

# *GCC ICI* *(Interactive Compilation Interface)*

*Grigori Fursin*

ALCHEMY Group  
INRIA Futurs  
France

*January, 2007*

*Funded by HiPEAC network*

# Outline

- **Introduction and Motivation**
- **Iterative Interactive Compiler Framework**
- **Interactive Compilation Interface (ICI)**
- **Tools and Experiments**
- **Conclusions and Future Work**

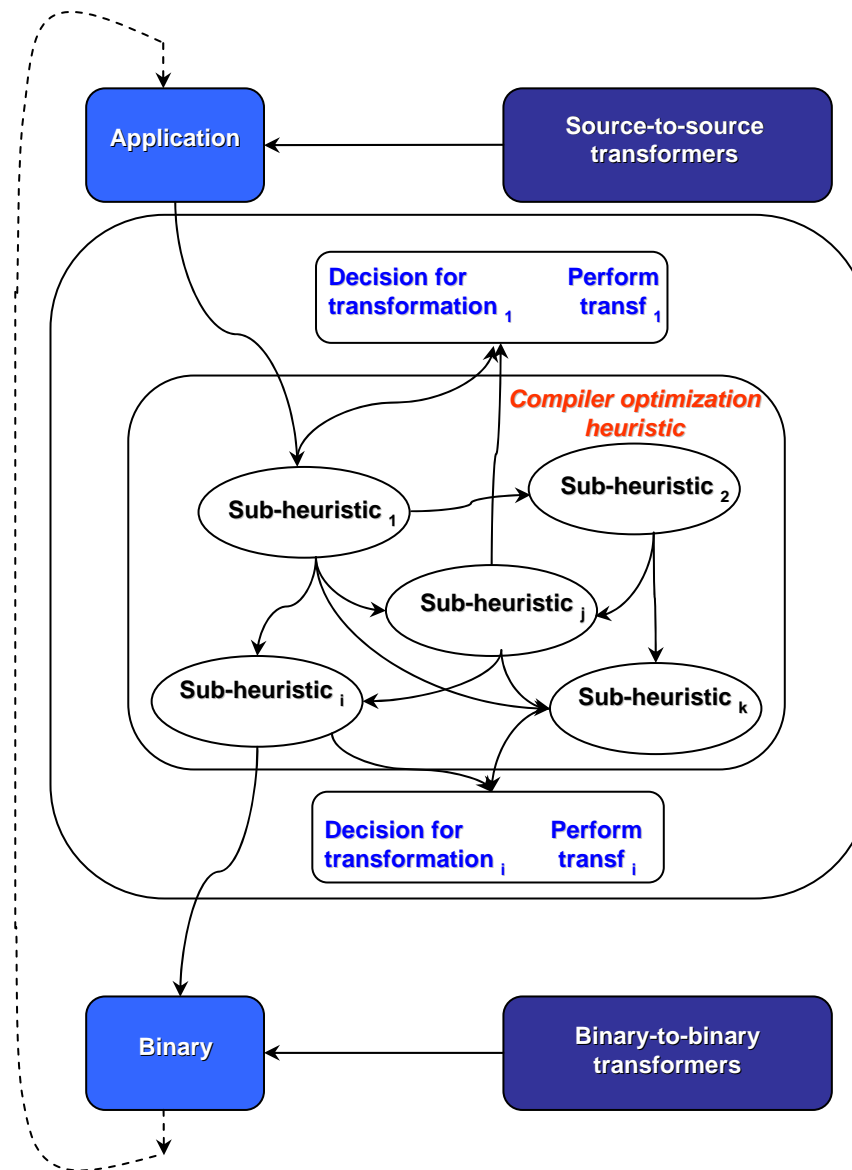
# Motivation

- Current compilers fail to deliver best performance on modern processors due to
  - rapidly evolving hardware
  - simplistic hardware models
  - fixed black-box optimization heuristics
  - inability to fine-tune applications
  - lack of run-time information
- Different research compilers or transformation tools
  - rewritten from scratch to “clean” internals and understand behavior (time consuming)
  - have many unnecessary duplications of other compiler internals
  - are often incompatible with each other and non-portable
  - usually support limited number of languages
  - still often have ambiguous and non-portable optimization heuristics

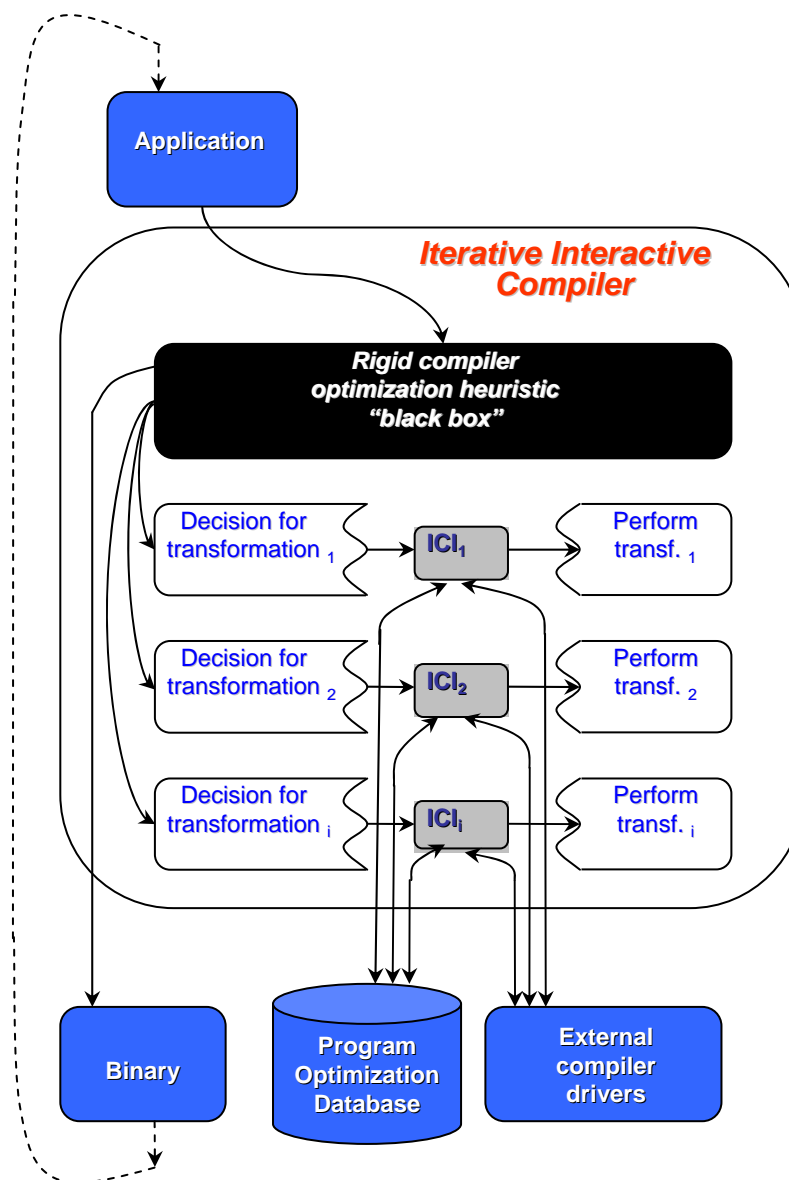
# Goals

- Instead of developing new compiler or transformations tools, modify current popular (non-research) rigid compilers into simpler transparent open transformation toolsets with externally tunable optimization heuristics through a standardized Interactive Compilation Interface (ICI)
- Control only decision process at global and local levels and avoid revealing all intermediate compiler representation to allow further transparent compiler evolution
- Narrow down optimization space by suggesting only legal transformations
- Enable iterative recompilation algorithm to apply sequences of transformations
- Treat current optimization heuristic as a black-box and progressively adapt it to a given program and given architecture
- Allow life-long, whole-program optimization research with optimization knowledge reuse

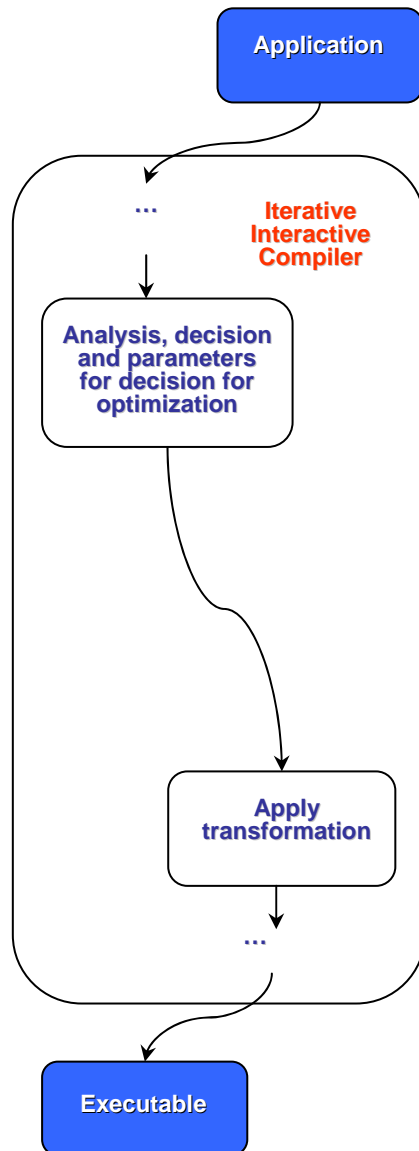
# Current Compilers



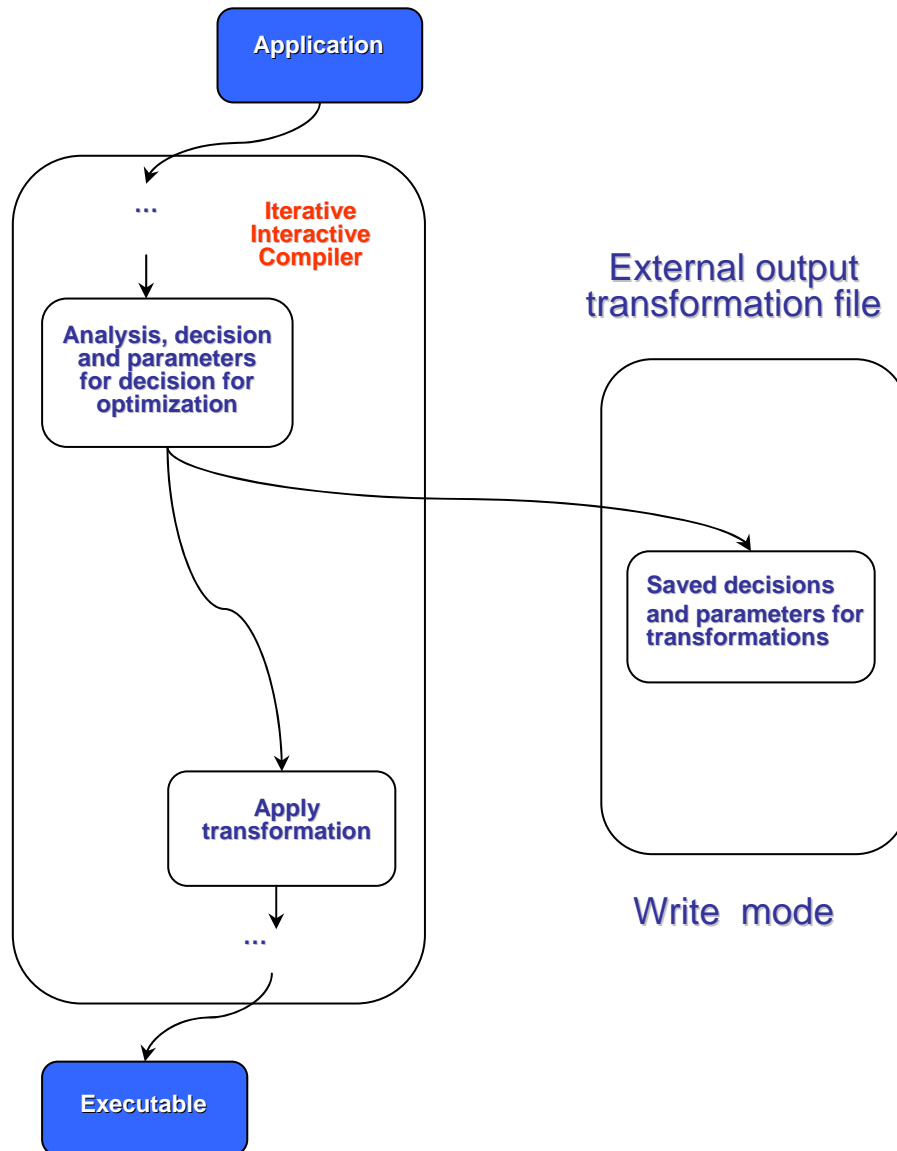
# Iterative Interactive Compiler Framework



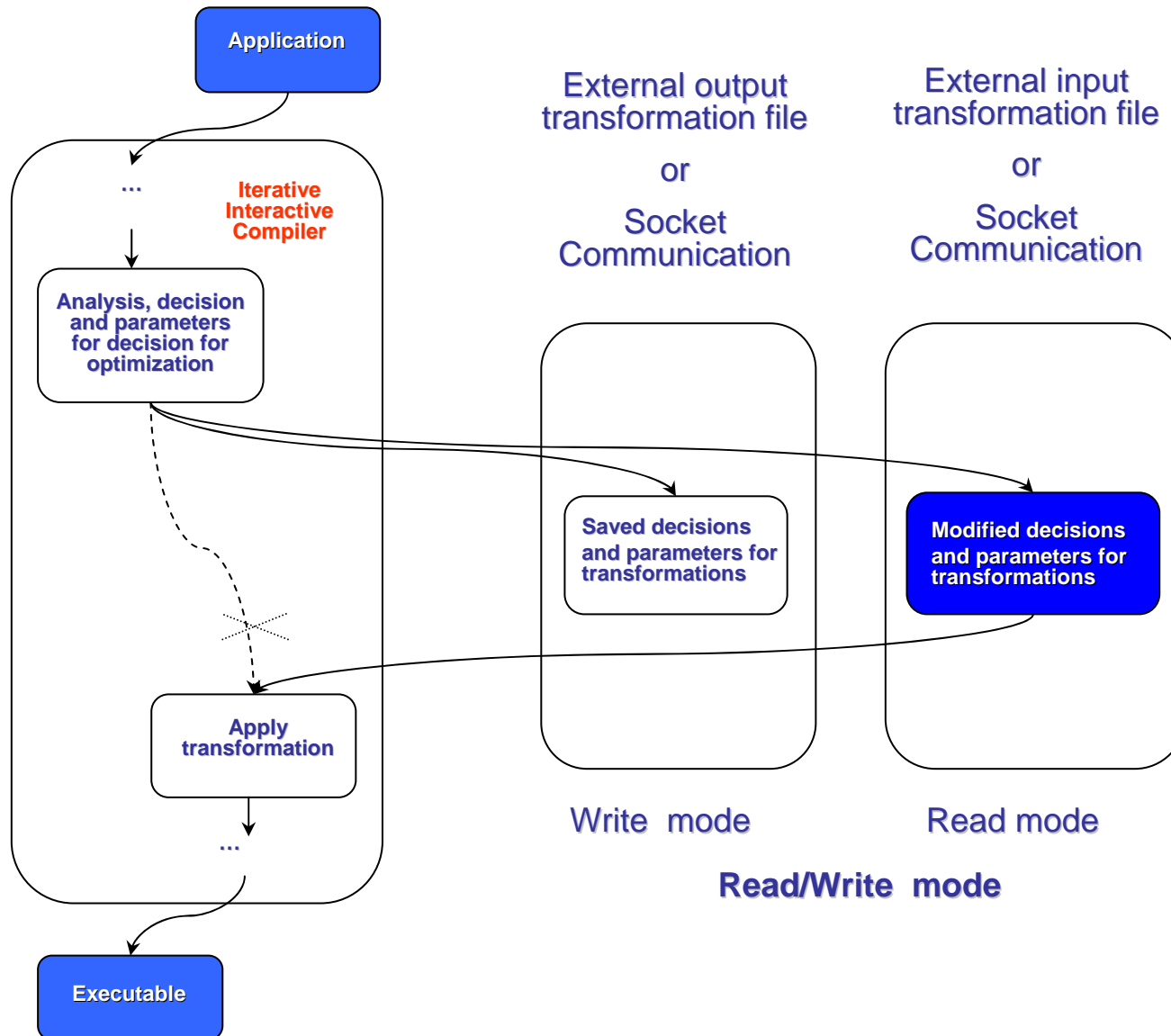
# Interactive Compilation Interface



# Interactive Compilation Interface



# Interactive Compilation Interface



# Interactive Compilation Interface

## Invoking ICI

- Through command line:

### Write mode:

```
gcc -fici-generate -ftree-loop-linear -funroll-loops *.c
```

### Read/Write mode:

```
gcc -fici-generate -fici-use -ftree-loop-linear -funroll-loops *.c
```

- Through environment variables  
(to enable transparent continuous optimizations):

### Write mode:

```
export GCC_ICI_GEN = 1  
make
```

### Read/Write mode:

```
export GCC_ICI_GEN = 1  
export GCC_ICI_USE = 1  
export GCC_ICI_OPTS = -ftree-loop-linear -funroll-loops  
make
```

# Current Implementation

## External output transformation xml file:

```
<?xml version="1.0"?>
<compiler_ici>
  <file_name="swim.f">
    <transformation name="unroll_and_peel">
      <function>calc1</function>
      <loop_number>4</loop_number>
      <depth>1</depth>
      <decision>4</decision>
      <factor>7</factor>
    </transformation>
    <transformation name="unroll_and_peel">
      <function>calc1</function>
      <loop_number>3</loop_number>
      <depth>1</depth>
      <decision>4</decision>
      <factor>7</factor>
    </transformation>
    ...
  </file_name>
</compiler_ici>
```

# Current Implementation

## External output transformation xml file:

```
<?xml version="1.0"?>
<compiler_ici>
  <file_name="swim.f">
    <transformation name="unroll_and_peel">
      <function>calc1</function>
      <loop_number>4</loop_number>
      <depth>1</depth>
      <decision>4</decision>
      <factor>7</factor>
    </transformation>
    <transformation name="unroll_and_peel">
      <function>calc1</function>
      <loop_number>3</loop_number>
      <depth>1</depth>
      <decision>4</decision>
      <factor>7</factor>
    </transformation>
    ...
  </file_name>
</compiler_ici>
```

## Supported optimizations:

### **global:**

- *program phase reordering*

### **local:**

- *loop interchange*
- *loop peeling*
- *loop unrolling*

***more optimizations soon ...***

# Current Implementation

## External output transformation xml file:

```
<?xml version="1.0"?>
<compiler_ici>
  <file_name="swim.f">
    <transformation name="unroll_and_peel">
      <function>calc1</function>
      <loop_number>4</loop_number>
      <depth>1</depth>
      <decision>4</decision>
      <factor>7</factor>
    </transformation>
    <transformation name="unroll_and_peel">
      <function>calc1</function>
      <loop_number>3</loop_number>
      <depth>1</depth>
      <decision>4</decision>
      <factor>7</factor>
    </transformation>
    ...
  </file_name>
</compiler_ici>
```

## Supported optimizations:

### **global:**

- *program phase reordering*

### **local:**

- *loop interchange*
- *loop peeling*
- *loop unrolling*

***more optimizations soon ...***

## Based on PathScale ICI (2004-2006)

- *inlining*
- *array padding (global/local)*
- *loop fusion/fission*
- *loop interchange*
- *loop blocking*
- *loop unrolling*
- *register tiling*
- *prefetching*

# Iterative Recompilation Algorithm

## Iterative Recompilation Algorithm to apply sequences of transformations:

clear *transformation\_file\_out.xml*

set PATHSCALE\_ICI\_W to 1

**compile program**

(write *transformation\_file\_out.xml*)

set PATHSCALE\_ICI\_R to 1

\_label\_recompile:

copy *transformation\_file\_out.xml* to

*transformation\_file\_in.xml*

modify *transformation\_file\_in.xml* if needed

**compile program**

(read *transformation\_file\_in.xml*,

write *transformation\_file\_out.xml*)

if *transformation\_file\_in.xml* not the same

as *transformation\_file\_out.xml*

go to \_label\_recompile

# GCC Instrumentation (Transformations)

`gcc/loop-unroll.c`

```
#include "fci.h"
```

```
static void
```

```
decide_unrolling_and_peeling (struct loops *loops, int flags) {
```

```
    ...
```

```
    decide_unroll_constant_iterations (loop, flags);
```

```
    if (loop->lpt_decision.decision == LPT_NONE)
```

```
        decide_unroll_runtime_iterations (loop, flags);
```

```
    if (loop->lpt_decision.decision == LPT_NONE)
```

```
        decide_unroll_stupid (loop, flags);
```

```
    if (loop->lpt_decision.decision == LPT_NONE)
```

```
        decide_peel_simple (loop, flags);
```

```
    /* GCC ICI */
```

```
    if (flag_ici_use)
```

```
        fci_unroll_in(get_name(current_function_decl), loop->num, loop->depth,  
                    &(loop->lpt_decision.decision), &(loop->lpt_decision.times));
```

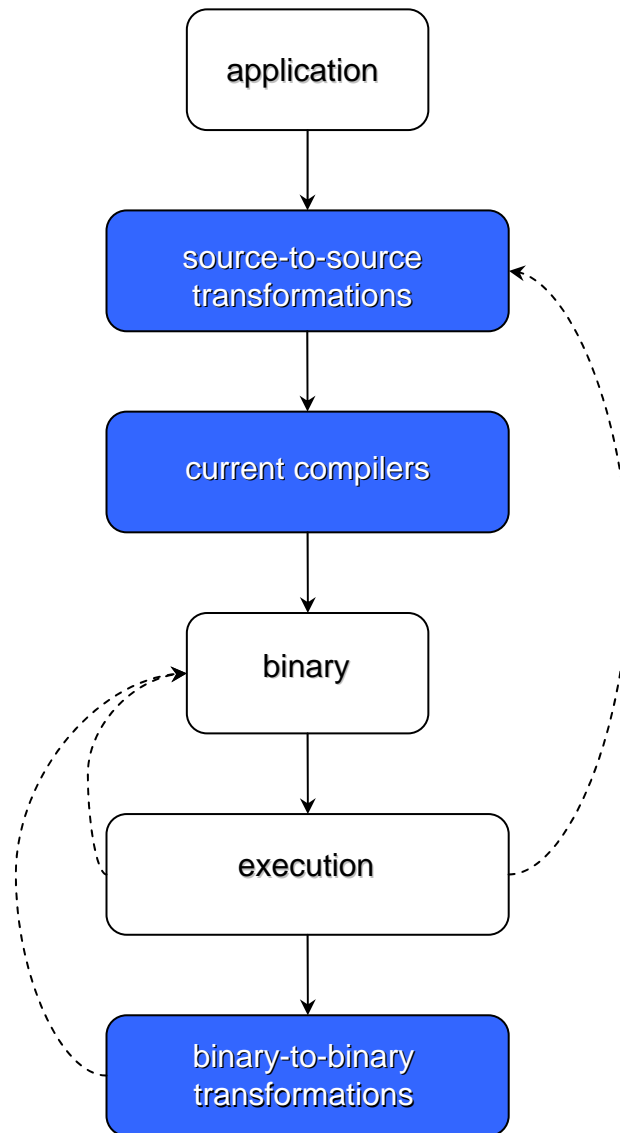
```
    if (flag_ici_generate)
```

```
        fci_unroll_out(get_name(current_function_decl), loop->num, loop->depth,  
                    &(loop->lpt_decision.decision), &(loop->lpt_decision.times));
```

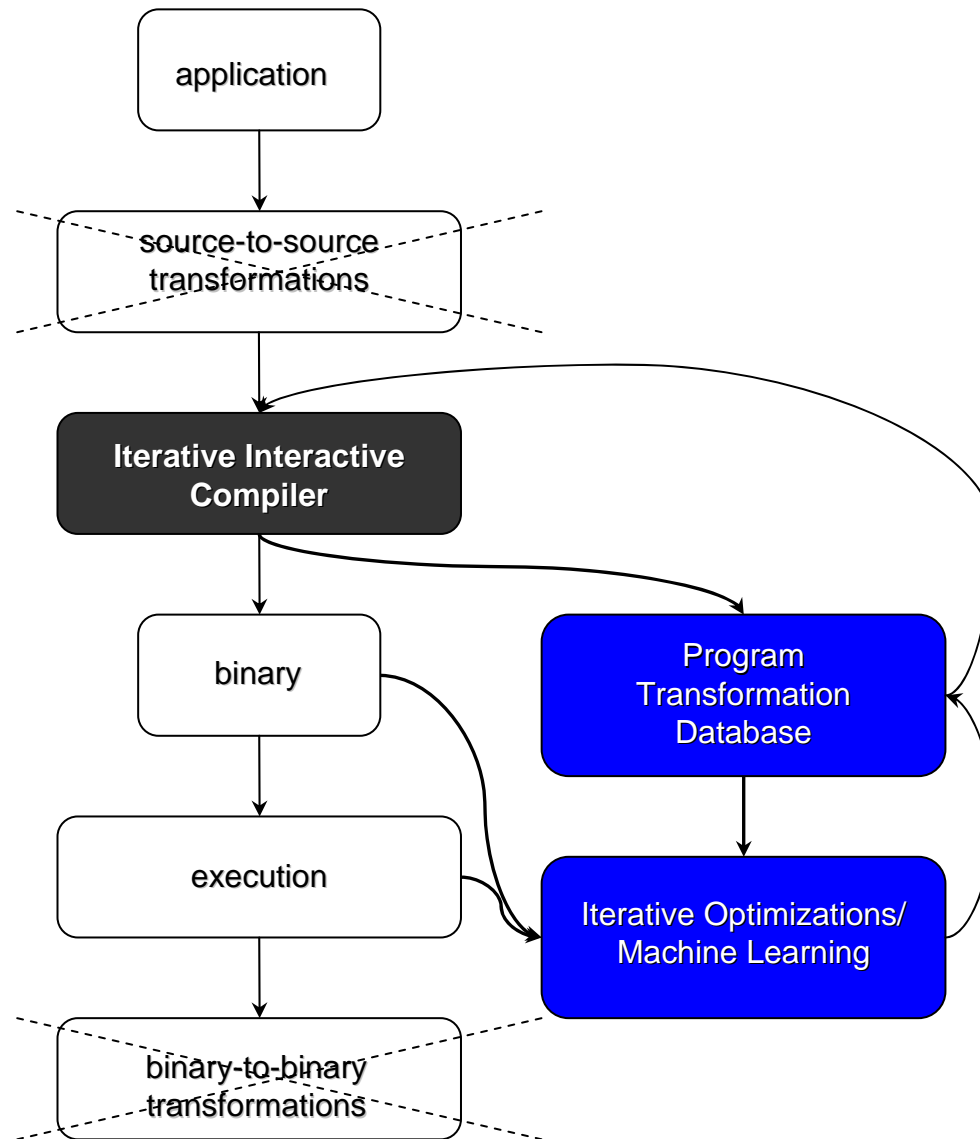
```
    loop = next;
```

```
}
```

# Iterative Continuous Optimizations



# Iterative Continuous Adaptive Optimizations



# Using Framework

## Porting from PathScale Continuous Optimization Framework (2003-cur) to GCC or developing:

- Continuous iterative optimization driver with run-time adaptation at function, loop-level or instruction level using low-overhead phase detection technique
- Driver to continuously collect all possible optimization parameters
- Driver to automatically and continuously rebuild compiler optimization heuristic, and adapt to a specific architecture using statistical methods and collective optimization knowledge reuse among different programs and architectures
- Prototype framework to replace a model-based compiler heuristic with automatically learned one by connecting ICI with WEKA - an open-source machine learning software package

# Preliminary Results

	Matmul Small dataset (3x3 matrices, 27400000 multiplications)	
Optimization	Execution time	Speedup
-O3	<b>5.0 s.</b>	
-O3 –funroll-all-loops (internal factor 7)	<b>3.9 s.</b>	<b>1.3</b>
-O3 –funroll-all-loops (factor 2 selected with GCC ICI)	<b>3.1 s.</b>	<b>1.6</b>

# Conclusions

- We demonstrate a simple, practical and non-intrusive way to turn current rigid compilers into powerful interactive transformation toolset with an Interactive Compilation Interface that allows to bias compiler optimization decisions externally
- We avoid the pitfalls of rigidifying the compiler internals while granting access to rich-enough features to take performance-critical decisions
- We considerably reduce optimization search space by analyzing and applying only legal transformations
- We develop tools for continuous collective life-long optimizations and knowledge reuse across different programs and architectures
- We use framework in EU projects to automatically adapt and optimize programs for performance, code size, power consumption, multiple ISA, etc

# Future work

- Porting ICI to GCC in collaboration with IBM, NXP (Philips), STMicro, ARC, multiple universities within HiPEAC network of excellence and within EU-funded projects MilePost, SARC and GGCC
- Adding more transformations and enabling phase-reordering at function level in GCC
- Unifying optimization naming conventions to enable portability and knowledge reuse to build optimization heuristics automatically
- Implementing run-time adaptation technique to select different program versions at run-time depending on program behavior
- Finishing framework for practical continuous life-long whole-program optimizations with statistical or machine learning techniques
- Porting ICI to JIT compilers (Jikes, .NET) to unify run-time optimizations

**Would like to participate?**

<http://sourceforge.net/projects/gcc-ici>

# Questions?

Software development web-site for GCC ICI:

*<http://sourceforge.net/projects/gcc-ici>*

Thanks to *Sebastian Pop, Cupertino Miranda* and *Hamid Daoud*  
for help with gcc modifications

Collaborations and Support:

*IBM, NXP (Philips), STMicro, ARC, CAPS, Universities within HiPEAC*

This work is funded by HiPEAC

*<http://www.hipeac.net>*

Contact e-mail: *[grigori.fursin@inria.fr](mailto:grigori.fursin@inria.fr)*

More information: *[http://fursin.net/research\\_desc.html](http://fursin.net/research_desc.html)*