

CerCo Work Package 5

Indexed Labels for Dependent Costs

Paolo Tranquilli

Unibo

March 16, 2012



Achievements

- Theoretical development of indexed labeling approach
- Application to the untrusted prototype:
 - Implementation of indexed labels
 - Optimisations (peeling, unrolling, constant/copy propagation, lazy code motion, simple peephole opt.)
 - Immediate arguments along compilation chain
 - Branch of Frama-C cost plugin



Quick recap on the labeling approach

- Inject **cost labels** at key points in the code
- Propagate them during compilation
- Assign costs to labels measuring the compiled code, lift them to source
- 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

$p \leftarrow 1$		$\alpha : p \leftarrow 1$
$i \leftarrow 2$		$i \leftarrow 2$
while $i < 7$ do	\mapsto	while $i < 5$ do
$p \leftarrow p * i$		$\beta : p \leftarrow p * i$
$i \leftarrow i + 1$		$i \leftarrow i + 1$
		$\gamma : \text{skip}$

trace: $\alpha \cdot \dots \cdot \beta \cdot \dots \cdot \beta \cdot \dots \cdot \beta \cdot \dots \cdot \gamma \cdot \dots$
costs: 122 484 484 484 41

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



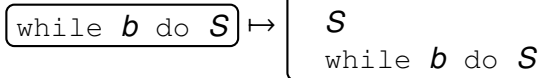
Limits of the previous approach

- 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)
- Common problem: cost labels occurring with different costs
- **Our solution:** dependent cost labels!

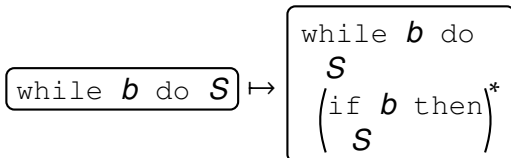


What loop optimisations?

Loop peeling:



Loop unrolling:



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



Running example – peeling

```

α : p ← 1
i ← 2
while i < 5 do
  β : p ← p * i
  i ← i + 1
γ : skip

```

↪

```

α : p ← 1
i ← 2
if i < 5 do
  β : p ← p * i
  while i < 5 do
    β : p ← p * i
    i ← i + 1
  γ : skip

```

trace: $\alpha \dots \beta \dots \beta \dots \beta \dots \gamma \dots$

costs: 42 41 246 246 31

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

Variable costs occur also with **caching**



Dependent labels: peeling

$\alpha : p \leftarrow 1$

$i \leftarrow 2$

$i_0 : \text{while } i < 5 \text{ do}$

$\beta\langle i_0 \rangle : p \leftarrow p * i$

$i \leftarrow i + 1$

$\gamma : \text{skip}$

\mapsto

$\alpha : p \leftarrow 1$

$i \leftarrow 2$

if $i < 5$ do

$\beta\langle 0 \rangle : p \leftarrow p * i$

$i_0 : \text{while } i < 5 \text{ do}$

$\beta\langle i_0 + 1 \rangle : p \leftarrow p * i$

$i \leftarrow i + 1$

$\gamma : \text{skip}$

trace: $\alpha \cdots \beta\langle 0 \rangle \cdots \beta\langle 1 \rangle \cdots \beta\langle 2 \rangle \cdots \gamma \cdots$

costs: 42 41 246 246 31

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



Dependent labels: peeling

$\alpha : p \leftarrow 1$

$i \leftarrow 2$

$i_0 : \text{while } i < 5 \text{ do}$

$\beta\langle i_0 \rangle : p \leftarrow p * i$

$i \leftarrow i + 1$

$\gamma : \text{skip}$

\mapsto

$\alpha : p \leftarrow 1$

$i \leftarrow 2$

if $i < 5$ do

$\beta\langle 0 \rangle : p \leftarrow p * i$

$i_0 : \text{while } i < 5 \text{ do}$

$\beta\langle i_0 + 1 \rangle : p \leftarrow p * i$

$i \leftarrow i + 1$

$\gamma : \text{skip}$

trace: $\alpha \cdots \beta\langle 0 \rangle \cdots \beta\langle 1 \rangle \cdots \beta\langle 2 \rangle \cdots \gamma \cdots$

costs: 42 41 246 246 31

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



Dependent labels: unrolling

```

α : p ← 1
i ← 2
i0 : while i < 5 do
    β⟨i0⟩ : p ← p * i
    i ← i + 1
γ : skip
    
```

 \mapsto

```

α : p ← 1
i ← 2
i0 : while i < 5 do
    β⟨2 * i0⟩ : p ← p * i
    i ← i + 1
    if b then
        β⟨2 * i0 + 1⟩ : p ← p * i
        i ← i + 1
γ : skip
    
```

```

trace: α ····· β⟨0⟩ ····· β⟨1⟩ ····· β⟨2⟩ ····· γ ·····
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:

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

where

- the loop is **single-entry**
(important with presence of `gotos`)
- i_k is different from indexes of containing loops
(in fact, i_k 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

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

```
if  $b$  then
   $S[i_k \mapsto 0]$ 
   $i_k : \text{while } b \text{ do}$ 
     $S[i_k \mapsto i_k + 1]$ 
```

Loop unrolling

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

```
 $i_k : \text{while } b \text{ do}$ 
   $S[i_k \mapsto 2 * i_k]$ 
  if  $b$  then
     $S[i_k \mapsto 2 * i_k + 1]$ 
```

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 (starting from `Cminor`), 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

- For every transformation \mathcal{T} there is an extension of \mathcal{T} to statuses \mathcal{S} :

$$\frac{P, \mathcal{S} \xrightarrow{\lambda} P', \mathcal{S}'}{\mathcal{T}(P), \mathcal{T}(\mathcal{S}') \xrightarrow{\lambda} \mathcal{T}(P'), \mathcal{T}(\mathcal{S}')}$$

Optimisation are particular kinds of transformations

- Apart from optimisations, semantics preservation proofs 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 $\alpha\langle l \rangle$ in compiled code get a cost $\tau(\alpha\langle l \rangle) \in \mathbb{N}$
- 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: $s ::= a * i_k + b$



Simple conditions: $\begin{cases} i_k == a \\ i_k \geq a \\ i_k \% a == b \ \&\& \ i_k \geq b' \end{cases}$

- Proof obligation:

$$\forall \alpha, l \text{ s.t. } \alpha\langle l \rangle \text{ occurs. } \forall C. \kappa(\alpha) \circ (l \circ C) = \kappa(\alpha\langle l \rangle)$$



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, and integrate the results with dependent costs.



Conclusions

Not showed 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)

