Split Compilation: an application to Just-in-Time Vectorization

Piotr Lesnicki, Albert Cohen, Grigori Fursin

Marco Cornero, Andrea Ornstein, Erven Rohou

Introduction

Context

distributing software as bytecode

- portability
- compact code
- ease compiler development

Problem

code optimization: how to vectorize in a JIT ?

CIL bytecode (.NET) produced by GCC

C

```
int a[10];
foo(){
 a[0] = 2;
}
```

- stack based language
- C arrays → unmanaged arrays

CIL

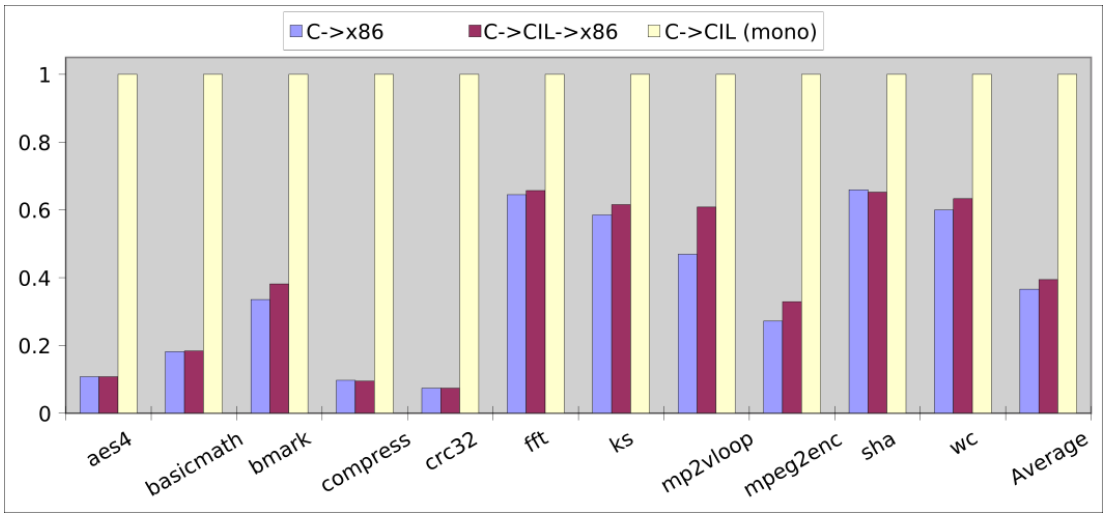
```
.field public  static
    valuetype 'array?[10]I32' 'a'

.class public explicit sealed serializable
    ansi 'array?[10]I32' extends
    ['mscorlib']System.ValueType
{
 .size 40
 .field [0] public specialname
    int32 'elem__'
}


.method public static int32
    'foo' () cil managed
{
 .locals (int32 'cilsimp.0')

 ldsflda valuetype 'array?[10]I32' 'a'
 conv.i
 ldc.i4  2
 stind.i4
 ldloc   'cilsimp.0'
 ret
.maxstack 2
}
```

GCC ?

- CLI branch [STMicroelectronics]:
  - back-end
  - front-end (experimental)

GCC ?

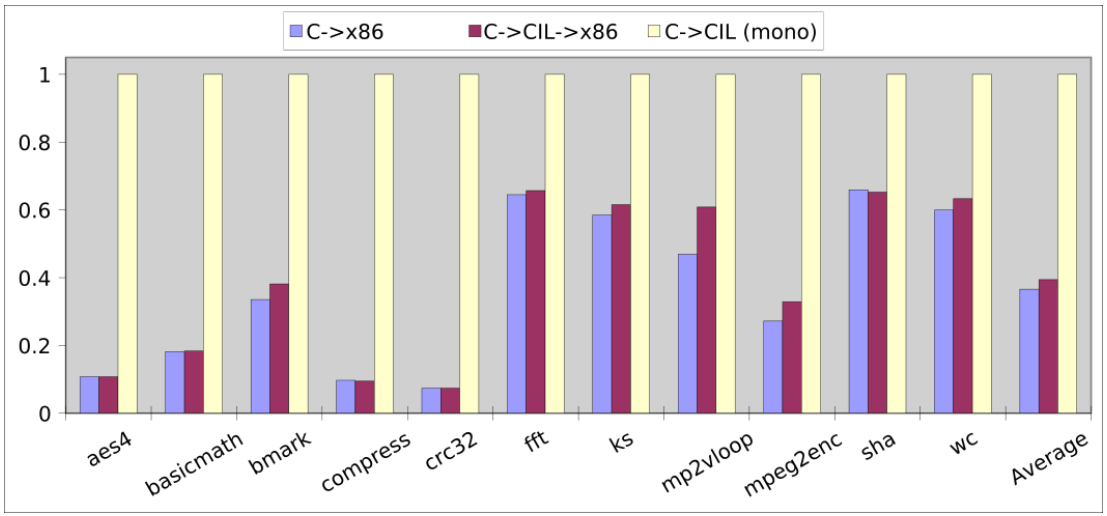
- CLI branch [STMicroelectronics]:
  - back-end
  - front-end (experimental)
- C $\longrightarrow$ CIL $\longrightarrow$ native: good perf. thanks to GCC optimizations



[R.Fernández-Pascual, Hipeac Internship]

GCC ?

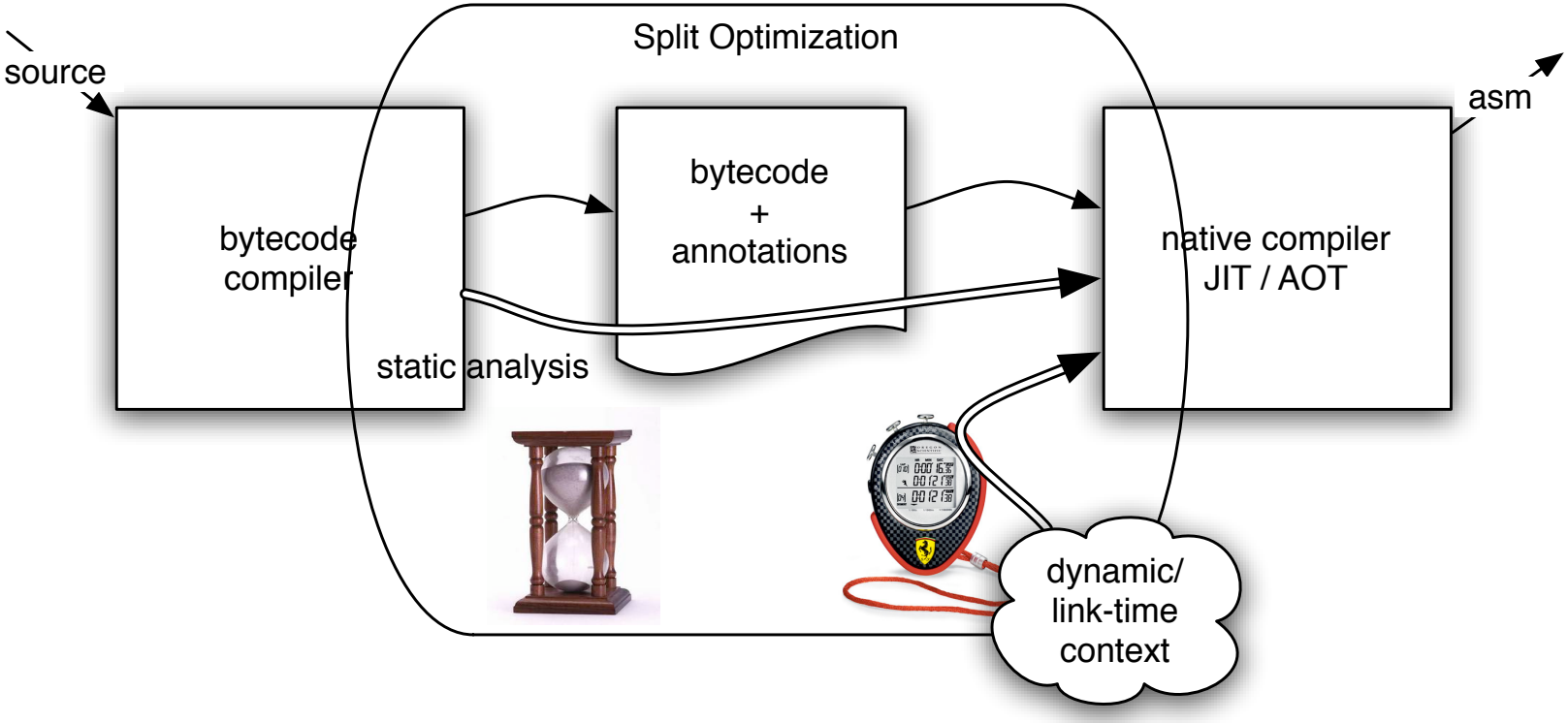
- CLI branch [STMicroelectronics]:
  - back-end
  - front-end (experimental)
- C $\longrightarrow$ CIL $\longrightarrow$ native: good perf. thanks to GCC optimizations



[R.Fernández-Pascual, Hipeac Internship]

- autovectorization

Split compilation



Annotations in CIL

Custom attribute

```
.field private int32 a
.custom instance
   void vector::.ctor() = (01 00 00 00)
```

- part of the standard
- structured (typed, class that extends `System.Attribute`)
- annotate declarations (classes, fields, methods)
- **not code**: difficult to annotate inside code blocks

Inform the JIT that the loop is vectorizable

first naive experiments

```
[vectorizable(loop=1,vector_size=16)] /* method annotation */
foo() {
  int a;
  int x[N],y[N],z[N];
  for(i=0; i<N; i++){
    z[i] = a*x[i] + y[i]
  }
}
```

information:

- loop is vectorizable
- how to strip-mine

issues:

- differentiate memory accesses and control flow
- fragile info. on loops
- hardwired vector size

The vectorizer

[Nuzman & al, *PLDI'06*]

- analysis
  - determines VF
  - dependences, matching patterns (reductions. . .)
  - access patterns (interleaving)
  - alignment (need peeling/versioning)
- transformation
  - loop versioning, peeling
  - instruction replacement
  - loop bounds

Issues for information passing

- fixed vector size in the vectorizer (we would like maximal)
- need of precise information (statements)
- custom attribute propagation across compiler passes

What kind of annotations?

using code versioning and intrinsics

- specific vector operations (interleaving)
- alignment (realign load)
- patterns (reduction, dot product)
- saturated arithmetic (max)

Vector type annotations

Type markers

```
/* type annotation:  array v has a vector access */
[vector ->] int v[N];
/* access marker:    v is accessed as a vector */
... = v.vec[i];
```

- encodes position and access type of vector operations

Vector type consistency

```
int a;
[vector ->] int v[N];
... =  a * v.vec[i];
```

incorrect: 'a' must be explicitely marked as "accessed as a vector" !

Annotations for loop vectorization

Adding information to variable declarations

```
 [vect (loop 1,induction var=i)]
 foo() {
  [vector ->] int a[1]; /* promoted to vector type */
  [vector ->] int x[N];
  [vector ->] int y[N];
   ...
  for(i=0; i<N; i++)
      z.vec[i] = a.vec * x.vec[i] + y.vec[i];
   }
}
```

- instruction to vectorize, where to vectorize
- `x` and `x.vec` are the same memory location, but accessed with another vector type

Vectorizing more complicated patterns ?

reduction

```
foo() {
  [vector ->] int c[1]; /* promoted to vector type */
  [vector ->] int a[N];
   for(i=0; i<N; i++){
    c.vec_reduction += a.vec[i];
  }
}
```

Playing with type-like attributes

strided accesses

```
foo() {
   for(i=0; i<len; i++){
    a[2*i] = b[4*i];
  }
}
```

```
foo() {
   ...
   [vector ->] int a[N];
   [vector ->] int b[N];
   ...
   for(i=0; i<len; i++){
    a.stride2[2*i] = b.stride4[4*i];
  }
}
```

Use for other hardware extensions

- alter bytecode semantics for GPU shaders/kernels
- typed and verifiable code snippets in the bytecode which satisfies some new hardware constrains
- could be separated for
  - source to source transformation
  - special GCC back-ends

cuda like notation

```
[cuda_global] float a[N];
[shared(64)] float *s;
[threadIdx.x]  int tid;
[blockIdx.x]  int bid;

[cuda_kernel(blocks=256, threads=64)]
foo(float *a,int nblocks){
  s[tid] = a[bid*blocks+tid];
  ...
}
```

Conclusion

- work in progress: simple annotation scheme
- opportunities to propagate vectorization information non-intrusively
- extensibility of a general purpose bytecode