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
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 *abstract invariants* (defined for all program points)
- Ordered in *weak topological ordering* of directed graphs:

\[ l_1^1 \cdots l_n^1 \ (l_{k_1}^1 \cdots l_{k_1}^n \ (l_{k_2}^1 \cdots l_{k_2}^n \ (l_{k_3}^1 \cdots ))) \]

where \( l_{k_i}^1 \) is necessarily either:

1. an entry point of a procedure (after a *branch-and-link* instruction)
2. the head of an intraprocedural loop (before a *conditional-branch* instruction, *bgt*, *beq*, etc.)
3. the hook point on the *caller* procedure after a procedure return (next instruction after a *branch-and-link*)
Relational (Big-Step) Semantics

```c
/* wcet = 336.0 cpu cycles */
int main(void) //28 {
    int y = factorial(2); //26
    return y; //10
}

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

int foo (int x) {
    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

```haskell
data Rel a = (Abstractable a, Stateable a) \Rightarrow 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 $\tau$:

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

```haskell
-- Type signatures
type RelAbs a = a -> Par a
type Invs a = Map LabelId (IVar a)

-- Function signatures
simulate :: (Cost a) \Rightarrow Instruction \rightarrow (CPU a) \rightarrow (CPU a)

-- Class definition
class Abstractable a where
    apply :: St a \rightarrow Rel (St a) \rightarrow RelAbs (Invs a) \rightarrow Par (St a)
    lift :: Rel (St a) \rightarrow RelAbs a \rightarrow RelAbs (Invs a)

instance (Cost a) \Rightarrow Abstractable (CPU a) where
    apply s r f = (f \circ invs) s >>= instrument r s
    lift r f cert = do cpu <- lookup cert (source r)
                       f cpu >>= chaotic (sink r) cert
```
Parallel Monad \((\text{Par})\), Peyton Jones \textit{et. al.}

\textbf{Par monad}

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

\textbf{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 $DF$

$$\begin{align*}
\textbf{data } \text{DG } a & = \text{Empty} \mid \text{Leaf } (\text{Rel } a) \mid \text{Seq } (\text{DG } a) (\text{DG } a) \\
& \mid \text{Rec } (\text{DG } a) (\text{DG } a) \mid \text{Alt } (\text{DG } a) (\text{DG } a) \\
& \mid \text{InLv } (\text{DG } a) (\text{DG } a)
\end{align*}$$

- **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*)

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

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

\[
ct ::= \text{ct}_1 \ast \text{ct}_2 \mid \text{ct}_1 \oplus \text{ct}_2 \mid \text{ct}_1 \&\& \text{ct}_2 \mid \text{ct}_1 \lt\lt \text{ct}_2 \mid \text{rt} \\
rt ::= A \mid [A, A] \mid \text{rt}_1 \rightarrow \text{Par rt}_2
\]

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
Two-Level Meta-Language (II)

- The right-image isomorphism $f$ is applied to obtain the system of semantic functions:

  $$ F = \langle f_1^{k_1}, f_2^{k_2}, \ldots, f_n^{k_n} \rangle $$

  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 $(\ast)$, $(\parallel)$, $(\oplus)$ and $(\wedge)$ 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

\((\ast) :: (a \to \text{Par } b) \to (b \to \text{Par } c) \to (a \to \text{Par } c)\)
\((f \ast g) \ s = f \ s \gg g\)

\((/ \ast) :: (a \to \text{Par } b) \to (c \to \text{Par } d) \to ((a, c) \to \text{Par } (b, d))\)
\((f \ / g) \ (s, t) = \text{liftM2 } (\lambda x y \to (x, y)) \ (f \ s) \ (g \ t)\)

\((+ \ast) :: (a \to \text{Par } a) \to (a \to \text{Par } a) \to (a \to \text{Par } a)\)
\((f + t) \ s = \begin{array}{l}
\text{do } s' \leftarrow t \ s \\
\qquad b \leftarrow \text{jump } s \ s' \\
\qquad \text{if } \neg (\text{fixed } s') \land b \ \text{then } (f \ast (f + t)) \ \text{else } \text{compl } s
\end{array}\)

\(\text{abst} \ :: \ \text{Rel } (\text{St } (CPU \ a)) \to \text{RelAbs } (\text{St } (CPU \ a))\)
\(\text{abst } r = \lambda s \rightarrow \text{let make } (\text{Exec } i) = \lambda \text{cpu} \rightarrow \text{return } $ \text{simulate } i \ \text{cpu} \ \text{side} \)
\(\text{make } (\text{Instrs } i \ l) = \lambda \text{cpu} \rightarrow \text{make } l \ \text{(simulate } i \ \text{cpu} \ \text{side)}\)
\(\text{step } = \text{make } (\text{instr } r)\)
\(\text{eval } = \text{lift } (\text{sink } r, \text{source } r) \ \text{step } \text{side}\)
\(\text{in } \text{apply } s \ r \ \text{eval}\)
Derivation of Meta-Programs

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

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

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

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: \( T^* \triangleq \bigsqcup_{n \geq 0} T^n \)

\[
= \bigsqcup_{n \geq 0} \left( \bigsqcup_{i \leq n} T^i \right) = \bigsqcup_{n \geq 0} (\lambda R \cdot 1_{\Sigma} \sqcup (TB R))^i(\bot) = \text{lfp}_\bot \lambda R \cdot 1_{\Sigma} \sqcup (TB R)
\]
Same semantic meaning, different (sequential) accesses to shared resources:

\[ \text{n. of } Traces = (\text{Thread}_{steps} + 1) \times \text{Main}_{interleaved} \]
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>

Compositional Timing Analysis on Multi-Core Architectures
## Shared Resources

### 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
**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**!

\[ \Rightarrow \textbf{Trade-off} \] 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)