Making Meta-Programming Predictable and Enjoyable

or

“opening the compiler box for normal application programmers”

Predictability

What is the current state of the foundations/technology/tools?

- Static type checking of multiple stages
- Error reporting to the “right” stage or abstraction level
- Avoid “surprises” arising with “soft” macro/transformation semantics?
- Avoid black-box Turing-complete meta-programming (simulates black-box compiler construction)?

These are key issues, especially if we are targetting productivity and/or performance benefits beyond compiler construction (or DSL implementation or language extension)

Enjoyability

What is the current state of the foundations/technology/tools?

- Expressiveness vs safety/predictability
- Is introspection or reflection doomed to be type unsafe? Problem with “opening types”? E.g., what about pattern matching like

```
match code_exp with
    .< Add .~x .~y >. -> .< 42 + .~y >.
  | .< fun x -> .~c_e >. -> .< let x = 42 in .~c_e >.
```

- What kind of “intrusion” really matters: syntax? semantics? surprises?

These are key issues, especially if we are targetting productivity and/or performance benefits beyond compiler construction (or DSL implementation or language extension)

A MOVING TARGET

THE *X*-LANGUAGE

A TOOL FOR EXPERT PROGRAMMERS TO DRIVE PROGRAM OPTIMIZATION WHILE MAINTAINING HIGH PRODUCTIVITY AND PORTABILITY

Scalable On-Chip Parallel Computing

Context

Massive parallelism on a chip

- Physically *distributed*, *layered* and *heterogeneous* resources
- Structure and nature of the hardware *exposed* to the software...

... need to be considered for *correctness* and/or *performance*

General-purpose applications need *choice* for scalable performance

- Towards *adaptive* programs (multi-version, continuous optimization)
- SW/HW *negociation*, from load balancing to algorithm selection

Scalable On-Chip Parallel Computing

Context

Massive parallelism on a chip

- Physically *distributed*, *layered* and *heterogeneous* resources
- Structure and nature of the hardware *exposed* to the software...

... need to be considered for *correctness* and/or *performance*

General-purpose applications need *choice* for scalable performance

- Towards *adaptive* programs (multi-version, continuous optimization)
- SW/HW *negociation*, from load balancing to algorithm selection

Impact on

Programming models

Optimizing compilers

Component models

Run-time systems

Goals of the X-Language

1. Compact representation of multiple program versions
   - ⟶ Derive multiple or multi-version programs from a single source
   - ⟶ Generate code at run-time if necessary
2. Explicit multiple optimization strategies
   - ⟶ Rely on predefined transformation primitives
   - ⟶ Declare high-level optimization goals rather than explicit transformations
3. Implement and apply custom optimizations
   - ⟶ Custom transformations can be implemented by expert programmers
   - ⟶ Derive decision trees automatically from abstract descriptions
4. Bring together individual transformations and actual performance measurements
   - ⟶ Implement local/layered learning/search strategies
   - ⟶ Couple with hardware counters, sampling mechanisms and phase detection

Key Design Ideas

Build on top of *multistage programming*

- Manipuate code expressions

  ```
  code c = ‘{ bar(42); ‘}
  ```

- Splice code into code

  ```
  ‘{ foo(‘%(c)); ‘}        // foo(bar(42));
  ```

- Generate and run code

  ```
  run(c);
  ```

- Cross-stage persistence

  ```
  int x = 42; code c = ‘{ foo(bar(x)); ‘}
  ```

Key Design Ideas

Build on top of *multistage programming*

- Manipuate code expressions
  `code c = ‘{ bar(42); ‘}`
- Splice code into code
  `‘{ foo(‘%(c)); ‘}      // foo(bar(42));`
- Generate and run code
  `run(c);`
- Cross-stage persistence
  `int x = 42; code c = ‘{ foo(bar(x)); ‘}`

Provide some form of *reflection* that *does not alter* observable semantics

- (Assuming transformation legality)
- Use code annotations: `#pragma xlang`

```
#pragma xlang transformation [ scope_name ] node_name_regexp
                             [ parameters ] [ additional_names ]
```

  Example: `#pragma xlang unroll loop1 4`
- Some kind of well-behaved, restricted AOP?

Features of the Language

Transformations primitives

- Loop transformations

  → unrolling, strip-mining, distribution, fusion, coalescing, interchange, skewing, reindexing, hoisting, shifting, scalar promotion, privatization

- Interprocedural transformations

  → inlining, cloning, partial evaluation, slicing

Features of the Language

Transformations primitives

- Loop transformations

  → unrolling, strip-mining, distribution, fusion, coalescing, interchange, skewing, reindexing, hoisting, shifting, scalar promotion, privatization

- Interprocedural transformations

  → inlining, cloning, partial evaluation, slicing

Compound transformations

- Composition of code generators (multi-stage evaluation with splicing)
- Sequence of annotation pragmas
- Procedural abstraction (build custom transformations from primitives)
- Control the application and parameters of each transformation

Features of the Language

Transformations primitives

- Loop transformations

  → unrolling, strip-mining, distribution, fusion, coalescing, interchange, skewing, reindexing, hoisting, shifting, scalar promotion, privatization

- Interprocedural transformations

  → inlining, cloning, partial evaluation, slicing

Compound transformations

- Composition of code generators (multi-stage evaluation with splicing)
- Sequence of annotation pragmas
- Procedural abstraction (build custom transformations from primitives)
- Control the application and parameters of each transformation

Static analyses (crude scalar data-flow information right now)

Features of the Language

Transformations primitives

- Loop transformations

  → unrolling, strip-mining, distribution, fusion, coalescing, interchange, skewing, reindexing, hoisting, shifting, scalar promotion, privatization

- Interprocedural transformations

  → inlining, cloning, partial evaluation, slicing

Compound transformations

- Composition of code generators (multi-stage evaluation with splicing)
- Sequence of annotation pragmas
- Procedural abstraction (build custom transformations from primitives)
- Control the application and parameters of each transformation

Static analyses (crude scalar data-flow information right now)

Dynamic analyses (only time measurement right now)

Example: Transformation Sequences

Each transformation regererates annotations for the next transformation

```
#pragma xlang name loop1
for (i=m; i<n; i++)
  a[i] = b[i];

#pragma xlang stripmine loop1 4 loop1_2 loop1_3
#pragma xlang unroll loop1_2
```

→

```
#pragma xlang name loop1
for (ii=m; ii+4<n; ii+=4) {
  #pragma xlang name loop1_2
  for (i=ii; i<ii+4; i++)
    a[i] = b[i];
  #pragma xlang name loop1_3
}
for (i=ii; i<n i++)
  a[i] = b[i];

#pragma xlang unroll loop1_2
```

Example: Transformation Sequences

Each transformation regererates annotations for the next transformation

```
#pragma xlang name loop1
for (ii=m; ii+4<n; ii+=4) {
  #pragma xlang name loop1_2
  for (i=ii; i<ii+4; i++)
    a[i] = b[i];
  #pragma xlang name loop1_3
}
for (i=ii; i<n i++)
  a[i] = b[i];

#pragma xlang unroll loop1_2
```

→

```
#pragma xlang name loop1
for (ii=m; ii+4<n; ii+=4) {
  #pragma xlang name loop1_2
  i = ii;
  a[i] = b[i];
  i = ii+1;
  a[i] = b[i];
  i = ii+2;
  a[i] = b[i];
  i = ii+3;
  a[i] = b[i];
}
#pragma xlang name loop1_3
for (i=ii; i<n i++)
  a[i] = b[i];
```

Example: Evaluating Multiple Versions

```
for (u=1; u<8; u++) {
  code c = ‘{
    #pragma xlang name loop1
    for (i=m; i<n; i++)
      a[i] = b[i];
    #pragma xlang stripmine loop1 u loop1_2 loop1_3
  ‘}
  run(c, &elapsed_time);
  // drive search/learning strategy from this evaluation
}
```

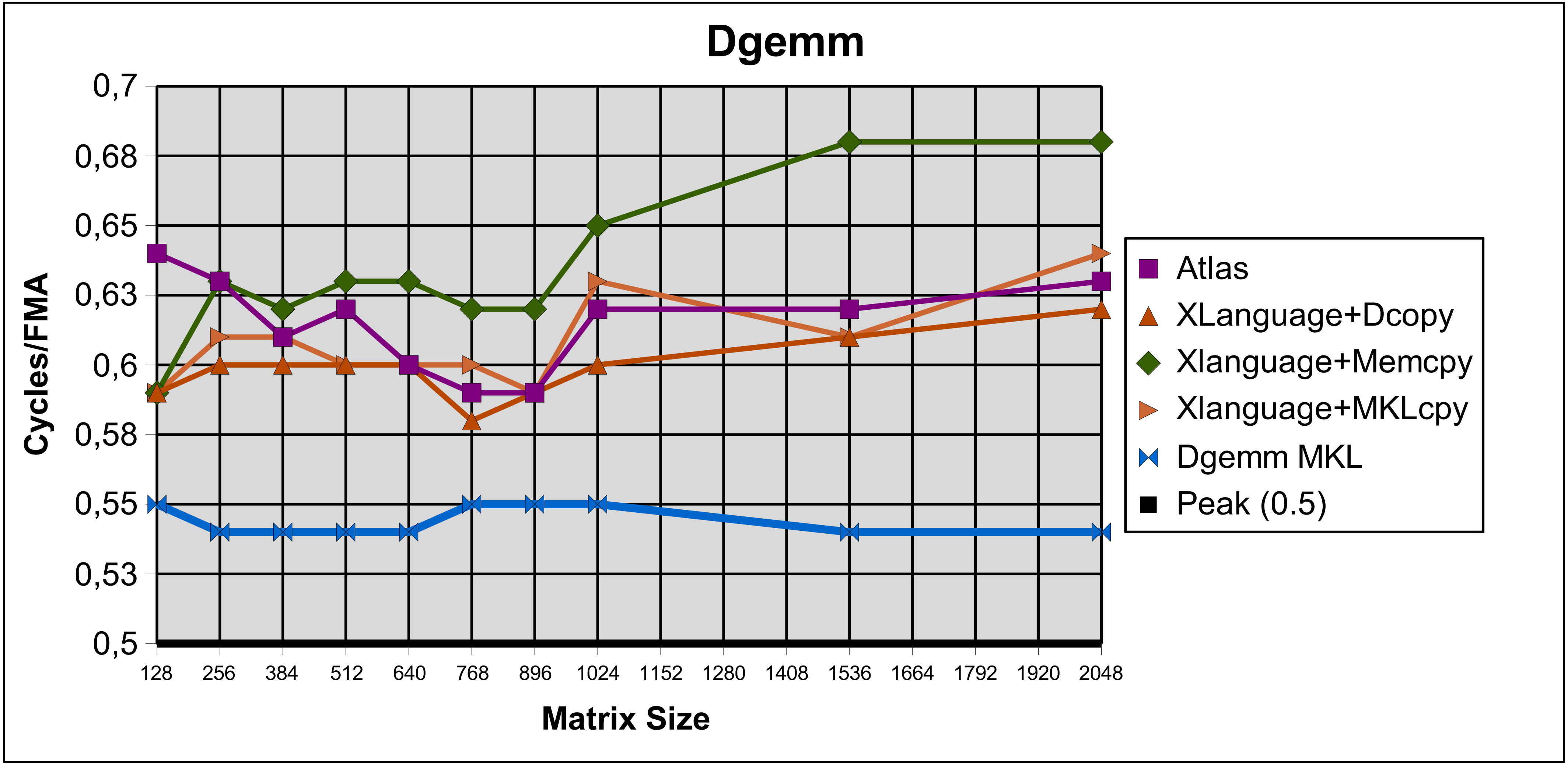
Full Example: Matrix Product in ATLAS

```
#pragma xlang name iloop
for (i=0; i<NB; i++)
  #pragma xlang name jloop
  for (j=0; j<NB; j++)
    #pragma xlang name kloop
    for (k=0; k<NB; k++) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
// Simplified transformation sequence for IA64
// (excluding search engine, pipelining, prefetch and page copying)
#pragma xlang stripmine iloop NU NUloop
#pragma xlang stripmine jloop MU MUloop
#pragma xlang interchange kloop MUloop
#pragma xlang interchange jloop NUloop
#pragma xlang interchange kloop NUloop
#pragma xlang fullunroll NUloop
#pragma xlang fullunroll MUloop
#pragma xlang scalarize_in b in kloop
#pragma xlang scalarize_in a in kloop
#pragma xlang scalarize_in&out c in kloop
#pragma xlang hoist kloop.loads before kloop
#pragma xlang hoist kloop.stores after kloop
```

Full Example: Matrix Product in ATLAS

```
#pragma xlang name iloop
for (i=0; i<NB; i++) {
 #pragma xlang name jloop
 for (j=0; j<NB; j+=4) {
  #pragma xlang name kloop.loads
  { c_0_0 = c[i+0][j+0]; c_0_1 = c[i+0][j+1];
    c_0_2 = c[i+0][j+2]; c_0_3 = c[i+0][j+3]; }
  #pragma xlang name kloop
  for (k=0; k<NB; k++) {
    { a_0 = a[i+0][k]; a_1 = a[i+0][k];
      a_2 = a[i+0][k]; a_3 = a[i+0][k]; }
    { b_0 = b[k][j+0]; b_1 = b[k][j+1];
      b_2 = b[k][j+2]; b_3 = b[k][j+3]; }
    { c_0_0=c_0_0+a_0*b_0; c_0_1=c_0_1+a_1*b_1;
      c_0_2=c_0_2+a_2*b_2; c_0_3=c_0_3+a_3*b_3; }
    // ...
  }
  #pragma xlang name kloop.stores
  { c[i+0][j+0] = c_0_0; c[i+0][j+1] = c_0_1;
    c[i+0][j+2] = c_0_2; c[i+0][j+3] = c_0_3; }
}}
// Remainder code
```

Preliminary Results on IA64



Main Limitations

1. Hard to understand and keep track of transformations effects
   - ⟶ *Build and manage long sequences of transformations*
   - ⟶ Convince the expert programmer that it saves him time

2. Define custom transformations, beyond combination of existing primitive ones
   - ⟶ *General kind of program construction*
   - ⟶ Algorithm selection

Conclusion: Future Optimizing Compilers

Research Directions

Compilers must do *tedious* things in a *predictable* manner...

... *but should not try to be smart*

→ Fully automatic framework for abstraction-penalty removal

→ Machine learning and rule-based system for architecture-aware optimizations

→ Let application experts tell what is important

Tightly coupled off-line and on-line optimization

→ Aggressive off-line analysis and narrowing of the optimization search-space

→ Low-overhead just-in-time/run-time transformations and code generation

Complement intermediate representations with program generators

→ Expose algebraic properties of the search space

→ Support global and complex transformation sequences

Progresses

SPIRAL

Tools for safe and efficient metaprogramming

Machine learning compilers