Indexed Labels for Loop Iteration Dependent Costs

Paolo Tranquilli

DISI – Università di Bologna Alma Mater

ETAPS '13, CerCo/QAPL Workshops
March 23, 2013

The CerCo project in a nutshell

Cerco:
Certified Complexity

The aim:
A compiler that is able to lift sound and precise resource consumption infos from object to source code, in a compositional and mechanically certified way

The technique:
The labeling approach (more on next slide)

The first target architecture:
The still widely used 8051 microcontroller (no cache, no pipeline, predictable clock timings)

The original labeling approach

- Inject cost labels at key points in source
- Propagate them during compilation
- Assign costs to labels via static analysis of the compiled code, lift them to source
- Each label must thus correspond to a block with $O(1)$ cost
- Paramount conditions for the labeling approach:
  in the compiled code labels occur
  - in each loop (for correctness)
  - at every branching (for preciseness)

Running example – Labeling

$$
\begin{array}{l}
p \leftarrow 1\\
i \leftarrow 2\\
\texttt{while}\ i < n\ \texttt{do}\\
\quad p \leftarrow p * i\\
\quad i \leftarrow i + 1
\end{array}
\quad \mapsto \quad
\begin{array}{l}
\alpha : p \leftarrow 1\\
i \leftarrow 2\\
\texttt{while}\ i < n\ \texttt{do}\\
\quad \beta : p \leftarrow p * i\\
\quad i \leftarrow i + 1\\
\gamma : \texttt{skip}
\end{array}
$$

trace: $\alpha \cdots\cdots \beta \cdots\cdots \beta \cdots\cdots \beta \cdots\cdots \gamma \cdots\cdots$

costs: 122 484 484 484 41

Statically computed costs: $\kappa(\alpha) = 122$, $\kappa(\beta) = 484$, $\kappa(\gamma) = 41$

Limits of the original approach

- Paramount conditions for the labeling approach: in the compiled code labels occur
  - in each loop (for correctness)
  - at every branching (for preciseness)
- If they are ensured in source code, the above can still fail if
  1. a high level instruction is mapped to a non-sequential block
  2. transformations rearrange the code (e.g. loop optimisations)
  3. the execution cost is context-dependent (e.g. cache, pipeline)
- Common problem: cost labels occurring with different costs
- **Our solution:** dependent cost labels!

What loop optimisations?

Loop peeling:

```
while b do S  ↦  if b then
                   S
                   while b do S
```

Loop unrolling:

```
while b do S  ↦  while b do
                   S
                   (if b then
                      S        )*
```

(more specialized form of loop unrolling are the norm. . .)

Running example – peeling

```
α : p ← 1
i ← 2
while i < n do
    β : p ← p ∗ i
    i ← i + 1
γ : skip
```

$\mapsto$

```
α : p ← 1
i ← 2
if i < n do
    β : p ← p ∗ i
    while i < n do
        β : p ← p ∗ i
        i ← i + 1
γ : skip
```

trace: $\alpha \cdots\cdots \beta \cdots\cdots \beta \cdots\cdots \beta \cdots\cdots \gamma \cdots\cdots$

costs: 42 41 246 246 31

Statically computed costs: $\kappa(\alpha) = 42$, $\kappa(\beta) =$???, $\kappa(\gamma) = 31$

Variable costs occur also due to cache or pipeline

Dependent labels: peeling

$\alpha : p \leftarrow 1$
$i \leftarrow 2$
i_0 : `while` $i < n$ `do`
  $\beta\langle i_0 \rangle : p \leftarrow p * i$
  $i \leftarrow i + 1$
γ : `skip`

$\mapsto$

$\alpha : p \leftarrow 1$
$i \leftarrow 2$
`if` $i < n$ `do`
  $\beta\langle 0 \rangle : p \leftarrow p * i$
  i_0 : `while` $i < n$ `do`
    $\beta\langle i_0 + 1 \rangle : p \leftarrow p * i$
    $i \leftarrow i + 1$
γ : `skip`

trace:	$\alpha \cdots\cdots \beta\langle 0 \rangle \cdots\cdots \beta\langle 1 \rangle \cdots\cdots \beta\langle 2 \rangle \cdots\cdots \gamma \cdots\cdots$
costs:	42 41 246 246 31

$\kappa(\alpha) = 42$, $\kappa(\beta) =$??? , $\kappa(\gamma) = 31$

Dependent labels: peeling

$$\begin{array}{l} \alpha : p \leftarrow 1 \\ i \leftarrow 2 \\ i_0 : \texttt{while}\ \ i < n\ \ \texttt{do} \\ \quad \beta\langle i_0 \rangle : p \leftarrow p * i \\ \quad i \leftarrow i + 1 \\ \gamma : \texttt{skip} \end{array} \quad \mapsto \quad \begin{array}{l} \alpha : p \leftarrow 1 \\ i \leftarrow 2 \\ \texttt{if}\ \ i < n\ \ \texttt{do} \\ \quad \beta\langle 0 \rangle : p \leftarrow p * i \\ \quad i_0 : \texttt{while}\ \ i < n\ \ \texttt{do} \\ \quad\quad \beta\langle i_0 + 1 \rangle : p \leftarrow p * i \\ \quad\quad i \leftarrow i + 1 \\ \gamma : \texttt{skip} \end{array}$$

trace:	$\alpha \cdots\cdots \beta\langle 0 \rangle$	$\cdots\cdots \beta\langle 1 \rangle$	$\cdots\cdots \beta\langle 2 \rangle$	$\cdots\cdots \gamma$	$\cdots\cdots$
costs:	42	41	246	246	31

$$\kappa(\alpha) = 42,\ \kappa(\beta) = (i_0 == 0)?41 : 246,\ \kappa(\gamma) = 31$$

Dependent labels: unrolling

$\alpha : p \leftarrow 1$
$i \leftarrow 2$
$i_0 : \texttt{while } i < n \texttt{ do}$
  $\beta\langle i_0 \rangle : p \leftarrow p * i$
  $i \leftarrow i + 1$
$\gamma : \texttt{skip}$

$\mapsto$

$\alpha : p \leftarrow 1$
$i \leftarrow 2$
$i_0 : \texttt{while } i < n \texttt{ do}$
  $\beta\langle 2 * i_0 \rangle : p \leftarrow p * i$
  $i \leftarrow i + 1$
  $\texttt{if } b \texttt{ then}$
    $\beta\langle 2 * i_0 + 1 \rangle : p \leftarrow p * i$
    $i \leftarrow i + 1$
$\gamma : \texttt{skip}$

trace: $\alpha \cdots\cdots \beta\langle 0 \rangle \cdots\cdots \beta\langle 1 \rangle \cdots\cdots \beta\langle 2 \rangle \cdots\cdots \gamma \cdots\cdots$
costs: 42 246 230 246 31

$$\kappa(\alpha) = 42,\ \kappa(\beta) = (i_0 \% 2 == 0) ? 246 : 230,\ \kappa(\gamma) = 31$$

The loop indexed labels approach in brief

- Annotate loops with indexes, which parametrize labels
- Loop optimisations transform these parameters
- Semantics keeps track of indexes, and compilation propagates them
  (no added difficulty to proofs of compilation passes)
- Dependent costs for labels are given with conditional expressions

The loop indexed labels approach in brief

- Annotate loops with indexes, which parametrize labels
- Loop optimisations transform these parameters
- Semantics keeps track of indexes, and compilation propagates them
  (no added difficulty to proofs of compilation passes)
- Dependent costs for labels are given with conditional expressions

Indexed labeling

- Labeling function $\mathcal{L}$ maps to labeled code
- It is parametrized with fresh indexes, initially unmodified:

$$\mathcal{L}\langle I\rangle(\texttt{while } b \texttt{ do } S) := \begin{array}{l} i_k\texttt{: while } b \texttt{ do} \\ \quad \alpha\langle I, i_k\rangle\texttt{: } \mathcal{L}\langle I, i_k\rangle(S) \\ \beta\langle I\rangle\texttt{: skip} \end{array}$$

where

- the loop is single-entry (important in the presence of `goto`s)
- i_k is different from indexes of containing loops (in fact, i_k can be sequence of fresh identifiers, k loop nesting)

The loop indexed labels approach in brief

- Annotate loops with indexes, which parametrize labels
- Loop optimisations transform these parameters
- Semantics keeps track of indexes, and compilation propagates them
(no added difficulty to proofs of compilation passes)
- Dependent costs for labels are given with conditional expressions

Loop transformations

Loop peeling

$i_k : \texttt{while } b \texttt{ do } S$ $\mapsto$

```
if b then
  S[i_k ↦ 0]
  i_k : while b do
    S[i_k ↦ i_k + 1]
```

Loop unrolling

$i_k : \texttt{while } b \texttt{ do } S$ $\mapsto$

```
i_k : while b do
  S[i_k ↦ 2 * i_k]
  if b then
    S[i_k ↦ 2 * i_k + 1]
```

Simple expressions generated by these transformations:

$$s ::= a * i_k + b$$

The loop indexed labels approach in brief

- Annotate loops with indexes, which parametrize labels
- Loop optimisations transform these parameters
- Semantics keeps track of indexes, and compilation propagates them
  (no added difficulty to proofs of compilation passes)
- Dependent costs for labels are given with conditional expressions

Indexes in source semantics

- Separate store for indexes: constant indexings C
- Needed operations:
  - $L \circ C$ evaluates a label (e.g. $\alpha\langle 2 * i_0 + 1\rangle \circ (i_0 \mapsto 2) = \alpha\langle 5\rangle$)
  - $C[i_k \downarrow 0]$ denotes setting i_k as 0 in C
  - $C[i_k \uparrow]$ denotes increment of i_k in C
- Unexciting management of indexes with active loops etc.
- $L : S \xrightarrow{L \circ C} S$: labels are emitted relative to C

Intermediate and target languages

- As loop structure is lost along compilation, indexes need to be managed elsewhere
- In each language down the compilation chain, add explicit pseudo-instructions:

emit cost label:	**emit** L	$\leftrightarrow$	$L \circ C$
index reset:	**reset** i_k	$\leftrightarrow$	$C[i_k \downarrow 0]$
index increment:	**inc** i_k	$\leftrightarrow$	$C[i_k \uparrow]$

Semantics preservation

- $P, \mathbb{S} \xrightarrow{\lambda}^* P', \mathbb{S}'$ iff $\mathcal{T}(P), \mathcal{T}(\mathbb{S}) \xrightarrow{\lambda}^* \mathcal{T}(P'), \mathcal{T}(\mathbb{S}')$
  Optimisations are particular kinds of transformations
- Only loop optimisations and the first pass use indexedness of labels
  All other passes are parametric in the type of cost labels:
  no added difficulty

The loop indexed labels approach in brief

- Annotate loops with indexes, which parametrize labels
- Loop optimisations transform these parameters
- Semantics keeps track of indexes, and compilation propagates them
  (no added difficulty to proofs of compilation passes)
- Dependent costs for labels are given with conditional expressions

Loop indexed costs

- All $\alpha\langle I\rangle$ in compiled code get a cost $\tau(\alpha\langle I\rangle) \in \mathbb{N}$
- Costs lifted to α giving expression $\tau(\alpha)$.
  That depends on the set of transformations
- E.g. $\alpha\langle 2 * i_0 + 1\rangle$ contributes when $i_0\%2 == 1$

Simple expressions: $s ::= a * i_k + b$

$\updownarrow$

Simple conditions:
$$\begin{cases} i_k == b & (a = 0) \\ i_k >= b & (a = 1) \\ i_k\%a == b' \;\&\&\; i_k >= b & (a > 1, b' = b \bmod a) \end{cases}$$

Towards cache analysis

- To exploit cache analysis in loops virtual loop peeling is performed
- Indexed labels allow to handle such virtual loop peeling
- Global abstract interpretation yields a cost per instruction
- Analysis categorizes variables in:
    - Always hit
    - Persistent: every access but the first is a hit
    - Other
- We can implement cache analysis for 8051 extensions by applying dependent costs.

Conclusions

Not shown here: instrumentation, dependent cost simplifications, implementation details

Perspectives:

- Abstract algebra for simple expressions/conditions?
- Loop optimisation is interesting in this framework, as it can be driven by cost annotations
- Dependency could be extended to variables. For example: loop reversing ($i_k \mapsto n - i_k$) or simple instructions compiled with branching code (e.g. shift in 8051)
- Accomodating pipeline (more in Gabriele Pulcini's talk, 16:00 in room E)