchmark ECSA estimations across all test cases.

optimization is that CFG paths become more balanced, with less variation between the possible execution paths. The effect of this modification can be seen in Figure 3. In the optimized version, the energy consumption across different test cases varies significantly, creating a large range between the upper and lower energy consumption bounds. Conversely, the unoptimized version shows a lower variation, thus narrowing the margin between the upper and lower bounds, but has a higher average energy consumption. Knowledge of such energy consumption behavior can be of value for applications like cryptography, where the power profile of systems can be monitored to reveal sensitive information in side channel attacks. In these situations, ECSA analysis can help code developers to design code with low energy consumption variation, so that any potential leak of information that could be obtained from power monitoring can be obfuscated.

5.8 Tools for Horn clause verification

Constrained Horn clauses (CHCs) are intermediate representations suitable for expressing the semantics of a variety of programming languages and computational models. As a result, they have become a popular formalism for verification [BGMR15, GK14]; attracting both the logic programming and software verification communities [BFRS14]. Several verification techniques and tools have been developed for CHCs, among others, SeaHorn [GKKN15], QARMC [GLPR12], VeriMap [DAFPP14], Convex polyhedral analyser [KG15], TRACER [JMNS12], ELDARICA [HKG+12], μZ [HBdM11] and Trace abstraction refinement tool [WJ15].

The ENTRA project has adopted (constrained) Horn clauses as an internal representation capable of representing source code, LLVM IR and ISA. The CiaoPP system incorporates Horn clause analysis tools, including a generic framework for resource analysis (see Section 5.1). The tools described in this section are development intended to extend and strengthen these tools. We present them here as stand-along tools operating on Horn clauses; in their intended application

the Horn clauses are derived from the application which is under (energy-aware) development.

State-of-the-art Horn clause verification tools verify functional properties of programs (properties relating program variables); they can also be used to verify non-functional properties like *energy* if we instrument programs (clauses) with energy counters.

Energy-instrumented clauses Let P be a set of CHCs and P_{en} be a set of CHC constructed as follows.

- For each predicate p of arity m define a predicate p' of arity $m+1$.
- For each clause in P of the form

  $$p(X) \leftarrow \phi, p_1(X_1), \ldots, p_n(X_n)$$

  we have a clause

  $$p'(X, K) \leftarrow \phi, p_1^{'}(X_1, K_1), \ldots, p_n^{'}(X_n, K_n), K = K_1 + \ldots + K_n$$

  in P_{en}, where $K_1, \ldots, K_n, K$ are energy variables added as the final argument for their respective predicates.

Energy forms a part of a program with program instrumentation, which allows the use of above mentioned tools to verify properties relating program's input variables (equivalently other variables) with its energy consumption.

During the ENTRA project, we have developed two Horn clause verification tools (for verifying functional properties, and thus energy properties via instrumentation), namely,

- `RAHFT`[9];
- `LHornSolver`[10].

Both of these tools are based on the *abstraction-refinement* scheme, but the second one only uses a linear Horn clause solver for solving non-linear Horn clauses, potentially allowed greater scalability. These tools are also able to produce witnesses showing that a property is satisfied or violated, in contrast to other resource analysis tools in the literature, including CiaoPP. Properties are represented as integrity constraints on the Horn clauses, which are easily expressed as assertions in the CiaoPP assertion language.

[9] available from `https://github.com/bishoksan/RAHFT`

[10] available from `https://github.com/bishoksan/LHornSolver`

Usage. The tools are all command-line runnable, taking a Horn clause file as input and generating various kinds of output file, containing invariants for each predicate, checks on the satisfaction of integrity constraints and counterexamples if such constraints are violated.

The tools use as backends powerful solvers such as the Parma Polyhedra Library [BHZ08] and the Yices SMT solver [Dut14], which need to be installed to run the tools.

5.9 Integrating energy modelling into the development process: A Makefile approach

This tool serves to demonstrate how the Instruction Set Simulation (ISS) modelling tools, and associated energy model that underpins much of the work used in ENTRA, can be integrated directly into the software development process.

The output of the research effort is described in more detail in Deliverable D2.3 and its predecessor, D2.2. In this deliverable, attachment D1.2.4 is a technical brief that gives a demonstration of the integration of the tools used in these works. The motivation behind the approach of the tool, is that the vendor's existing tools are Makefile based, therefore it is intuitive in enable energy modelling as part of this process.

A number of software components are required, some of which were developed during the project, whilst others are supporting libraries. These are listed in attachment D1.2.4. An example, based on the FIR filter code from work package 5, is used in the technical brief. The tool is principally command-line based, but also has the facility to display modelling results graphically. In the example, this is done via the default PDF viewer of the user's desktop environment, although this could be integrated into any other graphical tool.

The code for the tools has been provided to the project, so that it can be made available in an appropriate manner.

References

[BFRS14] Nikolaj Bjørner, Fabio Fioravanti, Andrey Rybalchenko, and Valerio Senni, editors. *Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014*, volume 169 of *EPTCS*, 2014.

[BGMR15] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015.

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

[BLG13] Z. Banković and P. Lopez-Garcia. Genetic Algorithm-based Allocation and Scheduling for Voltage and Frequency Scalable XMOS Chips. In *Hybrid Artificial Intelligent Systems (HAIS 2013)*, volume 8073 of *Lecture Notes in Computer Science*, pages 401–410. Springer, 2013.

[BLG15] Zorana Banković and Pedro Lopez-Garcia. Stochastic vs. Deterministic Evolutionary Algorithm-based Allocation and Scheduling for XMOS Chips. *Neurocomputing*, 150:82–89, February 2015.

[CFS12] Eugenio Capra, Chiara Francalanci, and Sandra Slaughter. Is software "green"? application development environments and energy efficiency in open source applications. *Information & Software Technology*, 54(1):60–71, 2012.

[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.

[Dew14] Robert Dewor. Energy efficiency of web applications. Master's thesis, Faculty of Economics, 2014.

[dSCM+12] Clauirton de Siebra, Paulo Costa, Regina C. G. Miranda, Fabio Q. B. da Silva, and André Luis M. Santos. The software perspective for energy-efficient mobile applications development. In Eric Pardede and David Taniar, editors, *The 10th International Conference on Advances in Mobile Computing & Multimedia, MoMM '12, Bali, Indonesia - December 03 - 05, 2012*, pages 143–150. ACM, 2012.

[Dut14] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[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.

[Gal14] J.P. Gallagher, editor. *Energy Optimization: Basic Static Techniques*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), August 2014. Deliverable 4.1, http://entraproject.eu.

[GK14] John P. Gallagher and Bishoksan Kafle. Analysis and transformation tools for constrained Horn clause verification. *TPLP*, 14(4-5 (additional materials in online edition)):90–101, 2014.

[GKKN15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[GLPR12] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 405–416. ACM, 2012.

[Goe13] Fethullah Goekkus. Energy efficient programming: an overview of problems, solutions and methodologies. Technical report, University of Zurich, 2013.

[HBdM11] Krystof Hoder, Nikolaj Bjørner, and Leonardo Mendonça de Moura. μZ- an efficient engine for fixed points with constraints. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 457–462. Springer, 2011.

[HK16] Simon J. Hollis and Steve Kerrison. Swallow: Building an Energy-Transparent Many-Core Embedded Real-Time System. In *2016 Design, Automation & Test in Europe (to appear)*. IEEE, March 2016.

[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.

[HLGL+16] R. Haemmerlé, P. Lopez-Garcia, U. Liqat, M. Klemen, J. P. Gallagher, and M. V. Hermenegildo. A Transformational Approach to Parametric Accumulated-cost Static Profiling. In *Thirteenth International Symposium on Functional and Logic Programming (FLOPS 2016)*, LNCS. Springer, 2016. To appear.

[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.

[KCWCW10] P.J. Krause, K. Craig-Wood, and N. Craig-Wood. Green ICT: Oxymoron, or call to innovation? In *Proc. Green IT*, Singapore, 2010.

[KG15] Bishoksan Kafle and John P Gallagher. Horn clause verification with convex polyhedral abstraction and tree automata-based refinement. *Computer Languages, Systems & Structures*, Nov. 2015. to appear.

[LA04] C. Lattner and V.S. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the 2004 International Symposium*

on Code Generation and Optimization (CGO), pages 75–88. IEEE Computer Society, March 2004.

[Lar11] Petter Larsson. Energy-efficient software guidelines. Technical report, Intel Software Solutions Group, 2011.

[LG16] Xueliang Li and John P. Gallagher. An energy-aware programming approach for mobile application development guided by a fine-grained energy model. Technical report, Roskilde University, February 2016. submitted for publication.

[LGK+16] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR. In M. Van Eekelen and U. Dal Lago, editors, *Foundational and Practical Aspects of Resource Analysis. Fourth International Workshop FOPARA 2015, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2016. To appear.

[LKS+14] 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 Gopal Gupta and Ricardo Pea, editors, *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 72–90. Springer, 2014.

[LLV14] LLVMorg. The LLVM Compiler Infrastructure, November 2014.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 88–98, 1995.

[MA13] Sara S. Mahmoud and Imtiaz Ahmad. A green model for sustainable software engineering. *International Journal of Software Engineering and Its Applications*, 7(4), July 2013.

[Mat] Mathematica. `http://www.wolfram.com/mathematica/`.

[NKD13] Stefan Naumann, Eva Kern, and Markus Dick. Classifying green software engineering - the GREENSOFT model. *Softwaretechnik-Trends*, 33(2), 2013.

[RJ97] Kaushik Roy and Mark C. Johnson. Software Design for Low Power. In Wolfgang Nebel and Jean P. Mermet, editors, *Low Power Design in Deep Submicron Electronics*, volume 337, pages 433–460. Kluwer Academic, 1997.

[SA12] B. Steigerwald and A. Agrawal. Green software. In San Murugesan and G. R. Gangadharan, editors, *Harnessing Green IT : Principles and Practices*, chapter 3. John Wiley & Sons, Hoboken, NJ, USA, 2012.

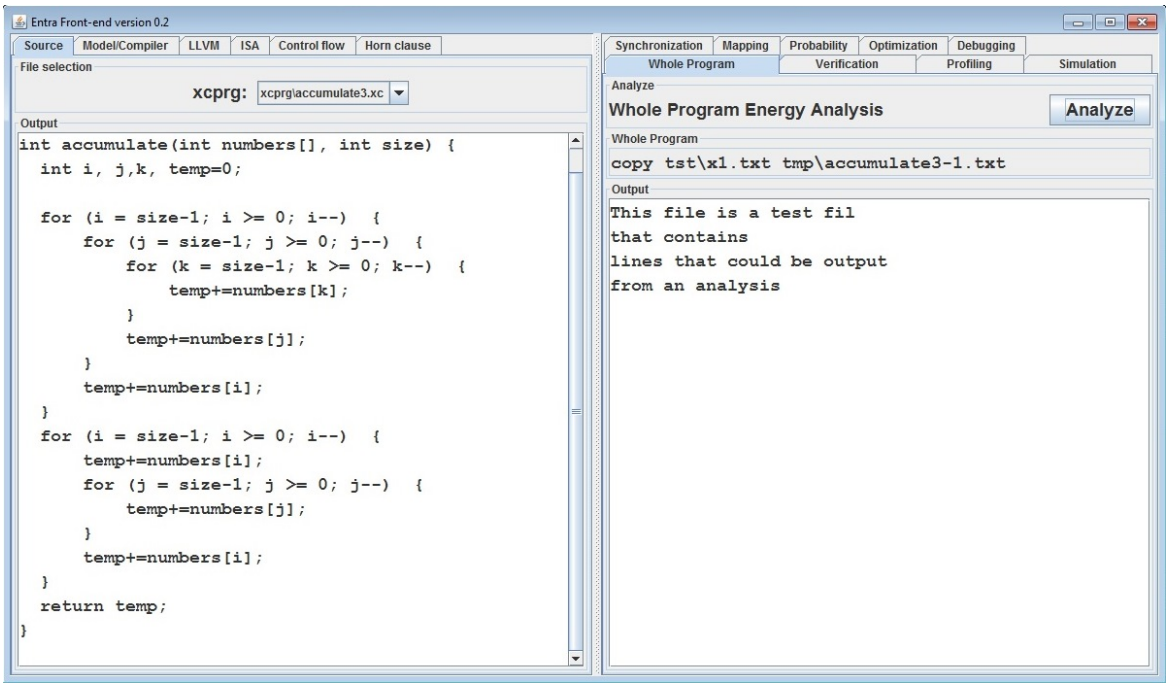
[WJ15] Weifeng Wang and Li Jiao. Trace abstraction refinement for solving horn clauses. Technical Report ISCAS-SKLCS-15-19, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Dec. 2015.

[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.

A ENTRA tools front end mini-manual

The ENTRA tools front end is meant as a front-end for command-line compiler and analysis tools. It is designed for analysing single file programs and for showing intermediate representations and analysis results from the program.

A.1 User interface



The left hand side contains tabs for the source program and intermediate representations. The right hand side contains output from a collection of analysis. The system uses a fixed directory for xc-programs and from the source code panel you can select a file. When you select a file the intermediate representations are generated by running a command line program. The analysis panels contain an **Analyze** button that can be used to call the command line program for the analysis. The second tab on the left hand side is a Model/Compiler which can be used to specify global parameters and to show the command lines for intermediate versions.

A.2 Analysis and intermediate version specification

The various intermediate representations and analysis tools are internally represented as **`Analysis`** objects.

```
Analysis(
  String shortName,    // short name for tabs
```

```
String longTitle,    // long title for the analysis
String inputText,    // in non-empty the panel analysis may take text input
String commandLine,  // the commandline that will be executed
String outputFile,   // name of file to display
Source source)       // reference to the source code
```

An analysis has a short name that is used in the tab, a long name which is used as title on the panel for the analysis. An analysis may have some extra input (a name of a variable, energy specification etc, and if the **inputText** parameter is non-empty the panel will contain a text field which can be passed to the command line program. The **commandLine** and **outputFile** is the command that is executed when you press the **Analyze** button and the file that will be shown in the panel. Both fields can reference the name of the source program file using parameter substitution:

`%f` is replaced by the source program file name (including path)

`%n` is replaced by the source program file name (excluding path and extension)

`%i` is replaced by the analysis input field

`%m` is replaced by input field on the "Model/Compiler" tab

A possible commandline could be

```
xc2ast.sh %f > tmp/%n_blocks.txt
```

and the **outputFile** would then be **tmp/%n_blocks.txt**

If the output file name ends with **.jpg** the output is shown as an image, otherwise it is expected to be a text file and shown as such.

A.3 Future versions

The Front-end is designed to be a "thin" version such that analysis tools and compilers are run as separate programs and should produce results that can be displayed by the front end. The input specification is fairly rudimentary and could be extended with sliders or list selection. If this is relevant for some analysis tools do pass on a request for it.

Attachments

Attachment D1.2.1

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

Accepted at the Foundational and Practical Aspects of Resource Analysis (FOPARA 2015)

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

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

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

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

Keywords: Energy consumption analysis, resource usage analysis, static analysis, embedded systems.

1 Introduction

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

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

The process of developing energy-efficient software can benefit greatly from static analyses that estimate the energy consumed by program executions without actually running them. Such estimations can be used for different software-development tasks, such as performing automatic optimizations, verifying energy-related specifications, and helping system developers to better understand the impact of their designs on energy consumption. These tasks often relate to the source code level. On the other hand, energy consumption analysis must typically be performed at lower levels in order to take into account the effect of compiler optimizations and to link to an energy model. Thus, the inference of energy consumption information for lower levels such as the Instruction Set Architecture (ISA) or intermediate compiler representations (such as LLVM IR [19]) is fundamental for two reasons: 1) It is an intermediate step that allows propagation of energy consumption information from such lower levels up to the source code level; and 2) it enables optimizations or other applications at the ISA and LLVM IR levels.

In this paper (an improved version of [20]) we propose a static analysis approach that infers energy consumption information at the ISA and LLVM IR levels, and reflects it up to the source code level. Such information is provided in the form of *functions on input data sizes*, and is expressed by means of *assertions* that are inserted in the program representation at each of these levels. The user (i.e., the "energy-efficient software developer") can customize the system by selecting the level at which the analysis will be performed (ISA or LLVM IR) and the level at which energy information will be output (ISA, LLVM IR or source code). As we will show later, the selection of analysis level has an impact on the analysis accuracy and on the class of programs that can be analyzed.

The main goal of this paper is to study the feasibility and practicability of the proposed analysis approach and perform an initial experimental assessment to shed light on the trade-offs implied by performing the analysis at the ISA or LLVM levels. In our experiments we focus on the energy analysis of programs written in XC [31] running on the XMOS XS1-L architecture. However, the concepts presented here are neither language nor architecture dependent and thus can be applied to the analysis of other programming languages (and associated lower level program representations) and architectures as well. XC is a high-level C-based programming language that includes extensions for concurrency, communication, input/output operations, and real-time behavior. In order to potentially support different programming languages and different program representations at different levels of compilation (e.g., LLVM IR and ISA) in the same analysis framework we differentiate between the *input language* (which can be XC source, LLVM IR, or ISA) and the *intermediate semantic program representation* that the resource analysis operates on. The latter is a series of connected code blocks, represented by Horn Clauses, that we will refer to as "HC

IR" from now on. We then propose a transformation from each *input language* into the HC IR and passing it to a resource analyzer. The HC IR representation as well as a transformation from LLVM IR into HC IR will be explained in Section 3. In our implementation we use an extension of the CiaoPP [12] resource analyzer. This analyzer always deals with the HC IR in the same way, independent of its origin, inferring energy consumption functions for all procedures in the HC IR program. The main reason for choosing Horn Clauses as the intermediate representation is that it offers a good number of features that make it very convenient for the analysis [22]. For instance, it supports naturally Static Single Assignment (SSA) and recursive forms, as will be explained later. In fact, there is a current trend favoring the use of Horn Clause programs as intermediate representations in analysis and verification tools [8, 15, 4, 3].

Although our experiments are based on single-threaded XC programs (which do not use pointers, since XC does not support them), our claim about the generality and feasibility of our proposed approach for static resource analysis is supported by existing tools based on the Horn Clause representation that can successfully deal with C source programs that exhibit interesting features such as the use of pointers, arrays, shared-memory, or concurrency in order to analyze and verify a wide range of properties [8, 15, 10]. For example [10] is a tool for the verification of safety properties of C programs which can reason about scalars and pointer addresses, as well as memory contents. It represents the bytecode corresponding to a C program by using (constraint) Horn clauses.

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

Our hypothesis is that the choice of level will affect the accuracy of the energy models and the precision of the analysis in opposite ways: energy models at lower levels (e.g. at the ISA level) will be more precise than at higher levels (e.g. XC source code), since the closer to the hardware, the easier it is to determine the effect of the execution on the hardware. However, at lower levels more program structure and data type/shape information is lost due to lower-level representations, and we expect a corresponding loss of analysis accuracy (without using complex techniques for recovering type information and abstracting memory operations). This hypothesis about the analysis/modelling level tradeoff (and potential choices) is illustrated in Figure 1. The possible choices are classified into two groups: those that analyze and model at the same level, and those that operate at different levels. For the latter, the problem is finding good mappings between software segments from the level at which the model is de-

fined up to the level at which the analysis is performed, in a way that does not lose accuracy in the energy information.

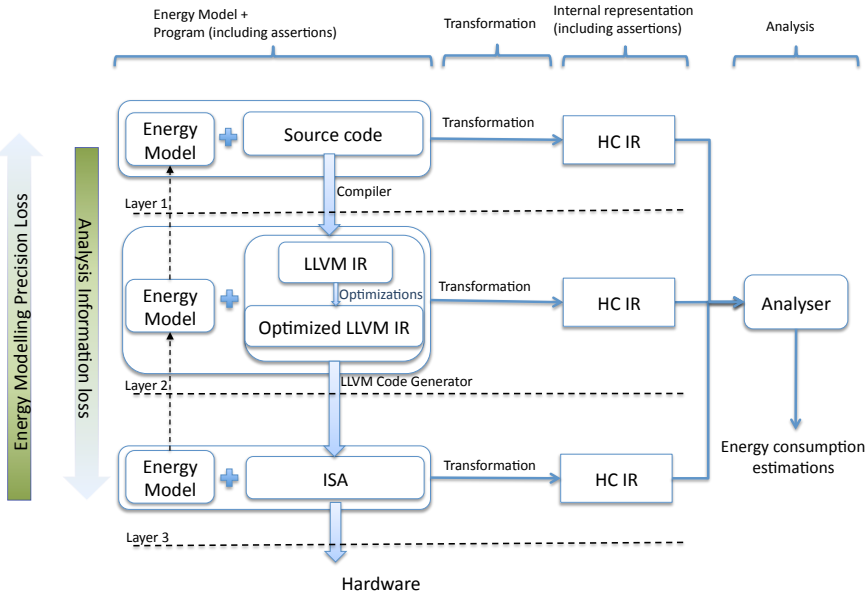


Fig. 1: Analysis/modelling level trade-off and potential choices.

In this paper we concentrate on two of these choices and their comparison, to see if our hypothesis holds. In particular, the first approach (choice 1) is represented by analysing the generated ISA-level code using models defined at the ISA level that express the energy consumed by the execution of individual ISA instructions. This approach was explored in [21]. It used the precise ISA-level energy models presented in [17], which when used in the static analysis of [21] for a number of small numerical programs resulted in the inference of functions that provide reasonably accurate energy consumption estimations for any input data size (3.9% average error vs. hardware measurements). However, when dealing with programs involving structured types such as arrays, it also pointed out that, due to the loss of information related to program structure and types of arguments at the ISA level (since it is compiled away and no longer relates cleanly to source code), the power of the analysis was limited. In this paper we start by exploring an alternative approach: the analysis of the generated LLVM IR (which retains much more of such information, enabling more direct analysis as well as mapping of the analysis information back to source level) together with techniques that map segments of ISA instructions to LLVM IR blocks [7]

(choice 2). This mapping is used to propagate the energy model information defined at the ISA level up to the level at which the analysis is performed, the LLVM IR level. In order to complete the LLVM IR-level analysis, we have also developed and implemented a transformation from LLVM IR into HC IR and used the CiaoPP resource analyzer. This results in a parametric analysis that similarly to [21] infers energy consumption functions, but operating on the LLVM IR level rather than the ISA level.

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

In summary, the original contributions of this paper are:

1. A translation from LLVM IR to HC IR (Section 3).
2. The integration of all components into an experimental tool architecture, enabling the static inference of energy consumption information in the form of *functions on input data sizes* and the experimentation with the trade-offs described above (Section 2). The components are: LLVM IR and ISA translations, ISA-level energy model and mapping technique (Section 4 and [17, 7]), and analysis tools (Section 5 and [25, 28]).
3. The experimental results and evidence of trade-off of precision versus analyzability (Section 6).
4. A sketch of how the static analysis system can be integrated in a source-level Integrated Development Environment (IDE) (Section 2).

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

2 Overview of the Analysis at the LLVM IR Level

An overview of the proposed analysis system at the LLVM IR level using models at the ISA level is depicted in Figure 2. The system takes as input an XC source program that can (optionally) contain assertions (used to provide useful hints and information to the analyzer), from which a *Transformation and Mapping* process (dotted red box) generates first its associated LLVM IR using the xcc compiler. Then, a transformation from LLVM IR into HC IR is performed (explained in Section 3) obtaining the intermediate representation (green box) that is supplied to the CiaoPP analyzer. This representation includes assertions that express the energy consumed by the LLVM IR blocks, generated from the information produced by the mapper tool (as explained in Section 4). The *CiaoPP analyzer* (blue box, described in Section 5) takes the HC IR, together with the assertions which express the energy consumed by LLVM IR blocks, and possibly some additional (trusted) information, and processes them, producing the analysis results, which are expressed also using assertions. Based on the procedural interpretation of these HC IR programs and the resource-related information

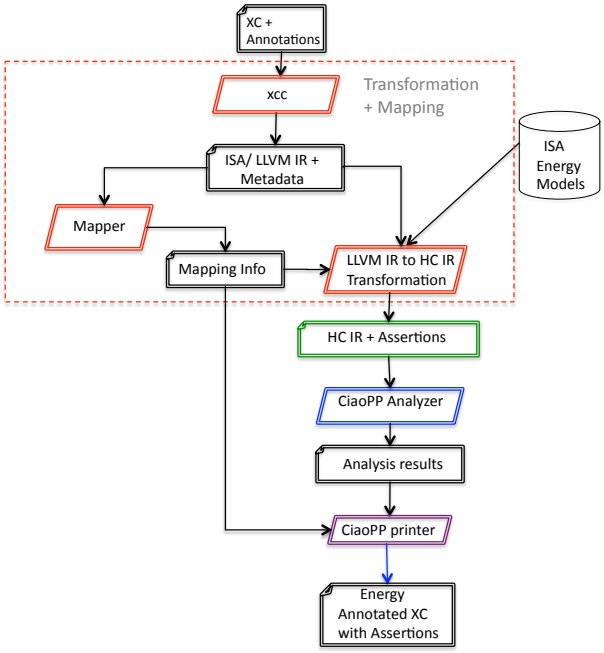


Fig. 2: An overview of the analysis at the LLVM IR level using ISA models.

contained in the assertions, the resource analysis can infer static bounds on the energy consumption of the HC IR programs that are applicable to the original LLVM IR and, hence, to their corresponding XC programs. The analysis results include energy consumption information expressed as functions on data sizes for the whole program and for all the procedures and functions in it. Such results are then processed by the *CiaoPP printer* (purple box) which presents the information to the program developer in a user-friendly format.

3 LLVM IR to HC IR Transformation

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

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

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

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

Each block has an entry point, that we call the *head* of the block (to the left of the $:-$ symbol), with a number of parameters $< params >$, and a sequence of steps (the *body*, to the right of the $:-$ symbol). Each of these S_i steps (or *literals*) is either (the representation of) an LLVM IR *instruction*, or a *call* to another (or the same) block. The transformation ensures that the program information relevant to resource usage is preserved, so that the energy consumption functions of the HC IR programs inferred by the resource analysis are applicable to the original LLVM IR programs.

The transformation also passes energy values for the LLVM IR level for different programs based on the ISA/LLVM IR mapping information that express the energy consumed by the LLVM IR blocks, as explained in Section 4. Such information is represented by means of *trust* assertions (in the Ciao assertion language [13]) that are included in the HC IR. In general, *trust* assertions can be used to provide information about the program and its constituent parts (e.g., individual instructions or whole procedures or functions) to be trusted by the analysis system, i.e., they provide base information assumed to be true by the inference mechanism of the analysis in order to propagate it throughout the program and obtain information for the rest of its constituent parts.

LLVM IR programs are expressed using typed assembly-like instructions. Each function is in SSA form, represented as a sequence of basic blocks. Each basic block is a sequence of LLVM IR instructions that are guaranteed to be executed in the same order. Each block ends in either a branching or a return instruction. In order to transform an LLVM IR program into the HC IR, we follow a similar approach as in a previous ISA-level transformation [21]. However, the LLVM IR includes an additional type transformation as well as better memory modelling.

The following subsections describe the main aspects of the transformation.

3.1 Inferring Block Arguments

As described before, a *block* in the HC IR has an entry point (head) with input/output parameters, and a body containing a sequence of steps (here, representations of LLVM IR instructions). Since the scope of the variables in LLVM IR blocks is at the function level, the blocks are not required to pass parameters while making jumps to other blocks. Thus, in order to represent LLVM IR blocks as HC IR blocks, we need to infer input/output parameters for each block.

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

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

where $next(b)$ denotes the set of immediate target blocks that can be reached from block b with a jump instruction, while $gen(b)$ and $kill(b)$ are the read and

written variables in block b respectively, which are defined as:

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

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

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

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

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

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

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

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

3.2 Translating LLVM IR Types into HC IR Types

LLVM IR is a typed representation which allows retaining much more of the (source) program information than the ISA representation (e.g., types defining compound data structures). Thus, we define a mechanism to translate LLVM IR types into their counterparts in HC IR.

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

At the HC IR level we use *regular types*, one of the type systems supported by CiaoPP [12]. Translating LLVM IR primitive types into regular types is straightforward. The *integer* and *character* types are abstracted as *num* regular type, whereas the *label*, *void*, and *metadata* types are represented as *atm* (atoms).

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

$$array\_type \rightarrow (nested)list$$
$$vector\_type \rightarrow (nested)list$$
$$structure\_type \rightarrow functor\_term$$

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

```
struct mystruct{                                  | :- regtype array1/1.
  int x;                                          | array1:=[] | [~struct|array1].
  int arr[5];                                     |
};                                                | :- regtype struct/1.
                                                  |
void print(struct mystruct [] Arg, int N)         | struct:=mystruct(~num,~array2).
{                                                 |
  ...                                             | :- regtype array2/1.
}                                                 | array2:=[] | [~num|array2].
```

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

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

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

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

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

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

3.3 Transforming LLVM IR Blocks/Instructions into HC IR

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

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

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

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

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

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

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

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

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

4 Obtaining the Energy Consumption of LLVM IR Blocks

Our approach requires producing assertions that express the energy consumed by each call to an LLVM IR block (or parts of it) when it is executed. To achieve this we take as starting point the energy consumption information available from an existing XS1-L ISA Energy Model produced in our previous work of ISA level analysis [21] using the techniques described in [17]. We refer the reader to [17] for a detailed study of the energy consumption behaviour of the XS1-L architecture, containing a description of the test and measurement process along with the construction and full evaluation of such model. In the experiments performed in this paper a single, constant energy value is assigned to each instruction in the ISA based on this model.

A mechanism is then needed to propagate such ISA-level energy information up to the LLVM IR level and obtain energy values for LLVM IR blocks. A set of mapping techniques serve this purpose by creating a fine-grained mapping

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

between segments of ISA instructions and LLVM IR code segments, in order to enable the energy characterization of each LLVM IR instruction in a program, by aggregating the energy consumption of the ISA instructions mapped to it. Then, the energy value assigned to each LLVM IR block is obtained by aggregating the energy consumption of all its LLVM IR instructions. The mapping is done by using the debug mechanism where the debug information, preserved during the lowering phase of the compilation from LLVM IR to ISA, is used to track ISA instructions against LLVM IR instructions. A full description and formalization of the mapping techniques is given in [7].

5 Resource Analysis with CiaoPP

In order to perform the global energy consumption analysis, our approach leverages the CiaoPP tool [12], the preprocessor of the Ciao programming environment [13]. CiaoPP includes a global static analyzer which is parametric with respect to resources and type of approximation (lower and upper bounds) [25, 28]. The framework can be instantiated to infer bounds on a very general notion of resources, which we adapt in our case to the inference of energy consumption. In CiaoPP, a resource is a user-defined *counter* representing a (numerical) non-functional global property, such as execution time, execution steps, number of bits sent or received by an application over a socket, number of calls to a predicate, number of accesses to a database, etc. The instantiation of the framework for energy consumption (or any other resource) is done by means of an assertion language that allows the user to define resources and other parameters of the analysis by means of assertions. Such assertions are used to assign basic resource usage functions to elementary operations and certain program constructs of the base language, thus expressing how the execution of such operations and constructs affects the usage of a particular resource. The resource consumption provided can be a constant or a function of some input data values or sizes. The same mechanism is used as well to provide resource consumption information for procedures from libraries or external code when code is not available or to increase the precision of the analysis.

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

```
:- resource energy.
```

We then provide assertions for each HC IR block expressing the energy consumed by the corresponding LLVM IR block, determined from the energy model, as explained in Section 4. Based on this information, the global static analysis can then infer bounds on the resource usage of the whole program (as well as procedures and functions in it) as functions of input data sizes. A full description of how this is done can be found in [28].

Consider the example in Figure 4 (right). Let P_e denote the energy consumption function for a predicate P in the HC IR representation (set of blocks with

the same name). Let c_b represent the energy cost of an LLVM IR block b. Then, the inferred equations for the HC IR blocks in Figure 4 (right) are:

$$alloca_e(N, Arr) = c_{alloca} + looptest_e(N, Arr)$$
$$looptest_e(N, Arr) = c_{looptest} + loopbody\_loopend_e(0 \neq N, N, Arr)$$
$$loopbody\_loopend_e(B, N, Arr) = \begin{cases} looptest_e(N-1, Arr) & \text{if } B \text{ is } \texttt{true} \\ + \ c_{loopbody} & \\ \\ c_{loopend} & \text{if } B \text{ is } \texttt{false} \end{cases}$$

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

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

Note that using average energy values in the model implies that the energy function for the whole program inferred by the upper-bound resource analysis is an approximation of the actual upper bound (possibly below it). Thus, theoretically, to ensure that the analysis infers an upper bound, we need to use upper bounds as well in the energy models. This is not a trivial task as the worst case energy consumption depends on the data processed, is likely to be different for different instructions, and unlikely to occur frequently in subsequent instructions. A first investigation into the effect of different data on the energy consumption of individual instructions, instruction sequences and full programs is presented in [26]. A refinement of the energy model to capture upper bounds for individual instructions, or a selected subset of instructions, is currently being investigated, extending the first experiments into the impact of data into worst case energy consumption at instruction level as described in Section 5.5 of [17].

6 Experimental Evaluation

We have performed an experimental evaluation of our techniques on a number of selected benchmarks. Power measurement data was collected for the XCore platform by using appropriately instrumented power supplies, a power-sense chip, and an embedded system for controlling the measurements and collecting the power data. Details about the power monitoring setup used to run our benchmarks and measure their energy consumption can be found in [17]. The main goal of our experiments was to shed light on the trade-offs implied by performing the analysis at the ISA level (without using complex mechanisms for propagating type information and representing memory) and at the LLVM level using models defined at the ISA level together with a mapping mechanism.

There are two groups of benchmarks that we have used in our experimental study. The first group is composed of four small recursive numerical programs that have a variety of user defined functions, arguments, and calling patterns

(first four benchmarks in Table 2). These benchmarks only operate over primitive data types and do not involve any structured types. The second group of benchmarks (the last five benchmarks in Table 2) differs from the first group in the sense that they all involve structured types. These are recursive or iterative.

The second group of benchmarks includes `fir(N)` and `biquad(N)`. The former is a (finite impulse response) filter program, which attenuates or amplifies one specific frequency range of a given input signal. It computes the inner-product of two vectors: a vector of input samples, and a vector of coefficients. The latter is an equaliser, which takes a signal and attenuates/amplifies different frequency bands. It 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`.

None of the XC benchmarks contain any assertions that provide information to help the analyzer. Table 1 shows detailed experimental results. Column **SA energy function** shows the energy consumption functions, which depend on input data sizes, inferred for each program by the static analyses performed at the ISA and LLVM IR levels (denoted with subscripts *isa* and *llvm* respectively). We can see that the analysis is able to infer different kinds of functions (polynomial, exponential, etc.). Column **HW** shows the actual energy consumption in nano-joules measured on the hardware corresponding to the execution of the programs with input data of different sizes (shown in column **Input Data Size**). **Estimated** presents the energy consumption estimated by static analysis. This is obtained by evaluating the functions in column **SA energy function** for the input data sizes in column **Input Data Size**. The value N/A in such column means that the analysis has not been able to infer any useful energy consumption function and, thus, no estimated value is obtained. Column **Err vs. HW** shows the error of the values estimated by the static analysis with respect to the actual energy consumption measured on the hardware, calculated as follows: $\textbf{Err vs. HW} = (\frac{\textbf{LLVM}(\textit{or } \textbf{ISA}) - \textbf{HW}}{\textbf{HW}} \times 100)\%$. Finally, the last column shows the ratio between the estimations of the analysis at the ISA and LLVM IR levels.

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

The experimental results show that:

- For the benchmarks in the first group, both the ISA- and LLVM IR-level analyses are able to infer useful energy consumption functions. On average, the analysis performed at either level is reasonably accurate and the relative error between the two analyses at different levels is small. ISA-level estimations are slightly more accurate than the ones at the LLVM IR level (3.9% vs. 9% error on average with respect to the actual energy consumption measured on the hardware, respectively). This is because the ISA-level analysis uses very accurate energy models, obtained from measuring directly at the ISA level, whereas at the LLVM IR level, such ISA-level model needs

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

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

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

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

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

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

to be propagated up to the LLVM IR level using (approximated) mapping information. This causes a slight loss of accuracy.

– For the second group of benchmarks, the ISA level analysis is not able to infer useful energy functions. This is due to the fact that significant program structure and data type/shape information is lost due to lower-level representations, which sometimes makes the analysis at the ISA level very difficult or impossible. In order to overcome this limitation and improve analysis accuracy, significantly more complex techniques for recovering type information and representing memory in the HC IR would be needed. In contrast, type/shape information is preserved at the LLVM IR level, which allows analyzing programs using data structures (e.g., arrays). In particular, all the benchmarks in the second group are analyzed at the LLVM IR level with reasonable accuracy (3% error on average). In this sense, the LLVM IR-level analysis is more powerful than the one at the ISA level. The analysis is also reasonably efficient, with analysis times of about 5 to 6 seconds on average, despite the naive implementation of the interface with external recurrence equation solvers, which can be improved significantly. The scalability of the analysis follows from the fact that it is compositional and can be performed in a modular way, making use of the Ciao assertion language to store results of previously analyzed modules.

7 Related Work

Few papers can be found in the literature focusing on static analysis of energy consumption. A similar approach to the one presented in this paper and our previously developed analysis [21] (from which it builds on) was proposed for upper-bound energy analysis of Java bytecode programs in [24], where the Jimple (a typed three-address code) representation of Java bytecode was transformed into Horn Clauses, and a simple energy model at the Java bytecode level [18]

was used. However, this work did not compare the results with actual, measured energy consumption.

In all the approaches mentioned above, instantiations for energy consumption of general resource analyzers are used, namely [25] in [24] and [21], and [28] in this paper. Such resource analyzers are based on setting up and solving recurrence equations, an approach proposed by Wegbreit [32] that has been developed significantly in subsequent work [27, 5, 6, 30, 25, 1, 28]. Other approaches to static analysis based on the transformation of the analyzed code into another (intermediate) representation have been proposed for analyzing low-level languages [11] and Java (by means of a transformation into Java bytecode) [2]. In [2], cost relations are inferred directly for these bytecode programs, whereas in [24] the bytecode is first transformed into Horn Clauses. The general resource analyzer in [25] was also instantiated in [23] for the estimation of execution times of logic programs running on a bytecode-based abstract machine. The approach used timing models at the bytecode instruction level, for each particular platform, and program-specific mappings to lift such models up to the Horn Clause level, at which the analysis was performed.

By contrast to the generic approach based on CiaoPP, an approach operating directly on the LLVM IR representation is explored in [9]. Though relying on similar analysis techniques, the approach can be integrated more directly in the LLVM toolchain and is in principle applicable to any languages targeting this toolchain. The approach uses the same LLVM IR energy model and mapping technique as the one applied in this paper.

A number of static analyses are also aimed at worst case execution time (WCET), usually for imperative languages in different application domains (see e.g., [33] and its references). The worst-case analysis presented in [16], which is not based on recurrence equation solving, distinguishes instruction-specific (not proportional to time, but to data) from pipeline-specific (roughly proportional to time) energy consumption. However, in contrast to the work presented here and in [23], these worst case analysis methods do not infer cost functions on input data sizes but rather absolute maximum values, and they generally require the manual annotation of loops to express an upper-bound on the number of iterations. An alternative approach to WCET was presented in [14]. It is based on the idea of amortisation, which allows to infer more accurate yet safe upper bounds by averaging the worst execution time of operations over time. It was applied to a functional language, but the approach is in principle generally applicable. A timing analysis based on game-theoretic learning was presented in [29]. The approach combines static analysis to find a set of basic paths which are then tested. In principle, such approach could be adapted to infer energy usage. Its main advantage is that this analysis can infer distributions on time, not only average values.

8 Conclusions and Future Work

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

Our results suggest that performing the static analysis at the LLVM IR level is a reasonable compromise, since 1) LLVM IR is close enough to the source code level to preserve most of the program information needed by the static analysis, and 2) the LLVM IR is close enough to the ISA level to allow the propagation of the ISA energy model up to the LLVM IR level without significant loss of accuracy for the examples studied. Our experiments are based on single-threaded programs. We also have focused on the study of the energy consumption due to computation, so that we have not tested programs where storage and networking is important. However, this could potentially be done in future work, by using the CiaoPP static analysis, which already infers bounds on data sizes, and combining such information with appropriate energy models of communication and storage. It remains to be seen whether the results would carry over to other classes of programs, such as multi-threaded programs and programs where timing is more important. In this sense our results are preliminary, yet they are promising enough to continue research into analysis at LLVM IR level and into ISA-LLVM IR energy mapping techniques to enable the analysis of a wider class of programs, especially multi-threaded programs.

Acknowledgements

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

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In R. D. Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.
3. N. Bjørner, F. Fioravanti, A. Rybalchenko, and V. Senni, editors. *Proceedings of First Workshop on Horn Clauses for Verification and Synthesis*, volume 169 of *EPTCS*, July 2014.

4. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
5. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.
6. S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.
7. K. Georgiou, S. Kerrison, and K. Eder. On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs. *ArXiv e-prints:1510.07095*, Oct. 2015.
8. S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution). In C. Flanagan and B. König, editors, *TACAS*, volume 7214 of *LNCS*, pages 549–551. Springer, 2012.
9. N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder. Static analysis of energy consumption for LLVM IR programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, New York, NY, USA, 2015. ACM.
10. A. Gurfinkel, T. Kahsai, and J. A. Navas. Seahorn: A framework for verifying C programs (competition contribution). In *Proc. of TACAS 2015*, volume 9035 of *LNCS*, pages 447–450. Springer, 2015.
11. K. S. Henriksen and J. P. Gallagher. Abstract interpretation of PIC programs through logic programming. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 184–196. IEEE Computer Society, 2006.
12. M. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
13. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012.
14. C. Herrmann, A. Bonenfant, K. Hammond, S. Jost, H.-W. Loidl, and R. Pointon. Automatic Amortised Worst-Case Execution Time Analysis. In *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OASIcs*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007.
15. H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A Verification Toolkit for Numerical Transition Systems - Tool Paper. In *Proc. of FM 2012*, volume 7436 of *LNCS*, pages 247–251. Springer, 2012.
16. R. Jayaseelan, T. Mitra, and X. Li. Estimating the worst-case energy consumption of embedded software. In *IEEE Real Time Technology and Applications Symposium*, pages 81–90. IEEE Computer Society, 2006.
17. S. Kerrison and K. Eder. Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Transactions on Embedded Computing Systems*, 14(3):1–25, April 2015.
18. S. Lafond and J. Lilius. Energy consumption analysis for two embedded Java virtual machines. *J. Syst. Archit.*, 53(5-6):328–337, 2007.

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

Attachment D1.2.2

Swallow: Building an Energy-Transparent Many-Core Embedded Real-Time System

Accepted at the International Conference on Design, Automation and Test in Europe (DATE 2016)

Swallow: Building an Energy-Transparent Many-Core Embedded Real-Time System

To appear in DATE 2016, Dresden, Germany.

Simon J. Hollis and Steve Kerrison
Department of Computer Science, University of Bristol, United Kingdom
E-mail: harryhollis@cantab.net, steve.kerrison@bristol.ac.uk

***Abstract*—*Swallow* is a many-core platform of interconnected embedded real time processors with time-deterministic execution and a cache-less memory subsystem. Its largest current configuration is 480 × 32-bit processors. It is open-source, designed from the ground up to allow the exploration of flexibility, scalability and energy efficiency in large systems of embedded processors. Further, it enables the behavior of various structures of parallel programs to be explored. It is a proof of concept and design example for other potential systems of this kind. We present the energy transparency features and proportional energy scaling of the system that allows it to be expanded beyond hundreds of cores. We discuss the design choices, construction and novel network implementation of *Swallow*. Currently, the system provides up to 240 GIPS, with each core consuming 71–193 mW, dependent on workload. Its power per instruction is lower than almost all systems of comparable scale. We discuss the challenges associated with efficiently utilizing this system, particularly communication/computation ratios, and give recommendations for future systems and their software.**

I. Introduction

Swallow is a scalable many-core system of multi-threaded real-time processors. Our contributions are the system itself, and the design process used to construct an energy-transparent multi-core research system. An energy-transparent system provides a predictable relationship between software execution and hardware energy consumption.

The *XMOS xCORE XS1-L* micro-architecture [1] is used for the underlying compute and network architecture, providing highly predictable time-deterministic program execution and a low latency network. This is the first example of this architecture assembled at this scale, with a network of 16 cores per *Swallow* board (slice), that can be assembled into systems of much larger sizes, such as in Fig. 1. The current largest demonstrated system is 480 cores. The hardware designs are available under an open source license.

Swallow has been developed as an experimental system for investigating techniques to improve energy efficiency in scalable parallel systems. Key aims of the platform include:

- Scale to hundreds of cores and beyond.
- Deliver proportional scaling in performance and energy.
- Make energy consumption transparent to aid parallel software design exploration for energy efficiency.
- Support a variety of parallel application types and data sharing methods [2], including groups of tasks, pipelines, client/server, message passing and shared memory.

Figure 1. An eight board, 128 core stack of *Swallow* slices.

- Use a real-time embedded system, in contrast to typical heavyweight parallelism of application specific or non-real-time domains.

To support these aims, we built a system that allows flexible expansion and provides energy measurement capabilities. This paper focuses on Swallow and the design challenges surrounding it. A distributed operating system has been developed for *Swallow* [3], and a wider study of benchmarks and program structures for *Swallow* and other platforms is future work.

In this paper, we first examine *Swallow*'s energy measurement capabilities in Section II and its appealing energy proportionality in Section III. Then, we discuss the design considerations and decisions in Section IV, before explaining the novel network implementation in Section V. This is followed by Section VI, a survey of related work, and finally conclusions are made in Section VII.

II. Energy measurement

A number of power measurement points are designed into the system to provide energy-transparency. *Swallow* cores are powered by five separate switch-mode power supplies, each fed from a main 5 V supply. Four of these deliver 1 V to two chips each (four cores), the fifth supplies 3.3 V for I/O, etc.

Each power supply has a shunt resistor on its output and associated probe points. We created a daughter-board that incorporates sensitive differential voltage amplifiers and a high-speed multi-channel analogue-to-digital converter. The resulting system is able to measure individual power supply energy consumption at up to 2 M samples/sec, or 1 M/s if all supplies are sampled simultaneously. Schematics are available online as part of the *Swallow* project open source contribution.

Table I
PER-BIT ENERGIES OF *Swallow* LINKS.

Link type	Data Rate	Max Link Power	Energy per bit
On-chip	250 Mbit/s	1.4 mW	5.6 pJ/bit
On-board, vertical	62.5 Mbit/s	13.3 mW	212.8 pJ/bit
On-board, horizontal	62.5 Mbit/s	12.6 mW	201.6 pJ/bit
Off-board, 30cm FFC	62.5 Mbit/s	680 mW	10880 pJ/bit

A novel feature of this energy measurement is that the measurement data can be collected on the *Swallow* slice itself. In this way, it is possible to create a program that can measure its own power consumption and adapt to the results. Alternatively, the results can be streamed out of the system using an Ethernet interface.

The separate measurement points allow *Swallow* to monitor the balance of energy consumed by the processor cores and external communication links. We present the energy-per-bit for each link in Table I. The system construction and network are described in more detail in Sections IV and V respectively.

The links consume approximately 200 pJ/bit for package-to-package data. The low value can be attributed to the link protocol, which requires only four wire transitions per byte of data. Therefore, the worst case energy usage in communication is half that of a naïve serial or parallel link. Once transmissions go off-board via long flexible cables, the capacitance of those cables becomes the dominating factor for energy, and the energy cost per bit rises by a factor of 50.

Comparing communication costs to computation, profiling of the XS1-L series of processors [4] shows that instructions cause core energy consumption of in the range of 1.0–2.25 μJ at 400 MHz and 1 V, including static power and dependent upon the operations they perform. We can consider this to be 31–70 nJ per bit operated upon. Data movement is therefore relatively inexpensive compared to data processing, before latency is considered.

III. ENERGY PROPORTIONALITY

In order for a many-core system to be scalable, its energy consumption must be both efficient and proportional to the computation it is undertaking. Modern high-performance computing centers consume vast quantities of energy, and energy density is the most important throttle of continued integration of more and more compute power into a fixed space. In this section, we demonstrate how *Swallow* is both energy efficient and energy proportional, and is therefore scalable.

A. Swallow is energy efficient

The processor and memory architecture in *Swallow* are targeted for the embedded application space, where low energy is a primary design goal. A single processor consumes a maximum of 193 mW when active, leading to 3.1 W/slice. Losses in the on-board power supplies and other support logic increase the overall power consumption to approximately 4.5 W/slice (equivalent to 260 mW/core), so a complete 480 core, 30 slice system consumes only 134 W.

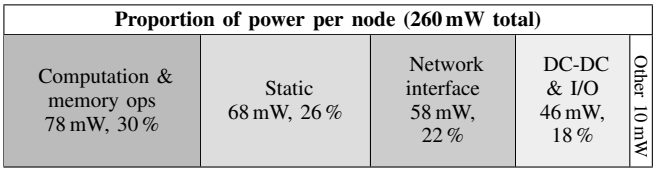


Figure 2. Power distribution for each *Swallow* processor node.

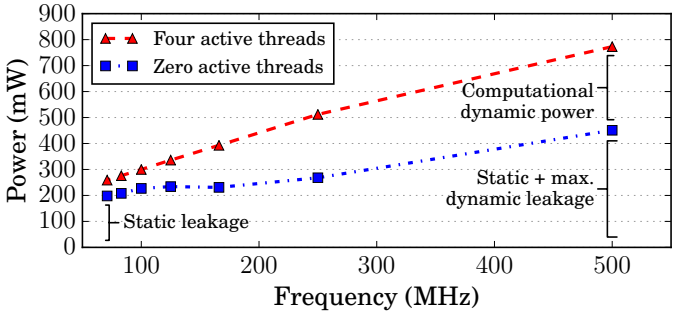


Figure 3. Power consumption with frequency scaling (four cores).

Overall, we see that approximately 18 % of power is used in power supply conversion, support logic and I/O, 30 % is consumed in performing computation, 26 % is spent in non-computational static and dynamic power, and 22 % is used for network interfacing (Fig. 2).

B. Swallow is energy proportional

The XS1-L used in *Swallow* supports dynamic frequency scaling, based on run-time load factors [1]. Fig. 3 shows how the power consumption of a group of four cores scales with their clock frequency.

The power consumed per core, P_c, ranges from 193 mW at 500 MHz to 65 mW at 71 MHz when under heavy load, and 113 mW at 500 MHz to 50 mW at 71 MHz when all cores and threads are idle. The characteristics are linear, giving a directly energy proportional response to clock speed, f, under load,

$$P_c = (46 + 0.30f) \text{ mW}. \quad (1)$$

Thus, static power is 46 mW and dynamic is 0.3 mW/MHz.

Although the current version of *Swallow* does not support voltage scaling, newer xCORE devices do support full DVFS. The additional power savings from voltage scaling on top of frequency scaling can be reliably calculated from knowing the power formula $P = CV^2f$, where C is the capacitance of the switching transistors and V is V_{dd}. We have determined the minimum allowable voltage experimentally to be 0.6 V at 71 MHz and 0.95 V at 500 MHz, and calculate the equivalent DVFS savings for *Swallow* in Fig. 4.

IV. DESIGN OF SWALLOW

The *Swallow* project sought to build a system from commercially available components, keeping costs and design effort low when compared to building a custom processor or collection of IPs in a bespoke SoC/NoC. This demonstrates that many-core systems can be built and used for research quickly, cheaply and with relatively low risk. The system is

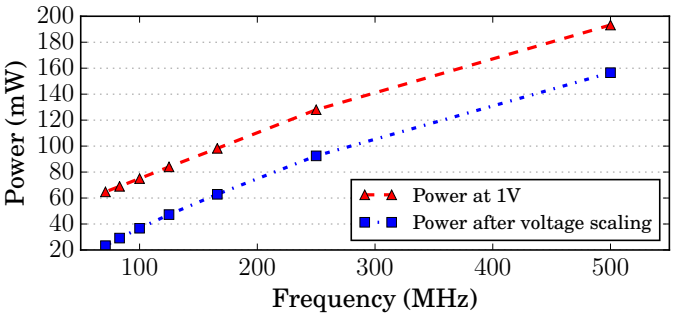


Figure 4. Impact of voltage and frequency scaling on power consumption (four active thread load) for one core.

also designed to be energy-transparent, giving insight into how many-core systems and software consume energy.

A. Processor selection

The choice of processor was governed by a desire to build a system of many predictable embedded processors, upon which novel parallel programming approaches and experiments could be performed, exploiting this predictability. This necessitates the processor have time-deterministic instruction execution in terms of both instruction scheduling and the behavior of the memory hierarchy. The interconnection capabilities must also scale, at least into the hundreds of cores. A number of candidate processors are summarized in Table II.

As is visible in this comparison, few commercial processors present the necessary characteristics of a scalable architecture. The XMOS XS1-L is the only candidate to provide all of these. We further investigated the architecture and determined that its feature set was an excellent match for our system requirements. Key characteristics of XS1 are:

- 64 KiB of single-cycle SRAM local memory per core;
- message passing between cores;
- fast on-chip links, tunable off-chip links;
- fixed instruction completion time for most instructions;
- hardware threads with no context overhead;
- ISA-level primitives for I/O and networking; and
- a network of up to 2^{16} interconnected XS1-L cores.

There are several chips available that are based on XS1-L architecture, with core counts of one or two, with a range of packages and integrated peripherals. To maximize the density of the final configuration a dual-core *XS1-L2A* device was selected. The maximum operating frequency is 500 MHz, yielding 500 MIPS potential throughput per core.

B. Scalable construction

For economic and reliability reasons, we decided to construct *Swallow* from *slices*. A slice is shown in Fig. 5. Each *Swallow* slice comprises sixteen processors across eight chips, with ten off-board network links.

Several components are highlighted in Fig. 5, including the chips, external interconnect headers along all four board edges, along with GPIO near the top edge of the board for further interfaces. The slices are 105 mm × 140 mm in size, with a maximum operating power of 5 W at 12 V. Slices are

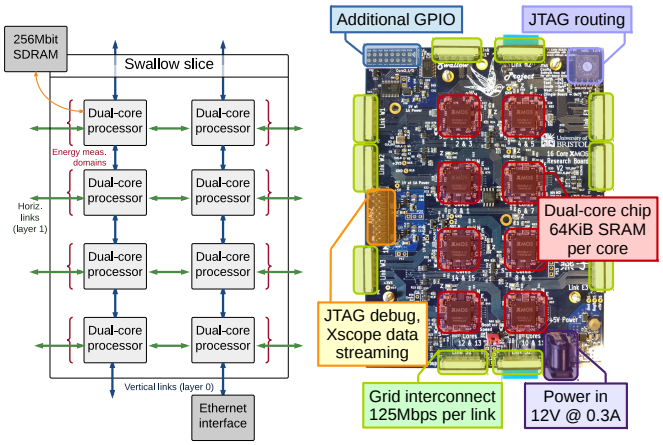


Figure 5. A *Swallow* slice, topology (left); photograph (right), one of forty boards that can be assembled into a grid.

connected with flexible FFC-type ribbon cables. Mounting holes allow vertical stacks to be assembled to minimize the system's footprint (see Fig. 1).

Forty *Swallow* slices have been manufactured, enabling the construction of a 640 processor system. However, yield issues, mostly with edge connectors, mean that the largest machine we have been able to build and test is 480 cores.

C. Computational throughput

Equation (2) shows that the per-thread Instructions Per Second, IPS_t, and processor aggregate throughput, IPS_c, scales according to the number of active threads, N_t, on each core.

$$\text{IPS}_t = \frac{f}{\max(4, N_t)}, \quad \text{IPS}_c = \frac{f \times \min(4, N_t)}{4} \tag{2}$$

This is a property of the XS1-L processor's four-stage execution pipeline [5] with overhead-free thread context switching.

D. Network-on-Chip Implementation

Swallow exploits the network technology of the XS1 architecture. Each XS1-L processor has a number of external communication links. Links are flexibly allocated, with software configurable network partitioning and routing. Links can be aggregated to form a single logical channel with increased bandwidth; or connected separately to different parts of the network, increasing the dimension of the network topology. Links are managed by a switch, with one switch per core in the XS1-L micro-architecture.

V. Swallow Network

Swallow contains a low latency interconnect, suitable for supporting arbitrary traffic types and a mix of parallel or non-parallel applications. In the following section we give an overview of the interconnect and implementation details.

A. Network topology

The device chosen for the *Swallow* system contains two cores and exposes four external network links, in the way shown in Fig. 6. The internal links have four times more

Table II
COMPARISON OF CANDIDATE SWALLOW PROCESSORS. ONLY THE XS1-L MEETS ALL REQUIREMENTS.

Processor	Features				Requirements	
	Cores × data width	Super-scalar	Cache	Typical memory configuration	Multi-core interconnect	Time deterministic
ARM Cortex M	1×32-bit	No	Optional	<varies>	No	W/o cache
ARM Cortex A, single core	1×32-bit	Yes	Yes	<varies>	No	No
ARM Cortex A, multi-core	4×32-bit	Yes	Yes	<varies>	Coherent mem.	No
Adapteva Epiphany	64×32-bit	Yes	No	Local + global SRAM	NoC + external	No
XMOS XS1-L	1×32-bit	No	No	Unified, single cycle SRAM	NoC + external	Yes
MSP430	1×16-bit	No	No	I-Flash + D-SRAM	No	Yes
AVR	1×8-bit	No	No	I-Flash + D-SRAM	No	No
Quark	1×32-bit	No	Yes	Unified DRAM	Ethernet	No

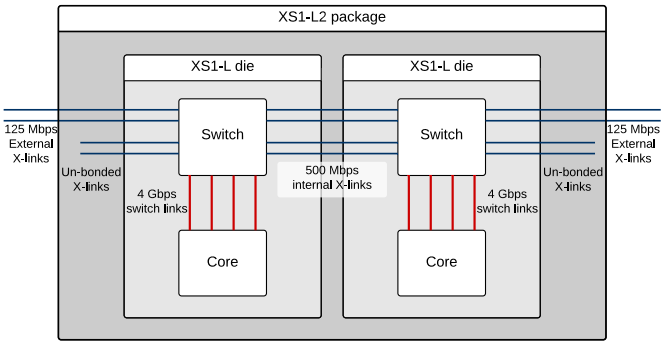


Figure 6. Network links of the XMOS XS1-L2 in a single *Swallow* node.

bandwidth than external links. Data words can be transferred from the core to the network hardware with *just three cycles* of latency (6 ns). This compares with 80 ns for the BlueGene/Q system [6]. In *Swallow*, four external links are then arranged to connect `North`, `South`, `East` and `West` to other devices.

An interesting artifact of *Swallow*'s device selection is that, as is evident in Fig. 6, it is not possible to make a conventional 2D mesh topology. The internal links are already utilized by core-to-core connections. This, combined with the pin-out of the package, means that the most effective grid-like structure is an *unwoven lattice*, shown in Fig. 7. This presents interesting routing challenges, requiring a unique strategy.

The unwoven lattice network is effectively composed of two layers, with each layer containing half of the available cores. One layer routes in the vertical dimension and the other layer routes in the horizontal dimension. Each node in the network also has a connection to a node in the opposite layer, which takes place within a chip package. This topology requires that two-dimensional routes be translated into a form of 2.5 D routing, where routing between layers is required to change horizontal/vertical direction. The dimension order routing [7] strategy that we use prioritizes the vertical dimension first. If a node is attached to the horizontal layer and a vertical communication is required, the message must therefore be sent to the other layer first. In this scheme, there will be at most two layer transitions; the exemplary case being two nodes attached to the horizontal layer that do not share the same vertical index.

Links between *Swallow* slices use flexible cables, allowing the physical topology of the network to be adjusted across a wide variety of configurations, further extending the range of experiments that can be carried out. New routing algorithms can simply be programmed in software to cope with these.

B. Network implementation

Each processor node in the XS1-L2 contains one core and one switch, which has four internal links and two external links, as per Fig. 7. Switches use wormhole routing with credit-based flow control. The instruction set abstracts the network into channel communication which can take the form of either channel switched or packet operation.

Routes are opened with a three byte header prefixed to the front of the first token emitted from a channel end. Any network links utilized along the route are held open until the source channel emits a closing control token. If the close token is never emitted, links are permanently held open, effectively creating a dedicated circuit between two endpoints. The overhead of packet data reduces throughput to approximately 87 % of the link speed, but is dependent upon the packet size.

Multiple links can be assigned to the same routing direction, where a new communication will use the next unused link. This increases bandwidth, provided the number of concurrent communications is equal to or greater than the number of links. This is particularly exploitable for on-chip communication, where there are four links between each core. Provided no more than three links are used for channel switching, packeted data can still flow through the network.

C. Network details

On- and off-chip links all use the same five wire protocol [1], but run at different speeds to preserve signal integrity. The connections and speeds used in *Swallow* are shown in Fig. 6. There is a maximum throughput of 500 Mbit/s per internal link and 125 Mbit/s per external link, providing 2 Gbit/s in-package and 500 Mbit/s externally.

Links send data in eight-bit tokens comprised of two-bit symbols. A token's transmit time is $3T_s + T_t$, where T_s is the inter-symbol delay and T_t the inter-token delay, measured in clock cycles. The fastest possible mode is $T_s = 2, T_t = 1$, yielding the aforementioned 500 Mbit/s at 500 MHz.

The total core-to-core latency for an eight-bit token is 270 ns. The total core-to-core latency for a 32-bit word between packages is 360 ns, equivalent to the time taken to for

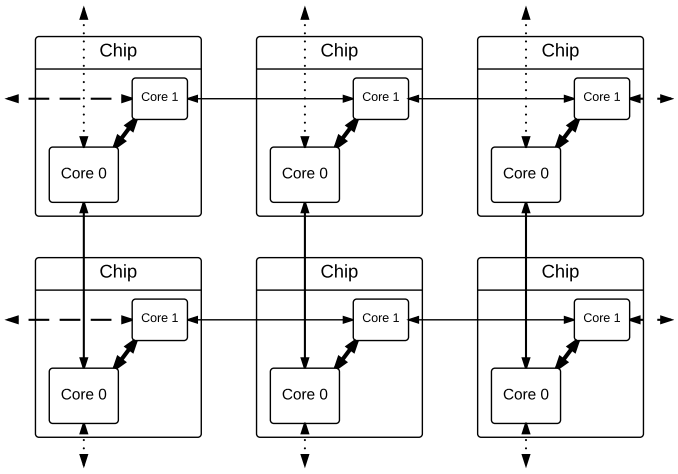


Figure 7. *Swallow*'s unwoven lattice network topology.

the sending thread to execute 45 instructions. Between two cores in a package, this reduces to 40 instructions. Core-local communication takes 50 ns, or approximately 6 instructions.

D. Ratio of communication to computation

The ratio of computation to communication is important to the performance of all systems, where a processor's ability to process data is governed by how quickly data can be moved through the system. This ratio has been termed $\frac{C}{C}$ in some work [8], but for clarity, we adopt the notation $\frac{E}{C}$ [9], where E is *execution* or computation, and C is communication. Communication can be considered movement of data to/from memory, or messages over a network. We define the scope of the ratios (single cores or cross-system) as appropriate.

On *Swallow*, it is possible for a single thread of execution to issue 125 MIPS. Instructions operating on 32-bit data give a maximum per-thread communication throughput of 4 Gbit/s. With four or more threads active threads, $E = 16$ Gbit/s. Core-local communication can sustain this data rate, such that $E = C$ and therefore $\frac{E}{C} = 1$. However, where communication is not core-local, worst case C becomes 250 Mbps and 62.5 Mbps for internal and external links respectively. Communication instructions will block if the output buffer is full.

The total bandwidth of all four package-internal links, $C = 1$ Gbps, gives $\frac{E}{C} = 16$. External links are a quarter of the bandwidth, so externally this ratio increases to 64, assuming non-contended links. If four threads contend for one link, $\frac{E}{C}$ becomes 256.

Considering a slice of sixteen cores, if we take the vertical bisection bandwidth, then $C = 250$ Mbps. If all available compute resource attempts to communicate over the bisection, then $E = 128$ Gbps and therefore $\frac{E}{C} = 512$, which is undesirable. The impact of this on a full *Swallow* system depends on the arrangement of slices and, most importantly, the communication patterns of the programs running upon it.

The systems described in Section VI have system wide computation to communication ratios ranging from 0.42 to 55. Larger, memory-oriented many-core processors such as the Xeon Phi achieve analogous ratios for FLOPS/memory word that sit in the top half of this range [10]. Based on the highly desirable core- and chip-local ratio, and the potential impact of contention over external links, the following considerations should be made when running applications on *Swallow*:

- Prefer core-local communication where possible; it is low latency, high bandwidth, with tight timing predictability.
- Chip-local communication should be the next preference. Link reservation through channel switching can allow similar levels of predictability, with a potential impact on any external traffic routed through the chip.
- Off-chip communication is the most contentious, with the least predictability in large systems, particularly with complex communication patterns.

These problems are analogous to issues in memory hierarchy of more conventional multi-core systems. However, in *Swallow*, more control is placed in the hands of the developer, where the allocation of work onto threads and cores, combined with sensibly scheduled, mostly localized communication, allowing predictability and good $\frac{E}{C}$ to be achieved.

E. External network interface

Communication into and out of a *Swallow* system is performed over an Ethernet bridge module. This module attaches to the *Swallow* network and is addressable as a node in the network, but forwards all data to and from an Ethernet interface. Using this bridge, it is possible to both load programs[1] into and stream data in/out of *Swallow* over Ethernet. *Swallow* supports up to two Ethernet modules per slice (on the `South` external links). Each bridge can support up to 80 Mbit/s of full-duplex data transfer.

VI. Related Work

There are few embedded systems made at the same scale as *Swallow*, with even fewer designed for general purpose computation. Here we identify noteworthy examples, and in Table III provide a comparison of scaling, technology and power characteristics of a number of recent systems.

The Tilera Tile [13] comes the closest to matching *Swallow*'s goals and form. The Tile64 is a 64-core device with five overlaid networks to provide low latency and high throughput between cores in a software configurable way. The effect is to provide a very agreeable computation to communication ratio of 2.4 with 64 cores, and general purpose computation is supported as well as sophisticated network traffic manipulation. The system is highly optimised for streaming traffic, but relies on adding additional networks to improve network performance in larger systems. Tilera's 64 core device [15] consumes 300 mW/core (19.2 W/device).

Adapteva's specialized floating point Epiphany [14] architecture has a similar grid structure to Tilera Tile, with three independent networks. It requires less than 2 W for a 28 nm 64-core device (31 mW/core).

The Centip3De system [12] aims to use 3-D stacked dies to implement a 64-core system based on the ARM Cortex-M3

[1] Swallow boot video: https://youtu.be/kUo11tTeYK0

Table III
COMPARISON OF SCALE, TECHNOLOGY AND POWER PROPERTIES OF RECENT MANY-CORE SYSTEMS.

System	ISA	Cores / chip	Total cores	Tech. node	Power / core	Frequency	µW/MHz
Swallow	XS1	2	16–480	65 nm	193 mW	500 MHz	300
SpiNNaker [11]	ARM9	17	1,036,800	130 nm	87 mW	200 MHz	435
Centip3De [12]	Cortex-M3	64	64	130 nm	203–1851 mW	20–80 MHz	2540–2300
Tile64 [13]	Tile	64	64–480	130 nm	300 mW	1000 MHz	300
Epiphany-IV [14]	Epiphany	64	64	28 nm	31 mW	800 MHz	38.8

processor. Whilst its scale is within an order of magnitude of *Swallow*, it relies on a series of crossbars and coherent DRAM storage and thus scalability to larger sizes may be restricted. Further, the design choices leave it with an undesirable computation to communication ratio of 55. Centip3De exploits near-threshold computing in 130 nm, small ARM Cortex-M3 cores and consumes 203–1851 mW/core, depending on its configuration. Centip3De's high power is mainly due its cache-centric design, which is not present in *Swallow*.

The SpiNNaker system [11] is the best provisioned system in the large scale. SpiNNaker, like Centip3De is based on ARM cores, connecting up to one million ARM9 parts via a highly-connected network. However, the system is targeted at solving a single problem, making it very difficult to overlay general computation tasks, and also making it hard to draw parallels with *Swallow* and the other systems mentioned above. It dissipates an average of 87 mW per core. It is more densely integrated than *Swallow*, with 17 cores per chip. If *Swallow* had these economies of scale, an equivalent level of power efficiency is very possible to obtain, however this would require the fabrication and packaging of new processors.

Swallow's power per core is in the middle of the surveyed range, in line with its operating frequency and process node with respect to the other systems.

VII. CONCLUSIONS

We have presented a many-core real-time embedded system, named *Swallow*, that has been demonstrated on a scale of up to 480 cores, and can be scaled into the thousands. The purpose of *Swallow* is to explore the challenges of building a networked many-core system with predictable embedded processors, and to demonstrate that such a feat is possible. It is being used to investigate energy efficiency of parallel programs and the impact of computation to communication ratios of such systems on these programs. *Swallow* demonstrates excellent core- and chip-local network latency and bandwidth, allow predictable program behavior. The effects of slower, more contended links can be mitigated through appropriate user-controlled communication patterns and allocation of resources.

Swallow is energy efficient, using only 193 mW/core with four active threads. XS1-L processors support dynamic frequency scaling and this allows an energy reduction to as little as 50 mW/core when idle. This could be lowered further in a future revision, by using configurable power supplies.

The platform serves as a new case study and data source in the spectrum of many-core systems, being the first with a strong focus on using real-time, deterministic general purpose embedded hardware.

OPEN SOURCE RELEASE

Swallow is an open source project released in several parts, with licenses appropriate for the hardware, operating system, supporting tools and application software. The latest status of all releases can be found at https://swallow-project.github.io.

ACKNOWLEDGMENT

The authors would like to thank Jamie Hanlon for his advice and technical discussions. The initial 10 prototypes of the *Swallow* system were kindly sponsored by XMOS Ltd. The full system was funded by the University of Bristol's research *Pump-priming* scheme. Research conducted on the *Swallow* platform and the presentation of this work has received funding from the European Union 7th Framework Program agreement no 318337, ENTRA - Whole-Systems Energy Transparency.

REFERENCES

[1] D. May *et al.*, "XS1-L System Specification," pp. 1–40, 2008.
[2] J. Diaz *et al.*, "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, pp. 1369–1386, Aug. 2012.
[3] S. J. Hollis *et al.*, "nOS: a nano-sized distributed operating system for resource optimisation on many-core systems," Tech. Rep., 2015.
[4] S. Kerrison *et al.*, "Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor," *ACM Transactions on Embedded Computing Systems*, vol. 14, pp. 56:1–56:25, Apr. 2015.
[5] XMOS, "XS1-L16A-128-QF124 Datasheet," 2014.
[6] S. Kumar, "Challenges of Exascale Messaging Library Design: Case Study with Blue Gene Machines," in *CASS Workshop*, 2012. [Online]. Available: http://www.ccs3.lanl.gov/cass2012/talks/kumar.pdf
[7] H. Sullivan *et al.*, "A large scale, homogeneous, fully distributed parallel machine, I," in *Proceedings of the 4th annual symposium on Computer architecture - ISCA '77*. New York, New York, USA: ACM Press, May 1977, pp. 105–117.
[8] M. Crovella *et al.*, "Using communication-to-computation ratio in parallel program design and performance prediction," in *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*. IEEE, 1992, pp. 238–245.
[9] D. May, "Communicating Processors: Past, Present and Future," in *Networks-on-Chip, International Symposium on*, 2008. [Online]. Available: http://www.cs.bris.ac.uk/~dave/nocs.pdf
[10] K. S. Solnushkin, "Memory Bandwidth for Intel Xeon Phi." [Online]. Available: http://clusterdesign.org/2013/02/memory-bandwidth-for-intel-xeon-phi-and-friends/
[11] S. Furber *et al.*, "Overview of the spinnaker system architecture," *IEEE Transactions on Computers*, vol. 62, pp. 2454–2467, 2013.
[12] D. Fick *et al.*, "Centip3De: A cluster-based NTC architecture with 64 ARM cortex-M3 cores in 3D stacked 130 nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 48, pp. 104–117, 2013.
[13] S. Bell *et al.*, "TILE64^{TM} processor: A 64-core SoC with mesh interconnect," in *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, vol. 51, 2008.
[14] Adapteva, "E64G401 EPIPHANY^{TM} 64-core microprocessor datasheet." [Online]. Available: http://www.adapteva.com/docs/e64g401_datasheet.pdf
[15] A. Agarwal *et al.*, "The Tile Processor Architecture, Embedded Multicore for Networking and Multimedia," in *Hot Chips*, 2007.

Attachment D1.2.3

A Transformational Approach to Parametric Accumulated-cost Static Profiling

Accepted at the 13th International Symposium on Functional and Logic Programming (FLOPS 2016)

A Transformational Approach to Parametric Accumulated-cost Static Profiling

R. Haemmerlé[1], P. López-García[1,2], U. Liqat[1],
M. Klemen[1], J.P. Gallagher[1,3], and M.V. Hermenegildo[1,4]

[1] IMDEA Software Institute
[2] Spanish Council for Scientific Research (CSIC)
[3] Roskilde University
[4] Technical University of Madrid (UPM)

Abstract. Traditional static resource analyses estimate the total resource usage of a program, without executing it. In this paper we present a novel resource analysis whose aim is instead the *static profiling of accumulated cost*, i.e., to discover, for selected parts of the program, an estimate or bound of the resource usage accumulated in each of those parts. Traditional resource analyses are parametric in the sense that the results can be functions on input data sizes. Our static profiling is also parametric, i.e., our accumulated cost estimates are also parameterized by input data sizes. Our proposal is based on the concept of cost centers and a program transformation that allows the static inference of functions that return bounds on these accumulated costs depending on input data sizes, for each cost center of interest. Such information is much more useful to the software developer than the traditional resource usage functions, as it allows identifying the parts of a program that should be optimized, because of their greater impact on the total cost of program executions. We also report on our implementation of the proposed technique using the CiaoPP program analysis framework, and provide some experimental results.

1 Introduction and Motivation

The execution of software consumes resources such as time, energy, and memory. The goal of automatic program resource analysis is to infer the resources that a program uses as a function of the size of the input data or other environmental parameters of the program, without actually executing the program. Previous work on this topic, mainly for inferring asymptotic time complexity bounds, goes back to the 1970s. Recent research has adapted and extended these techniques for inferring other resources, including for example energy [15, 14].

In this paper we investigate an extension of this problem which, although based on the same essential techniques, has a different range of applications. Rather than estimating the total resource usage of a program, we wish to perform *static profiling* of its resource usage. This means that we intend to discover, for selected parts of the program, an estimate of the resources used by those parts. As before, the estimates will be parameterised by input sizes. However, these

input sizes will be of the entry predicate/function, unlike the input sizes of the selected parts, as in the standard resource analysis.

There are several motivations for this research. Firstly, a profile of the resource usage of the program can show the developer which parts of the program are the most resource critical. For example, it can expose the cost of functions that are perhaps not particularly resource hungry by themselves but which are called many times. Such parts are natural targets for optimization, since there a small improvement can yield important savings. Secondly, there are cases where the overall resource functions of a program might not be obtainable. This can be for instance because some program parts are too complex for analysis or because the code for some parts is not available and the cost cannot even be reasonably estimated. In this case useful information may still be obtained by excluding such parts from the analysis, obtaining information about the resource usage for the rest of the program. Thirdly, resource usage models (for example Tiwari's energy consumption model [29]) are sometimes based on summing the individual resource usage of basic components of the program. The analysis presented here fits naturally with such models. Finally, in cases where a program has mutually recursive functions/predicates, the standard cost analysis infers similar resource functions for each recursive function. In such cases, a static profile finds precisely the resource functions for each mutually recursive part of the program, and helps identify the parts that are responsible for most of the cost.

The traditional profiling techniques are dynamic (i.e., require executing the program on some particular input) and are based either on code instrumentation, i.e., introducing additional pieces of code in the sections to be measured, or on running a process that performs the profiling together with the measured program. In both cases, the dynamic profiler introduces an overhead in the resource measured that needs to be properly discriminated, which is non trivial. For example, it may be the case that an instruction in the original program has a very different energy consumption in the presence of code added by the profiler just before it. In contrast, the static profiling approach we propose in this paper obtains safe upper and lower bounds on resource consumption, because it is based on the semantics of the program rather than particular executions of it. I.e., the results are valid for all possible program inputs.

Our starting point is the well-developed technique of extracting recurrence relations that express resource usage functions [32, 25, 6, 5, 7, 1]. These are then solved to get a closed-form function expressing the (bounds on) parameterised resource usage. In our work we will make use the CiaoPP program analysis framework, which includes a set of generic resource analyses based on these techniques. In particular, we will use the analysis described in [28]. CiaoPP operates on an intermediate semantic program representation based on Horn Clauses [16], that we will refer to as the "HC IR." By transforming the input language into this intermediate representation, the CiaoPP framework has been shown capable of analyzing imperative programs at the source, bytecode, or binary level with competitive precision and efficiency (see [16, 21, 20, 15, 14] for details).

Our approach to static profiling is based on a transformation that is performed at the level of the CiaoPP Horn Clause-based intermediate representa-

tion. The proposed transformation allows a standard cost analyzer (CiaoPP in our experiments) to statically infer functions that return bounds on *accumulated costs* depending on input data sizes, for a number of predefined program points of interest (predicates in our case), referred to as cost centers. Intuitively, given a program $\mathcal{P}$, the cost accumulated in a given predicate $\mathtt{p} \in \mathcal{P}$ is defined in the context of the execution of a single call to another predicate $\mathtt{q} \in \mathcal{P}$. It expresses the addition of (part of) the resource usages corresponding to the execution of all calls to predicate p generated by a single call to predicate $\mathtt{q} \in \mathcal{P}$.

In the rest of the paper, Section 2 presents informally a general model of (dynamic) profiling and how we turn it into a static version. Section 3 reviews established techniques for cost analysis based on extracting and solving cost relations. Section 4 formalizes our notion of *accumulated cost*. Section 5 describes the implementation of the technique, based on a source-to-source transformation. Section 6 reports some experimental results. In Section 7 we comment on some related work and finally Section 8 concludes, discussing future directions.

2 From Dynamic Profiling to Static Profiling

We start by presenting informally a general model of (dynamic) profiling and how we turn it into a static version. Our model is based on the notion of *cost centers*, inspired from the work of Sansom and Peyton Jones [26] and Morgan and Jarvis [18]. This approach was also applied to Logic Programs and extended to perform run-time checking of non-functional properties in [17]. Intuitively a cost center provides a dynamic scoping mechanism to uniquely attribute the execution costs of a part of the code to an identifier. The scope of the cost center is dynamic in the sense that execution costs of code that are not explicitly associated to a cost center are dynamically attributed to the same cost center as the caller. For a number of languages it is convenient to identify the cost centers with (a subset of) functions, procedures, or predicates. In this paper we follow this path. Alternatively, cost centers can be defined by special scoping constructs [26].

As an example,[5] assume that a programmer wishes to profile a program which uses the following `variance()` function (`variance()` naively computes the variance of an array of integers):

```
1 int  variance(int * arr, int size){
2   int tmp[size], i = size;
3   while(i > 0) {
4     i--;
5     tmp[i] = (arr[i] - mean(arr, size));
6     tmp[i] = tmp[i] * tmp[i]; }
7   return mean(tmp, size);
8 }
```

[5] As mentioned in the introduction, CiaoPP's analyses deal with programs written in such C-like languages (among others) by analyzing corresponding Horn Clause representations.

Assume that `mean()` is a given function that computes the mean of an integer array. First consider that both `mean()` and `variance()` are cost centers. In this case the actual execution costs of the code that appears textually within the `variance()` function will be aggregated at each call to such function and will be attributed to the `variance()` cost center. However the cost of calls to `mean()` –including those made from `variance()`– will not be attributed to `variance()`. Now consider the case where `variance()` is declared a cost center, but `mean()` is not. In this case the execution costs of calls to `mean()` made from the `variance()` function will be also aggregated to those of `variance()` (but not those made from other points in the program).

Returning to the case where both `variance()` and `mean()` are declared as cost centers, assume that the programmer profiles the energy consumption (measured as nano joules, nJ) of a call to the `variance()` function over the array $\{1, 2, 3, 4\}$, on some particular architecture. Assume that the result of the profiler is that 74.7 units of energy are accumulated in the `variance()` cost center and 464.4 units in the `mean()` cost center. Since `mean()` is called 4 times, the cost of a single call to it (with the array above) would be 116.1 nJ (464.4/4). If only `variance()` were declared a cost center, the profiler would have accumulated all the cost in it, i.e., $464.4 + 74.7$ nJ. In such a case, the cost measured by the profiler would be the same as what we call the standard cost of a (single) call to `variance()` with the given array (i.e., 539 nJ).

Since the accumulated value in the `mean()` cost center is much larger than that accumulated in the `variance()` cost center, this indicates that for this particular call most of the energy is consumed inside the `mean()` function, i.e., that this function is responsible for most of the standard cost of the call to `variance()`. This can be a strong indicator that it may be worthwhile to either optimize the body of `mean()` or try to reduce the number of times it is called. Note, however, that with just this data, which come from a run with a particular input, the programmer does not really have any guarantees that the results are representative of the general behavior of the program for all inputs. This problem is usually tackled by repeating the process on a large set of different inputs. This can lead to more indicative results, but still has several drawbacks. First, this process can be very long, because profiling usually introduces additional execution costs, which get multiplied by the number of inputs. Second, and more importantly, even if a large number of inputs is used, this still does not provide a strong guarantee, i.e., there may be some corner case inputs for which the call behaves in a very different way. Finally, the approach does not allow the comparison of the asymptotic cost accumulated in the different cost centers.

To overcome the problems outlined above, we propose to *statically* infer (lower and upper) bounds on the cost accumulated in the cost centers as functions of the sizes of the input data to the profiled call (the call `variance()` in our example). In the example above, the system we have implemented infers (for the resource "energy"[6]) that for a call to `variance()` with a list of size $size$, the costs

[6] Using as back-end analysis the energy analysis of [15, 14] on an XCore XS1 processor with the program compiled by the XMOS xcc compiler without optimization.

accumulated in the `variance` and `mean()` functions are $24.32 + \texttt{size} \times 12.59$ and $23.03 + 17.46 \times \texttt{size}^2 + 40.49 \times \texttt{size}$ energy units (nano joules) respectively. In this case the system infers these expressions for both the upper and lower bounds, which means that they are exact costs. Hence, the programmer does have the guarantee that for all non-trivial calls (i.e., for all calls with non-empty lists) *and for any input data*, the code of `mean()` consumes most of the energy. In this case an obvious improvement can be made, since the call to `mean(arr, size)` can be safely moved outside the loop:

```
1  int  variance(int * arr, int size){
2    int tmp[size];
3    int i = size;
4    int m = mean(arr, size);
5    while(i > 0) {
6      i--;
7      tmp[i] = (arr[i] - m);
8      tmp[i] = tmp[i] * tmp[i];
9    }
10   return mean(tmp, size);
11 }
```

For this version of the program, the system infers that the costs accumulated in the `variance()` and `mean()` functions are $28.18 + \texttt{size} \times 8.73$ and $46.06 + 34.92 \times \texttt{size}$ energy units (nano joules) respectively. For brevity and simplicity we chose a program that is rather naive and where the optimization is obvious (and would in fact be done by some compilers automatically), but the same reasoning applies to more complex cases that are not easy to spot without profiling information. Furthermore, the static profiling functions can also be used for guiding automatic optimization by the compiler.

3 The Classical Cost Relations-based Parametric Static Analysis

The approach to cost analysis based on setting up and solving recurrence equations was proposed in [32] and has been developed significantly in subsequent work. For example, in [25] an automatic upper-bound analysis was presented based on an abstract interpretation of a step-counting version of a functional program, in order to infer both execution time and execution steps. However, size measures could not automatically be inferred and the experimental section showed few details about the practicality of the analysis. In the context of Logic Programming, a semi-automatic analysis was presented in [6, 5] that inferred upper-bounds on the number of execution steps, given as functions on the input data sizes. This work also proposed techniques to address the additional challenges posed by the Logic Programming paradigm, as, for example, dealing with the generation of multiple solutions via backtracking. However, a shortcoming of the approach was its loss in precision in the presence of divide-and-conquer programs in which the sizes of the output arguments of the "divide" predicates

are dependent. This approach was later fully automated (by integrating it into the CiaoPP system and automatically providing *modes and size measures*) and extended to inferring both upper- and lower-bounds on the number of *execution steps* (which is non-trivial because of the possibility of failure) in [7, 10]. In addition, [7] introduced the setting up of non-deterministic recurrence relations for the class of divide-and-conquer programs mentioned above, and proposed a technique for computing approximated closed form bound functions for some of them. Such a technique was based on bounding the number of terminal and non-terminal nodes in the set of computation trees corresponding to the evaluation of the non-deterministic recurrence relations, and bounding the cost of such nodes. Non-deterministic recurrence relations were also used and further developed in [1] (named Cost Relations). The approach in [6, 5, 7] was generalized in [22] to infer *user-defined resources* (by using an extension of the Ciao assertion language [11]), and was further improved in [28] by defining the resource analysis itself as an *abstract domain* that is integrated into the PLAI abstract interpretation framework [19, 24] of CiaoPP, obtaining features such as *multivariance*, efficient fixpoints, and assertion-based verification and user interaction. A significant additional improvement brought about by [28] is that it is combined with a *sized types* abstract domain, which allows the inference of non-trivial cost bounds when they depend on the sizes of input terms and their subterms at any position and depth. Recently, many other approaches have been proposed for resource analysis [30, 12, 9, 13, 23, 8, 1, 2]. While based on different techniques, all these analyses infer, for all predicates `p` of a given program $\mathcal{P}$, an approximation of the notion of cost that we call the *standard cost* or *single call cost*. Most of them infer an upper bound, while others infer both upper and lower bounds. The following example shows this (for the case of CiaoPP) and also illustrates that this concept of cost may not be directly useful for locating performance bottlenecks.

Example 1. Consider the following implementation of an `eval(E,M,R)` predicate that evaluates modulo $2^{\mathtt{M}}$ a given expression `E` built from additions and multiplications. This implementation assumes that two predicates are given: `add(A,B,M,R)` and `mult(A,B,M,R)`, that respectively add and multiply two infinite precision numbers `A` and `B` modulo $2^{\mathtt{M}}$, and unify the result with `R`.

```
1 eval(const(A),M,R) :- eval_const(A,M,R).
2 eval(A+B,      M,R) :- eval_add(A,B,M,R).
3 eval(A*B,      M,R) :- eval_mult(A,B,M,R).
4
5 eval_const(A,_,R) :- R=A.
6 eval_add(A,B,M,R) :- eval(A,M,RA), eval(B,M,RB), add(RA,RB,M,R).
7 eval_mult(A,B,M,R):- eval(A,M,RA), eval(B,M,RB),mult(RA,RB,M,R).
```

For the sake of simplicity, assume that all the costs are null except those related to the evaluation of `add/4` and `mult/4`. Assume that the cost of the evaluation of `add(A,B,M,R)` is `M` and the cost of the evaluation of `mult(A,B,M,R)` is $\mathtt{M}^2$. Under these assumptions, the standard CiaoPP cost analysis infers that the cost of the evaluation of `eval(E,M,R)` is bounded by $(2^{\mathrm{depth}(\mathtt{E})} - 1) \times (\mathtt{M} + \mathtt{M}^2)$

where depth(E) stands for the depth of the expression E – note that the exact bound is $(2^{\mathrm{depth(E)}} - 1) \times \mathtt{M}^2$. However, such an analysis does not help finding precisely which part of the code is responsible for most of the cost. Indeed since all the predicates (`eval/3`, `eval_add/4`, and `eval_mult/4`) are mutually recursive, the system will infer a similar cost for `eval_add/4` and `eval_mult/4`. Furthermore, those costs will be expressed in terms of different input variables making the actual comparison difficult.

4 Parametric Accumulated-cost Static Profiling

We now formalize the new notion of cost that we propose, the *accumulated cost*, which has been intuitively described in Section 1. As mentioned before, our approach is based on the notion of cost centers: user-defined program points (predicates, in our case) to which execution costs are assigned during the execution of a program. Data about computational events is accumulated by the cost center each time the corresponding program point is reached by the program execution control flow.

We start by presenting a formal *profiled semantics* for Logic Programming. For this purpose we assume given a program $\mathcal{P}$. We also assume that each predicate $\mathtt{p}$ is associated with a cost $\mathrm{cost}_{\mathtt{p}} \in \mathbb{R}$ and that the cost centers are defined as a set $\Diamond$ of predicate symbols. In the following we will use overlined symbols such as $\bar{\mathtt{t}}$, $\bar{\mathtt{x}}$, or $\bar{\mathtt{e}}$ to denote a sequence of terms, variables, or arithmetic expressions.

We define a *predicate call with context* as a tuple of the form $\mathtt{r} : \mathtt{p}(\bar{\mathtt{t}})$, where $\mathtt{r}$, the *context*, is a cost center (i.e., a predicate from $\Diamond$) and $\mathtt{p}(\bar{\mathtt{t}})$ is a predicate call. Then, we define *profiled states* as tuples of the form $\langle \alpha \,;\, \theta \,;\, \kappa \rangle$ where α is a sequence of predicate calls with context, θ is a substitution that maps variables to calling data, and κ, the *cost assignment*, is a family of real numbers indexed by the cost centers $\Diamond$. The *profiled resource semantics* is defined as the smallest relation $\rightarrow_{\mathcal{P}}$ over profiled states satisfying:

$$\frac{\mathtt{q} = \mathrm{update}_{\Diamond}(\mathtt{p}, \mathtt{r}) \quad (\mathtt{p}(\bar{\mathtt{s}}) \ \mathtt{:-}\ \beta) \in \mathcal{P}\rho \qquad \sigma \text{ is an m.g.u. of } \bar{\mathtt{s}} \text{ and } \bar{\mathtt{t}}\theta}{\langle \mathtt{r} : \mathtt{p}(\bar{\mathtt{t}}), \alpha \,;\, \theta \,;\, \kappa \rangle \rightarrow_{\mathcal{P}} \langle \mathtt{q} : \beta, \alpha \,;\, \theta \circ \sigma \,;\, \kappa[\mathtt{q} \mapsto \kappa_{\mathtt{q}} + \mathrm{cost}_{\mathtt{p}}] \rangle}$$

$$\frac{\sigma \text{ is an m.g.u. of } \mathtt{t} \text{ and } [\mathtt{s}\theta]}{\langle \mathtt{r} : (\mathtt{t} \ \mathtt{is}\ \mathtt{s}), \alpha \,;\, \theta \,;\, \kappa \rangle \rightarrow_{\mathcal{P}} \langle \alpha \,;\, \theta \circ \sigma \,;\, \kappa \rangle}$$

where:

- $\mathtt{q} : \beta, \alpha$ is a notation for the sequence $\mathtt{q} : \mathtt{p}_1(\bar{\mathtt{s}}_1), ..., \mathtt{q} : \mathtt{p}_n(\bar{\mathtt{s}}_n), \alpha$, assuming β is the sequence $\mathtt{p}_1(\mathtt{s}_1), ..., \mathtt{p}_n(\mathtt{s}_n)$.
- $[\mathtt{s}]$ stands for the arithmetic evaluation of $\mathtt{s}$ (if $\mathtt{s}$ is not a ground arithmetic expression, then $[\mathtt{s}]$ is not defined, as well as the rule using it),
- ρ stands for a renaming with fresh variables,
- $\kappa[\mathtt{q} \mapsto c]$ is the assignment that maps $\mathtt{p}$ to c if $\mathtt{p} = \mathtt{q}$ or to $\kappa_{\mathtt{p}}$ otherwise, and
- $\mathrm{update}_{\Diamond}(\mathtt{p}, \mathtt{r})$ equals either $\mathtt{p}$ if $\mathtt{p} \in \Diamond$, or $\mathtt{r}$ otherwise.

The first rule can be understood as an extension of SLD resolution with cost. Concretely, the cost $\mathrm{cost}_{\mathtt{p}}$ of the called predicate $\mathtt{p}$ is added to the value of the

current cost center, the cost center being updated beforehand to the current predicate if the latter is in fact a cost center, and left unchanged otherwise. The latter rule characterizes the semantics of the built-in `is/2`, where we assume w.l.o.g. that the operation has no cost. Standard left-to-right evaluation is simply recovered by ignoring the cost assignment together with the calling contexts. In the following section, we will use the notation $(\alpha\,;\theta)$, where α is a sequence of predicate calls and θ a substitution, to denote a standard (non-profiled) LP state.

In the following, we use Π as the set of tuples of terms, and $\mathbb{R}$ to denote the set of real numbers. For any cost center $\mathtt{p} \in \Diamond$, the *profiled resource usage function* is the function $\mathcal{C}^{\mathtt{p}}_{\Diamond} : 2^{\Pi} \to 2^{\mathbb{R}^n}$ defined as:

$$\mathcal{C}^{\mathtt{p}}_{\Diamond}(\bar{T}) = \begin{cases} \left\{\kappa \mid \bar{\mathtt{t}} \in \bar{T} \;\&\; \langle \mathtt{p} : \mathtt{p}(\bar{\mathtt{t}})\,;\epsilon\,;\bar{0}\rangle \to^*_{\mathcal{P}} \langle \Box\,;\theta\,;\kappa\rangle\right\} & \text{if } \mathtt{p}(\bar{\mathtt{t}}) \text{ terminates} \\ & \text{universally } \forall \bar{\mathtt{t}} \in \bar{T} \\ \mathbb{R}^n & \text{otherwise} \end{cases}$$

where $\bar{0}$ stands for the trivial cost assignment that maps any cost center to 0, $\to^*_{\mathcal{P}}$ is the reflexive and transitive closure of $\to_{\mathcal{P}}$, $\Box$ denotes the empty sequence of predicate calls, ϵ is the identity substitution, and n is the number of cost centers. We use the "top" element in $2^{\mathbb{R}^n}$ (i.e., $\mathbb{R}^n$) to denote a "don't know" cost for non-terminating programs, which, for simplicity, are currently not defined in our framework. Note that the cost $\kappa_{\mathtt{p}}$ in an infinite derivation can be (asymptotically) different from $+\infty$ as (1) $\mathtt{p}$ can be the context of only a finite number of the steps involved in an infinite derivation, and (2) because costs of predicates can be zero or negative. The profiled semantics is a natural generalization of the standard resource usage semantics which is able to handle several costs which are accumulated in the cost centers. Indeed the resource usage function inferred by the standard analysis can be understood as the function $\mathcal{C}^{\mathtt{p}} = \mathcal{C}^{\mathtt{p}}_{\{\mathtt{p}\}}$ defined over a unique cost center.

$\mathcal{C}^{\mathtt{p}}_{\mathtt{q}}(\bar{T})$ denotes the cost accumulated in $\mathtt{q}$ from the calls $\mathtt{p}(\bar{\mathtt{t}})$ ($\bar{\mathtt{t}} \in \bar{T}$), that is, the union of the i^{th} component of all tuples in $\mathcal{C}^{\mathtt{p}}_{\Diamond}(\bar{T})$ if $\mathtt{q}$ is the i^{th} cost center in $\Diamond$ (formally $\mathcal{C}^{\mathtt{p}}_{\mathtt{q}}(\bar{T}) = \left\{\kappa_{\mathtt{q}} \mid \kappa \in \mathcal{C}^{\mathtt{p}}_{\Diamond}(\bar{T})\right\}$). In particular, if $\mathtt{p}(\bar{\mathtt{t}})$ deterministically succeeds (e.g., when it is obtained by translation of some imperative program) the cost accumulated in $\mathtt{q}$ from $\mathtt{p}(\bar{\mathtt{t}})$ is unique, i.e., $\mathcal{C}^{\mathtt{p}}_{\mathtt{q}}(\{\bar{\mathtt{t}}\}) = \{c\}$ for some $c \in \mathbb{R}$. In such a case, by a slight abuse of notation, we denote the unique value by $\mathcal{C}^{\mathtt{p}}_{\mathtt{q}}(\bar{\mathtt{t}})$.

Example 2. Consider the deterministic program given in example 1. If we profile the program, defining all the predicates of the program as cost centers except `add/4` and `mult/4`, the costs accumulated in `eval_const/3`, `eval_add/4` and `eval_mult/4` for a call of the form `eval(E,M,R)` are respectively bounded by 0, $(0.5 \times 2^{\mathtt{depth(E)}} \times \mathtt{M})$, and $(0.5 \times 2^{\mathtt{depth(E)}} \times \mathtt{M}^2)$. This makes it easier to spot the source of most of the cost, i.e., `eval_mult/4`. Therefore, to improve the efficiency of the whole program, it can be useful to concentrate on this predicate, either by optimizing its implementation or by reducing the number of times it is called.

We write $\mathtt{p} \rightsquigarrow \mathtt{q}$ if $\mathtt{q}$ is reachable from $\mathtt{q}$, that is, if $\mathtt{q}(\bar{\mathtt{t}}) \rightarrow^*_{\mathcal{P}} (\mathtt{p}(\bar{\mathtt{s}}), \alpha)$ for some calling data $\bar{\mathtt{t}}$ and $\bar{\mathtt{s}}$, and some sequence of calls α. Given a set $\Diamond$ of cost centers assigned to a program $\mathcal{P}$ and some predicate $\mathtt{p}$, we define the set of *reachable cost centers* from $\mathtt{p}$ as the sequence $\Diamond_{\mathtt{p}} = \{\mathtt{q} \mid \mathtt{q} \in \Diamond \wedge \mathtt{p} \rightsquigarrow^\star \mathtt{q}\}$.

Theorem 1. *Let $\mathcal{P}$ be a program and $\Diamond \subseteq pred(\mathcal{P})$ a set of cost centers for it. Then, for all $\mathtt{p} \in \Diamond$: for all $\bar{T} \subset \Pi$ it holds that: $\mathcal{C}_{\mathtt{p}}(\bar{T}) = \left\{\sum_{\mathtt{q} \in \Diamond_{\mathtt{p}}} \mathcal{C}^{\mathtt{p}}_{\mathtt{q}}(\bar{T})\right\}$. In particular, if $\mathtt{p}(\bar{\mathtt{t}})$ deterministically succeeds $\mathcal{C}_{\mathtt{p}}(\bar{\mathtt{t}}) = \sum_{\mathtt{q} \in \Diamond_{\mathtt{p}}} \mathcal{C}^{\mathtt{p}}_{\mathtt{q}}(\bar{\mathtt{t}})$.*

Note that theorem 1 provides the basis for a compositional and modular definition of the standard (i.e., single call) cost analysis, from the results of the accumulated cost analysis. Note also that (by definition of reachable cost center) $\mathtt{p}$ is always reachable from itself, even though $\mathtt{p}$ does not call itself.

5 Inferring Accumulated Cost via Transformation

As mentioned before, our implementation of the static profiler is based on a source-to-source transformation. In this section we show such a transformation that allows obtaining accumulated cost information for cost centers by performing a sized type analysis in CiaoPP. Basically, the transformation consists of adding *shadow arguments* to each predicate of the Horn clauses that represent the accumulated cost for each cost center.

5.1 The Transformation

In this section we assume there is exactly n cost centers and $\Diamond$ is defined as the family $\{\mathtt{p}_i\}_{i \in 0..n-1}$. The transformation proposed consists of adding $n+1$ shadow arguments to each predicate, such that on success those variables will be assigned to the costs accumulated in the program. There are n shadow arguments for the cost accumulated in the cost centers called by the predicate, and an additional one for the cost associated with the calling context, which is not known statically.

Formally, the transformation is defined by the functions $[\![\cdot]\!]_\Diamond$ and $[\![\cdot]\!]_n$ that respectively translate clauses and goals. The function $[\![\cdot]\!]_n : \mathcal{A}^* \rightarrow (\mathcal{A}^* \times E^{n+1})$ (E is the set of possibly non-ground arithmetic expressions) that translates sequences of atoms is defined recursively on the length of the goal as:

- $[\![\mathtt{q}(\bar{\mathtt{t}}), \alpha]\!]_n = ((\mathtt{q}(\bar{\mathtt{t}}, \bar{\mathtt{x}}), \beta), \bar{\mathtt{x}} + \bar{\mathtt{e}})$ where $(\beta, \bar{\mathtt{e}}) = [\![\alpha]\!]_n$
- $[\![\Box]\!]_n = (\Box, \bar{0})$

where $\bar{\mathtt{x}}$ (resp. $\bar{0}$) stands for a sequence of $(n+1)$ fresh variables (a sequence of $(n+1)$ zeros). On the other hand the function $[\![\cdot]\!]_\Diamond : \mathcal{C} \rightarrow \mathcal{C}$ is defined by cases as follows:

$$[\![\mathtt{q}(\bar{\mathtt{t}}) \texttt{ :- } \alpha]\!]_\Diamond = \begin{cases} (\mathtt{q}(\bar{\mathtt{t}}, \bar{\mathtt{x}}) \texttt{ :- } \beta, \\ \quad \bar{\mathtt{x}} \texttt{ is } \bar{\mathtt{e}}[\bar{\mathtt{e}}_n \leftarrow 0][\bar{\mathtt{e}}_i \leftarrow (\mathrm{cost}_{\mathtt{q}} + \mathtt{e}_i + \mathtt{e}_n)] & \text{if } \mathtt{q} = \mathtt{p}_i \in \Diamond \\ (\mathtt{q}(\bar{\mathtt{t}}, \bar{\mathtt{x}}) \texttt{ :- } \beta, \\ \quad \bar{\mathtt{x}} \texttt{ is } \bar{\mathtt{e}}[\bar{\mathtt{e}}_n \leftarrow (\mathrm{cost}_{\mathtt{q}} + \mathtt{e}_n)] & \text{otherwise} \end{cases}$$

where $(\beta, \bar{e}) = [\![\alpha]\!]_n$, $\bar{x}$ is a sequence of $n+1$ fresh variables, and $\bar{x}$ `is` $\bar{e}$ is a notation for x_0 `is` e_0, ..., x_n `is` e_n (assuming $\bar{x} = (x_0, \ldots, x_n)$ and $\bar{e} = (e_0, \ldots, e_n)$).

The translation of a clause is defined by case on the predicate `q` it defines. Suppose `q` is some cost center $p_i \in \Diamond$. In this case the costs associated with `q` itself (i.e., cost_q) are assigned to the argument corresponding to `q`, namely e_i. Furthermore the costs in evaluating `q` that are not associated to any other cost center (i.e., e_n) are also assigned to e_i. Thus we have $\bar{e}[\bar{e}_n \leftarrow 0][\bar{e}_i \leftarrow (\text{cost}_q + e_i + e_n)]$. On the other hand, if `q` is not a cost center, then the costs associated with `q` are associated to its context, namely e_n, and thus we have $\bar{e}[\bar{e}_n \leftarrow (\text{cost}_q + e_n)]$.

Example 3. We show now the translation of the code corresponding to our running example, given in Example 1, assuming that the cost centers are `eval/3`, `eval_const/4`, `eval_add/4`, and `eval_mult/4`. In the translation the output arguments `Ce`, `Cc`, `Ca`, and `Cm` correspond to the cost accumulated in the respective cost centers. On the other hand, the output `C` is the cost that has not been accumulated in any of the cost centers. Within the translation we leave the actual implementations of `add/4` and `mult/4` unspecified and marked by `(...)`.

```
1  eval(const(A),M,R,Ce,Cc,Ca,Cm,C) :-
2    eval_const(A,M,R,De,Dc,Da,Dm,D),
3    Ce is De+D, Cc is Dc, Ca is Da, Cm is Dm, C is 0.
4  eval(A+B,M,R,Ce,Cc,Ca,Cm,C) :-
5    eval_add(A,B,M,R,De,Dc,Da,Dm,D),
6    Ce is De+D,  Cc is Dc, Ca is Da, Cm is Dm, C is 0.
7  eval(A*B,M,R,Ce,Cc,Ca,Cm,C) :-
8    eval_mult(A,B,M,R,De,Dc,Da,Dm,D),
9    Ce is De+D,  Cc is Dc, Ca is Da, Cm is Dm, C is 0.
10 eval_const(A,_M,R,Ce,Cc,Ca,Cm,C) :- R=A,
11   Ce is 0, Cc is 0, Ca is 0, Cm is 0, C is 0.
12 eval_add(A,B,M,R,Ce,Cc,Ca,Cm,C) :-
13   eval(A,M,RA,De,Dc,Da,Dm,D), eval(B,M,RB,Ee,Ec,Ea,Em,E),
14   add(RA,RB,M,R,Fe,Fc,Fa,Fm,F),
15   Ce is De+Ee+Fe, Cc is Dc+Ec+Fc, Ca is Da+Ea+Fa+D+E+F,
16   Cm is Dm+Em+Fm, C is 0.
17 eval_mult(A,B,M,R,Ce,Cc,Ca,Cm,C) :-
18   eval(A,M,RA,De,Dc,Da,Dm,D), eval(B,M,RB,Ee,Ec,Ea,Em,E),
19   mult(RA,RB,M,R,Fe,Fc,Fa,Fm,F),
20   Ce is De+Ee+Fe, Cc is Dc+Ec+Fc, Ca is Da+Ea+Fa,
21   Cm is Dm+Em+Fm+D+E+F, C is 0.
22 add(RA,RB,M,R,Ce,Cc,Ca,Cm,C) :-
23   (...)
24   Ce is 0, Cc is 0, Ca is M, Cm is 0, C is 0.
25 mult(RA,RB,M,R,Ce,Cc,Ca,Cm,C) :-
26   (...)
27   Ce is 0, Cc is 0, Ca is 0, Cm is M*M, C is 0.
```

The following theorem states that the translation of a given program simulates the original one, while reifying the cost assignment as a first-order argument.

Theorem 2. *Assume a given program $\mathcal{P}$ profiled according n cost centers $\Diamond = \{\mathtt{p}_0, \ldots, \mathtt{p}_{n-1}\}$ and a predicate* `p` *different from* `is`*.*

(Soundness) *If $(\mathtt{p}(\bar{\mathtt{t}}, \bar{\mathtt{x}}) \,;\, \theta) \rightarrow^*_{[\![\mathcal{P}]\!]_\Diamond} (\Box \,;\, \sigma)$ (for some sequence of pairwise $\bar{x}$ distinct variables free in $\bar{t}$ and θ) then there exists a derivation of the form $\langle \mathtt{p}_i : \mathtt{p}(\bar{\mathtt{t}}) \,;\, \theta \,;\, \bar{0} \rangle \rightarrow^*_{\mathcal{P}} \langle \Box \,;\, \sigma' \,;\, \kappa \rangle$, with $\bar{\mathtt{t}}\sigma' = \bar{\mathtt{t}}\sigma$, $\kappa_{\mathtt{p}_j} = \mathtt{x}_j\sigma$ (for $j \in 1, ..., n-1$ and $j \neq i$), and $\kappa_i = \mathtt{x}_i\sigma + \mathtt{x}_n\sigma$.*

(Completeness) *If $\langle \mathtt{p}_i : \mathtt{p}(\bar{\mathtt{t}}) \,;\, \epsilon \,;\, \bar{0} \rangle \rightarrow^*_{\mathcal{P}} \langle \Box \,;\, \theta \,;\, \kappa \rangle$, then there exists a derivation of the form $(\mathtt{p}(\bar{\mathtt{t}}, \bar{\mathtt{x}}) \,;\, \epsilon) \rightarrow^*_{[\![\mathcal{P}]\!]_\Diamond} (\Box \,;\, \sigma)$, with $\bar{\mathtt{t}}\theta = \bar{\mathtt{t}}\sigma$, $\kappa_{\mathtt{p}_k} = \mathtt{x}_j\sigma$ (for $j \in 1, ..., n-1$ and $j \neq i$), and $\kappa_i = \mathtt{x}_i\sigma + \mathtt{x}_n\sigma$.*

5.2 Performing the Resource Usage Analysis

The Horn Clause program resulting from the transformation described above, whose predicates are augmented with shadow output arguments representing the accumulated cost for each cost center, is analyzed in order to infer lower and upper bounds on the sizes of such arguments, which actually represent bounds on the respective accumulated costs.

In order to obtain such bounds, we use the size analysis presented in [27, 28], integrated in the CiaoPP system. The goal of this analysis is to infer lower and upper bounds on the sizes of output arguments as a function on the sizes of input arguments. This analysis is based on the abstract interpretation framework present in CiaoPP, and basically infers *sized types* for output arguments. Sized types are representations that incorporate structural (shape) information and allow expressing both lower and upper bounds on the size of a set of terms and their subterms at any position and depth. For a more detailed explanation of this process, we refer the reader to [27].

Continuing with our running example, consider the output argument `Ca`, which represents the accumulated cost of the cost center `eval_add/4` when it is called from `eval/4`. In a preprocessing step, the program is unfolded in order to avoid mutual recursion, which makes the analysis harder. After the unfolding step, the analysis infers types for the predicate arguments by using an existing analysis for regular types [31]. This analysis infers that for a call to a transformed version of `eval/4` (with shadow variables) of the form:

```
eval(Exp,M,R,Ce,Cc,Ca,Cm,C)
```

with `Exp` and `R` bound and the rest of arguments as free variables, then `Ca` gets bound to a number upon success, i.e., a term of type `num`. From the inferred regular type, the analysis derives a *sized type schema*, which is just a sized type with variables in bound positions, along with a set of constraints over those variables.

In this case, the corresponding sized type for `num` is $\mathtt{num}^{(\alpha,\beta)}$, where α and β are variables representing lower and upper bounds on the size of the elements

that belong to such type. The metric we use for the size of a number is its actual value, since `num` is a basic type. For compound types, e.g., lists, trees or arithmetic expressions, we can use several metrics for the size of any term belonging to them, such as the depth of such term (as in our example), or the number of type rule applications needed for the type definition to succeed for such term.

The next step involves setting up recurrence relations between size variables. Thus, for β, that represents the upper bound of the size of `Ca`, we obtain the following equation (where $Size_{arg}^{pred}$ is the size of the argument arg corresponding to predicate $pred$):

$$\beta = Size_{Ca}^{eval}(Size_{exp}, M) = \begin{cases} 2 * Size_{Ca}^{eval}(Size_{exp} - 1, M) + M & \text{if } Size_{exp} > 1 \\ 0 & \text{otherwise} \end{cases}$$

At this point, we have obtained a recurrence relation that represents the size of the output argument. However, such expression is not useful for some applications. One disadvantage of using recurrence relations is that the evaluation of them given concrete input values usually takes longer than the evaluation of an equivalent non-recursive expression. In addition, it is not easy to see the complexity order of a given procedure just by looking at its recurrence relation, and the comparison with other functions is also more difficult. For this reason, the analysis uses a solver for obtaining closed-form representations for recurrence relations. Such closed forms can be either exact solutions or safe overapproximations. In our example, the closed-form version for the recurrence is:

$$\beta = Size_{Ca}^{eval}(Size_{exp}, M) = (2^{Size_{exp}} - 1) * M$$

Assuming that the metric for the size of arithmetic expressions is the depth of the term representing them, we have that $Size_{exp} = depth(exp)$. Thus, we can finally conclude that the accumulated cost of `eval_add/4` when called from `eval/3` (i.e., the size of `Ca` in the transformed version of the program), is given by

$$(2^{depth(exp)} - 1) * M$$

6 Experimental Results

We have performed an experimental evaluation of our techniques with the prototype implementation described in Section 5 over a number of selected benchmarks from [28]. The benchmarks are written directly as Horn Clause programs (in Ciao). In each benchmark, a number of predicates are marked as cost centers. The results are shown in Table 1. Static profiling was performed for each cost center, capturing the accumulated cost with respect to an entry predicate (marked with a *star*, e.g., $appendAll2^*$). While in the experiments both upper and lower bounds were inferred, for the sake of brevity we only show upper bound functions. Also, each clause body is assumed to have unitary cost.

Table 1. Experimental results.

Program	Cost-Center Predicate	Accumulated Cost UB	Static vs. Dyn	Standard Cost UB	#Calls
appendAll2	*appendAll2**	b_1	0%	$2b_1b_2b_3+b_1b_2+b_1$	1
	appendAll	b_1b_2	33%	b_1b_2	b_1
	append	$2b_1b_2b_3$	61%	β	$b_1b_2+b_1$
hanoi	*hanoi**	2^v-1	0%	$2^{v+1}-2$	1
	processMove	2^v-1	0%	1	2^v-1
coupled	*coupled**	1	0%	$v+1$	1
	f	$\frac{v}{2}+\frac{(-1)^v}{4}+\frac{3}{4}$	1.2%	v	$\frac{v}{2}-\frac{(-1)^v}{4}+\frac{1}{4}$
	g	$\frac{v}{2}+\frac{(-1)^v}{4}-\frac{1}{4}$	0%	v	$\frac{v}{2}+\frac{(-1)^v}{4}-\frac{1}{4}$
minsort	*minsort**	$\beta+1$	0%	$\frac{(\beta+1)^2}{2}+\frac{\beta+1}{2}$	1
	findmin	$\frac{(\beta+1)^2}{2}+\frac{\beta-1}{2}$	7%	β	$\beta+1$
dyade	*dyade**	β_1	0%	$\beta_1(\beta_2+1)$	1
	mult	$\beta_1\beta_2$	0%	β	β_1
variance_ naive	*variance**	1	0%	$2\beta^2$	1
	sq_diff	$\beta-1$	0%	$2\beta_2\beta_1-2\beta_2$	$\beta-1$
	mean	$2\beta^2-\beta$	0%	$\beta-1$	β
variance	*variance**	1	0%	$5\beta+3$	1
	sq_diff	β	0%	β	β
	mean	$4\beta+2$	0%	$2\beta+1$	2
listfact	*listfact**	β	0%	$\beta(\delta+2)$	1
	fact	$\beta\delta+\beta$	47%	$\delta+1$	β

– $ln^{(\alpha_i,\beta_i)}(n^{(\gamma_i,\delta_i)})$ represents the size of the list of numbers L_i, where β_i and δ_i (resp. α_i and γ_i) denote the upper (resp. lower) bounds on the length of the list and the size of its numbers respectively.
– $lln^{(a_1,b_1)}(lln^{(a_2,b_2)}(ln^{(a_3,b_3)}(n^{(a_4,b_4)})))$ represents the size of the list of lists of lists of numbers similarly.
– $n^{(\mu,v)}$ denotes the size of a number with lower- and upper-bounds μ and v respectively.

Column 1 of Table 1 shows the list of benchmarks while column 2 provides the list of cost centers for each benchmark. Column 3 shows the parametric accumulated cost inferred for each cost center, as a resource usage upper bound function on input data sizes of the entry predicate. Column 4 compares the parametric accumulated cost function of each cost center from column 3 with the results from a dynamic profiling tool [17]. Although the analysis infers upper bounds on the accumulated cost, for some benchmarks these are exact upper bounds (in fact, exact costs) and for others these are correct but relatively imprecise. The imprecision introduced in the benchmarks *listfact* and *appendAll2* is due to the fact that the cost not only depends on the input data sizes but also on the sizes of the sub-terms in the input data, since the analysis statically assumes an upper bound on the sizes of the sub-terms. Note that CiaoPP is the only analysis tool that infers concrete upper bound functions over sized types (costs that depend on the sizes of subterms) [28].

Column 5 shows for comparison the cost inferred by the standard (i.e., non-accumulated) cost analysis [28] for each program and its auxiliary predicates (also marked as cost centers). The comparison of the accumulated and standard cost functions (columns 3 vs. 5) shows the usefulness of our approach: the upper bounds on cost centers display accumulated costs for program parts that were not visible with the standard analysis. For instance, similarly to Example 1, the *coupled* benchmark has two auxiliary mutually recursive predicates f and g that are processing elements of a list alternatively until the list becomes empty. The standard analysis infers almost the same upper bound for both functions due to the mutual recursion, whereas the accumulated cost precisely points out the source of cost in the mutually recursive parts. Similarly, in *hanoi*, although the cost of *processMove* (processing a single *hanoi* move) is unitary, we can see that it is called an exponential number of times. The analysis is providing hints to the programmer about the parts of the program that are most profitable candidates for optimization. Note that the upper bound cost functions inferred by static profiling for each cost center predicate are on the input data sizes of the program (entry predicate), in contrast to the standard analysis where the cost functions are on the input data sizes of the predicate that the cost function corresponds to.

Finally, in column 6 an additional *#Calls* cost is presented, indicating the number of times each predicate is called, as a function of input data sizes of the entry predicate. These cost functions are inferred using the standard analysis by defining explicitly a *#Calls resource* for each cost center predicate. A big complexity order in the number of calls to a predicate (in relation to that of a single call) might give hints to reduce the number of calls to such predicate in order to effectively reduce its impact on the overall cost of the program (i.e., the cost of a call to the entry point). More interestingly, since both the *Accumulated* and *#Calls* costs of a predicate `q` are expressed as functions of input data sizes of the entry predicate, their quotient (Column 3 / Column 6) is meaningful and will give an approximation of the cost of a single call to `q` as a function of the input data sizes *of the entry predicate*. Note that the standard analysis (Column 5) also provides an upper-bound approximation of this cost but as a function of the input data sizes of predicate `q`.

7 Related Work

Static profiling in the context of Worst Case Execution Time (WCET) Analysis of real-time programs is presented in [4]. It proposes an approach to computing worst-case timing information for all code parts of a program using a complementary metric, called *criticality*. Every statement of a real-time program is assigned with a criticality value, expressing how critical the respective code is for the global WCET. Our approach is not limited to WCET, since it is able to obtain results for a general class of *user-defined* resources. Furthermore, our inferred metrics are parametric on the input data sizes of the main program, in contrast to the *criticality* metric, which is a numeric value in the range $[0, 1]$. In

addition, our approach is modular and compositional, able to compute accumulated costs w.r.t. calls originating from different procedures of the program, and not only the main program entry point. In [3] the authors present static profiling techniques to estimate the execution *likelihood* and *frequency* of program points in order to assess whether the cost of certain compile-time optimizations would pay off. To this end, they explore the use of some static analysis techniques for predicting the result of conditional branches, such as assuming uniform distribution over all branches, making heuristic based predictions, and performing value range propagation. In this context, our approach can be used to infer bounds on the number of times a certain program point will be called from a given entry point, as functions on input data sizes, in contrast with a single value representing the execution likelihood or frequency. Besides, since our techniques are supported mainly by the theory of abstract interpretation, the approximations inferred are *correct* by design.

8 Conclusions

In this paper we have presented a novel approach of *static profiling of accumulated cost* that infers upper- and lower-bounds of the resource usage accumulated in particular parts of a program as a functions on the input data sizes of the program. We have constructed a prototype implementation of the proposed approach using the CiaoPP program analysis framework. Preliminary experimental results with the tool support the usefulness of our approach where precise accumulated upper bound cost functions were inferred for parts of the program for which the standard analysis was not able to infer precise information. The upper bound functions inferred by the static profiling were also evaluated against a dynamic profiling tool [17], and showed promising accuracy for the static analysis. However in cases where the cost depended on the sizes of the sub-terms of the input, the upper bound accumulated cost loses precision.

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

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011.
2. E. Albert, S. Genaim, and A. N. Masud. More Precise yet Widely Applicable Cost Analysis. In *12th Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, volume 6538 of *Lecture Notes in Computer Science*, pages 38–53. Springer Verlag, January 2011.
3. C. Boogerd and L. Moonen. On the use of data flow analysis in static profiling. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 79–88, Sept 2008.

4. F. Brandner, S. Hepp, and A. Jordan. Static profiling of the worst-case in real-time programs. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS 2012, pages 101–110, New York, NY, USA, 2012. ACM.
5. S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
6. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.
7. S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.
8. J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In *PPDP*, pages 1–12. ACM, 2012.
9. B. Grobauer. Cost recurrences for DML programs. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 253–264, New York, NY, USA, 2001. ACM.
10. M. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
11. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012. http://arxiv.org/abs/1102.5497.
12. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems*, 34(3):14:1–14:62, 2012.
13. A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 331–342, New York, NY, USA, 2002. ACM.
14. U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Energy Consumption at Different Software Levels: ISA vs. LLVM IR. In *Proc. of the Foundational and Practical Aspects of Resource Analysis*, Lecture Notes in Computer Science. Springer, 2015. To Appear.
15. 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 *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 72–90. Springer, 2014.
16. M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in Lecture Notes in Computer Science, pages 154–168. Springer-Verlag, August 2007.
17. E. Mera, T. Trigo, P. López-García, and M. Hermenegildo. Profiling for Run-Time Checking of Computational Properties and Performance Debugging. In *Practical Aspects of Declarative Languages (PADL'11)*, volume 6539 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, January 2011.
18. R. G. Morgan and S. A. Jarvis. Profiling Large-Scale Lazy Functional Programs. *Journal of Functional Programing*, 8(3):201–237, 1998.

19. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.
20. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, pages 29–32, April 2008. Extended Abstract.
21. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 65–82. Elsevier - North Holland, March 2009.
22. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.
23. F. Nielson, H. Nielson, and H. Seidl. Automatic complexity analysis. In *Programming Languages and Systems*, volume 2305 of *Lecture Notes in Computer Science*, pages 243–261. Springer Berlin Heidelberg, 2002.
24. G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium (SAS 1996)*, number 1145 in Lecture Notes in Computer Science, pages 270–284. Springer-Verlag, September 1996.
25. M. Rosendahl. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 144–156. ACM Press, 1989.
26. Patrick M. Sansom and Simon L. Peyton Jones. Time and Space Profiling for Non-Strict, Higher-Order Functional Languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95*, pages 355–366, New York, NY, USA, 1995. ACM.
27. A. Serrano, P. Lopez-Garcia, F. Bueno, and M. Hermenegildo. Sized Type Analysis for Logic Programs (technical communication). In T. Swift and E. Lamma, editors, *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement*, volume 13, pages 1–14. Cambridge U. Press, August 2013.
28. 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.
29. V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: a First Step Towards Software Power Minimization. *IEEE Trans. VLSI Syst.*, 2(4):437–445, 1994.
30. P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proceedings of the International Workshop on Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, September 2003.
31. C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.
32. B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, September 1975.

Attachment D1.2.4

Integrating energy modelling into the development process: A Makefile approach

Integrating energy modelling into the development process: A Makefile approach

Steve Kerrison

December 2015

Abstract

This technical report demonstrates the integration of ENTRA energy modelling tools into the development workflow through the use of Makefiles. This is an experimental demonstration, showing a potential path to integration into development tools.

1 Tools

This demo uses the following tools and related material:

Compiler XMOS xTIMEcomposer version 14.0.2, available from the vendor website.
Simulator Axe open source XS1 simulator, modified per [1], available from
`https://github.com/stevekerrison/tool_axe/tree/credit`.
Model The `xmtracem` trace modeller. Experimental version used in [1].
Code A modified version of the parallel FIR implementation delivered in D5.1 of ENTRA.

These tools carry their own dependencies, but these are the top-level requirements of this demo.

2 Explanation of integration

For this demonstration, we use a vanilla Makefile structure, rather than the enhanced version that is typically provided with xTIMEcomposer build environments. This is in line with the deliverable D5.1, and simplifies the integration process for demonstrative purposes.

The FIR code is modified in four main ways:

1. The program performs a single iteration of the FIR filtering run on a set of 192 samples.
2. Workload is spread across two cores to demonstrate communication modelling.
3. Voltage adjustments are removed, for compatibility with `axe`.
4. Port based measurement triggering is issued, unifying model and hardware measurement methods.

All other changes related to the Makefile and development flow.

2.1 Makefile

The differences between the original Makefile and the modified Makefile are detailed in section 3. The important differences are explained here.

The Makefile now includes reference to the necessary model parameter files used by the energy modelling tool, `xmtracem`.

The use of the SLICEKIT's voltage adjustment code is removed for compatibility with the `axe` simulator, and so the Makefile is updated to reflect this.

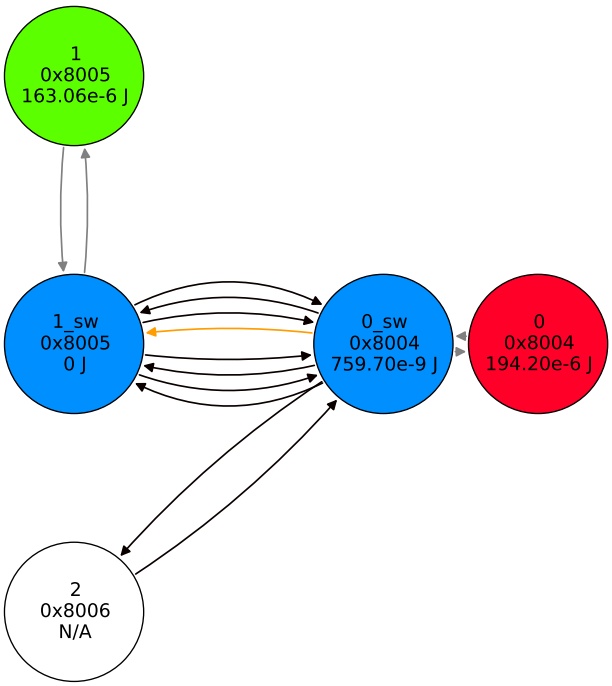


Figure 1: Two-core FIR modelling visualisation

The most important change is the addition of a `model` target, that will invoke the simulation and modelling process against the compiled program binary. This target invokes two commands: `axe`, the XS1 instruction set simulator, and `xmtracem` for modelling.

The `axe` tools is configured to emit JSON instruction traces. These are read by `xmtracem` and used to estimate the device energy consumption. A number of arguments are supplied to the tool, the most important of which are explained here:

model The energy model file.

vfs-params The voltage and frequency scaling parameters, which can be experimentally applied as per [1].

trigger This allows an I/O port to be used as a trigger for either energy modelling or hardware energy monitoring. In hardware, this I/O pin would be the trigger pin for the measurement hardware. In our modelling environment, `xmtracem` is used is a similar fashion. The modelling run will terminate as soon as the falling-edge of of the trigger signal is observed.

plot Specifies the output file for a visualisation of modelled energy consumption per core, switch and link. By default, this is the binary filename with "-energy.pdf" appended to it.

xn The location of the platform description. In current versions of `xmtracem`, this is extracted from the binary XE file, so the Makefile configures this likewise.

2.2 Running the modelling

Issuing the "make model" command will compile the code (if necessary), then run the simulator and trace modeller. After a few seconds, it will complete, producing a text report (below) as well as PDF visualisation (fig. 1).

The information in both forms requires some knowledge to interpret, which beyond the scope of this report.

Text output

```
$ make model
axe -t --trace-json fir.xe | xmtracem2.py --core-voltage=1.0 --model model-20150204.pkl \
 --vfs-params=vfs-params-20150716.json --trigger=0x10a00
 --xn=fir.xe --plot fir.xe-energy.pdf --nodump -
Connecting to stdin...
Model stats
===========
Base power:  97.24e-03 W
Max time: 0.000678718000001
Time (Wall | Recorded): 741.13e-06 S | 678.72e-06 S
 Core 1 (8005):
  Energy (static | dynamic      | comms        | total):
     28.20e-06 J | 134.86e-06 J | 0 J | 163.06e-06 J
  Total instructions: 119788, FNOPS: 10706

Time (Wall | Recorded): 711.34e-06 S | 678.72e-06 S
 Core 0 (8004):
  Energy (static | dynamic      | comms        | total):
     28.20e-06 J | 166.00e-06 J | 759.70e-09 J | 194.96e-06 J
  Total instructions: 302740, FNOPS: 27651

Total Energy (static   | dynamic      | comms        | total):
           56.40e-06 J | 300.86e-06 J | 759.70e-09 J | 358.02e-06 J
Total Power  (static   | dynamic      | comms        | total):
           83.10e-03 W | 443.28e-03 W |   1.12e-03 W | 527.50e-03 W
```

3 Summary

This report has shown modelling tools developed during the course of the ENTRA project can be integrated into development work-flow through standard Makefile-based methods.

The XMOS toolchain makes heavy use of rich Makefiles. Although complete integration would require more engineering effort, this report has shown a path towards this.

The state of the modelling and simulation tools used in this demo are in-line with with what is detailed in [1].

Makefile diff

```
$ diff par/Makefile par-model/Makefile
1c1,4
< XCFLAGS=-g -O3 SLICEKIT-A16.xn
---
> XNFILE=SLICEKIT-A16.xn
> XCFLAGS=-g -O3 $(XNFILE)
> MODELFILE=model-20150204.pkl
> VFSFILE=vfs-params-20150716.json
3,7c6,7
< fir.xe: main.o fir.o voltage.o
<  xcc $(XCFLAGS) -o fir.xe main.o fir.o voltage.o
<
< voltage.o: voltage.xc
<  xcc $(XCFLAGS) -c voltage.xc
---
> fir.xe: main.o fir.o
>  xcc $(XCFLAGS) -o fir.xe main.o fir.o
14a15,17
> clean:
>  rm -f *.o *.xe
>
19a23,26
>
> model: fir.xe
>  axe -t --trace-json fir.xe | xmtracem2.py --core-voltage=1.0 --model $(MODELFILE) \
>   --vfs-params=$(VFSFILE) --trigger=0x10a00 --xn=$? --plot $?-energy.pdf --nodump -
>
```

References

[1] S. Kerrison and K. Eder. "Modeling and visualizing networked multi-core embedded software energy consumption". In: *ArXiv e-prints* (Sept. 2015). arXiv: 1509.02830 [cs.DC]. URL: http://arxiv.org/abs/1509.02830.