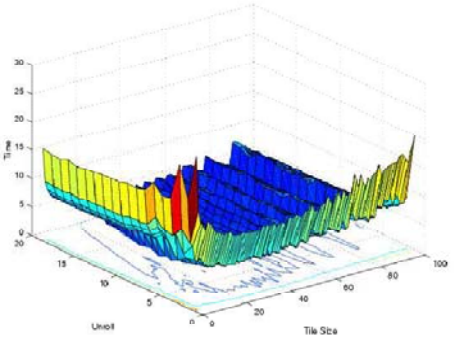
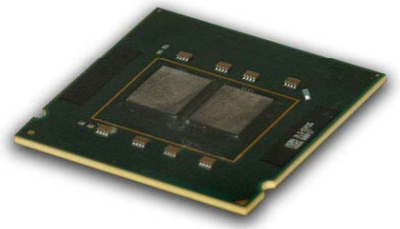


Dynamic compilation and run-time adaptation. Machine learning

Grigori Fursin

Alchemy group, INRIA Saclay, France



Course overview

Assume that all understand basics of computer architecture and compilation process.

Focus on compilers that map user program to machine code

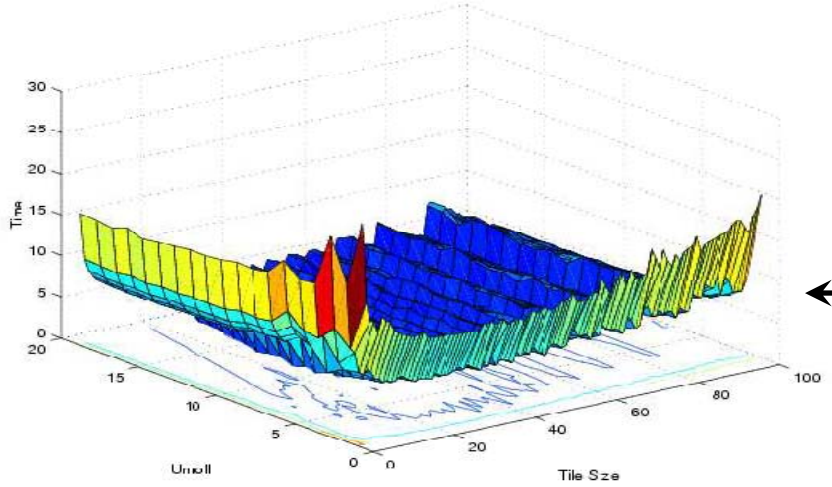
Explain general major compilation problems instead of focusing on individual components

Describe current major research areas for compilation and optimization

- *Reminder: Feedback directed compilation and optimization*
- *Dynamic compilation and optimization*
- *Machine learning and future directions*

Reminder

Optimization spaces (set of all possible program transformations) are large, non-linear with many local minima



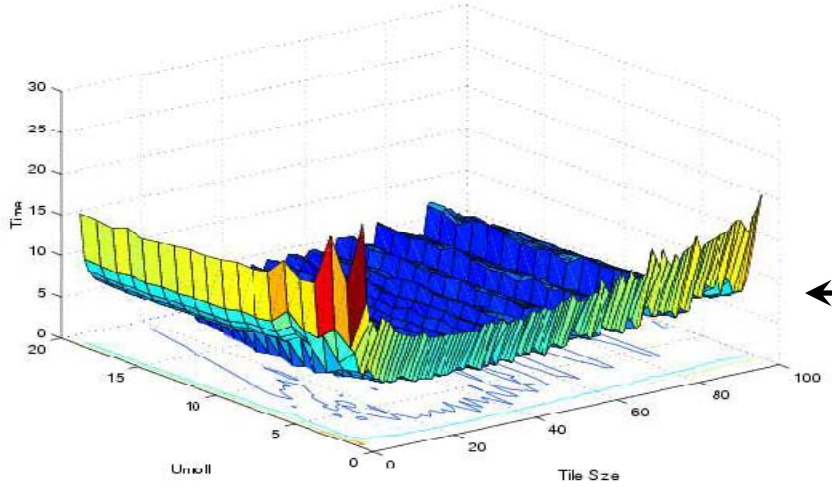
Finding a good solution may be long and non-trivial

matmul, 2 transformations, search space = 2000

swim, 3 transformations, search space = 10^{52}

Reminder

Optimization spaces (set of all possible program transformations) are large, non-linear with many local minima



Finding a good solution may be long and non-trivial

matmul, 2 transformations, search space = 2000

swim, 3 transformations, search space = 10^{52}

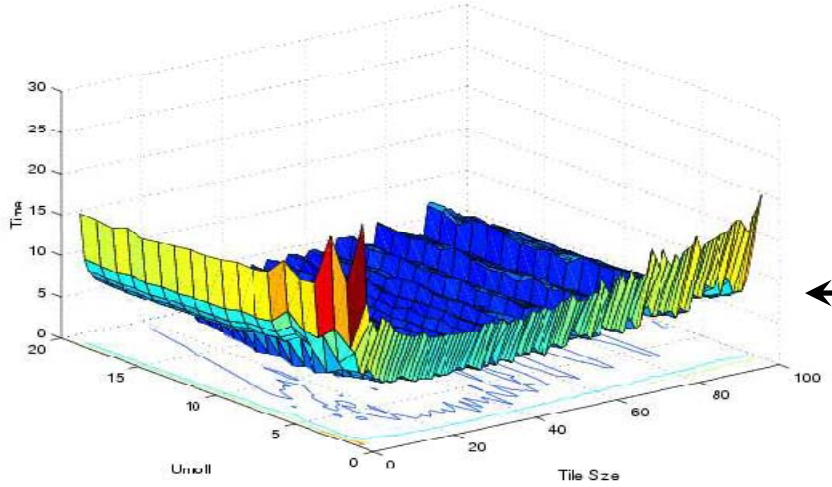
Recent technique - iterative compilation: learn program behavior across executions

High potential (O'Boyle, Cooper), but:

- slow
- the same dataset is used
- no run-time adaptation
- no optimization knowledge reuse

Reminder

Optimization spaces (set of all possible program transformations) are large, non-linear with many local minima



Finding a good solution may be long and non-trivial

matmul, 2 transformations, search space = 2000

swim, 3 transformations, search space = 10^{52}

Recent technique - iterative compilation: learn program behavior across executions

High potential (O'Boyle, Cooper), but:

- slow
- the same dataset is used
- no run-time adaptation
- no optimization knowledge reuse

Solving these problems is non-trivial

Dynamic techniques

- All today's techniques focus on delaying some or all of the optimizations to runtime

- This has the benefit of knowing the exact runtime control-flow, hotspots, data values, memory locations and hence complete program knowledge

- It thus largely eliminates many of the undecidable issues of compile-time optimization by delaying until runtime

- However, the cost of analysis/optimization is now crucial as it forms a runtime overhead. All techniques characterized by trying to exploit runtime knowledge with minimal cost

Background

- Delaying compiler operations until runtime has been used for many years
- Interpreters translates and execute at runtime
- Languages developed in the 60s ie Algol 68 allowed dynamic memory allocation relying on language specific runtime system to mange memory
- LISP has runtime type checking of objects
- Smalltalk in the 80s deferred compilation to runtime to reduce the amount of compilation otherwise required in the OO setting
- Java applications are compiled into bytecode to run on Java Virtual Machines (JVM) thus making them portable across architectures
- .NET applications (mainly for Windows) similarly execute in a run-time environment called Common Language Environment (CLR)

Runtime specialization

- For many, runtime optimization is “adaptive optimization”
- Although wide range of techniques, all are based around runtime specialization
- Constant propagation is a simple example
- Specializing an interpreter with respect to a program gives a compiler
- Can we specialize at runtime to gain benefit with minimal overhead?
Statically inserted selection code vs **parameterized code** vs **runtime generation**

Different techniques

Static code selection

```
IF (N<M) THEN
 DO I =1,N
  DO J =1,M
    ...
  ENDDO
 ENDDO
ELSE
 DO J =1,M
  DO I =1,N
    ...
  ENDDO
 ENDDO
ENDIF
```

Parameterized

```
IF (N<M) THEN
 U1 = N
 U2 = M
ELSE
 U1 = M
 U2 = N
ENDIF
DO I1 =1,U1
  DO I2= 1,U2
    ...
  ENDDO
ENDDO
```

Code generation

```
gen_nest1(fp,N,M)
(*fp)()
```

DyC

- One of the best known dynamic program specializations techniques based on dynamic code generation

- The user annotates the program defining where there may be opportunities for runtime specialization. Marks variables and memory locations that are static within a particular scope

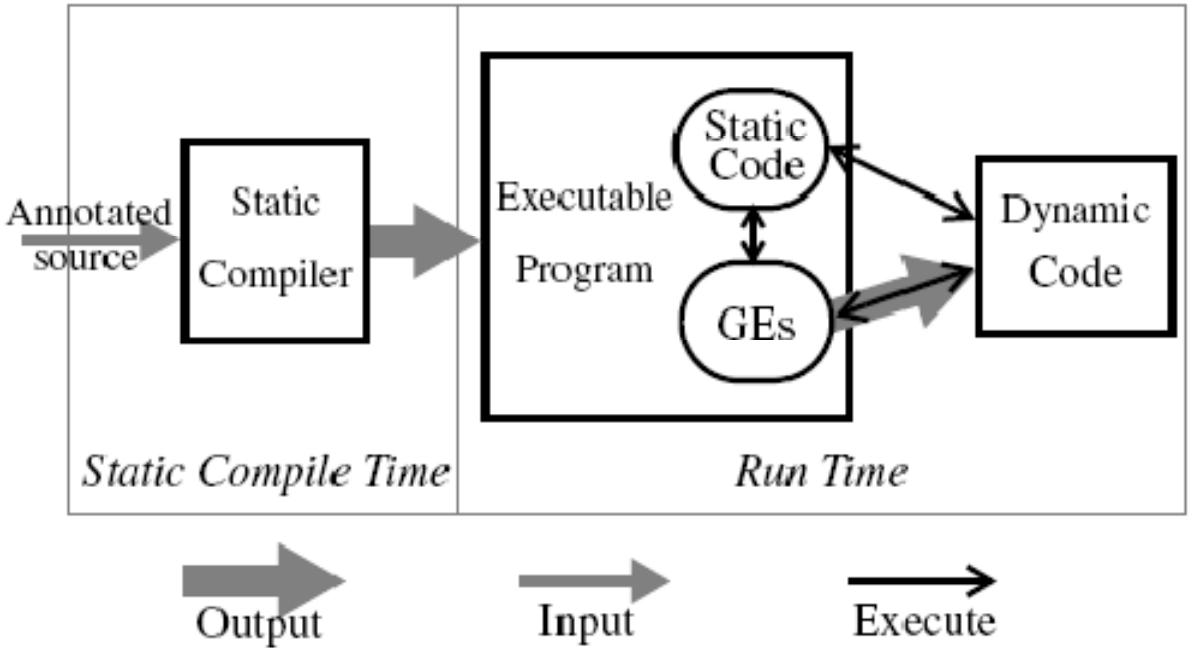
- The system generates code that checks the annotated values at runtime and regenerates code on the fly

- By using annotation, the system avoids over-checking and hence runtime overhead. However, this is at the cost of additional user overhead

DyC

Binding analysis examines all uses of static variables within scope

Dynamic compiler exploits invariance and specializes the code when invoked



DyC results

- Asymptotic speedups on a range of programs varies from 1.05 to 4.6
- Strongly depends on percentage of time spent in the dynamically compiled region. Varies from 9.9 to 100%
- Low overhead from 13 cycles to 823 cycles per instruction generated
- However relies on user intervention which ***may not be realistic*** in large applications
- Relies on user ***correctly annotating*** the code

Calpa for DyC

• Calpa is a system aimed at automatically identifying opportunities for specialization without user intervention.

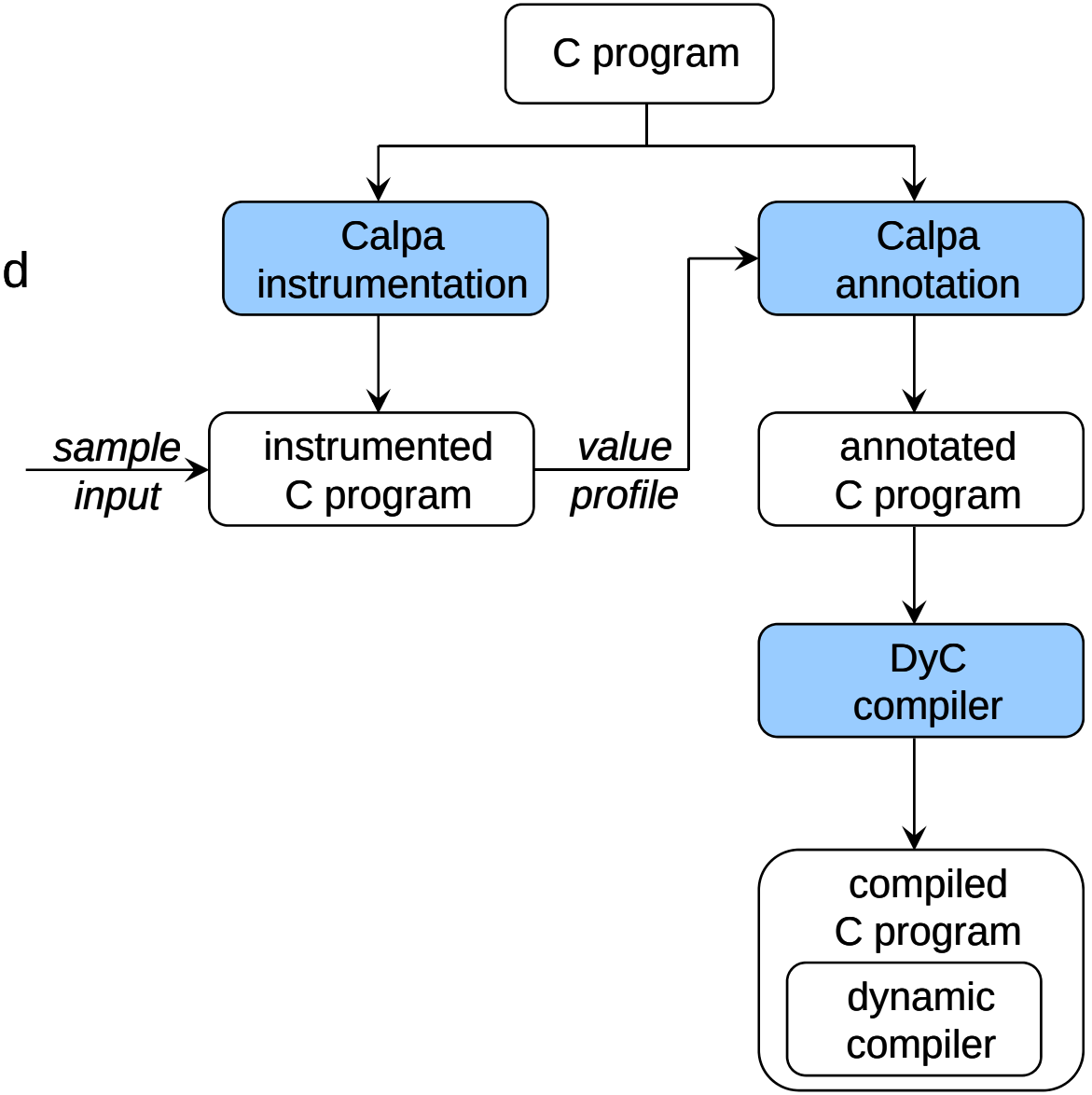
• It analyses the program for potential opportunities and determines the possible cost vs the potential benefit.

• For example, if a variable is multiplied by another variable which is known to be constant in a particular scope and if is equal to 0 or 1 then a cheaper code can be generated.

• If this variable is inside a deep loop then a quick test for 0 or 1 outside the loop will be profitable.

Calpa for DyC

- Calpa is a front-end to DyC
- It uses instrumentation to guide annotation insertion



Calpa for DyC

- Instruments code and sees how often variables change value. Given this data, Calpa determined the cost and benefit for a region of code.

- Number of different variants, cost of generating code, cache lookup. Main benefit determined by estimating new critical path.

- Explores all specialization up to a threshold. Widely different overheads 2 seconds to 8 hours. In two cases improves - from 6.6% to 22.6%.

- Calpa and DyC utilize selective dynamic code generation. Now look at fully dynamic schemes.

Dynamic binary translation

• The key idea is to take one ISA binary and translate it into another ISA binary at runtime.

• In fact this happens inside Intel processors where x86 is unpacked and translated into an internal RISC opcode which is then scheduled. The TransMeta Crusoe processor does the same. Same with IBM legacy ISAs.

• Why don't we do this statically? Many reasons!

• The source ISA is legacy but the processor internal ISA changes. It is impossible to determine statically what is the program. It is not legal to store a translation. It can be applied to a local ISA for long term optimizations without access to source codes.

DAISY

- One of the best known schemes came out of IBM headed by Kemal Ebcioglu.

- Aimed at translating PowerPC binaries to the IBM VLIW machine.

- Idea was to have a simple powerful in-order machine with a software layer handling complexities of PowerPC ISA.

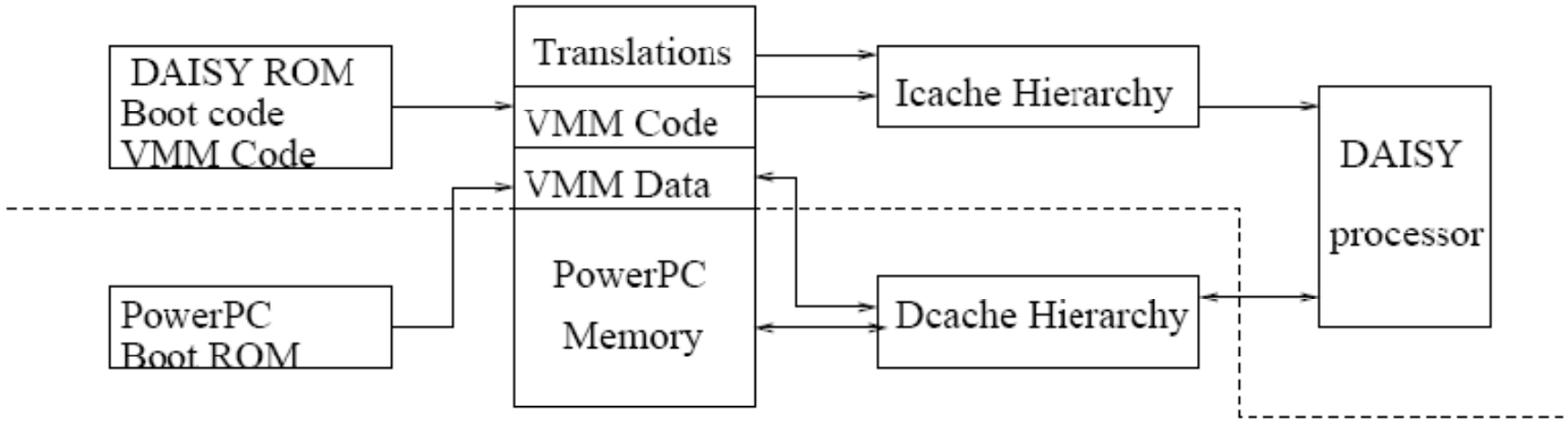
- Dynamic translation opens up opportunities for dynamic optimization.

- Concerned for industrial strength usage. Exceptions, self-modifying code etc…

DAISY

- At runtime, program path and data known. But need a low overhead scheme to make worthwhile.
- Specialization happens naturally as we know runtime value of variables.
- Can bias code generation to check for profitable cases.
- DAISY uses a code cache of recently translated code segment.
- Automatic superblock formation and scheduling.

DAISY structure

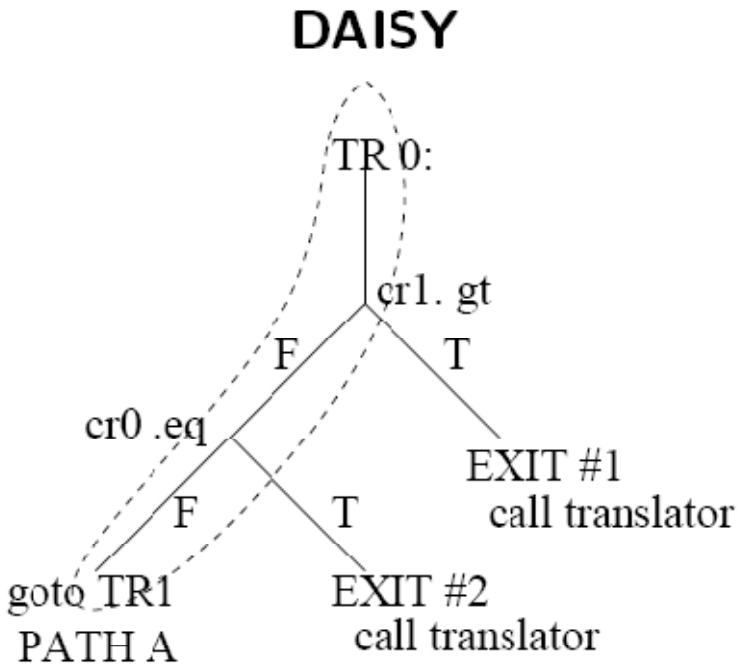


- Power PC code runs without modification.
- DAISY specific additions separated by dotted line.
- Initially interpret PowerPC instructions and then compile after hitting threshold.
- Then schedule and save instruction in cache (2-4k). Untaken branches are translated as (unused) calls to the binary translator.

DAISY example

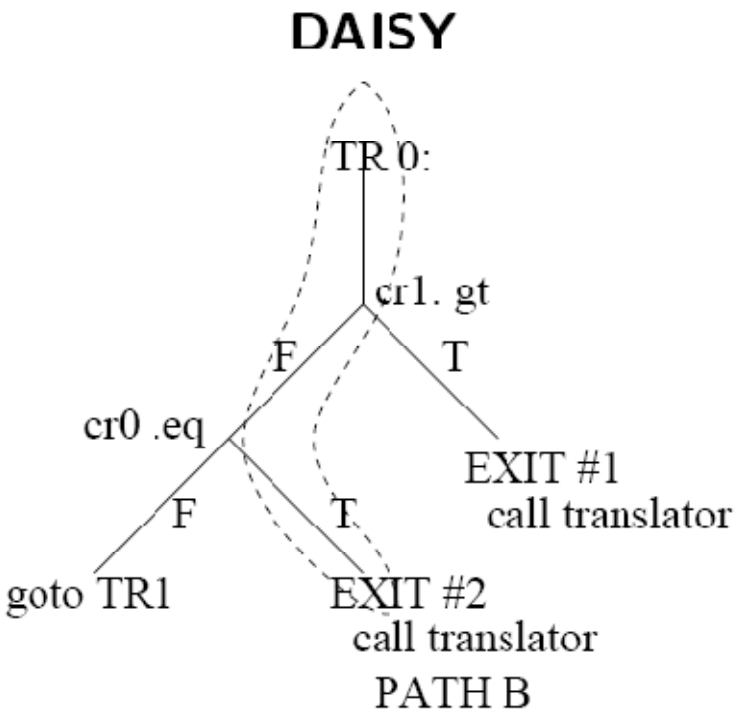
- Here the group is expanded to contain two conditionals

- Path A is encountered and translated



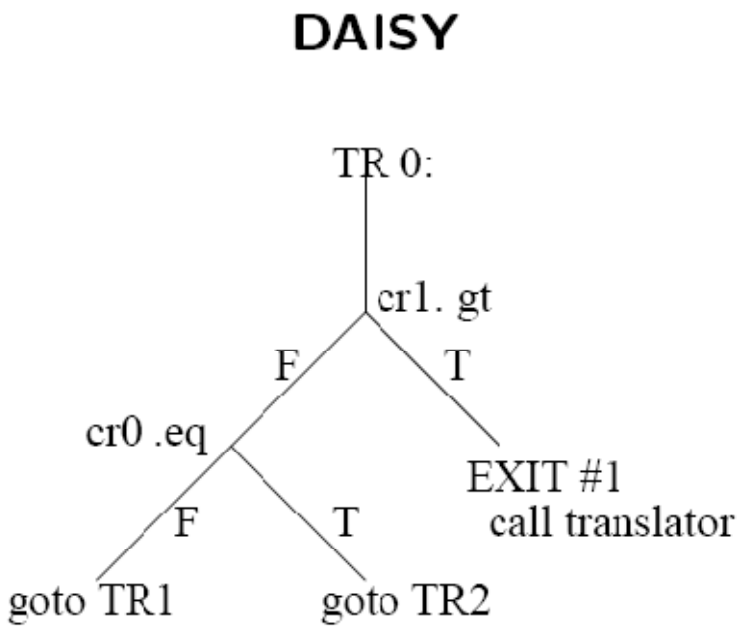
DAISY example

- When Path B is encountered for the first time
- Translator is called



DAISY example

- Code in cache is now updated
- Paths A and B require no further translation
- One untranslated path remaining
- Only translate and store code if needed



DYNAMO

• Similar to DAISY though focuses on binary to binary optimizations on the same ISA. One of the claims is that it allows compilation with -01 but overtime provides -03 performance.

• Catches dynamic cross module optimization opportunities missed by the static compiler. Code layout optimization allowing improved scheduling due to bigger segments. Branch alignment and partial procedural inlining form part of the optimizations

• Aimed as a way to improve performance from a shipped binary over time

• Unlike DAISY, have to use existing hardware - no additional fragment cache available

DYNAMO

- Initially interprets code. This is very fast as the code is native. When a branch is encountered check if already translated

- If it has been translated jump and context switch to the fragment cache code and execute. Otherwise if hot, translate and put in cache

- Over time the working set forms in the cache and Dynamo overhead reduces - less than 1.5

- Cheap profiling, predictability

- Linear code structure in cache makes optimization cheap. Standard redundancy elimination applied

Just in Time Compilation

• Key idea: lazy compilation. Defer compiling a section of high level code until it is encountered during program execution. For OO programs it has been shown that this greatly reduces the amount of code to compile. Krintz'00 shows 14 to 26% reduction in total time.

• Greater knowledge of runtime context allowing optimization to be focused on important parts of a program.

• However is Just in time really Just too late? Why wait until execution time to compile when the code may be lying around on disk for months beforehand?

• Main reason - dynamic linking of code especially in Java. This restricts the optimizations available.

Jikes

- Most Java compilers initially interpret, then compile and finally optimize based on frequency of use

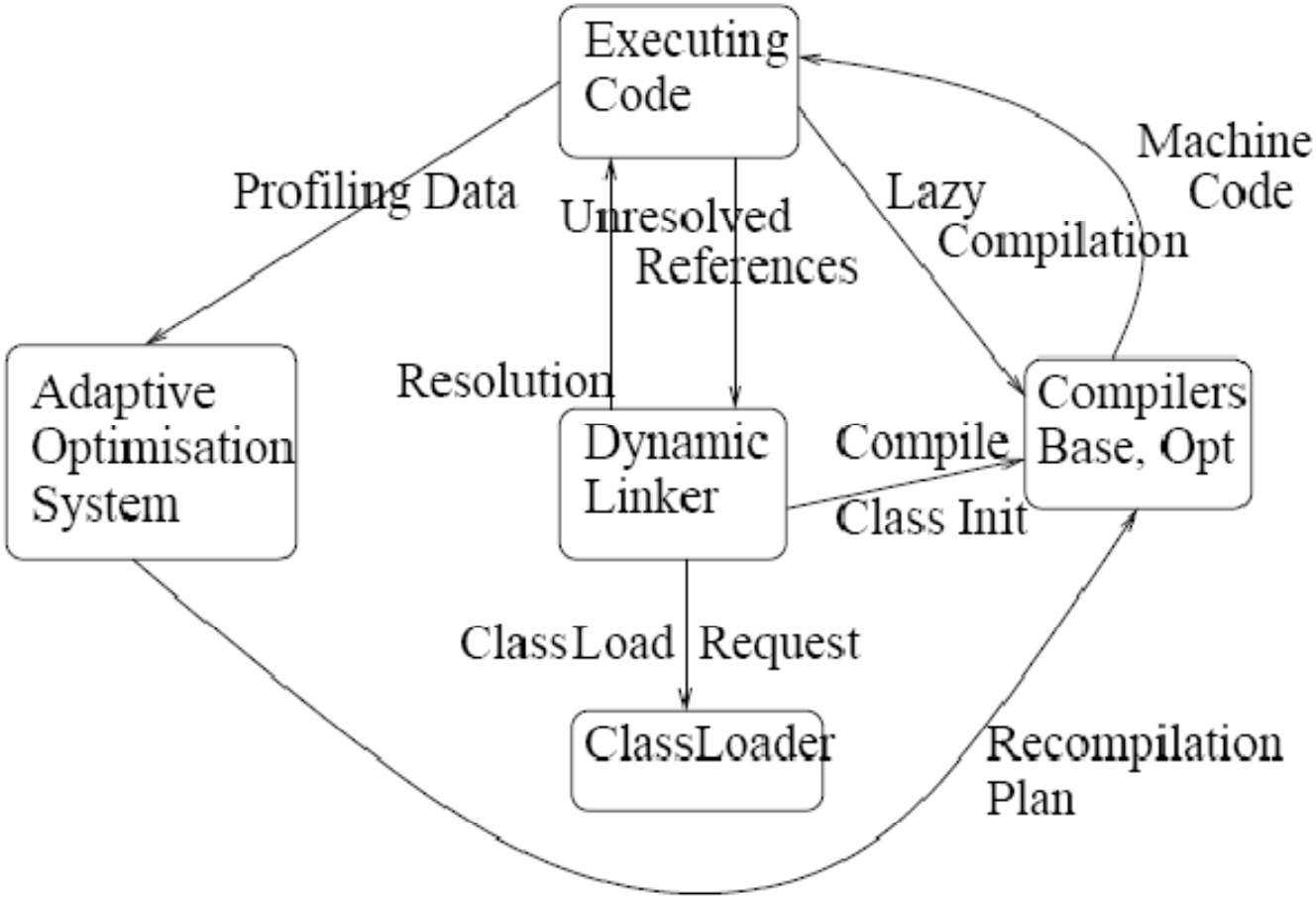
- Normally done on a per method basis

- Jikes instead directly compiles code when encountered to native machine code.

- Well known robust research compiler freely available.

- Much work centered around what level of optimization to apply and when to apply it.

Jikes structure

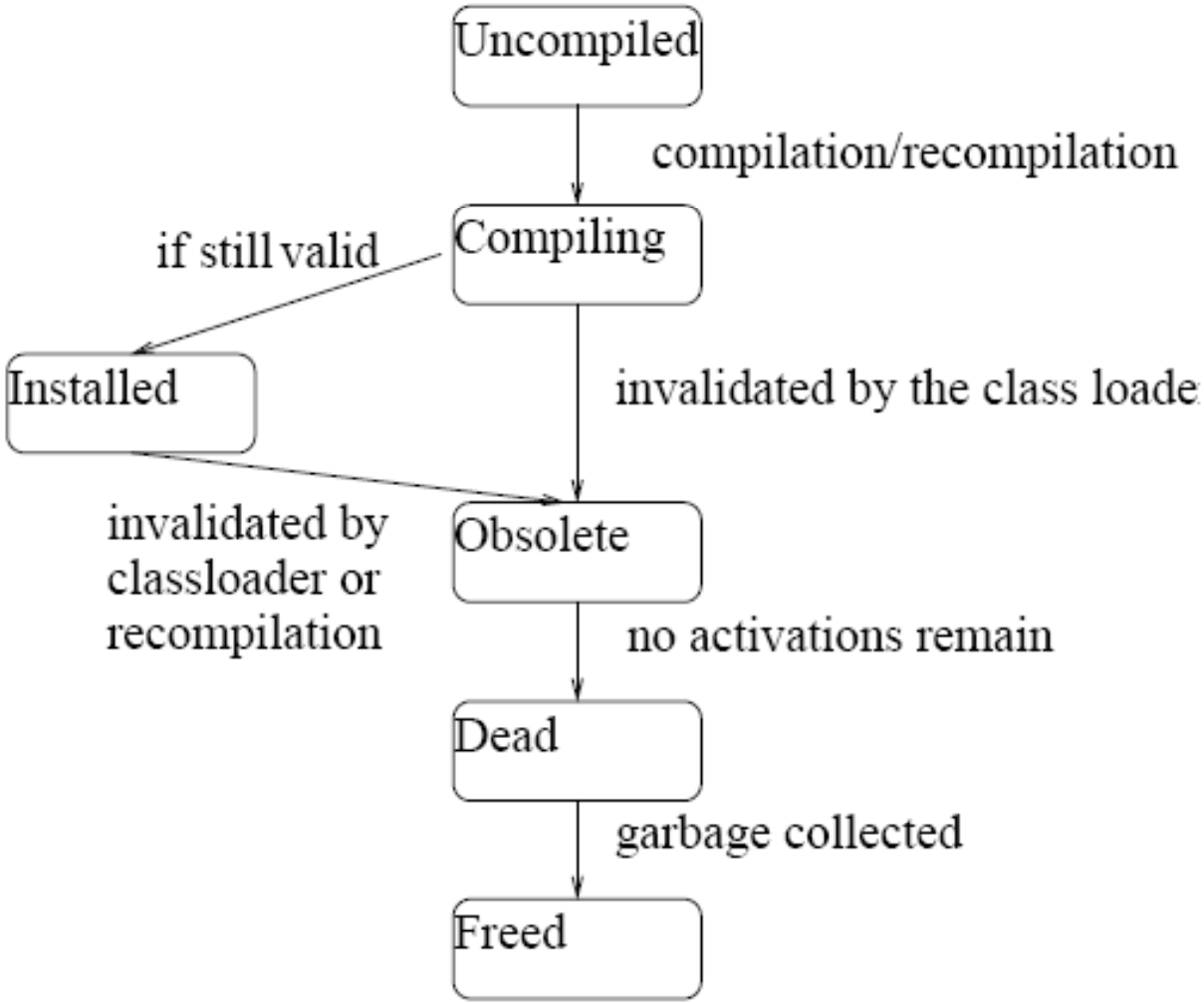


Jikes example

```
iload x                INT_ADD tint,xint,5          INT_ADD yint,xint,5
iconst 5               INT_MOVE yint,tint
iadd
istore y
```

- Simple example showing translation of byte code into native code
- Simple optimizations to remove redundant temporaries have a significant impact on later virtual to register mapping phases
- First version corresponds to baseline compiler, second to most basic optimizing compilation

Method life cycle



Jikes optimizations

- Jikes makes use of multiple optimization levels and uses these to carefully trade cost vs gain.

- Baseline translates directly into native code simulating operand stack. No IR, no register allocation. Slightly faster code than interpretation.

- Optimizing compiler. Translate into an IR with linear register allocation. 3 further optimization levels:

    - Level 0: Effective and cheap optimizations. Simple scalar optimizations and inlining trivial methods. All tend to reduce size of IR

    - Level 1: as 0 but with more aggressive speculative inlining. Multiple passes of level 0 opts and some code reorganizing algorithms.

    - Level 2: employs simple loop optimizations. Normalization and unrolling. SSA based flow-sensitive algorithms also employed.

Jikes optimizations

Compiler	Bytecodes/millisecond	Speed
Baseline	377.8	1.0
Level 0	9.29	4.26
Level 1	5.69	6.07
Level 2	1.81	6.61

- Only worthwhile compiling at a higher level if benefit outweighs cost.

- Adaptive algorithm compares cost of code under current level vs an increased level.

- Crucially depends on anticipated future profile which is unavailable. Solution - just guess - currently assume twice as long as now!

Jikes optimizations

- Krintz evaluates the adaptive approach

Compiler	Total time	Compile time
Baseline	**29.24**	**0.44**
Opt	**9.98**	**0.46**
Adapt	**8.97**	**0.48**

- Figures are time in seconds for SPECjvm98

- Total time is better for Adapt even though it has increased compile-time.

- Conclusion: ***knowing hotspots really helps optimization***

JIT conclusions

- JITs suffer from having the necessary info too late. Need to anticipate optimization opportunities.

- Many different optimization scenarios available. Adaptive option increases level of optimization when it recompiles increasingly used hotspots.

- As compile-time is part of runtime, important to find a trade-off between two.

ADAPT

• ADAPT is a mixed approach to optimization that combines static and iterative compilation in an on-line manner.

• Basically at runtime different options of a code section are run concurrently and the best-one selected. This is done in parallel on remote servers.

• Really trading space for time making an on-line technique viable as an on-line technique as long as sufficient space available.

Procs:

• Enables adaptive online iterative compilation.

Cons:

• Very complex recompilation framework.

• Only works for scientific programs with relatively static behavior.

• Uses multiple machines for evaluation which is not always practical.

Summary

- All schemes allow specialization at runtime to program and data.
- Staged schemes such as DyC are more powerful as they only incur runtime overhead for specialization regions.
- JIT and DBT delay everything to runtime leaving little optimization opportunities.
- All except ADAPT have a hardwired heuristic of what the best strategy is.
- Poor at adapting to new platforms.
- Apart from ADAPT, none looked at processor specific optimization. Mainly looked at architecture independent optimizations or standard backend scheduling or register allocation.
- Like PDC only used the data really for limited optimization goals rather than overcoming undecidability or processor behavior.
- None of the techniques would adapt their compilation approach in the light of experience.

Combine static and dynamic optimizations?

Static multi-versioning for different constraints (optimization cases) and run-time adaptation:

• Grigori Fursin, Albert Cohen, Michael O'Boyle and Olivier Temam. A Practical Method For **Quickly Evaluating Program Optimizations**. *Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, number 3793 in LNCS, pages 29-46, Barcelona, Spain, November 2005

Integration of the run-time adaptation into mainline GCC:

• Grigori Fursin, Cupertino Miranda, Sebastian Pop, Albert Cohen and Olivier Temam. **Practical run-time adaptation with procedure cloning to enable continuous collective compilation.** *GCC Developers' Summit*. Ottawa, Canada, July 2007

Adaptation for heterogeneous systems (CELL and GPU systems)

• *Victor Jimenez, Isaac Gelado, Lluis Vilanova, Marisa Gil, Grigori Fursin and Nacho Navarro.* ***Predictive runtime code scheduling for heterogeneous architectures.*** *To appear at the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009), Paphos, Cyprus, January 2009*.

Collaboration with IBM, UPC, STMicro

Run-time adaptation using procedure cloning

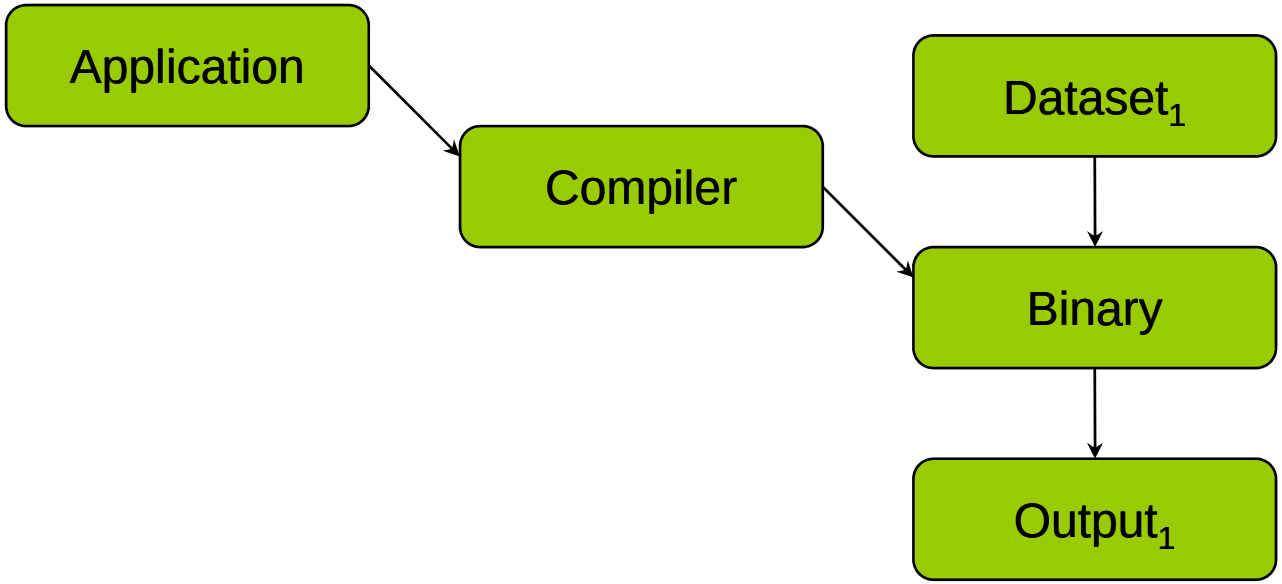
Any other ways to solve previous and the following problems?

- 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!

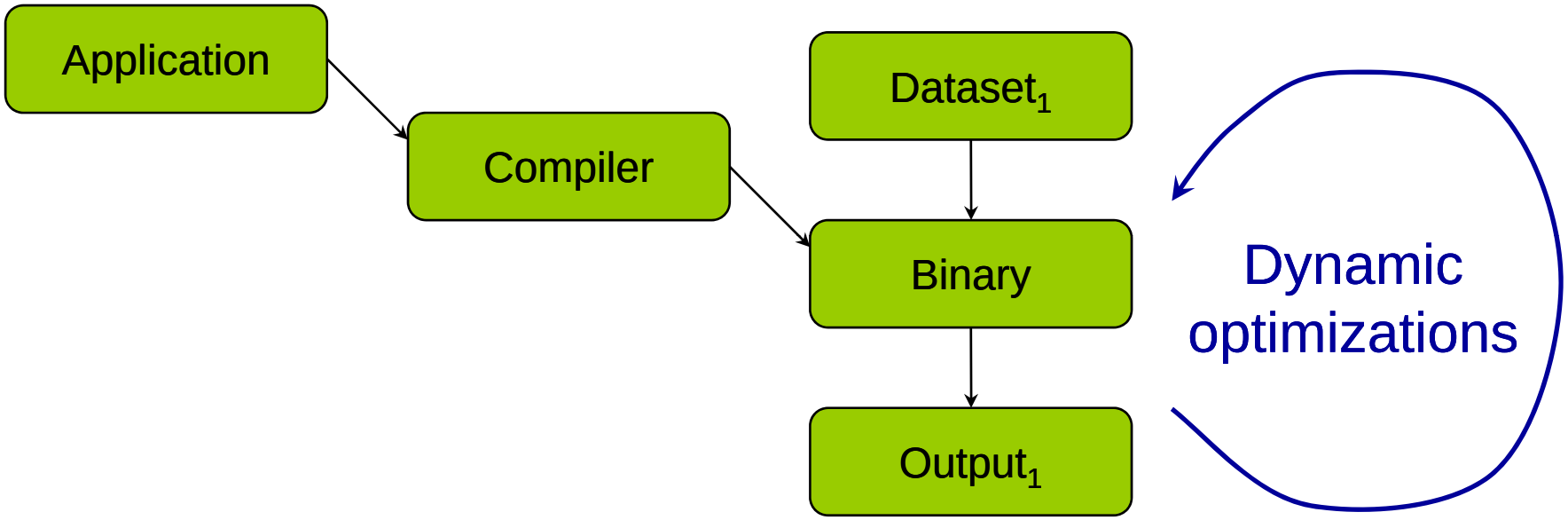
Current methods

Some existing solutions:



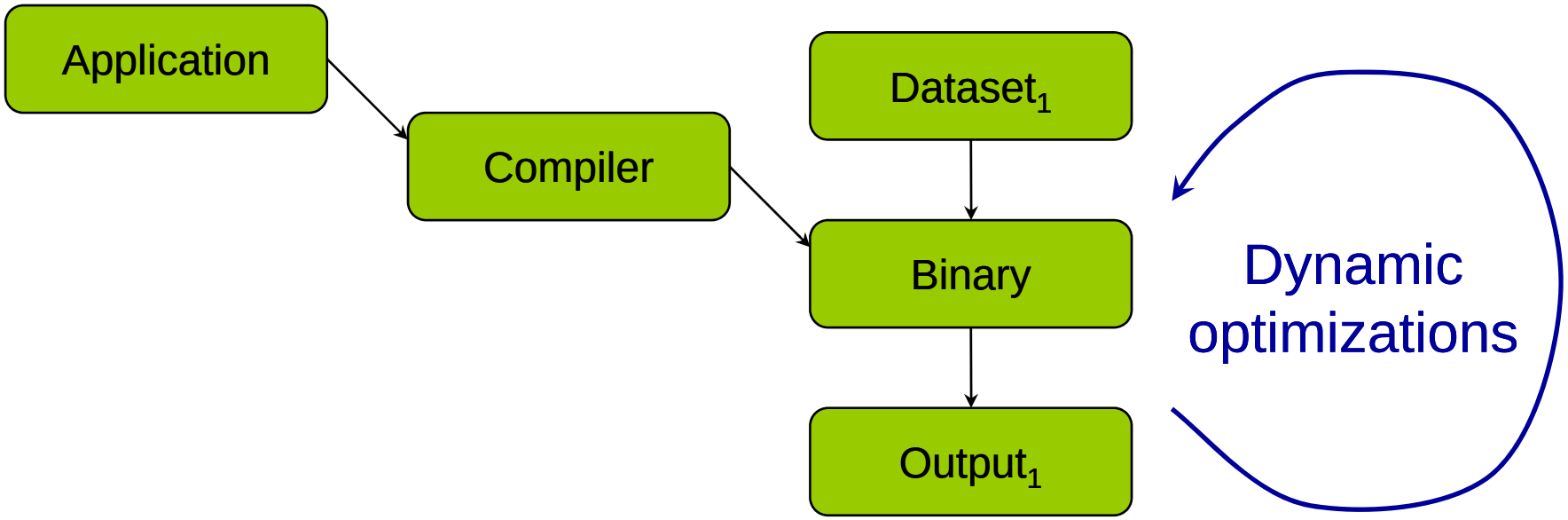
Current methods

Some existing solutions:



Current methods

Some existing solutions:

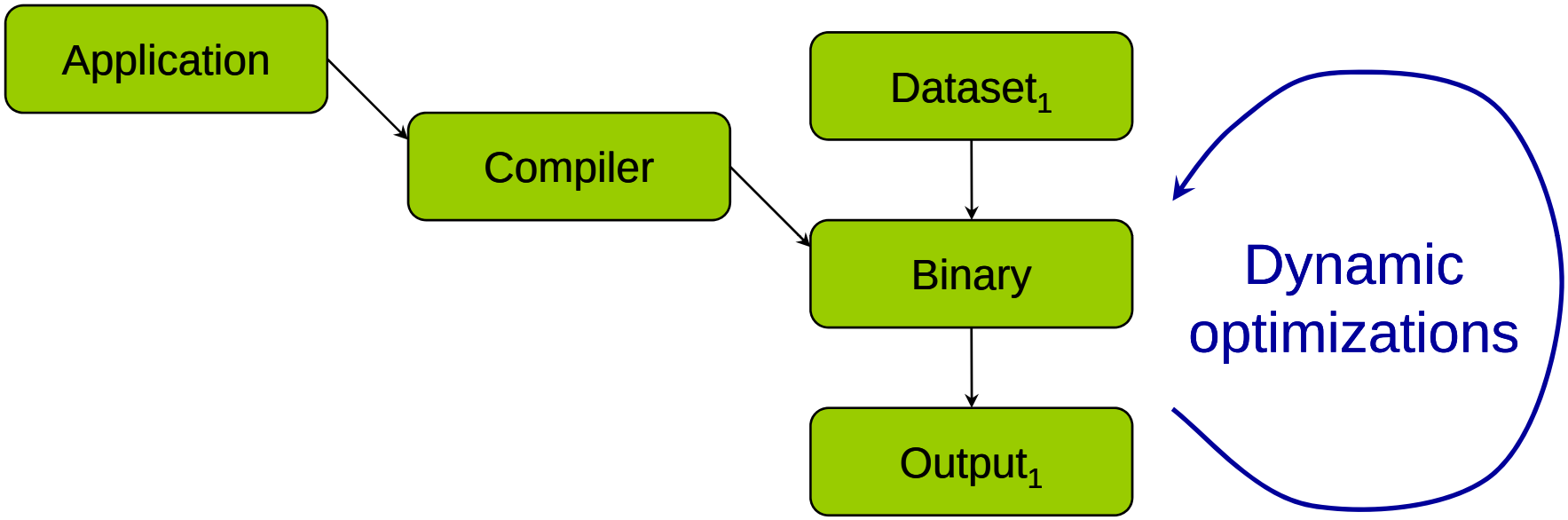


Pros: *run-time information,*

potentially more than one dataset

Current methods

Some existing solutions:



Pros: *run-time information,*

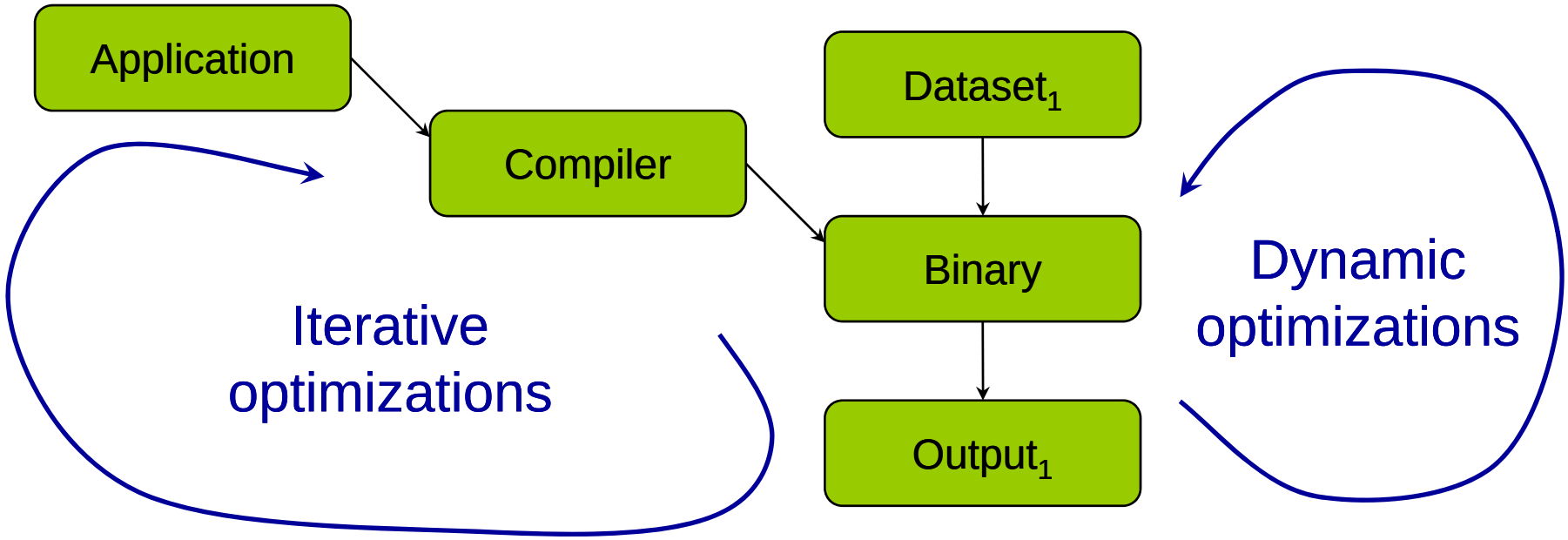
potentially more than one dataset

Cons: *restrictions on optimization time,*

simple optimizations

Current methods

Some existing solutions:



Pros: *run-time information,*

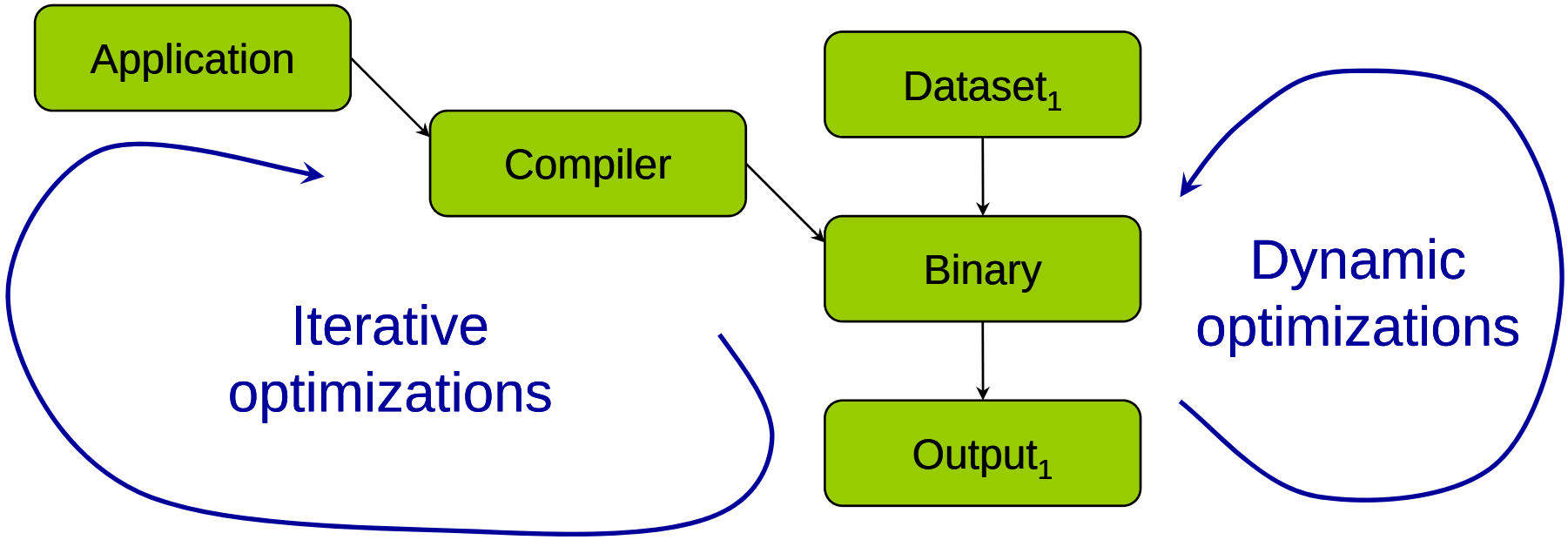
potentially more than one dataset

Cons: *restrictions on optimization time,*

simple optimizations

Current methods

Some existing solutions:



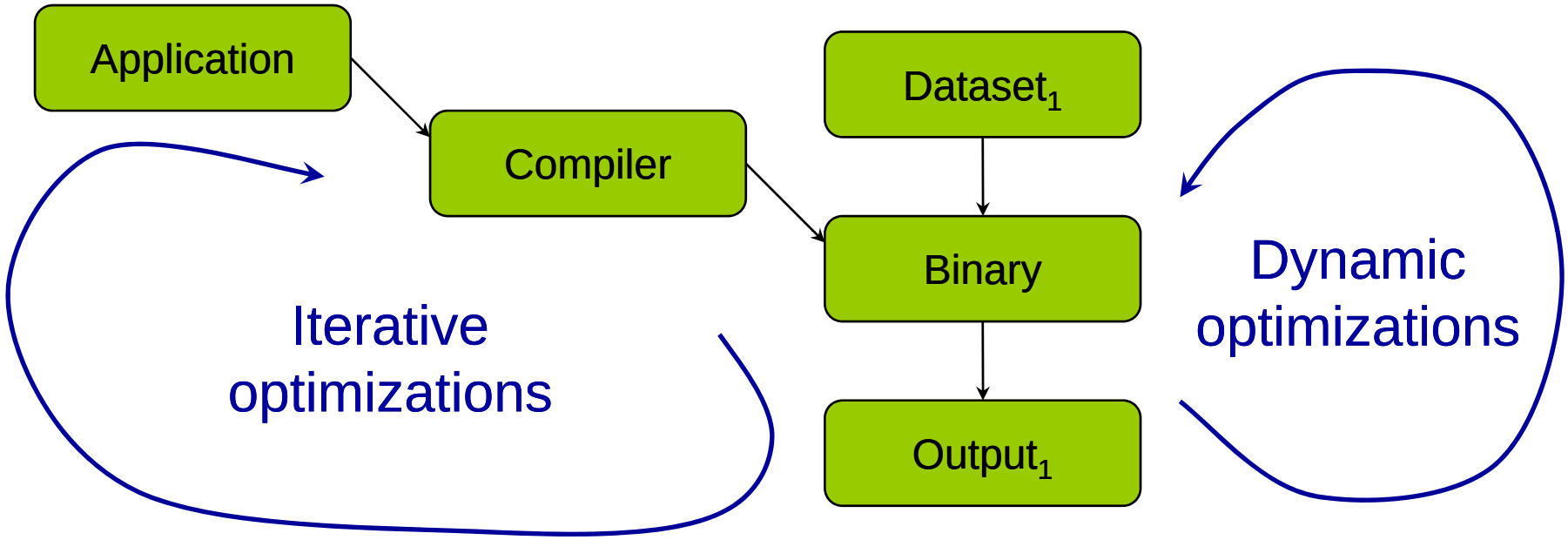
Pros: *powerful transformation space exploration*

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*

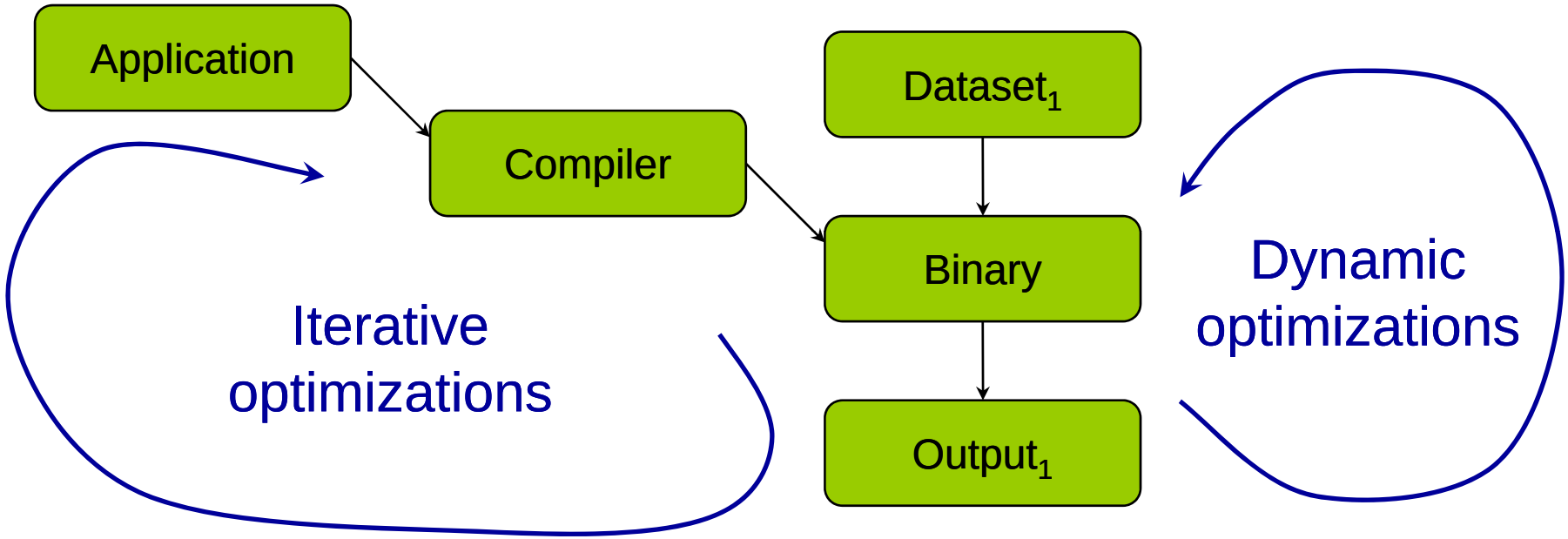
Cons: *slow, one dataset*

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

Run-time program behavior

Idea to enable easy static and dynamic optimizations:

- Most time during execution is spent in procedures/functions or loops
- Clone these sections and apply different transformations statically
- At run-time add run-time behavior analyzer routines and detect regular behavior
- Select appropriate code sections depending on run-time behavior of programs (code sections)
- Continuously recompile program with high-level transformations

Run-time program behavior

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

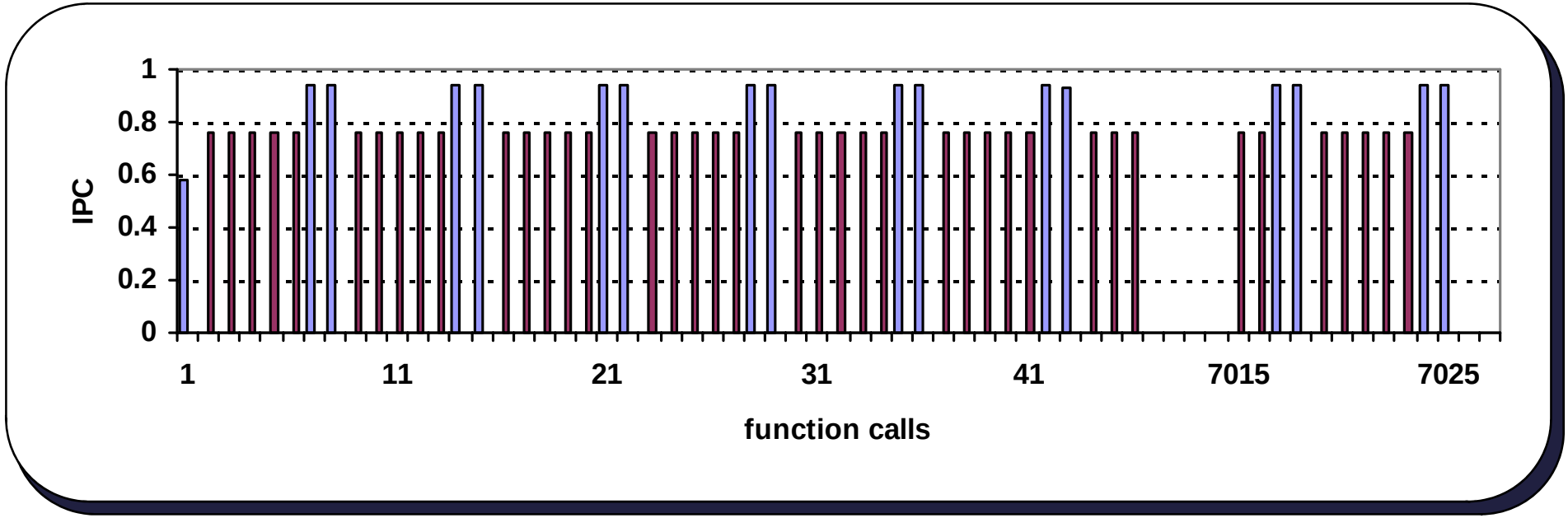
typically functions or loops

Run-time program behavior

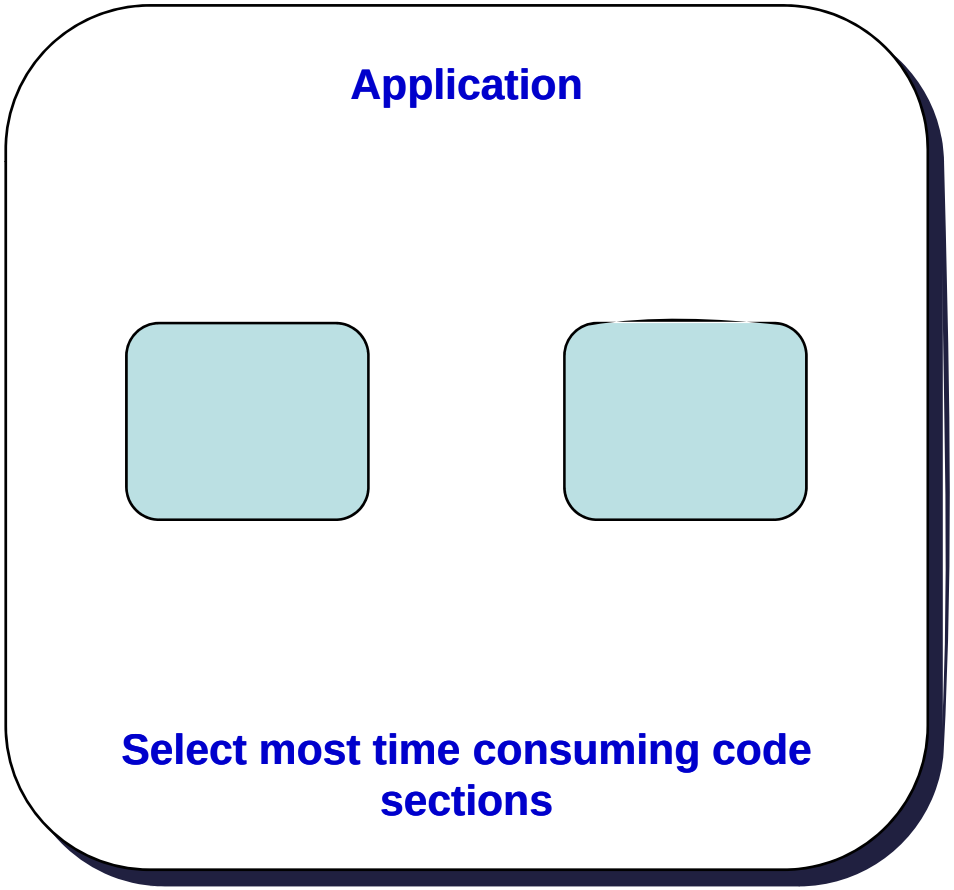
Repeatedly executed time-consuming parts of the code that allow powerful transformations:

typically functions or loops

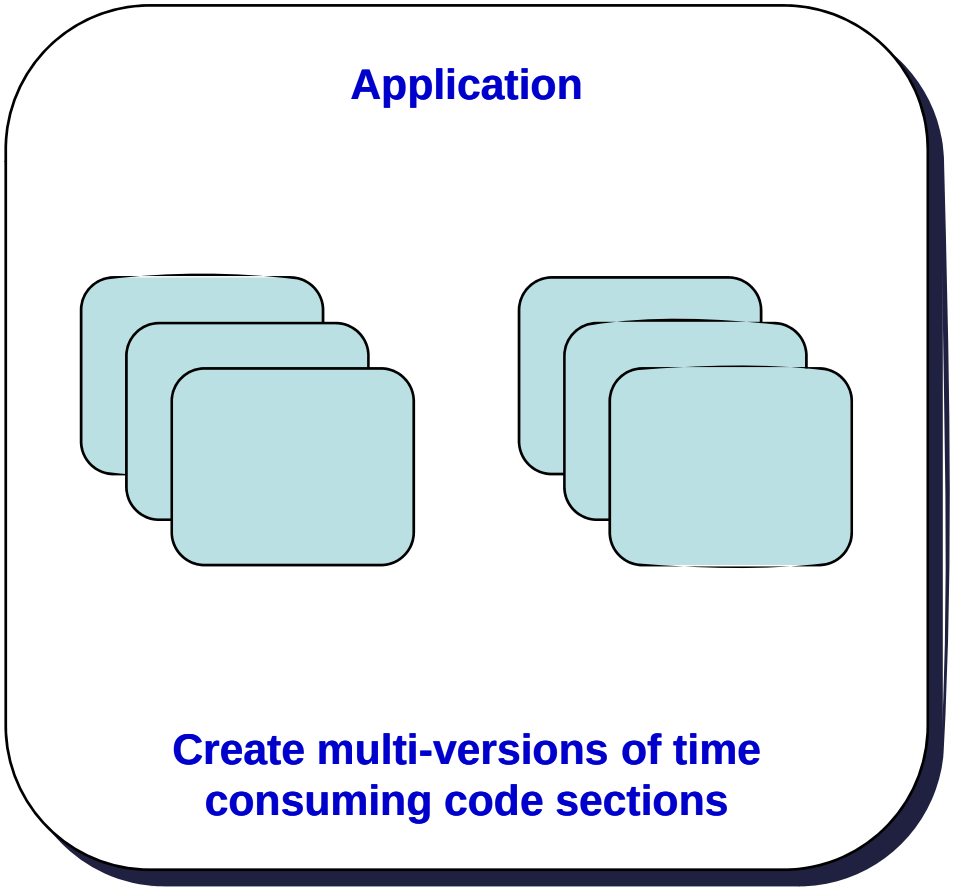
IPC for subroutine resid of benchmark mgrid across calls



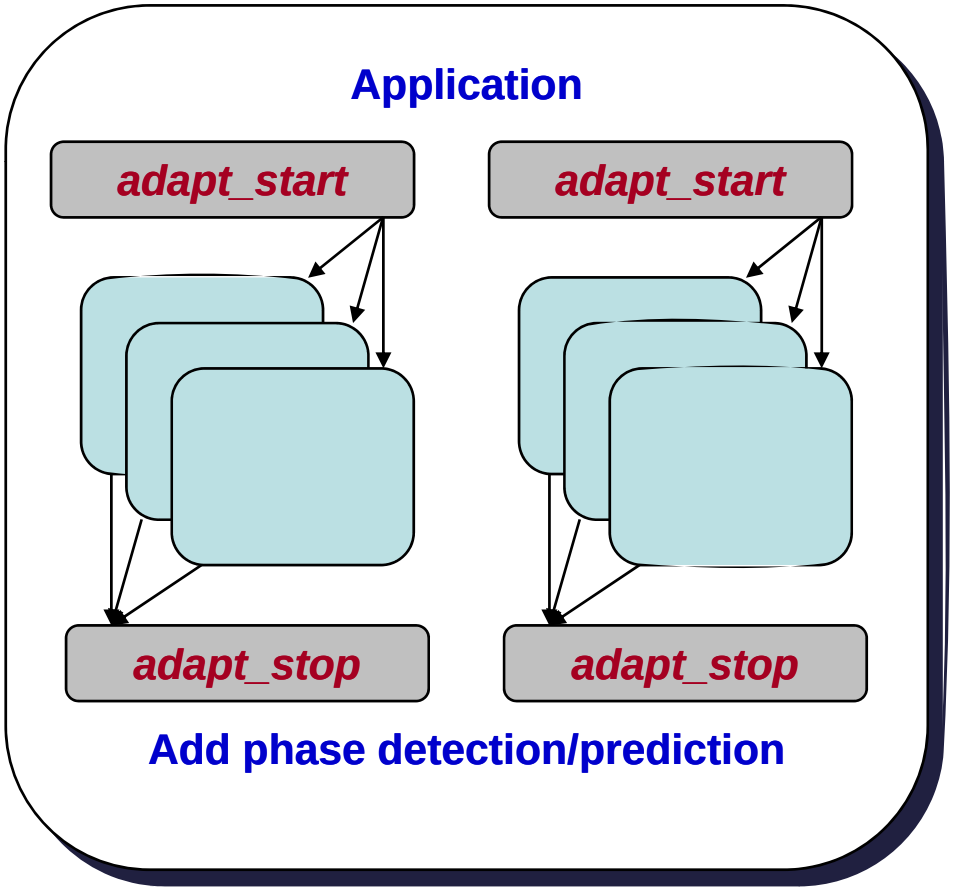
Our approach: static multiversioning



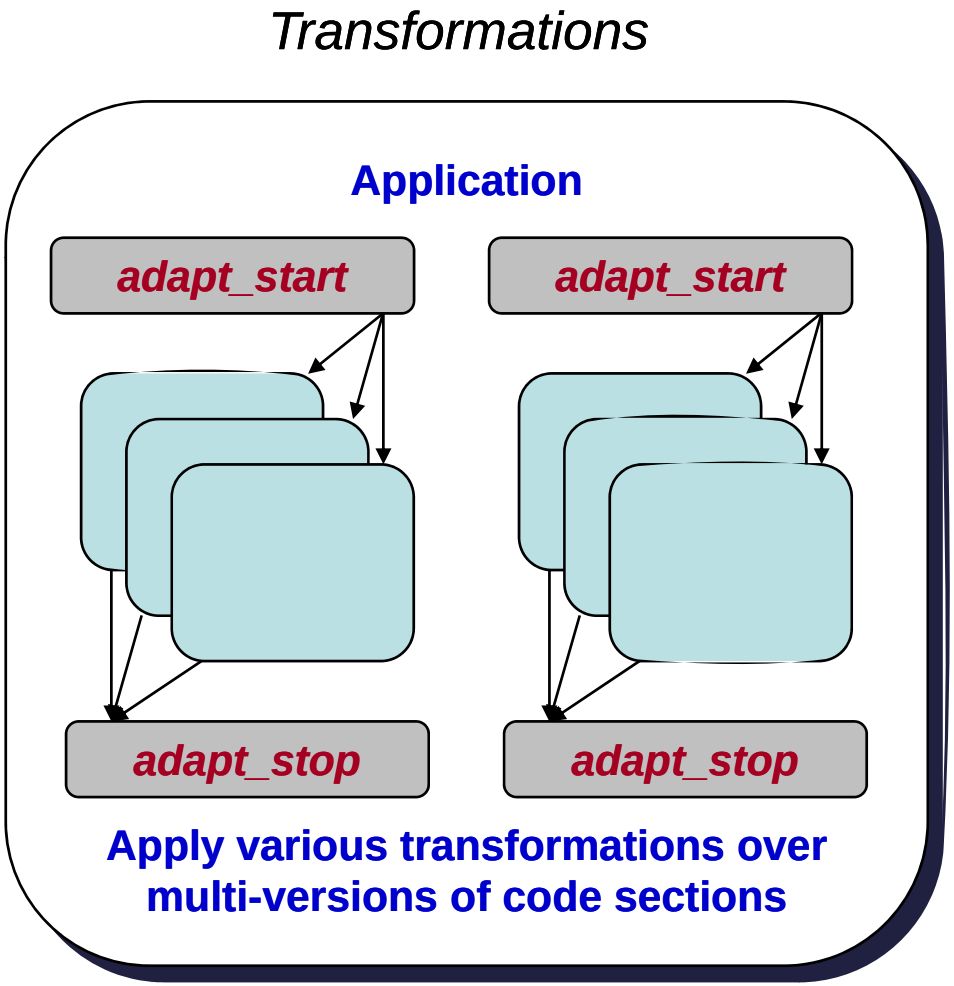
Our approach: static multiversioning



Our approach: static multiversioning

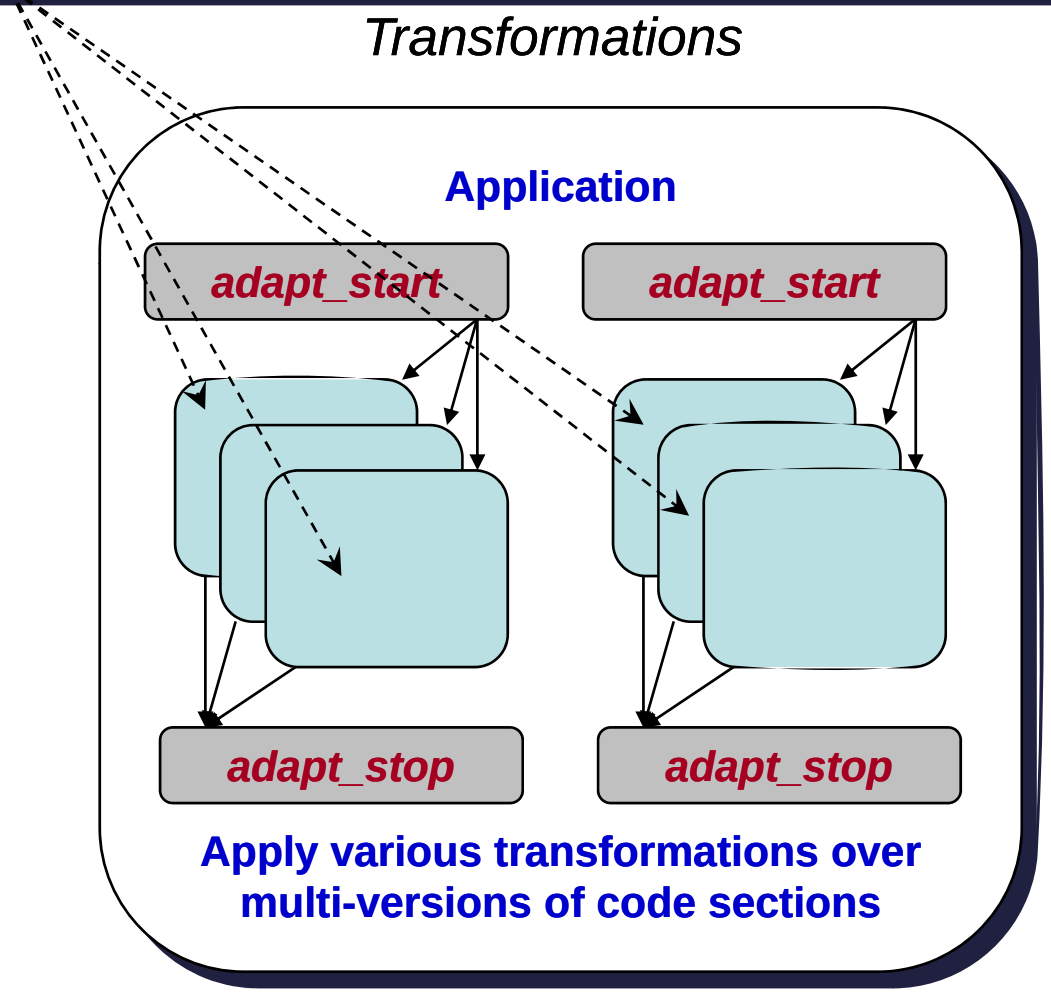


Our approach: static multiversioning

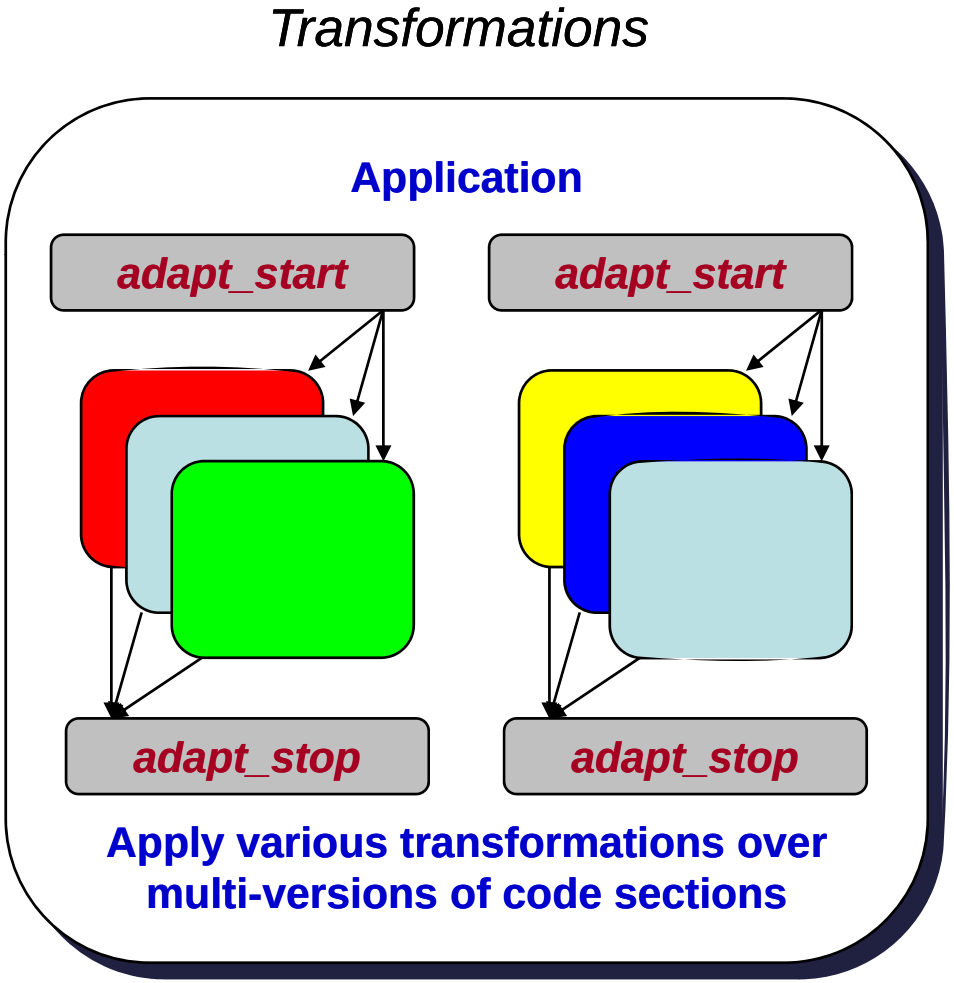


Our approach: static multiversioning

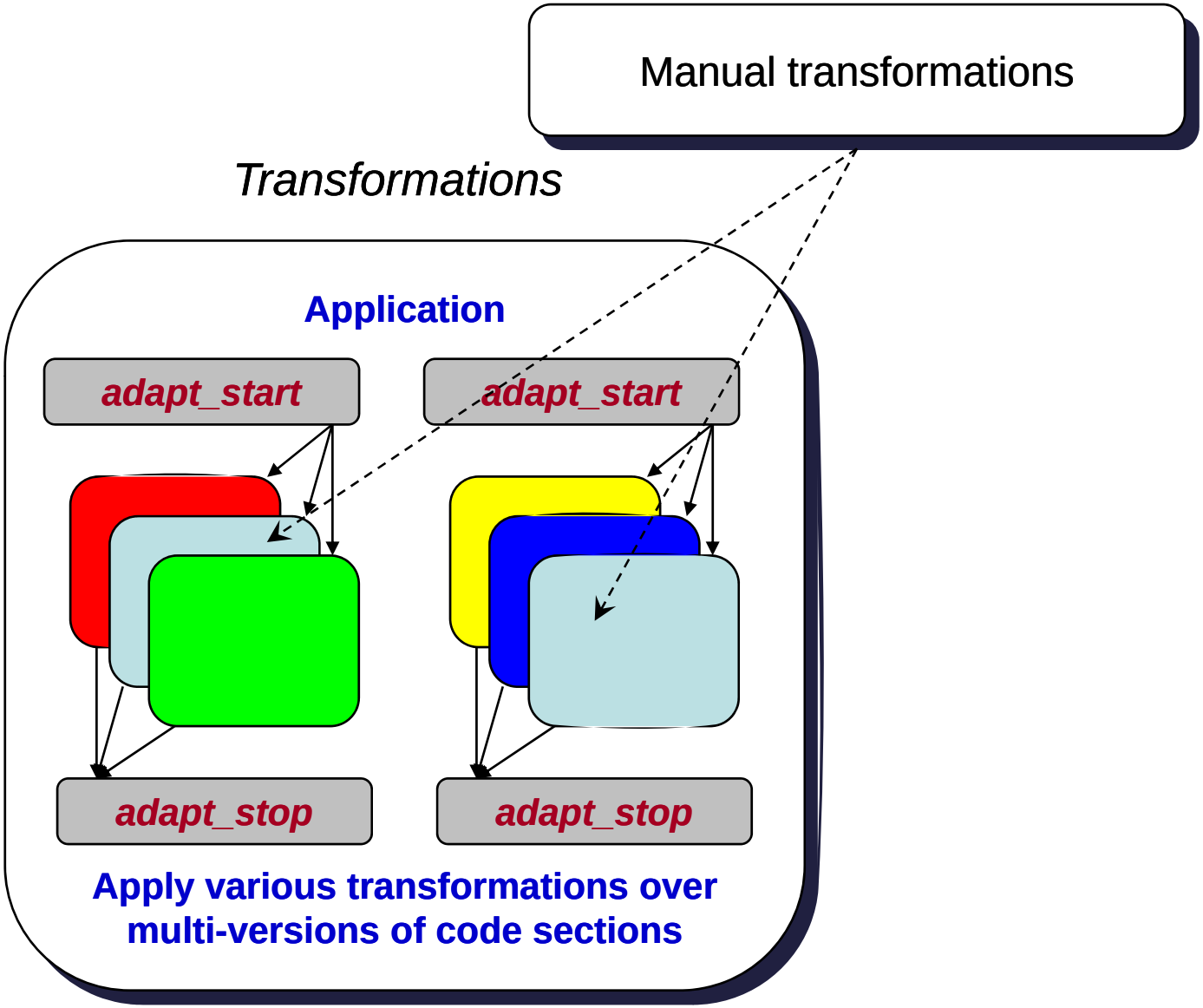
Fine-grain internal compiler (PathScale, Open64, ORC, GCC) transformations using Interactive Compilation Interface (ICI)



Our approach: static multiversioning

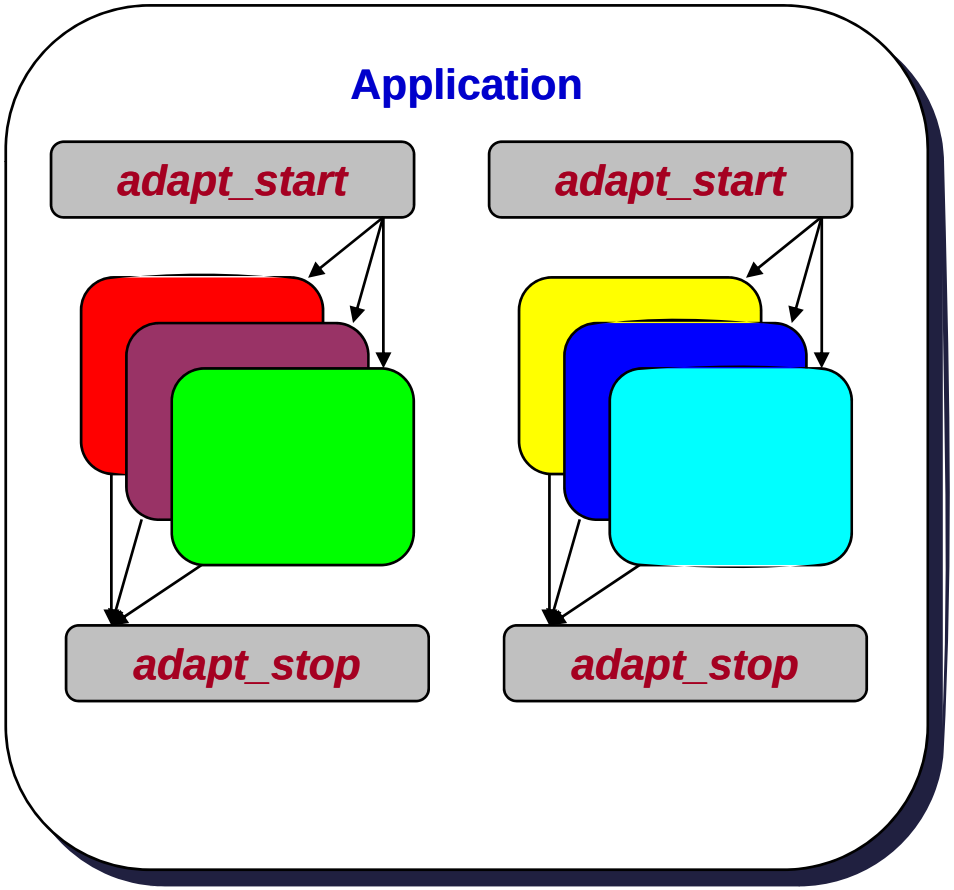


Our approach: static multiversioning



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

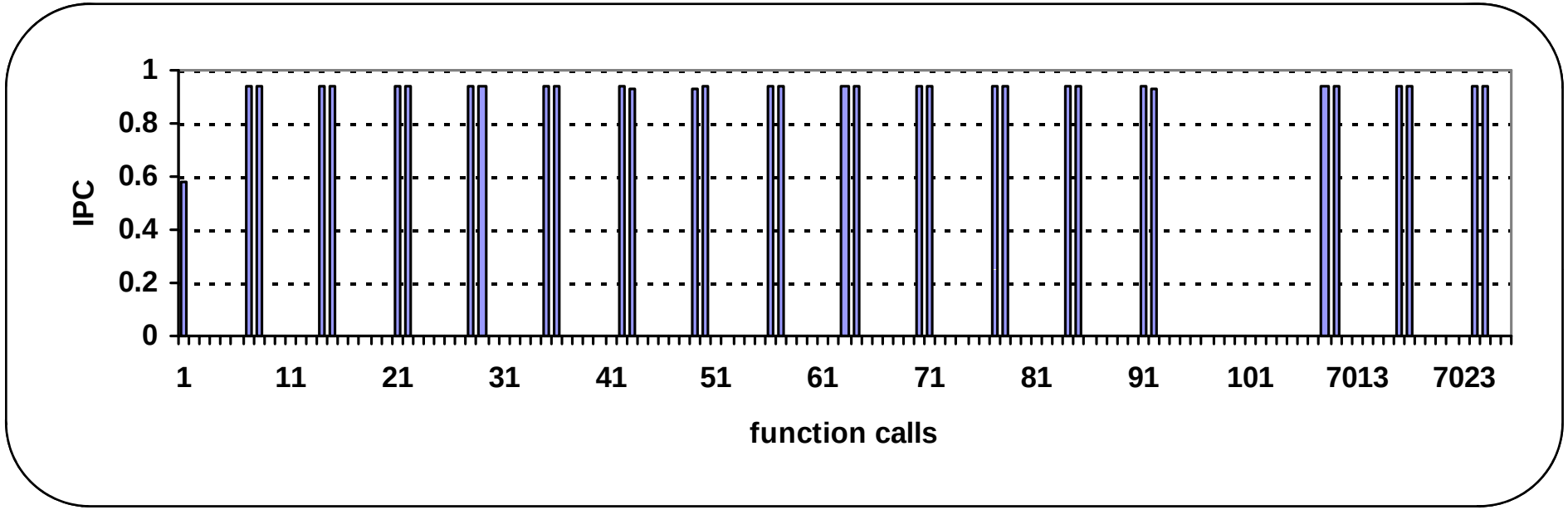
Depends on program behaviour

Programs with regular behavior

Programs with irregular behavior

Adaptation for regular behaviour

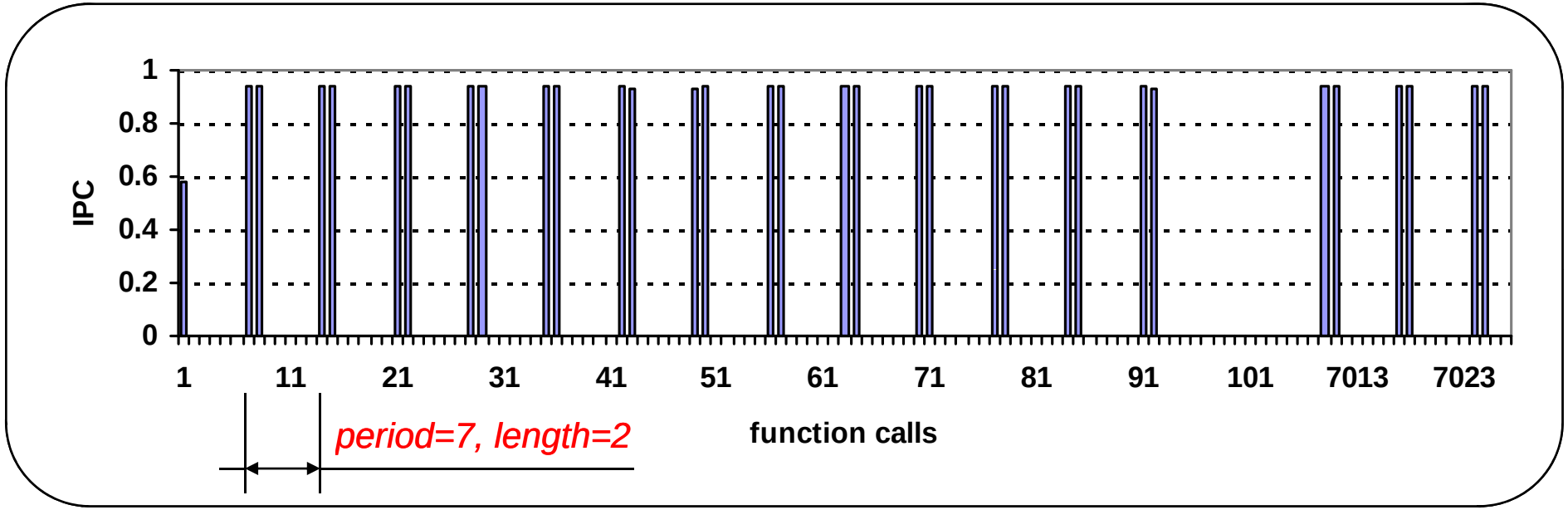
IPC for subroutine resid of benchmark mgrid across calls



- 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

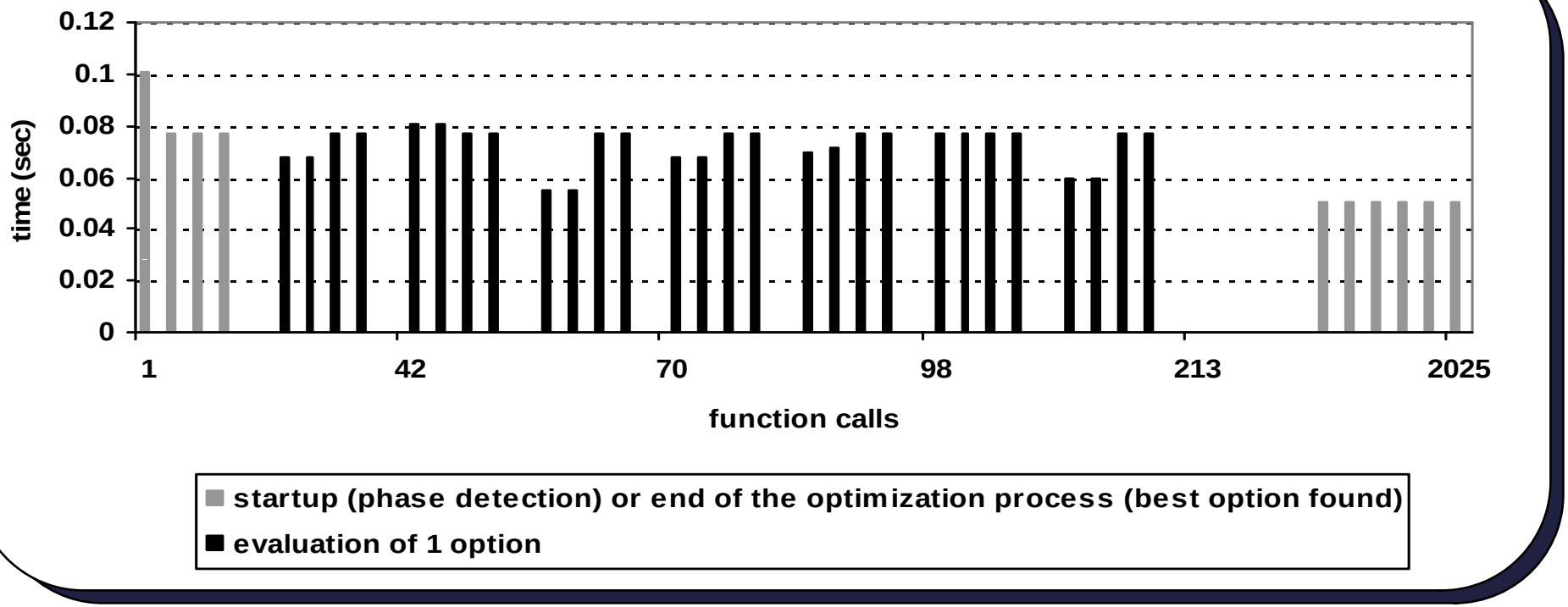
IPC for subroutine resid of benchmark mgrid across calls



- 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

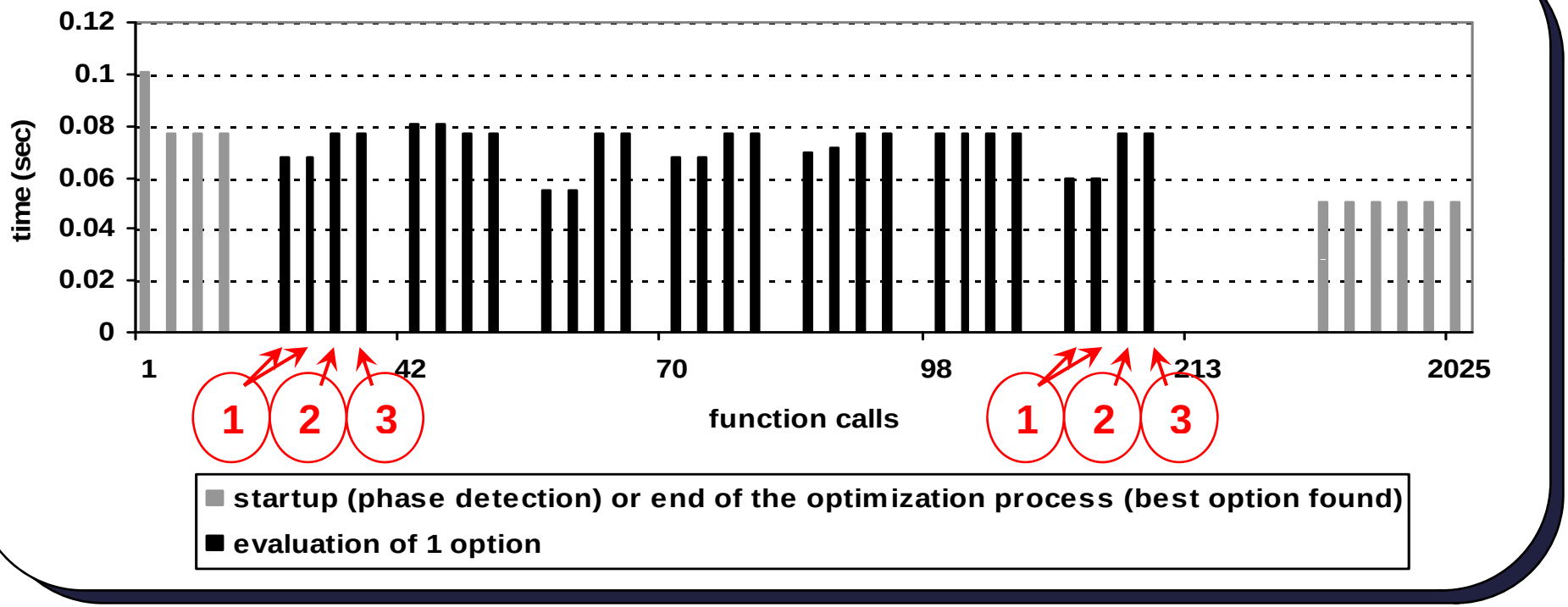
Execution times for subroutine resid of benchmark mgrid across calls



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)

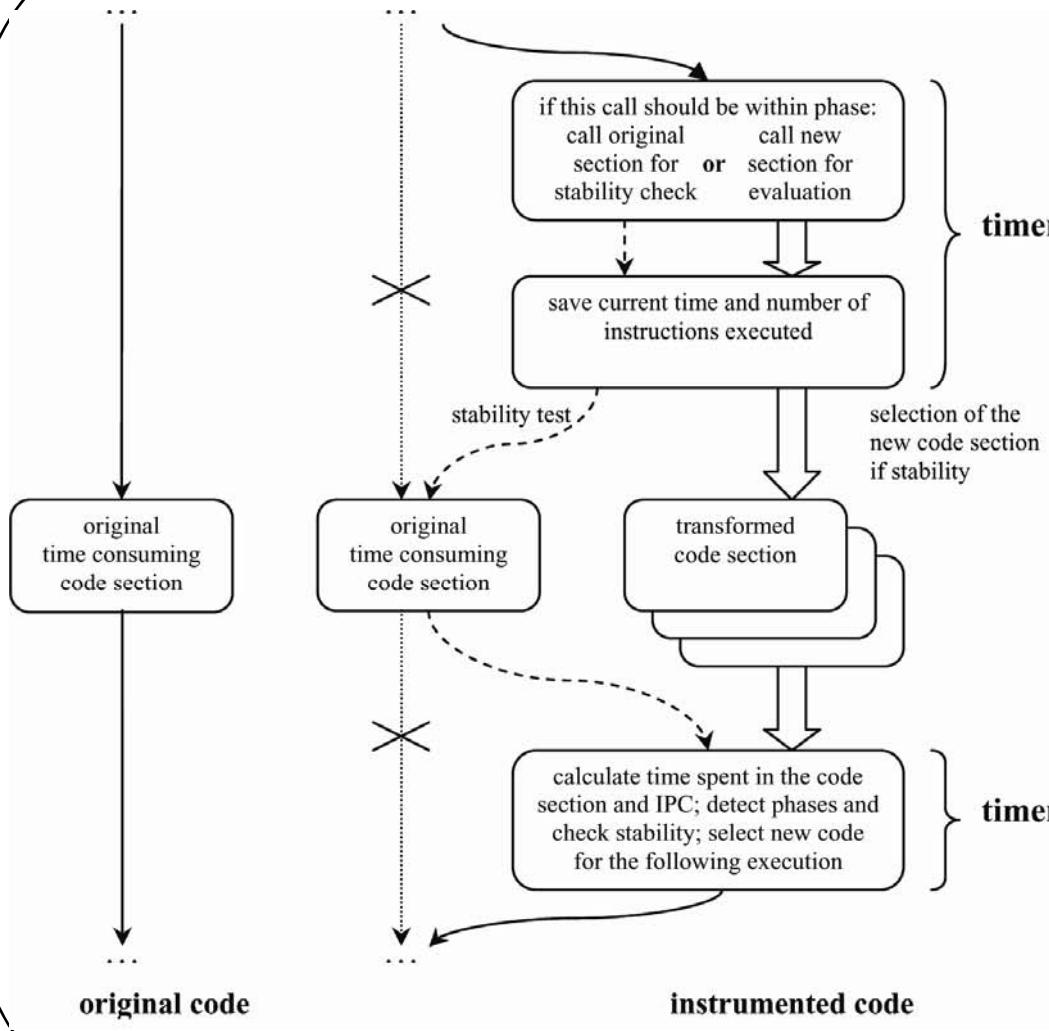
Adaptation for regular behaviour

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

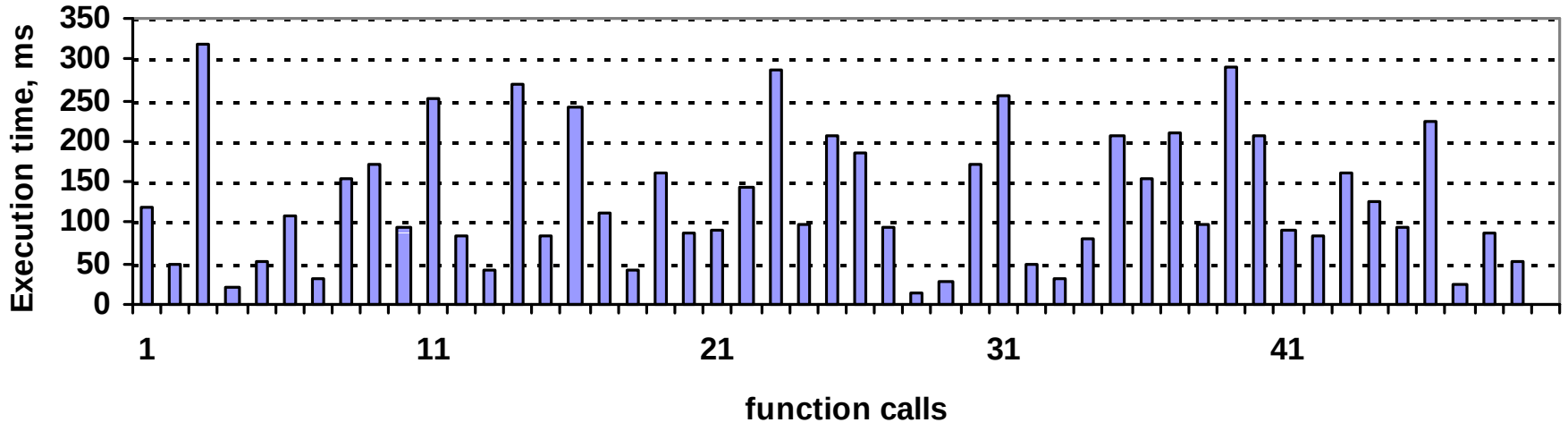
PDPT (Phase Detection and Prediction Table)

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

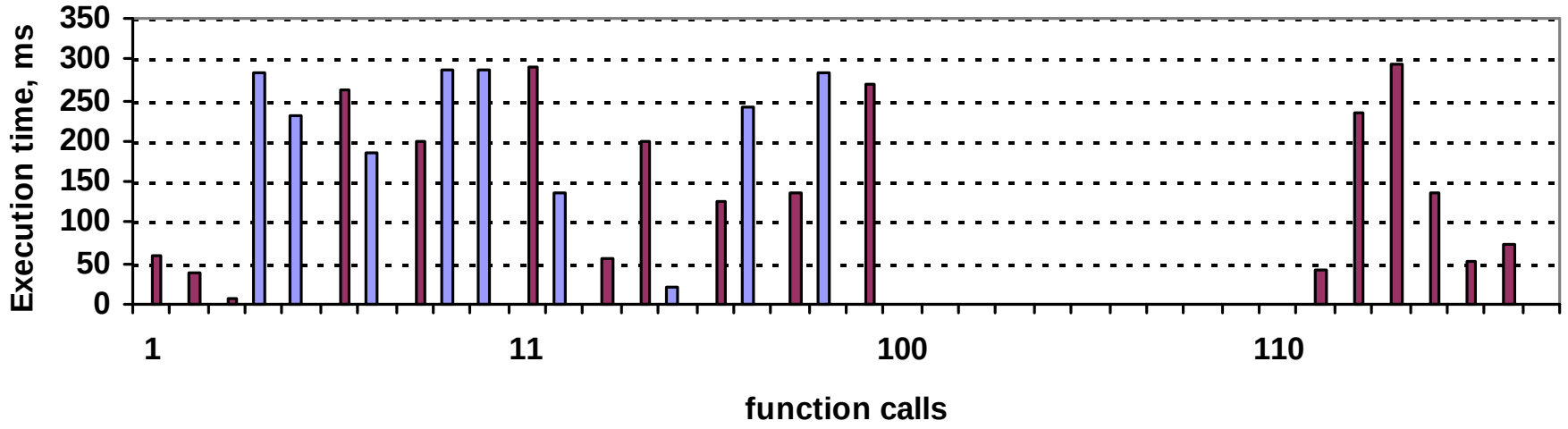
Adaptation for irregular behaviour

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



Adaptation for irregular behaviour

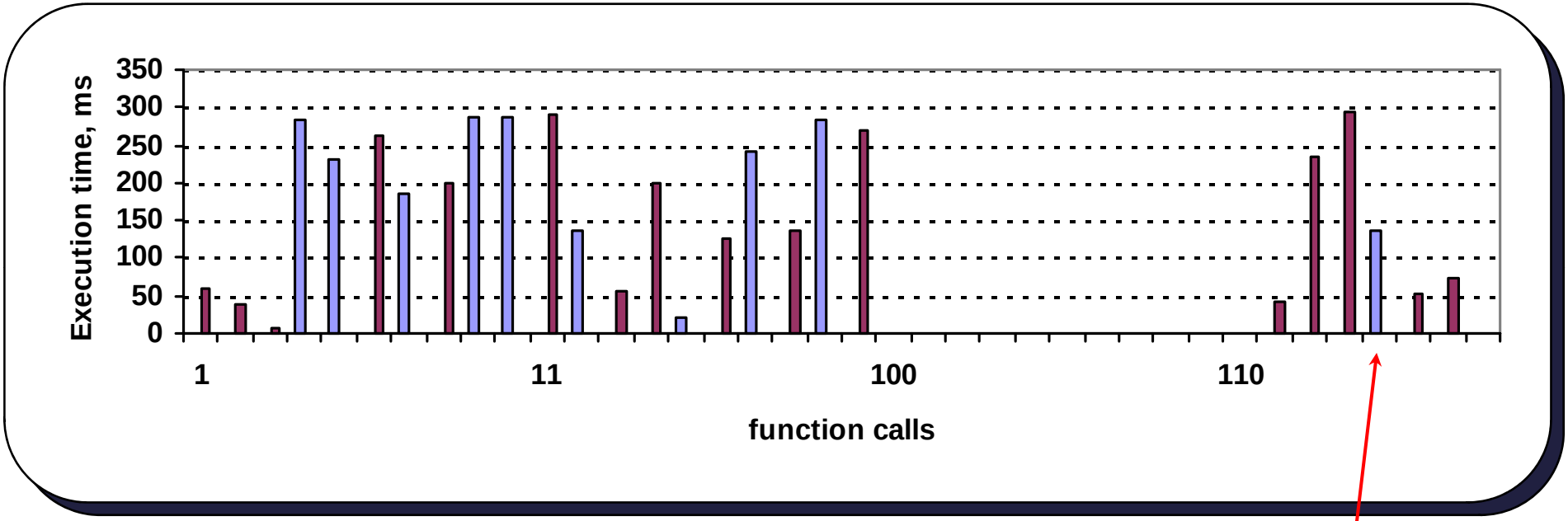
Execution time for library subroutine matmul (with 2 different versions)



- 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

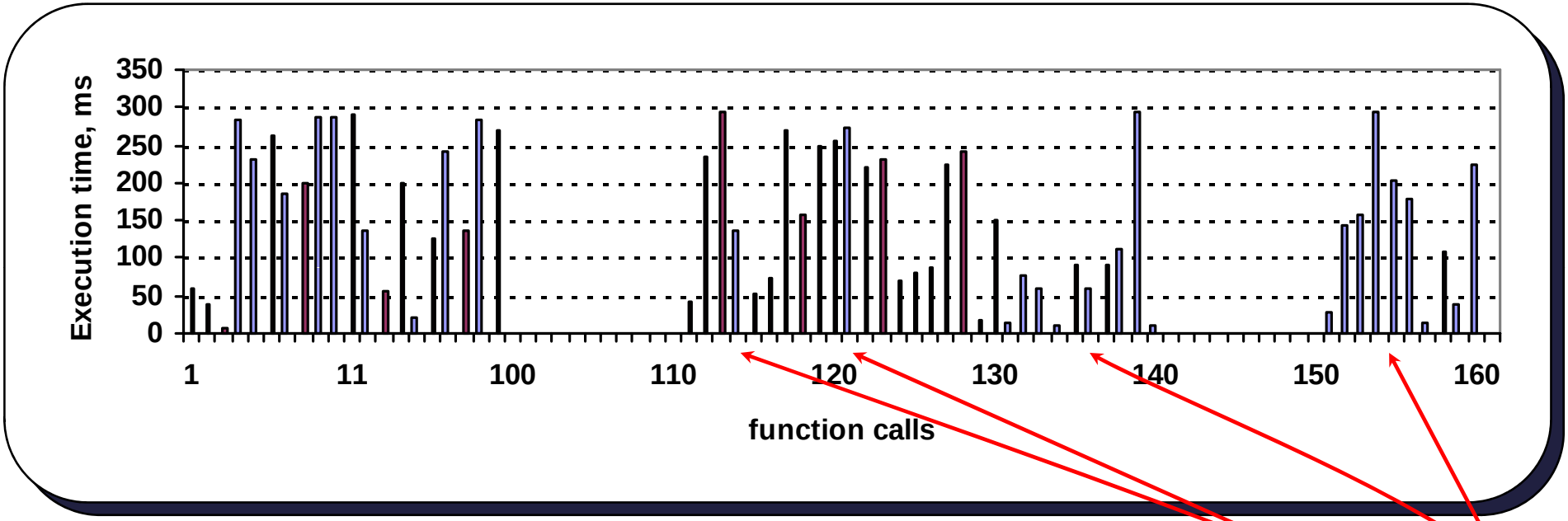
Execution time for library subroutine matmul (with 2 different versions)



- 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

Determine the effect of optimizations

Use gprof to collect time spent in functions and clones

$$avt\ (average\ time) = \frac{time\ spent\ in\ function}{number\ of\ calls}, \qquad s\ (speedup) = \frac{avt_{original}}{avt_{cloned}}$$

Continuous Optimization Framework

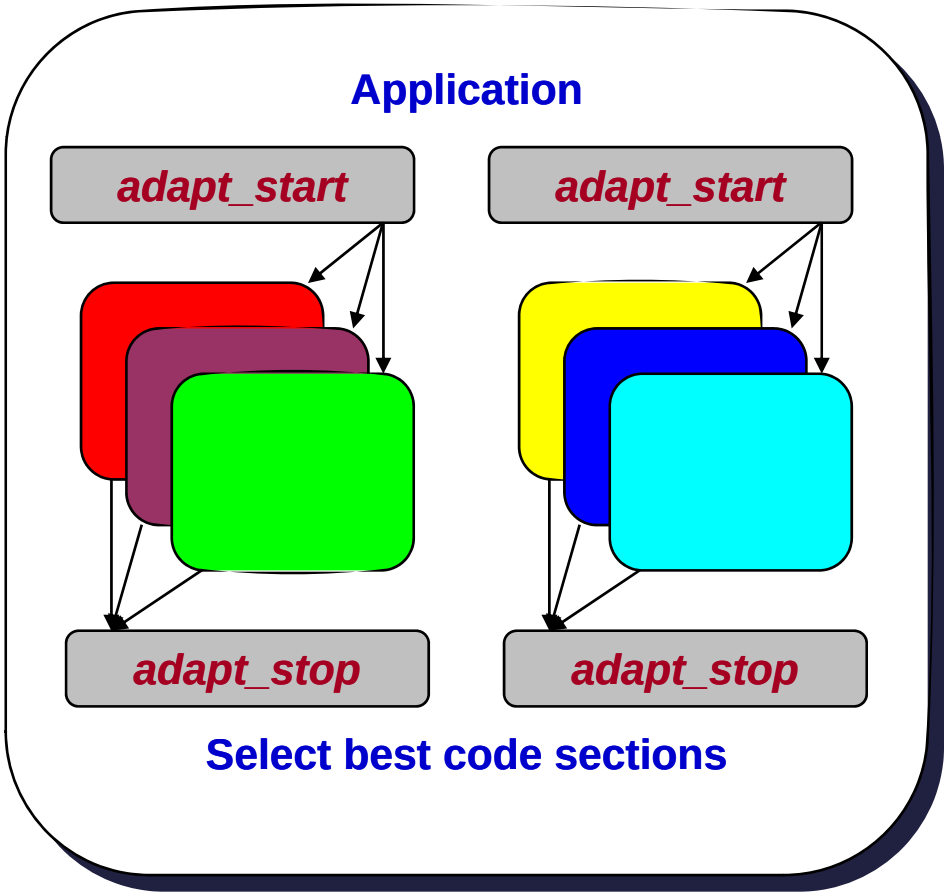
sequence of evaluations: speedups $s_1, s_2, \ldots s_n$

$$e\ (expected\ speedup) = \sum_{i=1}^{n} s_i / n$$

$$v\ (variance) = \sum_{i=1}^{n} (s_i - e)^2$$

Continuously monitor the variance to detect convergence across executions

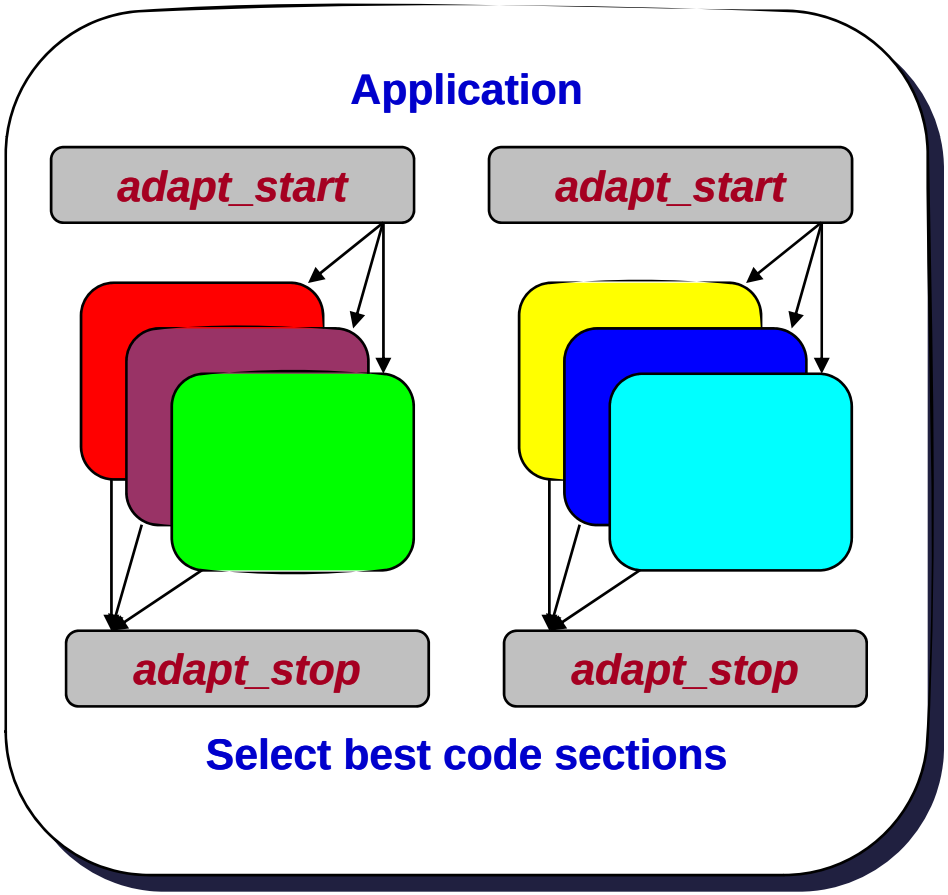
Removing adaptation overhead



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

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

Removing adaptation overhead

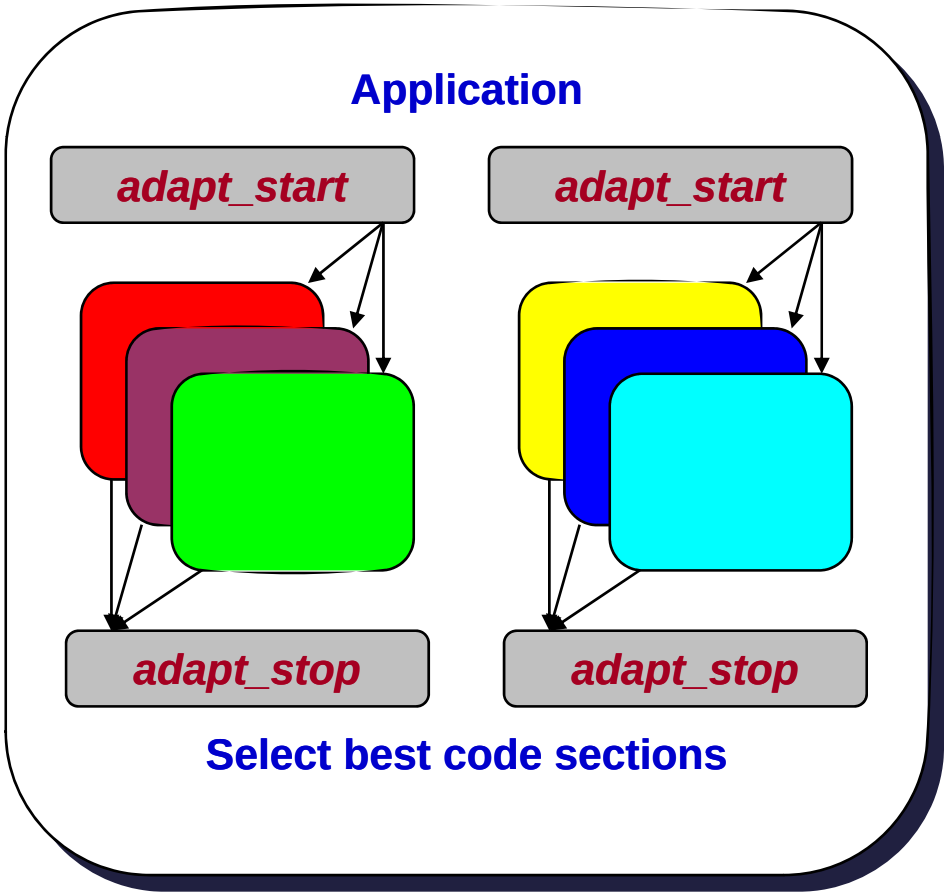


Calls to adaptation routines are not direct but through array of functions:

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:

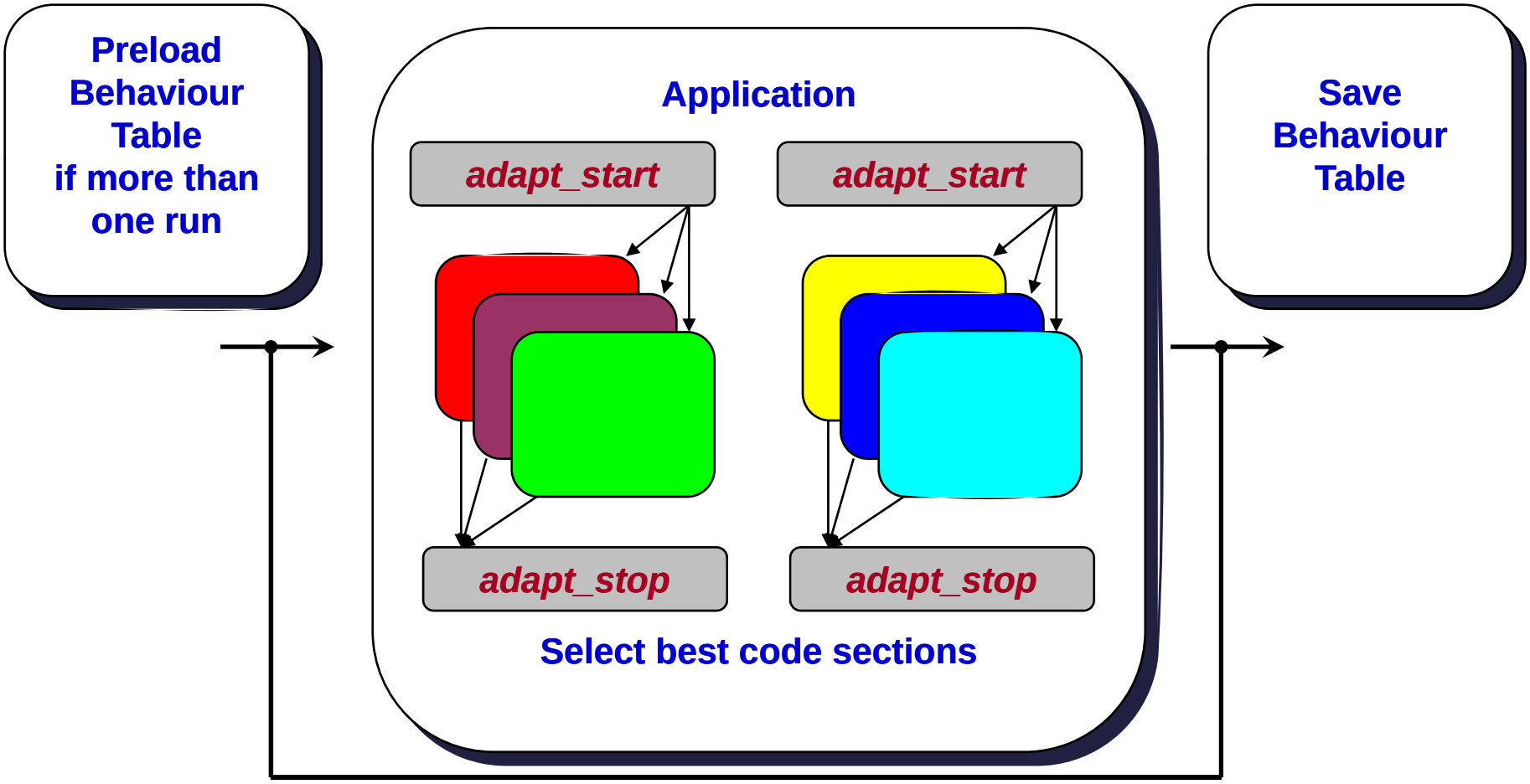
static void (*call1[..])();
static void (*call2[..])();

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

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

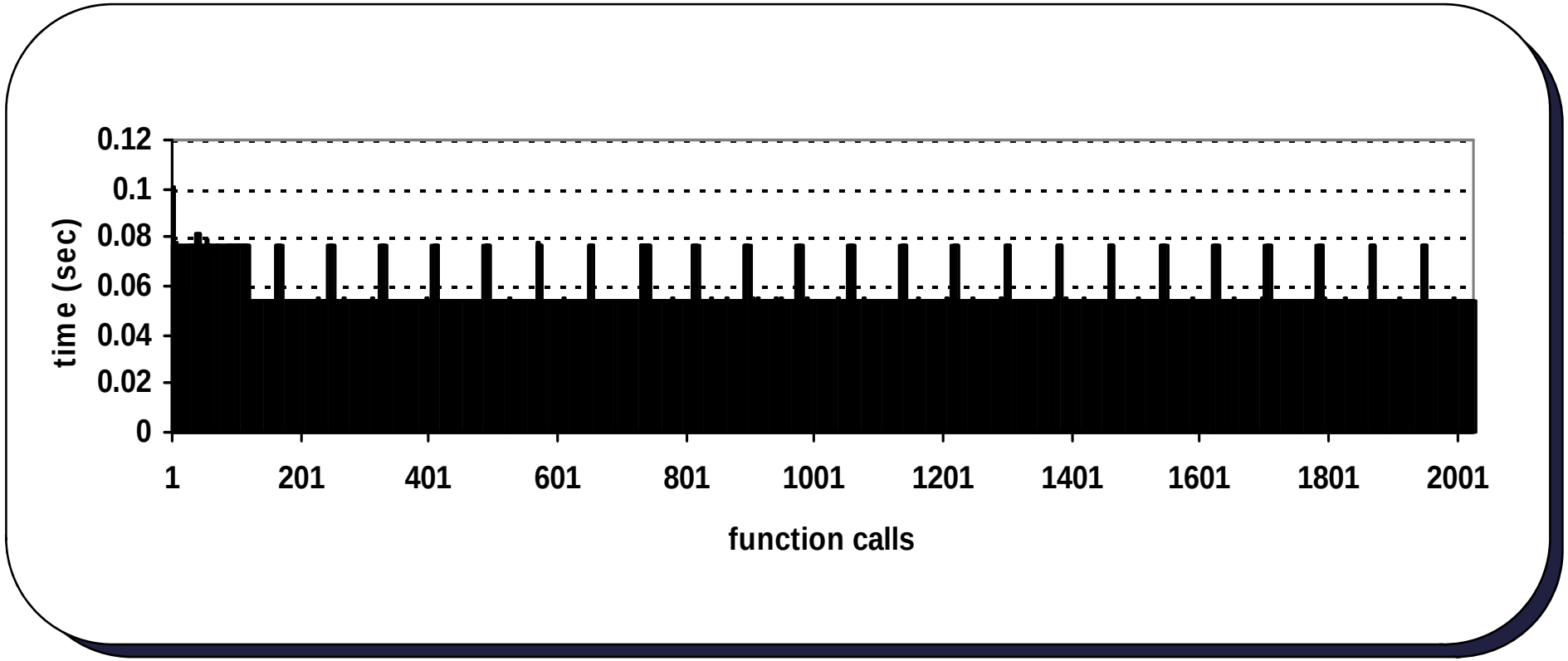
Continuous optimization and adaptation

One or multiple executions
with the same or different datasets:



Continuous optimization and adaptation

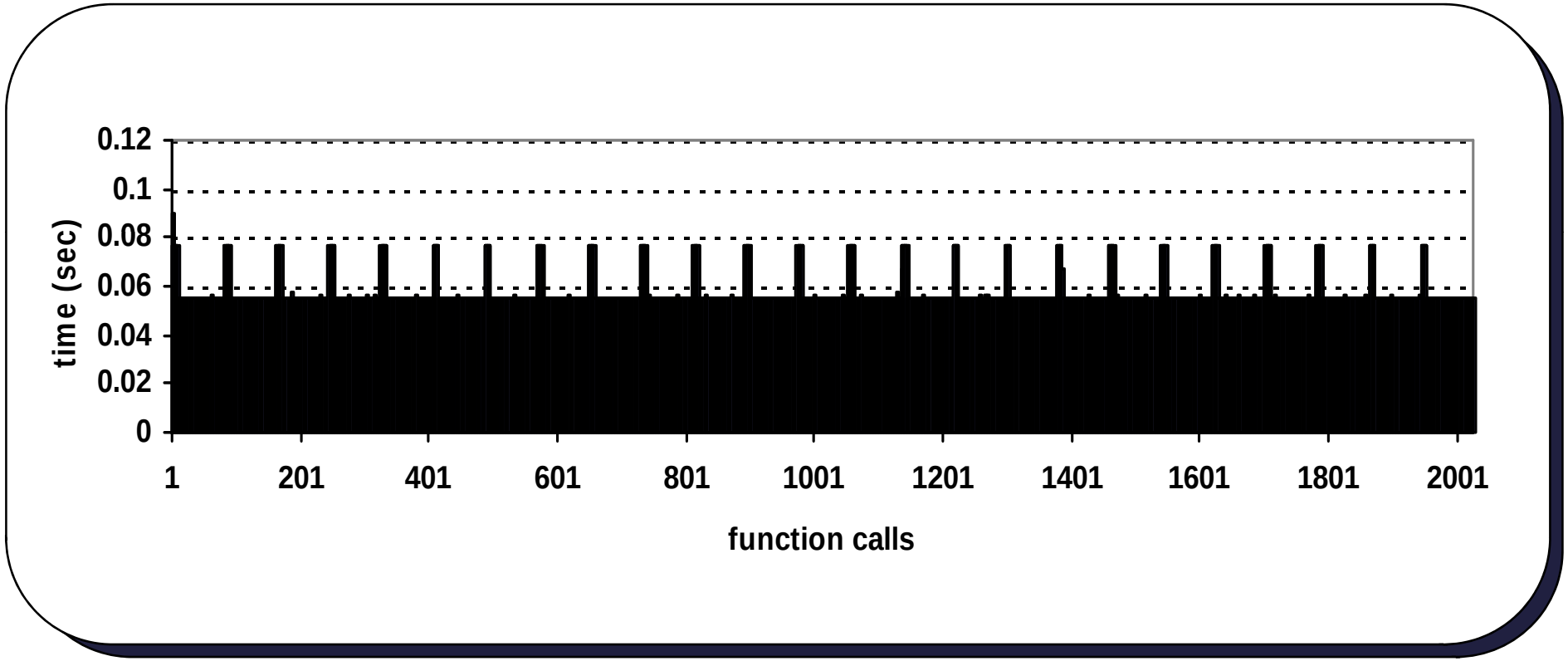
Execution times for subroutine resid of benchmark mgrid across calls



1st run

Continuous optimization and adaptation

Execution times for subroutine resid of benchmark mgrid across calls



2st run, same optimizations

Continuous optimization and adaptation

DEMO 2

Benchmark susan edges from MiBench

Clone function susan_edges and put to 2 separate files
Substitute susan_edges with the following:

```
susan_edges(in,r,mid,bp,max_no,x_size,y_size)
  uchar *in, *bp, *mid;
  int   *r, max_no, x_size, y_size;
{
float z;
int   do_symmetry, i, j, m, n, a, b, x, y, w;
uchar c,*p,*cp;

  if ((rand() % 2) == 0) susan_edges0(in,r,mid,bp,max_no,x_size,y_size);
  else                   susan_edges1(in,r,mid,bp,max_no,x_size,y_size);
}
```

compile: GCC –O1 *.c GCC –O3 *.c gcc –c –O1 susan.c, susan0.c & gcc –c –O3 susan1.c & gcc –O1 *.o
run
exec.time: 3.3 s. 4.0 s.
profile: susan_edges0: 1.18 s. (52 calls)
susan_edges1: 0.79 s. (48 calls)

Using this simple cloning technique can understand the influence of transformations on part of the code during one execution. Instead of random function can use some adaptation routines!

Conclusions

- No sophisticated dynamic optimization/recompilation frameworks;
- Allows complex sequences of compiler or manual transformations at run-time;
- Statically enables run-time optimizations for different constraints
- 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
- Can be used for parallel heterogeneous computing (compilation with different ISA for CELL or GPU-like architectures or various accelerators)
- Reliable, secure and easy to debug

Conclusions

However:

- Still no optimization knowledge reuse
- Better placement of instrumentation for adaptation is needed
- Better dataset specialization is needed (for library adaptation)
- Clustering of different behaviour is needed (different optimization scenarios)

Machine learning based optimizations

Overview

- Machine learning - what is it and why is it useful?
- Predictive modeling
- Loop unrolling and inlining
- Attempt to generalize program optimizations
- Limits and other uses of machine learning
- Future work and summary

Failings of previous approaches

• Before we have looked at techniques to overcome data dependent behavior and adaption to new processors

• However, we have not looked fundamentally at a *process of designing a compiler*

• All rely on a “clever” algorithm inserted into the compiler that determines which optimizations to apply at compile-time or runtime

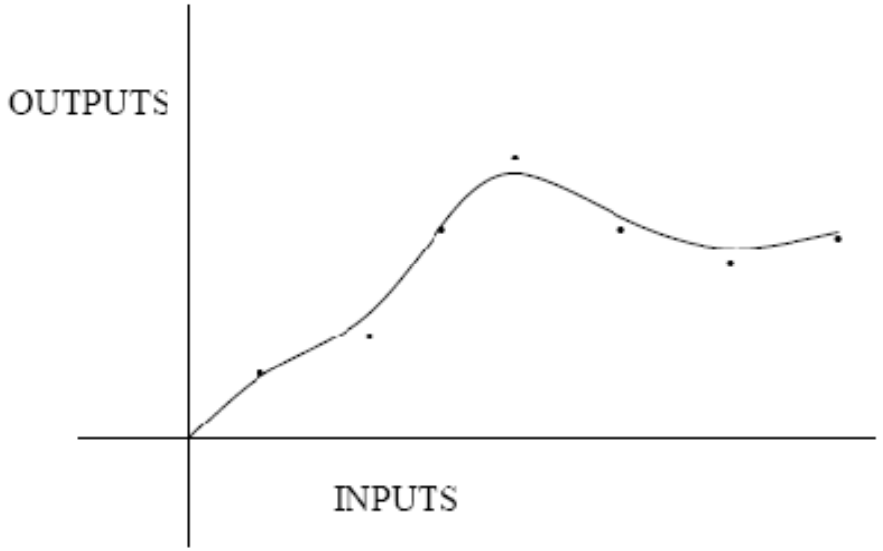
• Iterative compilation goes beyond this with no a priori knowledge but is not suitable for general compilations and does not adapt to changing data

• What we want is a smart compiler that *adapts its strategy* to changes in program, data and processor

Machine learning as a solution

• Well established area of AI, neural networks, genetic algorithms etc., but what has AI got to do with compilation?

• In a very simplistic sense machine learning can be considered as sophisticated form of curve fitting

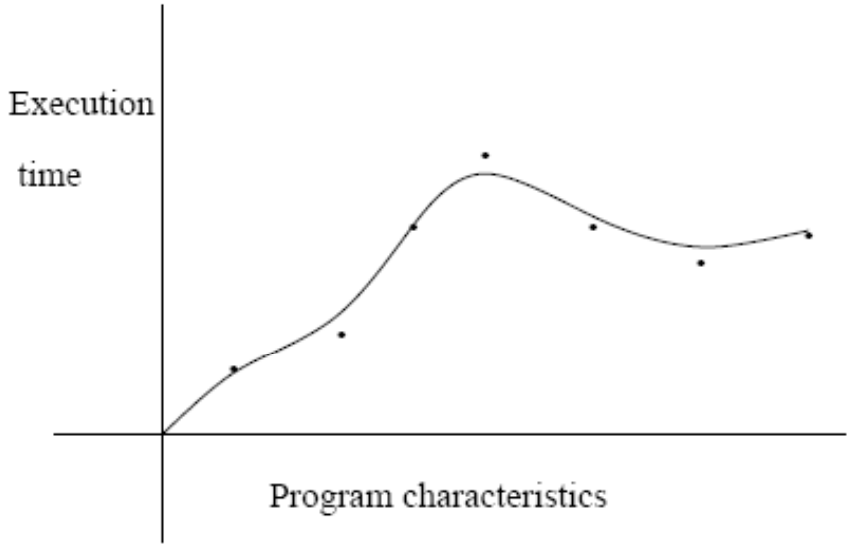


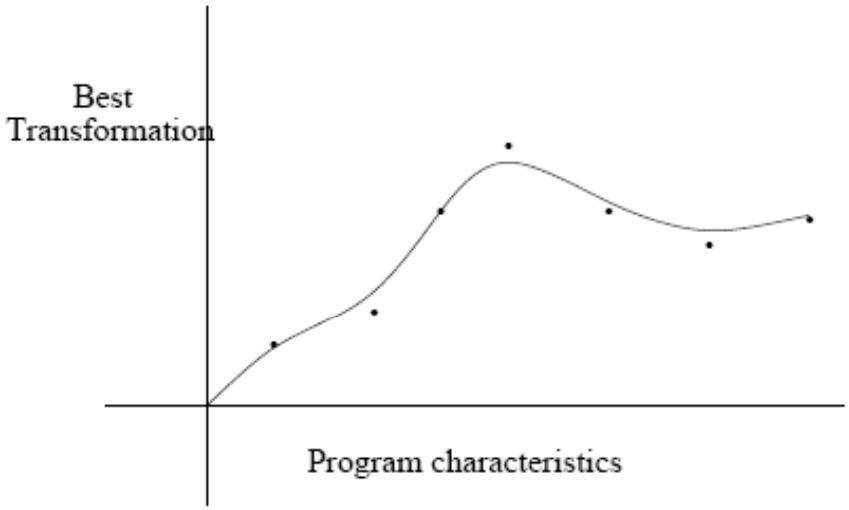
Machine learning

- Inputs: characteristics of a program and a processor

- Outputs: the optimization function we are interested in such as combination of execution time, code size, power, etc

- Theoretically predict future behavior and find the best optimization





Global optimization and predictive modeling

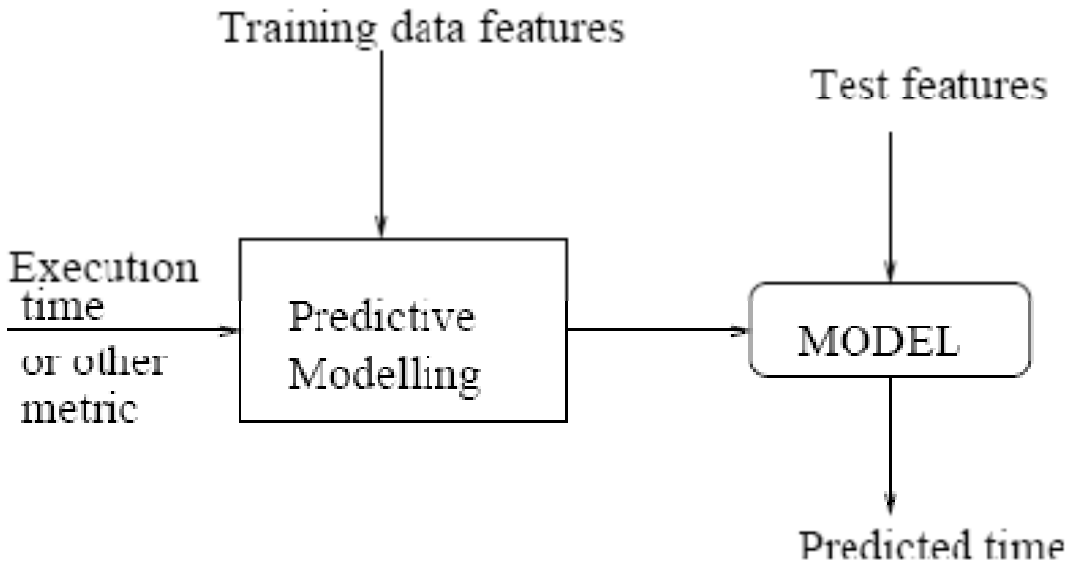
• For our purposes it is possible to consider machine learning as *global optimization* and *predictive modeling*

• *Global optimization* tries to find the best point in a space. This is achieved by selecting new points, evaluating them and then based on accumulated information selecting a new point as a potential optimum

• *Hill walking* and *genetic algorithms* are obvious examples. Very strong link with iterative compilation

• *Predictive modeling* learns about the optimizations space to build a model. Then uses this model to select the optimum point. Closely related to global optimization

Predictive modeling

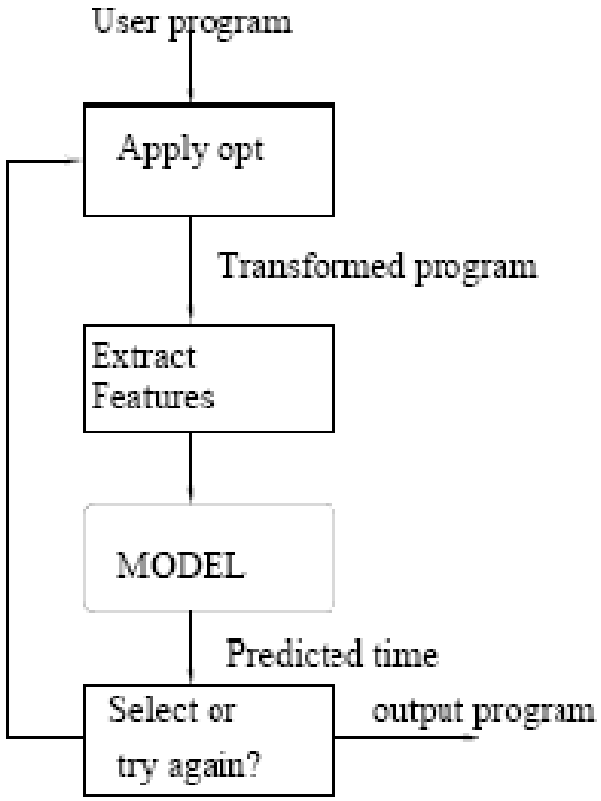


• Predictive modeling techniques all have the property that they try to learn a model that describes the correlation between inputs and outputs

• This can be a classification or a function or Bayesian probability distribution

• Distinct training and test data. Compiler writers don't make this distinction!

Predictive modeling as a proxy



- The model acts as a fast evaluator for program. Automates Soffa's performance prediction framework and speeds up iterative compilation.

- Feature selection and accuracy are main problems!

Training data

- Crucial to machine learning is correct selection of *training data*

- The data has to be rich enough to cover the space of programs likely to be encountered

- If we wish to learn over different processors so that the system can port then we also need sufficient coverage here too

- In practice it is very difficult to formally state the space of possibly interesting programs

- Ideas include typical kernels and compositions of them. Hierarchical benchmark suites could help here

Feature selection of programs

• Crucial problem with machine learning is *feature selection*. Which features of a program are likely to predict it's eventual behavior?

• In a sense, features should be a compact representation of a program that capture the essential performance related aspects and ignore the irrelevant

• Clearly, the number of spaces in the program is unlikely to be significant nor the user comments

• Compiler IRs are a good starting point as they are condensed program representation

• Loop nest depth, control-flow graph structure, recursion, pointer based accesses, data structure

Supervised learning

• Building a model based on given inputs and outputs is an example of *classical supervised learning*. We direct the system to find correlations between selected input features and output behavior

• In fact *unsupervised learning* may be more useful in the long run. Generate a large number of examples and features and allow the system to classify them into related groups with shared behavior

• This prevents missing important features and provide clues as to what aspects of a program are performance determining

• However, we need many more programs combinatorially than features to distinguish between them

Space to learn over

- Formalization of compiler optimization has not been taken really seriously

- However, in order to utilize predictive modeling, we need a descriptions of the program space that allows discrimination between different choices

- Rather than just having a sophisticated model, what we want is a system that given a program automatically provides the best optimization

- To do this means that we must have a good description of the transformation space

- The shape of the optimization space will be critical for learning. Clearly linear regression will not fit the spaces seen before

Which techniques work?

- Short answer: No one knows ;) !.. Fertile research area

- It depends on the structure of the problem space (distribution of minima) and representation of the problem

- One problem particular to compilation is that feature inputs vary in size: length of program, length of transformation sequence, order of transformations, etc

- Also we have no agreed way of representing our problem. Several of the following examples have used different techniques

- Safe to say that the level of ML sophistication is low. Seems that currently compiler writers tend to try simple things first without too much maths (though this is gradually changing with the *polyhedral transformations* being added to the mainline GCC and XLS compilers) !

Learning to unroll

- Monsifort uses machine learning to determine whether or not it is worthwhile unrolling a loop

- Rather than building a model to determine the performance benefit of loop unrolling, try to classify whether or not loop unrolling is worthwhile

- For each training loop, loop unrolling was performed and speedup recorded

- This output was translated into *“good”, “bad” or “no change”*

- The loop features were then stored alongside the output ready for learning

Learning to unroll

- Features used were based on inner loop characteristics

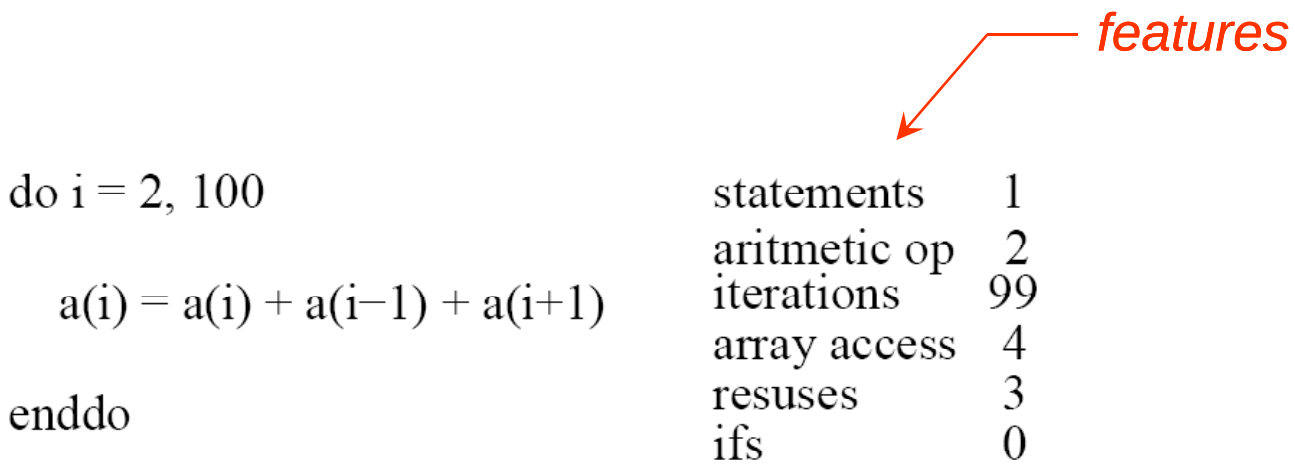
- The model induced is a partitioning of the feature space. The space was partitioned into those sections where unrolling is good, bad or unchanged

- This division was hyperplanes in the feature space that can easily be represented by a decision tree

- This learnt model is then easily used at compile time. Extract the features of the loop and see which section they belong too

- Although easy to construct, it requires regions in space to be convex. Not true for combined transformations

Learning to unroll



- Features try to capture structure that may affect unrolling decisions
- Again allows programs to be mapped to fixed feature vector
- Feature selection can be guided by metrics used in existing hand-written heuristics

Results

- Classified examples give correct result in 85% cases. Better at picking negative cases due to bias in training set

- Gave an average 4% and 6% reduction in execution time on Ultrasparc and IA64 compared to 1

- However g77 compiler is an easy compiler to improve upon at that time

- Basic approach - unroll factor not considered

Meta-compilation

- Name comes from optimizing a heuristic rather than optimizing a program

- Stephenson et al 2003 used *genetic programming* to tune *hyperblock selection*, *register allocation*, and *data prefetching* within the Trimaran's IMPACT compiler

- Represent heuristic as a parse tree. Apply mutation and cross over to a population of parse trees and measure fitness.

- Crossover = swap nodes from 2 random parse trees

- Mutate randomly: selected a node and replace with a random expression

Results

- Two of the pre-existing heuristics were not well implemented
- For hyperblock selection speedup of 1.09 on test set
- For data prefetching the results are worse - just 1.01 speedup
- The authors even admit that turning off data prefetching completely is preferable and reduces many of their gains
- The third optimization, register allocation is better implemented but only able to achieve on average a 2% increase over the manually tuned heuristic
- GP is not a focused technique, IMPACT is not of a commercial quality

Learning over UTF

• Shun (2004) uses Pugh's UTF framework to search for good Java optimizations

• Space of optimization to learn included entire UTF. Training data gathered by using a smart iterative search

• Then using a similar feature extraction to Monsifort classify all found results

• Uses nearest neighbour based learning able to achieve 70% of the possible performance found using iterative compilation on cross-validated test data

• Larger experimental set needed to validate results. Going beyond loop based transformations for Java

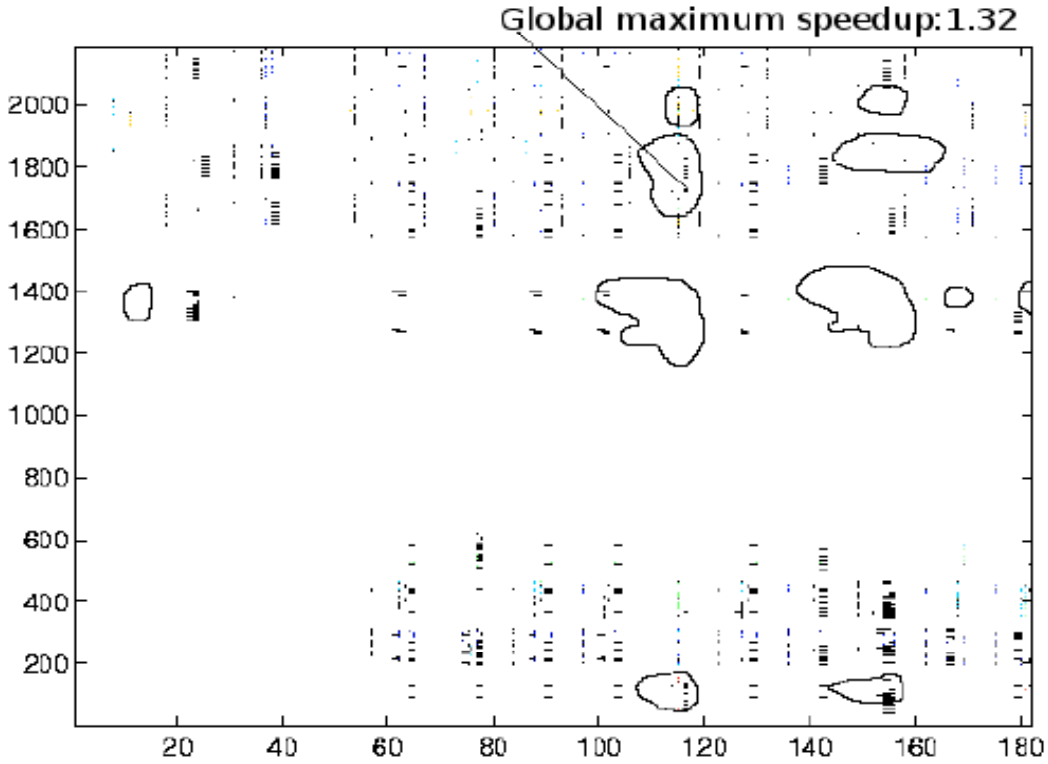
More general approaches?

Static characterization of programs

- Embedded systems application
  - UTDSP benchmarks: compute intensive DSP
  - AMD Au1500, gcc 3.2.1, -O3
  - TI C6713, TI compiler v2.21, -O3
- Exhaustively enumerated optimization search space
  - 14 transformations selected
  - all combinations of length 5 evaluated
- Allows comparison of techniques
  - How near the minima each technique approaches
  - Rate of improvement
  - Characterization of the space

F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M.F.P. O'Boyle, J.Thomson, M. Toussaint and C.K.I. Williams. Using Machine Learning to Focus Iterative Optimization. *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization (CGO)*, New York, NY, USA, March 2006

Static characterization of programs



Search space = **396000** program transformations

*Predict **2..10** best transformations from this space based on program features and previous optimization experience*

Focusing search (off-line training):

- Independent identically distributed (IID) model
- Markov model

Predicting best transformation for a new program:

- Static features
- Nearest neighbors classifier

Dynamic characterization of programs

Previously we used *static code features* to obtain good optimizations for new programs

However, it is difficult or impossible to characterize *program run-time behavior* on modern complex architecture using only static code features

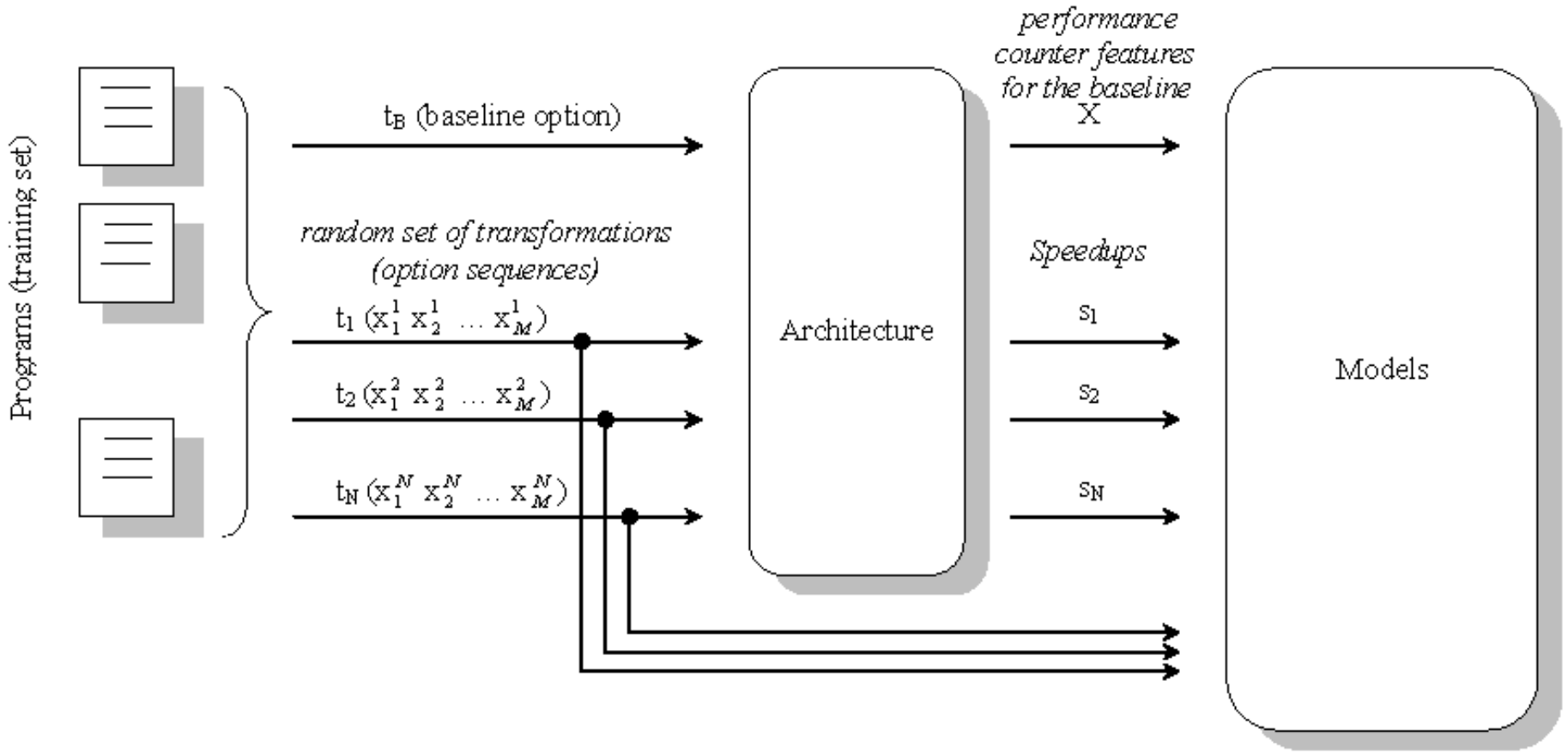
Performance counters provide a compact summary of a program's dynamic behavior

How to use them to select good optimization settings?

John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F.P.O'Boyle and Olivier Temam. Rapidly Selecting Good Compiler Optimizations using Performance Counters. Proceedings of the 5th Annual International Symposium on Code Generation and Optimization (CGO), San Jose, USA, March 2007

General optimizations

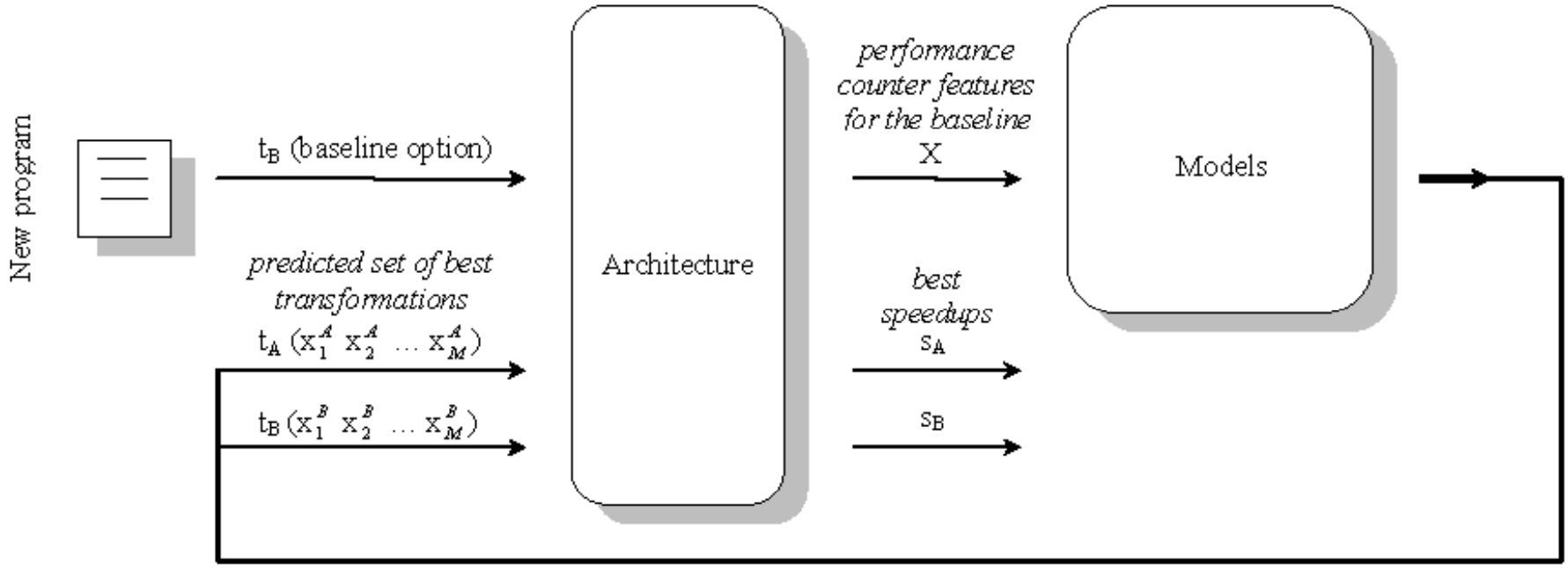
Predictive modeling using logistic regression



(a) Summary of the predictive modelling procedure. We use the features x, the transformations t, and (implicitly) the speed-ups $\{s\}$ for constructing the training data $< x , t >$. We then evaluate the mapping from the performance counters to the transformation sequences $x \rightarrow t$ by fitting a probabilistic model to the training set.

General optimizations

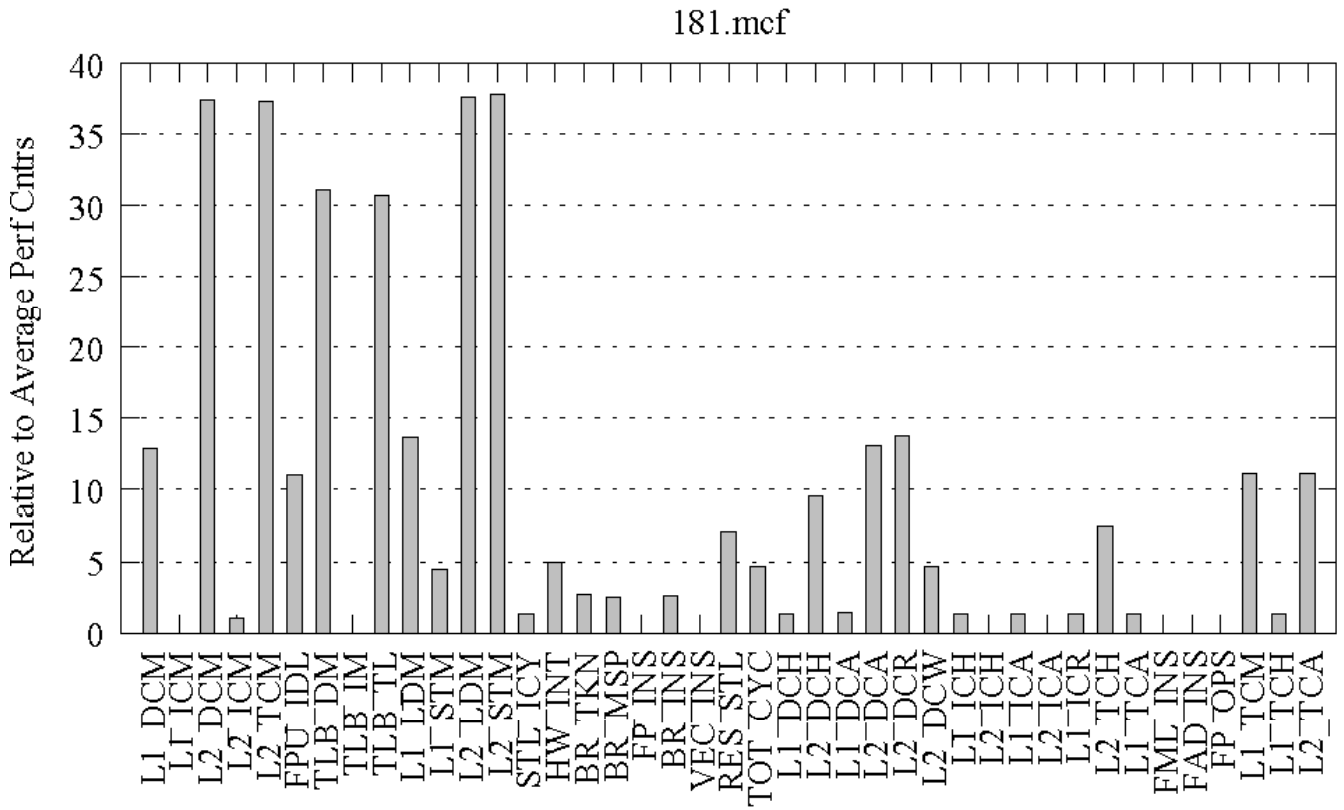
Using models



(b) Inference using a predictive model. Given a new benchmark, we first extract performance counter features. These features are then fed into our trained models which then output a set of transformation sequences to apply to the new benchmark.

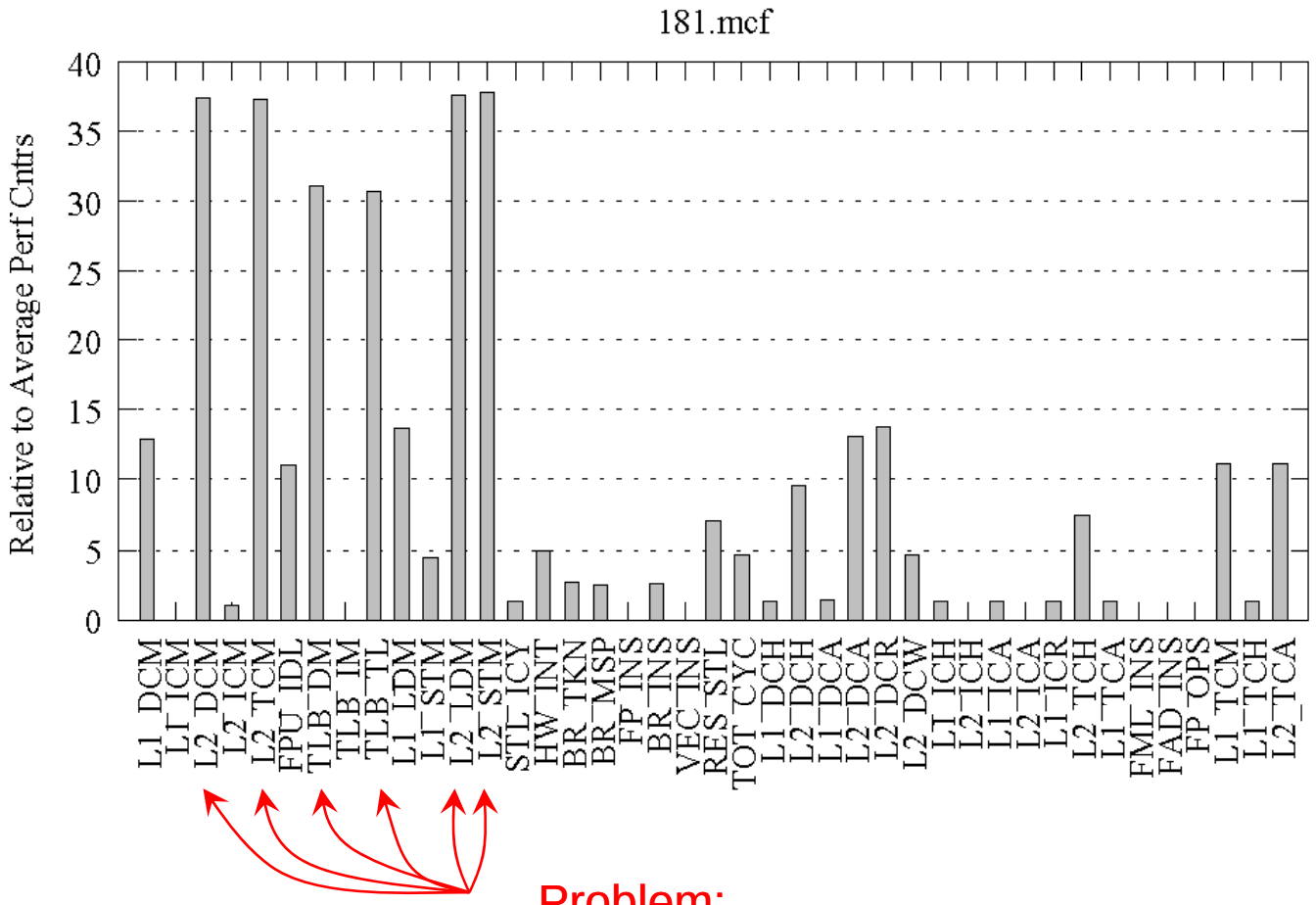
Dynamic characterization of programs

Performance counter values for 181.mcf compiled with -O0 relative to the average values for the entire set of benchmark suite (SPECFP,SPECINT, MiBench, Polyhedron)



Dynamic characterization of programs

Performance counter values for 181.mcf compiled with -O0 relative to the average values for the entire set of benchmark suite (SPECFP,SPECINT, MiBench, Polyhedron)

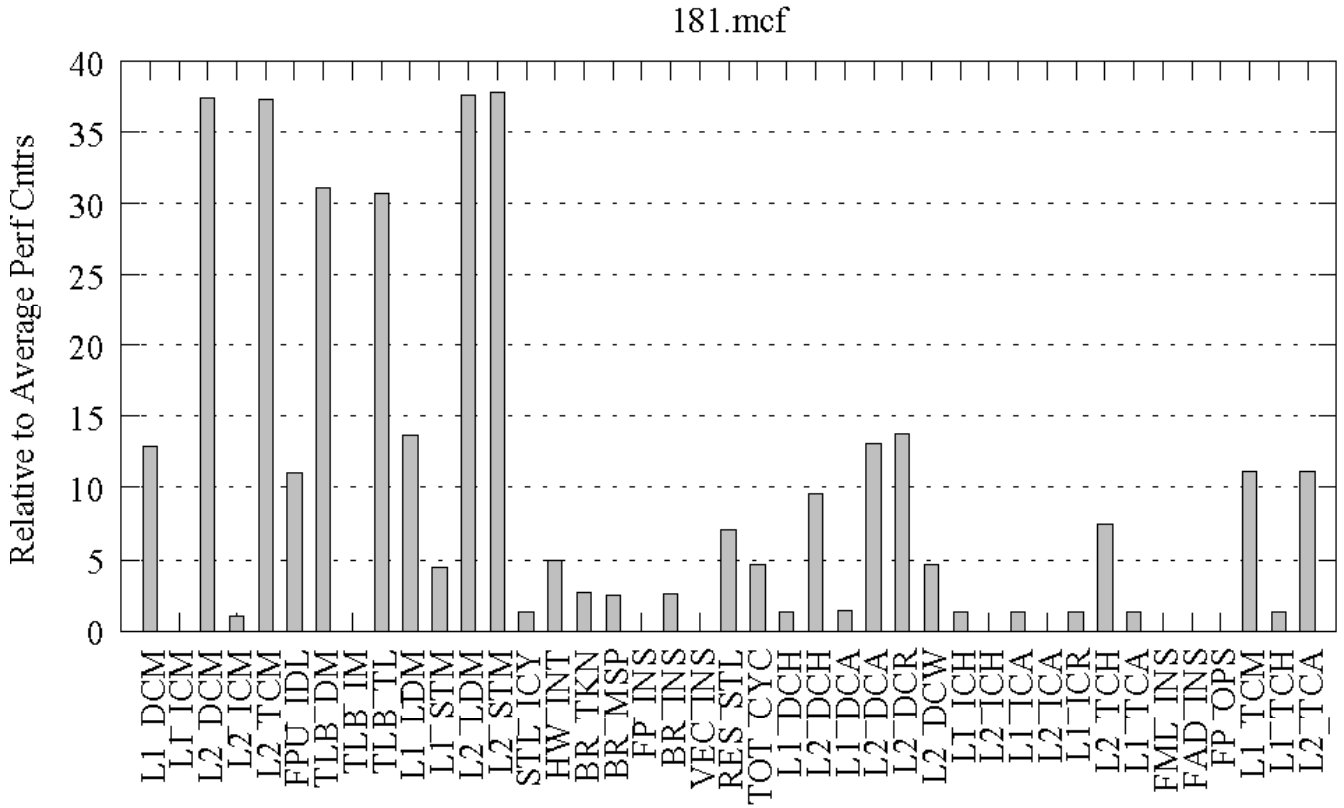


Problem:

greater number of memory accesses per instruction than average

Dynamic characterization of programs

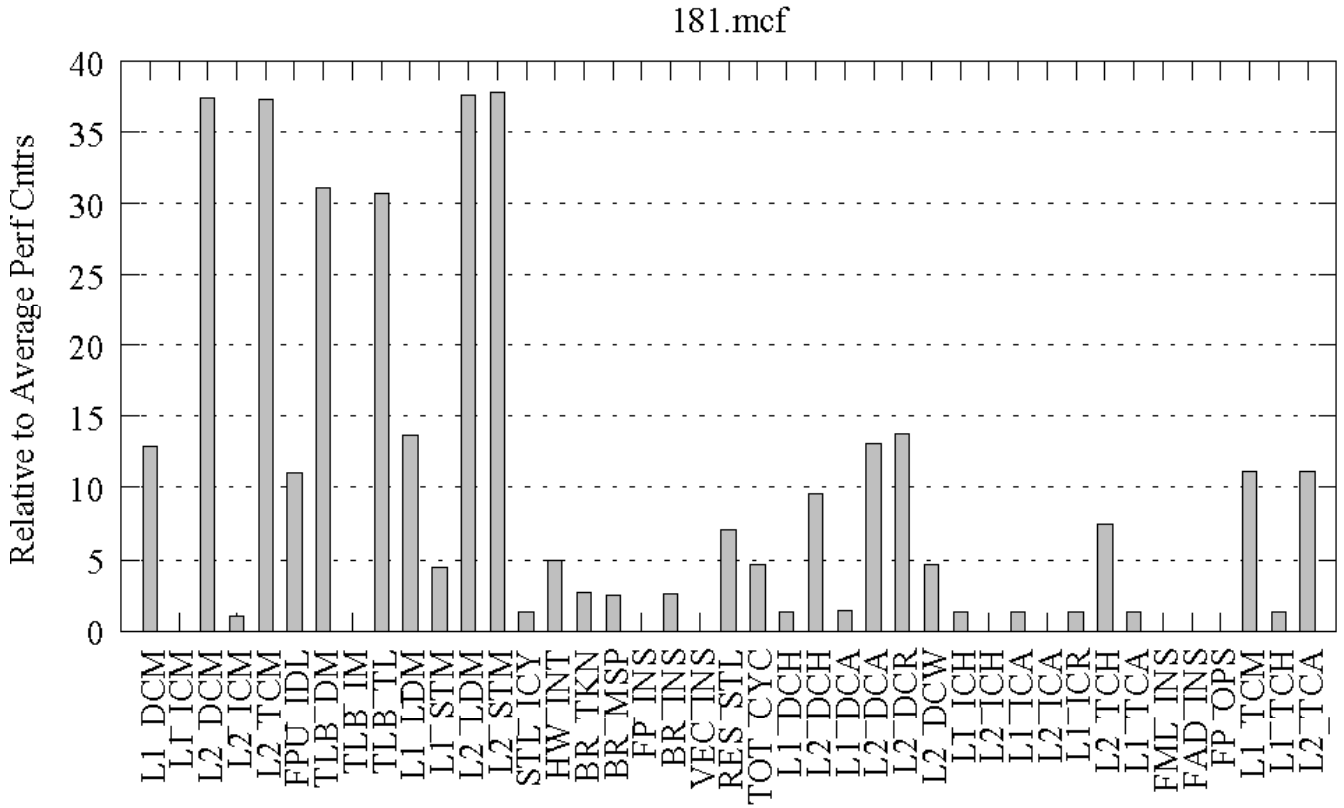
Performance counter values for 181.mcf compiled with -O0 relative to the average values for the entire set of benchmark suite (SPECFP,SPECINT, MiBench, Polyhedron)



Solving all performance issues one by one is slow and can be inefficient due to their non-linear dependencies …

Dynamic characterization of programs

Performance counter values for 181.mcf compiled with -O0 relative to the average values for the entire set of benchmark suite (SPECFP,SPECINT, MiBench, Polyhedron)



Solving all performance issues one by one is slow and can be inefficient due to their non-linear dependencies …

CONSIDER ALL PERFORMANCE ISSUES AT THE SAME TIME !

MILEPOST project

Machine Learning for Embedded Programs Optimization

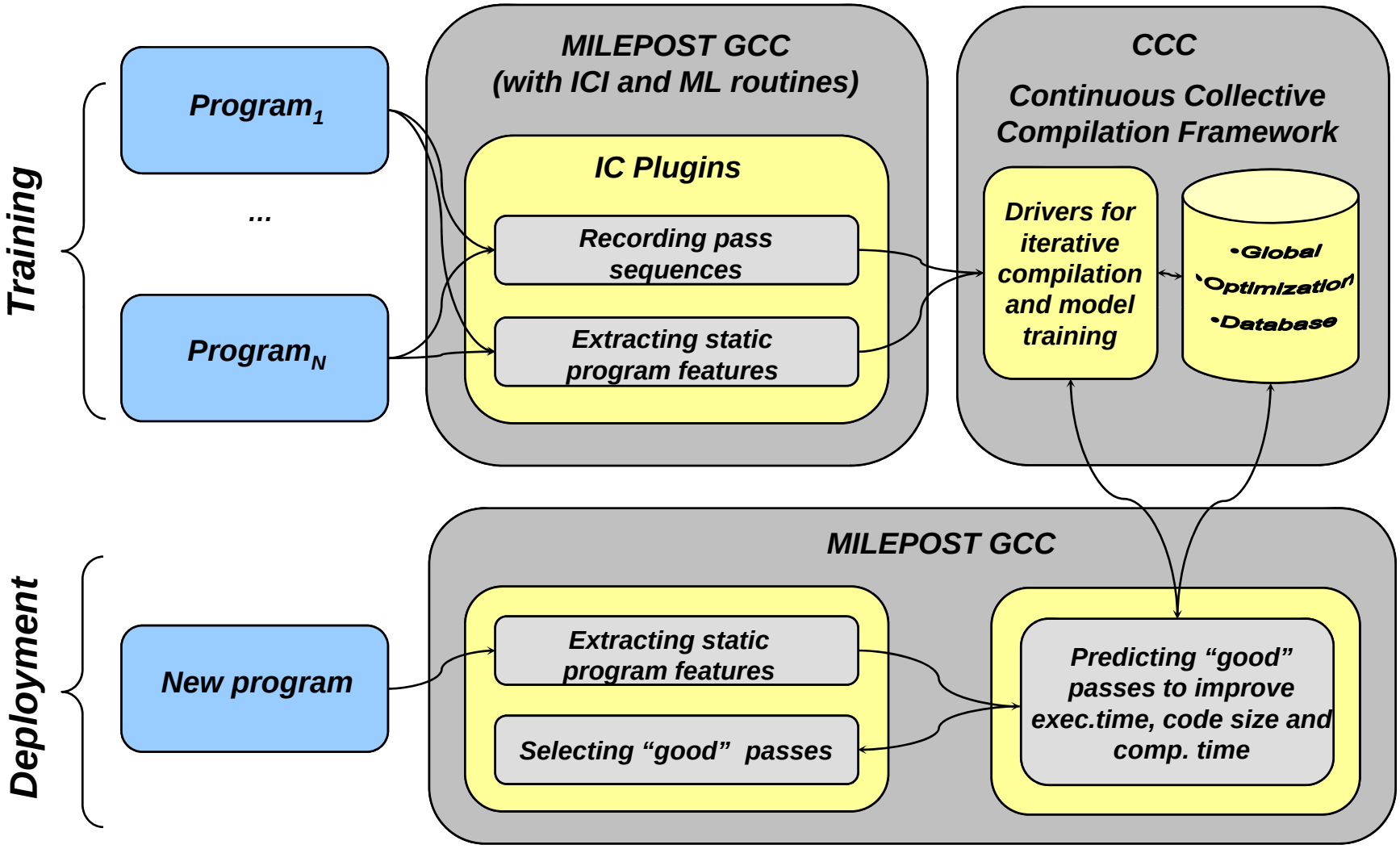
Objective is to develop compiler technology that can automatically learn how to best optimize programs for re-configurable heterogeneous embedded processors and dramatically reduce the time to market.

Partners:
INRIA, University of Edinburgh, IBM, ARC, CAPS Enterprise

Developed techniques and software are publicly available and hopefully will influence the future compiler developments

http://www.milepost.eu

MILEPOST framework



Feature extraction

We can now add new passes that are not included into default optimization heuristic but called through ICI.

Example: program static feature extractor (thanks to IBM Haifa)

export XSB_DIR="…"

export ICI_PLUGIN="$ICI_PLUGINS_HOME/extract-program-static-features.so"

export ICI_PROG_FEAT_PASS=fre

export ICI_PROG_FEAT_EXT_TOOL="$ICI_PLUGINS_HOME/ml-feat-proc"

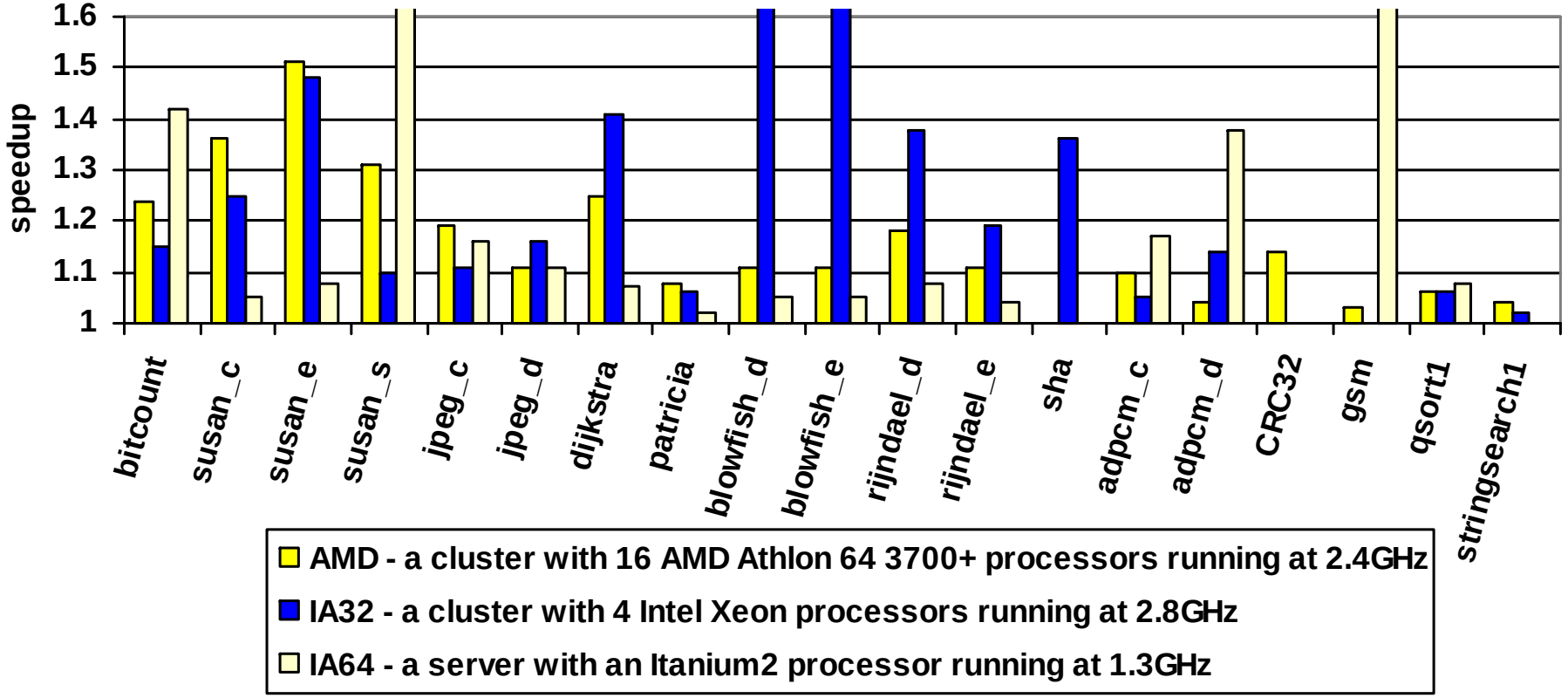
export ML_ST_FEAT_PROC="$ICI_PLUGINS_HOME/featlstn.P"

export ICI_USE=1

ft1 - Number of basic blocks in the method
…
***ft20 - Number of conditional branches in the method ***
***ft21 - Number of assignment instructions in the method ***
…
***ft22 - Number of binary integer operations in the method ***
***ft23 - Number of binary floating point operations in the method ***
…
***ft24 - Number of instructions in the method ***
***ft25 - Average of number of instructions in basic blocks ***
…
***ft54 - Number of local variables that are pointers in the method ***
***ft55 - Number of static/extern variables that are pointers in the method ***

Experiments

Generating training set to build model



Traditional iterative search:

500 random sequences of flags and associated passes (turned on or off)

Later “focused” search

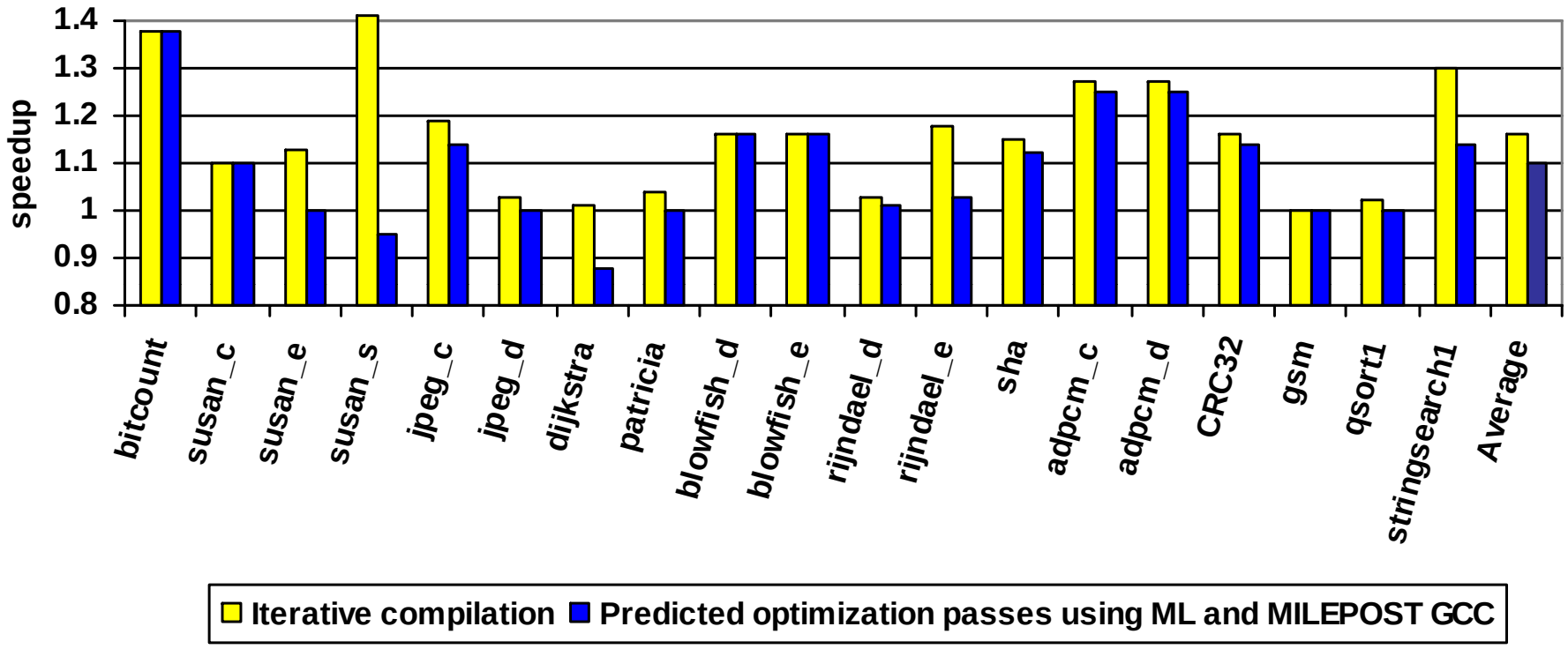
Experiments

Building and using model to predict optimizations

Use static or dynamic (hardware counters) program features to find similarities between programs to focus search for good optimizations

Similar to feedback directed optimizations, except we reuse "global optimization knowledge" and use program features to suggest good optimizations

Experiments



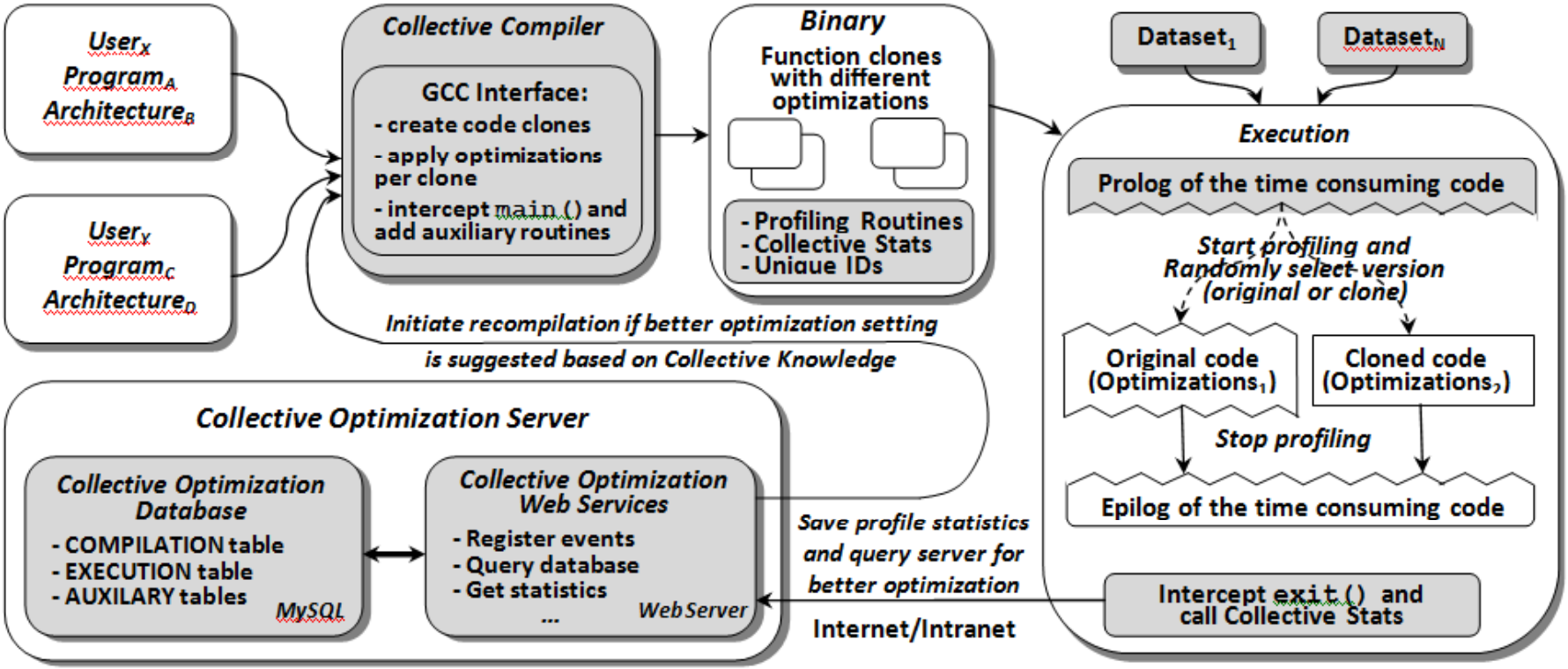
Evaluating model performance

(FPGA implementation of the ARC 725D processor)

Continuous Collective Compilation Framework

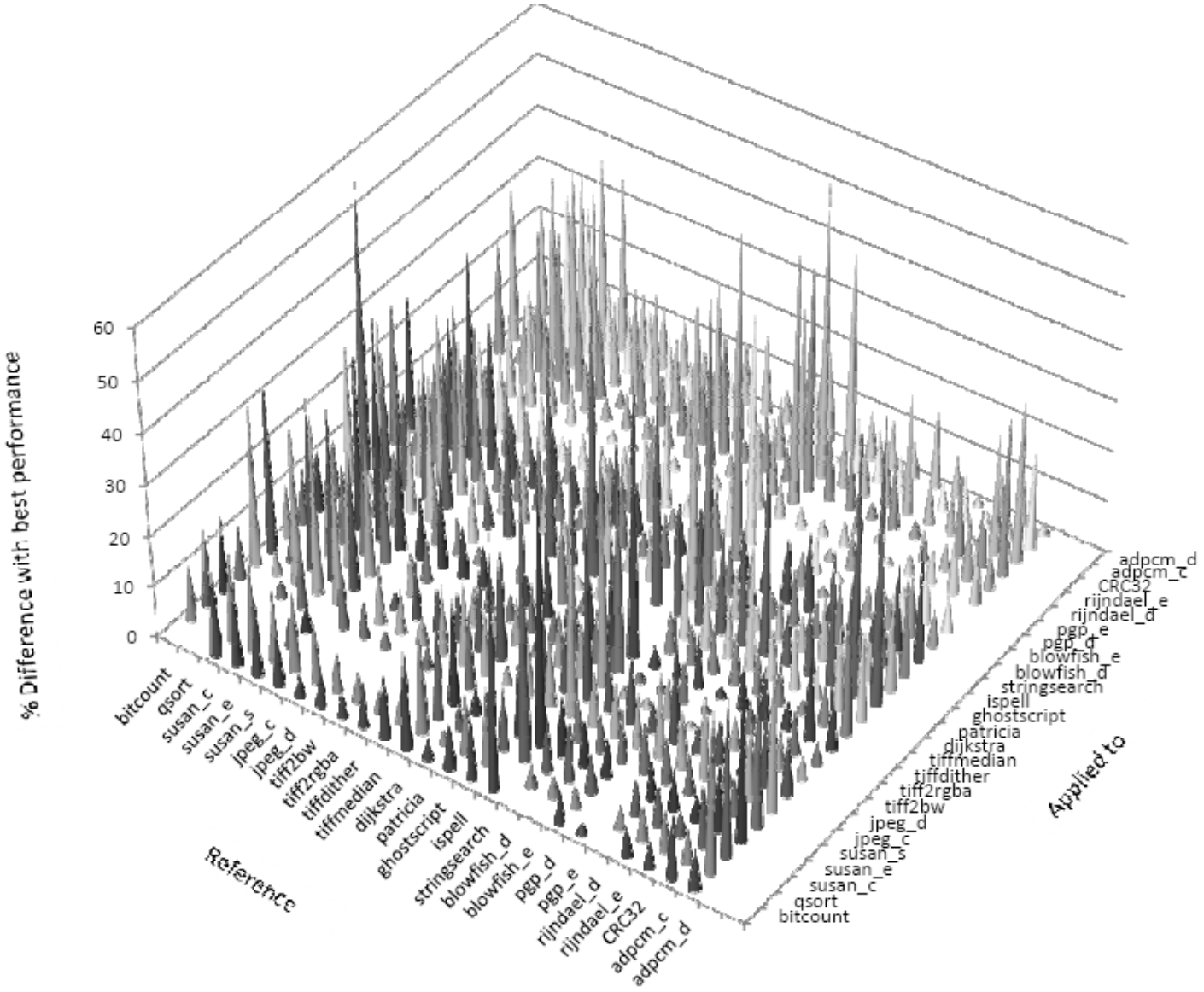
- Unify iterative experiments and optimization knowledge reuse
- Collect and analyze data from various partners in a global database:
  - **COMPILERS, DATASETS, ENVIRONMENTS, OPT_FLAGS_GLOBAL, PLATFORMS**
  - **PROGRAMS, PROGRAM_FEATURES, PROGRAM_PASSES,**
  - **STATS_COMP_GLOBAL_FLAGS, STATS_EXEC_GLOBAL**
- Support iterative compilation (flags & passes) with different strategies, transparent profiling using hardware counters collection using PAPI library
- During last 6 months around 2,000,000 executions on various platforms:
  - **x86, x8664, IA64**
  - **TMS320C6713**
  - **ICT GODSON2**
  - **ARC 700**
- Build machine learning models to improve GCC performance on average across all programs or for

Continuous Collective Compilation Framework



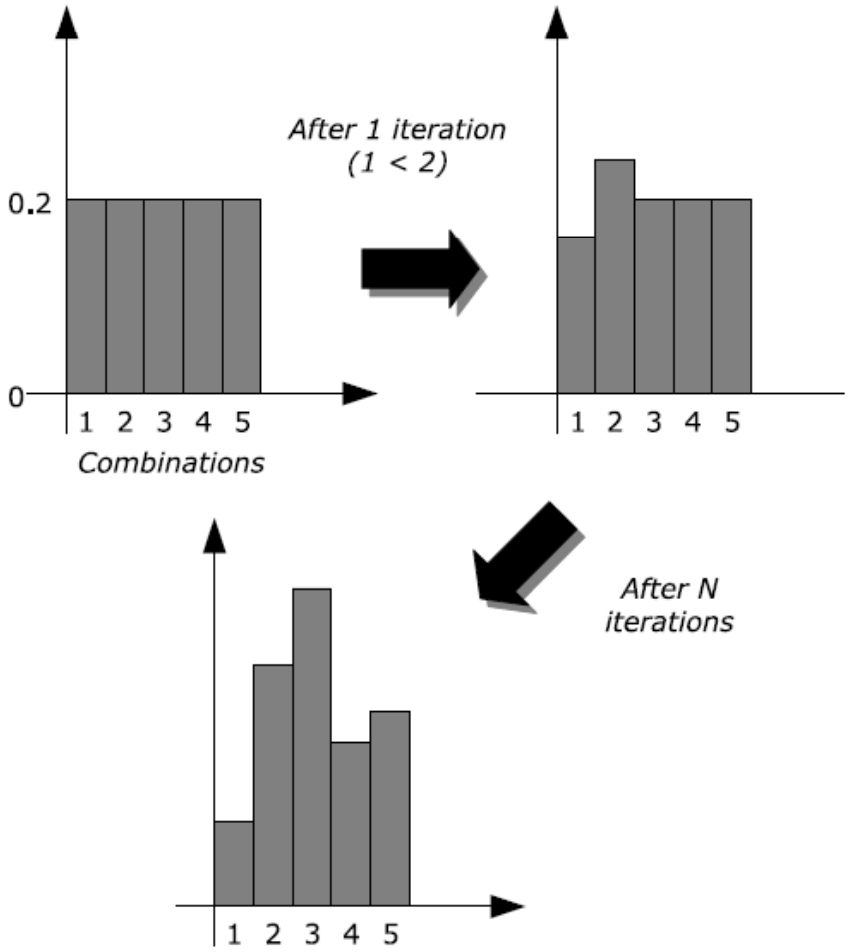
- *Static function versioning and run-time adaptation to avoid reference runs*
- *Unobtrusive collection of statistics in the Collective Database*
- *Suggest good optimizations based on Collective Knowledge*

Continuous Collective Compilation Framework



Learning across different datasets

Competition between optimization



Computing the probability distribution to select an optimization

combination based on continuous competition between combinations

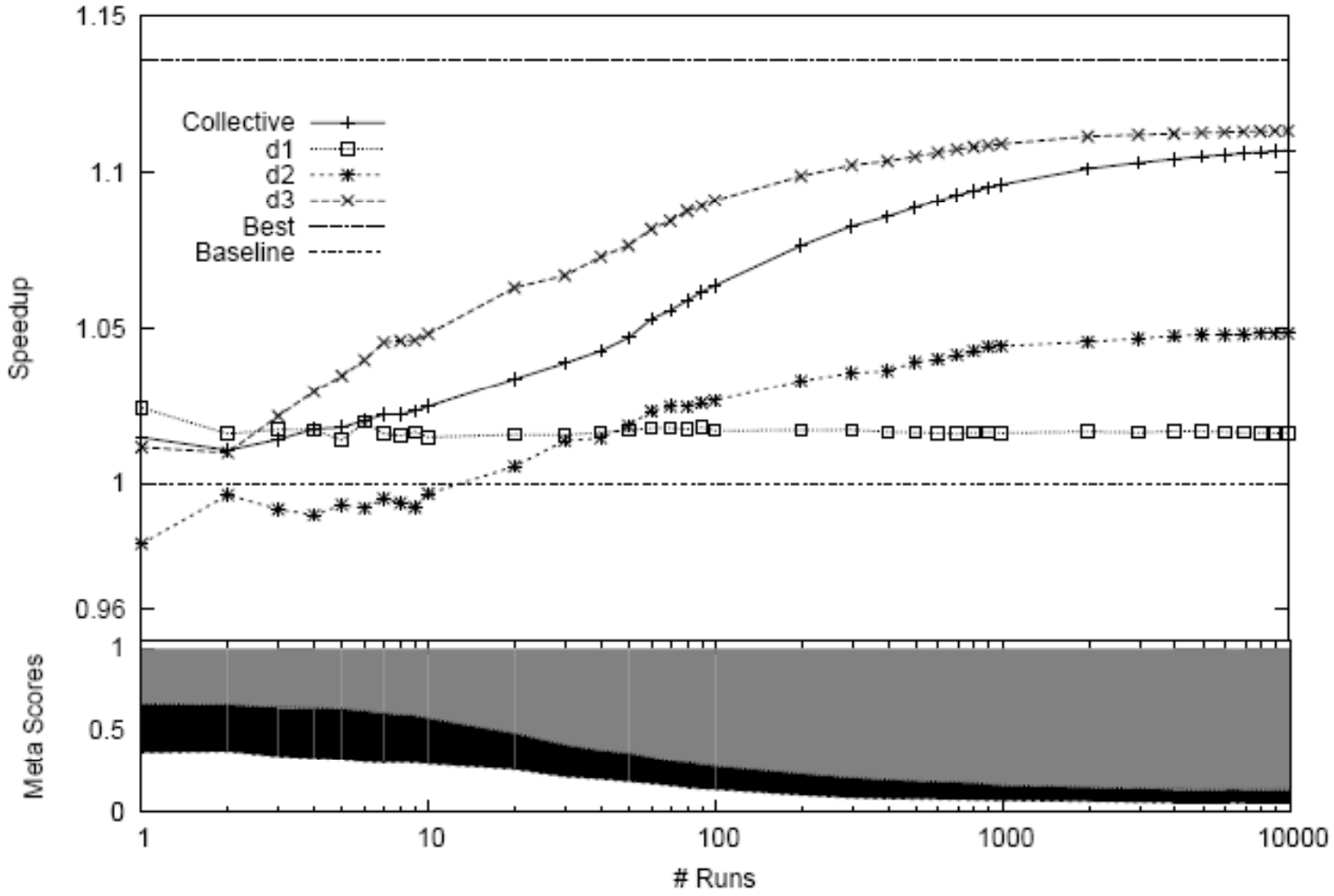
Maturation stages of a program

- *Stage 3: Program well known, heavily used*
- *Stage 2: Program known, a few runs only*
- *Stage 1: Program unknown*

There is a permanent competition between

the different stages distributions (d1, d2, d3)

Performance evaluation



Average performance of collective optimization and individual distributions

(bottom: meta-scores of individual distributions; grey is d3, black is d2, white is d1)

Machine learning for DSE

Speeding up Architecture Design Space Exploration

Problems:

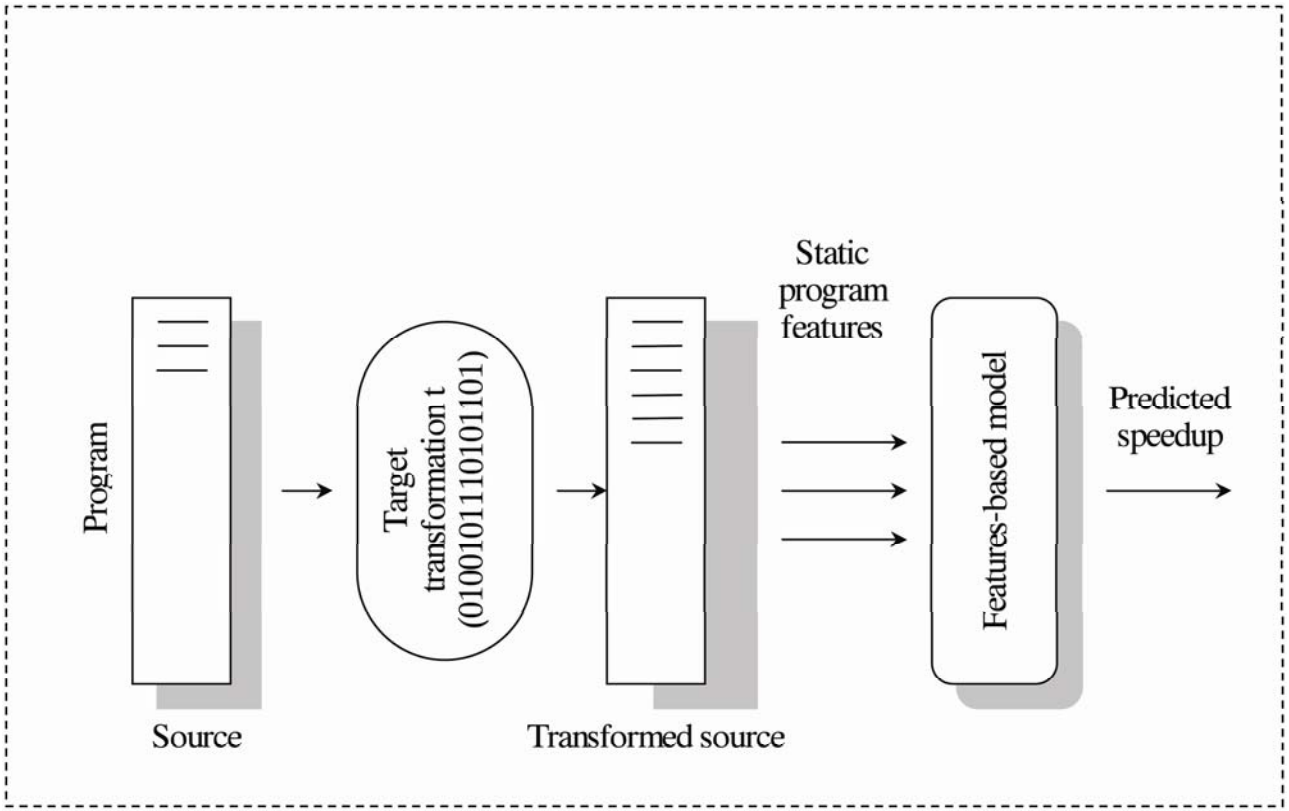
- Developing an optimizing compiler for new architecture is difficult particularly when only simulator is available
- Tuning such compiler requires many runs
- Simulators are orders of magnitude slower than real processors
- Therefore compiler tuning is highly restricted

Goal:

develop a technique to automatically build a performance model for predicting the impact of program transformations on any architecture, based on a limited number of automatically selected runs

John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael F.P. O'Boyle, Grigori Fursin and Olivier Temam. Automatic Performance Model Construction for the Fast Software Exploration of New Hardware Designs. International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2006), Seoul, Korea, October 2006

Machine learning for DSE

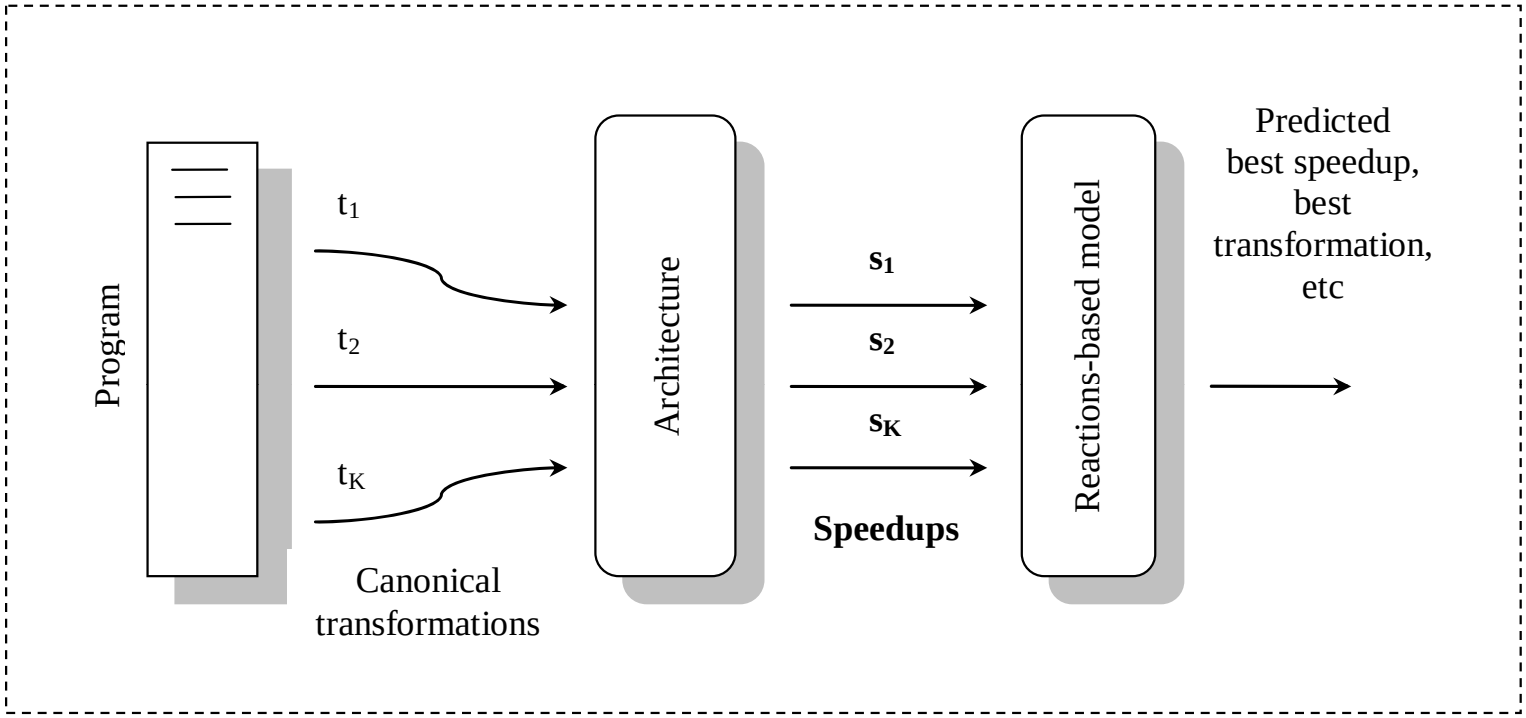


Features-based model

Input: *static features extracted from the transformed program at the source level*

Output: *program speedup*

Machine learning for DSE



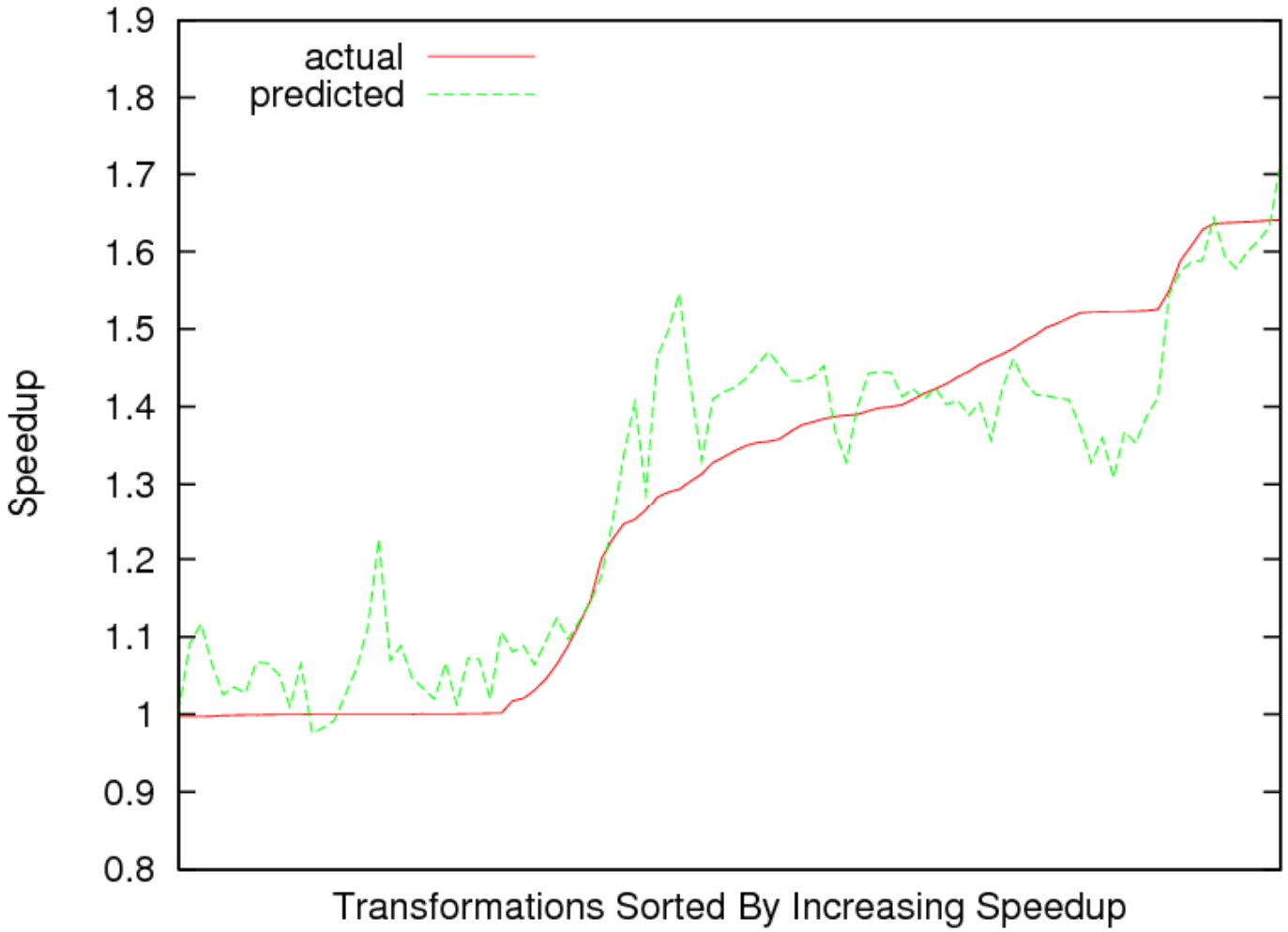
Reactions-based model

Input: *speedups on canonical transformation sequences*

Output: *transformation sequence speedup*

Machine learning for DSE

Speeding up Architecture Design Space Exploration



Reliable performance model on an unseen architecture after a few probes → fast search

Conclusions

• We believe that machine learning will revolutionize compiler optimization and will become mainstream within a decade for both compiler optimizations, run-time adaptation, parallelization and architecture design space exploration

• However, it is not a panacea, solving all our problems

• Fundamentally, it is an automatic curve fitter. We still have to choose the parameters to fit and the space to optimize over

• Complexity of space makes a big difference. Tried using Gaussian process predicting on PFDC'98 spaces - worse than random selection…

• Much remains to be done - fertile research area

Continuous Collective Compilation
http://gcc-ccc.sourceforge.net

Literature

- Hennessy and Patterson: *Computer Architecture: A Quantitative Approach (4th Edition)*, Morgan Kaufmann, 2006
- Steven Muchnick: *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997
- Randy Allen, Ken Kennedy: *Optimizing compilers for modern architectures*, Morgan Kaufmann, 2002
- Keith D. Cooper, Linda Torczon: *Engineering a Compiler*, Morgan Kaufmann, 2004

Literature

• D. Bacon, S. Graham and O. Sharp: Compiler Transformations for High-Performance Computing. ACM Computing Surveys, Volume 26, Issue 4, 1999

• R.C. Whaley, A. Petitet and J. Dongarra: ATLAS project, Parallel Computing, 2001

• S.L. Graham, P.B. Kessler, and M.K. McKusick: Gprof: A call graph execution profiler. Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, pages 120-126, June 1982

• T. Ball and J.R. Larus: Efficient Path Profiling, International Symposium on Microarchitecture, pages 46-57, 1996

• T. Ball, P. Mataga and M. Sagiv: Edge Profiling versus Path Profiling: The Showdown, In Symposium on Principles of Programming Languages, Jan. 1998

• B. Aarts, M. Barreteau, F. Bodin, P. Brinkhaus, Z.Chamski, H.-P. Charles, C. Eisenbeis,J. Gurd, J.Hoogerbrugge, P. Hu, W. Jalby, P.M.W. Knijnenburg, M.F.P O'Boyle, E. Rohou, R. Sakellariou, H. Schepers, A. Seznec, E.A. Stohr, M. Verhoeven and H.A.G. Wijshoff: OCEANS: Optimizing Compilers for Embedded Applications, in proceedings of EuroPar'97, LNCS-1300, pages 1351-1356, 1997

• F. Bodin, T. Kisuki, P. Knijnenburg,M. O'Boyle and E. Rohou: Iterative compilation in a non-linear optimisation space, in proceedings of the Workshop on Profile and Feedback Directed Compilation,1998

• K. D. Cooper, P. J. Schielke, and D. Subramanian: Optimizing for reduced code space using genetic algorithms, in proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pages 1–9, 1999

• G.G.Fursin, M.F.P.O'Boyle, and P.M.W. Knijnenburg: Evaluating Iterative Compilation, in proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC'02), College Park, MD, USA, pages 305-315, 2002

• K. D. Cooper, D. Subramanian, and L. Torczon: Adaptive optimizing compilers for the 21st century, journal of Supercomputing, 23(1), 2002

• G. Fursin: Iterative Compilation and Performance Prediction for Numerical Applications, Ph.D. thesis, University of Edinburgh, Edinburgh, UK, January 2004

Literature

• K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman: Acme: adaptive compilation made efficient, in proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pages 69–77, 2005

• B. Franke, M. O'Boyle, J. Thomson and G. Fursin: Probabilistic Source-Level Optimisation of Embedded Systems Software, in proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05), pages 78-86, Chicago, IL, USA, June 2005

• G. Fursin and A. Cohen: Building a Practical Iterative Interactive Compiler, in proceedings of the 1st International Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07), Ghent, Belgium, January 2007

• S. Triantafyllis, M. Vachharajani, N. Vachharajani and D. August: Compiler optimization-space exploration, in proceedings of the International Symposium on Code Generation and Optimization (CGO), pages 204–215, 2003

• P. Kulkarni, D. Whalley, G. Tyson and J. Davidson: Evaluating heuristic optimization phase order search algorithms, in proceedings of the International Symposium on Code Generation and Optimization (CGO'07), pages 157–169, March 2007

• G. Fursin, J. Cavazos, M.F.P. O'Boyle and O. Temam: MiDataSets: Creating The Conditions For A More Realistic Evaluation of Iterative Optimization, in proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007), Ghent, Belgium, January 2007

• B. Grant, M. Mock, M. Philipose, C. Chambers and S.J. Eggers: DyC: An Expressive Annotation-Directed Dynamic Compiler for C, Theoretical Computer Science, volume 248, number 1-2, pages 147-199, 2000

• M.Mock, C. Chambers and S.J.Eggers: Calpa: A Tool for Automating Selective Dynamic Compilation, International Symposium on Microarchitecture, pages 291-302, 2000

• K. Ebcioglu and E.R. Altman: DAISY: Dynamic Compilation for 100% Architectural Compatibility, ISCA, pages 26-37, 1997

• V. Bala, E. Duesterwald and Sanjeev Banerjia: Dynamo: A Transparent Dynamic Optimization System, ACM SIGPLAN Notices, 2000

• C. J. Krintz, D. Grove, V. Sarkar and Brad Calder: Reducing the overhead of dynamic compilation, Software Practice and Experience, volume 31, number 8, pages 717-738, 2001

• M.J. Voss and R. Eigenmann: ADAPT: Automated de-coupled adaptive program transformation, in proceedings of ICPP, 2000

Literature

• G. Fursin, A. Cohen, M.F.P. O'Boyle and O. Temam: A Practical Method For Quickly Evaluating Program Optimizations, in proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005), number 3793 in LNCS, pages 29-46, Barcelona, Spain, November 2005

• J.Lau, M.Arnold, M.Hind and B.Calder: Online Performance Auditing: Using Hot Optimizations Without Getting Burned, in proceedings of PLDI, 2006

• G. Fursin, C. Miranda, S. Pop, A. Cohen and O. Temam: Practical Run-time Adaptation with Procedure Cloning to Enable Continuous Collective Compilation, in proceedings of the GCC Developers' Summit, Ottawa, Canada, July 2007

• C. Lattner and V. Adve: Llvm: A compilation framework for lifelong program analysis & transformation, in proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, March 2004

• A. Monsifrot, F. Bodin, and R. Quiniou: A machine learning approach to automatic production of compiler heuristics, in proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications, LNCS 2443, pages 41–50, 2002

• M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly: Meta optimization: Improving compiler heuristics with machine learning, in proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03), pages 77–90, June 2003

• S. Long, M.F.P. O'Boyle: Adaptive Java optimisation using instance-based learning, in proceedings of ICS, 2004

• J. Cavazos, J.E.B.Moss, M.F.P.O'Boyle: Hybrid Optimizations: Which Optimization Algorithm to Use? in proceedings of CC, 2006

Literature

• F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint and C.K.I. Williams: Using Machine Learning to Focus Iterative Optimization. in proceedings of the 4th Annual International Symposium on Code Generation and Optimization (CGO), New York, NY, USA, March 2006

• John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F.P.O'Boyle and Olivier Temam: Rapidly Selecting Good Compiler Optimizations using Performance Counters, in proceedings of the 5th Annual International Symposium on Code Generation and Optimization (CGO), San Jose, USA, March 2007

• Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael O'Boyle and Oliver Temam: Enabling fast compiler optimization evaluation via code-features based performance predictor, in proceedings of the ACM International Conference on Computing Frontiers, Ischia, Italy, May 2007

• Grigori Fursin, Cupertino Miranda, Sebastian Pop, Albert Cohen and Olivier Temam. Practical Run-time Adaptation with Procedure Cloning to Enable Continuous Collective Compilation. Proceedings of the GCC Developers' Summit, Ottawa, Canada, July 2007

• Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, Francois Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle. MILEPOST GCC: machine learning based research compiler. Proceedings of the GCC Developers' Summit, Ottawa, Canada, June 2008

• Grigori Fursin and Olivier Temam. Collective optimization. To appear at the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009), Paphos, Cyprus, January 2009

Related Conferences

- Conference on Programming Language Design and Implementation (**PLDI**)
- International Conference on Code Generation and Optimization (**CGO**)
- Architectural Support for Programming Languages and Operating Systems (**ASPLOS**)
- Conference on Parallel Architectures and Compilation Techniques (**PACT**)
- International Conference on Compilers, Architecture and Synthesis for Embedded Systems (**CASES**)
- Symposium on Principles of Programming Languages (**PoPL**)
- Principles and Practice of Parallel Computing (**PPoPP**)
- International Symposium on Microarchitecture (**MICRO**)
- International Symposium on Computer Architecture (**ISCA**)
- Symposium on High-Performance Computer Architecture (**HPCA**)
- Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion (**SMART**)

Related Journals

- ACM Transaction on Architecture and Code Optimization
- IEEE Transaction on Computers
- ACM Transactions on Computer Systems
- ACM Transactions on Programming Languages and Systems
- IEEE Transaction on Parallel and Distributed Systems
- IEEE Micro

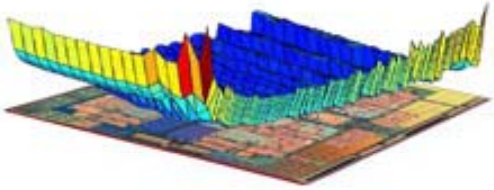
Miscellaneous

Machine Learning for Embedded Programs Optimisation *(MILEPOST)*

http://www.milepost.eu

Building intelligent self-tuning systems

http://unidapt.org



UNIDAPT GROUP

Network of Excellence on High Performance Embedded Architectures and Compilers *(HiPEAC)*

http://www.hipeac.net

Thanks

Thanks to Prof. Michael O'Boyle from the University of Edinburgh for providing some slides from his course on iterative feedback-directed compilation (2005)

Contact email:
grigori.fursin@inria.fr

More information about research projects and software:
http://fursin.net/research

Lecture and publications on-line:
http://fursin.net/research_teaching.html