A Declarative Compositional Timing Analysis for Multicores Using the Latency-Rate Abstraction

Vítor Rodrigues
Benny Akesson
Simão Melo de Sousa
Mário Florido

PADL'13
21^{st} of January

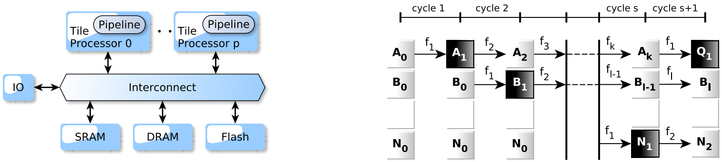
Outline

Introduction

- The main timeliness criteria in embedded real-time systems is the worst-case execution time (WCET). Depends both on:
  1. Structure of the source code: loop iterations.
  2. Timing-influencing hardware components: caches and pipelines.
- The state space of both input data and hardware states is too large to be exhaustively explored $\Longrightarrow$ Abstract Interpretation?
- In general, the complexity of multicore timing analysis is also affected by the predictability of access times to shared resources
  - Unless resources are *composable*, scheduling arbitration produce different intermediate hardware states $\Longrightarrow$ “architectural flows”.
  - In multicore architectures, the number of “architectural flows” (a.k.a. *interleavings*) is not feasible to compute!

Overview of Approach

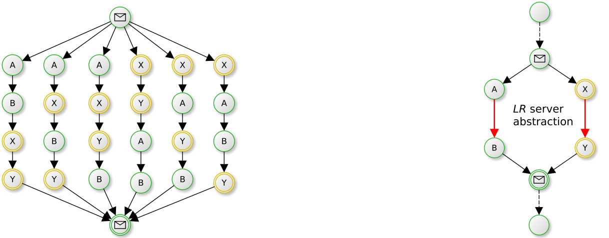
- Architectures with several ARM9 cores with shared resources.
- Each core has cache memories and 5-stages, in-order pipelines.
- The functions $f_1, f_2, \ldots, f_k, \ldots,$ specify the effect of pipeline state transformations across a variable number of pipeline steps.
- "Hybrid" pipeline states include instruction vectors of size N, adjoined with timing properties (CPI), e.g. $1, 2, \ldots, s, s+1$.



(a) Generic multicore architecture (b) Functional overview of pipeline steps

Figure 1: Functional model of a pipeline in a multicore architecture

- Let P_1 and P_2 be two processes running two processor tiles.
- Composability in the value domain $\Longrightarrow$ interleavings depend on the scheduling made by the arbiter of the shared resource.
- For non-composable arbiters, interleavings are required. However, if access times can be predictable $\Longrightarrow$ compositional timing analysis.



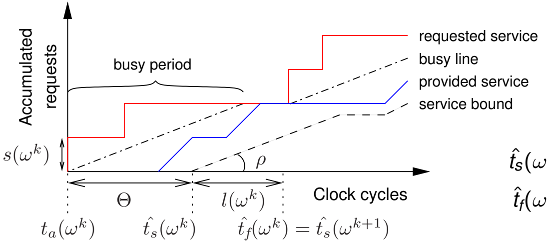
(a) Non-compositional timing analysis with architectural flows between P_1 and P_2

(b) Compositional timing analysis considering only control flows

Figure 2: A and B belong to P_1 and X and Y belong to P_2

Latency-Rate Servers

- Calculate upper bounds on the access times to shared resources to remove the variation in interference between cores.
- The formal of $\mathcal{LR}$-server model provides a timing abstraction applicable to most resource arbiters, e.g. TDM and RR . . .
- Model parameters: guarantees a minimum allocated bandwidth, ρ, after a maximum service latency (interference), Θ.



$$\hat{t}_s(\omega^k) = \max(t_a(\omega^k) + \Theta, \hat{t}_f(\omega^{k-1}))$$
$$\hat{t}_f(\omega^k) = \hat{t}_s(\omega^k) + s(\omega^k)/\rho$$

Meta-Semantic Formalism

- Constructive fixpoint semantics based on expressions of a two-level denotational meta-language aiming at compositionality.
- Automatically generate type-safe abstract interpreters for free for a variety of control-flow patterns, including architectural flows.
- Domain definitions are factored into a *core semantics* at compile-time (ct) and *abstract interpretation* at run-time (rt).

$$\mathrm{ct} \triangleq \mathrm{ct}_1 * \mathrm{ct}_2 \mid \mathrm{ct}_1 \mid\mid \mathrm{ct}_2 \mid \mathrm{ct}_1 \oplus \mathrm{ct}_2 \mid \mathrm{ct}_1 \oslash \mathrm{ct}_2 \mid \text{split rt} \mid \text{merge rt} \mid \mathrm{rt}$$

$$\mathrm{rt} \triangleq \Sigma \mid (\Sigma \times \Sigma) \mid \mathrm{rt}_1 \to \mathrm{rt}_2$$

- Using *refunctionalization* and *chaotic iteration strategies*, interpretations of the higher-order (point-free) combinators generate the “code” of a *meta*-program. Let T be a relation:

$$T^{\star} \triangleq \bigsqcup_{n \geqslant 0} T^{n} = \bigsqcup_{n \geqslant 0} \left(\bigsqcup_{i \leqslant n} T^{i} \right) = \bigsqcup_{n \geqslant 0} (\lambda R \bullet ((u\,T)\; b\; (u\,R)))^{i} (\bot_{\Sigma})$$

Automatic Generation of Fixpoint Interpreters

- The relational semantics of a program P is the set of input-output relations $\tau \subseteq (\Sigma_P \times \mathit{Instrs} \times \Sigma_P)$.
- The abstract syntax of program paths is a *dependency graph*, defined by $(\mathbf{G}\ a)$, that represents a mimic of the execution order.

$$\textbf{data Rel } a = (a, \mathit{Instr}, a)$$

$$\begin{aligned}\textbf{data G } a = {} & \textbf{Empty} \mid \textbf{Leaf}\ (\textbf{Rel}\ a) \mid \textbf{Seq}\ (\textbf{G}\ a)\ (\textbf{G}\ a) \mid \textbf{Unroll}\ (\textbf{G}\ a)\ (\textbf{G}\ a) \\ \mid {} & \textbf{Unfold}\ (\textbf{G}\ a)\ (\textbf{G}\ a) \mid \textbf{Choice}\ (\textbf{Rel}\ a)\ (\textbf{G}\ a)\ (\textbf{G}\ a) \mid \textbf{Conc}\ (\textbf{G}\ a)\ (\textbf{G}\ a)\end{aligned}$$

- Derivation of *meta*-programs: syntactic phrases are dependency graphs and denotations are the *core semantics*.
- Combinators are compiled into typed-λ-calculus. For example:

$$\begin{aligned}&(\ast) :: (a \to b) \to (b \to c) \to (a \to c)\\&(f \ast g) = \lambda s \to (g \circ f)\ s\end{aligned}$$

- Algebra of higher-order relational combinators.
- The function *derive* generates *meta*-programs with unified type.
- The function *refunct* performs the refunctionalization of the datatype (**Rel** *a*).

```
derive :: (a → a) → G a → (a → a)
derive p (Leaf r)     = p ∗ refunct r
derive p (Seq a b)    = derive (derive p a) b
derive p (Conc a b) = let is = interleavings a b
                              ms = map (derive (create b)) is
                          in p ∗ scatter (length ms) ∗ (distribute ms) ∗ reduce
```

- The function *interleavings* obtains the set of architectural flows.
- The function *create* initializes the hardware state of the 2^{nd} core.

```
scatter :: Int → a → [a]
scatter = replicate

distribute :: [a → a] → [a] → [a]
distribute = zipWith (λf a → f a)

reduce :: (Lattice a) ⇒ [a] → a
reduce = foldl join bottom
```

Pipeline Analysis

- Pipeline analysis by abstract interpretation introduces the notion of *resource association*. An “hybrid” pipeline state P is:

$$\begin{aligned} P &\triangleq (\mathit{Time} \times \mathit{Pc} \times \mathit{Demand} \times R'^{\sharp} \times D'^{\sharp} \times M'^{\sharp} \times \mathit{Coord}) \\ \mathit{Coord} &\triangleq [\mathit{TimedTask}]_N \\ \mathit{TimedTask} &\triangleq (\mathit{Cycles} \times \mathit{Stage} \times \mathit{Task}) \\ \mathit{Task} &\triangleq (\mathit{Instr} \times \mathit{Pc} \times \mathit{Demand} \times R'^{\sharp} \times D'^{\sharp} \times M'^{\sharp}) \end{aligned}$$

- Our functional approach to pipeline analysis is done at 3 levels:
  1. The transformer F_T as a morphism on the domain *TimedTask*;
  2. The transformer F_P as a morphism on the domain P, which uses F_T to compute the new elements inside the N-sized vector *Coord*;
  3. The transformer $F_P^{\sharp}$ as a morphism on sets of hybrid states $P^{\sharp} \triangleq 2^P$, using F_P to transform the hybrid states in the input set.

$$F_P(i)(p) \triangleq toContext(i) \circ [F_T \circ fromContext(p)]_N$$

$$F_P^{s^i_{k+1}}(i)(p) \triangleq F_P(i)(F_P^{s^i_k}(i)(p))$$

$$F_P^{5+} \triangleq F_P^{s^i_{WB}}$$

$$F_P^\sharp(i)(p^\sharp) \triangleq \{F_P^{5+}(i)(p) \mid p \in p^\sharp\}$$

- Consider F_T when the current *Stage* is **FI** (“Fetch”):

```
fetchInstr :: (Cycles a) ⇒ a → Task → TimedTask a
fetchInstr cycles t@Task { taskNextPc = pc, taskImem = m}
  = let (classification, opcode, m′) = getMem♯ m pc
          i = decode opcode
          buffer′ = setReg♯ bottom R15 (pc + 4)
    in if classification ≡ Hit
          then let t′ = t { taskInstr = i, taskNextPc = pc′, taskImem = m′ }
                 in TimedTask { property = fetched cycles, stage = DI,
                                task = Fetched t′ buffer′ }
          else let t′ = t { taskInstr = i, taskNextPc = pc′, taskImem = m′ }
                 in TimedTask { property = missed p, stage = FI,
                                task = Stalled Structural t′ buffer′ }
```

Haskell definitions for resource sharing

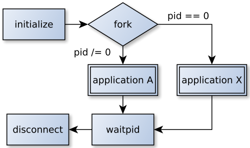
- Let **WCET** be an instantiation of the type class (*Cycles a*).

```
data WCET = WCET { cycles :: Int, ta :: Int, core :: Int, tf :: Int, delay :: Int }
```

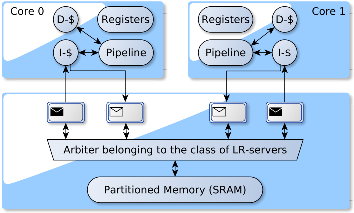
- According to the $\mathcal{LR}$-model, the function *missed* defines a cache miss in terms of an arrival time (*ta*) and a previous finish time (*tf*):

```
missed w@WCET { cycles = c, ta, tf }
  = let busy = ta + theta < tf
        d    = if busy then 1/rho else theta + (1/rho)
        tf'  = if busy then tf else ta
    in w { cycles = c + round d, tf = tf' + d, delay = d }
```

- Example:



(a) Example of a multi-process program



(b) Simplified multicore architecture

Experimental Results

- Assume large private data cache memories (D-\$).
- TDM arbitration with frame size of 2.

Table 1: Comparison results for architectural flows, composable TDM

No. instructions "application A"	No. instructions "application X"	No. of interleavings	Results (CPU cycles/sec.)	Architectural Flows (TDM)	Composable TDM
4	5	126	WCET	179	185
			Analysis Time	57.0	0.17
5	5	252	WCET	188	188
			Analysis Time	140.3	0.18
6	5	462	WCET	195	195
			Analysis Time	588.7	0.43

- $\mathcal{LR}$ abstraction with $\Theta = 1$ and $\rho = 0.5$.
- Every request requires $\Theta + 1/\rho$ cycles to complete.

Table 2: WCET results for some of the Mälardalen benchmarks

Benchmark	No. Source Loop Iterations	$\mathcal{LR}$-server (WCET)	No. Cache Misses	TDM (WCET)	Overhead (%)	Analysis Time in sec. ($\approx$)
bsort	156	1459	152	1311	10.1	0.9
crc	459	3160	304	2826	10.6	15.0
fibcall	111	994	59	885	11.0	2.3
matmult	287	2580	188	2343	9.2	5.2

Summary

- The type system of Haskell is used to define a type safe and parameterizable denotational fixpoint semantics.
- Fixpoint compositional algorithms are automatically generated, unifying first-order data flow with higher-order control flow.
- The complexity of the multicore analysis is reduced by using a provably sound $\mathcal{LR}$ abstraction on resource scheduling.
- For a simplified architecture, the compositional timing analysis in multicore environments yields:
    - loss of precision in order of 10% on average;
    - factor 100 reduction in terms of analysis time.
- The precision of the WCET is very sensitive to the architecture considered.