Dependent Labelling for Pipelines and Caches

Claudio Sacerdoti Coen

May 16, 2013

SEVENTH FRAMEWORK PROGRAMME

Outline

Outline

Problem Statement

The basic labelling approach statically assigns a constant cost to every labelled basic block but

the execution cost on modern hardware is parametric on the internal state of hardware components, which is history dependent

Towards a solution

The dependent labelling approach statically assigns to every labelled basic block a cost parametric on the state

and it lifts the cost from the object to the source code

Problems:

1. What cost models can be lifted? How?
   (the internal hardware state is not visible in the source code)
2. Can costs associated to pipelines/caches be lifted?
3. Can we reason on the source code on the lifted cost models?

Plan of the talk

1. We provide a generic, abstract description of modern hardware
2. We introduce restrictions on the cost models and a notion of view as an approximation of states
3. We show how to instrument the source code with low level state variables (for views) and update functions (over views)
4. We argue that simple pipelines can be analyzed in this way
5. We describe the difficulties with caches and hint at research directions to also include them

Outline

Hardware state

Hardware state

Hardware state $= \Sigma \times \Delta$

Σ = data states (registers, PC, memory, ...)

Δ = internal state of pipelines, caches, ...

Δ does not affect the functional behaviour.

Decoded instructions

Decoded instruction

Decoded instruction $= \mathcal{I} \times \Gamma$

$\mathcal{I} =$ instructions (opcodes)

$\Gamma =$ operands after fetching and decoding

Fetching

$fetch : \Sigma \rightarrow \mathcal{I} \times \Gamma$

Not the actual fetching that uses Δ too.

Abstract semantics

Semantics

$\longrightarrow: \Sigma \to \Sigma$	data state transition
$\Longrightarrow: \Sigma \times \Delta \to \Delta$	internal state transition
$K : \mathcal{I} \times \Gamma \times \Delta \to \mathbb{N}$	cost function

Execution history and path

Execution history: $\sigma_0 \longrightarrow \sigma_1 \longrightarrow \sigma_2 \ldots$

Execution path: the stream of program counters only

The execution path is what the labelling approach captures.

Outline

Approximated and operand insensitive models

Exact vs approximated models

K is exact if the cost is real, it is approximated if it returns an upper bound.

Operand insensitive models

$K : \mathcal{I} \times \cancel{\Gamma} \times \Delta \rightarrow \mathbb{N}$

Most modern architectures are either operand insensitive or have approximated operand insensitive models with modest jitter. WCET tools use these models.

Operand sensitivity is problematic for the labelling approach (how to map Γ to the source world?).

Views and cost dependent on views only

Views

$(\mathcal{V}, |.|)$ where
$\mathcal{V}$ is a finite non empty set
$|.| : \Delta \to \mathcal{V}$ is forgetful

Cost functions dependent on a view only

$K' : \mathcal{I} \times \mathcal{V} \to \mathbb{N}$

$K(I, \delta) = K'(I, |\delta|)$

With an abuse of terminology, we will identify K with K'.

Execution history dependent views

Execution history dependent views

$(\mathcal{V}, |.|)$ is execution history dependent with lookahead n if there exists $\hookrightarrow : PC^n \times \mathcal{V} \to \mathcal{V}$ such that for all (σ, δ) and $pc_1, \ldots, pc_n$ such that every pc_i is the program counter of σ_i defined by $\sigma \longrightarrow^i \sigma_i$, we have $(\sigma, \delta) \Longrightarrow \delta'$ iff $((pc_1, \ldots, pc_n), |\delta|) \hookrightarrow |\delta'|$.

Notes:

1. The evolution of the views can be predicted from the execution paths only, that are the ones we track in the labelling approach
2. The lookahead is in the future (e.g. for pipelines or for determining the outcome of a conditional branch)
3. Can be lifted compositionally to $EP \times \mathcal{V} \to \mathcal{V}$ (compositional = associativity of lifting)

Data independent cost models

Data independent cost models

K is data independent if it is dependent on a view that is execution path dependent.

Outline

A simple pipeline model

Assumptions:

- Pipeline with n stages without branch prediction and hazards
- The actual value of the operands has no influence on stalls nor on the execution cost. The type of operands, however, can.

For example, reading the value 4 from a register may stall a pipeline

States and a view on them

Pipeline states

$\Delta = (\mathcal{I} \times \Gamma \cup \mathbb{1})^n$
(an n-uple of decoded instructions or bubbles)

Not the real state.

A data-independent view

$(\{0,1\}^n, |.|)$ where
$|(x_1, \ldots, x_n)| = (y_1, \ldots, y_n)$ and y_i is 1 iff x_i is a bubble.

A view is representable by a single integer.

Cost model and update function

$K : PC \times \{0,1\}^n \to \mathbb{N}$
$\hookrightarrow : PC^n \times \{0,1\}^n \to \{0,1\}^n$

n look-aheads are required because of prefetching

Outline

Assumptions

1. The cost model is data independent. It is defined by $(\hookrightarrow, K)$ over the view $(\mathcal{V}, |.|)$
2. Every function call in C is immediately followed by a label

Notes:

- Condition 2 can easily be enforced
- It puts more burden on the management of cost annotations and requires proofs that every call is terminating
- The labelling approach under 2 only requires associativity of cost composition; without 2 it also requires commutativity (which fails when the state is history dependent)
- Structured traces still necessary (can be simplified)

Static analysis

We statically compute:

Transition function over labels

$\hookrightarrow : \mathcal{L} \times \mathcal{L} \times \mathcal{V} \to \mathcal{V}$
$(L, L', v) \hookrightarrow v'$ iff $((pc_0, \ldots, pc_n, pc'_0, \ldots, pc'_m), v) \hookrightarrow v'$ where $(pc_0, \ldots, pc_n)$ are the program counters of the block labelled by L and $(pc'_0, \ldots, pc'_m)$ are those of the block labelled with L'.

The lookahead is lifted from PCs to the next label
(Skipping some problems/solutions here)

Cost function over labels

$K : \mathcal{V} \times \mathcal{L} \to \mathbb{N}$
$K(v_0, L) = K(pc_0, v_0) + K(pc_1, v_1) + \ldots + K(pc_n, v_n)$
where for every $i < n$, $((pc_i, \ldots, pc_{i+l}), v_i) \hookrightarrow v_{k+1}$
where l is the lookahead and $pc_0, \ldots, pc_n$ the PCs of the instructions in the basic block

Static analysis and the preservation of the cost model

The static analysis should be performed for every initial view.
E.g. for pipelines, it requires 2^n code traversals.

The cost of the static analysis remains linear in the program size.

Both the cost and the update functions are compositionally lifted to work over labelled traces (requires proof of associativity).

The proof of correctness of the static analysis is preserved.

The proof of preservation of labelled traces is preserved.

Corollary: the cost computed for a labelled trace on the labelled source code is the real cost of the object code.

Code instrumentation

Problem:

the current view is not visible in the source code, but is necessary to express costs and infer cost invariants

Solution:

- The current view is exposed in the code via a global variable `__view`
- The K function is exposed in the code and used to update the global `__cost` variable
- A global variable `__label` holds the last visited label
- The $\hookrightarrow$ transition function is exposed in the source code and used with $\{$__label$\}$ to update the `__view`

```
int fact(int n)
{
  int i;
  int res;

  res = 1;
  for (i = 1; i <= n; i = i + 1) {

    res = res * i;
  }

  return res;
}

int main(void)
{
  int t;

  t = fact(10);

  return t;
}
```

```
int fact(int n)
{
  int i;
  int res;

  res = 1;
  for (i = 1; i <= n; i = i + 1) {

    res = res * i;
  }

  return res;
}

int main(void)
{
  int t;

  t = fact(10);

  return t;
}
```

```
int __next(int label1, int label2, int view) {
       if (label1 == 0)                                  return 0;
  else if (label1 == 0 && label2 == 1)                   return 1;
  else if (label1 == 1 && label2 == 2)                   return 2;
  else if (label1 == 2 && label2 == 2 && view == 2) return 3;
  else if (label1 == 2 && label2 == 2 && view == 3) return 2;
  else if (label1 == 2 && label2 == 3 && view == 2) return 1;
  else if (label1 == 2 && label2 == 3 && view == 3) return 0;
  else if (label1 == 3 && label2 == 4 && view == 0) return 0;
  else if (label1 == 3 && label2 == 4 && view == 1) return 0;
}
```

```
int __K(int view, int label) {
       if (view == 0 && label == 0) return 3;
  else if (view == 1 && label == 1) return 14;
  else if (view == 2 && label == 2) return 35;
  else if (view == 3 && label == 2) return 26;
  else if (view == 0 && label == 3) return 6;
  else if (view == 1 && label == 3) return 8;
  else if (view == 0 && label == 4) return 6;
}
```

```
int __cost = 8;
int __label = 0;
int __view;

void __cost_incr(int incr) {
  __cost = __cost + incr;
}

int fact(int n)
{
  int i;
  int res;
  __view = __next(__label,1,__view); __cost_incr(_K(__view,1)); __label = 1;
  res = 1;
  for (i = 1; i <= n; i = i + 1) {
     __view = __next(__label,2,__view); __cost_incr(__K(__view,2)); __label = 2;
    res = res * i;
  }
  __view = __next(__label,3,__view); __cost_incr(K(__view,3)); __label = 3;
  return res;
}

int main(void)
{
  int t;
  __view = __next(__label,0,__view); __cost_incr(__K(__view,0)); __label = 0;
  t = fact(10);
  __view = __next(__label,4,__view); __cost_incr(__K(__view,4)); __label = 4;
  return t;
}
```

Open Questions and problems

1. How does the invariant generators work on the instrumentation?
2. Are automatic provers still performant enough?
3. Recover all techniques used in WCET for graceful degradation (e.g. full unrolling, de-parameterization of blocks with log jitter, etc.)
4. How to reduce the size of the cost and update functions (e.g. by transfering source code control flow analysis to the object code to determine the reachable views or by performing abstract interpretation over the object code)

Requires full implementation and testing.

Outline

Dealing with caches

Major problem:

Caches do not have a data independent cost model

PROARTIS:

Hardware producers should provide probabilist caches that admit a data independent probabilist cost model

Basic idea: the placement and replacement algorithms are based on a uniform distribution. Lower performances in the average case (but not much worse), higher predictability.

Static Probabilistic Time Analysis

Basic idea (from PROARTIS and alts.):

1. Integer costs are replaced by integral random variables
2. Addition is replaced by convolution (which is associative)
3. Static probabilistic cache analysis for basic blocks is explained in PROARTIS papers
   (but some problems are not spotted and solved...)

No changes are necessary to the (dependent) labelling approach!

Open Questions and problems

1. How can we reason symbolically on convolutions?
   (computing them takes exponential time/space; how does PROARTIS planned to solve the issue?)
2. The cardinality of the set of cache states is too large:
   how to induce views dinamically?
   (e.g. by transfering source code control flow analysis to the object code to determine the reachable views or by performing abstract interpretation over the object code)
3. Is it possible to handle a larger set of cost models to do non probabilistic case anlysis with the labelling approach?
   (same techniques as above)

Requires further theoretical investigations before implementation and testing.