A methodology and a tool for source level timing/energy estimation of software

Prof. William Fornaciari, fornacia@elet.polimi.it, home.dei.polimi.it/fornacia
Project Manager and Contact

Prof. Carlo Brandolese, brandole@elet.polimi.it
Chief Developer

Politecnico di Milano – DEI
Milano, Italy
Outline

Motivation
  – Software energy and timing estimation
  – Goals

Modeling approach & Toolchains
  – Source-level models
  – Dynamic models
  – Target processor characterization models

Results
  – Estimation
  – Analysis
Low power and energy aware design

- **Practical market issue**
  - Increasing market share of mobile, asking for longer cruising life
  - Limitations of battery technology

- **Economic issue**
  - Reducing packaging costs and achieving energy savings

- **Technology issue**
  - Enabling the realization of high-density chips (heat poses severe constraints to reliability)
Motivation - Application scenarios

High performance

Low power

Low energy

Large data centers, or data-intensive

General purpose, and multimedia

Reliability, and battery-supplied

Smartphones, mobile multimedia

Wireless Sensor Networks
Software energy and timing estimation

- Evaluation of the performance of an embedded system
  - Must consider specific contributions related to software execution
    - User application
    - Operating System & BSPs
    - User data
  - Should be performed as early as possible
    - To allow exploring different implementation alternatives
  - Should not depend on the availability of the target platform
    - ISS-based solutions do exist
    - Better if performed directly on the host machine
  - Should provide data at level at which the application is conceived
    - The source code level
  - Should be as fast and accurate as possible
    - A fast estimation toolchain allows automated design space exploration
Goals

- **The SWAT Project(**) aims at three main goals
  - Defines a general estimation approach
  - Defines
    - Execution time models
    - Energy consumption models
  - Defines low-level, target independent software metrics
    - To enable a more comprehensive analysis of the application
    - To support the designer in optimizing the application
  - Implements tools to support the models
    - As modular and flexible as possible

(*) The SWAT Project is part of the EU-Funded Project IP 247999 Project COMPLEX: "COodesing and power Management in PPlatform-based design space Exploration"
Modeling approach

- Energy consumption and execution time of an embedded application depend on
  - The structure of the application
    - Its source code and, consequently, its assembly translation
  - The stimuli
    - Specific data that the application processes
  - The execution platform
    - The target microprocessor
    - The operating system, if any
    - The third-party libraries

- All these aspects
  - Are complex to model as a whole with sufficient generality
  - Should be "decoupled" and modelled independently
Modeling approach

- The SWAT methodology decouples the different aspects

- **Source code model**
  - Captures the structure of the code at a high level of abstraction
  - No dependence on data or platform is accounted for

- **Stimuli model**
  - Collects set of representative execution profiles

- **Execution platform**
  - The target microprocessor is modeled at ISA level
  - The operating system and all third-party are statistically characterized operating at binary level, as source code may not be available
Source-level models

- To model the source code, several representations have been used in literature
  - Concrete and abstract syntax trees
  - Static metrics mutated from software engineering analyses
  - Black-box function-level analysis
  - Intermediate representation

- **SWAT models the source**
  - Using the LLVM Compiler Infrastructure
  - Starting from the LLVM intermediate representation
  - Adapting it to the specific purpose
    - For static modelling, several semantic aspects are irrelevant and the original LLVM code is simplified
    - For dynamic modelling, the LLVM code is augmented with suitable probes
Source-level models

- The simplified LLVM code is called "Basic-block model"

```c
int sumsqr( int* x, int n ) {
    int i, t = 0;
    for( i = 0; i < n; i++ )
        t += x[i] * x[i]
    return t;
}
```

```llvm
define i32 @sumsqr(i32* %x, i32 %n) nounwind {
    ...
    bb7:
        %8 = load i32* %i, align 4
        %9 = load i32** %1, align 8
        %10 = getelementptr inbounds i32* %9, i64 %9
        %...
}
```

Back-annotation info
Static & dynamic energy and timing (not yet available)
Basic-block model
Source-level models

- Formally the basic-block model is expressed by the representation matrix:

\[
R = \begin{bmatrix}
R_{1,1} & \cdots & T_{1,L} \\
\vdots & \ddots & \vdots \\
R_{B,1} & \cdots & T_{B,L}
\end{bmatrix}
\]

- Where
  - Each row corresponds to a basic-block
  - Each column corresponds to an LLVM instruction

- The element \( R_{i,j} \) of the representation matrix
  - Indicate how many LLVM instructions of type \( j \) are present in basic block \( i \)
Dynamic models account for data dependence

- Collect profiling information by executing the code
  - On the host platform, since the LLVM code can be targeted to the host
  - On an target ISS, if available (very slow)
  - On the actual target, using in-circuit debugging facilities

For OS-less application host execution is preferrable

- Must provide stubs for emulating devices

The result of execution is a basic-block trace

- The trace can be stored for an analysis over time
- The trace can be compacted to provide execution counts only
Dynamic model

- **SWAT provides a very flexible tracing support**
  - Can trace basic-blocks, functions, values, stack, ...

- **Tracing is based on a three-steps process**
  - **Meta-instrumentation**
    - Add tags to the LLVM code containing structured information about it
    - Tags are added as comments, so the resulting code is still LLVM-compliant
    - This phase is independent from the information to be traced
  - **Instrumentation expansion**
    - Expands tags into specific probes (function calls)
    - Uses a custom, awk-like, set of expansion rules
  - **Execution**
    - Produces the dynamic information for the specific simulus or set of stimuli
    - Data can be dumped in time-trace or collected as counts in memory
The dynamic model "enriches" the basic-block model.
Dynamic model

- Formally the basic-block counts are represented as

\[ p = [p_1 \ p_2 \ \ldots \ p_B] \]

- Where each element is associated to a basic block

- The resulting LLVM instruction counts are thus:

\[ s = p \cdot \mathbf{R} = [s_1 \ s_2 \ \ldots \ s_L] \]

- Where each element is the cumulative execution count of a specific LLVM instruction
Dynamic model

- To target the model to the specific processor we define a "translation matrix"

\[
T = \begin{bmatrix}
T_{1,1} & \cdots & T_{1,K} \\
\vdots & \ddots & \vdots \\
T_{L,1} & \cdots & T_{L,K}
\end{bmatrix}
\]

- Where
  - Each row corresponds to an LLVM instruction
  - Each column corresponds to an instruction of the target set

- The element \( T_{i,j} \) of the representation matrix
  - Indicate how many target instructions of type \( j \) are statistically used by the compiler to translate an LLVM instruction of type \( i \)
Dynamic model

- The energy and timing model of LLVM instruction for the specific target instruction set can thus be expressed as

\[
t(\mathcal{L}_m) = \sum_{i=1}^{K} T_{m,i} \cdot t(\mathcal{T}_i) = T_m \cdot [t(\mathcal{T}_1) \ldots t(\mathcal{T}_K)]^T = T_m \cdot K^T_t
\]

\[
e(\mathcal{L}_m) = \sum_{i=1}^{K} T_{m,i} \cdot e(\mathcal{T}_i) = T_m \cdot [e(\mathcal{T}_1) \ldots e(\mathcal{T}_K)]^T = T_m \cdot K^T_e
\]

- Where
  - \( \mathcal{L}_i \) is an LLVM instruction
  - \( \mathcal{T}_i \) is a target instruction
  - \( t(\cdot) \) indicates the estimated execution time
  - \( e(\cdot) \) indicates the estimated energy
Dynamic model

- Combining all the equations described so far yields the total estimated energy and execution time of the code

\[ c_e = p \cdot R \cdot T \cdot K_e^T \]
\[ c_t = p \cdot R \cdot T \cdot K_t^T \]

- In other words, combining
  - Execution count per each basic-block
  - LLVM instruction per each basic-block
  - Target instruction per each LLVM instruction
  - Energy and execution time per each target instruction

- ...leads to the overall estimates
Target processor characterization

- As we have seen
  - The model of the source code is expressed as LLVM code
  - The actual execution involves instructions of the target processor

- The translation matrix links these two levels
  - It expresses the "average" way the compiler translates an LLVM into one or more target instruction

- As an example:
  \[
  \text{extractelement}_{\text{LLVM}} = T_{3,1} \cdot \text{add}_{\text{TARGET}} + \\
  = T_{3,16} \cdot \text{ld}_{\text{TARGET}} + \\
  = T_{3,32} \cdot \text{shl}_{\text{TARGET}} + \\
  = T_{3,36} \cdot \text{sub}_{\text{TARGET}}
  \]
Target processor characterization

- The translation of an instance of an LLVM instruction
  - Depends on the context and optimizations selected

- The coefficients collects in the translation matrix
  - Can only represent an "average" translation
  - Should be derived analyzing a very large training set

- The characterization procedure is structured in steps
  - Select a suitably large set of source code benchmarks
  - Translate each source into LLVM
    - Count LLVM instruction frequency
  - Transalte each source into the target assembly
    - Count LLVM instruction frequency
  - Build a multilinear problem and solve it in least-square sense
Target processor characterization

- Formally, let
  - \( S = \{S_0, S_1, \ldots, S_N\} \) be the set of training source codes
  - \( L_{i,j} \) the count of LLVM instruction of type \( j \) in the source code \( i \)
  - \( D_{i,k} \) the count of target instruction of type \( k \) in the source code \( i \)

- The desired multilinear model can thus be expressed as

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

- In compact matrix form:

\[
L_j = Dx_j
\]
Target processor characterization

- This model is though too general
  - It assumes that the translation of each LLVM instruction may potentially involve \textit{all} target instructions!

- Realistically, the translation will only use a subset of target instruction
  - Knowing the semantics of both instruction sets, is easy to define a reasonable subset of target instructions

- This is modeled by the set of vectors
  \[ M_j = \{m_{j,1}, m_{j,2}, \ldots, m_{j,K}\} \quad m_{j,k} = \{0, 1\} \]

- where a 1 indicates that
  - LLVM instruction j
  - Might be potentially translated using also target instruction k
Target processor characterization

- To integrate these hypotheses into the model:
  \[ L_j = D \text{diag}(M_j) x_j = D'_j x_j \]

- From a "physical" point of view
  - Negative coefficients have no meaning
  - Coefficients must be constrained to be positive or null

- The constrained problem is thus expressed as
  \[ ||D'_j x_j - L_j||_2^2 \text{ with } x_j \geq 0 \]

- Each solution vector \( x_j \) collects the coefficients of the model of LVVM instruction \( j \)

- The translation matrix is thus simply given by
  \[ T' = [ x_0^T \ x_1^T \ \cdots \ x_L^T ] \]
Target processor characterization

- This characterization process
  - Is almost totally automatic
    - Only the translation “hypothesis” vectors must be coded manually
  - Has been implemented in the SWAT toolchain
The SWAT toolchain

- SWAT collects a set of tools analysis
  - Organized into several toolchains or flows

- In particular:
  - Target characterization flow
  - Estimation & back-annotation flow
  - Analysis flow
  - Generic tracing toolchain

- A few prototyping optimization tools and flows have also been implemented
  - Operating modes optimizer
  - Application parameters exploration and optimization
  - High-level transformation "hint" generator
The SWAT toolchain

- Toolchains are organized as follows
  - Front-End: Source code modelling
  - Core: Perform specific analyses
  - Back-End: Post-processing & Report generation
Estimation results

- Energy estimation errors (WCET Benchmarks)

![Bar chart showing estimation errors in percentage and absolute value for different benchmarks.]

**Average:**
- Percentage: -2.32%
- Absolute value: +5.98%
Estimation results

- Data dependency of energy estimation errors (WCET Benchmarks)
  - Same benchmark different data, namely the size of the array to be compressed

![Estimation error graph]

Estimation error (%)

Average: -3.74
Analysis Results

- Application summary & structure
Analysis Results

- Function models
Analysis Results

- Estimates overview & detailed charts
Analysis Results

- Dynamic analysis: Basic-block execution count
Analysis Results

- Trace analysis: Stack size over time
Work in progress

**SWAT opt. Toolchain (1)**

- Based on the same front-end of the estimation flow and using the results extracted from the dynamic models and the execution traces, the main optimization flow apply a set of fuzzy rules on selected portions of the application to suggest the most promising transformations to apply.
**Work in progress**

- **SWAT opt. Toolchain (2)**
  
  A second optimization flow explores compiler optimization options in order to determine the transformation mix that best fits the specific application. To this purpose, the MOST design space exploration engine is used to generate sets of transformation mixes for the LLVM optimizer until the best optimization recipe is found.
Moving up to system level: RTRM (BBQ)

Hierarchical

Run-Time Resource Manager (BBQ)

Scenarios

Requirements Aggregation

Resources Accounting

Symmetric

Asymmetric

Retargetability

P

D1

D2

P

Dn

Pg
Conclusions

- Software characterization is of paramount importance for embedded mobile applications
- There is room for optimization at source level
- Pressure is also for moving up the perspective
  - Time adaptability
  - Retargetability
  - System-wide resources management

- For more info
  - Prof. William Fornaciari
  - fornacia@elet.polimi.it
  - home.dei.polimi.it/fornacia

Thank you!