Continuous run-time adaptation and optimization for statically compiled programs

Grigori Fursin

ALCHEMY Group
INRIA Futurs
France

January, 2007

Partly funded by HiPEAC network

Outline

- Introduction and motivation
- Static code versioning
- Low-overhead adaptation techniques
- Continuous optimizations and adaptation
- Conclusions and future work

Introduction and motivation

Why adapt?

- Different program context
- Different run-time behavior
- Different system load
- Different available resources
- Different architectures & ISA

Introduction and motivation

Why adapt?

- Different program context
- Different run-time behavior
- Different system load
- Different available resources
- Different architectures & ISA

For each case we want to find and use best optimization settings!

Run-time program behavior

At which granularity?

Run-time program behavior

At which granularity?

Repeatedly executed time-consuming parts of the code that allow powerful transformations:

typically functions or loops

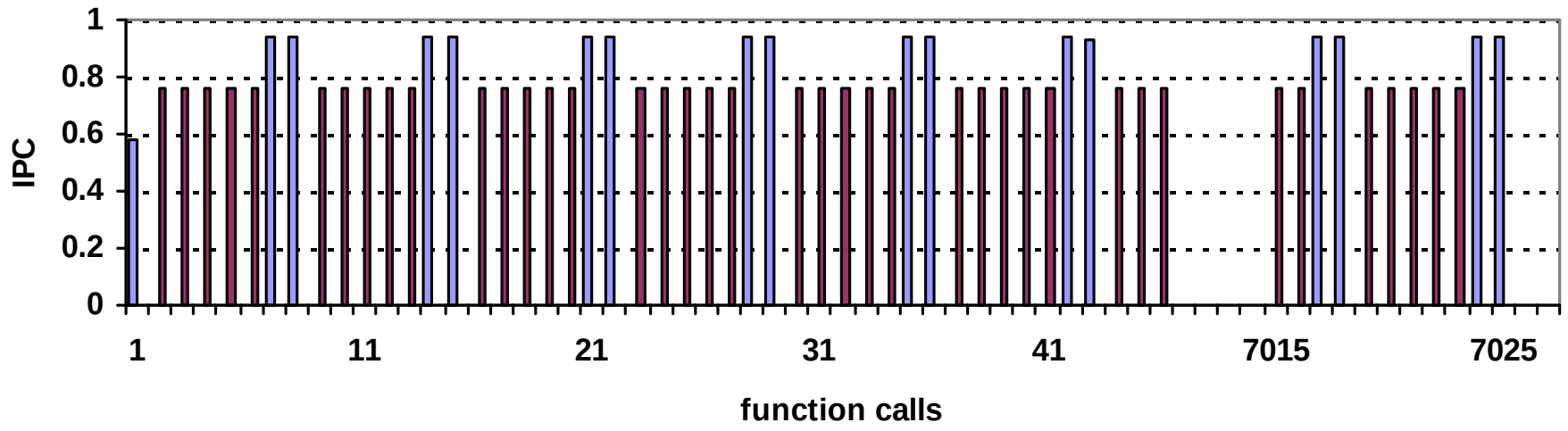
Run-time program behavior

At which granularity?

Repeatedly executed time-consuming parts of the code that allow powerful transformations:

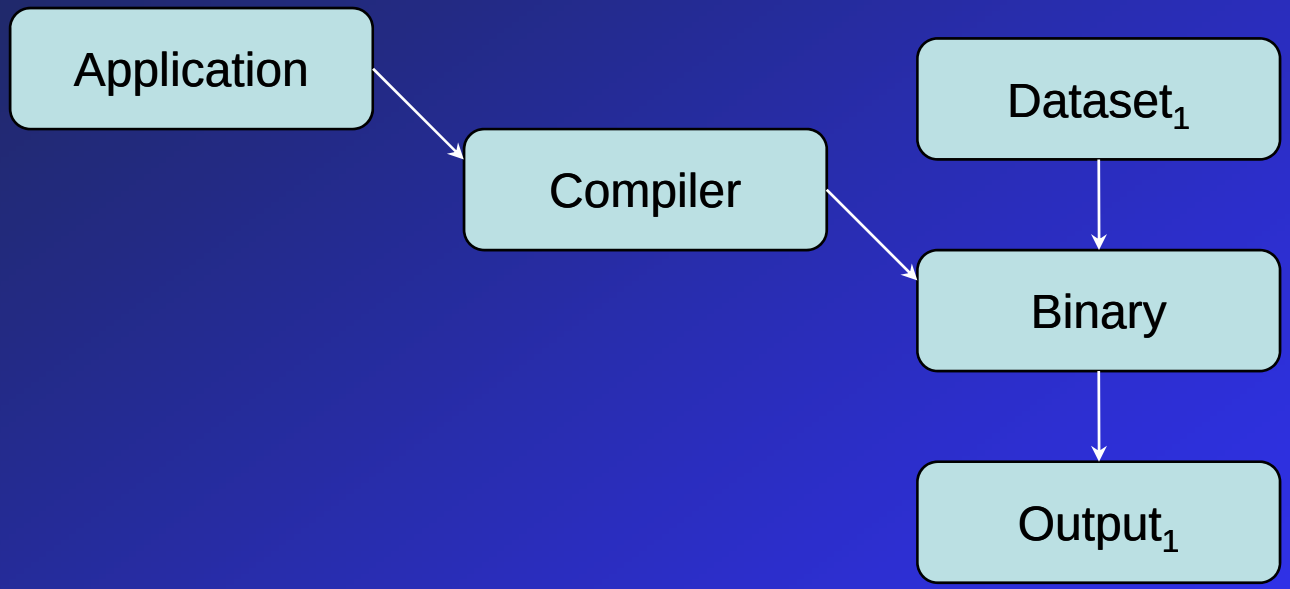
typically functions or loops

IPC for subroutine resid of benchmark mgrid across calls



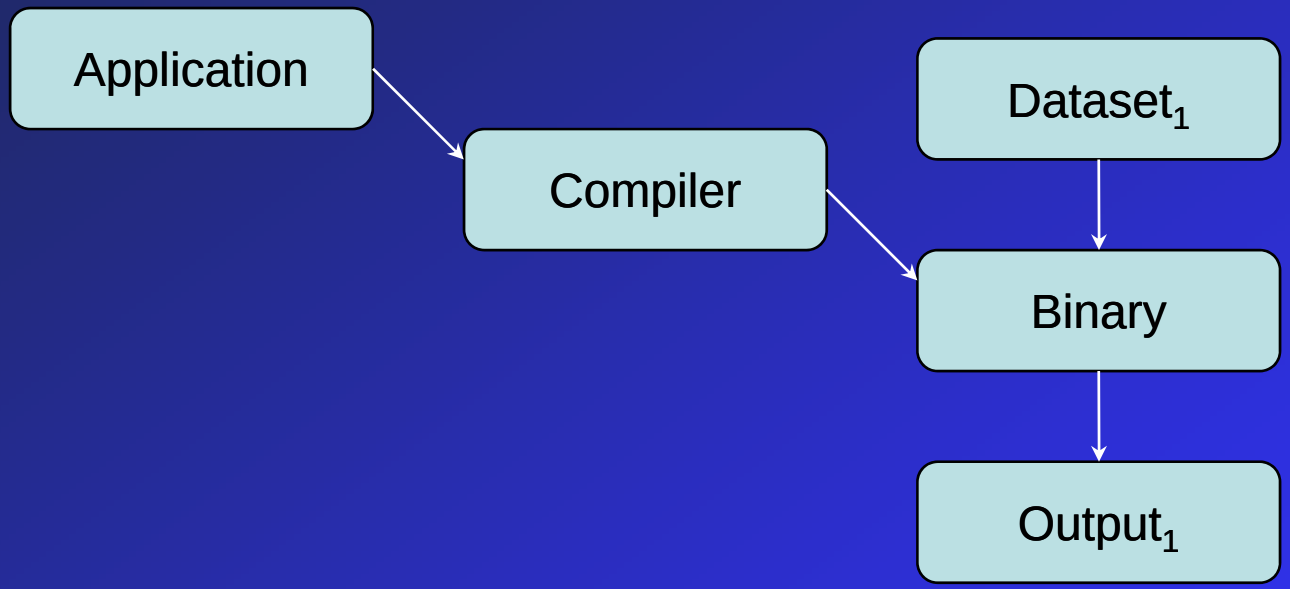
Current methods

Problem with traditional optimization approaches:



Current methods

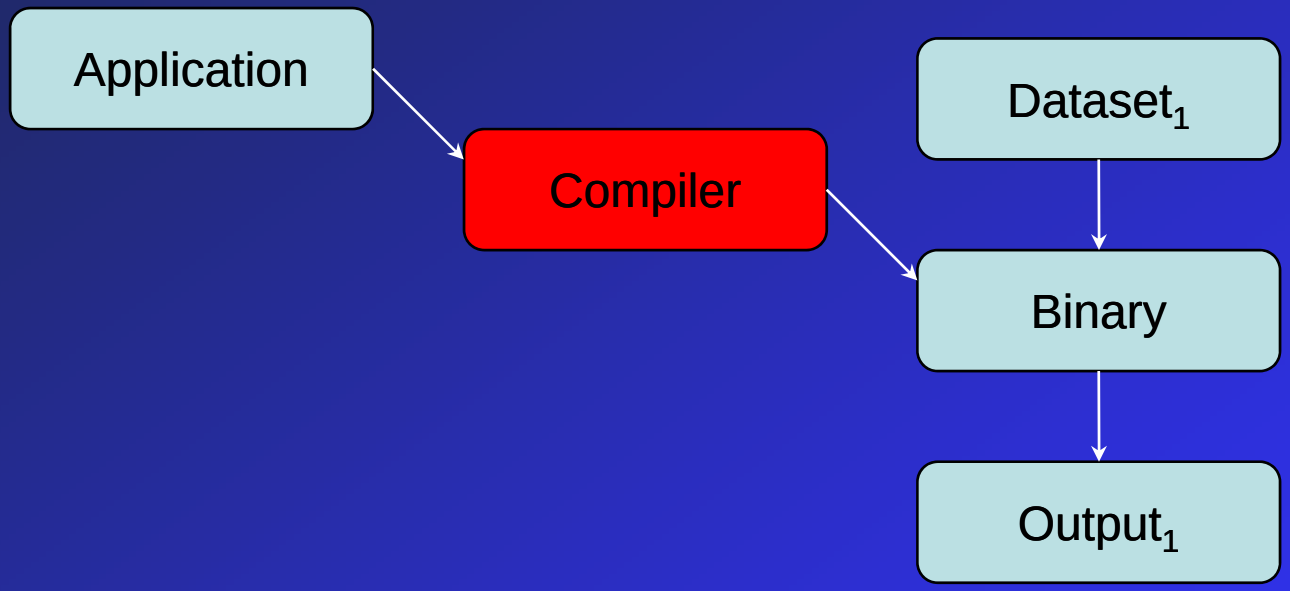
Problem with traditional optimization approaches:



Static optimizations:

Current methods

Problem with traditional optimization approaches:

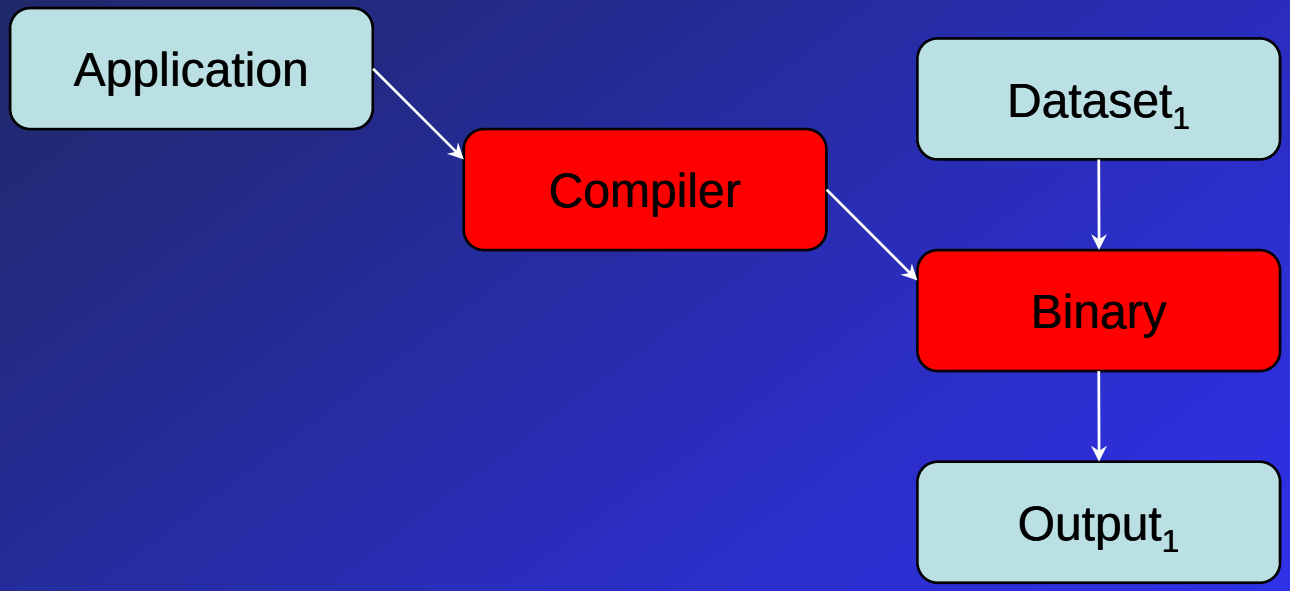


Static optimizations:

- *rigid optimization heuristics with simplified hardware models,*

Current methods

Problem with traditional optimization approaches:

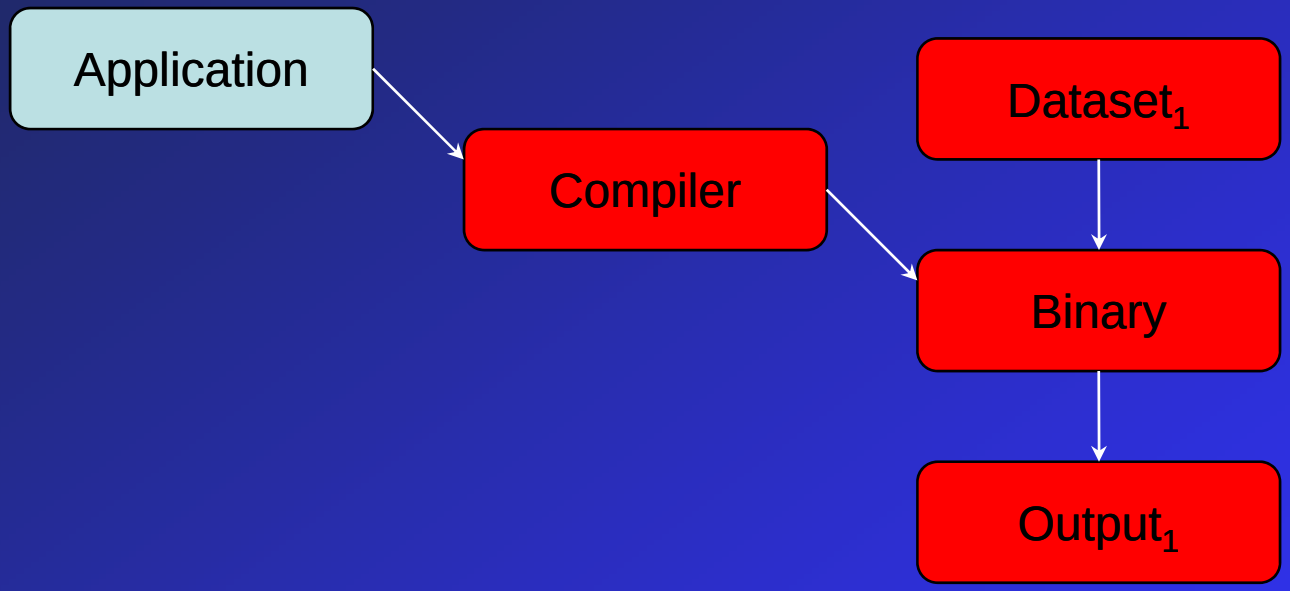


Static optimizations:

- *rigid optimization heuristics with simplified hardware models,*
- *lack of run-time information*

Current methods

Problem with traditional optimization approaches:

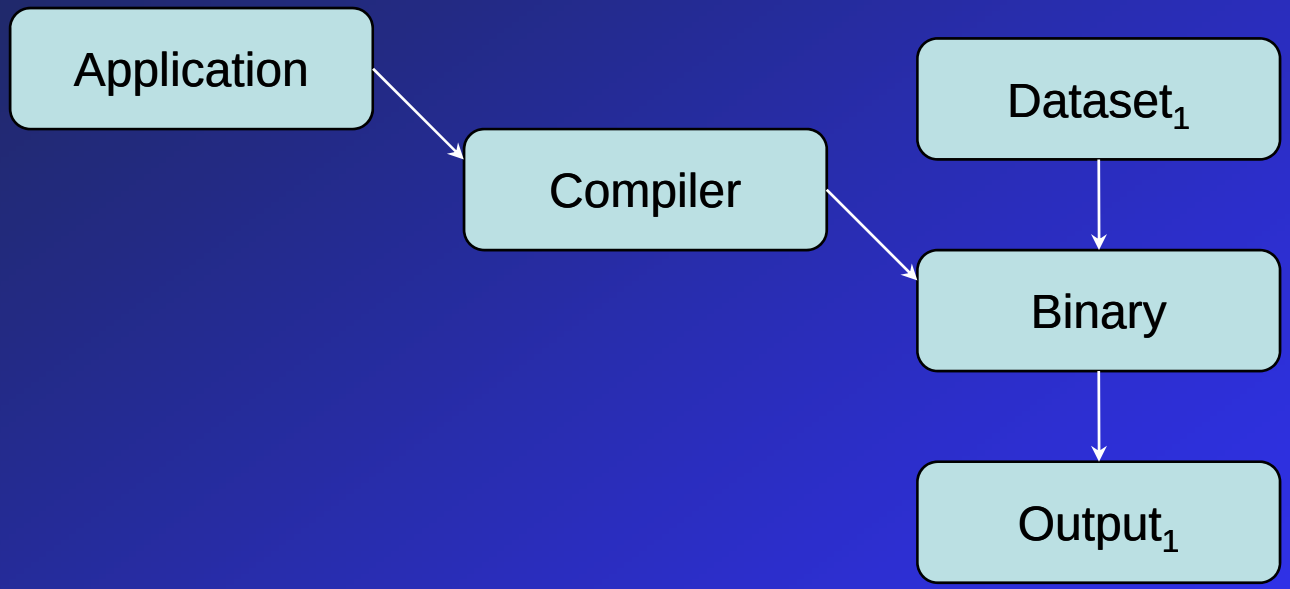


Static optimizations:

- *rigid optimization heuristics with simplified hardware models,*
- *lack of run-time information*
- *one dataset*

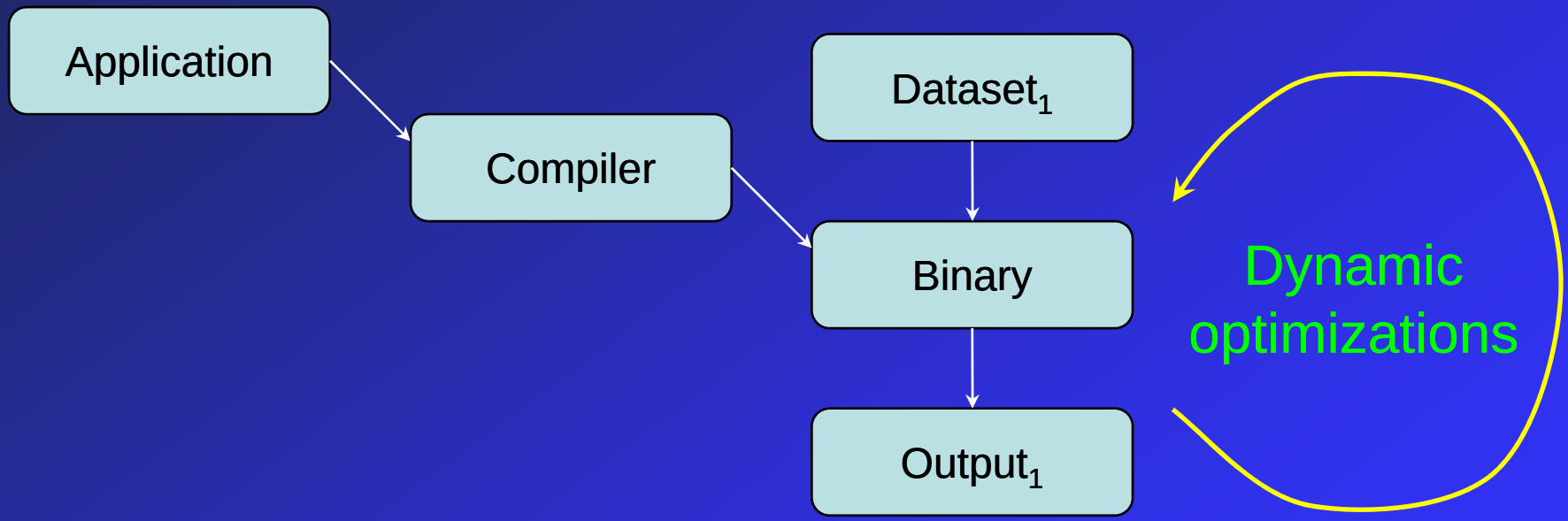
Current methods

Some existing solutions:



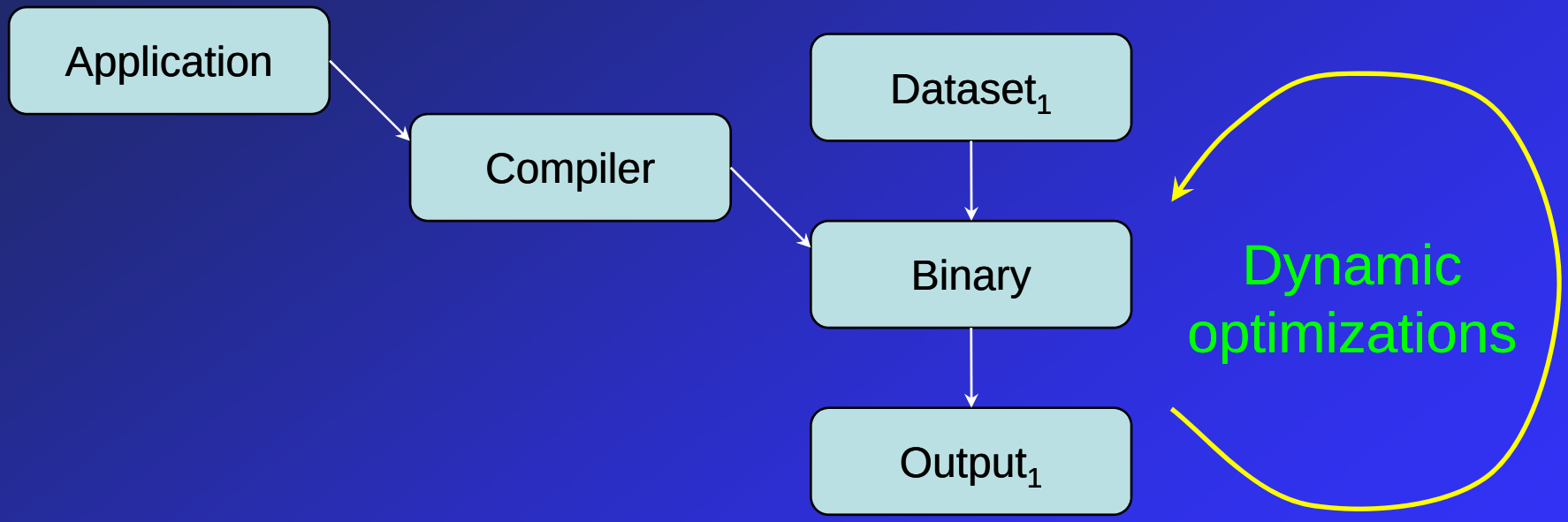
Current methods

Some existing solutions:



Current methods

Some existing solutions:

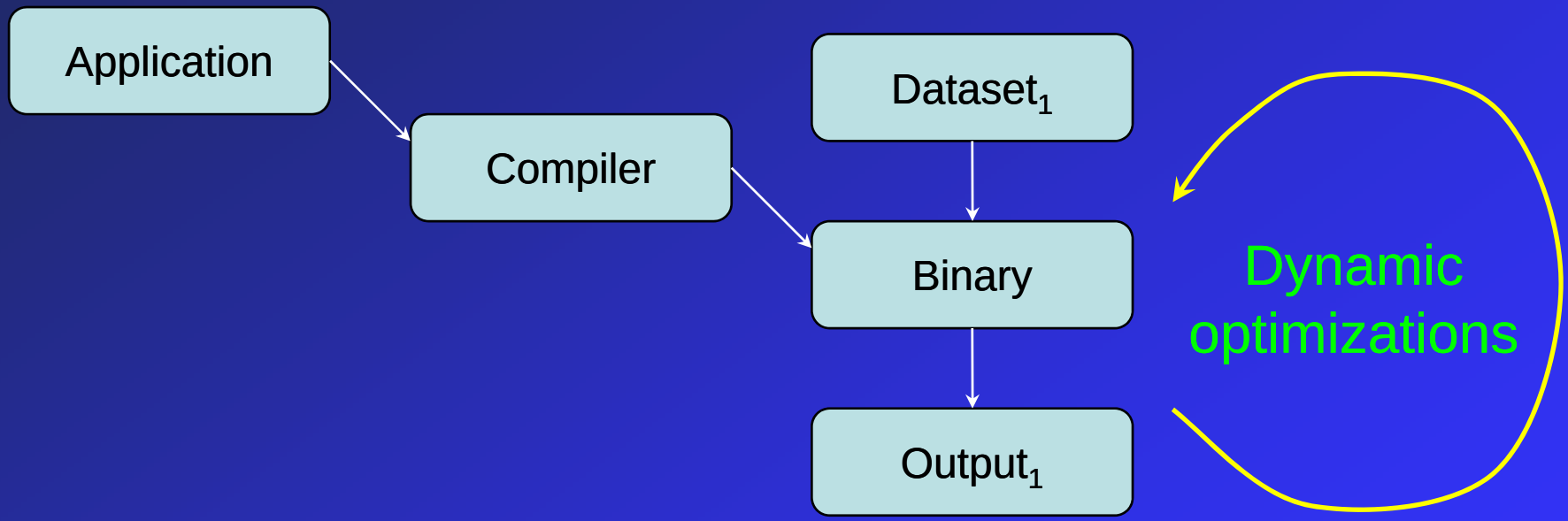


Pros: *run-time information,*

potentially more than one dataset

Current methods

Some existing solutions:



Pros: *run-time information,*

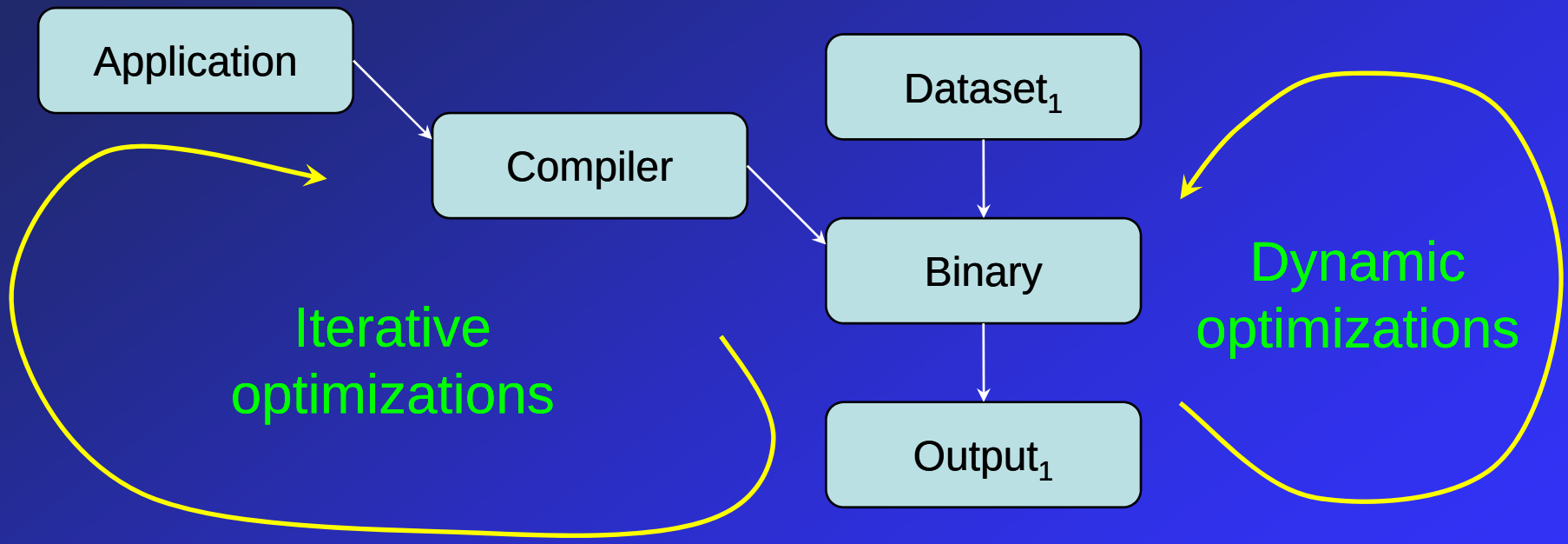
potentially more than one dataset

Cons: *restrictions on optimization time,*

simple optimizations

Current methods

Some existing solutions:



Pros: *run-time information,*

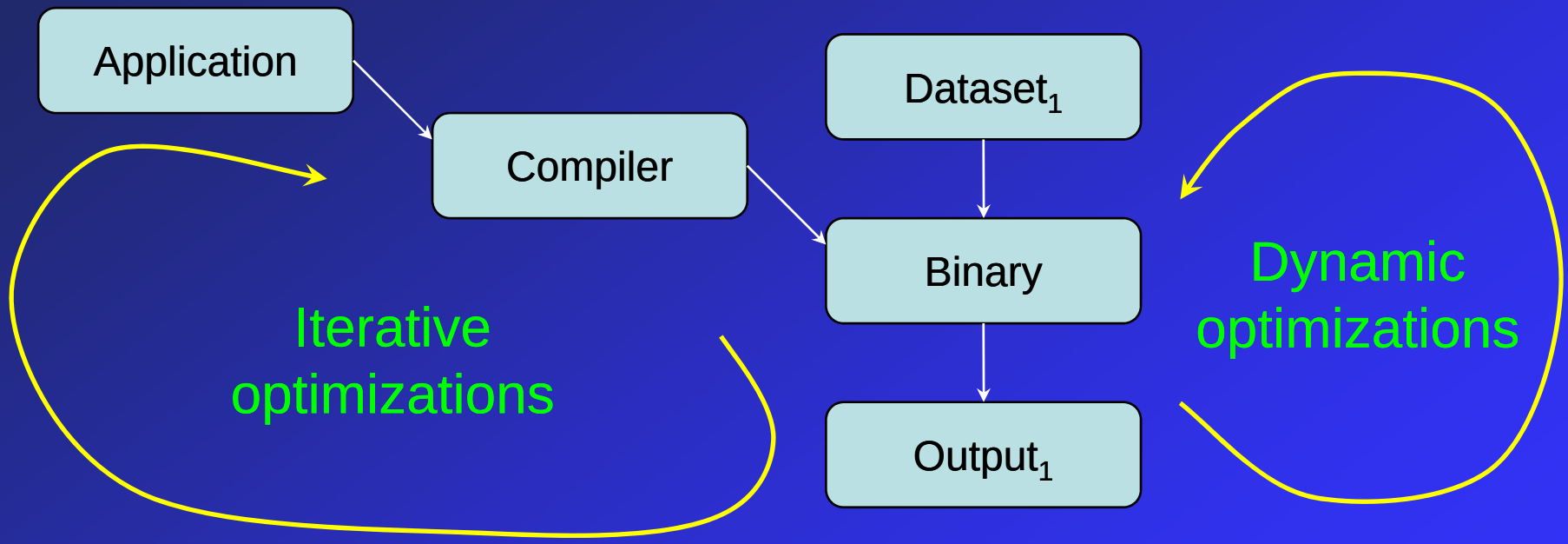
potentially more than one dataset

Cons: *restrictions on optimization time,*

simple optimizations

Current methods

Some existing solutions:



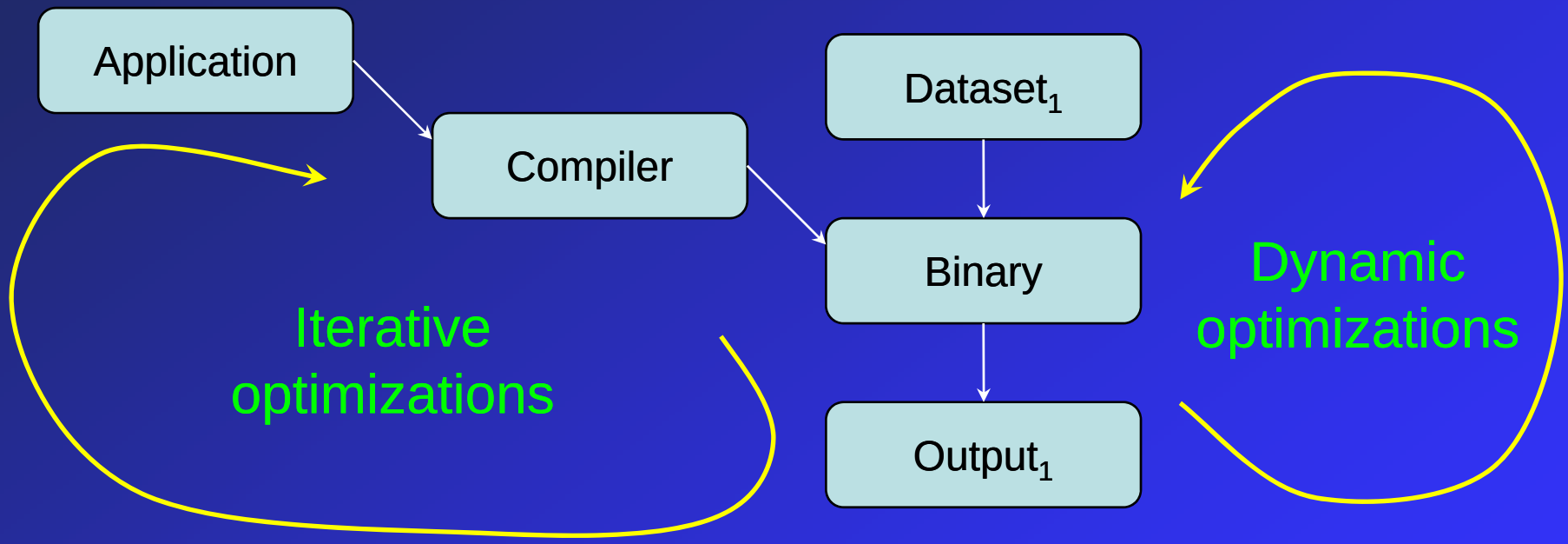
Pros: *powerful transformation space exploration*

Pros: *run-time information, potentially more than one dataset*

Cons: *restrictions on optimization time, simple optimizations*

Current methods

Some existing solutions:



Pros: *powerful transformation space exploration*

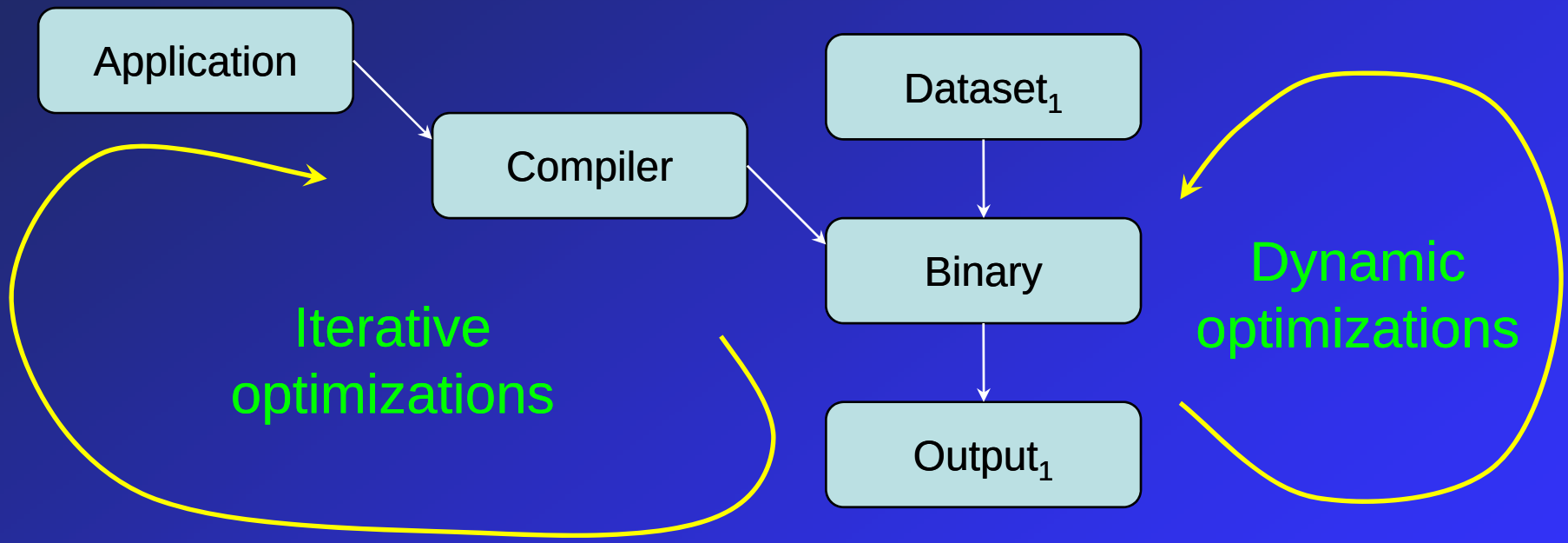
Cons: *slow, one dataset*

Pros: *run-time information, potentially more than one dataset*

Cons: *restrictions on optimization time, simple optimizations*

Current methods

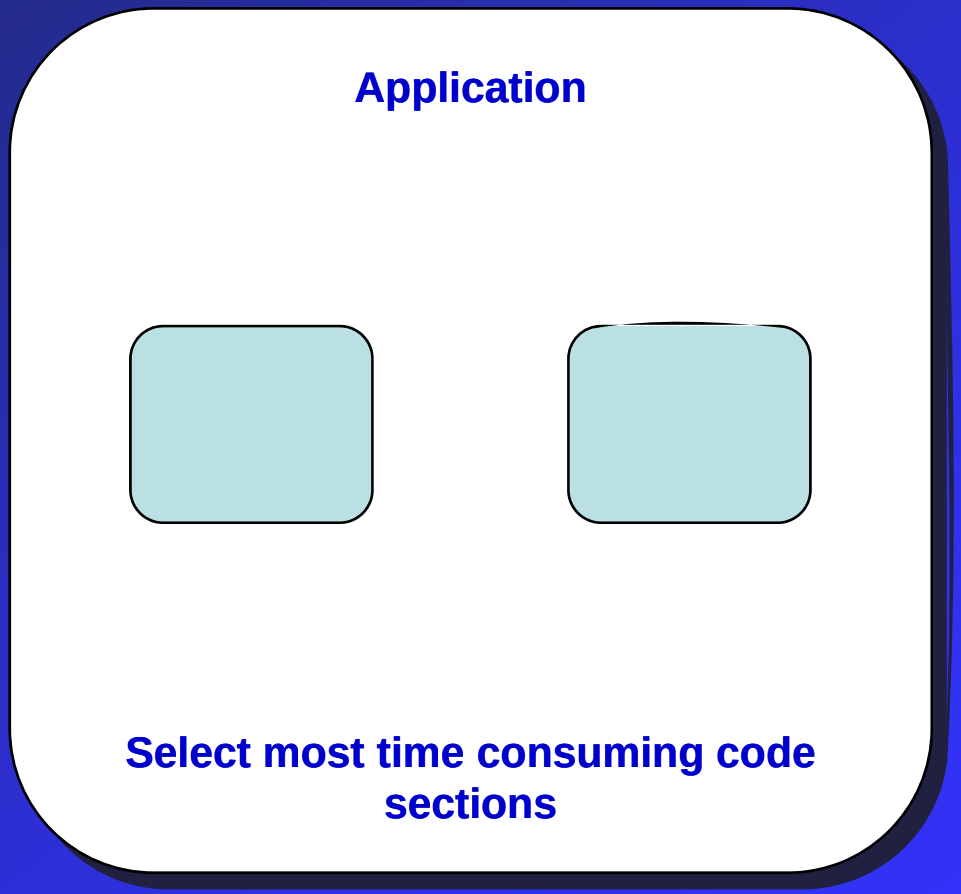
Can we combine both?



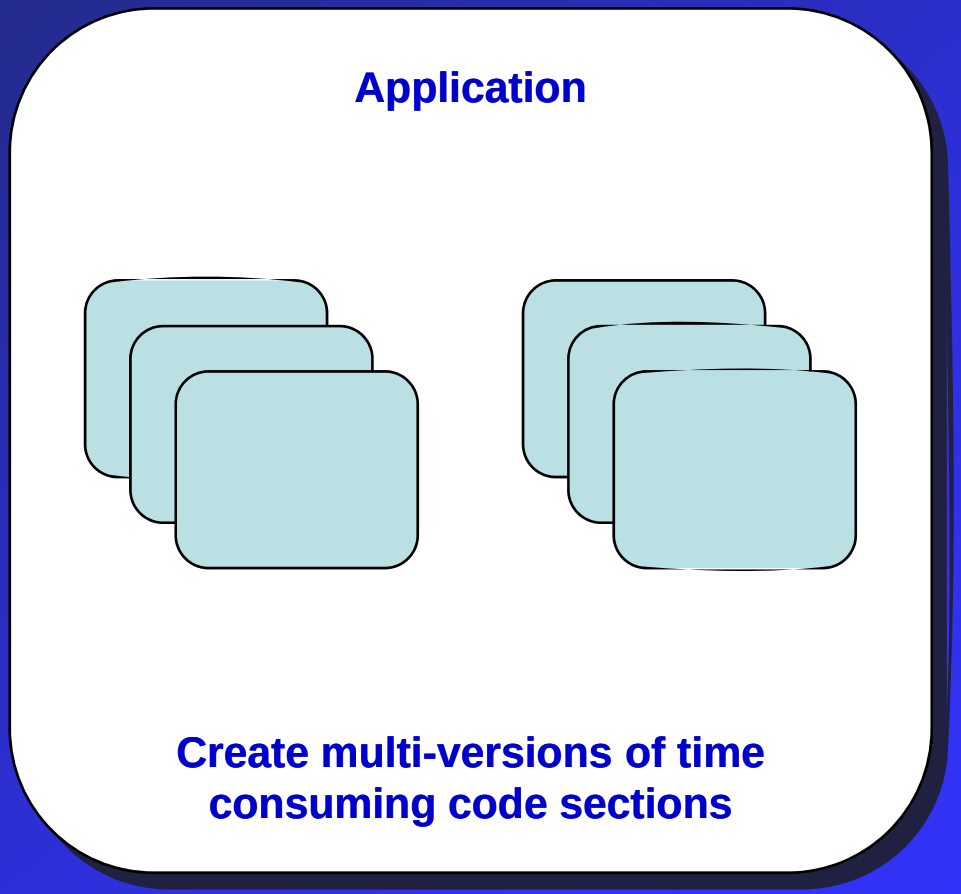
Combination of

powerful transformation space exploration,
run-time information
self-adaptable code

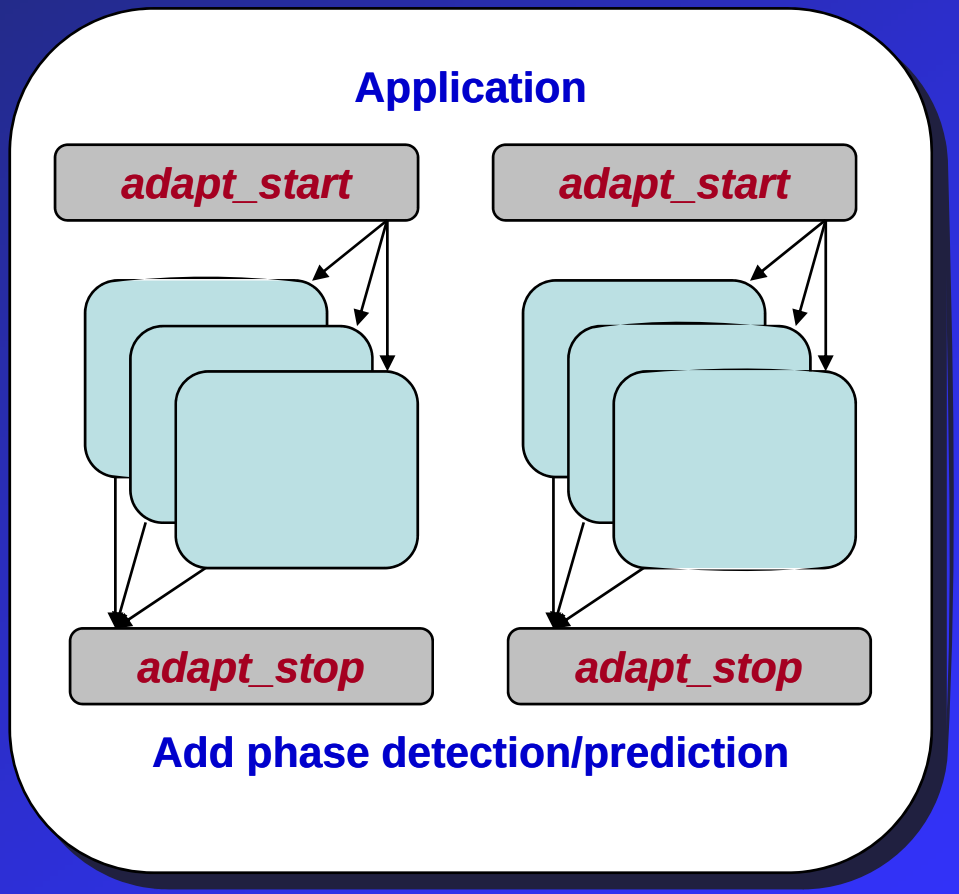
Our approach: static multiversioning



Our approach: static multiversioning

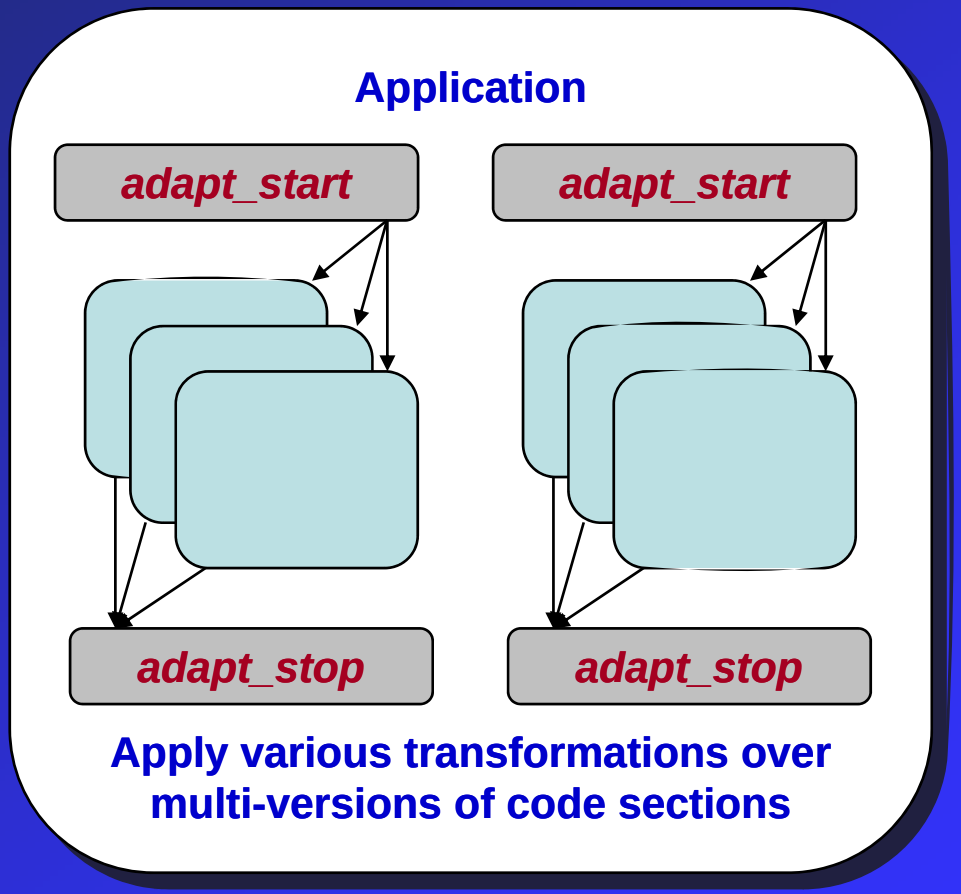


Our approach: static multiversioning



Our approach: static multiversioning

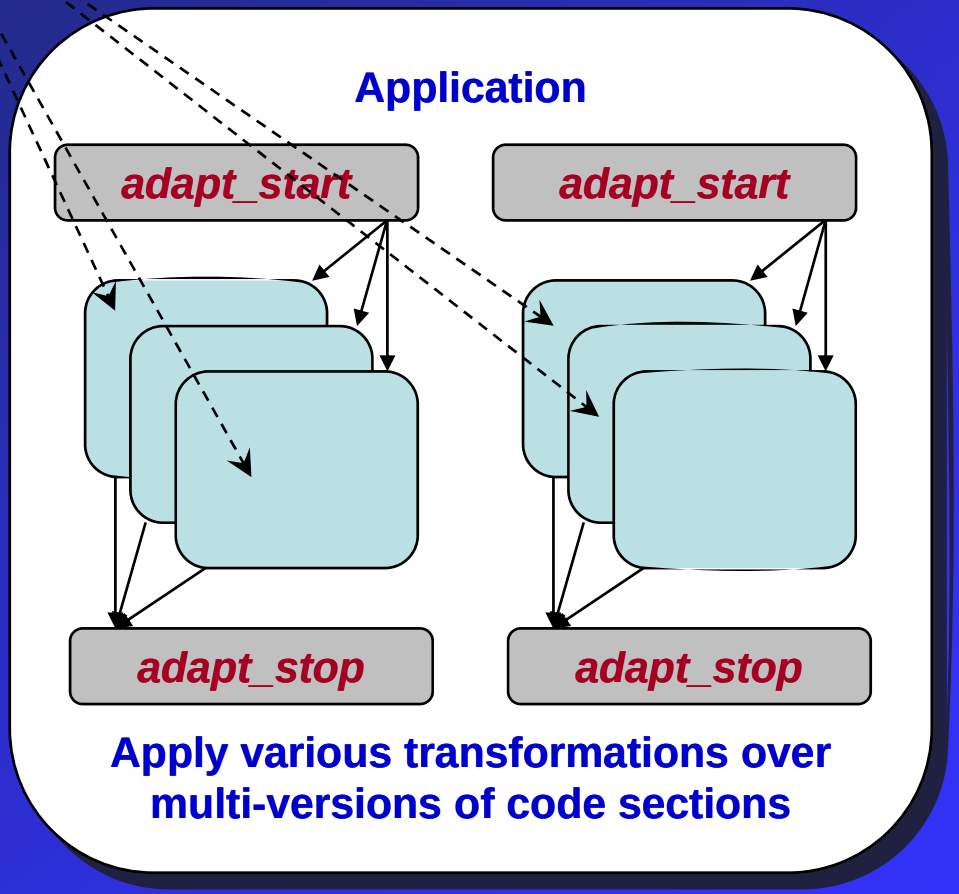
Transformations



Our approach: static multiversioning

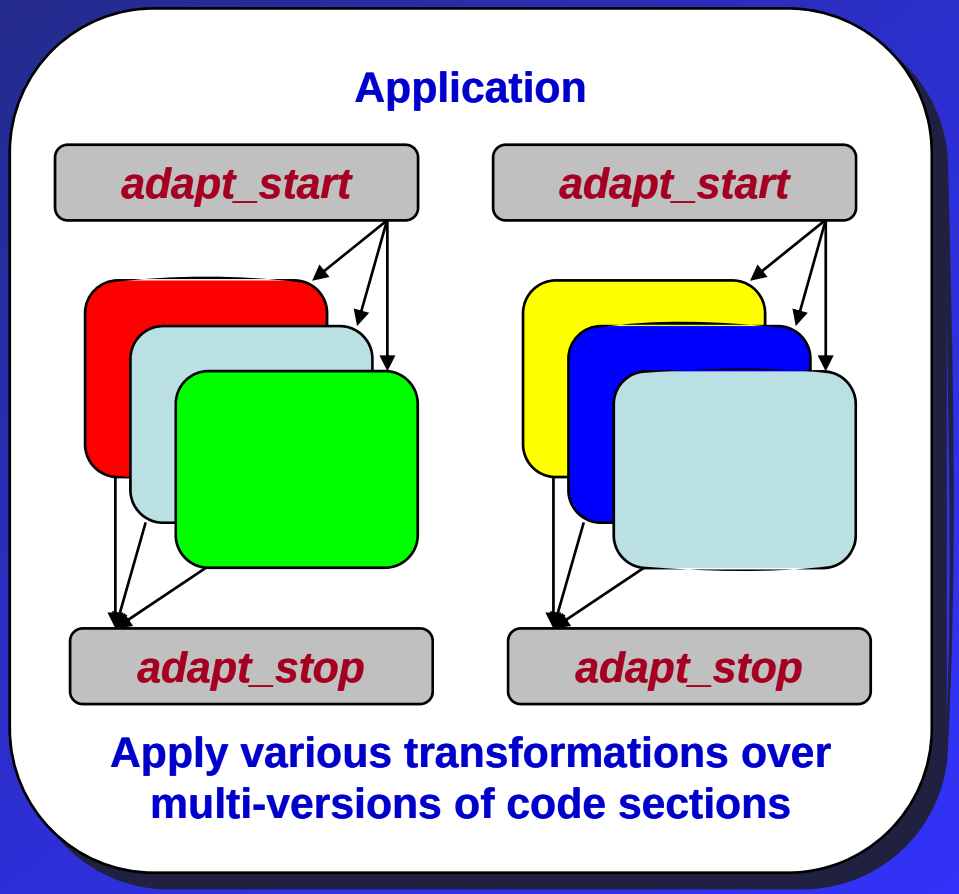
Fine-grain internal compiler (PathScale, Open64, ORC, gcc) transformations using Interactive Compilation Interface (ICI)

Transformations

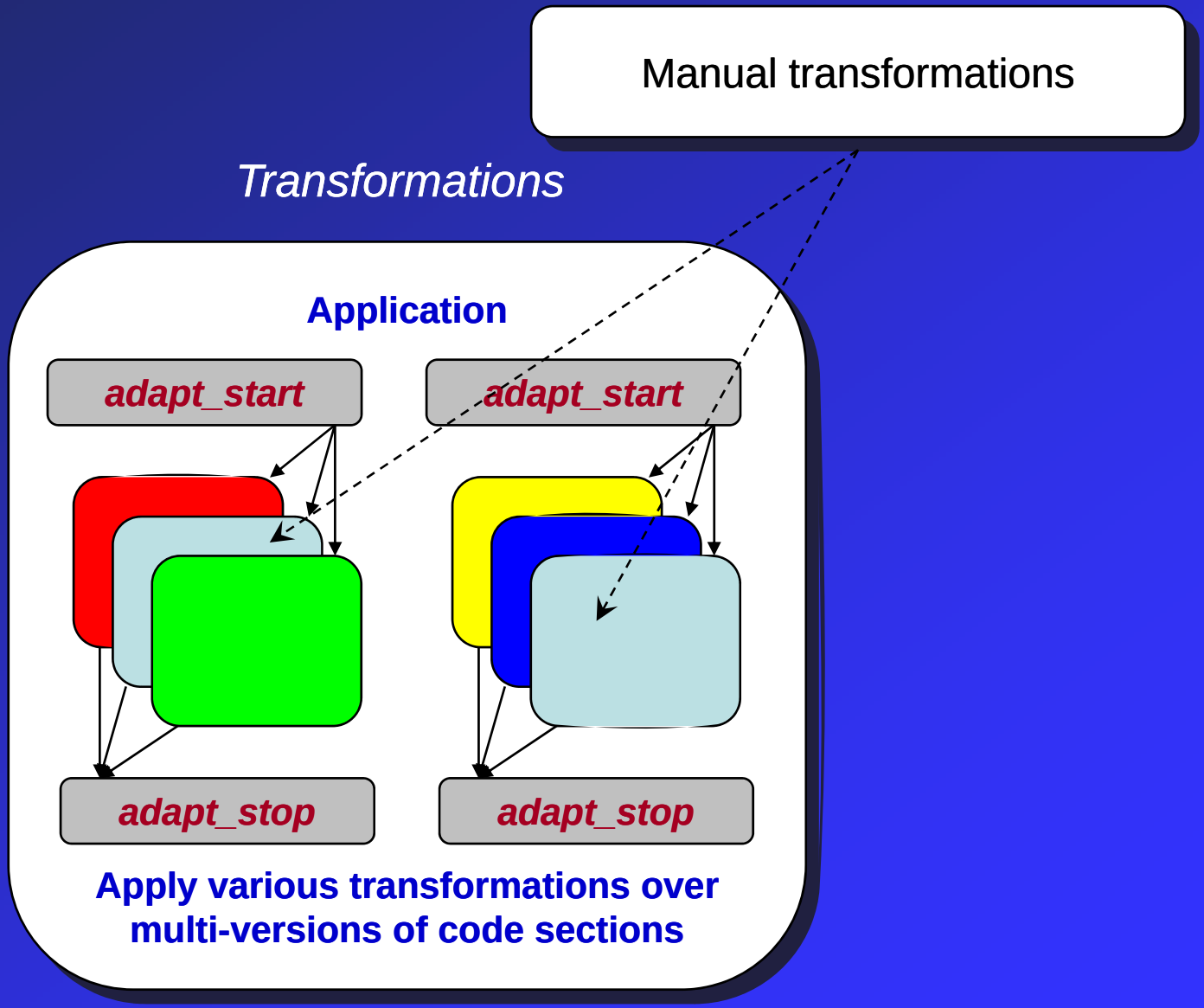


Our approach: static multiversioning

Transformations

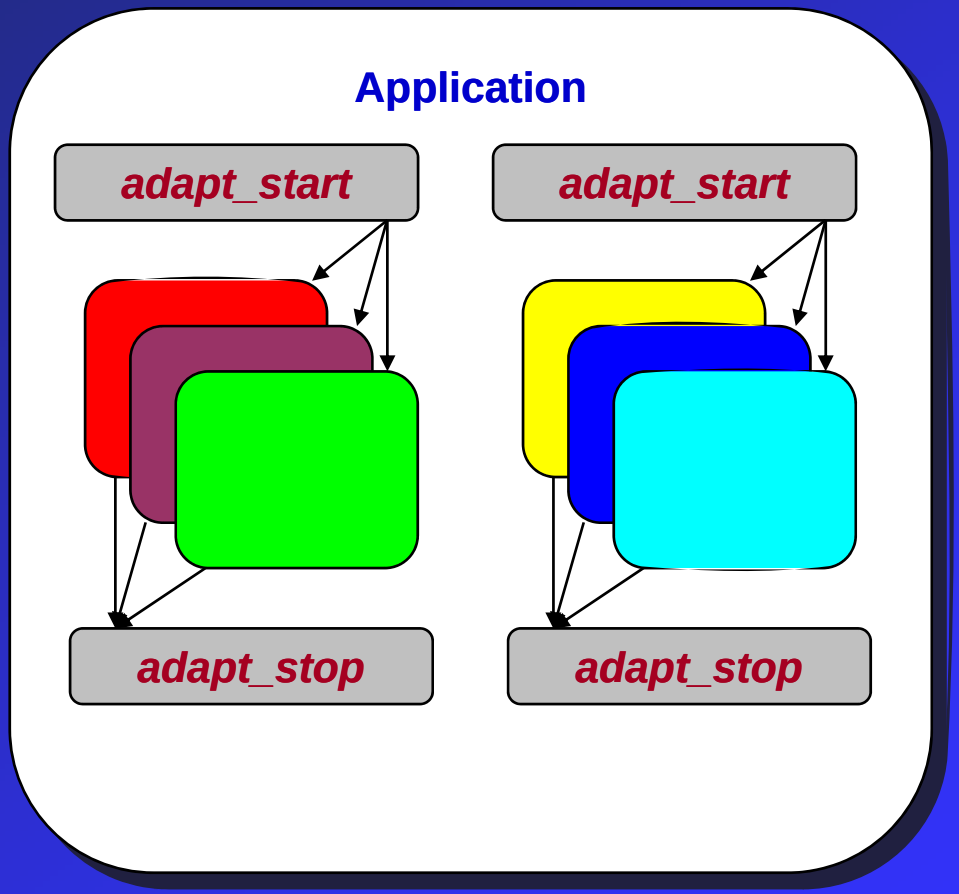


Our approach: static multiversioning



Our approach: static multiversioning

Final instrumented program



Our approach: static multiversioning

```
void mult(int NM)
{
 int i, j, k;
 int fselect;
 co_adapt_select(&fselect);
 if (fselect==1)  mult_clone(NM);

 co_adapt_start(1,0);
 for (i = 0; i < NM; i++)
  for (j = 0; j < NM; j++)
   for (k = 0; k < NM; k++)
    c_matrix[i+NM*j]=c_matrix[i+NM*j]+a_matrix[i+NM*k]*b_matrix[k+NM*j];
 co_adapt_stop(1,0);
}

void mult_clone(int NM)
{
 int i, j, k;
 co_adapt_start(1,1);
 for (i = 0; i < NM; i++)
  for (j = 0; j < NM; j++)
   for (k = 0; k < NM; k++)
    c_matrix[i+NM*j]=c_matrix[i+NM*j]+a_matrix[i+NM*k]*b_matrix[k+NM*j];
 co_adapt_stop(1,1);
}
```

Run-time Adaptation

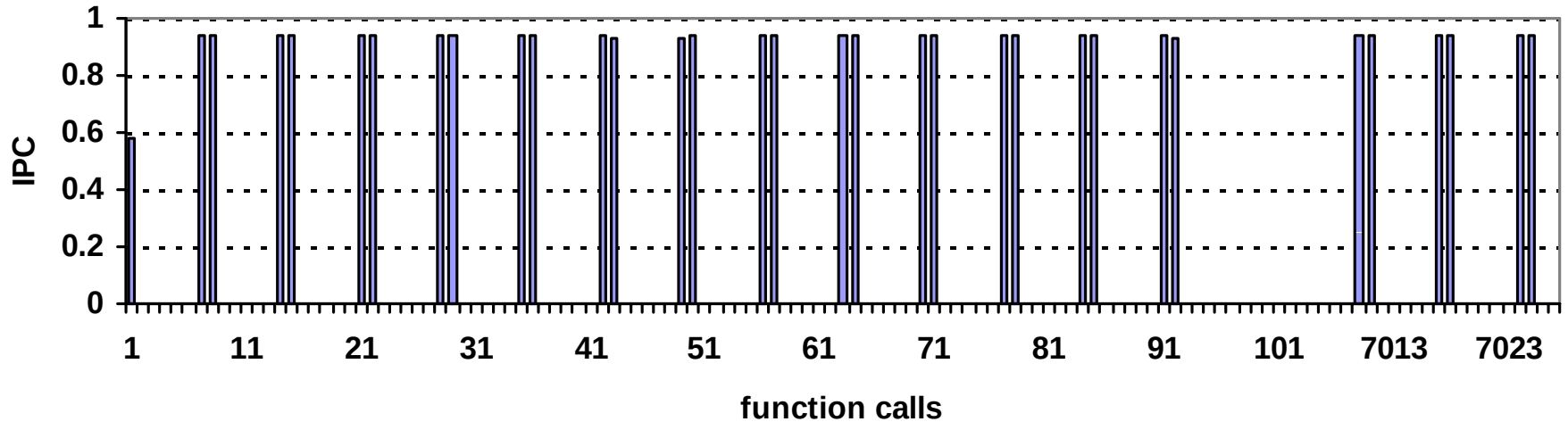
Depends on program behaviour

Programs with regular behavior

Programs with irregular behavior

Adaptation for regular behaviour

IPC for subroutine resid of benchmark mgrid across calls

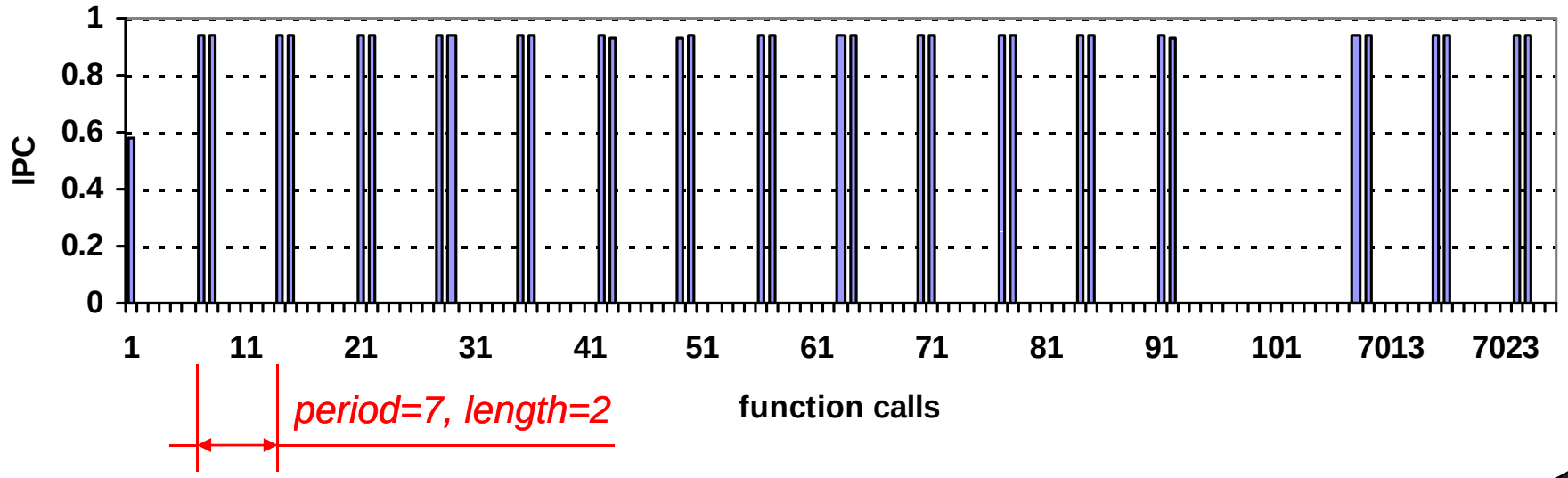


- Detect regular (stable) patterns of behaviour (phases) - we define stability as 3 consecutive or periodic executions with the same IPC
- Predict further occurrences with the same IPC

(using period and length of regions with stable performance)

Adaptation for regular behaviour

IPC for subroutine resid of benchmark mgrid across calls

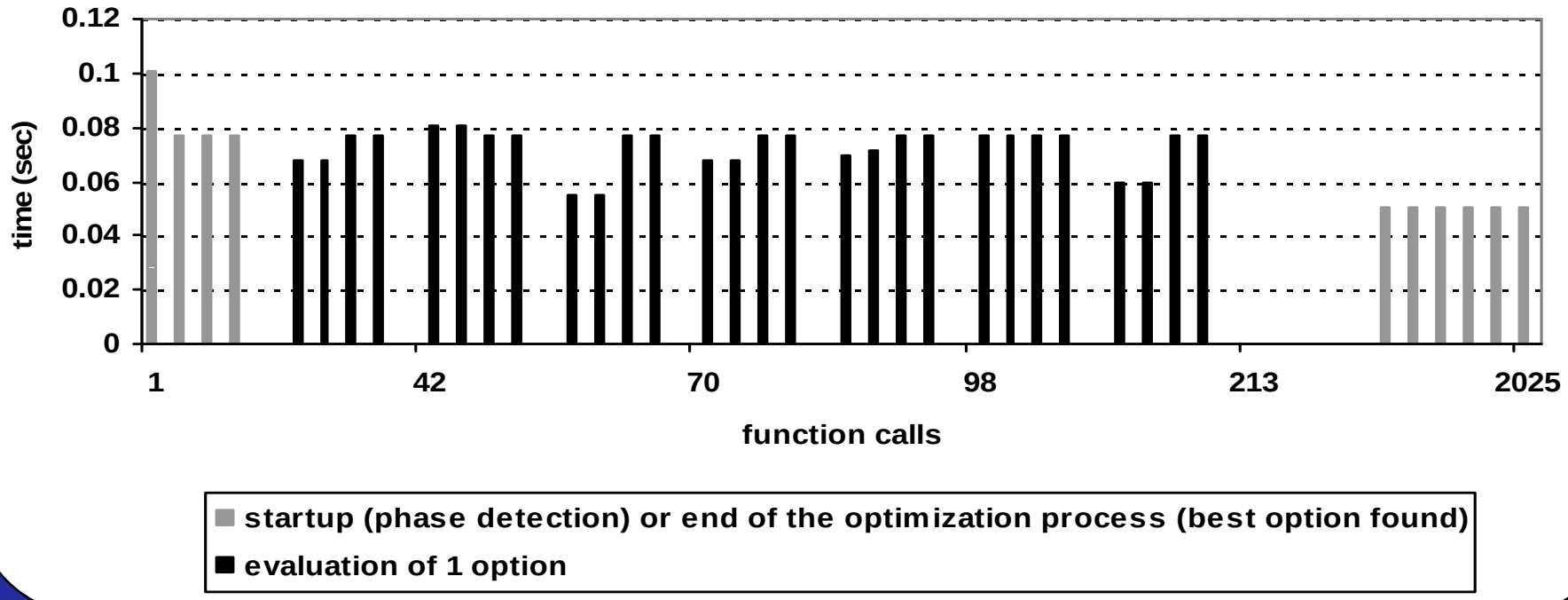


- Detect regular (stable) patterns of behaviour (phases) - we define stability as 3 consecutive or periodic executions with the same IPC
- Predict further occurrences with the same IPC

(using period and length of regions with stable performance)

Adaptation for regular behaviour

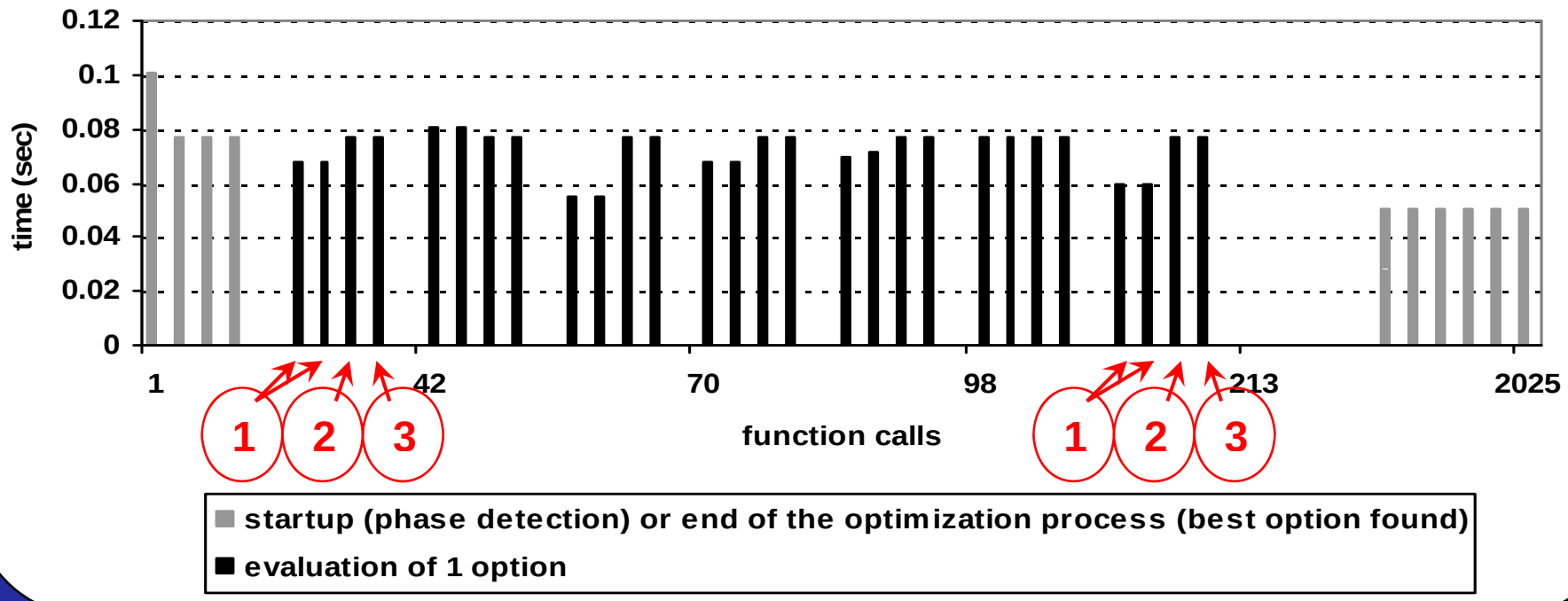
Execution times for subroutine resid of benchmark mgrid across calls



1) Consider new code version evaluated after 2 consecutive executions of the code section with the same performance
2) Ignore one next execution to avoid transitional effects
3) Check baseline performance (to verify stability prediction)

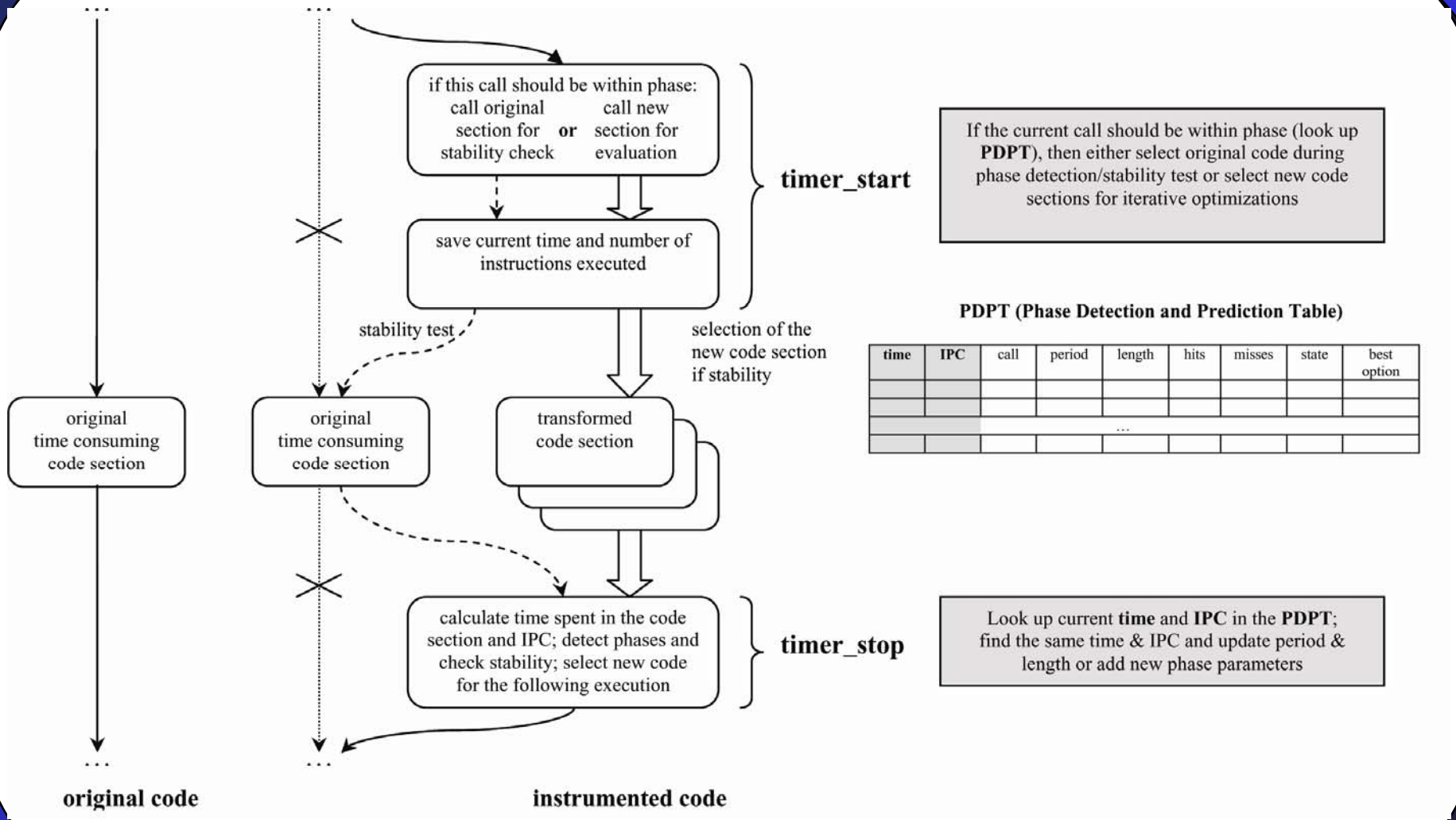
Adaptation for regular behaviour

Execution times for subroutine resid of benchmark mgrid across calls



1) Consider new code version evaluated after 2 consecutive executions of the code section with the same performance
2) Ignore one next execution to avoid transitional effects
3) Check baseline performance (to verify stability prediction)

Adaptation for regular behaviour



If the current call should be within phase (look up **PDPT**), then either select original code during phase detection/stability test or select new code sections for iterative optimizations

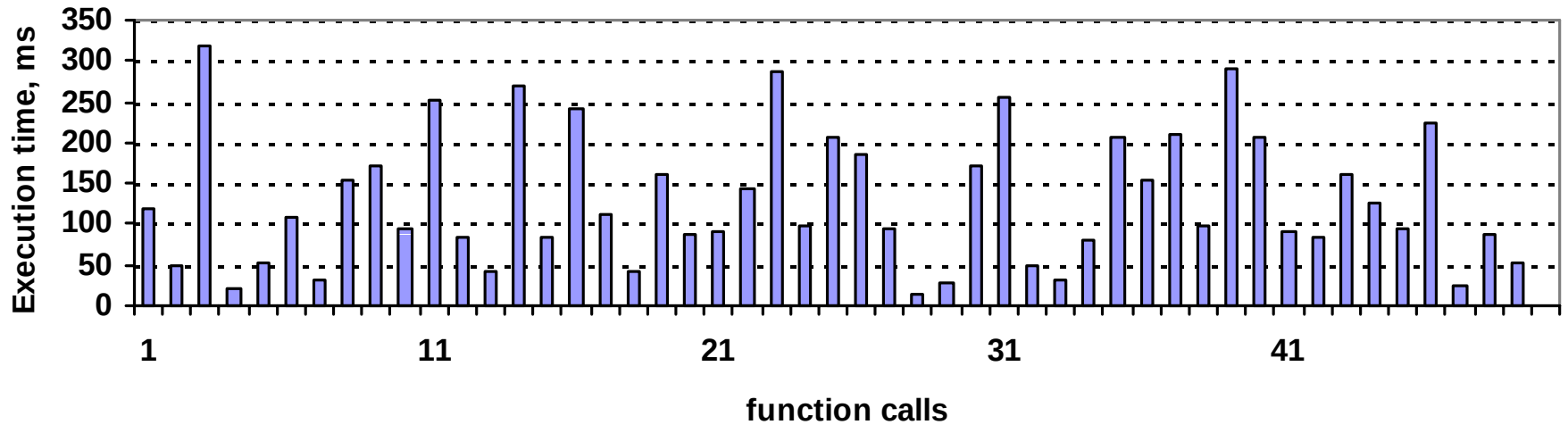
PDPT (Phase Detection and Prediction Table)

time	IPC	call	period	length	hits	misses	state	best option
		...						

Look up current **time** and **IPC** in the **PDPT**; find the same time & IPC and update period & length or add new phase parameters

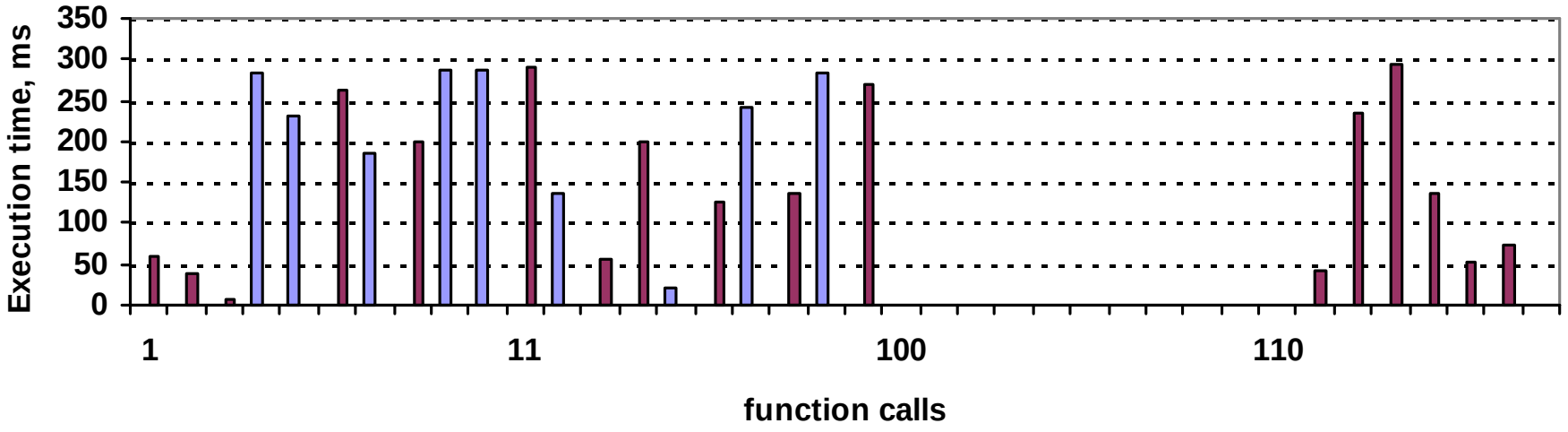
Adaptation for irregular behaviour

Execution time for library subroutine matmul (with 2 different versions)



Adaptation for irregular behaviour

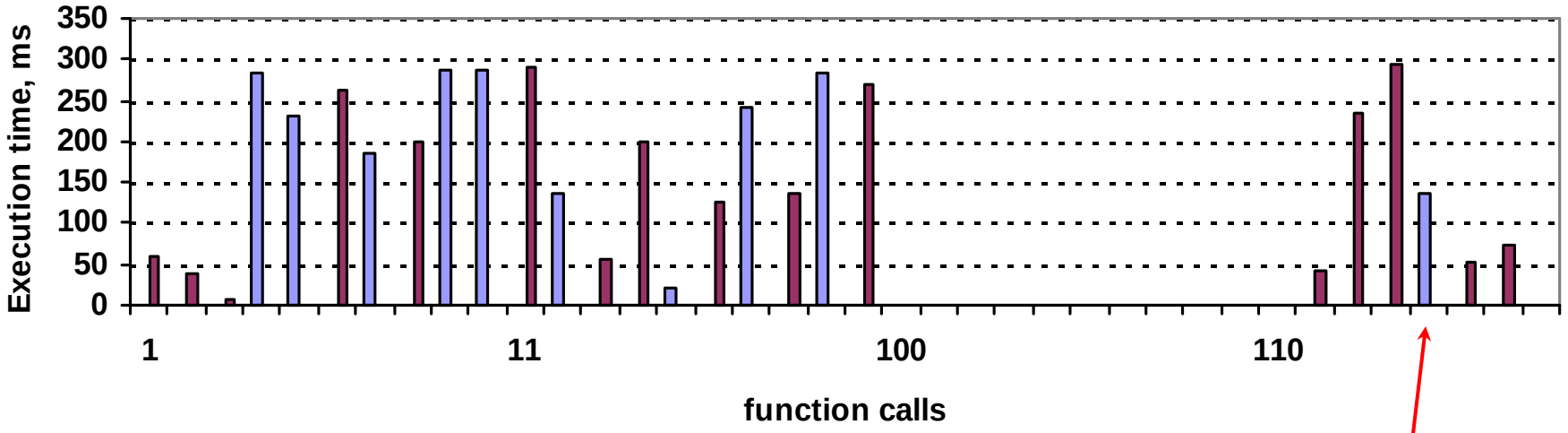
Execution time for library subroutine matmul (with 2 different versions)



- Select versions randomly during a time slot
- At each step calculate execution time per function call and variance
- When variance for all versions is less than some threshold select the best one

Adaptation for irregular behaviour

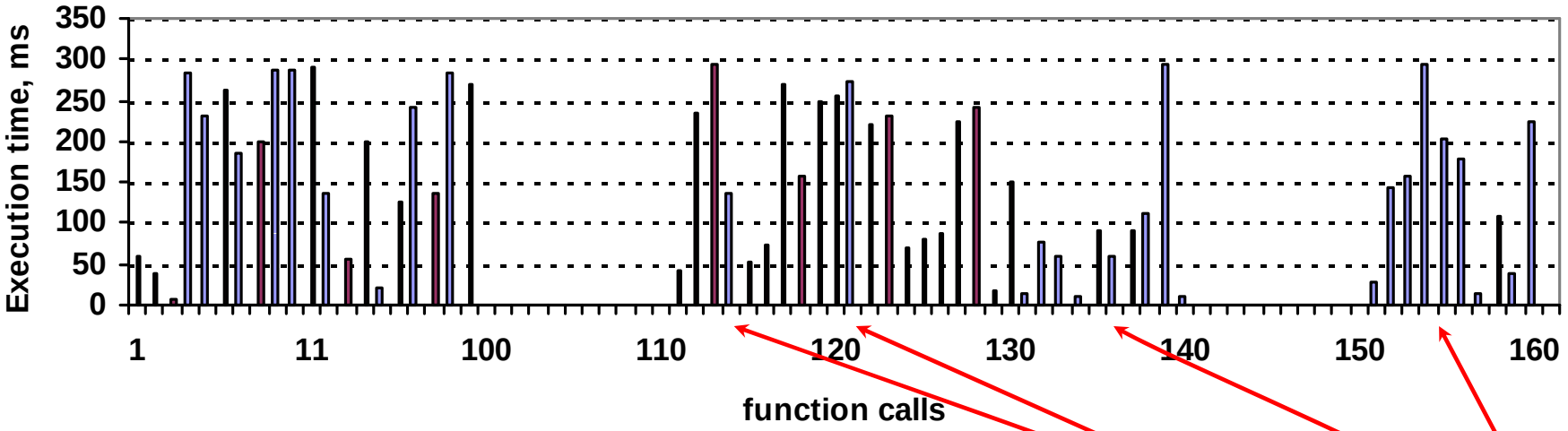
Execution time for library subroutine matmul (with 2 different versions)



- Select versions randomly during a time slot
- At each step calculate execution time per function call and variance
- When variance for all versions is less than some threshold select the best one
- Periodically select non-best version to check if behavior changed

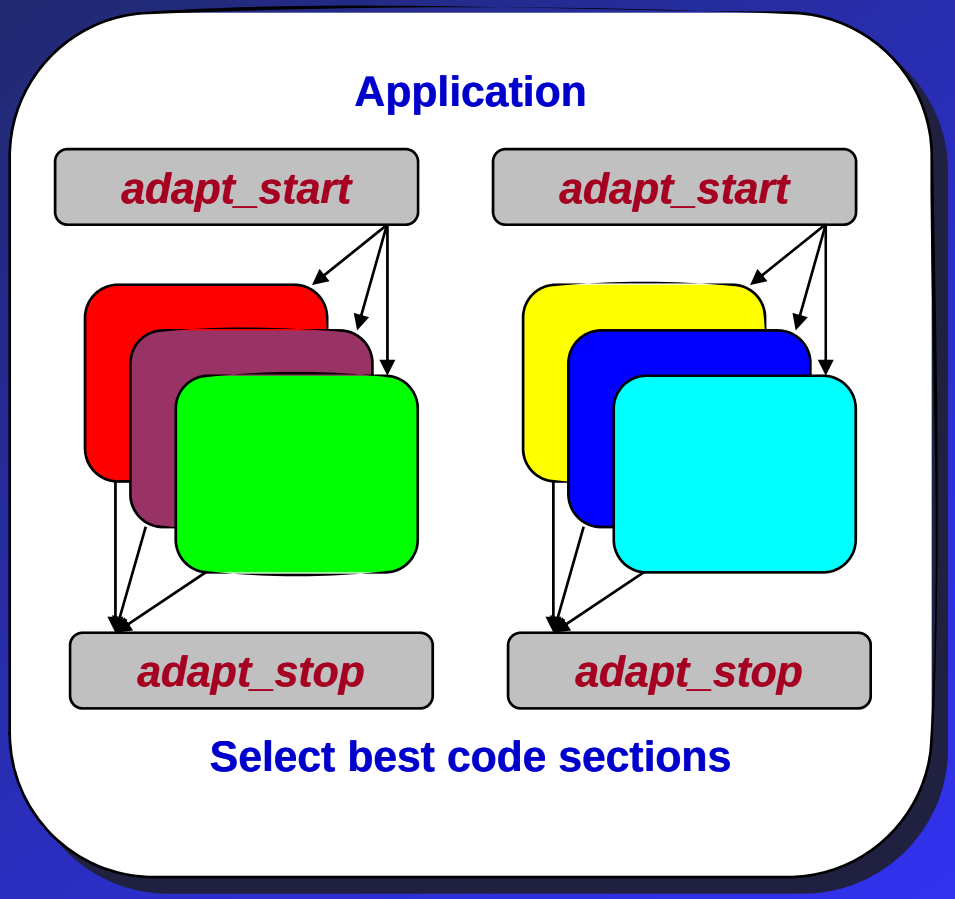
Adaptation for irregular behaviour

Execution time for library subroutine matmul (with 2 different versions)



- Select versions randomly during a time slot (adaptation slot)
- At each step calculate execution time per function call and variance
- When variance for all versions is less than some threshold select the best one
- Periodically select non-best version to check if behavior changed
- If the variance increases, adapt again

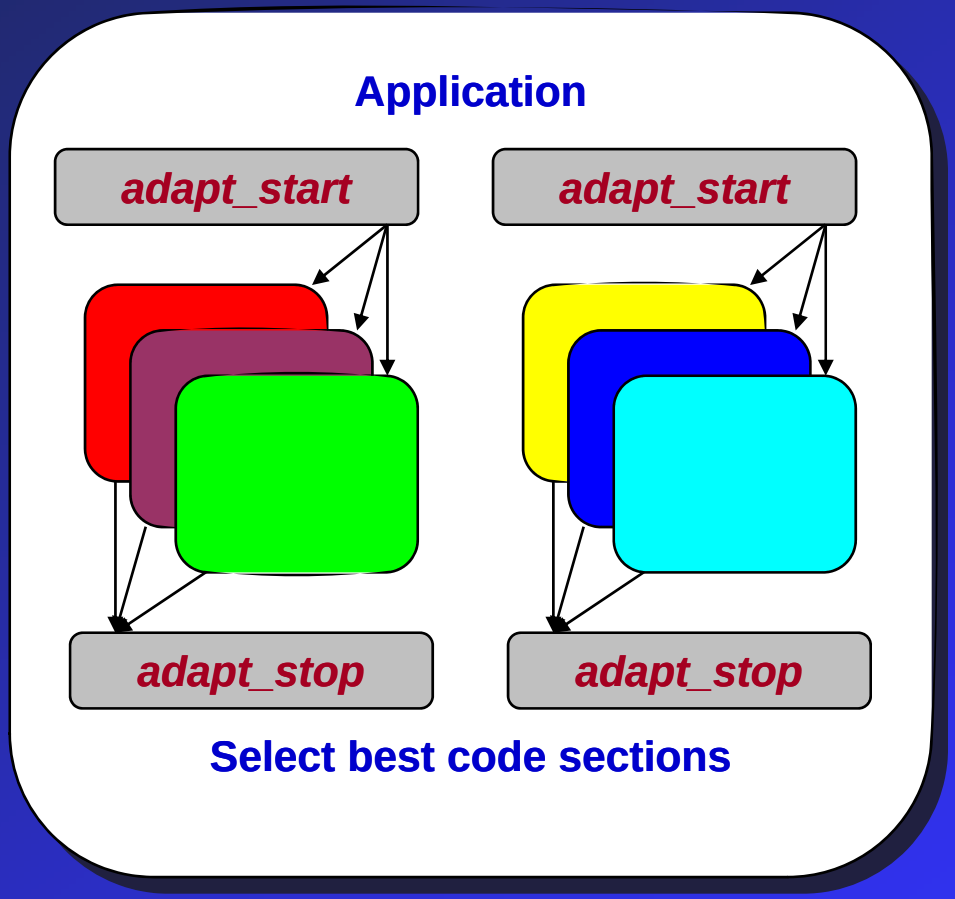
Removing adaptation overhead



Calls to adaptation routines are not direct but through array of functions:

```
static void (*call1[ .. ])();
static void (*call2[ .. ])();
```

Removing adaptation overhead

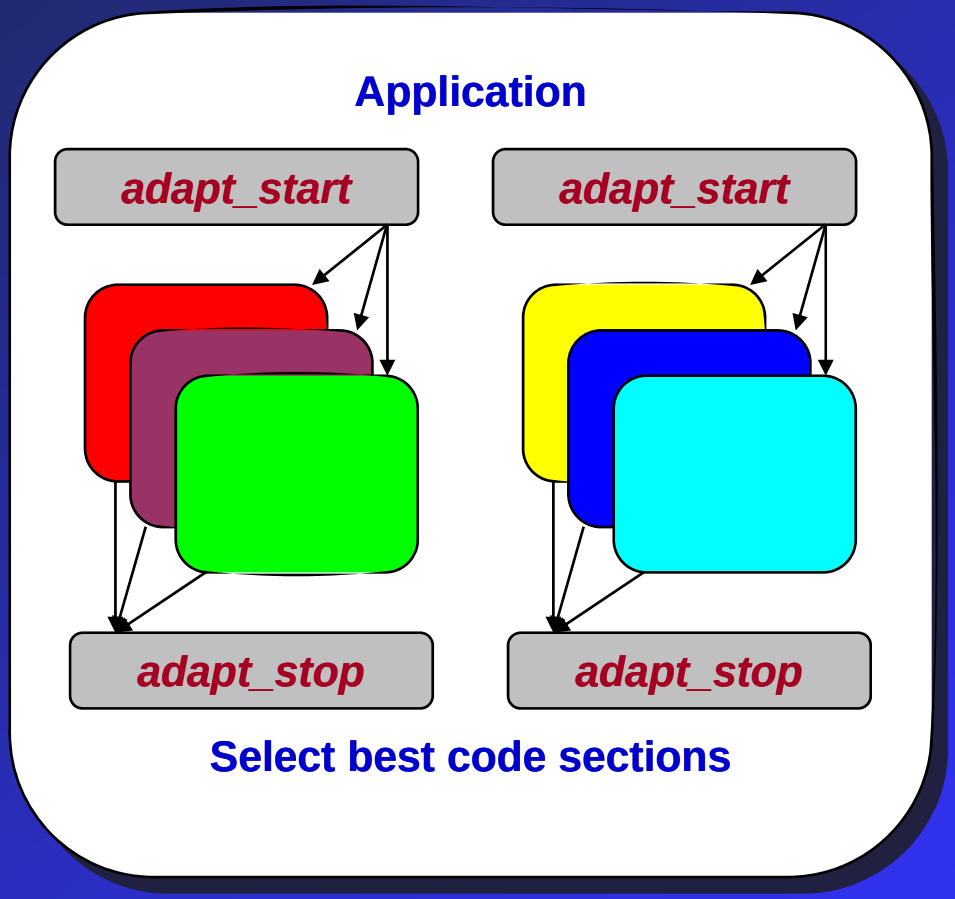


Calls to adaptation routines are not direct but through array of functions:

static void (*call1[..])();
static void (*call2[..])();

If high-overhead is detected – substitute call with dummy function

Removing adaptation overhead



Calls to adaptation routines are not direct but through array of functions:

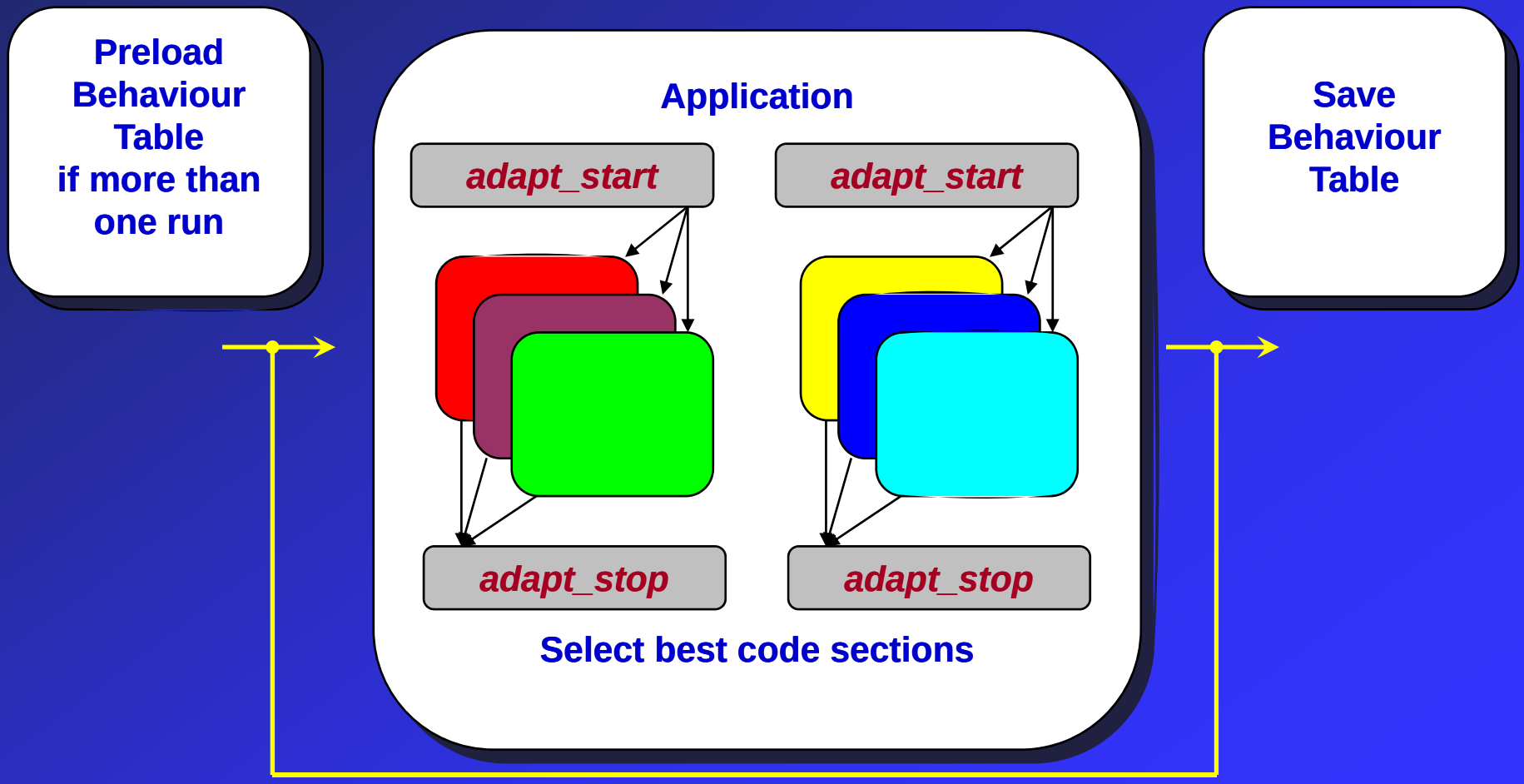
static void (*call1[..])();
static void (*call2[..])();

If high-overhead is detected – substitute call with dummy function

To be able to adapt to new program behaviour later at run-time, periodically restore all calls to adptation routines

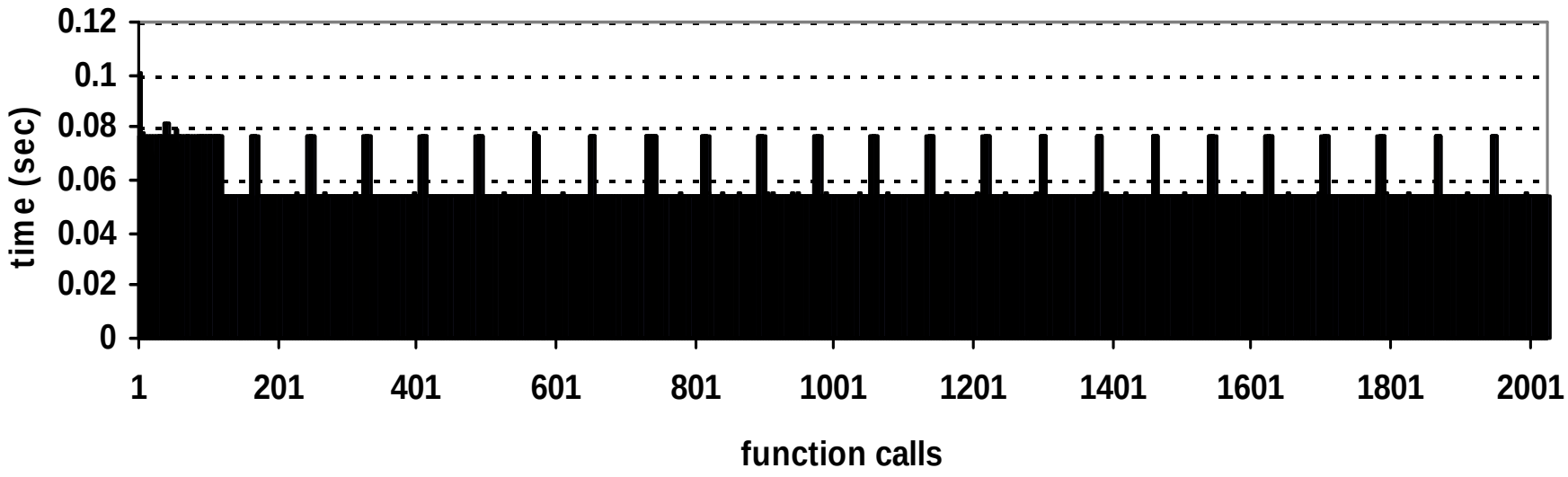
Continuous optimization and adaptation

One or multiple executions
with the same or different datasets:



Continuous optimization and adaptation

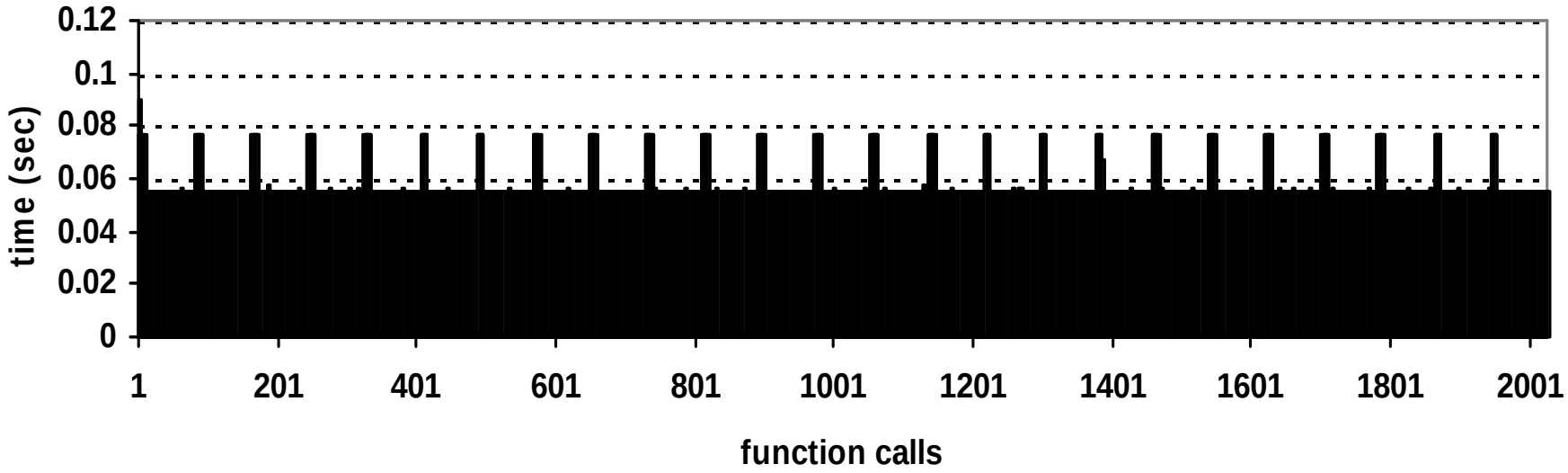
Execution times for subroutine resid of benchmark mgrid across calls



1st run

Continuous optimization and adaptation

Execution times for subroutine resid of benchmark mgrid across calls



2st run, same optimizations

Conclusions

- No sophisticated dynamic optimization/recompilation frameworks;

- Allows complex sequences of compiler or manual transformations at run-time;

- Uses simple low-overhead adaptation technique (for codes with regular and irregular behaviour);

- Combines manual and compiler transformations due to the source-to-source versioning approach

- Enables self-tuning applications adaptable to program and system behaviour, and portable across different architectures

- Enables continuous optimizations across runs with different datasets, transparently to a user

- Reliable, secure, with easy debugging

Current and future work

- public software release soon
- better predictions for version selection
    - use hardware counters
    - use machine learning
    - use decision trees on parameters
- auto-placement of instrumentation (for adaptation)
- adaptation for parallel heterogeneous systems
- continuous optimization framework
- VM implementation (JIT, .NET)
- OS kernel adaptation

Questions?

This work is partly funded by HiPEAC network

http://www.hipeac.net

Thanks to Cupertino Miranda (INRIA engineer) for the help with implementations

SMART workshop and GCC tutorial at HiPEAC conference in Ghent, 28th of January, 2007

Contact e-mail: grigori.fursin@inria.fr

More information, updates and software will be available here soon:

http://fursin.net/gfursin/research_desc.html