POET

POET Overview

17.2.2005

C. Brandolese, D.P. Scarpazza

Overview

- Power Estimation
  - Assembly-level
    - User code, Libraries, Operating System calls
  - Source-level
    - Full applications
- Power Optimization
  - Source level
    - Coding guidelines, optimization guidelines
- Demo

Assembly level power estimation

- Constructive
  - Total energy obtained as sum of elementary contributions related to either:
    - Assembly instructions
    - Functional units within the pipeline
  - Accurate, sufficiently fast
- General
  - Abstract model of a CPU
  - Good accuracy of the estimates

Assembly level power estimation

- From the abstract CPU model
  - We derive estimates of the average current absorbed per clock cycle by all instructions of the specific instruction set
- But…

$$E = V_{dd} \cdot I_{ave} \cdot T = V_{dd} \cdot I_{ave} \cdot N_{ck} \cdot T_{ck}$$

- Thus
  - Nominal execution times are inaccurate
  - Real execution time of instructions is essential

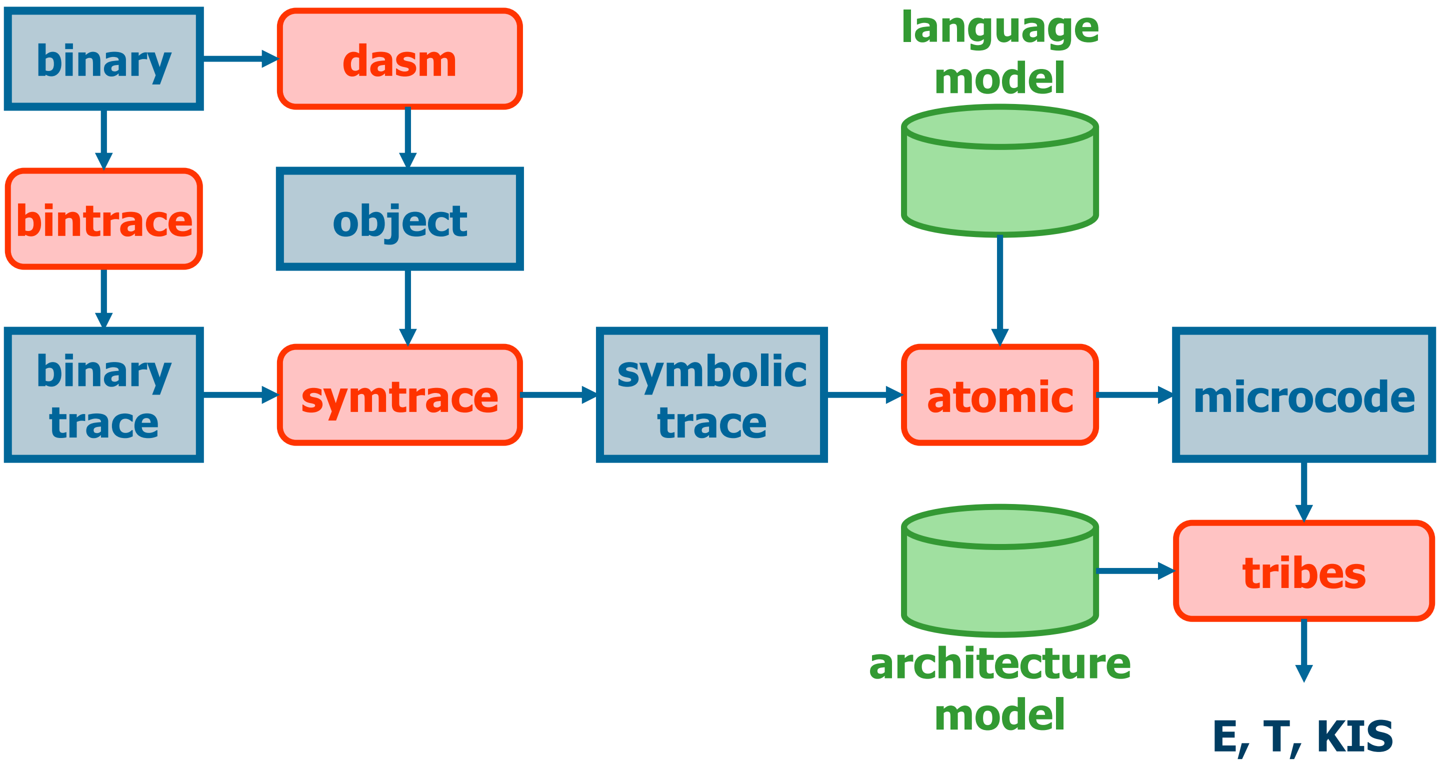
Behavioral simulation

- To obtain real execution times
  - The behavior of CPU must be modeled
    - Pipeline(s)
    - Cache(s)
  - We ignore explicit contributions due to
    - Data dependecies
    - Inter-instruction effects
  - These effects are accounted for statistically
- Output data used for source-level models

POET

Behavioral simulation

- Simulation toolchain

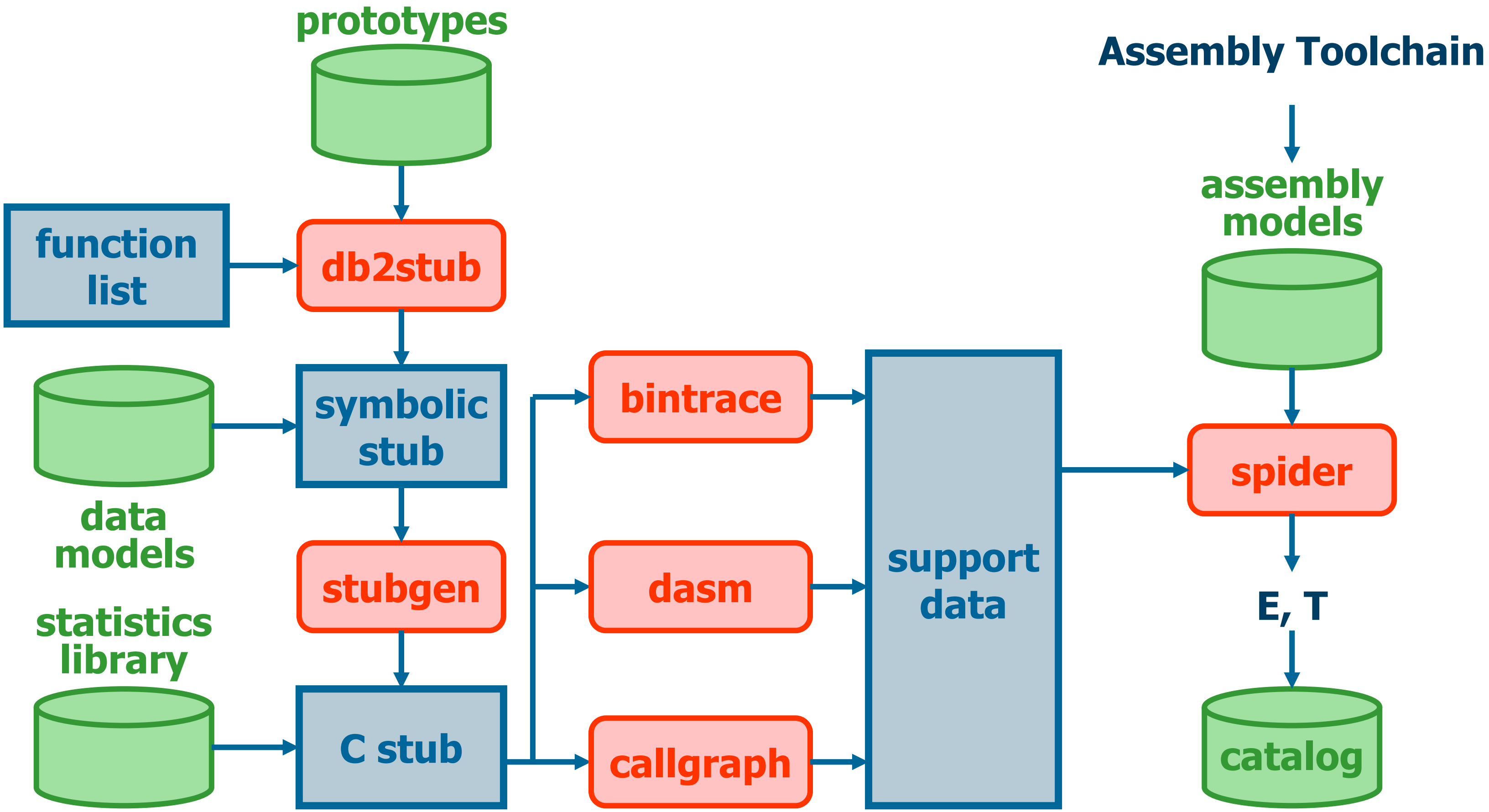


Library function characterization

- Third-party library functions
  - Often provided as binaries
    - No source code
  - Very used in building applications
- They can be usefully pre-characterized
  - Using the assembly-level toolchain
  - Feeding them with significant data
  - Extracting statistical model
- Models will be used at source-level

Library function characterization

- Library characterization toolchain



OS function characterization

- Some library functions
  - Are wrappers around system calls
- Assembly code executed in kernel mode
  - Is not accessible to our tracing tools
  - Is too complex to be simulated
- We thus resorted to measurements
  - On prototyping boards
  - Writing suitable drivers and stubs
  - Statistically modeling the raw results

Source-level power estimation

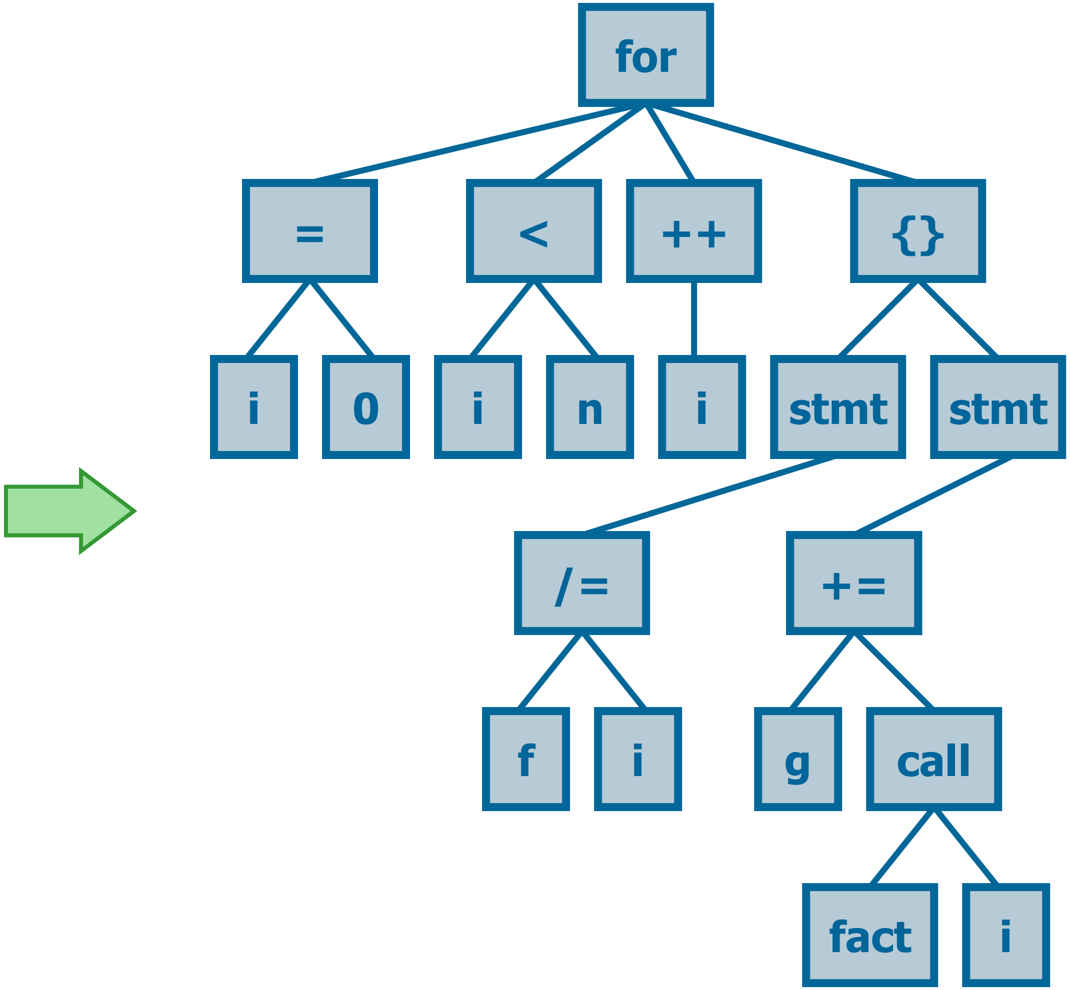
- Source code of an embedded application can be seen as structured into
  - User code
  - Library function calls (models available)
  - Operating system calls (models available)
- User code is mostly written in C
  - Estimates should refer to C-level
  - The approach should be independent from the target platform

Source-level power estimation

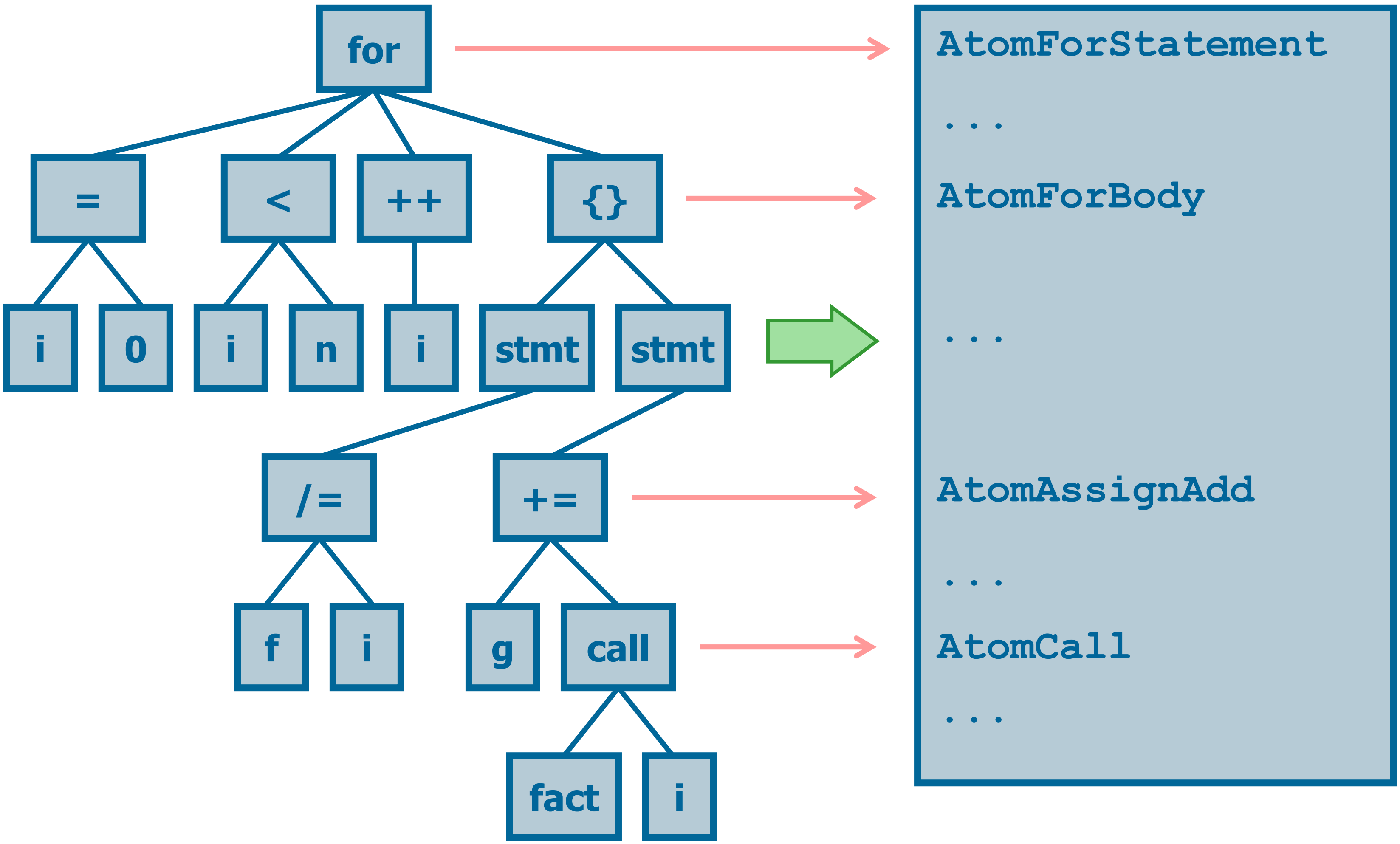
- Source code is parsed and decomposed
  - Parse tree made of nodes
  - Types and symbols tables
- Nodes are annotated
  - Elementary cost placeholders called atoms
- Atoms are translated
  - KIS instructions
- The process is a pseudo-compilation

C model: ANSI grammar

```
for(i=0;i<n;i++){
   f /= i;
   g += fact( i );
}
```



C model: Atom definition



C model: KIS definition

`AtomForStatement`	`1 cjt`
`...`	`...`
`AtomForBody`	`1 cjn + 1 jump`
`...`	`...`
`AtomAssignAdd`	`1 alul`
`...`	`...`
`AtomCall`	`4 mvld + 1 call`
`...`	`...`

C model: KIS costs

`1 cjt`	`13.776`
`...`	`...`
`1 cjn + 1 jump`	`17.304 + 6.042`
`...`	`...`
`1 alul`	`12.548`
`...`	`...`
`4 mvld + 1 call`	`186.097 + 52.509`
`...`	`...`

KIS costs

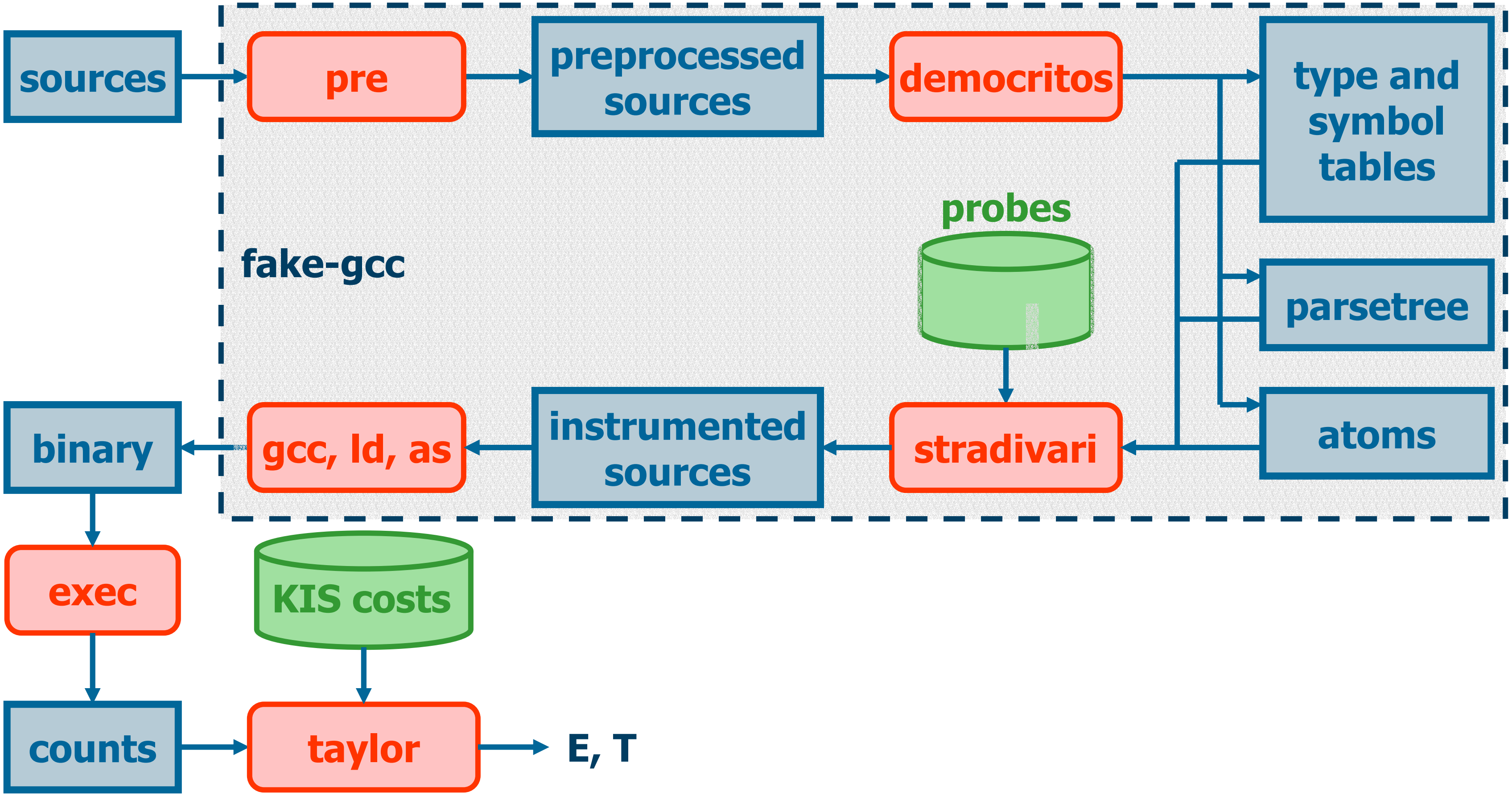
- KIS
  - Small set of assembly level instruction-classes
  - Used to model "atomic" operations
  - Fixed for all processors
- Given a real instruction set
  - Each instruction is mapped to a KIS class
- Costs of KIS classes
  - Suitable average over all real instructions that have been mapped onto that class

Profiling

- The source code parse tree is used to
  - Find optimal instrumentation points
  - Rewrite an instrumented version of the source code for profiling purposes
- The output of profiling
  - Reports the counts for all nodes
- Combining static data from KIS cost and profiling counts gives dynamic estimates

Source-level power estimation

- Estimation toolchain



Estimation results

- Thanks to
  - The information in the parse tree
  - The unit costs of atoms
- Cost are back-annotated to the source
- Shown to the user at source-level
  - Per line
  - Per function
  - Per file
  - Overall

Source-level optimization

- The first step for optimization is selecting
  - Critical functions
  - Critical code sections
- Selection is based on energy threshold
  - Relative to the overall energy absorbed by the application with the given set of data
- Critical portions define the initial scopes

Source-level optimization

- Interactive optimization is based on
  - A set of fuzzy rules
  - An inferential engine
- Each rule has
  - A fitness function
    - Implemented as a stand-alone tool
    - Returning a value in the range [0;1]
  - A threshold
  - An optimization guideline

Inferential optimization engine

- Each rule is applied on the initial scopes
  - If its fitness is greater than its threshold, then we say that the rule has fired
  - A fired rule produces
    - A suggestion for optimization (not always)
    - A new output scope
- Rules are reapplied until the set of scope does not change any longer
- This produces all optimization directives

Inferential optimization engine

- Optimization toolchain