Compositional Timing Analysis on Multi-Core Architectures

Vítor Rodrigues
Simão Melo de Sousa and Mário Florido

LIACC - Universidade do Porto, Portugal

Le Bois du Lys – November 2011
Problem Statement

Program Semantics for the Design of Multi-Core Embedded Systems

Support composable software components accompanied with design techniques that reduce the degree of interference between the components. (Cullmann et. al., 2010)

Proposed Approach

1. Semantic Building Blocks
   - Independent timing behavior
   - Higher-order timing compositionality
   - Timing predictability of control and architectural flows (computable)

2. Deterministic Parallel Execution
   - Fork semantic building blocks
   - Same semantic meaning for all (dynamic) schedulings
General Approach

- WCET analysis framework on the (pure) functional setting
  - Declarative – no internal state of the static analyzer
- Extended with a deterministic parallel computational model
- Dynamic construction of dataflow networks (meta-traces)
- Reuse the ILP component used for single-core

Meta-Semantic Formalism

- Polymorphic, provides a parametrized fixpoint semantics for free at denotational level (*WFLP, 2011*)
- Higher-order algebraic properties at relational level
- Decomposes the program with a variable granularity using transformation (*FOPARA, 2011*)
Weak Topological Order

Program Points

- Correspond univocally to program states
- Associated to \textit{abstract invariants} (defined for all program points)
- Ordered in \textit{weak topological ordering} of directed graphs:

\[
I_1^1 \cdots I_n^1 (I_{k_1}^{1} \cdots I_{k_1}^{n_1} (I_{k_2}^{1} \cdots I_{k_2}^{n_2} (I_{k_3}^{1} \cdots ))))
\]

where $I_{k_i}^1$ is necessarily either:

1. an entry point of a procedure (after a \textit{branch-and-link} instruction)
2. the head of an intraprocedural loop (before a \textit{conditional-branch} instruction, \texttt{bgt}, \texttt{beq}, etc.)
3. the hook point on the \textit{caller} procedure after a procedure return (next instruction after a \textit{branch-and-link})
Program Semantics

Relational (Big-Step) Semantics

int main(void) { //28
    int y = factorial(2); //26
    return y; //10
}

int factorial(int a) { //25
    if (a==1)
        return 1; //22
    else {
        a = factorial(a-1); //45
        return a; //28
    }
}

int foo (int x) { //20
    while (x>0) {
        x--;
    }
    return x;
}

w.t.o. is: 0 ... 5 (11 12 ... 16 20 ... (22 17 ... 21 22 ... 25 (6 ... 10)))

iteration strategy: 0 ... 5 11 ... 16 20 21 [22 17 ... 21]* 23 ... 25 6 ... 10
Trace Semantics

- Trace semantics is obtained from the relational semantics by means of a refinement process (Galois connection):
  - Follows every possible program path from the set of relations

Goal

- Meta-traces automatically generated as an interpretation of the weak topological order
- Apply a fixpoint sequential algorithm à la Gauss-Seidel
Relational (Big-Step) Semantics (II)

Transition System $\langle \Sigma, S, \tau \rangle$

- $\Sigma$ is a set of states
- $S$ is the syntactical domain
- $\tau \subseteq (\Sigma \times S \times \Sigma)$ is a ternary relation that, given a syntactical object $s \in S$, establishes a input-output relation between a state and its possible successors

```
data Rel a = (Abstractable a, Stateable a) ⇒ Rel (a, Expr, a)
data Expr = Exec Instruction | Cons Instruction Expr
type BigStep a = [Rel a]
```

Failed attempt: A syntax object is either a plain expression or an expression composed in parallel with the big-step semantics of an entire thread
Denotational Semantics

Relational Abstraction

Given the state vector $\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_l \rangle$, where $l$ is a label identifier, we apply the right-image isomorphism $f$ applied to every transition relation $\tau$:

$$f[\tau] \triangleq \lambda s \cdot \lambda \sigma \cdot \{ \sigma' | \exists \sigma' \in \Sigma : \langle \sigma, s, \sigma' \rangle \in \tau \}$$

**type** $\text{RelAbs} \ a = a \to \text{Par} \ a$

**type** $\text{Invs} \ a = \text{Map LabelId} (\text{IVar} \ a)$

simulate :: $(\text{Cost} \ a) \Rightarrow \text{Instruction} \to (\text{CPU} \ a) \to (\text{CPU} \ a)$

**class** Abstractable $a$ where

apply :: $\text{St} \ a \to \text{Rel} (\text{St} \ a) \to \text{RelAbs} (\text{Invs} \ a) \to \text{Par} (\text{St} \ a)$

lift :: $\text{Rel} (\text{St} \ a) \to \text{RelAbs} a \to \text{RelAbs} (\text{Invs} \ a)$

instance $(\text{Cost} \ a) \Rightarrow \text{Abstractable} (\text{CPU} \ a)$ where

apply $s \ r \ f = (f \circ \text{invs}) \ s \Rightarrow \text{instrument} \ r \ s$

lift $r \ f \ \text{cert} = \text{do} \ \text{cpu} \leftarrow \text{lookup cert} (\text{source} \ r)$

$f \ \text{cpu} \Rightarrow \text{chaotic} (\text{sink} \ r) \ \text{cert}$
Parallel Monad \((\text{Par})\), Peyton Jones \textit{et. al.}\n
**Par monad**

- Explicit granularity with \(\text{IVars}\)
- Denote continuations free from side-effects
- Concurrency model is deterministic, while allowing parallel scheduling implementations

**IVars**

- Communication abstraction between threads
- Either \textit{empty} or \textit{full}, supporting two operations: \textit{put} and \textit{get}
- Establish the dependencies between states and provide thread synchronization upon \textit{get} requests
Intermediate Graph Language

Mimics the execution order of trace semantics and connect the relations $\tau$ in order to obtain a dependency graph $DF$

\[
data \ DG \ a = \text{Empty} \mid \text{Leaf} \ (\text{Rel} \ a) \mid \text{Seq} \ (DG \ a) \ (DG \ a) \\
\mid \text{Rec} \ (DG \ a) \ (DG \ a) \mid \text{Alt} \ (DG \ a) \ (DG \ a) \\
\mid \text{InLv} \ (DG \ a) \ (DG \ a)
\]

- **Sequential, Alternative and Recursive** computational patterns plus the **Interleaving** architectural pattern
- Implicitly conveys the **merge over all paths** fixpoint solution
- To ensure termination, the least fixpoint solution is computed by performing joins at merge points (heads)

\[
0 \ldots 5 \ 11 \ldots 16 \ 20 \ 21 \ [22 \ 17 \ldots 21]^* \ 23 \ldots 25 \ 6 \ldots 10
\]
Two-Level Meta-Language

Two levels: compile (ct) and run (rt) times

\[
ct ::= \begin{array}{l}
ct_1 * ct_2 \mid ct_1 \oplus ct_2 \mid ct_1 || ct_2 \mid ct_1 \& ct_2 \\
ct_1 < ct_2 \mid ct_1 > ct_2 \mid rt
\end{array}
\]

\[
rt ::= \begin{array}{l}
A \mid [A, A] \mid rt_1 \Rightarrow Par rt_2
\end{array}
\]

1. Provide fixpoint semantics at denotational level (data-flow specification at run-time)
2. Using relational higher-order combinators (control-flow at compile-time)
   - Expresses abstract syntax trees of meta-traces: compact representation of the structure and allowed behavior of the program
The right-image isomorphism \( f \) is applied to obtain the system of semantic functions:

\[
F = \langle f^{k_1}_1, f^{k_2}_2, \ldots, f^{k_n}_n \rangle
\]

where \( n \) is the number of relations and \( k_n \) is the set of outgoing state labels.

Each \( f_n \) is partially applied to the syntactical object \( s_n \) so that a function are treated as a binary relation.

The operators (\( * \)), (\( || \)), (\( \oplus \)) and (\( \land \)) are binary operators over relations yielding a new relation.

Mechanism based on the point-free notation that allows the argument passed to be included in the return type.
Relational Abstraction

\[(\ast) :: (a \rightarrow Par b) \rightarrow (b \rightarrow Par c) \rightarrow (a \rightarrow Par c)\]

\[(f \ast g) \; s = f \; s \gg g\]

\[(/): (a \rightarrow Par b) \rightarrow (c \rightarrow Par d) \rightarrow ((a, c) \rightarrow Par (b, d))\]

\[(f / g) \; (s, t) = \text{liftM2} \; (\lambda x \; y \rightarrow (x, y)) \; (f \; s) \; (g \; t)\]

\[+ : (a \rightarrow Par a) \rightarrow (a \rightarrow Par a) \rightarrow (a \rightarrow Par a)\]

\[(f + t) \; s = \text{do} \; s' \leftarrow t \; s \]

\[b \leftarrow \text{jump} \; s \; s' \]

\[\text{if } \neg (\text{fixed} \; s') \land b \; \text{then} \; (f \ast (f + t)) \; \text{else} \; \text{compl} \; s\]

\[\text{abst} : Rel \; (St \; (CPU \; a)) \rightarrow \text{RelAbs} \; (St \; (CPU \; a))\]

\[\text{abst} \; r = \lambda s \rightarrow \text{let} \; \text{make} \; (\text{Exec} \; i) = \lambda \text{cpu} \rightarrow \text{return} \; \$ \; \text{simulate} \; i \; \text{cpu} \; \text{side}\]

\[\text{make} \; (\text{Instrs} \; i \; l) = \lambda \text{cpu} \rightarrow \text{make} \; l \; (\text{simulate} \; i \; \text{cpu} \; \text{side})\]

\[\text{step} = \text{make} \; (\text{instr} \; r)\]

\[\text{eval} = \text{lift} \; (\text{sink} \; r, \text{source} \; r) \; \text{step} \; \text{side}\]

\[\text{in} \; \text{apply} \; s \; r \; \text{eval}\]
Derivation of Meta-Programs

The function \( \text{derive} \) traverses a dependency graph \( DF \) and recursively “compiles” a meta-program using the function \( \text{abst} \)

\[
\text{derive} :: \text{RelAbs} (\text{State} (\text{CPU} a)) \rightarrow DF (\text{State} (\text{CPU} a)) \\
\rightarrow \text{RelAbs} (\text{State} (\text{CPU} a))
\]

\[
\begin{align*}
\text{derive } f \text{ Empty} &= \text{return} \\
\text{derive } f \text{ (Leaf } r \text{)} &= f \ast \text{abst } r \\
\text{derive } f \text{ (Seq } a \text{ } b \text{)} &= \text{derive} (\text{derive } f \text{ } a) \text{ } b \\
\text{derive } f \text{ (Rec (Leaf } r \text{) } g \text{)} &= f \ast ((\text{derive return } g) + \text{abst } r) \\
\text{derive } f \text{ (Par } a \text{ } b \text{)} &= \text{let} \text{ left } = \text{derive return } a \\
& \quad \text{right } = \text{derive return } b \\
& \quad \text{in} \ f \ast \text{split } \ast (\text{left} \div \text{right}) \ast \text{wide}
\end{align*}
\]

- Meta-traces are fed into the static analyzer
- During fixpoint computation actual traces will be created by expanding the relational operators
- Kleenian constructive fixpoint semantics:
  \[
  T^* \triangleq \bigcup_{n \geq 0} T^n \\
  = \bigcup_{n \geq 0} \left( \bigcup_{i \leq n} T^i \right) = \bigcup_{n \geq 0} (\lambda R \cdot 1_{\Sigma} \sqcup (T b R))^i(\bot) = \text{lfp}_\bot \lambda R \cdot 1_{\Sigma} \sqcup (T b R)
  \]
Same semantic meaning, different (sequential) accesses to shared resources:

\[ n. \text{ of } Traces = (Thread_{steps} + 1) \times Main_{interleaved} \]
The WCET of a given meta-trace in a multi-core environment is divided in three phases:

1. **Simultaneous value, cache, pipeline and program flow analysis**
2. **Graph reconstruction** that infers, for every program state, which was the core that processed it
   - Redefine the *flow conservation* constraints
3. **Linear Programming**
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 2

void *ThreadBody(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Inside_thread_%ld\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for (t = 0; t < NUM_THREADS; t++){
        printf("creating_thread_%ld\n", t);
        rc = pthread_create(&threads[t], NULL, ThreadBody, (void *)t);
        if (rc){
            printf("error");
            exit(-1);
        }
    }
    pthread_exit(NULL);
}

<table>
<thead>
<tr>
<th>Thread</th>
<th>CPU cycles in 1 Core</th>
<th>CPU cycles in 2 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>448</td>
<td>347</td>
</tr>
<tr>
<td>ThreadBody</td>
<td>-</td>
<td>101</td>
</tr>
</tbody>
</table>
Objective

- Use the compositional properties of the meta-language in order to analyze the compositional timing behavior of the ARM7

Knowns

- Abstract timing model eliminates timing accidents by stalling the pipeline until the accident is resolved
- The penalties associated to cache misses and bus occupancy are simply added to the best execution time

Requirement

- The composability of the timing behavior of each software component does not lead to unpredictable timing behavior
Shared Resources (II)

First Attempt

- Continuation-based *Par* monad has explicit granularity
- Parallel deterministic execution
- The algebraic properties of the meta-language completely abstracts one task from concurrently running tasks
- From the **access patterns** to these shared resources, add safe constant bounds to the overall execution time: $\lambda C \cdot (T \ast C)$

To be sound is too **pessimistic**!

$\implies$ **Trade-off** between precision and efficiency: Complete bottom-up approach + algebraic composition + design of abstract domains
Achievements in the Denotational Setting

1. Definition of a polymorphic meta-language and a parametrized fixpoint semantics for free:
   - Upper level of the meta-language

2. Data-flow definitions from the abstract interpretation literature are in direct correspondence with declarative code and used as parameters by the fixpoint semantics:
   - Reduces the semantic gap (lower level of the meta-language)

3. Easy to handle recursion: loop unrolling results from the expansion of the recursive operator \( \oplus \) at run-time:
   - Constructive aspect of the fixpoint algorithm

4. "Iteration strategies are not taken as random: it exactly follows the syntactic structure of the program, computing fixpoints as would a denotational semantics do" (Pichardie et al, 2010)