Efficient Compilation of .NET Programs for Embedded Systems

Olivier Sallenave  Roland Ducournau

Cortus SA
–
LIRMM
(France)

MASPEGHI/ICOOLPS 2010
Maribor, Slovenia
Outline

1. Motivation
2. Type analysis
3. Coloring
4. Evaluation
Outline

1. Motivation
2. Type analysis
3. Coloring
4. Evaluation
Embedded systems

Long development cycles
- Specific constraints
- Increasing design complexity

Low-end systems
Still programmed in C or assembly

The problem
High-level languages poorly adapted for constrained devices
Approach
Global compilation

Closed-world assumption
Knowledge of the whole program $\Rightarrow$ more efficient implementation

Implications
Disable dynamic loading and reflection
Enable .NET for devices with <100KB memory

Target architecture

Cortus APS3 (32-bit RISC processor)
- Small size
- Low power consumption
Compilation scheme
Ahead-of-time compilation

C# code \rightarrow CIL bytecode \rightarrow C code \rightarrow APS3 machine code

Microsoft C# compiler

Global compilation

- No runtime penalty (no JIT)
- Compile under the **closed-world assumption**
- Focus on optimizing **object mechanisms**

Efficient Compilation of .NET Programs for Embedded Systems

Motivation | Type analysis | Coloring | Evaluation
Outline

1 Motivation
2 Type analysis
3 Coloring
4 Evaluation
Type analysis

- Constructs a **call graph** of the program
- Approximates **dynamic types**
- Many algorithms exist (+/- precision)

Optimizations

- Monomorphic **method calls** → implement as static calls
- Safe **subtype checks** → remove
- Unreachable **members/code** → remove
Implementation details

Our algorithm

- Based on RTA (maintain a set of instantiated classes)
- Meta-model to represent classes/members
- Takes into account .NET specificities

Meta-model

Distinguishes identity/implementations of methods
- Dead identity → remove entry in method tables
- Dead implementation → remove code

Stack state simulation

.NET VM is stack-based ⇒ static type of operands?
- Compute stack state for each instruction
Array covariance (Java/.NET)

**Principle**

\[ B <: A \Rightarrow B[\] <: A[] \]  
\[ \textless: \text{subtyping relationship} \]

```java
class B : A {}
...
void meth(A[] tab, A x, int i)
{
    tab[i] = x;
}
```

**Drawback**

Array assignments are **unsafe** → need subtype check
Eliminating subtype checks

\[ P(x) = \{ \text{possible types of } x \} \]

\text{approximate } x \text{'s type}

\text{The problem for array assignments}

t is unknown at compile-time (element type of the assigned array)
\Rightarrow \text{approximate it as well}
Optimizing array covariance

\[ P(x) = \{ \text{possible types of } x \} \]
\[ E(tab) = \{ \text{possible element types for } tab \} \]  

\[ \text{tab}[i] = x \]
- Safe if \( \forall p \in P(x), \forall e \in E(tab), \ p <: e \)
- Error if \( \forall p \in P(x), \forall e \in E(tab), \neg(p <: e) \)
- Otherwise, need subtype check

Effectiveness

Efficient if a \textbf{few} array types are instantiated
Outline

1 Motivation
2 Type analysis
3 Coloring
4 Evaluation
Single subtyping

Virtual method tables
Straightforward extensions of the superclass table

No binary tree dispatch
- No branch prediction
- Tends to increase code size
Multiple subtyping

- Insert empty entries in method tables
- Maintain single **subtyping invariants**
- Drawback is additional **space cost**
- Minimize the number of holes is NP-hard
Abstract types

- Their method tables **do not exist** at runtime
- Consider it in the coloring algorithm
  → allocate holes in **abstract tables** first
Implementation details

What remains in method tables

Only necessary data
- Methods called *polymorphically* at least once
- Type IDs for *subtype check targets* (Cohen test)
- Some holes

Fill empty entries

Insertion of holes is the drawback of coloring
⇒ use them to store static fields
Outline

1. Motivation
2. Type analysis
3. Coloring
4. Evaluation

Efficient Compilation of .NET Programs for Embedded Systems
### Program reduction

#### Dead types

<table>
<thead>
<tr>
<th></th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>552</td>
</tr>
<tr>
<td>interfaces</td>
<td>24</td>
</tr>
</tbody>
</table>

#### Dead members

<table>
<thead>
<tr>
<th></th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-virtual</td>
<td></td>
</tr>
<tr>
<td>fields</td>
<td>795</td>
</tr>
<tr>
<td>methods</td>
<td>886</td>
</tr>
<tr>
<td>virtual</td>
<td></td>
</tr>
<tr>
<td>methods (id)</td>
<td>384</td>
</tr>
<tr>
<td>methods (impl)</td>
<td>651</td>
</tr>
<tr>
<td>bytecode size</td>
<td>82KB</td>
</tr>
</tbody>
</table>
## Object mechanisms

### Compile-time resolution

<table>
<thead>
<tr>
<th></th>
<th>resolved</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>method calls</strong></td>
<td></td>
</tr>
<tr>
<td>static</td>
<td>2999</td>
</tr>
<tr>
<td>virtual</td>
<td>599</td>
</tr>
<tr>
<td></td>
<td>85%</td>
</tr>
<tr>
<td><strong>subtype checks</strong></td>
<td></td>
</tr>
<tr>
<td>downcasts</td>
<td>84</td>
</tr>
<tr>
<td>array assignments</td>
<td>199</td>
</tr>
<tr>
<td></td>
<td>11%</td>
</tr>
<tr>
<td></td>
<td>81%</td>
</tr>
</tbody>
</table>

### Runtime structures

<table>
<thead>
<tr>
<th></th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>entries in</strong></td>
<td></td>
</tr>
<tr>
<td>method tables</td>
<td>2974</td>
</tr>
<tr>
<td>instance layouts*</td>
<td>707</td>
</tr>
<tr>
<td></td>
<td>82%</td>
</tr>
<tr>
<td></td>
<td>15%</td>
</tr>
</tbody>
</table>

* one layout per class
Summary

- Noticeable **reduction** of the programs
- Array covariance is almost **safe** (almost no overhead)

Bench programs (**.NET Micro Framework**)

Only a **few holes** should result from coloring:
- Most classes do not implement interfaces
- Average method table size is small
- Holes can be filled with static fields
Future work

- Null check elimination
- Genericity (mixed implementation)
- Evaluate on bigger programs
- Profiling
- Garbage collection

Additional features

- Generate debug informations
- Multithreading
Thank you!

http://www.cortus.com