# COMPLEX

## COdesign and power Management in PLatform-based design space EXploration

<table>
<thead>
<tr>
<th>WP no.</th>
<th>Deliverable no.</th>
<th>Lead participant</th>
</tr>
</thead>
<tbody>
<tr>
<td>WP2</td>
<td>D2.2.2</td>
<td>PoliMi</td>
</tr>
</tbody>
</table>

**WP2**

## Final report on embedded software estimation and model generation

Prepared by Carlo Brandolese, Gianluca Palermo, William Fornaciari (PoliMi), Héctor Posadas, Fernando Herrera, Pablo Peñil, Eugenio Villar (UC), Francisco Ferrero, Raúl Valencia (GMV), Bart Vanthournout (SNPS)

Issued by PoliMi

Document Number/Rev. COMPLEX/PoliMi/R/D2.2.2/1.0

Classification COMPLEX Public

Submission Date 2012-01-09

Due Date 2011-11-30

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)


This document may be copied freely for use in the public domain. Sections of it may be copied provided that acknowledgement is given of this original work. No responsibility is assumed by COMPLEX or its members for any application or design, nor for any infringements of patents or rights of others which may result from the use of this document.
History of Changes

<table>
<thead>
<tr>
<th>ED.</th>
<th>REV.</th>
<th>DATE</th>
<th>PAGES</th>
<th>REASON FOR CHANGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>CB</td>
<td>1.0</td>
<td>2012-01-09</td>
<td>81</td>
<td>Final release version</td>
</tr>
</tbody>
</table>
## Contents

1. Scope of the document ................................................................. 6
2. Software Estimation Flow Overview ............................................. 7
   2.1 System-Level Software Modelling ........................................... 8
   2.2 Detailed Software Modelling ................................................ 9
   2.3 Task-based Virtual Platform Simulation .................................. 11
3. System-Level Software Modelling and Estimation ......................... 14
   3.1 Introduction ........................................................................ 14
   3.2 Features ........................................................................... 15
   3.3 Dependency on third-party tools .......................................... 15
   3.4 Integration ......................................................................... 16
   3.5 SCoPE+ Prototype ............................................................ 16
      3.5.1 Installation .................................................................. 16
      3.5.2 Execution of a simple example ..................................... 17
4. Detailed Software Modelling ....................................................... 20
   4.1 Target processor characterization ......................................... 21
      4.1.1 Theoretical foundation .................................................. 21
      4.1.2 Characterization flow .................................................... 23
   4.2 Estimation and back-annotation .......................................... 24
   4.3 Instrumentation and tracing ................................................. 26
Base Tools .................................................................................. 28
   4.3.1 swat-analyze .................................................................. 28
   4.3.2 swat-ba .......................................................................... 32
   4.3.3 swat-bbmodel ............................................................... 32
   4.3.4 swat-instrexpand ........................................................... 33
   4.3.5 swat-lnmodel ............................................................... 34
   4.3.6 swat-minstr ................................................................... 34
   4.3.7 swat-qq ........................................................................ 35
   4.3.8 swat-todyn ................................................................. 36
   4.3.9 swat-trp ........................................................................ 36
   4.3.10 swat-uniqid .................................................................. 38
4.4 Core Tools .............................................................................. 38
   4.4.1 swat-characterize ........................................................... 39
   4.4.2 swat-core-ba ............................................................... 40
   4.4.3 swat-core-tr ............................................................... 41
4.5 File formats ............................................................................ 42
   4.5.1 Configuration file format .................................................. 42
      4.5.1.1 Section [global] .......................................................... 42
      4.5.1.2 Section [project] .......................................................... 43
      4.5.1.3 Section [report] ........................................................... 43
      4.5.1.4 Section [target] ............................................................ 44
      4.5.1.5 Section [compilers] ...................................................... 44
      4.5.1.6 Section [cfg] ............................................................. 45
      4.5.1.7 Section [trace-<name>] ............................................. 45
   4.5.2 Model and rules formats ...................................................... 46
      4.5.2.1 Source code basic-block model .................................... 46
      4.5.2.2 LLVM instruction set model ....................................... 46
      4.5.2.3 Target processor instruction set model ......................... 47
4.5.2.4 Instrumentation expansion rules ................................................................. 47
4.5.2.5 LLVM meta-instrumented format ............................................................... 48
4.5.3 Output report formats ................................................................................. 48
  4.5.3.1 q101 - Basic-block cost ............................................................................ 49
  4.5.3.2 q102 - Basic-block size ........................................................................... 49
  4.5.3.3 q103 - Basic-block count ........................................................................ 49
  4.5.3.4 q110 - Selected basic-block plain .......................................................... 49
  4.5.3.5 q111 - Selected basic-block clustered ..................................................... 50
  4.5.3.6 q120 - Memory pressure per basic-block ............................................... 50
  4.5.3.7 q130 - Full forward control-flow graph ............................................... 50
  4.5.3.8 q131 - Full backward control-flow graph ............................................. 50
  4.5.3.9 q132 - Control-flow graph paths ............................................................ 51
  4.5.3.10 q133 - Control-flow graph loops ......................................................... 51
  4.5.3.11 q134 - Control-flow graph in-degrees and out-degrees ....................... 51
  4.5.3.12 q140 - Variable definitions and uses count ......................................... 51
  4.5.3.13 q201 - Function cost ............................................................................. 52
  4.5.3.14 q202 - Function size ............................................................................. 52
  4.5.3.15 q203 - Function arguments ................................................................. 52
  4.5.3.16 q210 - Selected function plain ............................................................. 52
  4.5.3.17 q220 - Memory pressure per function ............................................... 53
  4.5.3.18 q230 - Function stack size ................................................................. 53
  4.5.3.19 q240 - Instruction usage per function .................................................. 53
  4.5.3.20 q241 - Instruction class usage per function .......................................... 55
  4.5.3.21 q301 - Program cost ............................................................................. 56
  4.5.3.22 q310 - Instructions statistics ................................................................. 56
  4.5.3.23 q311 - Instruction classes statistics ...................................................... 56
  4.5.3.24 q320 - Inlining statistics per function ................................................... 56
  4.5.3.25 q321 - Inlining statistics per call point ................................................ 57
  4.5.3.26 q401 - Stack size dynamic bounds ...................................................... 57
  4.5.3.27 q901 - Basic-block-level metrics ........................................................ 57
  4.5.3.28 q902 - Function-level metrics ............................................................. 58
4.5.4 Output trace files formats ........................................................................... 58
  4.5.4.1 t801 - Basic-block id trace ................................................................. 58
  4.5.4.2 t802 - Function id trace ........................................................................ 59
  4.5.4.3 t803 - Function entry/exit trace ........................................................... 59
  4.5.4.4 t804 - Function entry/exit and basic-block id trace ............................. 60
  4.5.4.5 t805 - Function argument actual value trace ....................................... 60
4.6 Dependency on third-party tools ................................................................. 61
4.7 Integration ....................................................................................................... 61
5 Task-based Virtual Platform Simulation ......................................................... 62
  5.1 Overview ....................................................................................................... 62
  5.2 Functional modeling .................................................................................... 62
    5.2.1 Overview of SystemC modeling API for task-based functional models .... 62
    5.2.2 Generic Task Library .............................................................................. 65
    5.2.3 Table based Task Graph Description .................................................... 69
    5.2.4 Constraint modelling ............................................................................ 69
  5.3 Architecture modeling .................................................................................. 71
    5.3.1 Modeling API’s for abstract resources ................................................... 71
    5.3.2 Scheduling API’s and processing model ................................................. 73
  5.4 Power Modelling .......................................................................................... 76
5.4.1 Extensions for Power modelling ................................................................. 76
5.4.2 Integration with analysis infrastructure ......................................................... 77
5.5 Dependency on third-party tools ...................................................................... 78
5.6 Integration ........................................................................................................ 78
6 Summary ............................................................................................................ 79
7 References ......................................................................................................... 80
1 Scope of the document

This deliverable is the result from Task T2.2: Embedded software (Participants: PoliMi, UC, GMV, SNPS - Start: M4 - End: M24).

The main focus of this Task is source level modelling of the application software. The proposed approach consists of separating the modelling process in a static phase (related to the source code structure only), a dynamic and simulation based phase to collect profiling data and a post-processing phase leading to high-level lumped models to be used for design space exploration. A lightweight flow is also envisaged to support run-time management as described in Task T3.5.

UC, together with GMV, have modelled the embedded SW at system-level, before architectural mapping, exploring the impact of the allocation of a concrete concurrent resource to a certain platform component. SW performance, when running on a concrete processor are estimated using an abstract, high-level model of the processor and the RTOS.

Techniques have been developed ensuring fast and seamless model generation for different architectural mappings. PoliMi has developed a detailed source code model to derive energy and timing estimations related to the behaviour of the software components. The model has the primary goal of describing the software independently from the underlying executor, the compilation tool chain used and, most importantly, the external stimuli. This approach allows performing the complex and time-consuming modelling phase statically at compile-time and thus obtaining an instrumented version of the application to be used for simulation and profiling purposes. A post-processing phase, eventually, combines dynamic data with the elementary contributions derived lower-level models (microprocessor, ISA, buses, memories, etc.). Thanks to the accurate profiling data obtained during emulation/simulation, a precise relation with high-level constructs and a detailed knowledge of the underlying memory subsystem, the models and tools will isolate contributions due to the memory subsystem and its structure.

A reduced-complexity model and a lightweight instrumentation approach have also been studied in order to support run-time management of non-functional aspects.

Synopsys extended its SystemC environment with a prototype of an abstract source-code based simulation technology that allows creating task-based virtual platform simulations supporting functional and behavioural power estimation.

The main achievements to be documented in the present deliverable are:

1. Modelling of the embedded software at system-level, before architectural mapping (Section 2.3.1).

2. Development of source code model to derive energy and timing estimations related to the behaviour of the software components (Section 4).

3. Preliminary SystemC environment with a prototype of an abstract source code based simulation technology that allows creating task-based virtual platform simulations supporting functional and behavioural power estimation.

A short summary of the deliverable is provided in Section 5.
2 Software Estimation Flow Overview

This Section describes the overall software estimation flow, the methodologies and theoretical models behind it and the tools that have been implemented, extended or redesigned to support it. The COMPLEX view of the system (platform and application) encompasses both hardware and software components, treating them as homogeneously as possible. Figure 1 highlights the portions of the flow covered by this Deliverable. The excluded portions, though not treated in this document, are of utmost importance for the software modelling and estimation flow, since they provide models, characterization data and integration support to the tools and methodologies considered here.

Figure 1 – Software estimation flow within the COMPLEX framework.
2.1 System-Level Software Modelling

In COMPLEX, a new tool, SCoPE+ has been developed to support system-level performance estimation. The integration of SCoPE+ in the COMPLEX flow supports the system-level DSE cycle in COMPLEX. SCoPE+ enables the fast estimation of time and power consumption figures at a sufficient accuracy to enable early design decisions, such as finding the optimum platform and the optimum mapping of application components onto platform components. This way, system-level DSE enables a first level of decisions in the COMPLEX flow, which can be later refined in the Detailed Software Modelling.

For this, provided a minimum accuracy on the performance estimations, the focus has been put on speeding-up simulation and the exploration of different architectural solutions, since the goal is to find solutions after the search of a maximum set of points in the exploration space (ideally, the whole set) with a minimum effort. In order to get such speed, SCoPE+ relies on the following features:

1. **Native simulation.** The estimation is done by automatic and transparent instrumentation of the source code of the application and execution on the host machine.

2. **An implementation independent API for the description of the application,** which eliminates the need for manual refinement or adaptation of the code among different iterations of the exploration loop.

3. **Configurable executable specification and input formats supporting the definition of a Design Space, and of Design Space Point.** A configurable executable specification makes unnecessary the recompilation of a new executable specification for exploring a new solution. The executable specification already reflects a set of possible solutions. The SCoPE+ performance model supports a XML-based interface which facilitates the interaction with exploration tools and thus the automation of the design space exploration (DSE).

SCoPE+ takes as input the description of the application, the platform description and the mapping of the application onto the platform. SCoPE+ will present several possibilities for the specification of each of them:

1. **Application description:** Platform Independent Code (CFAM API) and under a RTOS API (POSIX, uC/OS-II, Win32), CFAM description

2. **Platform Description:** SystemC-like, XML, IP-XACT (HW)

3. **Mapping Description:** SystemC-like, XML

4. **Configuration Description:** XML

As an output, SCoPE+ will perform a functional validation, enabling a print-out through console at the different points where an application can pass. SCoPE+ dumps to the console (see Section 3.5.2) also performance figures for execution times, power consumption, and other ones (temperature, CPU load, etc), broken down at least for each CPU resource. SCoPE+ can also provide these figures in XML format for a set of requested output metrics (reported in D3.4.2).
2.2 Detailed Software Modelling

The detailed software modelling flow has the goal of producing execution time and energy consumption estimates of portions of the application being considered. The two key aspects of the methodology are the following:

1. **Decoupling of static and dynamic aspects.** The static aspects are captured by the *static model* of the application and are only related to its structure and functional behaviour. The dynamic aspects are captured by the *dynamic model* and are only influenced by data dependencies. They represent an instance (or the union of several instances) of execution of the application.

2. **Independence of the models from the target environment.** Both the static and dynamic models are dimensionless models (number of instructions, clock cycles, number of executions, number of calls, ...), and, as such, are independent from the actual execution environment of the application. The actual characteristics of the target platform are accounted for only in a post processing phase. To improve performance and simplify the estimation flow, it is though possible to anticipate the integration of target-dependent data into the static and dynamic models. This opportunity, though, is only to be regarded as an additional feature of the flow that does not affect its founding principles.

The input of the flow is constituted by:

1. **Configuration file.** This file specifies all the aspects of the flow and most of the command-line parameters to be passed to the different tools constituting the flow itself. The flow is, in fact, constituted by a number of tools integrated in a cascaded way. Each tool usually reads the outputs of the previous tool uses some of the options collected in this overall configuration file, and produces some output files, to be fed to the next tool in the chain. This arrangement – though not optimal in terms of processing time, due to extensive usage of files – allows easy modularization and maintenance of the individual tools.

2. **Sources.** The set of source files constituting all or part of the application and that are the object of the current analysis. These files must be ANSI C or ISO C99 files. Note that not all of the features of the ISO C99 standard have been tested.

3. **Extra sources.** The set of source file that are part of the application but are excluded from the analysis and estimation process. These files will be compiled and linked but neither modelled nor instrumented.

4. **Application libraries.** Some part of the application may be not available in the form of C source code but only as precompiled libraries (or, equivalently, as a set of object files). Such libraries must be compiled for the host machine where the execution of the model will take place. More complex arrangements using target-specific instruction set simulators are also possible, but are very specify for every environment.

5. **Target processor model.** A model summarizing the size, timing and energy characteristics of the target core. This model is at instruction-set level and is the result of the characterization phase. The level of accuracy of the model can vary from simple average values to very detailed instruction model.
6. **API model.** A model summarizing the relevant metrics for the functions that belong to the lowest firmware levels and are not part of the source-level estimation process. These functions typically are part of the HAL/BSP or the operating system.

7. **Device models.** A set of FSM-based models of the devices of the core. Such models are stimulated with execution traces made of sequences of “events”. An event plays the role of an FSM input, thus the set of possible events is the alphabet of the FSM. Upon receiving an event a model evolves by potentially changing its state and by “consuming” some time and energy. These FSM-based models do not need to mimic the real behaviour of the FSM describing the top-level hardware architecture of the device but only to account for its timing and energy behaviours.

The completeness and richness of the output of the flow can vary depending on the options specified in the configuration file. In its most complete form is constituted by:

1. **Overall estimates.** The total size, execution time and energy consumed by the application executed with the stimuli used for the analysis. This information is actually found in one of the report files described below.

2. **Detailed estimates.** Such size, time and energy estimates are split per basic-block, per source line or per function. This allows the developer to gain more and deeper insights into the structure of its application. Most notably, such reports are the basis for back-annotation.

3. **Back-annotated source code.** All source files that have been analysed are back-annotated with size, timing and energy estimates. Due to the C coding style of the developer and to the optimization process some costs cannot be associated to a specific source line. When in doubt, thus, such costs are either distributed uniformly to all non-zero cost lines of the function or are associated to the first line of the function itself. The overall cost of a function is guaranteed to include all and only the contributions of its body and the “hidden” contribution due to the instruction of the call/return sequences.

4. **Analysis reports.** A wide set of reports, known as q-files, summarizing static and dynamic information related to the application at different levels of granularity, namely: LLVM instruction level, basic-block level, function level, group of functions level and application level.

5. **Traces.** Traces, known as t-files, are special (usually quite large) forms of reports where information is not lumped onto basic-blocks, functions, etc. but is rather seen over time. Traces can be “primary”, i.e. directly produced by the execution of the application thanks to the instrumentation, or “derived”, that is the result of a post-processing phase on a primary trace.

6. **Report.** A rich and detailed HTML report summarizing most of the outputs and analyses produced by the tools. Several results are also presented in graphical form.

The model, the structure of the flow, the individual tools, the file formats and the command-line options of each tool are presented in Section 4.
2.3 Task-based Virtual Platform Simulation

2.3.1 Goal

The term task modeling refers to the modeling of an application as a set of tasks that can communicate with other tasks. This model of the application can be either a functional or a nonfunctional model. The goal of the model is to capture the properties of the application, such as the amount of processing time required to execute the task and the amount of data that is required to be transferred to perform certain operations. Typically, you start with a nonfunctional model to capture the performance-related properties with minimal effort. Over time, the model can grow into a fully functional model.

Once a task graph of an application has been created, it can be executed on its own or it can be mapped unto one or more Virtual Processing Units (VPUs) in a hardware system. The goal of making a model of your application is to do performance analysis. You can consider the following kinds of analysis:

- The load of the application on the processing element. Can the application be mapped on one processing element or does it require to be split over multiple? The behavior of the software and the load of the processing elements can be analyzed.

- When the application is mapped on a VPU, the complete model (VPU and software) approximates the traffic that would be generated by a real processor executing real software. This allows you to measure the load of the application on the hardware system and it allows finetuning of the hardware system (mainly load of the bus and memory subsystem). So the software and the VPU serve as a workload model to make hardware trade-offs or to optimize the hardware.

2.3.2 Concept

When building a task graph of an application, the first step is to split up the application in tasks. This is done by extracting the parallelism from the application. Each piece of the application that can run separately is modeled as a task. At the same time, you have to determine what the inputs and outputs of the tasks are. Tasks receive their inputs and send their outputs over communication channels.

The individual tasks can be fully functional or completely nonfunctional. Nonfunctional tasks are modelled very quickly, that is, you just annotate the time or the amount of cycles which the processing that would normally be done would take. These numbers are either obtained from previous experience or measurements or are estimates of what the processing should take. In the latter case, you can see these estimates as “budgets” that will be assigned to the software which needs to be developed. When the tasks are modeled, they can be instantiated and connected to create the task graph.
2.3.3 Methodology

As shown in this figure, the task flow consists of the following steps:

1. Starting from reference code or from a data-flow graph, create a nonfunctional task graph.

2. You can refine this task graph: Improve the annotations, evaluate the communication between tasks (for example, variable versus FIFO communication), split up tasks in smaller tasks, and so on. At any point in time, you can map this task graph to a platform (on one or more VPU blocks).

3. Execute the task graph mapped to a platform. In this phase, you investigate the load the tasks are causing on the processing elements and the load the application is causing on the platform. In this phase, you can analyze the bus and memory architecture of the platform to see if it supports the application in all circumstances.

4. Find a mapping of the application on the fine-tuned platform. Both the application and the platform can go to implementation now.

2.3.4 Framework overview

In order to create a task graph of an application the Task Modeling API is available. This is a modeling API that sits on top of the SystemC and TLM modeling standards. The benefit of this approach is that a tight integration with platform modeling becomes possible.

The individual tasks can communicate with other tasks through regular SystemC ports and channels. When all the tasks are connected, you have a task graph. This task graph can be executed standalone, using the default task manager. A task manager controls the execution of the different tasks in the task graph. A component of the task manager is the scheduler, which determines which task should be run next. During execution, the tasks annotate the time it takes to execute by using consume statements. These consume statements are converted into
simulation time by a processing model, which is a plug-in for the task manager. The task manager can be customized with a user-defined scheduling algorithm and a user-defined processing model. In a stand-alone scenario, the system looks as in Figure 2.

![Figure 2](image2.jpg)

Figure 2 – System view in a stand-alone scenario.

A task graph can also be mapped on one or more VPUs (Virtual Processing Units). When communicating tasks are split over multiple VPUs, the original connection has to be refined. Instead of a direct connection, the communication needs to happen over the hardware of the system. The original channel is replaced by a driver on each VPU which takes care of the hardware communication and synchronization. The processing model can now also add additional communication to reflect the load/store activity of the processor. The overall picture looks as shown in Figure 3.

![Figure 3](image3.jpg)

Figure 3 – Overall system view.
3 System-Level Software Modelling and Estimation

3.1 Introduction

Embedded design flows usually start developing platform-independent codes, containing the system functionality. Once the functionality of these codes has been verified, the implementation of these codes into the target platform is considered. First, the designers have to decide the resource allocation for those platform independent codes. After that, the codes are refined depending on the allocation decided.

Decisions on resource allocation have large impact on system performance. To optimize the decision process, designers require estimations of the performance that can be obtained with the different allocations. Estimations of the performance of the possible SW allocations are then very important, since the majority of the system functionality is commonly implemented in SW. This is especially important in the COMPLEX project, where a SW oriented approach has been defined as one of its initial characteristics. Thus, in task 2.2 solutions for modelling and estimating the effect of SW allocations of functional components have been developed. The goal is to analyze the original system-level code working as SW without requiring any manual recoding. Then the estimations obtained have to be integrated in the SCoPE+ system simulation environment, developed in the COMPLEX project as an evolution of the original SCoPE tool.

In order to integrate SW performance estimations in the design process, the estimation process must be fast and easy to adapt to the different possible allocations. Thus, the estimation process must be able to automatically evaluate the initial functional models developed following the specifications defined in WP1, that is, without manual modifications or intervention.

Commonly, these estimations are obtained using simulation techniques. However, simulations based on the execution of cross-compiled binary code, such as ISSs and binary translators, require code completely refined for the target platform. Application SW must be recoded using the target operating system API. Additionally, communications between components mapped to HW and SW requires SW drivers to perform HW/SW communications.

Thus, in order to optimize design flows, it is required a solution capable of estimating time and power consumption of initial designs with no SW recoding efforts. Previous native technologies, such as SCoPE, minimize the recoding efforts, since they can simulate partially refined SW codes. However, some effort for adapting the original codes to the operating system API provided is required. Additionally, HW/SW communication still depends on drivers.

To overcome that, the tool SCoPE+ is being developed with the capability of integrating the simulation and estimation of SW components performance from initial, platform-independent codes. Then, the simulation and estimation of platform-independent codes performed by the SCoPE+ tool enable evaluating the SW effects resulting from different HW/SW partitioning and resource allocations at the beginning of the design process, with no additional effort. As a result, functional components can be directly simulated as SW components, since no differences are required between them by the simulation engine.
A platform-independent entry has been developed to enable the use of the SCoPE+ tool in different environments. This entry is based on the CFAM infrastructure, defined in WP1, and developed in T2.2 and T2.5. The use of this entry also enables the automatic simulation and exploration of models developed in UML/MARTE following the specification defined in WP1. For that purpose several generators have been developed in WP1, generators that automatically create CFAM compatible files from the UML/MARTE models.

### 3.2 Features

To enable simulating and evaluating SW allocations of platform-independent codes in different resources, different features for enhancing SW modeling have been added in task T2.2. These are the following ones:

- Component-based Platform Independent front end for System Functionality (CFACM and CFAM APIs)
- Improvement of application-code performance estimation
- Modelling of SW/SW communications
- System-Level modelling of multi-OS execution
- System-Level modelling of HW/SW communication
- Multilevel simulation
- Component Traversing Flows

These features have been integrated within the SCoPE+ simulation engine, developed in T3.1. In D3.1.1, a technical explanation of how the simulation engine has been enhanced for supporting the simulation of these features is reported. These features are introduced in the confidential version of this document.

### 3.3 Dependency on third-party tools

SCoPE+ requires two external elements to enable the simulation and evaluation of the application SW components:

- Linux/Unix 32 bits platform
- GNU C/C++ compiler > 4.0
- SystemC 2.2.0
- flex 2.5.33
- bison (GNU bison) 2.3

All this tooling is open-source, thus no external limitations are imposed to the use of the SCoPE+ tool.
3.4 Integration

Additionally, in the COMPLEX design flow, other tools will directly interact with SCoPE+. Strictly speaking, these tools are not required to use SCoPE+ (such requirements have been already reported in previous section) for estimating the performance of a CFAM implementation independent code over a specific architecture.

However, the synergic cooperation of SCoPE+ with these tool is what better exploits the SCoPE+ capabilities for building a high-level DSE environment. These tools are the following ones

- Eclipse + Papyrus: for generating the UML/MARTE models
- COMPLEX Eclipse Plugin (on top of Eclipse + Acceleo): Includes the generators developed in WP2 (see D2.2.2), which produce, from the UML/MARTE model, the CFAM code which serve as input to SCoPE+
- MOST: Required to select the different configurations to be estimated and to analyze the output results in order to select the optimal configurations

Additionally, in COMPLEX, a suitable integration of SCoPE+ in the system-level DSE cycle is done by employing the following interfaces:

- For the generation, from the COMPLEX UML/MARTE specification [D2.1.1] and code sources, of the configurable executable specification, with the possibility of reflecting a Design Space, comprising a set of platforms and possible mappings:
  2. Platform and mapping Description: XML System Description (XML SD) and XML Design Space (XML DS).
- For establishing the specific platform and mapping, from the exploration tool:
  3. Platform and mapping Configuration: XML Description (see D3.4).

3.5 SCoPE+ Prototype

At M24, a distribution of SCoPE+ which contains all the features required for SW performance estimation has been released. It also integrates additional features, which regard to the integration task (T2.5), namely custom HW estimation and the integration of SystemC stimuli. In D1.3.1 the status of the tool and of the different features is reported. In D2.5.2, details which regard to the integration of the SW estimation features of SCoPE+ within the COMPLEX framework are given.

Following subsections give a quick manual about how to install, run a simple example with SCoPE+ and observe performance estimation results.

3.5.1 Installation

SCoPE+ release is composed of two packages:
• An improved SCoPE simulation infrastructure (SCoPE2.0Beta6)
• The SCoPE+ plugin that implements the CFAM layer (CFAM v0.2.6) and other additional components

In order to install SCoPE 2.0 from the sources it is required to:

• Download the corresponding tar file (SCoPE-2.0.0beta6.tgz)
• Uncompress it
• Update the environment variable $SCOPE_HOME with the installation path
• Update the PATH environment variable with the following command:

```
export PATH=$PATH:$SCOPE_HOME/bin
```
• Update the LD_LIBRARY_PATH environment variable with

```
export LD_LIBRARY_PATH=
	$LD_LIBRARY_PATH:$SCOPE_HOME/lib
```
• Execute `make` in the main folder of the uncompressed tree

To install the SCoPE+ plugin:

• Download the corresponding tar file (CFACM-v0.2.6.tgz) from the web site
• Uncompress it
• Update the environment variable $CFAM_HOME with the installation path
• Execute `make` in the main folder of the uncompressed tree

### 3.5.2 Execution of a simple example

To initially check the execution of the SCoPE+ tool, a toy example has been developed and delivered with SCoPE+. GMV and UC developed in UML/MARTE example. The architecture of the toy example is shown the next figure. The toy example is a simplified version of the Use Case 3 example (the SSA system).
The example was translated to the CFAM API. Indeed, this example supported the development and check of the SCoPE+ CFAM plug in. Thus, initially, the translation was manual, and, after the delivery of the transformation tools reported in D2.2.2, it can be done automatically through MARTE2CFAM transformation tool of the COMPLEX eclipse plug in.

To execute the toy example in the SCoPE+ tool, only the following steps:

- Move to the exampled directory:

  \texttt{cd \$CFAM\_HOME/examples/example\_rpc}

- Create the executable performance model:

  \texttt{make scope}

- Run the example:

  \texttt{make run\_scope}

As a result the output of the example is shown in the console:
Figure 5 – Toy example output (track of functional execution).

After the end of the simulation, the estimated performance figures are dumped.

```
ATOS: 0

Number of new processes created: 7
Number of new processes destroyed: 6
Mean process duration (process start - process end): 0.142867 sec
Last SW execution time: 1 sec

Process PID: 4
  Thread TID: 5, name: main_func, User time: 0 ns
  Thread TID: 24, name: component_executable_function, User time: 838337 ns
  Thread TID: 25, name: component_executable_function, User time: 568364 ns

Process PID: 6
  Thread TID: 7, name: cfam_default_system_creation, User time: 8 ns

Process PID: 8
  Thread TID: 9, name: cfam_default_system_creation, User time: 8 ns

Process PID: 10
  Thread TID: 11, name: cfam_default_system_creation, User time: 8 ns

Process PID: 12
  Thread TID: 13, name: cfam_default_system_creation, User time: 8 ns

Process PID: 14
  Thread TID: 15, name: cfam_default_system_creation, User time: 8 ns

Process PID: 16
  Thread TID: 17, name: cfam_default_system_creation, User time: 8 ns

Total User time: 0.00142475 sec
Total Kernel time: 0.00028584 sec

processor Processor 0_0_0
  Number of thread switches: 244
  Number of context switches: 83
  Running time: 1376944 ns
  Use of cpu: 8.1376944
  Instructions executed: 157655
  Instruction cache misses: 270
  Data cache hits: 71791
  Data cache misses: 1
  Data cache write backs: 0
  Core Energy: 315312 nJ
  Core Power: 0.315312 mW
```

Figure 6 – Toy example output (Global performance figures).
4 Detailed Software Modelling

The detailed software modelling methodology and flow has the primary goal of providing estimations of the size, execution time and energy consumption of a given application for a specific target core processor.

As outlined, the estimation process is split into different phases, schematically shown in Figure 7 and described in the following.

---

**Target processor characterization.** This phase has the goal of providing a simple static yet accurate model of the behaviour of the target processor. This model allows linking the abstract, target-independent, source-level model of the application to the specific executor. It is important noting that this phase needs to be executed only once per each target processor being considered.

**Source-level static modelling.** This step builds a data-independent, static model of the source code based on an intermediate representation expressed in the form of a pseudo-assembly LLVM code. Such a model is then transformed into a simplified representation of the characteristics of single basic blocks. Conceptually, such models are totally decoupled from the specific characteristics of the target processor. In practice, though, target executor information is combined with the code model in this phase. Static modelling does not depend on data and thus must be executed only once per each application.
Application dynamic modelling. The dynamic modelling phase, in its simplest form, collects profiling information at basic block and/or function levels. Clearly, the dynamic phase depends on the actual data fed to the application.

Analysis and postprocessing. This phase concludes the estimation flow by combining static source code models with dynamic information. The output are size, execution time and energy estimates of the target application at different levels of abstraction, namely, basic block, source code line, function and entire application. In addition to these overall figures, a set of analyses on the static and dynamic structure of the code is preformed to derive a detailed characterization of the application.

4.1 Target processor characterization

The target processor characterization flow has the goal of associating size, execution time and energy consumption costs to the elementary entities constituting the static source code model, that the LLVM instructions.

4.1.1 Theoretical foundation

When a given function – or, in general, a given source code – is translated into LLVM code it will result in a set of basic blocks in turn constituted by a sequence of LLVM instructions. Since a basic block is either executed completely or not executed at all, its cost merely equals the sum of the costs of individual LLVM instructions. Such base cost of LLVM instructions not only depend on the target processor but also on the behaviour of the target compiler. The characterization phase accounts for both these dependencies by deriving a statistical relation that links the result of compilation for the LLVM virtual machine with the result of compilation for the target processor.

Consider as an example a simple LLVM instruction such as:

%2 = add i32 %1, -1

that adds the constant -1 to the 32-bit integer virtual register named %1 and stores the result into the new virtual register named %2. Depending on the context where this instruction appears and on the specific target instruction set, the same operation might be rendered for the target architecture in different ways. For example, if the variable corresponding to virtual register %1 is not yet stored into one of the general purpose register of the target processor but is rather available on the stack, the code might look like:

LDD 8,SP  // Load %1 from stack into data register D
ADDD $-1
STD 10,SP  // Stores %2 back on the stack

If, on the other hand, the virtual variable %1 were already (due to previous calculation) available in a register while %2 is not reused for some time, the code might have the form:

ADDD $-1
STD 10,SP  // Stores %2 back on the stack

Yet another possible translation, in case %2 can be left in register D for future reuse, is:
The target compiler might possibly decide to optimize this instruction by translating it as:

```plaintext
SUBD $1
```

or, with a further optimization, as:

```plaintext
DECD // Decrements register D
```

Other possible translation might require moving the variable `%1` from a different register, say `Y`, to the accumulator `D` before performing the actual operation, as in:

```plaintext
XGDY // Swaps D with Y
DECD // Decrements register D
```

Of course it is possible to continue with similar examples. In conclusion, it is impossible to deterministically know how a specific LLVM instruction will be translated into target assembly code without knowing the context and the precise behaviour of code generation and assembly-level optimization algorithms of the target compiler.

Our proposal and flow are based on the idea of statistically characterize such a complex translation process. As a result we might conclude, for example, that the LLVM `ADD` instruction is, on average, translated with:

- `0.973 ADDD`
- `0.142 LDD`
- `0.121 STD`
- `0.002 INC`
- `0.001 XGDY`

Such a statistical characterization must be performed on a very large training set of source codes, so that we can reasonably guarantee that the translation and optimization algorithms are stimulated as exhaustively as possible.

The process of characterization, thus, starts from a large set of `N` source codes (more than 2,000 source code, for an overall line of code count of more than 200,000) and translates each source code `S_i` both into its LLVM representation `A_{L,i}` and its target assembly code `A_{T,i}`. Now let `L_{i,j}` be the number of LLVM instructions of type `j` present in the code `A_{L,i}` and `T_{i,k}` the number of target instructions of type `k` present in the target code `A_{T,i}`. Based on these counts we can now build a set of equations for each LLVM instruction, each equation of the set being derived from one of the source codes in the training set:

```plaintext
L_{i,j} = \sum_{i=1}^{K} D_{i,k} \cdot x_{j,k}
```

This is a set of `N` equations in the `K` unknowns `x_{j,k}` that can be written in matrix form as:

```plaintext
L_j = D \times x_j
```
where \( \mathbf{L}_j \) is the vector of LLVM instruction counts, \( \mathbf{D} \) is the target instruction count (\( N \) rows, one per each source) and \( \mathbf{x}_j \) is the \( K \times 1 \) vector expressing the model of LLVM instruction of type \( j \) in terms of the instructions of the target processor.

This formulation, though conceptually correct, assumes that a generic LLVM instruction may be translated with a combination of whatever target instructions. This assumption, in practice, is too general and leads to a scarcely significant solution. A more realistic model should consider the fact a certain LLVM instruction is likely to be translated by using a subset of all target instructions. For example, it is reasonable to suppose that the LLVM ADD instruction will never be translated using call, return or branch instructions of the target assembly.

In our formulation this can be accounted for by defining a model vector \( \mathbf{M}_j \) of binary coefficients \( m_{j,k} \) with the following meaning: if \( m_{j,k} = 1 \) then the target instruction of type \( k \) may be used to translate LLVM instruction of type \( j \), while if \( m_{j,k} = 0 \) then target instruction of type \( k \) will never be used to translate LLVM instruction of type \( j \). Of course, the model vectors must be defined a priori and provided as input to the problem.

With this extension, the new problem can be formulated as:

\[
\mathbf{L}_j = \mathbf{D} \times \text{diag}(\mathbf{M}_j) \times \mathbf{x}_j
\]

Finally, we have decided to constrain the coefficients \( \mathbf{x}_j \) to have non-negative values, since they would have a difficult physical interpretation. This leads to the final problem formulation in the form of a bound-constrained least square problem, namely:

\[
\left\| \mathbf{D} \times \text{diag}(\mathbf{M}_j) \times \mathbf{x}_j - \mathbf{L}_j \right\|_2^2 \quad \text{with} \quad \mathbf{x}_j \geq 0 \quad \forall j
\]

By composing the \( L \) vectors \( \mathbf{x}_j \) into an \( L \times K \) matrix \( \mathbf{Z} \) defined as:

\[
\mathbf{Z} = \begin{bmatrix}
\mathbf{x}_1 \\
\vdots \\
\mathbf{x}_L
\end{bmatrix}
\]

we obtain the so-called translation matrix, that allows to statistically correlate any LLVM program to its corresponding target assembly program.

### 4.1.2 Characterization flow

The characterization flow, depicted in Figure 8 takes as input the training set of source codes and the set of model vectors \( \mathbf{M}_j \) and compiles each source into LLVM and target assembly codes. Then the number of LLVM instructions and target instructions are counted and the resulting counts are used to construct the least square problem that is then solved with a suitable algorithm to produce the desired model.

The non-negative least square problem is solved using the Lawson–Hanson method implemented described in [26].
Figure 8 – Target processor characterization flow.

The flow is implemented by the tool swat-characterize, whose command line interface is described in Section 4.4.1.

4.2 Estimation and back-annotation

The estimation and back-annotation flow implements the core functionalities of the estimation methodology. Such flow is implemented by the core tool swat-core-ba whose usage is described in Section 4.4.2. The theoretical foundation of this flow have already been completely defined and summarized in Deliverable D2.2.1.

The flow is structured as schematically depicted in Figure 9. The first step reads the input C source files and, using the LLVM compiler, generates the initial pseudo-assembly LLVM representation store in .ll files. Using these files the tools swat-bbmodel and swat-uniqid build the basic-block level model of the application (see Sections 4.3.3 and 4.3.10). The former tool produces a compact representation of the relevant information concerning each single basic-block, while the latter assigns a application-wide unique identifier to all functions and all basic-block. Such identifiers are necessary to disambiguate references to functions and files having the same name and to speed-up all look-up processes necessary throughout the flow. The output is a set of .bbmodel files and the two name/id map files with suffixes .fnmap, for functions, and .bbmap for basic-blocks. The information collected in the basic-block model files account for the static, structural aspect of the application only but include target related informations, such as LLVM instruction base costs. Such information is collected from the target CPU model library.
Using the information in the basic-block models and starting from the pseudo assembly LLVM files, the subsequent step performs meta-instrumentation. This is done using the `swat-minstr` tool, an LLVM assembly parser and transformation tool built on top of the LLVM framework (see Section 4.3.6). The output `.mi.ll` file is again an LLVM symbolic assembly program, augmented with special comments holding all the necessary information concerning basic-blocks, function calls and returns, function arguments and other global information. Such data actually represents a fusion of the static source model with the target processor model.

![Diagram](diagram.png)

Figure 9 – Estimation and back-annotation flow.

The next step consists in generating the actual instrumented LLVM assembly code, saved in the `.i.ll` files. This process, performed by the tool `swat-instrexpand` is basically a sort of rule-based macro-expansion of the special comments into suitable function calls (see Section 4.3.4). The functions to be used for the required tracing process are specified in the specific rule file and are implemented in a binary support library being part of the SWAT distribution. This mechanism allows a seamless customization and extension of the tracing capabilities of the framework. A custom trace can in fact be generated by defining a new expansion rule file and by implementing the required trace functions.

All the instrumented LLVM files are then compiled to the host assembly language and assembled into object files. Furthermore, all the source files excluded from the analysis but being part of the application are compiled directly into object files on the host machine. The overall set of object files is then linked, together with the tracing support library to produce the `a.out` executable.
This executable is then run on the host machine to produce the execution trace. Execution requires preparing the environment and passing the necessary command line arguments to the application. This is done directly by the `swat-core-ba` tool.

The raw trace – collecting execution information over time – is then “folded” over the static model by calculating the execution counts of each basic-block during the whole application run. This phase is performed by the tool `swat-trp`, which is a general trace processor that can be configured to perform different kinds of analyses (see Section 0).

The last phase of the estimation process consists in combining the static models in the `.bbmodel` files with the dynamic profile of the application. This is performed by the `swat-todyn` tool that does not create new dynamic model files but rather enriches the static `.bbmodel` files with all the dynamic information (see Section 4.3.8). Dynamic figures are then organized on a per-function basis and collected into a report by the general analysis tool `swat-analyze` (see Section 4.3.1).

To perform back-annotation, two more steps are necessary. First of all the basic-block models have to be split and recombined in order to reorganize dynamic figures per line of source code. This is done by the tool `swat-lnmodel` that produces `.lnmodel` files (see Section 4.3.5). It is worth noting that, while the overall function costs are always correct, the exact attribution of a cost to a specific source code line is strongly influenced by the optimizations performed by the compiler. Using higher the optimization levels is more likely to produce a less accurate attribution of costs to the source code.

Once all the per-line costs have been calculated, the tool `swat-ba` simply produces an annotated `.ba` source file, that is the original source file in which each line is prefixed with one, two or three columns indicating the corresponding size, execution time and energy consumption (see Section 4.3.2).

This operation concludes the estimation and back-annotation process.

### 4.3 Instrumentation and tracing

The instrumentation and tracing flow, schematically shown in Figure 10, shares a significant portion with the estimation and back-annotation flow, namely the front-end that receives as input the source files and produces as main outputs the instrumented LLVM code and the basic-block models. For a detailed description of this portion of the flow see Section 4.2.

The output of the front-end is thus constituted essentially by the instrumented `.i.ll` LLVM assembly files. From this point on, the flow can perform different operations depending on what is requested by the user. In all cases the first step consists in compiling the LLVM input files into host assembly and then into object files.
In a first scenario, these object files can be the final output of the instrumentation flow. Alternatively, is a similar way, the output might as well be a library built linking all the objects (and possibly additional objects and libraries) into a single library.

This are the cases, for example, when the user wants to links such files within a SystemC simulation framework or wants to combine the instrumented application with a specific simulation support framework.

A similar scenario is that of the integration of an augmented source model within the BAC++ simulation and estimation framework. The back-end portion of the flow for this kind of scenario is depicted in Figure 11.
In this case, since the application and the tracing support library are developed in C, it is necessary to provide a wrapper encapsulating C++ method calls within C functions. This arrangement – that guarantees maximum flexibility and independence between the two frameworks – is schematically depicted in Figure 12.

The final output of the flow may consist either in a set of instrumented object files or in a single library collecting all such files.

### Base Tools

#### 4.3.1 swat-analyze

This tool provides a large and extensible set of analysis functions.

**Synopsys**

```bash
swat-analyze <options> <files>
```

**Options**

- `-help`
  
  Prints a short description of the tool options.

- `-version`
  
  Prints the tool version.

- `-swat-debug`
  
  Produces a verbose debugging output of the execution.

- `-report`
  
  Produces the analysis output in human-readable tabular format. Such formats are essentially a formatted version of the output files whose format and contents are described in Section 4.5.3.

- `-aa-classes`
Calculates the statistics of the static and dynamic usage of LLVM instructions classes. The classes considered are ialu (integer arithmetic and logic operations), falu (floating-point arithmetic and logic operations), ldst (load/store operations), flow (flow-control instructions), conv (integer/floating-point/pointer conversion and sign extensions), othr (other instructions).

-aa-cost
Overall cost statistics. This analysis calculates the overall execution time and energy consumption of the whole application.

-aa-inline [{ -compact | -detailed }]
Function inlining analysis. This analysis counts static and dynamic calls of all the functions of the project and produces a report indicating those functions whose inlining is likely to produce a higher reduction of execution time and energy saving. The analysis can be performed at two different levels of granularity: compact and detailed. The -compact options produces a result where all call-points of a given function are considered together, while the -detailed option produces individual statistics for each call-point of a given functions. This analysis allows not only supporting the decision whether or not to inline a function, but also which specific calls to operate on. This, in turn, enables simultaneous size and execution time optimizations.

-aa-insn
Calculates the statistics of the static and dynamic usage of LLVM instructions.

-backward
Backward CFG analysis. See -bb-cfg.

-bb-cfg [{-forward|-backward|-degrees|-loops|-paths}]
Extracts the basic-block control flow graph of a function. This is the core analysis supporting more detailed analyses, enabled by the different sub-options. The basic analysis produces a list of basic-block pairs. Each pair represents a couple of CFG nodes connected by a CFG edge. The basic-block are listed in the order predecessor-successor in the case of a forward analysis (-forward sub-option) or in the order successor-predecessor in the case of a reverse analysis (-backward sub-option. The in-degree and out-degree of each CFG node can be calculated using the -degrees sub-option. Finally, all the paths and all the loops in a function’s CFG can be extracted by specifying the -paths and/or the -loops sub-options respectively. It is worth noting that tool only considers paths beginning at the function’s unique entry point and terminating at the function’s unique exit point.

-bb-cost
Cost statistics at basic-block level.

-bb-count
Basic-block count per functions.

-bb-defuse
Variable definitions and usage statistics at basic-block level.
-bb-mempressure
Calculates memory pressure statistics at basic-block level. The memory pressure of a basic-block is defined as the ratio between the static number of memory access operations and the overall number of LLVM instructions.

-bb-select -threshold <percent> [{ -plain | -cluster }]
Filters and sorts basic-blocks whose cumulated relative cost is greater than the threshold specified with the sub-option -threshold. The plain selection mechanism (-plain sub-option) selects single basic-blocks, while the clustered selection mechanisms (-cluster sub-option) considers pairs of basic-blocks that are adjacent in the CFG.

-bb-size
Computes basic-block size statistics, in terms of number of instructions.

-bbmap <file>
Specifies the basic-blocks map file. This is necessary for some of the analyses.

-cluster
Cluster-based basic-block selection. See -bb-select option.

-compact
Compact inlining analysis. See -aa-inline option.

-degrees
Calculates in- and out-degrees of nodes in CFG. See -bb-cfg option.

-detailed
Detailed inlining analysis. See -aa-inline option.

-fn-classes
Calculates the statistics of the static and dynamic usage of LLVM instructions classes, per each individual function. The classes considered are ialu (integer arithmetic and logic operations), falu (floating-point arithmetic and logic operations), ldst (load/store operations), flow (flow-control instructions), conv (integer/floating-point/pointer conversion and sign extensions), othr (other instructions).

-fn-cost
Cost statistics at function level.

-fn_insn
Calculates the statistics of the static and dynamic usage of LLVM instructions per each individual function.

-fn-mempressure
Calculates memory pressure statistics at function level. The memory pressure of a function is defined as the ratio between the static number of memory access operations and the overall number of LLVM instructions.
-fn-select -threshold <percent>
Filters and sorts functions whose cumulated relative cost is greater than the threshold specified with the sub-option -threshold.

-fn-size
Function size statistics. This analysis calculates the number of basic-blocks of each function and the minimum, average and maximum size of the basic-blocks.

-fn-stack-size -frame-size <bits> -types-size <string>
Calculates the stack size of each function. To this purpose the activation frame size in bits must be specified with the sub-option -frame-size. To perform this calculation, the user must provide information about the number of bits used by the architecture to store the basic data types of the C language. The string specified with the -types-size sub-option carries this information and is a colon-separated list of integers corresponding to the following types:

- char
- short int
- int / long int
- long long int
- float
- double
- long double
- pointer

-fnmap <file>
Specifies the function map file. This is necessary for some of the analyses.

-forward
Forward CFG analysis. See -bb-cfg option.

-frame-size <bits>
Activation frame size in bits. See -fn-stack-size option.

-loops
Finds all loops from CFG analysis. See -bb-cfg option.

-paths
Finds all paths from CFG analysis. See -bb-cfg option.

-plain
Plain basic-block selection. See -bb-select option.

-threshold <percent>
Selection threshold. See -bb-select and -fn-select options.

-types-size <string>
Processor data types size. See -stack-size option.

Files

$PROJECT/*.bbmodel
Source model files of the application under analysis.

$PROJECT/*.ll
LLVM assembly files of the application under analysis.

$\text{PROJECT}/*\text{.bbmap}$

Basic-block id map file.

$\text{PROJECT}/*\text{.bbmodel}$

Function id map file.

4.3.2 swat-ba

Performs size, timing and energy figures back-annotation onto the C source code. The output is the original C source file where each line is prefixed by three columns of numeric data.

Synopsys

`swat-ba <options> <file>`

Options

- `-help`
  
  Prints a short description of the tool options.
  
- `-version`
  
  Prints the tool version.

- `-format <fmt>`
  
  Specifies the format to be used to annotate numeric data in the C source file. The format is specified according to the printf syntax.

- `-o <file>`
  
  The output back-annotated file.

Files

$\text{PROJECT}/*\text{.c}$

Source file of the application under analysis.

$\text{PROJECT}/*\text{.llmodel}$

Corresponding source model file of the application under analysis.

4.3.3 swat-bbmodel

Builds the basic-block model of a source file starting from its LLVM representation.

Synopsys

`swat-bbmodel <options> <file>`

Options

- `-help`
  
  Prints a short description of the tool options.
-version
   Prints the tool version.

-cpu <file>
   Specifies the CPU model file.

-exclude-functions <file>
   Specifies the file containing a list of functions to be excluded from the analysis.

-exclude-opcodes <file>
   Specifies the file containing a list of the excluded LLVM op-codes. The model will be built ignoring these op-codes.

-fn-args
   Includes in the model the list of function arguments of each call instruction.

-o <file>
   The output model file.

Files

$PROJECT/*\.ll
   LLVM file to be modeled.

$SWAT_ROOT/etc/cpu/*/cpu
   CPU model file of the target architecture.

4.3.4 swat-instrexpand

Performs rule-based expansion of a meta-instrumented LLVM file.

Synopsys

   swat-instrexpand <options> <file>

Options

   -help
      Prints a short description of the tool options.

   -rules <file>
      Specifies the instrumentation expansion rule file.

   -version
      Prints the tool version.

Files

$PROJECT/*\.mi\.ll
   Meta-instrumented LLVM file to be processed.

$SWAT_ROOT/etc/rules/*/rules
Instrumentation expansion rule file. See Section 4.5.2.4 for details.

4.3.5  swat-lnmodel

Transforms a basic-block model file into a source-line based model file. This files are used to build the back-annotated source files.

**Synopsys**

```
swat-lnmodel <options> <file>
```

**Options**

- `-help`
  
  Prints a short description of the tool options.

- `-o <file>`
  
  The output model file.

- `-version`
  
  Prints the tool version.

**Files**

- `$PROJECT/* .mi.ll`
  
  Meta-instrumented LLVM file to be processed.

- `$SWAT_ROOT/etc/rules/*.rules`
  
  Instrumentation expansion rule file. See Section 4.5.2.4 for details.

4.3.6  swat-minstr

Performs LLVM assembly meta-instrumentation. The generated output is a new LLVM file that is functionally equivalent to the input file and is augmented by special comments.

**Synopsys**

```
swat-minstr <options> <file>
```

**Options**

- `-help`
  
  Prints a short description of the tool options.

- `-bb-model <file>`
  
  Basic block model file name

- `-fn-filter-in <file>`
  
  Name of a file containing a list of functions to be included in the meta-instrumentation process. If this option is specified, only the functions listed will be processed.

- `-fn-filter-out <file>`
Name of a file containing a list of functions to be excluded from the meta-
instrumentation process. If this option is specified, all but the functions listed will be
processed.

```
-fnmap <file>
```
Functions id map file.

```
-o <file>
```
The output model file.

```
-version
```
Prints the tool version.

Files

```
$PROJECT/*.[lL]l
```
LLVM file to be processed.

```
$PROJECT/*.[bB]bmodel
```
Source model file associated to the LLVM file to be processed.

### 4.3.7 swat-qq

Performs all the available analyses on the application and collects the results in two output
files (see Sections 4.5.3.27 and 4.5.3.28) and optionally generates an HTML graphical report.

**Synopsys**

```
swat-qq <options>
```

**Options**

```
-help
```
Prints a short description of the tool options.

```
-config <file>
```
The configuration file. See Section 4.5.1 for details.

```
-info
```
Scans the current directory looking for report files and prints a report of the available
reports indicating the nature of their content.

```
-swat-debug
```
Produces a verbose debugging output of the execution.

```
-version
```
Prints the tool version.

Files

```
$SWAT_ROOT/etc/html/*
```
HTML and CSS templates and other scripts used for report generation.
4.3.8 **swat-todyn**

Performs static to dynamic transformation of a basic-block or line-based model file using profiling information.

**Synopsys**

```
swat-todyn <options> <files>
```

**Options**

- **-help**
  Prints a short description of the tool options.
- **-bb-count <file>**
  The basic-block count file collection profiling information.
- **-version**
  Prints the tool version.

**Files**

```
$PROJECT/*.bbmodel, $PROJECT/*.lnmodel
```

Static model files.

```
$PROJECT/*.q103
```

The basic-block count file.

4.3.9 **swat-trp**

Performs different types of execution trace post-processing.

**Synopsys**

```
swat-trp <options>
```

**Options**

- **-help**
  Prints a short description of the tool options.
- **-version**
  Prints the tool version.
- **-config <file>**
  The configuration file. See Section 4.5.1 for details.
- **-swat-debug**
  Produces a verbose debugging output of the execution.
- **-allocation-file <file>**
  Specifies the function allocation file. This file is a list of lines of the form:
  
  `<function>:<mode>`
where `<mode>` can be one of the following:

- `<mode-name>` A specific mode name available for the target CPU. The name of the mode file is specified in the configuration file.
- `inherit` Specifies that the function will dynamically inherit its mode from the caller.
- `force` Specifies that the function will dynamically force its mode to all callees.

This option is only significant when the `-fn-allocation` option is specified.

**-bb-count**

Converts a basic-block trace into a basic-block execution count file.

**-fn-allocation** `-allocation-file <file>`

Calculates energy and time based on the allocation of functions to specific power states, i.e. specific voltage/frequency modes of the target core. See the `-allocation-file` option for details.

**-fn-list** `<file>`

Top level functions list file. Specifies which function to include in the hierarchical cost analysis. See `-fn-toplevel` option for details.

**-fn-name** `<function>`

Specifies a single function to be included in the hierarchical cost analysis. See the `-fn-toplevel` option for details.

**-fn-stackbound** `-ssa-file <file>`

Calculates the maximum size of the stack.

**-fn-stacksize** `-ssa-file <file>`

Generates a trace indicating the evolution of the stack size over time.

**-fn-top-level** `{ -fn-list <file> | -fn-name <function> } [-full]`

Performs top-level hierarchical collection of execution time and energy of one or more functions. If the sub-option `-full` is specified, data for individual functions is generated in detailed format.

**-full**

Generates a report with data for individual function executions. See `-fn-top-level` option for details.

**-o** `<file>`

The output model file.

**-pipe**

Reads the trace from a pipe.

**-report**
Generates the output in report format (if applicable)

```plaintext
-ssa-file <file>
  Static stack analysis report file name. See -fn-stack-size option and the
  -fn-stack-bound option for details.
```

```plaintext
-trace <file>
  The input trace file name.
```

```plaintext
-verbose
  Generates a verbose output trace (if applicable)
```

Files

```plaintext
$PROJECT/*.*tNNN
  The input trace file.
```

### 4.3.10 swat-uniqid

Make all the basic-blocks and functions identifiers unique all over the entire project.

**Synopsys**

```plaintext
swat-uniqid <options> <files>
```

**Options**

```plaintext
-help
  Prints a short description of the tool options.
```

```plaintext
-bb-map <file>
  The basic-block map file that will be generated.
```

```plaintext
-bb-map <file>
  The function map file that will be generated.
```

```plaintext
-version
  Prints the tool version.
```

Files

```plaintext
$PROJECT/*.*bbmodel
  Model files of the project.
```

```plaintext
$PROJECT/*.*q103
  The basic-block count file.
```

### 4.4 Core Tools

Core tools implement the main processing and analysis flows. Such tools are built by combining base tools and additional glue logic.
4.4.1  **swat-characterize**

Implements the target processor characterization flow.

**Synopsys**

```
swat-characterize <options>
```

**Options**

The tool accepts the following options.

- **help**
  
  Prints a short description of the tool options.

- **version**
  
  Prints the tool version.

- **swat-debug**
  
  Produces a verbose debugging output of the execution.

- **clean**
  
  Removes all temporary files generated by the tool.

- **root <path>**
  
  Specifies the path where to store intermediate files and the final models. The specified path must also contain the two model files llvm.model, target.model and the bash script target.cleancmd (see below).

- **target-compiler <compiler>**
  
  The name of the target compiler.

- **optimization-level { O0 | O1 | O2 | O3 }**
  
  Specifies the optimization level to be used by both the llvm-gcc and target compilers.

- **passes <string>**
  
  Specifies which passes of the characterization process to run. The string `<string>` is a combination of:

  - a  Run all passes
  - l  LLVM compilation
  - L  LLVM instruction count
  - t  Target compilation
  - T  Target instruction count
  - e  Build tables and equations
  - s  Solve equations
  - m  Builds target model
Note that, excluding 'a', each pass requires all those that precede it. Specifying 'a' alone is a synonym for 'LlTTesm'. This option is useful both for debugging and for building different target processors models using several sets of model vectors.

Files

The tool uses the following files:

```
$SWAT_ROOT/etc/src/characterize/*.c
```

The input training set of source files.

```
llvm.model
```

The file expressing in readable form the model vectors of LLVM instruction. See Section 0 for details.

```
target.model
```

The file containing the base costs of each target instruction. See Section 4.5.2.3 for details.

```
target.cleancmd
```

This is a script file that reads a target assembly file and produces as output a list of the op-codes found in the assembly code, one per each line.

4.4.2 swat-core-ba

Implements the main estimation and back-annotation flow.

Synopsys

```
swat-core-ba <options>
```

Options

The tool accepts the following options.

```
-help
```

Prints a short description of the tool options.

```
-version
```

Prints the tool version.

```
-swat-debug
```

Produces a verbose debugging output of the execution.

```
-config <file>
```

Specifies the configuration file.

Files

The tool uses the following files:

```
$PROJECT/*.swatcfg
```

The configuration file. See Section 4.5.1 for details.
$\text{PROJECT}/*.c
Source files of the application under analysis.

$\text{PROJECT}/\text{Makefile}
If available, a makefile to compile the application.

$\text{PROJECT}/\text{wrapper.sh}
If available, a wrapper script to invoke the application.

$\text{PROJECT}/*.ba
Output back-annotated source files.

$\text{PROJECT}/*.q<\text{NNN}>
Output analyses files. See Section 4.5.2 for details.

$\text{PROJECT}/*.r<\text{NNN}>
Output analyses files, in plain-text report form.

$\text{PROJECT}/*.t<\text{NNN}>
Output trace files, usually used as intermediate files.

In addition to those explicitly listed above, the tool uses and produces several other files. See the description of the base tools for details. The following base tools are used:

\begin{verbatim}
swat-bbmodel   swat-uniqid
swat-lnmodel   swat-minstr
swat-instrexpand swat-todyn
swat-trp       swat-analyze
swat-ba
\end{verbatim}

Individual tools are invoked with command line options defined by to the overall configuration file. The format of this file is described in Section 4.5.1.

\section*{4.4.3 \texttt{swat-core-tr}}

Implements the core instrumentation and tracing flow.

\textbf{Synopsys}

\begin{verbatim}
swat-core-tr \texttt{<options>}
\end{verbatim}

\textbf{Options}

\begin{itemize}
\item [-help]
  Prints a short description of the tool options.
\item [-version]
  Prints the tool version.
\item [-swat-debug]
  Produces a verbose debugging output of the execution.
\end{itemize}
-config <file>
   Specifies the configuration file.

-trace <name>
Specifies the trace to be performed. The configuration file must contain a section named [trace=<name>] that describes all the details of the trace process requested.

Files

$PROJECT/*.swatcfg
   The configuration file. See Section 4.5.1 for details.

$PROJECT/*.c
   Source files of the application under analysis.

$PROJECT/Makefile
   If available, a makefile to compile the application.

$PROJECT/wrapper.sh
   If available, a wrapper script to invoke the application.

In addition to those explicitly listed above, the tool uses and produces several other files. See the description of the base tools for details. The following base tools are used:

swat-bbmodel     swat-uniqid
swat-minstr      swat-instrexpand

Individual tools are invoked with command line options defined by to the overall configuration file. The format of this file is described in Section 4.5.1.

4.5 File formats

4.5.1 Configuration file format

The general syntax of the configuration file is very simple. It is structured into a list of sections, each constituted by a list of variable definitions. A section begins with a line of the form:

[<section-name>]

and ends either at the beginning of a new section, or at the end of file. Section names are tool-specific. Within a section, variables are assigned values using the syntax:

<variable> = <value>

where the variable names are tool- and section- dependent. The main sections (i.e. excluding debugging sections) currently supported are listed and described in the following.

4.5.1.1 Section [global]

This section describes general settings for the toolchain. The variables in this section are:
quiet = { true | false }
If set to true, the output of the application being analysed will be redirected to /dev/null while executing.

format = <string>
Specifies the floating point format to be used for back-annotation according the format specifiers defined by the C standard for the printf function.

stream = <number>
Specifies the file id number to be write the trace to. This can be a file id between 3 and 255. Files 1 (stdout) and 2 (stderr) should be avoided to prevent intermixing the trace output with other informational or error messages and with possible application output.

4.5.1.2 Section [project]

project = <string>
The name of the project. The name will be used for all intermediate and output files referring to the whole project. If omitted, the basename of the current directory is assumed.

sources = { <string-list> | all }
The list of source files to be analysed or the string all to indicate that all C source files in the current directory must be considered.

extra = <string-list>
The list of additional C source files needed to compile the application but excluded from the analysis process.

filterin = <string-list>
The list of functions to be considered for analysis. By specifying this variable, the only functions being analysed will be those listed here. Note that the functions specified here must be defined in one of the source files included in the analysis.

filterout = <string-list>
The list of functions to be excluded from analysis. The functions analysed will be all those defined in the source files indicated by the sources variable, except those listed here.

script = <string>
The name of a script that works as a wrapper around the application and that invokes it with suitable arguments. Such a script is used mainly when the application command line is complex and tedious to write.

args = <string>
The arguments to be passed to the application when executing it.

4.5.1.3 Section [report]
Configures the report generation.

generate-html = { true | false }
Specifies whether to generate or not the detailed HTML report. Note that this option implies execution of a full analysis with swat-qq.

output-dir = <string>
Specifies the output report directory. The main report file is index.html and is located in this directory. If omitted the relative path ./report is assumed.

### 4.5.1.4 Section [target]

This section collects the information about the target hardware platform.

**cpu = <string>**

Specifies the file containing the target CPU model. The file has the .cpu suffix and is located in $SWAT_ROOT/etc/cpu.

**cpu-modes = <string>**

Specifies the file containing the target CPU operating modes model. The file has the .modes suffix and is located in $SWAT_ROOT/etc/cpu.

**types-size = <string>**

Specifies the size in bit of the POD types for the target architecture. It is a colon-separated list of integers corresponding to the following types:

- char
- short int
- int / long int
- long long int
- float
- double
- long double
- pointer

These sizes are used to determine the stack size.

**frame-size = <number>**

Indicates the size in bits of the core part of the stack, typically a base pointer and a link register.

### 4.5.1.5 Section [compilers]

Specifies the options to be passed to the LLVM and the host compilers for compilation, optimization and linking. Normally, all files are compiled and linked, but if a more complex build mechanism is necessary, it can be specified in a makefile.

**host-ccflags = <string>**

Host compiler compilation flags.

**host-ldflags = <string>**

Host compiler linking flags.

**host-makefile = <string>**

Name of the makefile to be used to build the application on the host.

**llvm-ccflags = <string>**
LLVM compiler compilation flags.

`llvm-optflags = <string>`
LLVM compiler optimization flags.

### 4.5.1.6 Section [cfg]

This section specifies the control-flow graph analyses to be performed. CFG analyses can be disabled since they may require very long times for complex functions.

`run-loops = { true | false }`
Enables or disables the loop analysis. This analysis produces a list of all the loops present in the application.

`run-paths = true`
Enables or disables the path analysis. This analysis produces a list of all the execution paths present in the application. The number of paths may easily be very large, thus consider disabling this analysis if not strictly necessary.

### 4.5.1.7 Section [trace-<name>]

This is a generic section specifying the details of a specific tracing process. It currently consist of the following variables, but additional options are likely to be required to implement more complex tracing mechanisms.

`rules = <string>`
The instrumentation expansion rule file. The file has the `.rules` suffix and is located in `$SWAT_ROOT/etc/rules`.

`library = libswat-tracing.a`
The tracing support binary library. It implements the actual tracing functions. It is a static C library with `.a` suffix and is located in `$SWAT_ROOT/lib`.

`mode = { file | memory | pipe }`
Specifies the tracing mode, i.e. where trace data are supposed to be written. Note that the pipe option is currently not fully supported (not all the base and core tools can read/write from/to a pipe).

`binary = { executable | library | objects }`
Specifies how to treat the output of the instrumentation phase. If `objects` is specified, the instrumented sources are just compiled to object files but no linking takes place. If `library` is specified, the generated objects are linked into a static library. Finally, if `executable` is specified, the objects are linked in a binary executable file that can be run on the host machine. The experimental `source` option is also available to rewrite C source code after instrumentation, but at the present status of development it is not reliable enough.

`execute = { true | false }`
Specifies whether to execute or not the generated binary. Of course this option is only available when the output is a binary executable.
4.5.2 Model and rules formats

This section describes the most important model and file format that are manipulated by the different tools and tool-chains.

4.5.2.1 Source code basic-block model

The source code basic-block and line model files are structured as a list of colon-separated fields. The meaning of the fields is described below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>n/a</td>
<td>Function start source line</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>Function end source line</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>n/a</td>
<td>Source line</td>
</tr>
<tr>
<td>6</td>
<td>string</td>
<td>n/a</td>
<td>Basicblock name</td>
</tr>
<tr>
<td>7</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock id</td>
</tr>
<tr>
<td>8</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock start source line</td>
</tr>
<tr>
<td>9</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock end source line</td>
</tr>
<tr>
<td>10</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock size (instructions)</td>
</tr>
<tr>
<td>11</td>
<td>int</td>
<td>n/a</td>
<td>Execution count</td>
</tr>
<tr>
<td>12</td>
<td>float</td>
<td>cc</td>
<td>Static execution time</td>
</tr>
<tr>
<td>13</td>
<td>float</td>
<td>A</td>
<td>Static energy (average current)</td>
</tr>
<tr>
<td>14</td>
<td>float</td>
<td>cc</td>
<td>Dynamic execution time</td>
</tr>
<tr>
<td>15</td>
<td>float</td>
<td>A</td>
<td>Dynamic energy (average current)</td>
</tr>
<tr>
<td>16</td>
<td>string</td>
<td>n/a</td>
<td>Opcodes model list</td>
</tr>
</tbody>
</table>

The opcode list, in turn, is structured as a list of tuples enclosed in parenthesis. Each tuple is a comma-separated list of fields with the following meaning.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int</td>
<td>n/a</td>
<td>Source line</td>
</tr>
<tr>
<td>2</td>
<td>string</td>
<td>n/a</td>
<td>Opcode name</td>
</tr>
<tr>
<td>3</td>
<td>string</td>
<td>n/a</td>
<td>Opcode argument (optional)</td>
</tr>
<tr>
<td>4</td>
<td>float</td>
<td>cc</td>
<td>Static execution time</td>
</tr>
<tr>
<td>5</td>
<td>float</td>
<td>A</td>
<td>Static energy (average current)</td>
</tr>
</tbody>
</table>

4.5.2.2 LLVM instruction set model

The file describes the model of LLVM instructions in terms of target processor instructions. It is constituted by lines of the form:

\[<llvm-insn>;<target-insn1>;<target-insn2> ...\]

where \(<llvm-insn>\) is the name of an LLVM instruction and \(<target-insn<i>\)> is a target assembly instruction that is supposed to potentially contribute to the translation of the LLVM instruction in the heading of the line. Each line of the file must contain the name of exactly one LLVM instruction followed by zero or more target assembly instructions. If the
list of target assembly instructions is empty, no model will be constructed for that LLVM instruction.

4.5.2.3 Target processor instruction set model

The target model file collects the basic cost figures for the target microprocessor. It is constituted by a set of lines of the form:

    <target-instr>:<size>:<time>:<energy>

expressing the average size (bytes) of the instruction, its average execution time (clock cycles) and the average energy consumption, expressed as average current absorbed per clock cycle.

4.5.2.4 Instrumentation expansion rules

A rule file consists of three sections: a name section, a declaration section and an expansion rule section. A section is constituted by one or more rules, one per each line.

The name section has the form:

    /nm/<symbolic-name>/

Where nm is a fixed tag and <symbolic-name> is a human-readable name indicating the file name suffix of the trace file that will be generated. The declaration section collects the declarations of the instrumentation function that will be used. Each line has the form:

    /mi/<decl>/

where mi is a fixed tag and <decl> is a valid LLVM function declaration. One such declaration must exist for each of the function specified in the expansion rule section.

The expansion rule section is a list of expansion rules, each with the form

    /<tag>[(<field>=<value>)]/<call>/

where:

- <tag> is one of the tags bb, fc and fe, indicating basic-block instrumentation, function-call instrumentation and function-exit instrumentation respectively.

- <field>=<value> is an optional condition that determines whether the expansion must be applied to the specific meta instrumentation line. The <field> indicates one of the available fields capture form the meta-instrumentation line (see below) and the value is a valid possible value of the field.

- <call> is a valid LLVM call instruction of one of the instrumentation functions previously declared. The arguments of the function may be literal constants or fields captured from the meta-instrumentation line and macro-expanded during the process.

Valid fields are:

- $id Function or basic-block id, depending on the context
- $size Basic-block or stack size, depending on the context
4.5.2.5 LLVM meta-instrumented format

The meta-instrumented file format is a valid LLVM program, augmented by special comments introduced by one of the following tags

```plaintext
;;mi;;
;;bb;;<args>
;;fc;;<args>
;;fe;;<args>
```

followed by a list of fields depending on the specific tag.

4.5.3 Output report formats

All output report format are structured into lines, each composed by a list of colon-separated fields. A field can be:

- A number, integer or floating-point.
- A string.
- A list of tuples.

Lists are sequences of tuples enclosed in parentheses and constituted by a comma-separated list of items.

Each line refers to a single entity. An entity may be:

- A basic-block
- A source code line
- A function
- A CFG path
- A CFG loop
- An LLVM instruction

The beginning of the line specifies the entity in string form and, when applicable, in numeric form by means of a unique integer identifier. The rest of the line collects a set of information on the specified entity.

The following sections describe the formats and the information in each report file generated.
4.5.3.1 q101 – Basic-block cost

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>string</td>
<td>n/a</td>
<td>Basicblock name</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock id</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock execution count</td>
</tr>
<tr>
<td>6</td>
<td>float</td>
<td>cc</td>
<td>Basicblock static time</td>
</tr>
<tr>
<td>7</td>
<td>float</td>
<td>cc</td>
<td>Basicblock dynamic time</td>
</tr>
<tr>
<td>8</td>
<td>float</td>
<td>cc</td>
<td>Basicblock static energy</td>
</tr>
<tr>
<td>9</td>
<td>float</td>
<td>cc</td>
<td>Basicblock dynamic energy</td>
</tr>
</tbody>
</table>

4.5.3.2 q102 – Basic-block size

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>string</td>
<td>n/a</td>
<td>Basicblock name</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock id</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>insn</td>
<td>Basicblock size</td>
</tr>
</tbody>
</table>

4.5.3.3 q103 – Basic-block count

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock count</td>
</tr>
</tbody>
</table>

4.5.3.4 q110 - Selected basic-block plain

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>string</td>
<td>n/a</td>
<td>Basicblock name</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock id</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock execution count</td>
</tr>
<tr>
<td>6</td>
<td>float</td>
<td>cc</td>
<td>Basicblock static time</td>
</tr>
<tr>
<td>7</td>
<td>float</td>
<td>cc</td>
<td>Basicblock dynamic time</td>
</tr>
<tr>
<td>8</td>
<td>float</td>
<td>cc</td>
<td>Basicblock static energy</td>
</tr>
<tr>
<td>9</td>
<td>float</td>
<td>cc</td>
<td>Basicblock dynamic energy</td>
</tr>
</tbody>
</table>
4.5.3.5 q111 - Selected basic-block clustered

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>string</td>
<td>n/a</td>
<td>First basicblock name</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>First basicblock id</td>
</tr>
<tr>
<td>5</td>
<td>string</td>
<td>n/a</td>
<td>Second basicblock name</td>
</tr>
<tr>
<td>6</td>
<td>int</td>
<td>n/a</td>
<td>Second basicblock id</td>
</tr>
<tr>
<td>7</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock execution count</td>
</tr>
<tr>
<td>8</td>
<td>float</td>
<td>cc</td>
<td>Basicblock static time</td>
</tr>
<tr>
<td>9</td>
<td>float</td>
<td>cc</td>
<td>Basicblock dynamic time</td>
</tr>
<tr>
<td>10</td>
<td>float</td>
<td>cc</td>
<td>Basicblock static energy</td>
</tr>
<tr>
<td>11</td>
<td>float</td>
<td>cc</td>
<td>Basicblock dynamic energy</td>
</tr>
</tbody>
</table>

4.5.3.6 q120 - Memory pressure per basic-block

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>string</td>
<td>n/a</td>
<td>Basicblock name</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock identifier</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock start line</td>
</tr>
<tr>
<td>6</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock end line</td>
</tr>
<tr>
<td>7</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock operations count</td>
</tr>
<tr>
<td>8</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock memory access count</td>
</tr>
<tr>
<td>9</td>
<td>float</td>
<td>%</td>
<td>Basicblock memory pressure index</td>
</tr>
</tbody>
</table>

4.5.3.7 q130 - Full forward control-flow graph

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>string</td>
<td>n/a</td>
<td>First basicblock name</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>First basicblock id</td>
</tr>
<tr>
<td>5</td>
<td>string</td>
<td>n/a</td>
<td>Second basicblock name</td>
</tr>
<tr>
<td>6</td>
<td>int</td>
<td>n/a</td>
<td>Second basicblock id</td>
</tr>
</tbody>
</table>

4.5.3.8 q131 - Full backward control-flow graph

The file is organized into fields as described by the following table.
### 4.5.3.9 q132 - Control-flow graph paths

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>string</td>
<td>n/a</td>
<td>First basicblock name</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>First basicblock id</td>
</tr>
<tr>
<td>5</td>
<td>string</td>
<td>n/a</td>
<td>Second basicblock name</td>
</tr>
<tr>
<td>6</td>
<td>int</td>
<td>n/a</td>
<td>Second basicblock id</td>
</tr>
</tbody>
</table>

### 4.5.3.10 q133 - Control-flow graph loops

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>string</td>
<td>n/a</td>
<td>First basicblock name</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>First basicblock id</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>bb</td>
<td>Path length</td>
</tr>
<tr>
<td>6</td>
<td>list</td>
<td>n/a</td>
<td>(Name,Id)(Name,Id)...</td>
</tr>
</tbody>
</table>

### 4.5.3.11 q134 - Control-flow graph in-degrees and out-degrees

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>string</td>
<td>n/a</td>
<td>Basicblock name</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock id</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock in-degree</td>
</tr>
<tr>
<td>6</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock out-degree</td>
</tr>
</tbody>
</table>

### 4.5.3.12 q140 - Variable definitions and uses count

The file is organized into fields as described by the following table.
Field | Type | Units | Description
--- | --- | --- | ---
1 | string | n/a | Function name
2 | int | n/a | Function id
3 | string | n/a | Basicblock name
4 | int | n/a | Basicblock id
5 | int | n/a | Variables definitions
6 | int | n/a | Variables uses

### 4.5.3.13 q201 - Function cost

The file is organized into fields as described by the following table.

Field | Type | Units | Description
--- | --- | --- | ---
1 | string | n/a | Function name
2 | int | n/a | Function id
3 | int | n/a | Function execution count
4 | float | cc | Function total time
5 | float | cc | Function total energy
6 | float | cc | Function average time
7 | float | cc | Function average energy

### 4.5.3.14 q202 - Function size

The file is organized into fields as described by the following table.

Field | Type | Units | Description
--- | --- | --- | ---
1 | string | n/a | Function name
2 | int | n/a | Function id
3 | int | bb | Basic block count
4 | int | insn | Basic block minimun size
5 | float | insn | Basic block average size
6 | int | insn | Basic block maximun size
7 | int | insn | Basic block total size

### 4.5.3.15 q203 - Function arguments

The file is organized into fields as described by the following table.

Field | Type | Units | Description
--- | --- | --- | ---
1 | string | n/a | Function name
2 | int | n/a | Function id
3 | int | n/a | Number of formal arguments
4 | int | bytes | Total size of formal arguments

### 4.5.3.16 q210 - Selected function plain

The file is organized into fields as described by the following table.
### 4.5.3.17 q220 - Memory pressure per function

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function identifier</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock start line</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock end line</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock operations count</td>
</tr>
<tr>
<td>6</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock memory access count</td>
</tr>
<tr>
<td>7</td>
<td>float</td>
<td>%</td>
<td>Basicblock memory pressure index</td>
</tr>
</tbody>
</table>

### 4.5.3.18 q230 - Function stack size

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>bytes</td>
<td>Function stack size</td>
</tr>
</tbody>
</table>

### 4.5.3.19 q240 - Instruction usage per function

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>n/a</td>
<td>add static count</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>add dynamic count</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>n/a</td>
<td>alloca static count</td>
</tr>
<tr>
<td>6</td>
<td>int</td>
<td>n/a</td>
<td>alloca dynamic count</td>
</tr>
<tr>
<td>7</td>
<td>int</td>
<td>n/a</td>
<td>and static count</td>
</tr>
<tr>
<td>8</td>
<td>int</td>
<td>n/a</td>
<td>and dynamic count</td>
</tr>
<tr>
<td>9</td>
<td>int</td>
<td>n/a</td>
<td>ashr static count</td>
</tr>
<tr>
<td>10</td>
<td>int</td>
<td>n/a</td>
<td>ashr dynamic count</td>
</tr>
<tr>
<td>11</td>
<td>int</td>
<td>n/a</td>
<td>bitcast static count</td>
</tr>
<tr>
<td>12</td>
<td>int</td>
<td>n/a</td>
<td>bitcast dynamic count</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>13</td>
<td>int</td>
<td>n/a</td>
<td>br static count</td>
</tr>
<tr>
<td>14</td>
<td>int</td>
<td>n/a</td>
<td>br dynamic count</td>
</tr>
<tr>
<td>15</td>
<td>int</td>
<td>n/a</td>
<td>call static count</td>
</tr>
<tr>
<td>16</td>
<td>int</td>
<td>n/a</td>
<td>call dynamic count</td>
</tr>
<tr>
<td>17</td>
<td>int</td>
<td>n/a</td>
<td>extractelement static count</td>
</tr>
<tr>
<td>18</td>
<td>int</td>
<td>n/a</td>
<td>extractelement dynamic count</td>
</tr>
<tr>
<td>19</td>
<td>int</td>
<td>n/a</td>
<td>extractvalue static count</td>
</tr>
<tr>
<td>20</td>
<td>int</td>
<td>n/a</td>
<td>extractvalue dynamic count</td>
</tr>
<tr>
<td>21</td>
<td>int</td>
<td>n/a</td>
<td>fadd static count</td>
</tr>
<tr>
<td>22</td>
<td>int</td>
<td>n/a</td>
<td>fadd dynamic count</td>
</tr>
<tr>
<td>23</td>
<td>int</td>
<td>n/a</td>
<td>fcmp static count</td>
</tr>
<tr>
<td>24</td>
<td>int</td>
<td>n/a</td>
<td>fcmp dynamic count</td>
</tr>
<tr>
<td>25</td>
<td>int</td>
<td>n/a</td>
<td>fdiv static count</td>
</tr>
<tr>
<td>26</td>
<td>int</td>
<td>n/a</td>
<td>fdiv dynamic count</td>
</tr>
<tr>
<td>27</td>
<td>int</td>
<td>n/a</td>
<td>fmul static count</td>
</tr>
<tr>
<td>28</td>
<td>int</td>
<td>n/a</td>
<td>fmul dynamic count</td>
</tr>
<tr>
<td>29</td>
<td>int</td>
<td>n/a</td>
<td>fpext static count</td>
</tr>
<tr>
<td>30</td>
<td>int</td>
<td>n/a</td>
<td>fpext dynamic count</td>
</tr>
<tr>
<td>31</td>
<td>int</td>
<td>n/a</td>
<td>fpext static count</td>
</tr>
<tr>
<td>32</td>
<td>int</td>
<td>n/a</td>
<td>fpext dynamic count</td>
</tr>
<tr>
<td>33</td>
<td>int</td>
<td>n/a</td>
<td>fptosi static count</td>
</tr>
<tr>
<td>34</td>
<td>int</td>
<td>n/a</td>
<td>fptosi dynamic count</td>
</tr>
<tr>
<td>35</td>
<td>int</td>
<td>n/a</td>
<td>fptoui static count</td>
</tr>
<tr>
<td>36</td>
<td>int</td>
<td>n/a</td>
<td>fptoui dynamic count</td>
</tr>
<tr>
<td>37</td>
<td>int</td>
<td>n/a</td>
<td>fptrunc static count</td>
</tr>
<tr>
<td>38</td>
<td>int</td>
<td>n/a</td>
<td>fptrunc dynamic count</td>
</tr>
<tr>
<td>39</td>
<td>int</td>
<td>n/a</td>
<td>getelementptr static count</td>
</tr>
<tr>
<td>40</td>
<td>int</td>
<td>n/a</td>
<td>getelementptr dynamic count</td>
</tr>
<tr>
<td>41</td>
<td>int</td>
<td>n/a</td>
<td>icmp static count</td>
</tr>
<tr>
<td>42</td>
<td>int</td>
<td>n/a</td>
<td>icmp dynamic count</td>
</tr>
<tr>
<td>43</td>
<td>int</td>
<td>n/a</td>
<td>indirectbr static count</td>
</tr>
<tr>
<td>44</td>
<td>int</td>
<td>n/a</td>
<td>indirectbr dynamic count</td>
</tr>
<tr>
<td>45</td>
<td>int</td>
<td>n/a</td>
<td>insertelement static count</td>
</tr>
<tr>
<td>46</td>
<td>int</td>
<td>n/a</td>
<td>insertelement dynamic count</td>
</tr>
<tr>
<td>47</td>
<td>int</td>
<td>n/a</td>
<td>inttoptr static count</td>
</tr>
<tr>
<td>48</td>
<td>int</td>
<td>n/a</td>
<td>inttoptr dynamic count</td>
</tr>
<tr>
<td>49</td>
<td>int</td>
<td>n/a</td>
<td>load static count</td>
</tr>
<tr>
<td>50</td>
<td>int</td>
<td>n/a</td>
<td>load dynamic count</td>
</tr>
<tr>
<td>51</td>
<td>int</td>
<td>n/a</td>
<td>lshr static count</td>
</tr>
<tr>
<td>52</td>
<td>int</td>
<td>n/a</td>
<td>lshr dynamic count</td>
</tr>
<tr>
<td>53</td>
<td>int</td>
<td>n/a</td>
<td>mul static count</td>
</tr>
<tr>
<td>54</td>
<td>int</td>
<td>n/a</td>
<td>mul dynamic count</td>
</tr>
<tr>
<td>55</td>
<td>int</td>
<td>n/a</td>
<td>or static count</td>
</tr>
<tr>
<td>56</td>
<td>int</td>
<td>n/a</td>
<td>or dynamic count</td>
</tr>
<tr>
<td>57</td>
<td>int</td>
<td>n/a</td>
<td>phi static count</td>
</tr>
<tr>
<td>58</td>
<td>int</td>
<td>n/a</td>
<td>phi dynamic count</td>
</tr>
<tr>
<td>59</td>
<td>int</td>
<td>n/a</td>
<td>ptrtoint static count</td>
</tr>
</tbody>
</table>
The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>n/a</td>
<td>ialu static uses</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>ialu dynamic uses</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>n/a</td>
<td>falu static uses</td>
</tr>
<tr>
<td>6</td>
<td>int</td>
<td>n/a</td>
<td>falu dynamic uses</td>
</tr>
</tbody>
</table>

### 4.5.3.20 q241 - Instruction class usage per function

The file is organized into fields as described by the following table.
4.5.3.21 q301 - Program cost

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Metric name</td>
</tr>
<tr>
<td>2</td>
<td>float</td>
<td>n/a</td>
<td>Metric value</td>
</tr>
</tbody>
</table>

4.5.3.22 q310 - Instructions statistics

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Instruction opcode</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Static uses</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>n/a</td>
<td>Dynamic uses</td>
</tr>
<tr>
<td>4</td>
<td>float</td>
<td>n/a</td>
<td>Dynamic/static usage ratio</td>
</tr>
</tbody>
</table>

4.5.3.23 q311 - Instruction classes statistics

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Instruction class</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Static uses</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>n/a</td>
<td>Dynamic uses</td>
</tr>
<tr>
<td>4</td>
<td>float</td>
<td>n/a</td>
<td>Dynamic/static usage ratio</td>
</tr>
</tbody>
</table>

4.5.3.24 q320 - Inlining statistics per function

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Callee function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Callee function id</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>n/a</td>
<td>Static calls count</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>Dynamic calls count</td>
</tr>
<tr>
<td>5</td>
<td>float</td>
<td>n/a</td>
<td>Inlining index</td>
</tr>
</tbody>
</table>
4.5.3.25 q321 - Inlining statistics per call point

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Callee function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Callee function id</td>
</tr>
<tr>
<td>3</td>
<td>string</td>
<td>n/a</td>
<td>Caller file name</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>Caller source line</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>n/a</td>
<td>Dynamic calls count</td>
</tr>
</tbody>
</table>

4.5.3.26 q401 - Stack size dynamic bounds

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Metric name</td>
</tr>
<tr>
<td>2</td>
<td>float</td>
<td>n/a</td>
<td>Metric value</td>
</tr>
</tbody>
</table>

4.5.3.27 q901 – Basic-block-level metrics

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>string</td>
<td>n/a</td>
<td>Basicblock name</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock id</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock Execution Count</td>
</tr>
<tr>
<td>6</td>
<td>float</td>
<td>cc</td>
<td>Basicblock Static Time</td>
</tr>
<tr>
<td>7</td>
<td>float</td>
<td>cc</td>
<td>Basicblock Dynamic Time</td>
</tr>
<tr>
<td>8</td>
<td>float</td>
<td>mJ</td>
<td>Basicblock Static Energy</td>
</tr>
<tr>
<td>9</td>
<td>float</td>
<td>mJ</td>
<td>Basicblock Dynamic Energy</td>
</tr>
<tr>
<td>10</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock Size</td>
</tr>
<tr>
<td>11</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock Count</td>
</tr>
<tr>
<td>12</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock Start Line</td>
</tr>
<tr>
<td>13</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock End Line</td>
</tr>
<tr>
<td>14</td>
<td>int</td>
<td>insn</td>
<td>Basicblock Operations Count</td>
</tr>
<tr>
<td>15</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock Memory Access Count</td>
</tr>
<tr>
<td>16</td>
<td>float</td>
<td>n/a</td>
<td>Basicblock Memory Pressure Index</td>
</tr>
<tr>
<td>17</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock Next</td>
</tr>
<tr>
<td>18</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock Prev</td>
</tr>
<tr>
<td>19</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock Path Length</td>
</tr>
<tr>
<td>20</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock Loop Size</td>
</tr>
<tr>
<td>21</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock In Degree</td>
</tr>
<tr>
<td>22</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock Out Degree</td>
</tr>
<tr>
<td>23</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock Variables Definitions</td>
</tr>
<tr>
<td>24</td>
<td>int</td>
<td>n/a</td>
<td>Basicblock Variables Uses</td>
</tr>
</tbody>
</table>
4.5.3.28 q902 - Function-level metrics

The file is organized into fields as described by the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>string</td>
<td>n/a</td>
<td>Function name</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>n/a</td>
<td>Function id</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>n/a</td>
<td>Function Execution Count</td>
</tr>
<tr>
<td>4</td>
<td>float</td>
<td>cc</td>
<td>Function Total Time</td>
</tr>
<tr>
<td>5</td>
<td>float</td>
<td>mJ</td>
<td>Function Total Energy</td>
</tr>
<tr>
<td>6</td>
<td>float</td>
<td>cc</td>
<td>Function Average Time</td>
</tr>
<tr>
<td>7</td>
<td>float</td>
<td>mJ</td>
<td>Function Average Energy</td>
</tr>
<tr>
<td>8</td>
<td>int</td>
<td>n/a</td>
<td>Function Basicblock Count</td>
</tr>
<tr>
<td>9</td>
<td>int</td>
<td>n/a</td>
<td>Function Basicblock Minimum Size</td>
</tr>
<tr>
<td>10</td>
<td>float</td>
<td>n/a</td>
<td>Function Basicblock Average Size</td>
</tr>
<tr>
<td>11</td>
<td>int</td>
<td>n/a</td>
<td>Function Basicblock Maximum Size</td>
</tr>
<tr>
<td>12</td>
<td>int</td>
<td>n/a</td>
<td>Function Basicblock Total Size</td>
</tr>
<tr>
<td>13</td>
<td>int</td>
<td>n/a</td>
<td>Function Start Line</td>
</tr>
<tr>
<td>14</td>
<td>int</td>
<td>n/a</td>
<td>Function End Line</td>
</tr>
<tr>
<td>15</td>
<td>int</td>
<td>n/a</td>
<td>Function Operations Count</td>
</tr>
<tr>
<td>16</td>
<td>int</td>
<td>n/a</td>
<td>Function Memory Access Count</td>
</tr>
<tr>
<td>17</td>
<td>float</td>
<td>n/a</td>
<td>Function Memory Pressure Index</td>
</tr>
<tr>
<td>18</td>
<td>int</td>
<td>byte</td>
<td>Function Stack Size</td>
</tr>
<tr>
<td>19</td>
<td>int</td>
<td>n/a</td>
<td>Function Static Calls Count</td>
</tr>
<tr>
<td>20</td>
<td>int</td>
<td>n/a</td>
<td>Function Dynamic Calls Count</td>
</tr>
<tr>
<td>21</td>
<td>float</td>
<td>n/a</td>
<td>Function Inlining Index</td>
</tr>
</tbody>
</table>

4.5.4 Output trace files formats

Trace files are usually large files reporting a dynamic view of the application execution. The format of traces is less standardized than that of output reports since it needs to be somewhat optimized to reduce the overall size of the traces.

Currently all tools and flows only support text-based tracing. This is clearly not the most efficient solution, as binary files would save a lot of space. This approach, though, allows reading the content of files and better understanding their meaning, which is crucial for a prototype tool.

Tool front-ends and back-ends can nevertheless easily substituted with binary read/write interfaces when the maturity of the toolchain will justify it.

4.5.4.1 t801 - Basic-block id trace

This trace is a simple list of basic-block identifiers. Each line has the format:

```
<bbid>
```

The trace represents the exact execution of basic-blocks over time in a specific run.
4.5.4.2 t802 - Function id trace

This trace is a simple list of function identifiers. Each line has the format:

<fnid>

The trace represents the exact execution of functions over time in a specific run. The function identifiers can be:

Positive integers. Indicate calls to functions declared and defined in the project and included in the current analysis.

Negative integers. Indicate calls to function declared in the project but defined in external (binary or source) libraries or excluded from the current analysis.

Zero. Indicating functions introduced by the LLVM compilation flow, but irrelevant for the analysis. Such functions are usually intended for debugging purposes only.

It must be noted that function execution is trace upon entering the function. This means that execution of the following code:

```
Function main
Function foo
Function bar

main() {
    // BB1
    foo()
    // BB2
    bar()
    // BB3
}
```

will produce the trace:

```
main
foo
bar
```

where names have been used instead of identifiers for readability and BBn indicates a generic LLVM basic-block. As it can be noted, the trace does not explicitly show returns to main() after the execution of foo() and bar().

4.5.4.3 t803 - Function entry/exit trace

This trace is an extended version of the previous one that adds explicit information about entries and exits form the function being executed. It is a list of “events” of the form:

```
fc:<fnid>
```

or

```
fe:<fnid>
```
where fc (function call) indicate the point in the source code immediately before function entry and fe indicates the point immediately after function exit. The code of the previous example would produce the following trace, where, again, identifiers have been replaced with function names for readability:

```
fc: main
fc: foo
fe: foo
fc: bar
fe: bar
```

Thanks to this information and a simple stack-based post-processing algorithm it is possible to reconstruct the exact function call stack trace of the application.

### 4.5.4.4 t804 - Function entry/exit and basic-block id trace

This trace further enriches the trace with the identifiers of the basic-blocks actually executed. The trace consist of three types of “events” represented in the form:

```
fc:<fnid>
fe:<fnid>
bb:<bbid>
```

Again referring to the sample code previously considered, a possible execution trace might look like the following:

```
fc: main
bb: BB1
fc: foo
bb: BB4
bb: BB5
fe: foo
bb: BB2
fc: bar
bb: BB6
fe: bar
bb: BB3
```

where, again, function names and basic-block names are used for clarity instead of ids.

### 4.5.4.5 t805 - Function argument actual value trace

This trace reports the values of one or more actual parameters for given function calls. The format of the trace is similar to the function call trace, but each line is enriched with a list of values of the actual parameters. The format of each line is:

```
fa:<fnid>::<arg1>[:<arg2>...]
```
Where act1, act2, ... are the values of the parameters. The trace engine currently supports only integer, floating-point and pointers scalar values.

### 4.6 Dependency on third-party tools

The application only depends on open source free software. This section lists the dependencies on tools and packages, indicating whether they are mandatory or optional. Missing optional tools does not prevent the core functionality of the toolchain, but limits some of its functionality.

<table>
<thead>
<tr>
<th>Tool / Library</th>
<th>Version</th>
<th>M/O</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM framework</td>
<td>2.8</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>LLVM gcc</td>
<td>2.8-4.2</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>GNU cflow</td>
<td>1.4</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>GNU octave</td>
<td>3.2.3</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>GNU gnuplot</td>
<td>4.2</td>
<td>O</td>
<td>Used for report generation</td>
</tr>
<tr>
<td>GNU source-highlight</td>
<td>3.1.5</td>
<td>O</td>
<td>Used for report generation</td>
</tr>
<tr>
<td>graphviz</td>
<td>2.20.2</td>
<td>O</td>
<td>Used for report generation</td>
</tr>
</tbody>
</table>

### 4.7 Integration

The SWAT toolchain has been successfully integrated with:

**MOST**

The integration is achieved by means of suitable configuration files and perl/bash scripts.

**BAC++**

The integration is achieved by the swat-core-tr core toolchain. The output of SWAT is an instrumented version of the input application that can be linked with the BAC++ core.

**Synopsys**

The integration for task-level analysis and simulation is achieved by the swat-trp tool. The output of SWAT produces a text report of the figures to be used within the Synopsys task model.
5 Task-based Virtual Platform Simulation

5.1 Overview

As described in Section 2.3, the goal of the Task-based Virtual Platform Simulation is to enable a performance analysis of an application mapping on a particular system platform. In order to enable the Task-based Virtual Platform Simulation, the following tasks need to be performed:

1. Model the individual tasks of the application. Once the tasks are modeled, they can be instantiated and connected in Platform Creator. The result is a task graph. Task can be modelled functionally, which implies the actual algorithm-code is used to model the behaviour of the different components of the algorithm, this is particularly useful to model data dependent dataflow between the different components of the algorithm. It is also possible to model a task ‘non-functionally’, in which case the behaviour of the task is not modelled, only the data-dependencies as well as the load/impact on the architecture. The latter allows for much quicker task-graph development.

2. Execute the task graph stand-alone. This means that it is not yet mapped to a processing element, but it runs with a global task manager (the default task manager). The goal is to analyze the behavior of the application before the partition step happens. The results can be used to direct the application mapping or to finetune the architecture ahead of the mapping step.

3. Capture the platform. The platform will contain one or more VPU blocks that will run the application. The level of detail required in the platform model depends on the focus of the performance analysis. If the key aspect under investigation is the interconnect and memory subsystem then it may be sufficient to work with a simple VPU model. If however the goal is to determine the best application mapping, then it may be required to add additional details to the internals of the VPU model, e.g. cache behaviour and more detailed processing models (see later).

4. Map the application to the platform. The task graph is mapped to the different VPUs of the system. Connections between tasks that are mapped on different VPUs need to be resolved. The connection is refined with the appropriate drivers. The goal of the drivers is to provide with a model of the actual interaction between tasks, especially the interaction with the interconnect and memory subsystem should be refined.

5. Finally the complete platform can be analyzed. The Framework provides with extensive analysis capabilities that allow to analyze performance and power and to validate the system against constraints set by the user.

5.2 Functional modeling

5.2.1 Overview of SystemC modeling API for task-based functional models

The language used to do task modeling is SystemC. A task is modeled as a SystemC thread. For communication with other tasks, this thread is part of an sc_module. This module can have ports for communication with other tasks. The communication happens over channels.
The software can access an API - the Task Modeling API - to annotate execution times and traffic to be generated and to pass control over to other tasks. Each task has a priority which can be used by the scheduler and tasks can be grouped in jobs. A job is identified by its job ID. The Task Modeling API is the API for communication of the tasks with the task manager.

The task manager controls the states of the tasks it manages. The methods of the API are related to:

1. Task switching
2. Annotation of time and traffic it takes to execute a task
3. Start/stop/create new tasks
4. Get/set priorities and job IDs for each task
5. Some special functions

The following code shows the usage of the API in a very simple example. Only the code implementing the task is shown here.

```cpp
virtual void task() {
    while (true) {
        for (char i = 0; i < 10; i++) {
            // processing before put takes 20 cycles
            tm_consume(20);
            // pass the data to the next task with a blocking put call
            // if the channel would not be ready to accept, a task switch
            // will occur
            p->put(i);
            // some more processing takes 20 cycles
            tm_consume(20);
            // the task passes control back to the scheduler and wants
            // to be reactivated in 20 cycles.
            tm_wait(20);
        }
        // this task is done: it suspends itself
        tm_suspend();
    }
}
```

Figure 13 shows the task-state diagram. It shows all possible states for a task and the typical transitions between the states:
A task is first created, then started. After the start, the task is in the **TM_TASK_READY** state. All tasks in the **TM_TASK_READY** state can be activated by the scheduler. When a task is activated, it goes to the **TM_TASK_RUNNING** state. When a task is running, it can access the Task Modeling API. By calling `tm_wait()` without arguments, a task passes control back to the scheduler and returns to the **TM_TASK_READY** state. When a task calls wait for event or wait for time, the task goes into the **TM_TASK_WAITING** state. It stays there until the wait condition is done.

A task can suspend or destroy other tasks. In this case, that task goes to the **TM_TASK_SUSPENDED** or **TM_TASK_DESTROYED** state. The difference between these states is that a suspended task can be resumed, while a destroyed task can never become active again. You can call the API to suspend or destroy a task, independent of the state that task is in. The suspension or destruction is treated as a request and will only be handled the next time the task reaches the **TM_TASK_READY** state. This means that when you suspend a waiting task, the task will not be suspended until the wait condition was fulfilled.

For simplicity, stopping tasks is not shown in the above figure. From any state, a task can be stopped. When a task is stopped, it is either taken back to the **TM_TASK_CREATED** state (where it waits for a start) or it is completely deleted. Whether a task is taken to the **TM_TASK_CREATED** state or is deleted depends on its restartable property. The difference between suspending/resuming and stopping starting a task is:

- When suspending/resuming a task, the state of the task is kept. On resume, the task continues from its current location.
- When stopping/starting a task, the task is really reset after the stop. When it starts again, execution starts from the beginning.

![Task State Diagram](image.png)
All communication between tasks happens explicitly through ports. Between the ports, a channel takes care of the communication synchronization. The interfaces, ports, and channels that are used are regular SystemC interfaces, ports, and channels.

Table 1 gives an overview of the different communication ‘protocols’ that are supported for task communication.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scml_tm_put_get</td>
<td>Groups the blocking and nonblocking put and get interfaces. This is meant for passing individual objects between tasks over a variable or a FIFO channel.</td>
</tr>
<tr>
<td>scml_tm_indexed_put_get</td>
<td>Groups the blocking and nonblocking indexed put and get interfaces. This requires a buffer channel where an array of types or objects can be passed with an index.</td>
</tr>
<tr>
<td>scml_tm_array</td>
<td>Offers locked access to an array. These interfaces are based on the scml_array interface.</td>
</tr>
<tr>
<td>scml_tm_memory</td>
<td>Offers read and write access of character arrays to and from memory based on an address.</td>
</tr>
<tr>
<td>scml_tm_event</td>
<td>Offers an interface for pure event-based synchronization.</td>
</tr>
</tbody>
</table>

Table 1 - Protocol summary

Tasks communicate through channels that inherit from scml_tm_task_module as they need to access the Task Modeling API. For example, when a task does a blocking write to a channel, but the write cannot happen because the current value has not been read yet, the channel will call tm_wait() to stop the execution of the current thread (the one initiating the write) and pass control over to another thread.

The basic channels that are provided in the model library are:

- Variable channel: Implements blocking and nonblocking put and get of a single variable.
- FIFO channel: Implements blocking and nonblocking put and get of variables.
- Buffer channel: Implements blocking and nonblocking put and get of an array of variables.
- Array channel: Implements locked array access.

**5.2.2 Generic Task Library**

As explained in Section 2.3, a typical use case for the task graph model is to have a non-functional model for the application behaviour. In order to support this use model a Generic Task Library has been created. The Generic Task Library enables the rapid creation of application task graphs using a set of highly configurable task models. The data-flow model of computation is used to describe the execution precedence in the task graph.

The Generic Task Library contains the following configurable models:
- Autonomous task without data dependencies
- Feed-forward task with inputs and outputs
- Sink task with inputs only
- Source task with outputs only

- Function blocks for modelling task behaviour
  - CPU function to annotate processing time, fetch-, load-, and store probability
  - Memory function to model explicit memory access

- Processing models for VPU traffic generation
  - Simple: stochastic traffic generator
  - Cache: includes stochastic level 1 cache model

- Drivers
  - Put/get fifo drivers, memory post driver

- Helper Blocks
  - PV and post multiplexer

The **autonomous task** is a task that runs without data-dependencies, it runs independently of other tasks but can trigger function blocks to model task behaviour. The autonomous behaviour can be triggered through a trigger port. The parameters of this task are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_behavior</td>
<td>Determines initial state of the task (Ready, Created, or Suspended, see notes).</td>
</tr>
<tr>
<td>priority (integer)</td>
<td>Influences the priority scheduler, ignored by round-robin scheduler; 0 is highest priority</td>
</tr>
<tr>
<td>job_id (integer)</td>
<td>Associate multiple tasks into a single job by giving them the same job_id. Commands are available to handle start and stop jobs during the simulation</td>
</tr>
<tr>
<td>verbose</td>
<td>Switch on diagnostic run-time messages</td>
</tr>
<tr>
<td>start_delay_in_ns (integer)</td>
<td>Initial delay, only executed once at simulation start.</td>
</tr>
<tr>
<td>wait_delay_in_ns (integer)</td>
<td>Self-activation delay, executed after each iteration</td>
</tr>
<tr>
<td>iterations(integer)</td>
<td>Number of activations during a simulation</td>
</tr>
<tr>
<td>NBR_CALL_PORTS(integer)</td>
<td>Number of optional ports to connect function blocks</td>
</tr>
<tr>
<td>TRIGGER_PORT(0 or 1)</td>
<td>Optional port, connect initial trigger</td>
</tr>
</tbody>
</table>

The **source task** models a task that generates tokens on its output port, it has an autonomous behaviour that can be triggered via a trigger port. The parameters for the source task are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NBR_PRE_CALL_PORTS(integer)</td>
<td>Number of ports to connect function blocks. These function blocks are called before the output samples are produced</td>
</tr>
<tr>
<td>NBR_POST_CALL_PORTS(integer)</td>
<td>Number of ports to connect function blocks. These function blocks are called after the output samples are produced</td>
</tr>
<tr>
<td>NBR_PUT_PORTS(integer)</td>
<td>Number of data output ports</td>
</tr>
<tr>
<td>TRIGGER_PORT(0 or 1)</td>
<td>Optional port, connect initial trigger</td>
</tr>
</tbody>
</table>

Data rates can be model via parameters on the output port(s) of the source task

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>put_rate</td>
<td>Float number, that specifies the number of samples per task activation; e.g. <em>&quot;1.2&quot;</em> specifies that after 5 task activations 6 samples are generated</td>
</tr>
</tbody>
</table>
The sink task is used to end the simulation after a certain number of samples have been received, the sink task behaviour is not autonomous, it is completely dependent on the arrival of input samples. The parameters for the sink task are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto_stop (bool)</td>
<td>When true the sink task stops the simulation as soon as the expected number of input samples is reached</td>
</tr>
<tr>
<td>NBR_PRE_CALL_PORTS (integer)</td>
<td>Number of ports to connect function blocks. These function blocks are called before the input samples are consumed</td>
</tr>
<tr>
<td>NBR_POST_CALL_PORTS (integer)</td>
<td>Number of ports to connect function blocks. These function blocks are called after the input samples are consumed</td>
</tr>
<tr>
<td>NBR_PUT_PORTS (integer)</td>
<td>Number of data input ports</td>
</tr>
</tbody>
</table>

The input port of the sink task has a parameter ‘get_samples(integer)’. This parameter specifies the total number of expected samples per input port. The parameter is ignored unless auto_stop is set to true. The simulation will end as soon as all input ports have received the expected number of samples.

The feedforward task is a task which behaviour depends on the arrival of input data on the input ports and which generates output data on its output ports. The behaviour is not autonomous. The parameters for the feedforward task are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_rates_csv</td>
<td>Is a comma separated value list of float numbers indicating the number of samples consumed per activation</td>
</tr>
<tr>
<td>put_rates_csv</td>
<td>Is a comma separated value list of float numbers indicating the number of samples produced per activation</td>
</tr>
<tr>
<td>init_put_samples_csv (string)</td>
<td>Is a comma-separated-value list of integer numbers that specify the number of initially generated samples. This may be necessary to avoid deadlocks in feedback loops</td>
</tr>
<tr>
<td>or_gating (bool)</td>
<td>If false (default), the task executes when all get ports have data. If true, the task executes as soon as 1 get port has data</td>
</tr>
<tr>
<td>NBR_PRE_CALL_PORTS (integer)</td>
<td>Number of ports to connect function blocks. These function blocks are called before the input samples are consumed</td>
</tr>
<tr>
<td>NBR_CALL_PORTS (integer)</td>
<td>Number of ports to connect function blocks. These function blocks are called after the input samples are consumed.</td>
</tr>
<tr>
<td>NBR_POST_CALL_PORTS (integer)</td>
<td>Number of ports to connect function blocks. These function blocks are called after the output samples are produced</td>
</tr>
</tbody>
</table>

The put- and get_rates_csv are parameters that are passed on to the put and get ports of the feedforward task where they have a similar behaviour as described for the source task.

In order to model the behaviour of a task each of the generic task model has a configurable number of CALL ports, these should be used to add Function and memory models. The
function and memory models also have CALL ports so that they can be chained into complex behaviours.

The **Function Model** is intended to model CPU functions, the goal of this model is to enable to annotate the task graph so that the execution time of the task graph can be modelled. The drawback of this approach is that the clear separation between function and architecture is somewhat lost since the annotations will have to take into account the type of processing unit the function is mapped to. The annotations are still independent of detailed architectural features like clock frequency, cache size and interconnect and memory subsystem, they only depend on the processing type (CPU, DSP, VLIW...)

The parameters for the function model are as follows:

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>processing_cycles</td>
<td>Number of processing cycles per activation will be multiplied with clock period to determine processing delay</td>
</tr>
<tr>
<td>fetch_ratio</td>
<td>Probability that the task performs an instruction fetch</td>
</tr>
<tr>
<td>load_ratio</td>
<td>Probability that the task performs an data load operation</td>
</tr>
<tr>
<td>store_ratio</td>
<td>Probability that the task performs a data store operation</td>
</tr>
<tr>
<td>insn_addr_offset</td>
<td>Additional offset to the addresses of the instruction fetches, added to the instruction memory base address of the VPU</td>
</tr>
<tr>
<td>insn_memory_size</td>
<td>Size of the memory region that is accessed by the instruction fetches, if 0 the default instruction memory size of the VPU is used</td>
</tr>
<tr>
<td>data_addr_offset</td>
<td>Additional offset to the addresses of the load and store operations, added to the data memory base address of the VPU</td>
</tr>
<tr>
<td>data_memory_size</td>
<td>Size of the memory region accessed by the load and store operations, if 0 the default data memory size of the VPU is used</td>
</tr>
</tbody>
</table>

The **Memory Function** is used to model the explicit generation of memory requests. The goal is to model the memory requirements of an application in the task graph. Later when mapping the task graph to an architecture the memory functions can be used to model the detailed interaction of the application with the interconnect and memory subsystem. A memory function will be used to model for example the interaction of an digital imaging application with the image that is assumed to be centrally stored for the complete task graph. In combination with a function model which focuses on the interconnect and memory traffic as a consequence of the instruction execution and local data in the algorithm, the memory function can be used to model the interaction with globally centralized data for the application.

The parameters of the memory function are as follows:

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>burst_size</td>
<td>Specifies the size of a single burst access</td>
</tr>
<tr>
<td>read</td>
<td>Specifies whether the memory function will perform read or write accesses</td>
</tr>
<tr>
<td>total_data_size</td>
<td>Specifies the total amount of data to be read or written. When this size is read the accesses will start again at address 0.</td>
</tr>
<tr>
<td>data_size_per_activation</td>
<td>Specifies the amount of data to be read or written per activation</td>
</tr>
</tbody>
</table>

In a pure functional simulation the address used by the memory function is not relevant, hence the choice to always work from address 0, this allows a point-to-point connection between a
memory function and a memory model. When the task graph gets mapped a memory post driver is required which will convert the memory requests into bus transactions.

5.2.3 Table based Task Graph Description

In order to simplify the creation of task graphs and their mapping, a table based approach is developed that allows to specify common task graphs in a very simple format without losing the configurability and the flexibility that is provided through the Generic Task Library. The table format is based on the csv (comma separated values) format so that the table can be edited and processed from within excel.

An example task graph definition is shown below:

<table>
<thead>
<tr>
<th>task_name</th>
<th>trigger</th>
<th>connection</th>
<th>start_delay_in_ns</th>
<th>wait_delay_in_ns</th>
<th>iterations</th>
<th>put_rates_csv</th>
<th>get_samples_csv</th>
<th>auto_stop</th>
<th>job_id</th>
<th>priority</th>
<th>processing_cycles</th>
<th>preamble</th>
<th>task_name</th>
<th>trigger</th>
<th>connection</th>
<th>start_delay_in_ns</th>
<th>wait_delay_in_ns</th>
<th>iterations</th>
<th>put_rates_csv</th>
<th>get_samples_csv</th>
<th>auto_stop</th>
<th>job_id</th>
<th>priority</th>
<th>processing_cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preamble</td>
<td>Assemble</td>
<td>C_Pre_Ass</td>
<td>0</td>
<td></td>
<td>1000</td>
<td>321</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SignalFieldModulator1</td>
<td>SignalField_IFFT</td>
<td>C_SFM1_SFM1</td>
<td>1000</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SignalFieldModulator2</td>
<td>Assemble</td>
<td>C_SFM1_SFM2</td>
<td>1000</td>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source</td>
<td>DataField1</td>
<td>C_SRC_DF1</td>
<td>1000</td>
<td></td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataField1</td>
<td>Assemble</td>
<td>C_DF1_DF1</td>
<td>1000</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataField1</td>
<td>Assemble</td>
<td>C_DF1_DF2</td>
<td>1000</td>
<td>100</td>
<td>216</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataField2</td>
<td>Assemble</td>
<td>C_DF2_DF2</td>
<td>1000</td>
<td></td>
<td>216</td>
<td>2.21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataField3</td>
<td>Assemble</td>
<td>C_DF3_DF3</td>
<td>1000</td>
<td></td>
<td>477</td>
<td>1.21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assemble</td>
<td>Assemble</td>
<td>C_DF3_Ass</td>
<td>1000</td>
<td>576</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each line in the table specifies one instance of a task. The topology of the task graph is derived from the ‘trigger’ column. The table specifies the task type (source, sink, feedforward), the number of input and output ports and the connectivity of the ports. The remaining parameters are annotated to the respective task instances, each column should correspond to an actual parameter for that task, empty cells are skipped. With each task it is possible to associate 1 function model, the parameters for the function model can be added through additional columns. In this way the function graph is an overlay of the task graph.

Also the mapping of the task graph can be specified in this same table, in the mapping column the VPU to which a task should be assigned is specified. Task connections crossing VPU boundaries are automatically mapped to sc_fifo’s.

This approach is currently further elaborated to enable support for memory functions and more complex task graphs (e.g. multiple instantiation of task graphs or arrays of task graphs)

5.2.4 Constraint modelling

The modelling API described in the previous section focuses on the creation of functional models for an application. All the API’s and objects in this modelling environment are implemented with the necessary analysis hooks so that the end-user does not need to provide with any further instrumentation to enable the analysis environment that is provided along with the modelling infrastructure. The analysis infrastructure of the task based virtual platform modelling environment is based on the following concepts:

1. Instrumentation points (IPT): these are interfaces that are embedded in the models and provide with the hooks for an analysis application to extract data to be stored in an analysis database. The modelling objects described in the previous paragraph and the resource objects described in the following section are instrumented with a number of instrumentation points to provide the relevant data to the analysis tools.

2. Monitors: these are filters that either poll the instrumentation point interfaces or are being called by these interfaces and can be used to accumulate, correlate or do any
operation on the information provided by the IPT’s and store that in the analysis database. Monitors can combine information from different IPT’s. To reduce simulation overhead and database size, monitors are inactive by default (they are not connected to the IPTs) only through user interaction in the analysis tool they get enabled.

3. Analysis post processing: describes the final processing done in the analysis tools before creating a human readable or viewable output.

Constraint modelling is essentially an analysis feature. Constraints do not contribute to the functionality of the design. They are used to verify whether a design behaves within certain limitations. However it is not possible to hide constraints within the functional modelling objects since constraints are by definition orthogonal to the functionality and the range of possible constraints is too wide to provide with an approach only based on post processing. Therefore there is a need for a modelling API for constraint modelling.

Given the description above of the analysis infrastructure that is provided in the task based virtual platform modelling environment, constraints are split into instrumentation points, monitors and post processing. The modelling API for constraint modelling provides with the IPT hooks. Given that constraints should not be limited by the platform or application structure we also need to provide with a way to refer to a constraint independently of the current task or module in the design. Therefore each constraint should be labelled with an unique ID and a constraint manager is required.

The constraint interface of the preliminary version has been maintained. It allows to measure timing constraints. The constraint interface has the following 3 methods:

1. start(unsigned long long ID, const std::string& point_name)
2. stop(unsigned long long ID, const std::string& point_name)
3. cancel(unsigned long long ID)

These provide with a start and stop point for a certain constraint. The constraint is identified by an ID and name. It is also possible to cancel a constraint. E.g. when defining a constraint for the latency of a data transfer over an interconnect network, the cancel method can be used whenever the data transfer is dropped or cancelled along the way.

The constraint manager: scml_tm_constraint_manager is a singleton class that keeps track of all constraints that are started and stopped. Monitors can be written that interface with the constraint manager to extract information regarding the constraints in a design.

The monitors that are available in the current, preliminary version constraints are defined when configuring the analysis setup (when enabling the monitors in the design). The resulting analysis data can be represented in an analysis tool after post processing as in the example shown in the picture of Figure 14.
These graphs show the following:

- In the 2 top views: the start and stop times of different constraints in the design. The colouring indicates whether the constraint was met or not.
- In the 2 bottom views the graph shows the success and failure rate of different constraints in the design.

### 5.3 Architecture modeling

#### 5.3.1 Modeling API's for abstract resources

As described in the introduction in Section 2.3, a task graph is executed through a Task Manager. In a standalone simulation the complete functional application model is executed on a single, native Task Manager. Introducing multiple Task Managers allows to represent the different compute and OS resources in a system. By embedded the Task Managers into a Virtual Processing Unit the platform dependent resource interfaces can be represented. The combination of these orthogonal modelling API’s is required to support an exploration methodology where tasks can easily be mapped to different compute resources without the need for code changes, it also enables the reuse of abstract compute models for complex computation resources.

A task manager is the unit that implements the Task Modeling API and manages the state of tasks. The task manager activates the tasks and determines when a new task should be activated. A task manager has a scheduler. The scheduler is called by the task manager and decides which task will be run next. All tasks can run on their own without a VPU by using the default task manager. If a task is not mapped on a VPU, it is automatically controlled by the default task manager. Next to the scheduler, the task manager also has a processing model. There is a default processing model, but a custom version can be used.

The default task manager has a set of configuration options:

- **clock_period**: Is an integer configuring the time of the clock period in nanoseconds.
- **pre_emption**: Is a boolean specifying whether or not tasks should be pre-empted if the runnable queue changes.
- **scheduler**: Sets the scheduler to be used. It can be round-robin, priority, or a name of a custom-provided scheduler.

- **time_slice**: Is an integer configuring the time of the time slice in nanoseconds.

- **time_slicing**: Is a boolean specifying whether or not time slicing is enabled. Time slicing means that after a fixed time interval, the currently running task is preempted and a new task is scheduled.

A Virtual Processing Unit (VPU) is a processing resource for a number of tasks. A VPU has its own task manager. The tasks that run on a VPU are controlled by its task manager.

The Task Modeling library provides three different types of VPU blocks. The most general version has the following external interface:

- A set of memory ports. The number of memory ports is configurable. The memory ports are TLM2 ports.

- A set of interrupt ports. The number of interrupt ports is configurable.

- A clock port.

The task manager of the VPU has the same configuration parameters as the default task manager, except for the **clock_period**, which is derived from the clock port. The tasks running on a VPU need to interact with the rest of the hardware system. For this purpose, driver modules and interrupt-handling modules are available. These modules are connected to the ports of the VPU (the memory ports and the interrupt pins), which are connected to the rest of the hardware system. A VPU system looks as shown in Figure 15.

![Figure 15 – Example of a VPU system.](image-url)
In this figure, two tasks (Task A and Task B) communicate directly with each other. Each of the tasks also communicates with the rest of the system outside the VPU. The tasks are connected to drivers. These drivers implement the put or get interface (or any other interface) which the task uses. The driver translates the communication from the task into TLM2 accesses in the hardware system. For convenience, there is a transactor between the driver and the VPU port. The transactor handles the TLM2 communication on the ports of the VPU. It offers a simple post interface to all modules that want to access these ports.

There is also an interrupt driver module. This module monitors the interrupt pin and notifies clients when interrupts have occurred. In the above figure, the clients are the mailbox driver and the DMA driver. The interrupt driver also has a connection to the data port of the VPU. Once an interrupt is received, the interrupt driver needs to access the hardware to find out where the interrupt is coming from.

The figure also shows the processing model. This module handles the processing of tasks that were executed and annotated the traffic required for their execution.

### 5.3.2 Scheduling API’s and processing model

Custom schedulers can be plugged into the task manager. A scheduler needs to implement the following interface:

```cpp
class scml_tm_scheduler_if {
public:
    scml_tm_scheduler_if(scml_tm_task_api* tm_api);
    virtual ~scml_tm_scheduler_if();
    virtual bool preempt_on_interrupt(scml_tm_task_api::tm_task_id running_task) {
        return true;
    }
    virtual bool preempt_on_time_slice_end(scml_tm_task_api::tm_task_id running_task,
                                           const std::list<scml_tm_task_api::tm_task_id>& runnable_tasks) {
        return true;
    }
    virtual bool preempt_on_runnable_queue_change(scml_tm_task_api::tm_task_id running_task,
                                                 const std::list<scml_tm_task_api::tm_task_id>& runnable_tasks) {
        return true;
    }
    virtual scml_tm_task_api::tm_task_id select_task(const std::list<scml_tm_task_api::tm_task_id>& runnable_tasks) = 0;
protected:
    scml_tm_task_api* m_api;
};
```

The scheduler should basically pick a task from a list of tasks. It has access to the task-modeling API implementation to access the properties of tasks (like priority).

If the scheduler needs access to custom properties in the task, it can get a pointer to the task module to access its custom API. The retrieved scml_tm_module can further be downcasted to the custom base type:

```cpp
const scml_tm_module* tm_get_task_module_ptr(tm_task_id id);
```
Additionally, a custom scheduler can overrule default pre-emption behavior by overriding the preempt_on_* functions:

- `preempt_on_interrupt`: Checks whether the currently running task should be interrupted because an interrupt occurred.
- `preempt_on_time_slice_end`: Checks whether the currently running task should be interrupted because the time slice period ended.
- `preempt_onRunnable_queue_change`: Checks whether the currently running task should be interrupted because there was a change in the runnable queue.

Whenever a custom implementation is made, it should be registered with the scheduler factory before the system is constructed. For example:

```cpp
scml_tm_factory::inst().add_scheduler_creator(
    "my_scheduler",
    new scml_tm_scheduler_creator<my_scheduler>
);
```

The scheduler is registered with a name. To use the scheduler, you should set the name of your scheduler by means of the scheduler task-manager configuration parameter.

The processing model is not a regular task, but processes the annotations that a task has done while it was executed. When a task gives control back to the scheduler, the processing model for that task is executed. When the processing model is executing, the task that did the annotations is in the RUNNING state. When the processing model is executing, it can use the `tm_advance_time()` API call to advance the SystemC time. Advancing the time means as much as “call me back in x amount of time.” The task manager calls back the processing model after the specified amount of time has passed, unless there was a pre-emption event.

Dedicated calls for the processing model to advance the time:

```cpp
void tm_advance_time( const sc_core::sc_time& time )
void tm_advance_time( unsigned int cycles )
```

where:

- `time` specifies the amount of time that must be passed before the processing model is reactivated again.
- `cycles` specifies the amount of time that must be passed before the processing model is reactivated again in cycles. The absolute time is obtained from multiplication of the number of cycles with the clock period.

The processing model is the module that processes the consumes that were annotated by the tasks running on the VPU. A processing model should implement the following interface:

```cpp
virtual void process_consume(const sc_core::sc_time& time);
virtual void process_consume(const scml_tm_consume_data* data) = 0;
```

where `sc_time` and `scml_tm_consume_data` consume annotations should be processed.
For the consumption of time, a default implementation that just advances the time is available. This default implementation is shown here:

```cpp
virtual void process_consume(const sc_core::sc_time& time) {
    tm_advance_time(time);
}
```

`process_consume()` with `sc_time` is called for every `tm_consume(time)` annotation that was made during the execution of a task.

For the consumption of data, any data inheriting from `scml_tm_consume_data` can be passed from the tasks to the processing model. The processing model should downcast the `scml_tm_consume_data` pointer to the type it expects. The `scml_tm_consume_data` base class contains one member:

```cpp
struct scml_tm_consume_data {
    unsigned int processing_cycles;
    virtual ~scml_tm_consume_data() {}
    scml_tm_consume_data() : processing_cycles(0) {}
};
```

Having the `processing_cycles` member in the base class provides minimal compatibility between different processing models. If a processing model cannot downcast to the type it can work with, it can simply advance the time for the amount of cycles specified in this member.

Interrupt-handling modules should inherit from `scml_tm_isr_module`. This module is intended for handling asynchronous SystemC events. This may be an interrupt, but also other SystemC events. This module does synchronization with the SystemC world. The difference with drivers is that in this case the module listens to a SystemC event. This listening, however, does not happen from a “task”, but happens in the regular SystemC context. Whenever an interesting event happens, the task manager is asked to schedule a task that handles the event.

Modules inheriting from `scml_tm_isr_module` should implement:

- `void interrupt_service_routine()`: This is the interrupt task. It is scheduled by the task manager and you can use the Task Modeling API in this task. In this task, you can communicate with the hardware system to see what caused the interrupt and the interrupt can be distributed to the other tasks that are waiting for it. This method should not have a while(true) loop. Whenever it is finished, it is automatically restarted when the next activation is required.

- A SystemC method sensitive to the event you listen too. This event can be the interrupt pin in case of an interrupt handler. Whenever the event is activated, `handle_interrupt()` should be called. This call informs the task manager that the task responsible for handling the interrupt should be scheduled.
5.4 Power Modelling

5.4.1 Extensions for Power modelling

Currently a preliminary approach is taken to Power modelling. It is based on the existing infrastructure and a couple of additional modelling components that can be used in the platform model. Two types of power-awareness have to be added to the platform model, to enable more detailed analysis of power-related aspects of the design.

- Static and dynamic power consumption of the VPU based on the non-functional static and dynamic models presented in Section 4.
- Impact of (external) power management strategies on power consumption as well as system performance.

For the second goal, the current approach is to use the task based virtual platform simulation environment in combination with a functional power manager model to analyse the impact of the power management strategy on the system performance.

For this purpose a special processing model has been developed that monitors the state of the VPU and provides with an additional communication channel that indicates the active and idle states. This information is available to the power manager to make decisions on frequency changes, to disable components or to reconfigure parts of the system.

To obtain more detailed information about the energy consumption of the different components, an annotation-based approach is proposed. For the VPU this information will be built into the processing model, and be based on the activity as it is reported through the `consume()` calls from the different tasks. For the dynamic power estimations of the basic blocks of the tasks, block-annotated C++ (BAC++) is used to add such non-functional properties to the functional description of the tasks. In COMPLEX, these BAC++ annotations are already used for the estimation of custom hardware components, as described in [20].
Building upon the same underlying simulation techniques, the integration with the general tracing and analysis infrastructure can be used (Section 5.3.2)

Annotation information about non-functional aspects of behaviour to the functional description is done on the basis of basic blocks; hence the name block annotated C++ (BAC++). The power estimations for the dynamic model generate the corresponding consume() calls for the processing model of the VPU, which then translates the obtained information based on the scheduling information and the power state to the common tracing framework during the simulation. Since the estimates are obtained during simulation, data dependencies and control flow within the task can be considered more accurately than in a purely static model. Compared to the hardware annotations, timing information is given either in terms of abstract execution times, or as VPU clock cycles. Depending on the scheduling strategy of several tasks mapped to the same VPU, the consume calls are serialised for the processing model according to the multiplexed usage of the virtual CPU.

The static power estimation model has to be annotated to the VPU component directly. These static models can be extended by a Power-State Machine (PSM) based model that takes several operation modes of the VPU into account, defined by the power management strategy. The processing model of the VPU can then report the static power consumption based on the current power state to the analysis infrastructure as well.

5.4.2 Integration with analysis infrastructure

The analysis framework for the task based virtual platform simulation environment is based on a flexible and extendable infrastructure. Any value in the simulation can be traced and traces can be processed into new analysis views. At the moment the integration of power analysis is limited to an activity view as shown in Figure 17, future extensions that will be developed during the project should allow for more detailed analysis views.

In [20], an overview over the intended tracing and analysis infrastructure for custom hardware components is given. All power-aware components are reporting their power-consumption to the central analysis infrastructure that performs the recording of the dynamically observed power values for later analysis in specific views. Depending on the required estimation accuracy for the particular component, the local power traces can be pre-processed. This can be done e.g. by applying an averaging sliding window algorithm.

To enable a common view of the power-consumption of the whole system, the power tracing mechanisms for the task based virtual platform simulation are based on the same observer techniques. Due to the separation of the tasks, their basic annotations, and the power-aware processing model of the VPU, this unified analysis infrastructure across different domains of the system (software, custom hardware, black-box IP) will become possible.
5.5 Dependency on third-party tools

The Task based Virtual Platform simulation depends on the block annotated C++ (BAC++) for the power modelling approach.

As described in section 5.4.2 the Task based Virtual Platform simulation is integrated with the simulation and profiling tools developed in Task 3.1

5.6 Integration

The Task based Virtual Platform simulation toolchain does not depend on external tools.
6 Summary

This document presented the overall approach to software modelling and estimation and the way the models can be aggregated into a system-level virtual platform simulation engine.

The core portion of the document describes the three models and toolchains that constitute the software part of the COMPLEX flow, namely:

2. Detailed modelling and estimation.
3. Task-based virtual platform simulation.

Each Section presents an introduction to the methodology and a summary of previous works, followed by a description of the proposed methodology and an overview of the toolchain supporting it.
7 References


