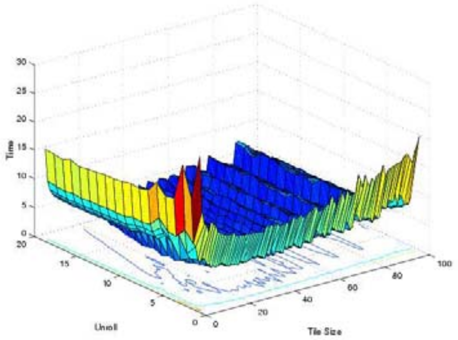
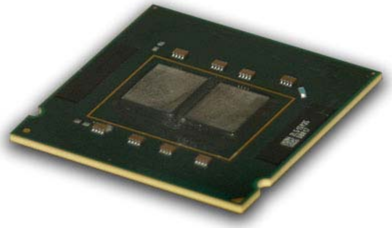


Adaptive and feedback driven compilation and optimization

Grigori Fursin

Alchemy group, INRIA Futurs, France



My background

- *Ph.D. degree from the University of Edinburgh, UK (1999 - 2004)*

  Program iterative optimizations and performance prediction

- *Research scientist at INRIA Futurs, France (2004 …)*

  Iterative feedback directed compilation
  Run-time adaptation and optimization
  Machine learning
  Architecture design space exploration

- *Collaborations:*

  IBM, NXP, STMicro, ARC, ARM, CAPS Enterprise
  University of Edinburgh
  Universitat Politechinca de Catalunya (UPC)
  University of Illinois at Urbana-Champaign (UIUC)

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

- *Motivation*
- *Background*
- *Feedback directed compilation and optimization*
- *Dynamic compilation and optimization*
- *Machine learning and future directions*

Motivation

Are compilers important?

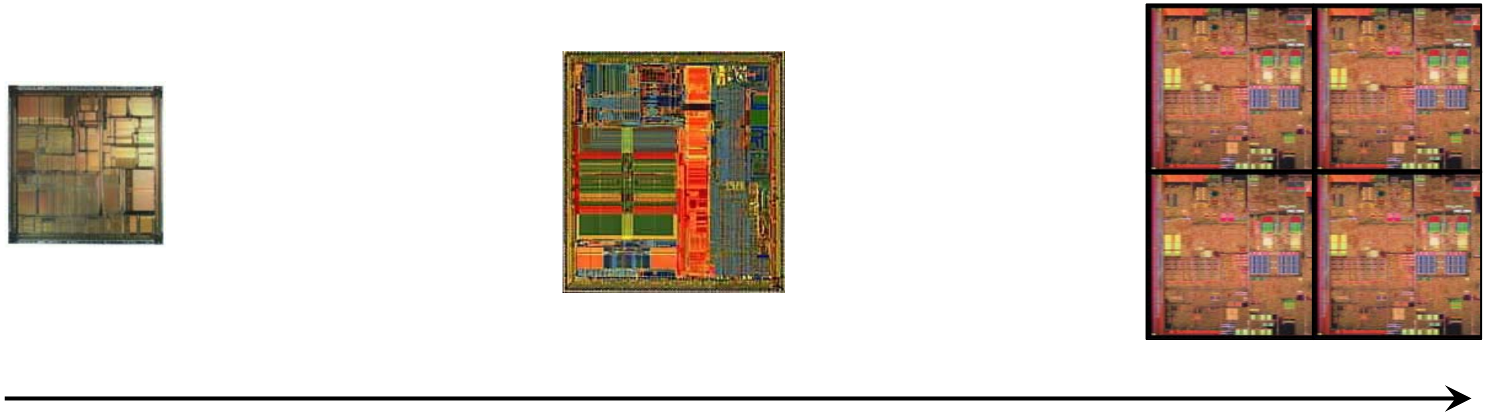
Motivation

Current innovations in science and industry demand ever-increasing computing resources while placing strict requirements on ***system performance, power consumption, size, response, reliability, portability and design time***.

Motivation

Current innovations in science and industry demand ever-increasing computing resources while placing strict requirements on ***system performance, power consumption, size, response, reliability, portability and design time***.

High-performance computing systems rapidly evolve toward ***complex heterogeneous multi-core systems***

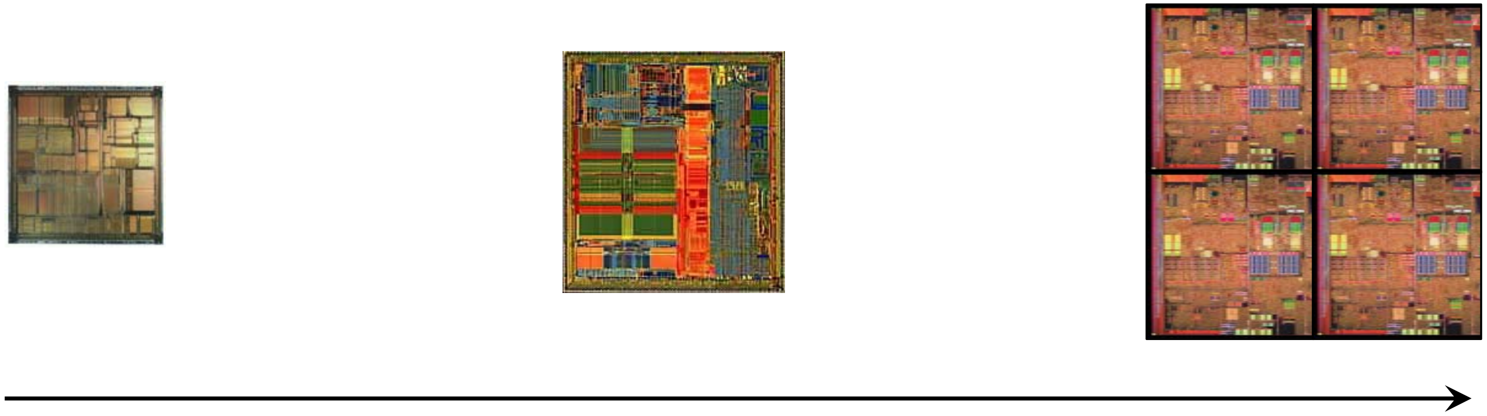


dramatically increased optimization time

Motivation

Current innovations in science and industry demand ever-increasing computing resources while placing strict requirements on ***system performance, power consumption, size, response, reliability, portability and design time***.

High-performance computing systems rapidly evolve toward ***complex heterogeneous multi-core systems***



dramatically increased optimization time

Optimizing compilers play a key role in ***producing executable codes quickly and automatically*** while satisfying all the above requirements for a broad range of programs and architectures.

Motivation

Is it easy?

What are the challenges?

Motivation

Is it easy?

What are the challenges?

Before answering these questions we need to look at the basics of the current compilers

Compiler background

- Compilers translate user programs to machine code
- Translation must be correct
- Needed to hide machine complexity
- Compilers need to optimize code to satisfy various requirements
- Compilers automatically translate. Can we automate compiler construction?
- Compilers generating compilers exits - GCC, CoSy
- Automatic construction of compiler optimization is very challenging

Compiler background

Some current popular static optimizing compilers for Linux:

- GCC (GNU Compiler Collection)

  http://gcc.gnu.org

- Open64

  http://www.open64.net

- Intel Compilers

  http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284264.htm

- PathScale Compilers

  http://www.pathscale.com

Compiler structure

- Compiler structure changed little since 1950s: consists of a linear sequence of passes

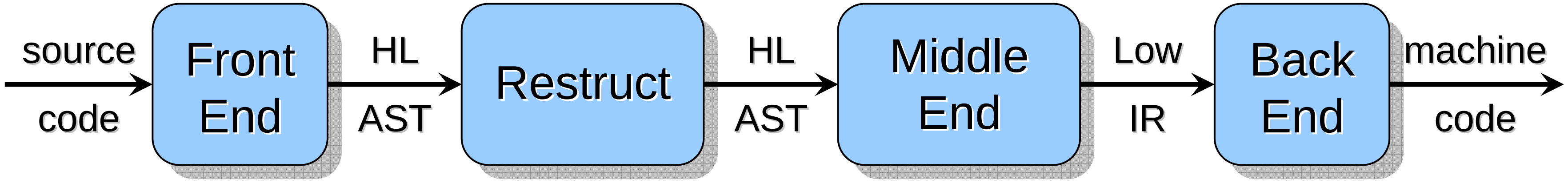
  - Lexical Analysis: Finds and verifies basic syntactic items, lexems, tokens using finite state automata

  - Syntax Analysis: Checks tokens following a grammar and builds an Abstract Syntax Tree (AST)

  - Semantic Analysis: Checks that all names are consistently used and builds a symbol table

  - Code optimization and generation: Optimize code using different intermediate formats (IR) and generate machine instructions for a specific architecture while keeping the meaning of the program

Compiler structure



- Front End translates “strings of characters” into a structured High Level Abstract Syntax Tree (AST)

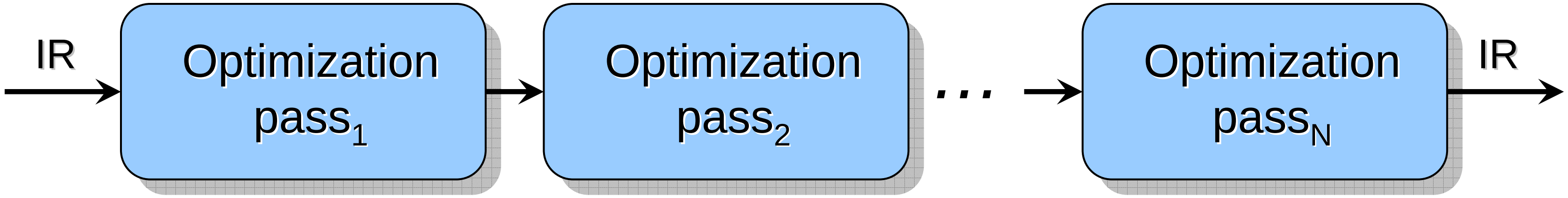
- Restructurer and Middle End performs machine independent optimizations including “source-to-source transformations” and outputs a Lower Level Intermediate Representation (IR)

    - Can be several IRs to simplify program anlsysis, optimizations and code generation

    - Many choices for IR (affect form and strength of program analysis and optimizations)

- Back End generally performs machine code generation including instruction scheduling and register allocation

Optimizer structure



Many optimization passes (*inlining; dead code elimination; constant propagation; loop transformations including loop tiling, interchange, fusion-fision, vectorization, unrolling; automatic parallelization, etc*) with the fixed linear order

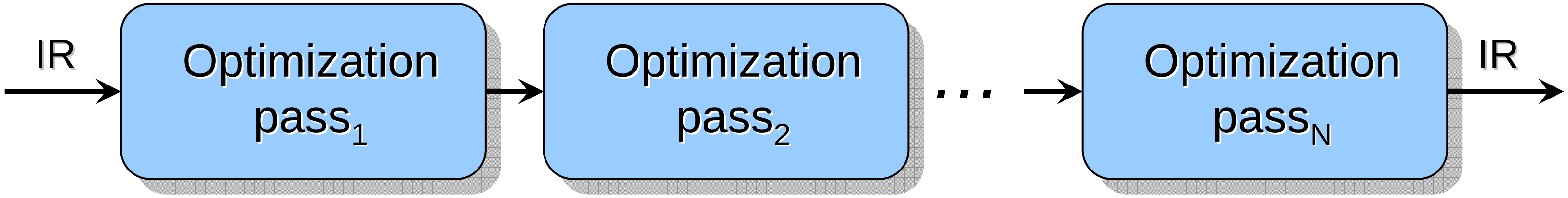
Optimization passes can be often turned on and off using compiler command line flags

Passes are generally applied to either the whole program (Inter-Procedural Optimizations) or at a function (procedure) level.

Transformations within passes are often applied on a loop or basic-block level with the fixed linear order and can be parametric

Some transformations can be selected by compiler command line flags but optimization heuristic is often hidden from the user

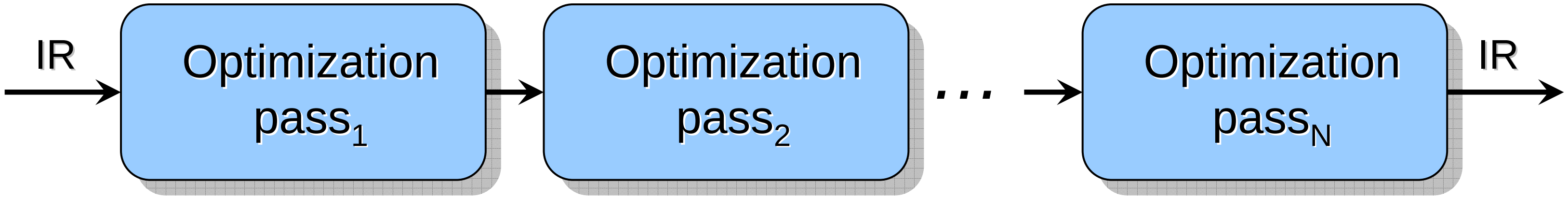
Optimizer structure



Is this working well?

(DEMO$_1$)

Optimizer structure

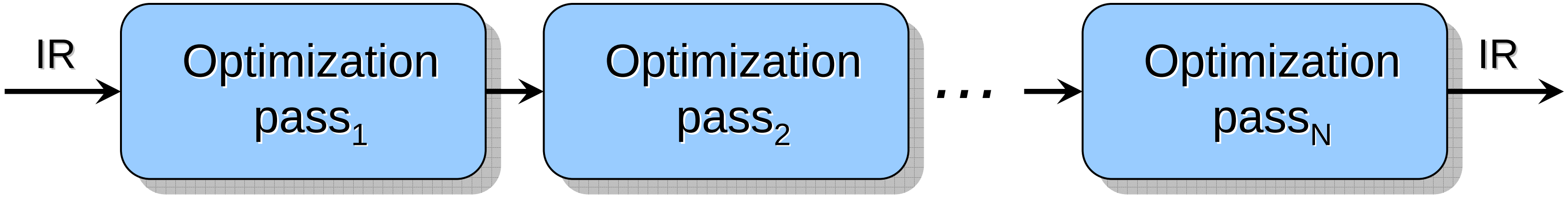


Matmul benchmark and GCC 4.2.x compiler:

1) gcc -O3 -funroll-loops matmul.c [matrix size 160x160]

Using funroll-loops over default -O3 optimization level gives around 15% improvement in execution time on x86 architecture

Optimizer structure



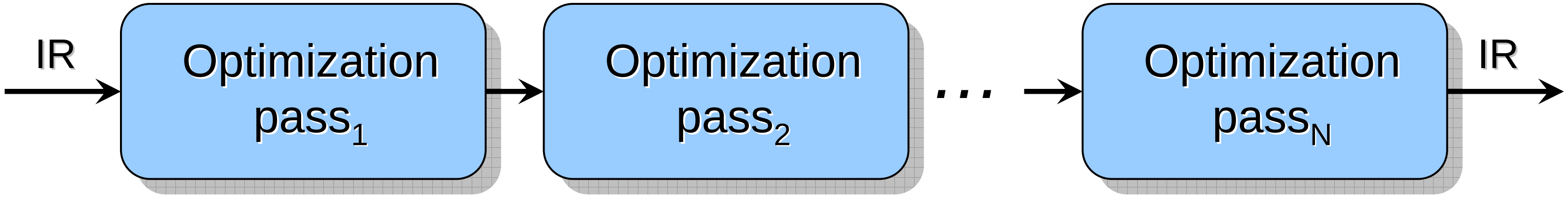
Matmul benchmark and GCC 4.2.x compiler:

1) gcc -O3 -funroll-loops matmul.c [matrix size 160x160]

Using funroll-loops over default -O3 optimization level gives around 15% improvement in execution time on x86 architecture

Wow! Found good compiler flag! Let’s use it all the time!

Optimizer structure



Matmul benchmark and GCC 4.2.x compiler:

1) gcc -O3 -funroll-loops matmul.c [matrix size 160x160]

Using funroll-loops over default -O3 optimization level gives around 15% improvement in execution time on x86 architecture

Wow! Found good compiler flag! Let’s use it all the time!

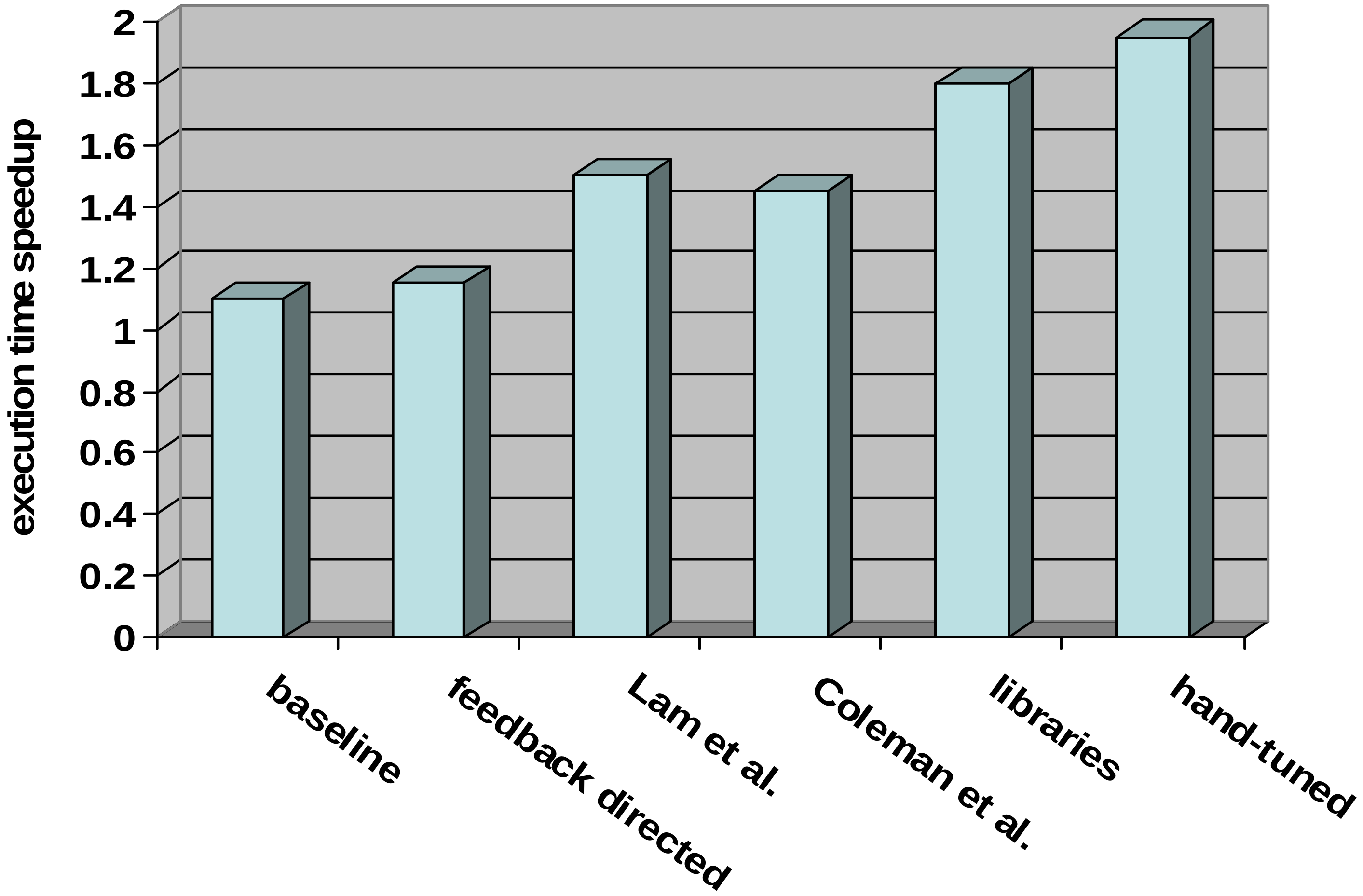
2) gcc -O3 -funroll-loops matmul.c [matrix size 3x3]

Using funroll-loops over default -O3 optimization level degrades performance by about 10%

So, selecting this flag is not always good!

Room for improvement?

This problem is not new (40+ years)



(Optimizing matrix multiply code)

Challenges

- Optimizer has to exploit all architectural features
  - Instruction and thread level parallelism
  - Effective management of memory hierarchy (registers, caches, memory, disk)
- Optimization at many levels: source, internal formats, assembler
- Optimization at many scopes: (whole program, function/procedure, loop, basic block)
- Which optimizations to use?
- What is the best order of optimizations?
- How to select right transformation parameters?
- What if transformation parameters depend on run-time information?

Challenges

Machine dependent optimizations vs. independent optimizations

Optimizations typically split into those that are always worthwhile and machine specific

Challenges

Machine dependent optimizations vs. independent optimizations

Optimizations typically split into those that are always worthwhile and machine specific

Example: Common sub-expression elimination

Aim: prevent redundant recalculation of terms

```
a = b + c + f          t = b + c
d = b + c + e          a = t + f
                       d = t + e
```

Seems always like a good idea: 4 adds vs. 3

Challenges

Machine dependent optimizations vs. independent optimizations

Optimizations typically split into those that are always worthwhile and machine specific

Example: Common sub-expression elimination

Aim: prevent redundant recalculation of terms

```
a = b + c + f          t = b + c
d = b + c + e          a = t + f
                       d = t + e
```

Seems always like a good idea: 4 adds vs. 3

However: potentially additional variable - pressure on register allocation!

Challenges

Machine dependent optimizations vs. independent optimizations

- Rapidly evolving architectural features strongly determine the best code sequence
- Rarely are all instructions of equal cost. Even if they have the same latency, not all function units support all functions.
- The more complex the hardware, the harder it is to determine the best code sequence
- Mixed multimedia instructions of different ISA for heterogeneous systems - which version to select?

Challenges

Classic optimization: Static analysis and transformation

- Statically (at compile time) analyze the program and transform it based on architectural features (such as ISA, memory hierarchy, etc) and requirements (such as reducing execution time or program size)

  Example of stride-1 access. Array C has row-major layout. Makes sense to traverse data row-wise.

```
for (i = 0; i<n; i++)

for (j = 0; j<n; j++)

    a[j][i] + b[i];
```

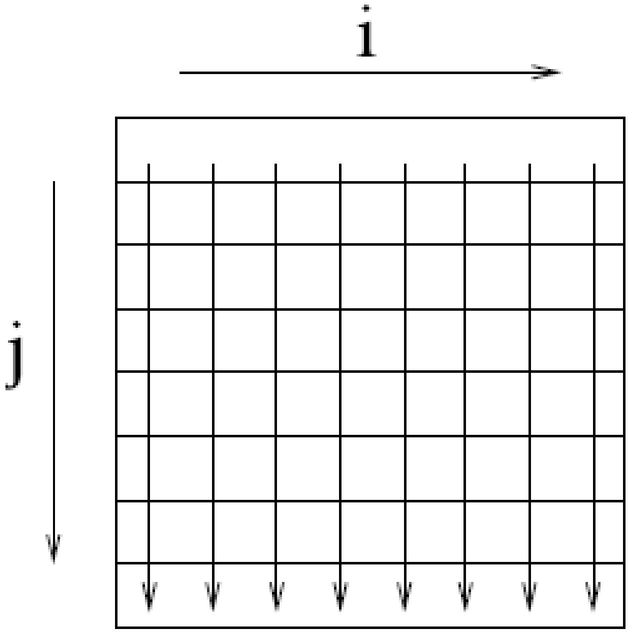
This code traverses the array column-wise

Does not exploit spatial locality. Can have excessive cache misses.

Challenges

Poor stride

```
for (i = 0; i<n; i++)
for (j = 0; j<n; j++)
        a[j][i] + b[i];
```



- Neighboring fetched elements not referenced until much later
- Cache line probably evicted by then

Challenges

Classic optimization: Static analysis and transformation

- Static analysis suggests that the innermost iterator should be in outermost subscript - should be transformed!
- Transform - apply code restructuring to achieve this - loop interchange

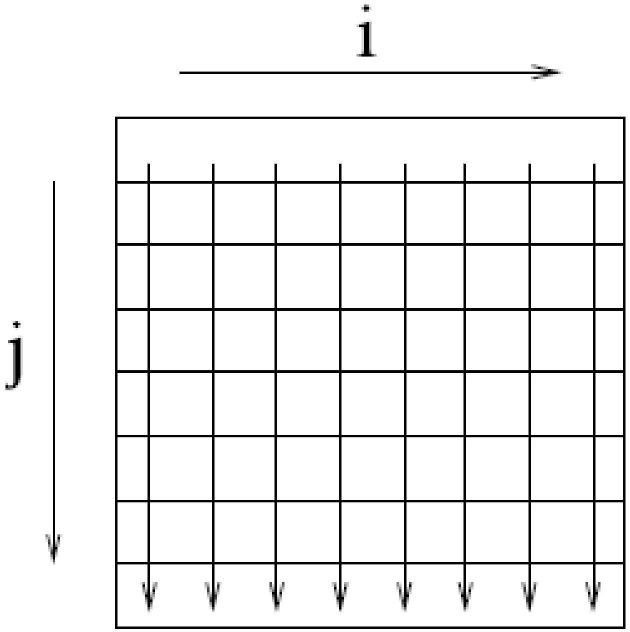
```
for (j = 0; j<n; j++)
for (i = 0; i<n; i++)
    a[j][i] + b[i];
```

- This code now traverses the array row-wise!
- Linear analysis and transformations can bring dramatic performance improvements

Challenges

Improved stride

```
for (j = 0; j<n; j++)
for (i = 0; i<n; i++)
      a[j][i] + b[i];
```



- Neighboring fetched elements referenced immediately
- Cache line unlikely to be evicted

Challenges

Classic optimization: Static analysis and transformation

- However does not consider other costs. i.e. b[i] is no longer invariant - temporal locality lost
- Uses idealized model of machine. No account of memory hierarchy, cache replacement policy etc.
- If any of this were to change, no way of changing the compiler
- Fundamentally each analysis has a small focused scope and hardware issue to reduce complexity.
- No theory/practice to integrate views.

Challenges

Some other transformations: Loop Unrolling

original loop:

```
do i = 1, n
  S1(i)
  S2(i)
  …
end do
```

unrolled loop (u - unroll factor):

```
do i = 1, n, u
   S1(i)
   S2(i)
   …
   S1(i+1)
   S2(i+1)
   …
   S1(i+u-1)
   S2(i+u-1)
   …
end do
```

loop body replicated u times

```
do j = i, n
   S1(j)
   S2(j)
   …
end do
```

processing all remaining elements

Which unrolling factor to choose?

Challenges

Some other transformations: Loop Tiling

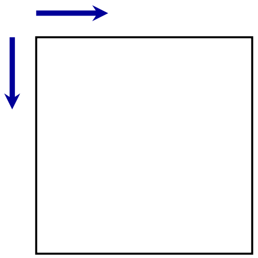
original loop nest:

```
do I = 1, N
  do J = 1, N
    A(I,J) = A(I,J) + B(I,J)
    C(I,J) = A(I-1,J) * 2
  end do
end do
```

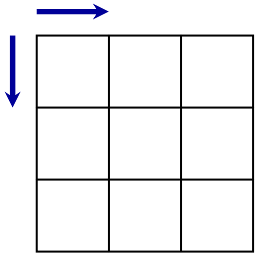
transformed loop nest:

```
do IT = 1, N, SS
  do JT = 1, N, SS
    do I = IT, MIN(N, IT+SS-1)
      do J = JT, MIN(N, JT+SS-1)
        A(I,J) = A(I,J) + B(I,J)
        C(I,J) = A(I-1,J) * 2
      end do
    end do
  end do
end do
```

iteration space of the original loop:



iteration space of the transformed loop:



Motivation

Current state-of-the-art compilers and optimizers often fail to deliver best performance on modern systems due to *fundamental reason of complexity and undecidability*

- lack of run-time information - impossible to know the best code sequence at compile-time
- simplistic hardware models for rapidly evolving processor architecture while its behavior with out-of-order execution and caches is non-deterministic
- long chain of optimization passes - difficult to predict best order, inevitably loss of information along the path
- fixed black-box optimization heuristics and inability to fine-tune applications
- inability to reuse optimization knowledge among different programs and architectures
- inability to adapt to varying program and system behavior at run-time

Motivation

Current compiler and optimization technologies should be revisited to keep pace with rapidly evolving hardware

Need static compilers that can continuously and automatically learn how to optimize programs, and have an ability to adapt at run-time for different behavior and constraints

Formalization of optimization

Compilation as Optimization

- Define “formal” optimization problem: minimize objective function over a space of options.

- Objective function is execution time, though code size, power and other constraints can be important.

- Optimization search space: all possible equivalent programs

- Objective function is undecidable in general

- Optimization space: infinite

Formalization of optimization

Intractability

- Solving an undecidable problem over an infinite space is clearly not feasible so simplification is necessary
- Traditionally have broken the problem into sub-problems based on certain assumptions
- Solve the problem by looking at each in isolation:
  - *Code generation* - determining the best code for an expression is NP
  - *Scheduling* - determining the best order of instruction is NP
  - *Register allocation* determining the best use of registers to minimize memory traffic is NP

Formalization of optimization

How to overcome?

Two main problems:

- *Complexity* of processor architecture, *undecidability* of program

Both problems arise from trying to optimize statically at compile time

- Have to *guess a tractable model*, have to *guess about data input*

- Pros and Cons to all approaches. Depends highly on application scenario

Formalization of optimization

Taxonomy:

2 main causes: program undecidability and processor complexity

- Variables (what): Program (P), Data (D) and Processor (proc)
- Variables (when): design, compile or runtime
- 2 sides of adaption: portability and specialization
- Examine all techniques in this light

Formalization of optimization

Taxonomy:

- Program (P), Data (D) and Processor (proc)
- time = f(T(P),D,proc), Pick Transformation T to minimize f
- Standard compilation (SC) typically has a hardwired model of proc built in
- SC also has an ad hoc view of typical programs (often biased by SPEC!) with a *compiler strategy* that is biased to them
- SC applies the strategy at compile time making no reference to data
- Data in no way affects SC behavior - just guess a “typical” input set

Formalization of optimization

Taxonomy:

Design time:

- Build a compiler: encode compiler optimization strategy. Typically a time consuming manual process. Takes many person-years. Particular to one processor, data and programs unknown

Compile time:

- Examine program and apply transformations based on design time encoded strategy. Can take a reasonable amount of time. Must be less than accumulated runtime throughout lifetime of program
- Processor assumed, program known, data unknown

Run-time:

- Most knowledge about application available: processor, program and data
- Least amount of time available to do anything about it!
- Typically compilers do nothing - leave to independent runtime system/OS

Formalization of optimization

Taxonomy: Adaptation = Portability + Specialization

Compiler technology not normally discussed in this manner.

Appears an infrastructure rather than optimization issue.

Portability:

• Ability to MODIFY behavior to changing circumstances, changing data, program, processor

Specialization:

• Ability to EXPLOIT fixed, known features: processor, program and data

Natural tension between the two: *flexibility vs rigidity*

Formalization of optimization

Taxonomy: current static compilers

- What and when to port/specialize:
  processor, program, data, design, compile, runtime

- Currently: specialize to processor at design time
  BUT cannot easily port to a new processor

- Portable across a wide range of programs and data
  at compile and runtime BUT

- Do not specialize to runtime data or program/processor interaction

- Very little exploitation of dynamic runtime knowledge/
  Adaption to changing processor or data not considered

Formalization of optimization

What are the ways to solve this problems?

Feedback directed compilation

- Profile feedback directed compilation
- Application tuning
- Iterative compilation
- Efficient searching
- Conclusion

Feedback directed compilation

Feedback directed (profile directed compilation)

- Directly addresses problem of compile time unknown data

- Key (simple) idea: run program once and collect some useful information

- Use this runtime information to improve program performance

- In effect move the first runtime info into the compile time phase

- Makes sense if gathering the profile data is cheap and user willing to pay for 2 compiles. Can still use after first compile.

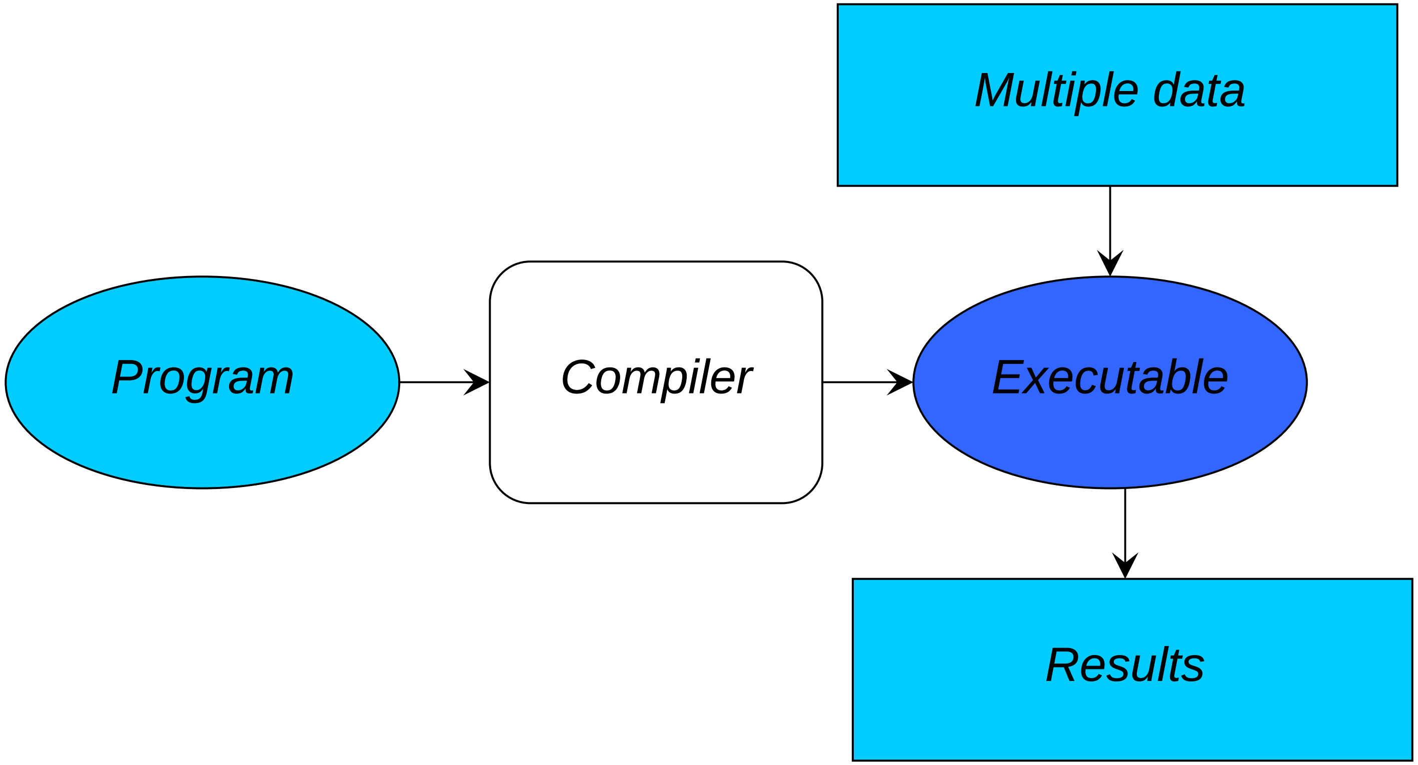
- Allows specialization to run-time data – what are pros and cons?

Feedback directed compilation

Off-line vs on-line compilation

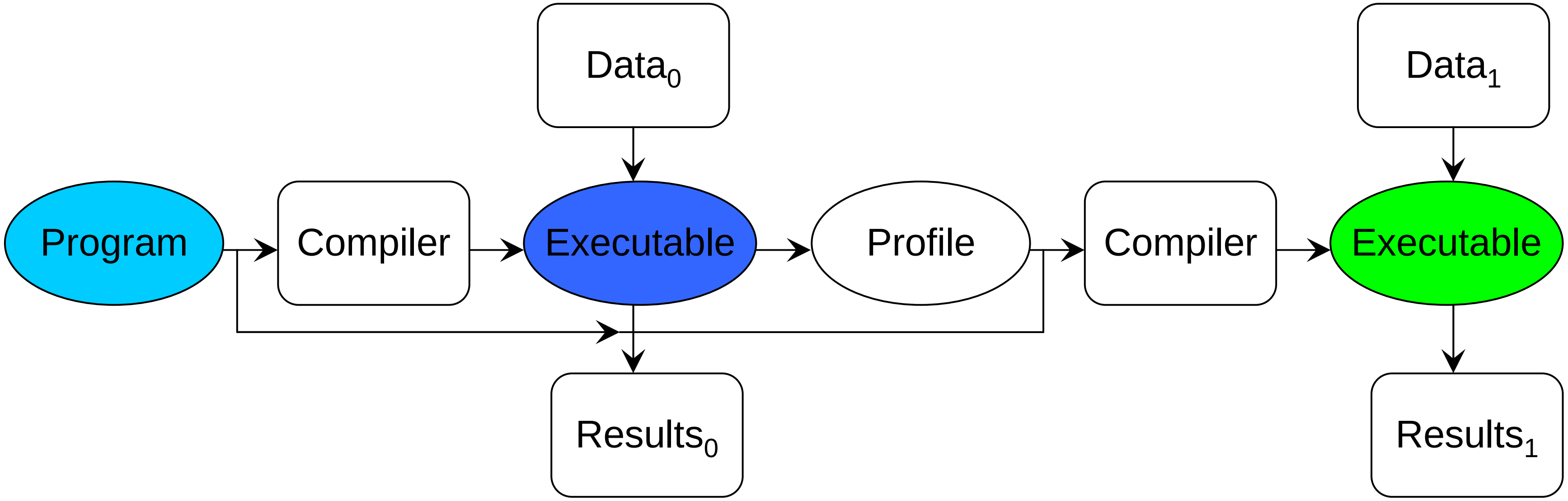
- Profile directed compilation is one example of off-line optimization
- Information is gathered and utilized before the “production” run
- On-line schemes gather information and dynamically change program as it runs.
- Off-line schemes work on basis that costs incurred at compile-time are outweighed by improved runtime. Can be more aggressive than on-line schemes.

Feedback directed compilation



Traditional compilation model

Feedback directed compilation



Profile information as an additional output

Data can change from run to run. Executable is still correct.

Feedback directed compilation

Brief history

- The use of profiling to aid program performance has been around for a while
- prof, gprof (1982). A tool to help developers to understand their code. Instrumentation at compile time and then sampled at runtime
- Hardware analysis (1980s). Monitor program behavior and adapt. Branch prediction - pipelines means need to guess which branch to take
- Edge/node based profile information for compilers 1990s
- Path based profiling Larus + Ball late 1990s, Smith 2000

Feedback directed compilation

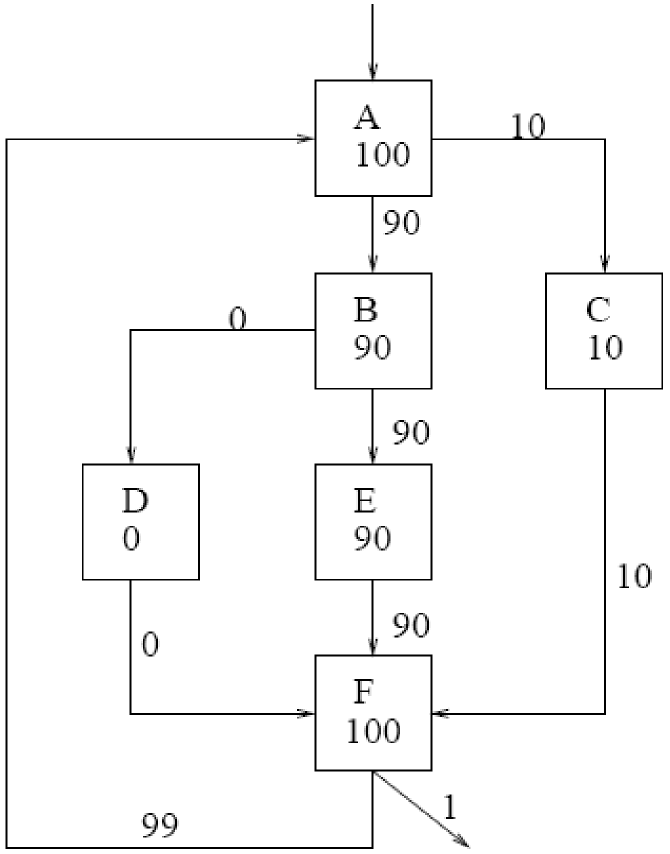
PDC for classic optimization

- Record frequently taken edges of program control-flow graph
- IMPACT compiler in 1990s good example of this but also used earlier - Josh Fisher et al, Multiflow.
- Use weight information of edges and paths in graph to restructure control-flow graph to enable greater optimization
- Main idea: merge frequently executed basic blocks increasing sizes of basic block if possible (superblock/hyperblock) formation. Fix up rest of code.
- Allows improved scheduling of instructions and more aggressive scalar optimizations at expense of code size

Feedback directed compilation

PDC example 1

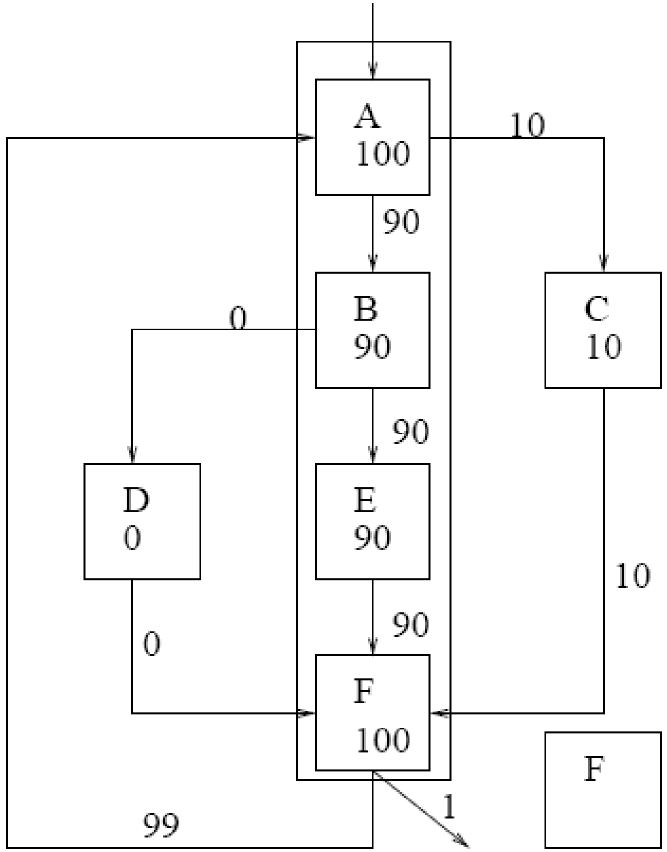
- Sequence of basic blocks
- Frequency of execution on edges and nodes
- Primarily ABEF
- Other entry/exit control-flow prevents merging
- Super-block -frequently executed path
- Merge and tidy-up
- Optimize larger unit



Feedback directed compilation

PDC example 1

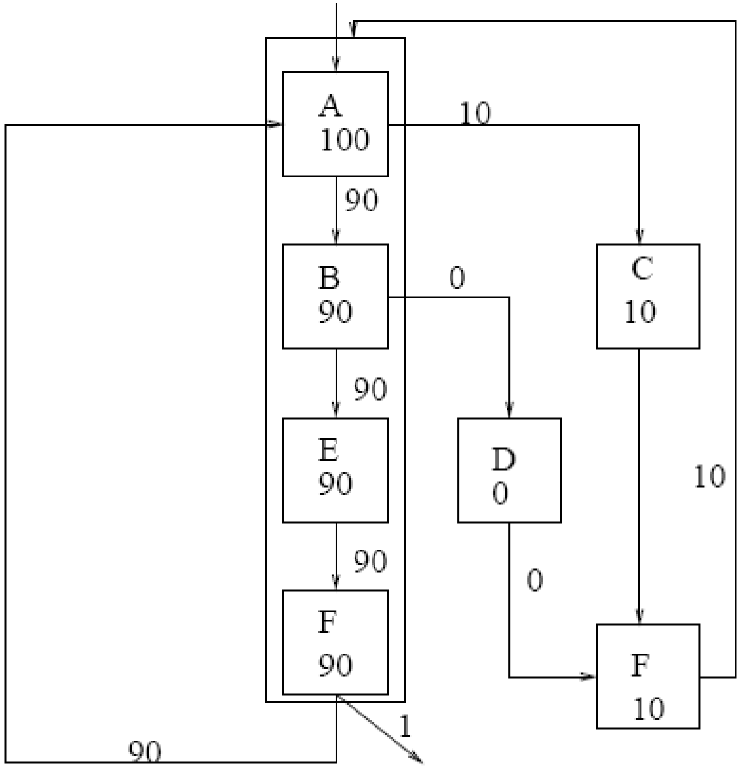
- Selecting the trace
- Start at most frequent block
- Add blocks on most frequent successors
- Repeat on other nodes
- Done in both control-flow directions
- Do on remaining nodes



Feedback directed compilation

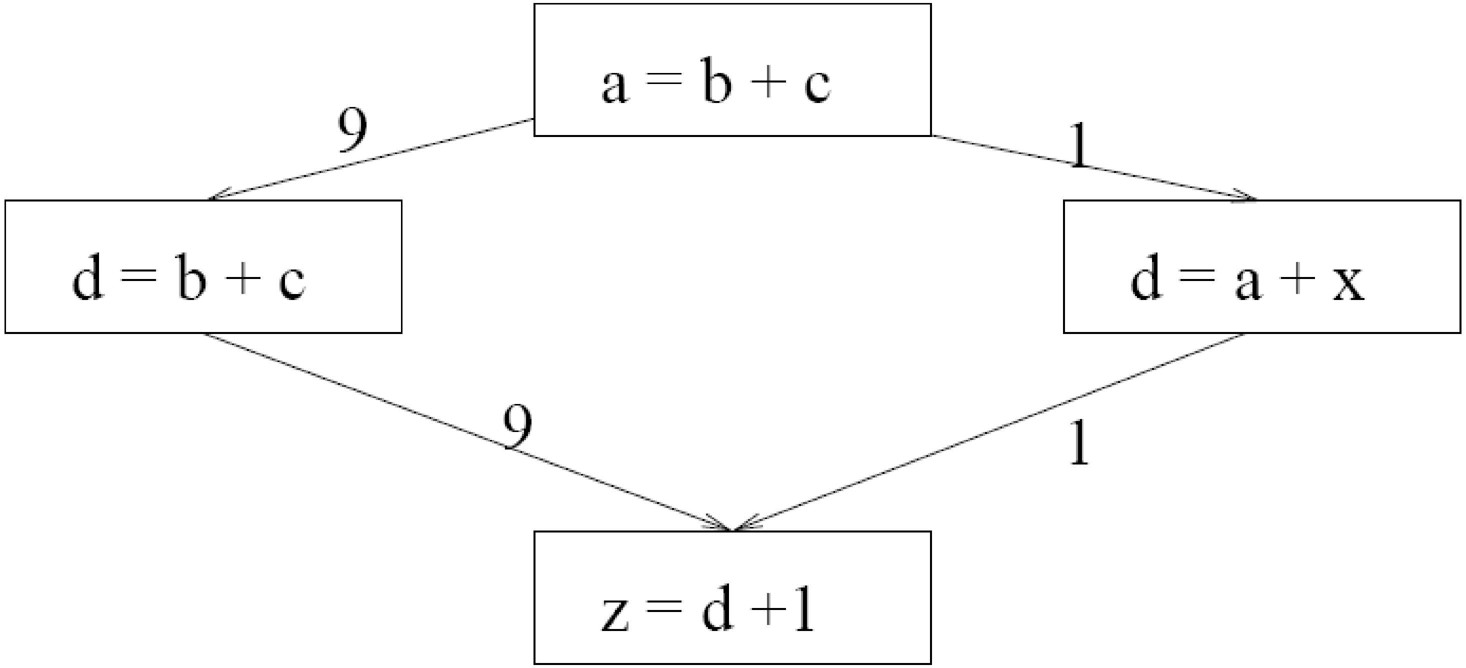
PDC example 1

- Tail Duplication
- Duplicate first block with external entry edges
- But not the head
- Redirect incoming edges
- Duplicate outgoing
- Repeat
- Much code duplication



Feedback directed compilation

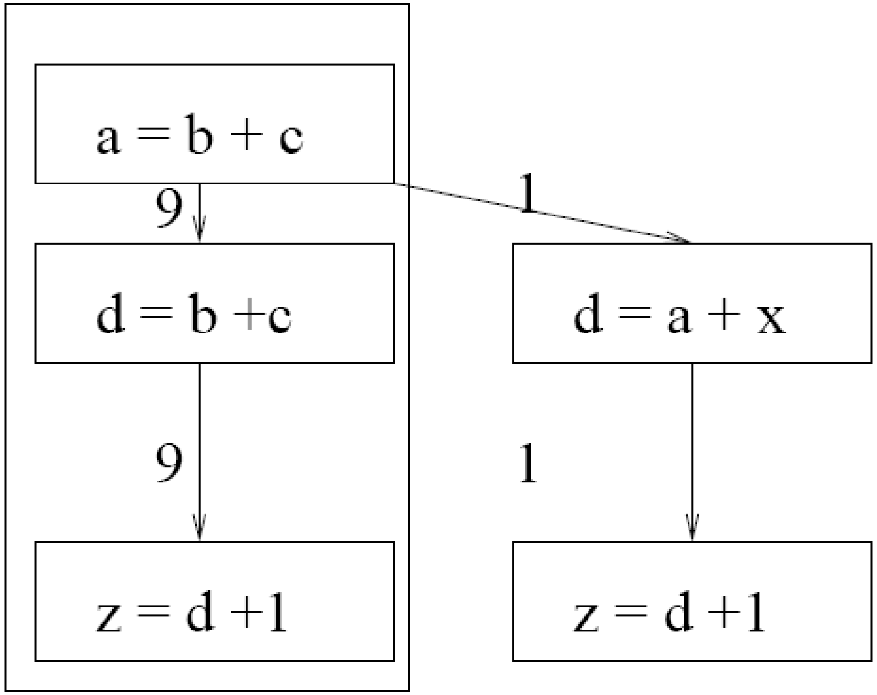
PDC example 2



Common b + c on frequently taken path

Feedback directed compilation

PDC example 2

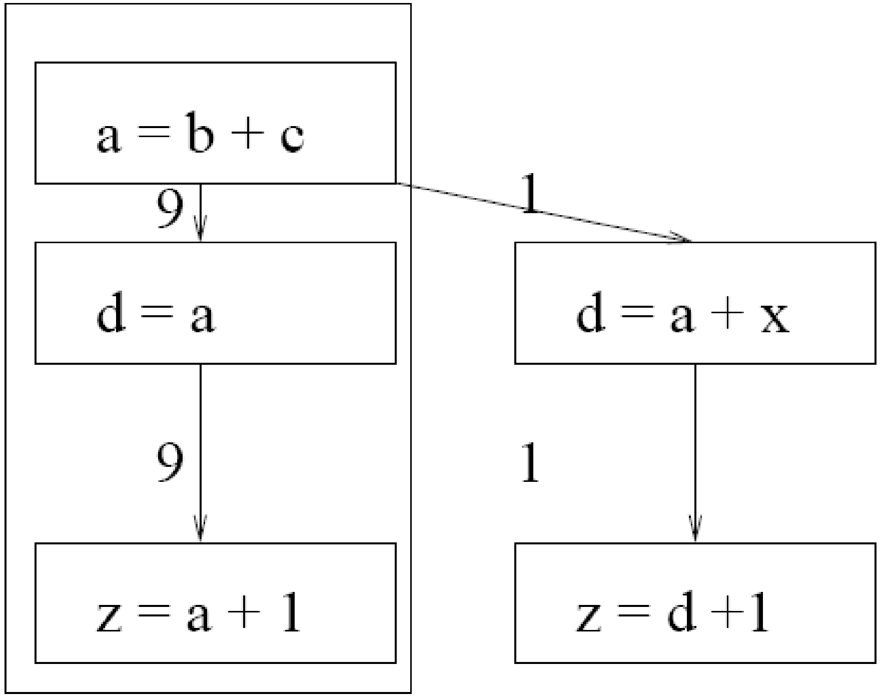


Replicate first node on main path with external incoming edge

Now separate paths

Feedback directed compilation

PDC example 2



Applying CSE eliminates redundant computation at cost of additional code

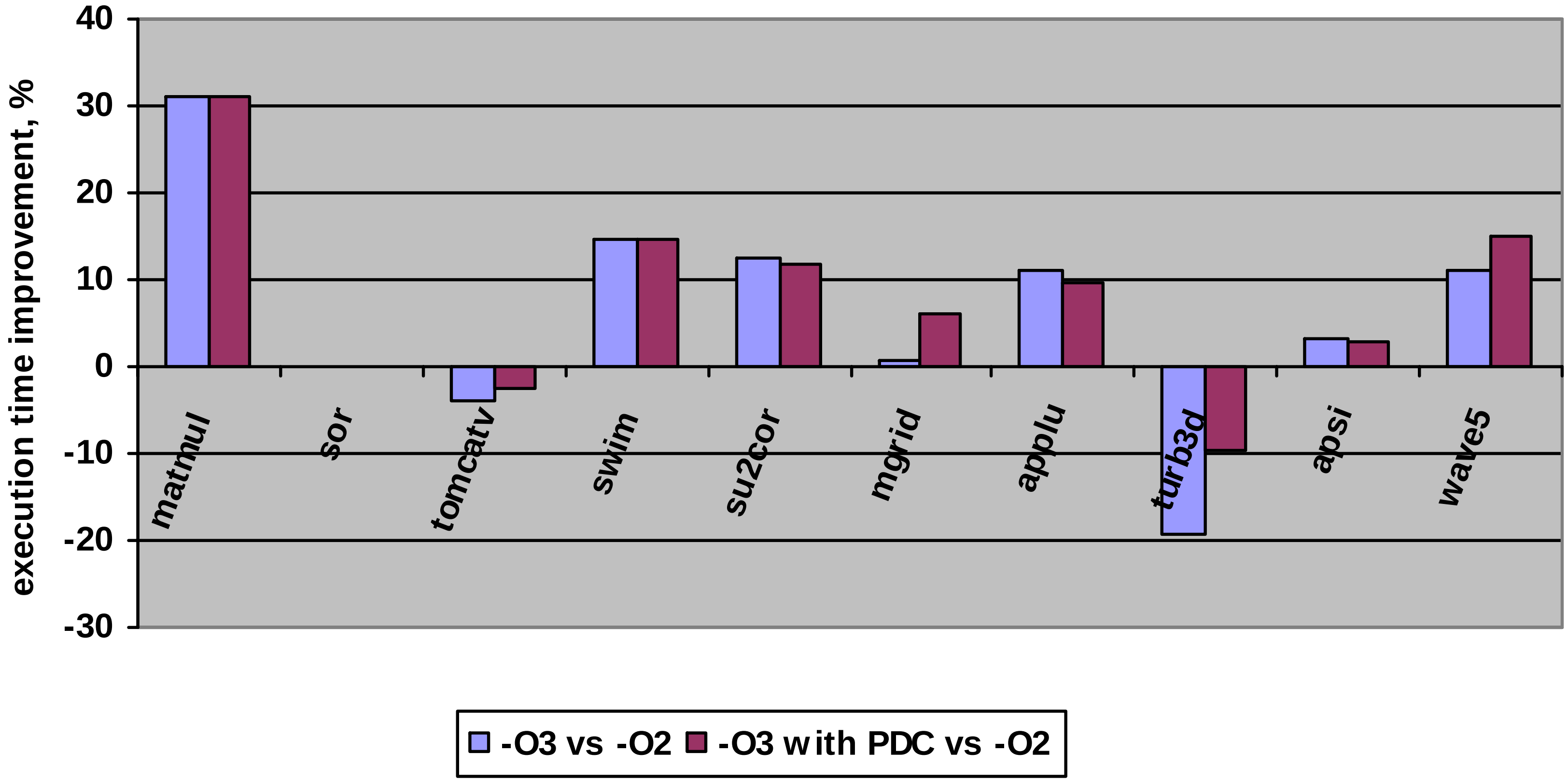
Feedback directed compilation

Edge vs Path profiling

- Overlapping paths cannot be distinguished by edge profiling
- Path profiling allows much greater accuracy
- However, combinatorial explosion in paths. Cycles in graphs leads to potentially unbounded number
- In practice Edge/node profiling only captures around 40-50
- Larus and Ball ’99 developed an efficient path profiler that avoids these problems. In practice the benefit achieved was small though
- Mike Smith at Harvard extended this idea for more targeted optimization

Feedback directed compilation

Some results when using PDC (Fursin'2002)

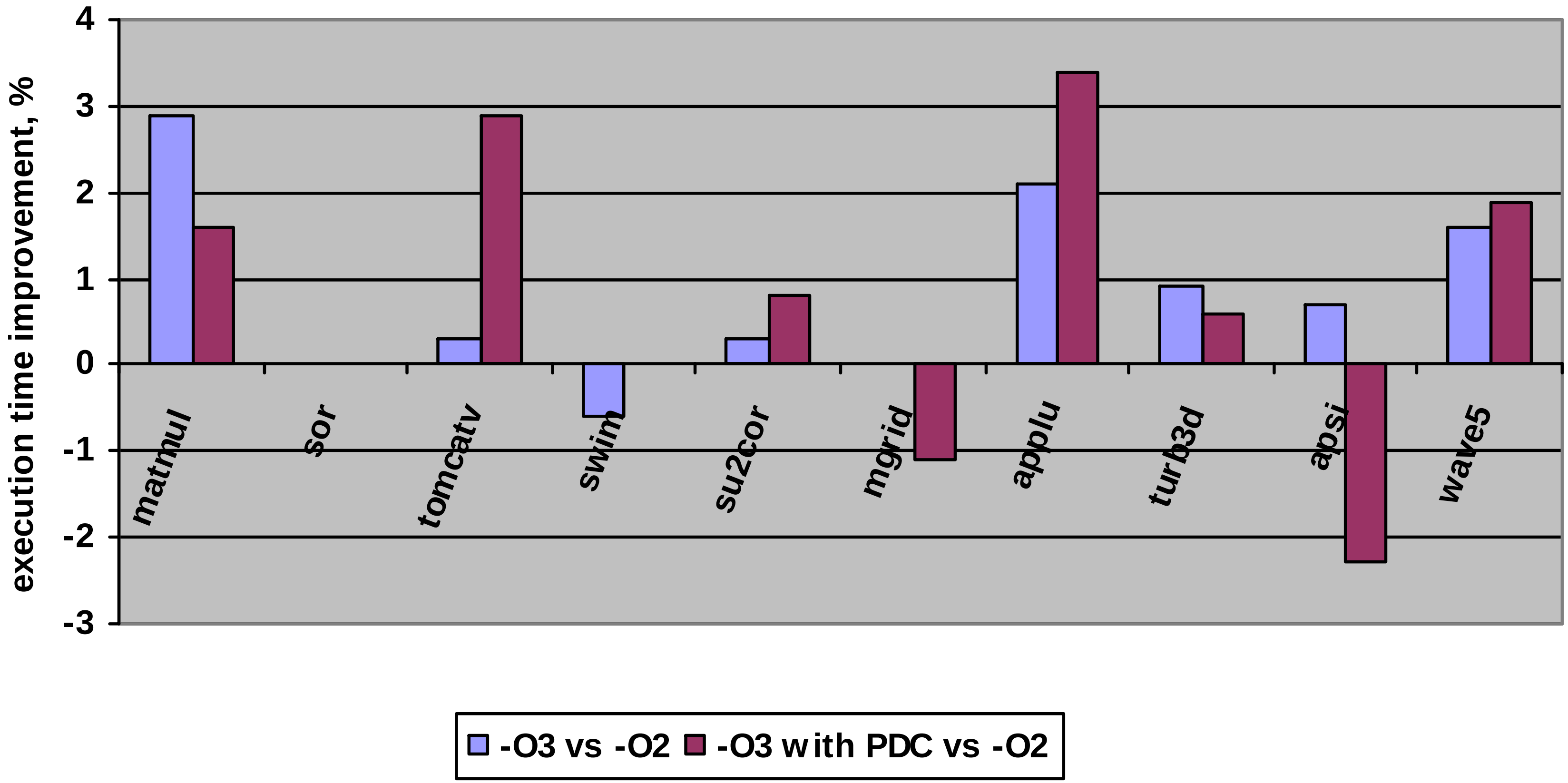


SPEC CPU95

Alpha compiler (21264)

Feedback directed compilation

Some results when using PDC (Fursin'2002)



SPEC CPU95

Intel Compiler (Pentium III) – poor improvement

Extremely well studied benchmarks

Feedback directed compilation

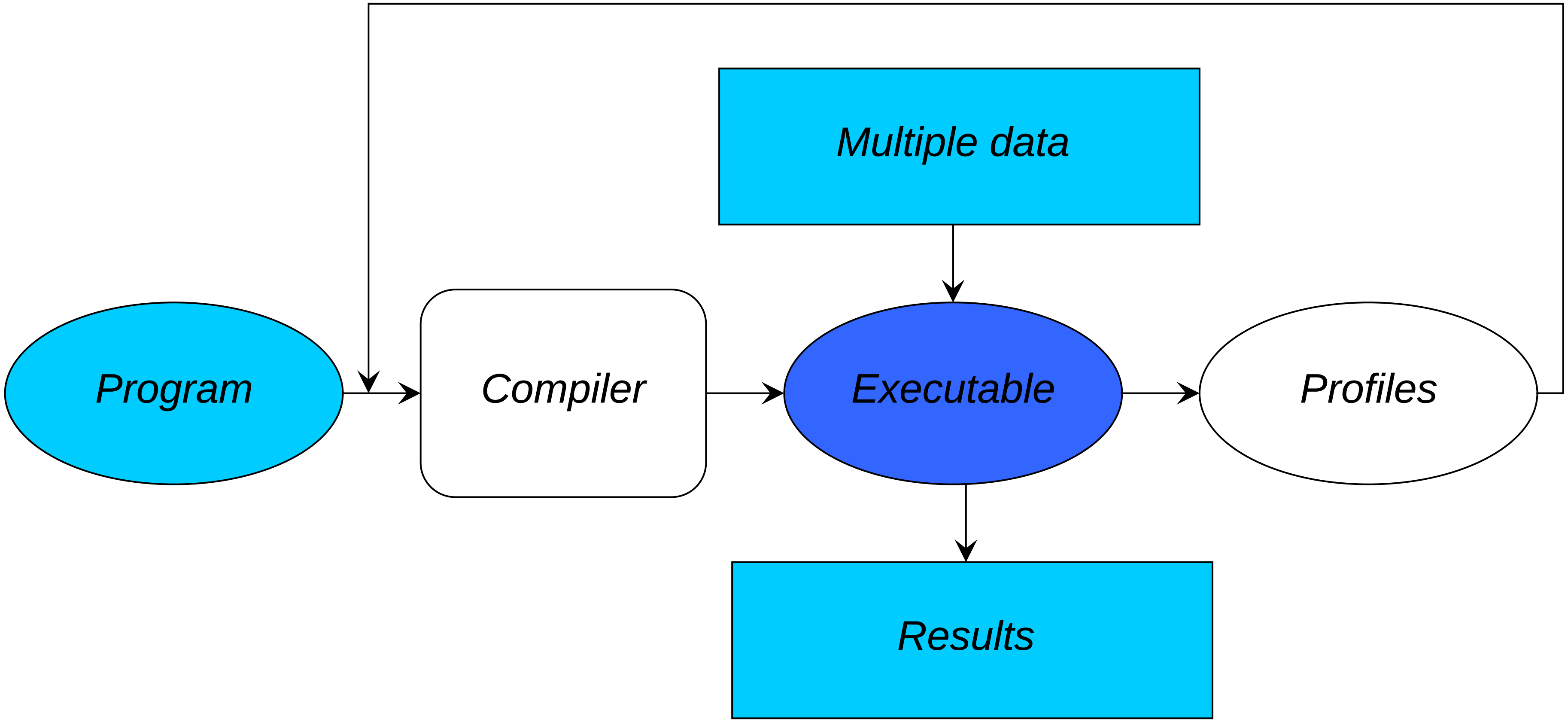
Beyond PDC

- Although useful, the performance gains are modest
- Challenge of undecidability and processor behavior not addressed.
- What happens if data changes on the second run?
- Really focuses on persistent control-flow behavior
- All other information i.e. run-time values, memory locations accessed are ignored

- Can we get more out of knowing data and its impact on program behavior?

Feedback directed compilation

Evolution of PDC



PDC with multiple (iterative) compiles

Feedback directed compilation

Automatic library tuning

- A different off-line approach that exploits knowledge gained by running the program in the optimization process
- There is a (growing) family of application specific approaches to library tuning
- Rather than recording path information for later optimization – just record execution time
- Try many different versions of the program and select the best for that machine. Key issue is how different programs are generated.
- In effect move run-time into design time.

Main examples ATLAS, PHiPAC and FFTW

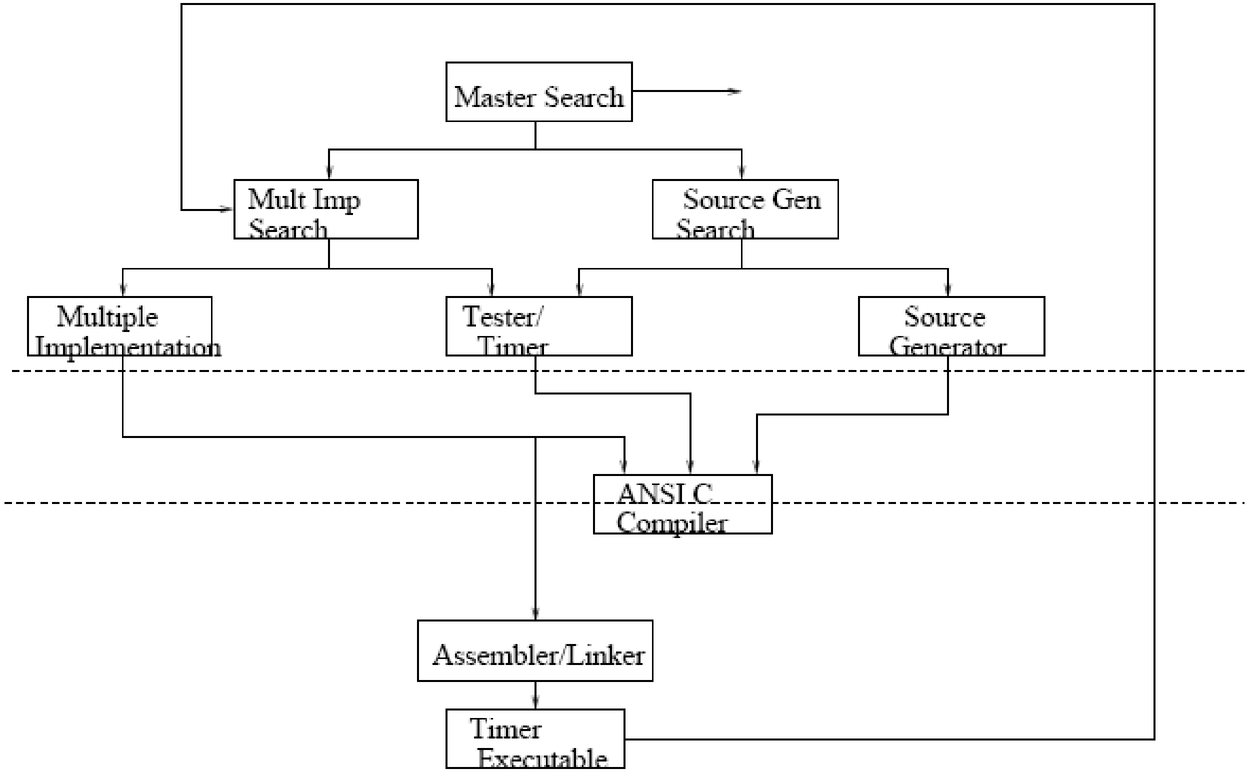
Feedback directed compilation

ATLAS

- An automatic method of tuning linear algebraic libraries for differing processors
- It is domain specific and only focuses on tuning the core GEMM routine for a specific processor.
- Takes an ad-hoc approach - generate different versions and measure them against anything available - including vendor supplied libraries and pick the best
- It tries different software pipelining and register tiling parameters and enumerates them all, selecting the best. The space of options is derived from explicit knowledge of the application behavior.

Feedback directed compilation

ATLAS



Broken down into application specific, generic and platform specific sections

Feedback directed compilation

ATLAS

- Regularly outperforms the best existing approaches. Now the standard approach to library generation.
- Adaption?: Very portable - works on any platform AND specializes to the particular processor
- BUT specialized to a particular application -no portability across programs no exploitation of runtime data as static control-flow
- PHiPAC tries to exploit data patterns in sparse structures by trying simple optimizations off-line and applying them at run-time when data encountered.
- However - domain specific, not generalizable or widely automatable

Feedback directed compilation

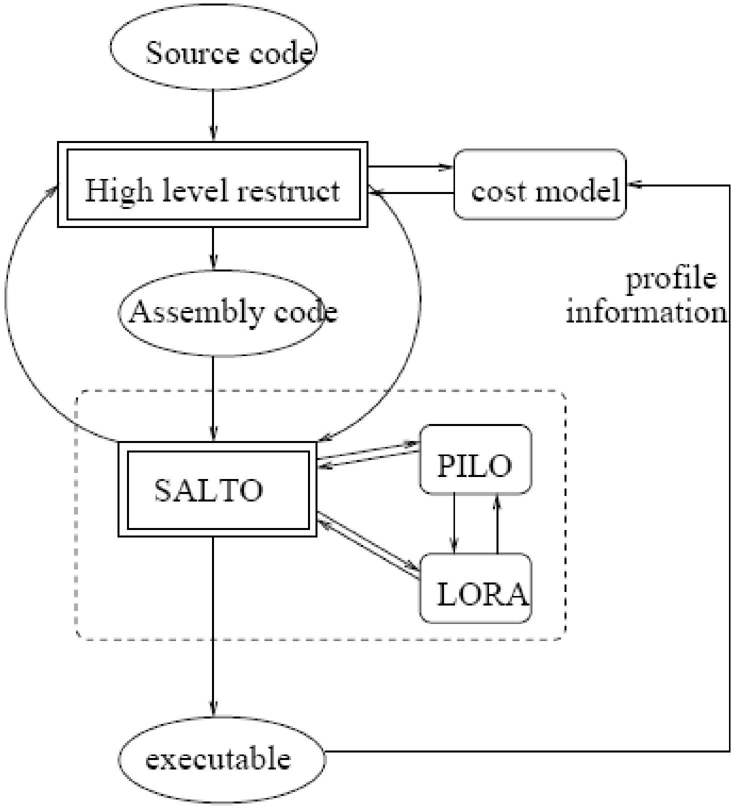
Iterative compilation

- Iterative compilation started in 1997 with the OCEANS project
- Similar in spirit to automatic tuning except the space of tuning is in fact the entire program transformation space
- In a sense it is direct implementation of the formal compiler optimization problem. Find transformation T that minimizes cost.
- Main ideas was to combine high and low level optimization and use cost models to guide selection
- Highly ambitious but immature infrastructure prevented much progress

Feedback directed compilation

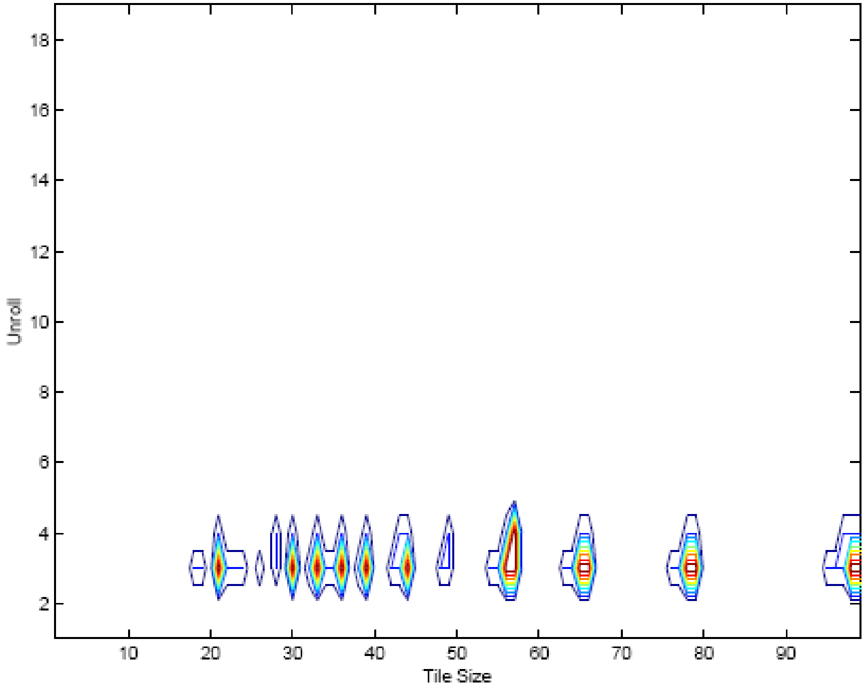
OCEANS

- Similar iterative structure to ATLAS
- Main work on searching for best tile and unroll parameters PFDC'98



Feedback directed compilation

matrix multiply, N=400, UltraSparc, exhaustive search

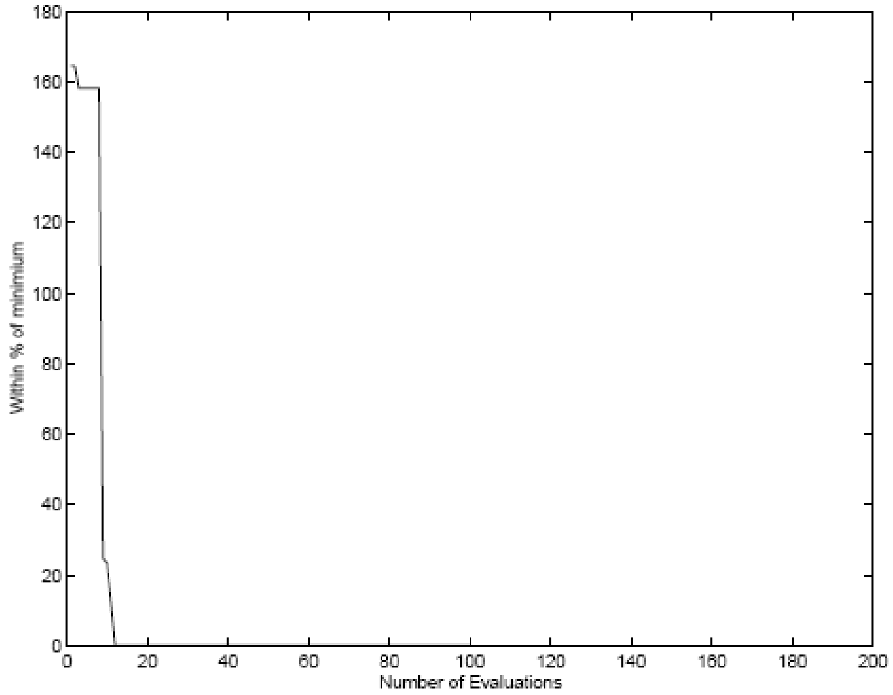


Minimum at: Unroll=3, Tile size=57

Near minimum: 2.6%, original 4.99 sec, minimum 0.56 sec

Feedback directed compilation

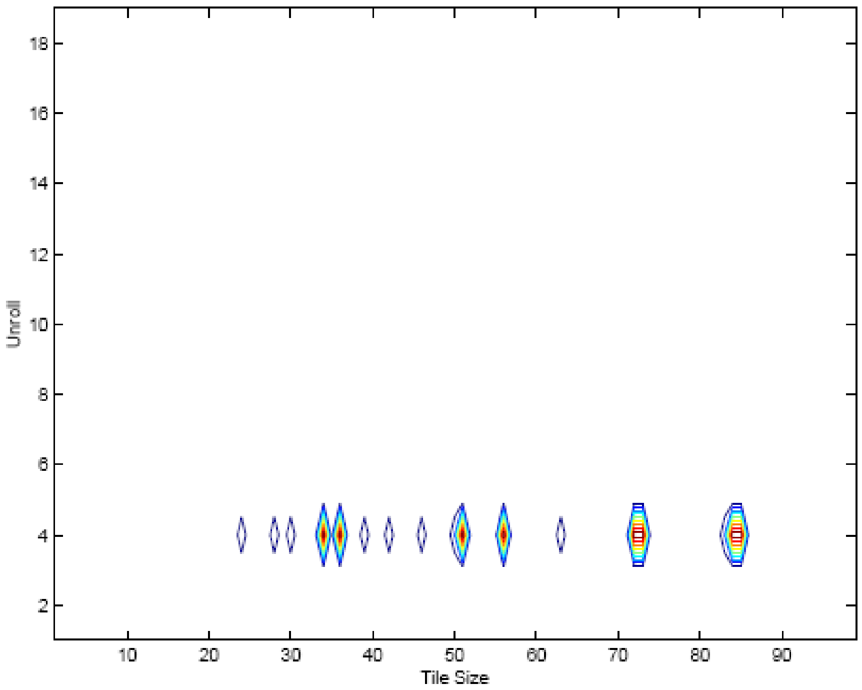
matrix multiply, N=400, UltraSparc, random search



50 steps: within 0.0%. Initially 2.65 times slower than minimum

Feedback directed compilation

matrix multiply, N=512, Alpha, exhaustive search

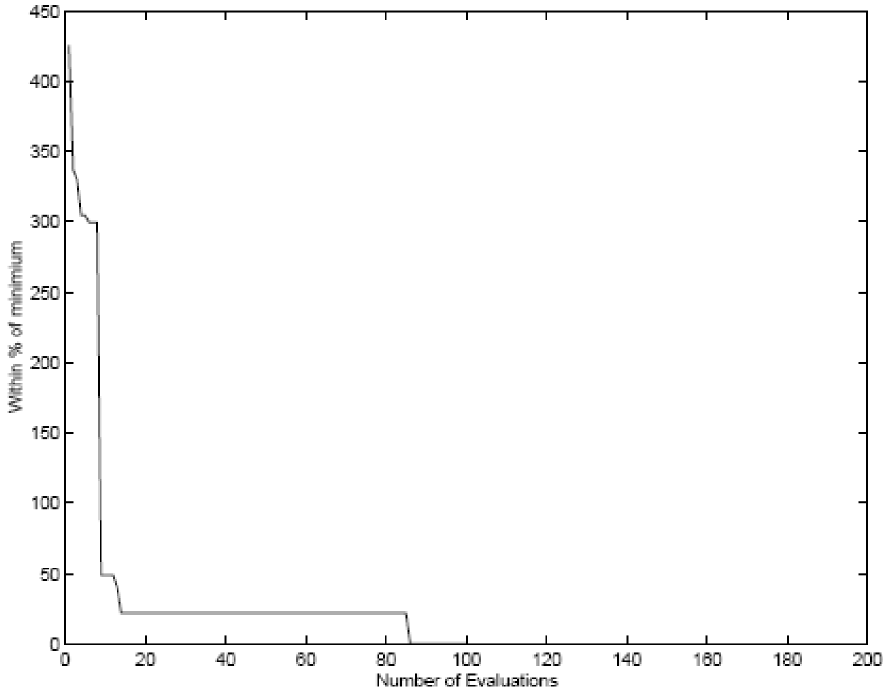


Minimum at: Unroll=4, Tile size=85

Near minimum: 0.9%, original 31.72 sec, minimum 3.34 sec, maximum 81.40 !

Feedback directed compilation

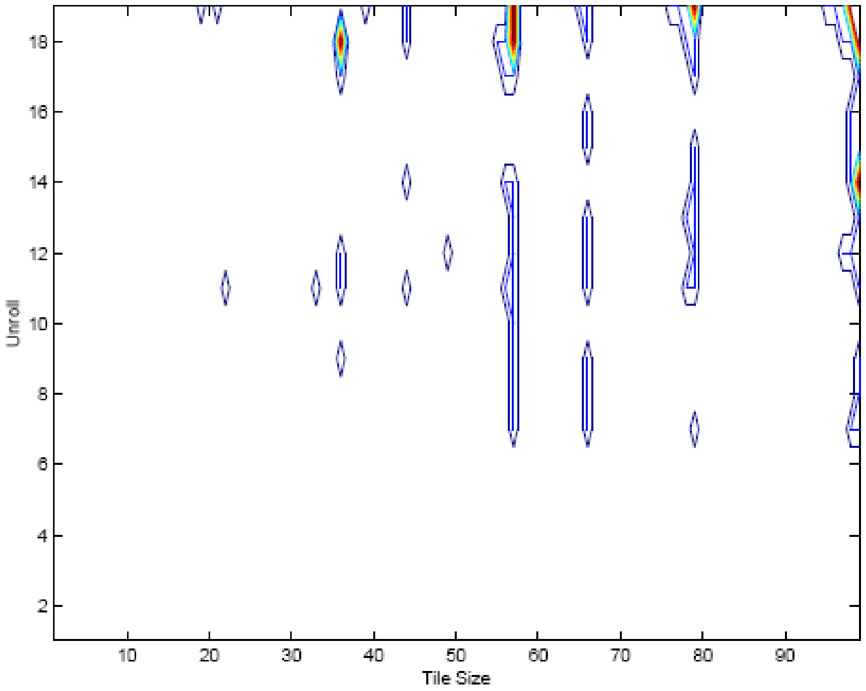
matrix multiply, N=512, Alpha, random search



50 steps: within 21.9%. Originally 5.25 times slower than minimum

Feedback directed compilation

matrix multiply, N=400, Pentium Pro, exhaustive search

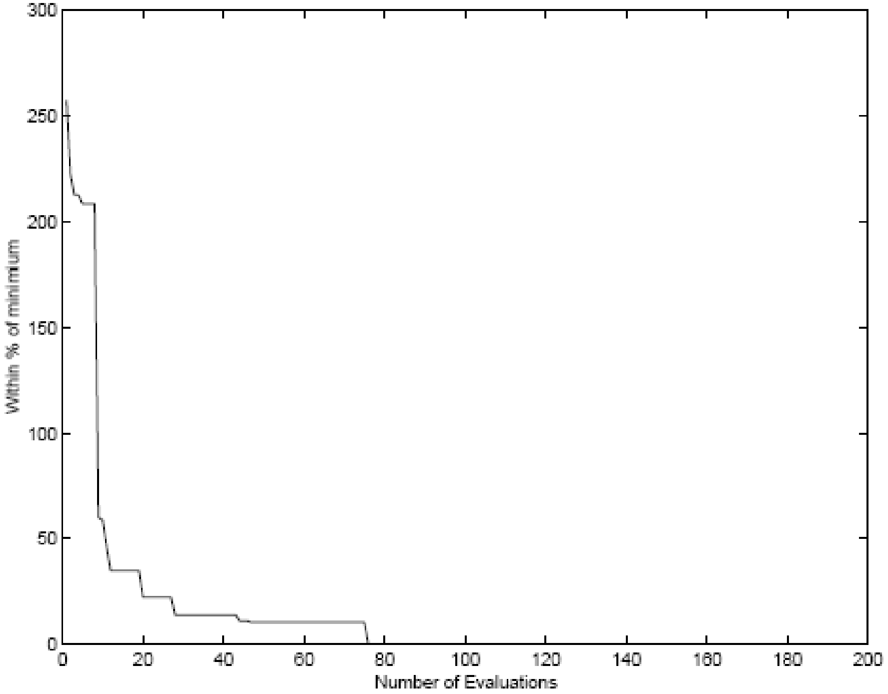


Minimum at: Unroll=19, Tile size=57

Near minimum: 4.3%, original 4.88 sec, minimum 1.43 sec

Feedback directed compilation

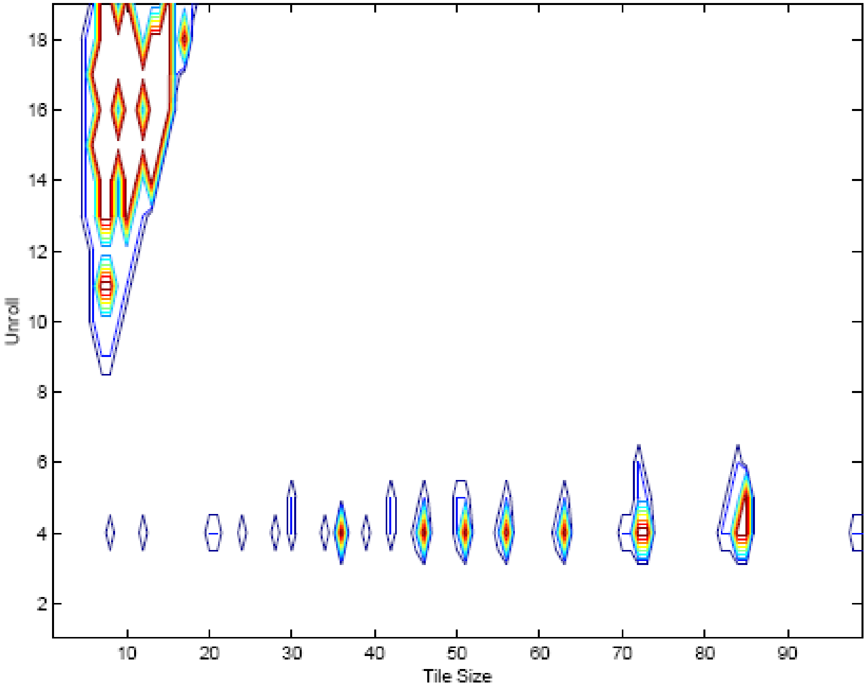
matrix multiply, N=400, Pentium Pro, random search



50 steps: within 10.5%

Feedback directed compilation

matrix multiply, N=512, R10000, exhaustive search

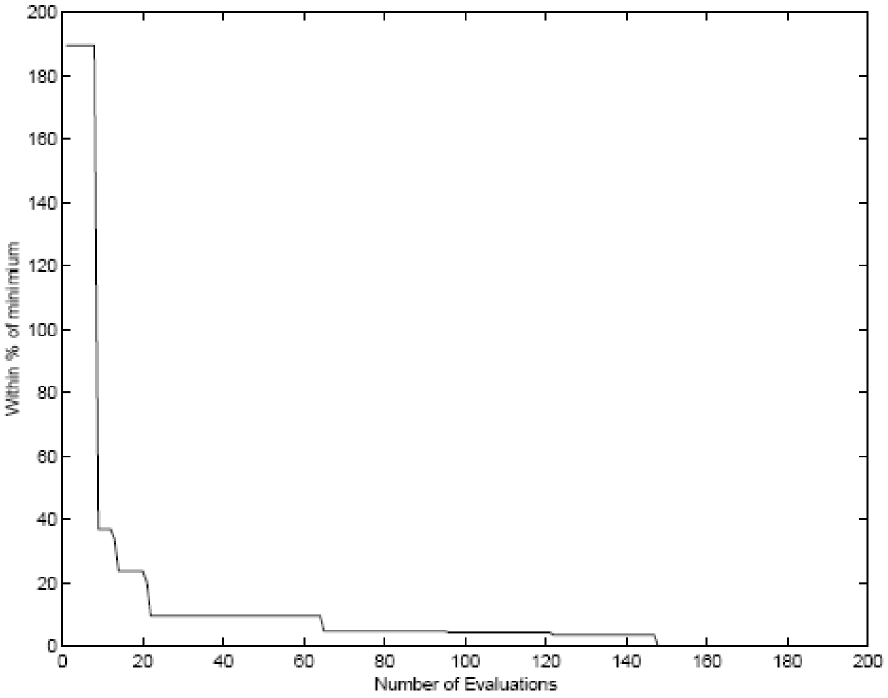


Minimum at: Unroll=4, Tile size=85

Near minimum: 7.2%, original 2.79 sec, minimum 1.09 sec

Feedback directed compilation

matrix multiply, N=512, R10000, random search



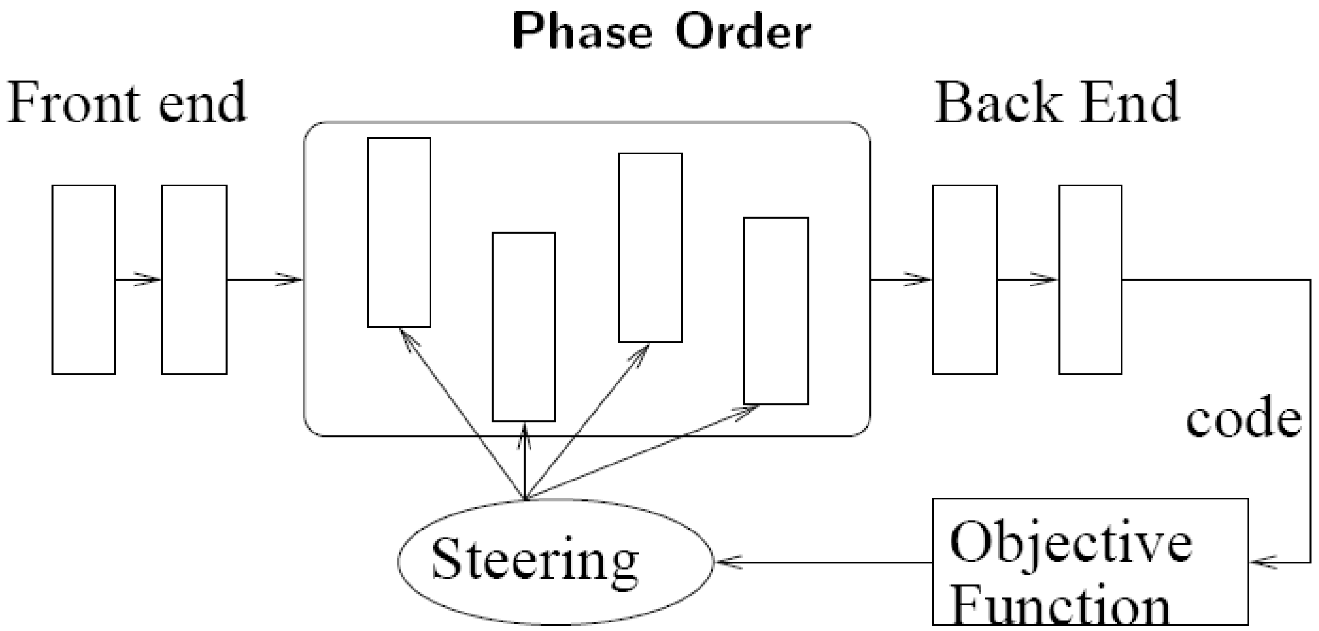
50 steps: within 4.9%

Feedback directed compilation

Phase order

- Oceans work looked at parameterized high level search spaces (tiling, unrolling). Restricted by compilers and only small kernel exploration
- Impressive search results due to “tuned” heuristic and small spaces. In practice depends on space shape
- Keith Cooper et al ’99 onwards also looked at iterative compilation
- Cooper’s search space was the orderings of phases within a compiler
- Lower level and not tied to any language. More generic and explores the age-old phase ordering problem more directly

Feedback directed compilation



- Cooper has found improvements up to 25% over default sequences
- Examined search heuristics that find good points quickly
- However, evaluation approach is strange and results don’t seem portable

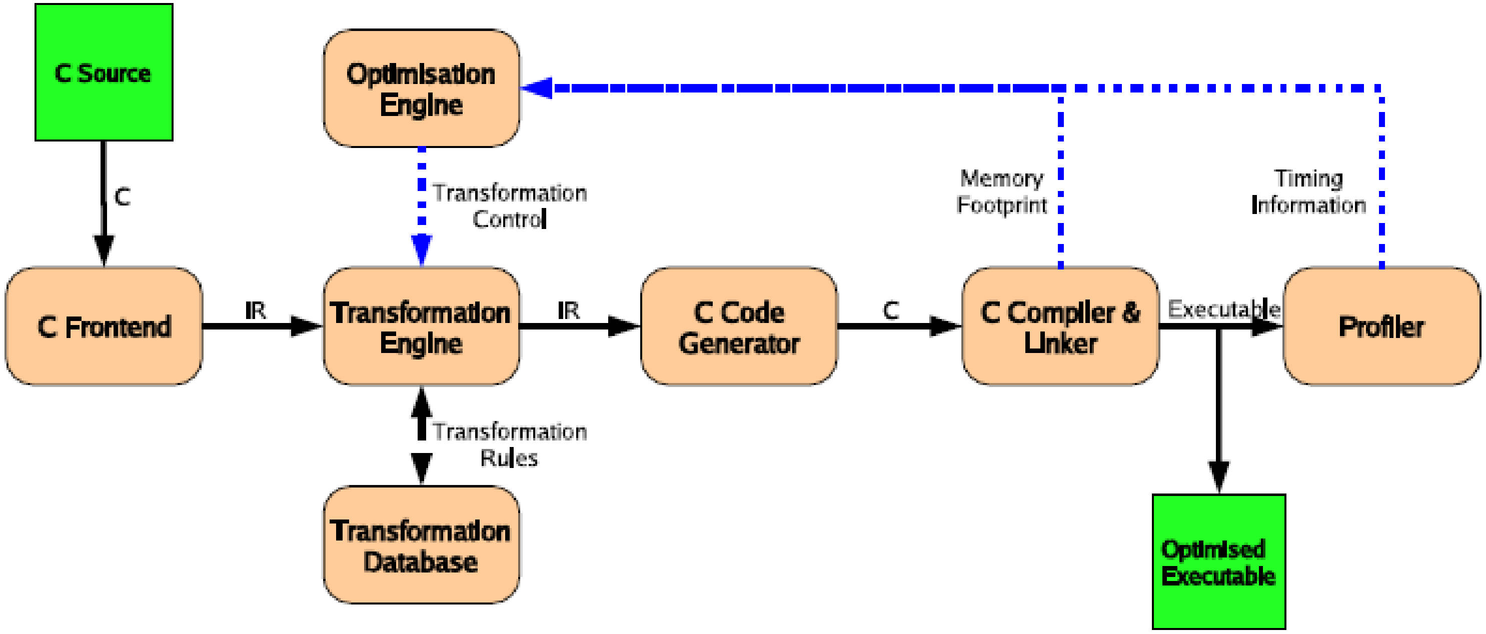
Feedback directed compilation

DSP systems

- Iterative compilation proved to be useful for embedded applications or libraries.
- It is difficult to improve on embedded compilers and hard to get access to internals. HLT is attractive but pointers cause problems
- Franke et al 2005 overcomes this with a pointer recovery + SUIF based transformation explorer. Uses 2 search strategies

Feedback directed compilation

DSP framework



Using this framework to exhaustively explore and characterize the optimization space

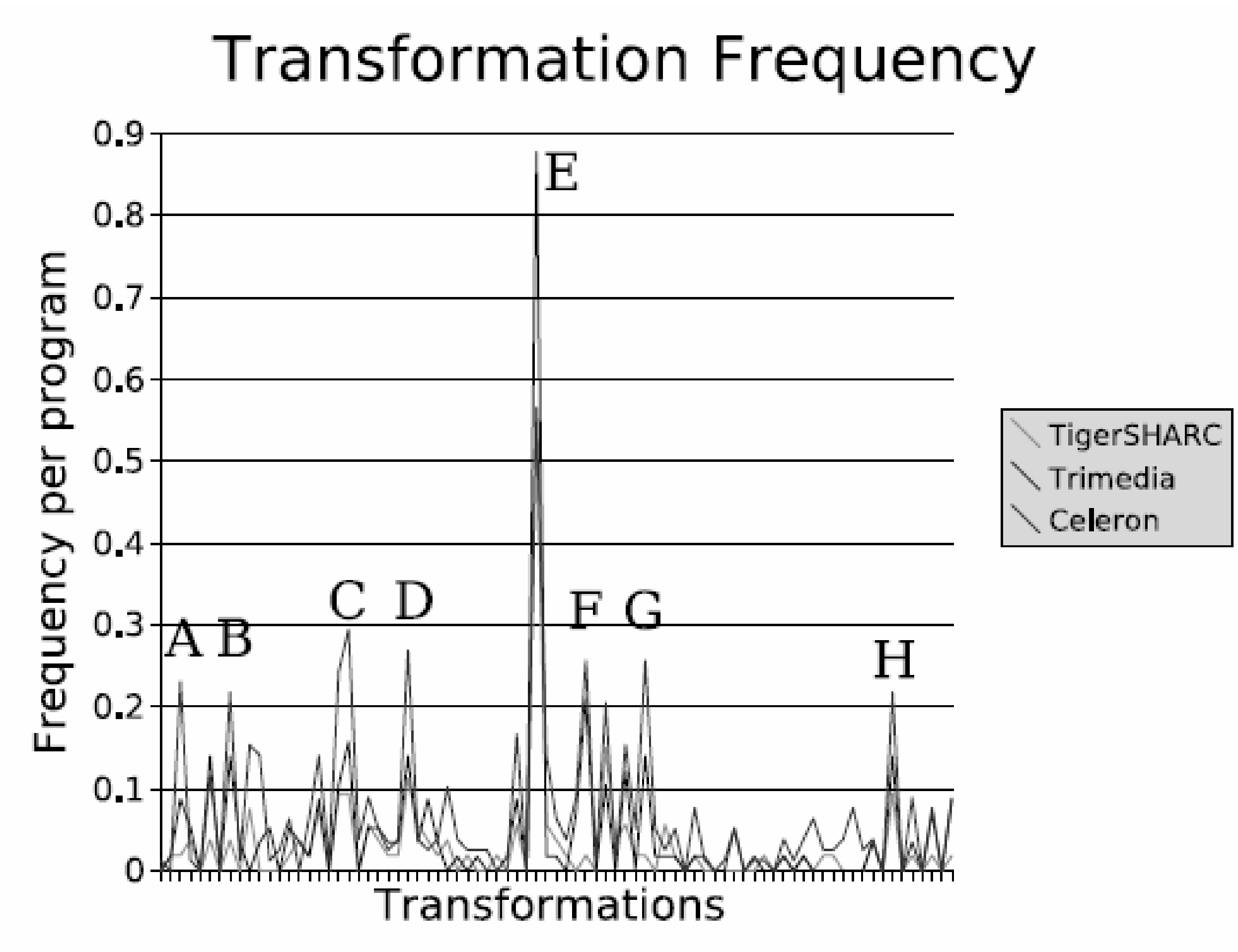
Feedback directed compilation

Franke et al

- Looks through space of 80^{80} transformations on 3 platforms for UTDSP benchmark suite. Not feasible to do exhaustively. Really stresses SUIF
- 2 algorithms. Trade-off between coverage and focus. Random search - select a random length up to 80. Then randomly select any transformation for each location. Lots of redundant transformations.
- PBIL: Population based inference learning. Modify probability of selecting transformation based on previous trials. Only examine effective transformations
- Average 41% reduction. PBIL finds the best in majority of cases but Random best has higher speed up.

Feedback directed compilation

Impact of transformations



Feedback directed compilation

Results

- Tried 500 runs. On UTDSP benchmark: TriMedia average speedup of 1.43 and 1.73 for TigerSharc
- Shows that HLT can give a big win compared to backend optimizations
- Also compared GCC and ICC on embedded Celeron
- Original: ICC 1.22 faster than GCC
- GCC + IC: speedup of 1.54 - better than ICC
- BUT ICC + IC: speedup of 2.14

Feedback directed compilation

Interactive Compilation Interface (Fursin et al'2005)

- 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 or local level and avoid revealing all intermediate compiler representation to allow further transparent compiler evolution

- Narrow down optimization space by suggesting only legal transformations

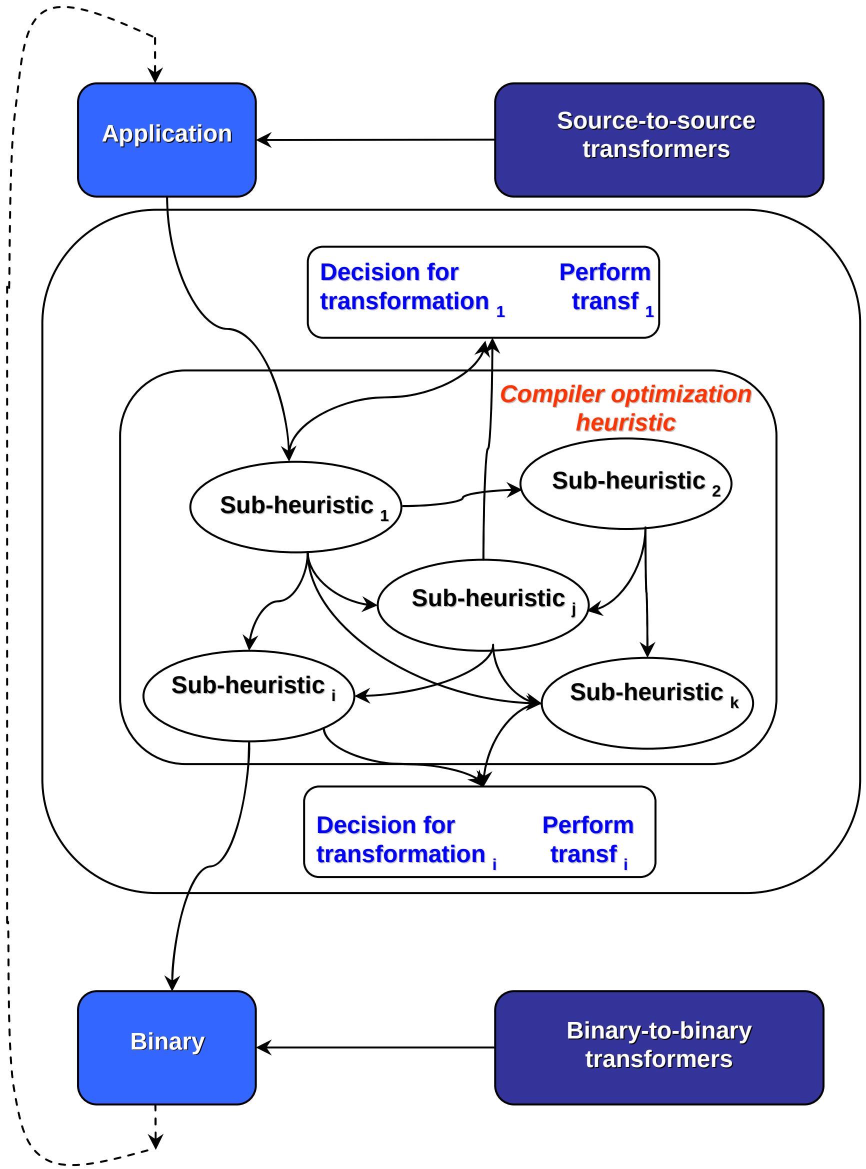
- Enable iterative recompilation algorithm to apply sequences of transformations

- 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

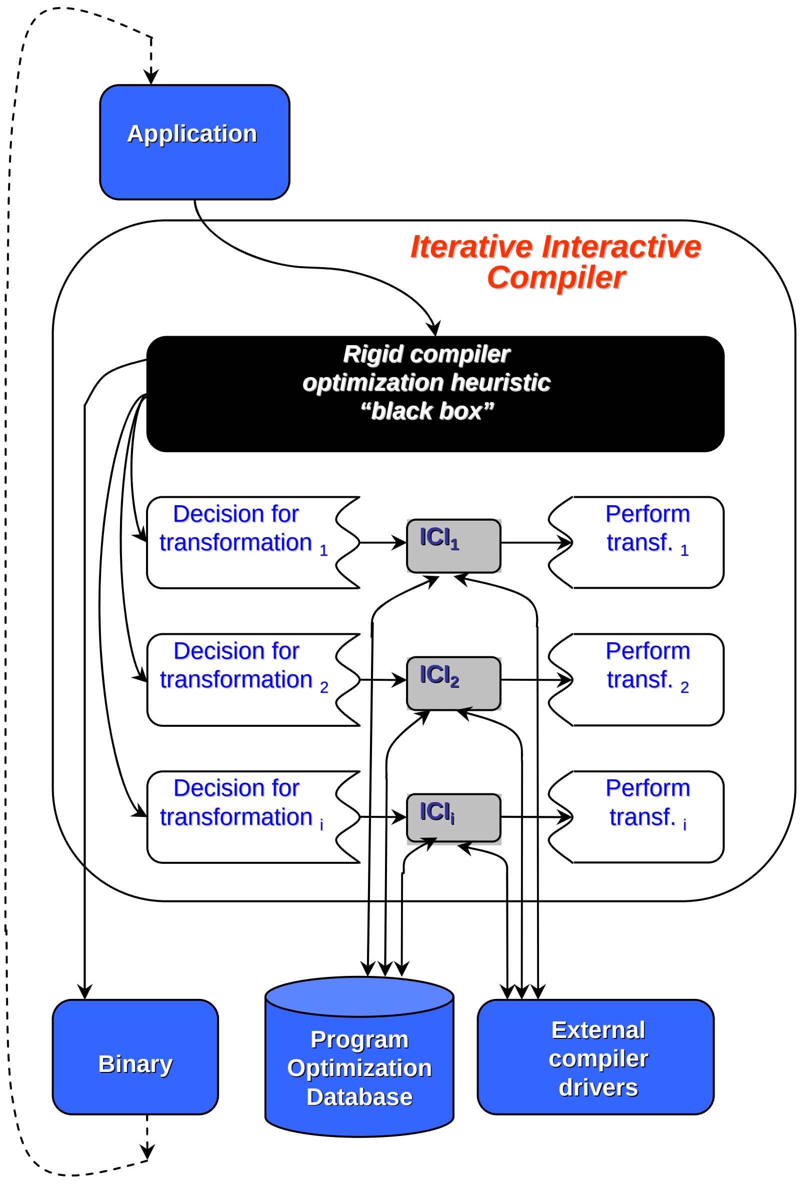
Feedback directed compilation

Interactive Compilation Interface


Application
Source-to-source transformers
Decision for transformation $_1$
Perform transf $_1$
Compiler optimization heuristic
Sub-heuristic $_1$
Sub-heuristic $_2$
Sub-heuristic $_j$
Sub-heuristic $_i$
Sub-heuristic $_k$
Decision for transformation $_i$
Perform transf $_i$
Binary
Binary-to-binary transformers

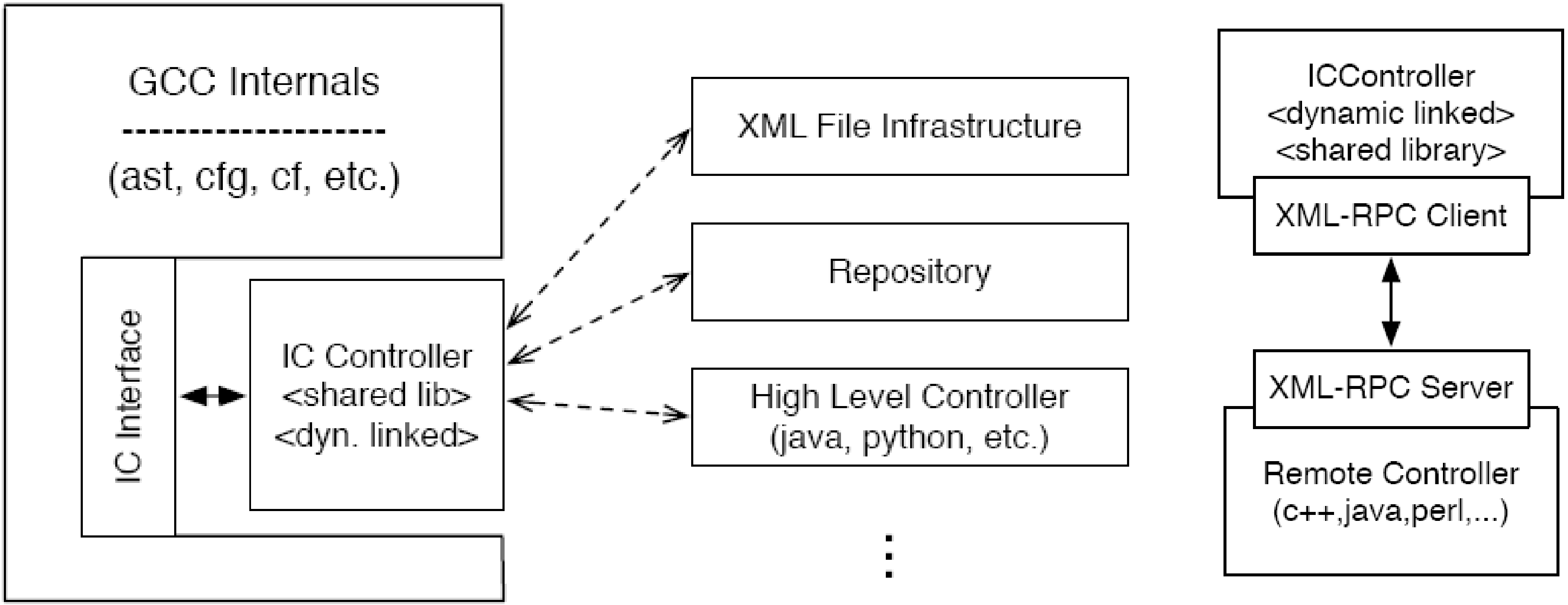
Feedback directed compilation

Interactive Compilation Interface


Application
Iterative Interactive Compiler
Rigid compiler optimization heuristic "black box"
Decision for transformation 1
ICI1
Perform transf. 1
Decision for transformation 2
ICI2
Perform transf. 2
Decision for transformation i
ICIi
Perform transf. i
Binary
Program Optimization Database
External compiler drivers

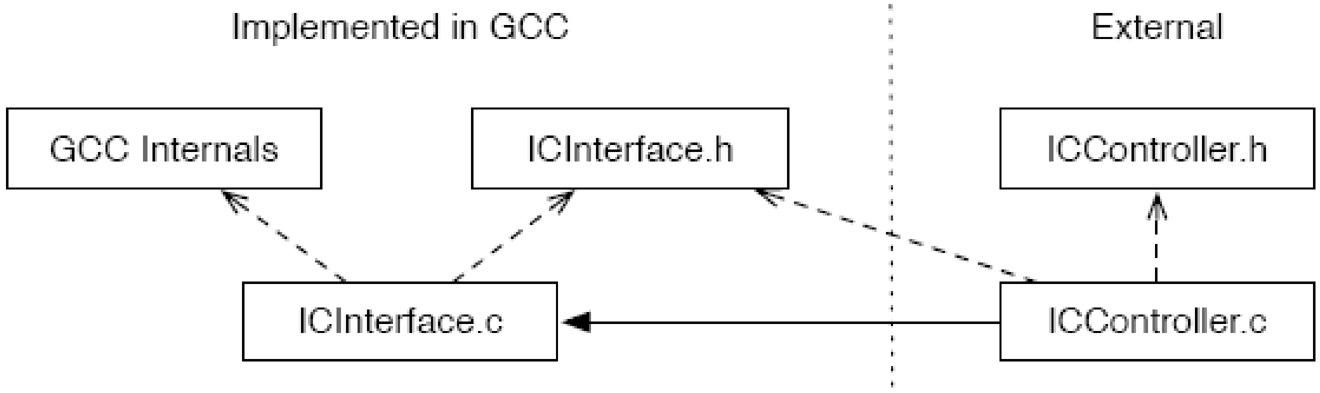
Feedback directed compilation

Interactive Compilation Interface



moves toward simpler modular compiler

Feedback directed compilation



```
int get_interface_version (void);
void clean_scope (void);
bool scope_to_function (char *func_name);
bool scope_to_loop (int loop);
void *get_feature (char *feature_name);
char **get_available_features_for_type (int type);
bool run_pass (char *pass_name);
bool unroll_loop( int factor, enum UNROLL_TYPE type);
bool loop_interchange (int loop_number);
bool loop_fusion (int nr_of_consecutive_loops);
bool function_inline (int call_id);
```

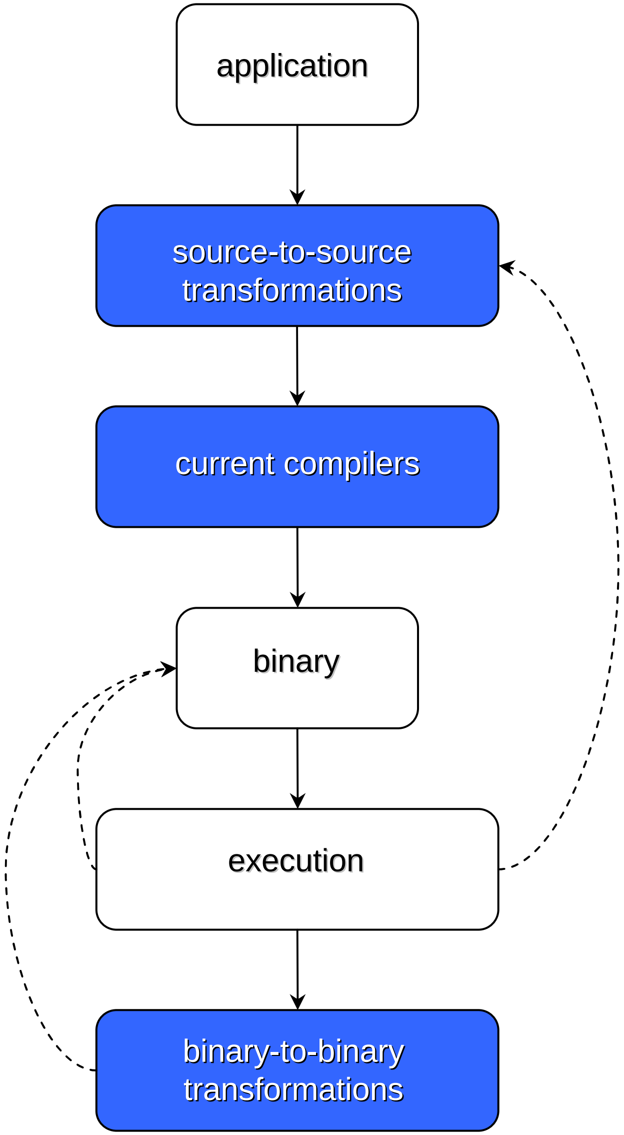
Feedback directed compilation

Interactive Compilation Interface

```
#include "ic-controller.h"
#include "ic-interface.h"
bool start (char *params)
{
  int *version = get_interface_version ();
  bool ret = (*version > 100) ? true : false;
  free(version);
  return ret;
}
void stop (void)
{
  /* nothing to be done; */
}
void controller (void)
{
  char **passes = get_feature ("global_passes");
  char **functions = get_feature ("functions");
  char **tmp, **tmp1;
  // IPA passes
  for (tmp = passes; *tmp != NULL; tmp++)
  {
    char *pass_name = *tmp;
    // run_pass should never return false, since we are performing same pass
    // order as GCC.
    run_pass(pass_name);
    free(pass_name);
   }
```

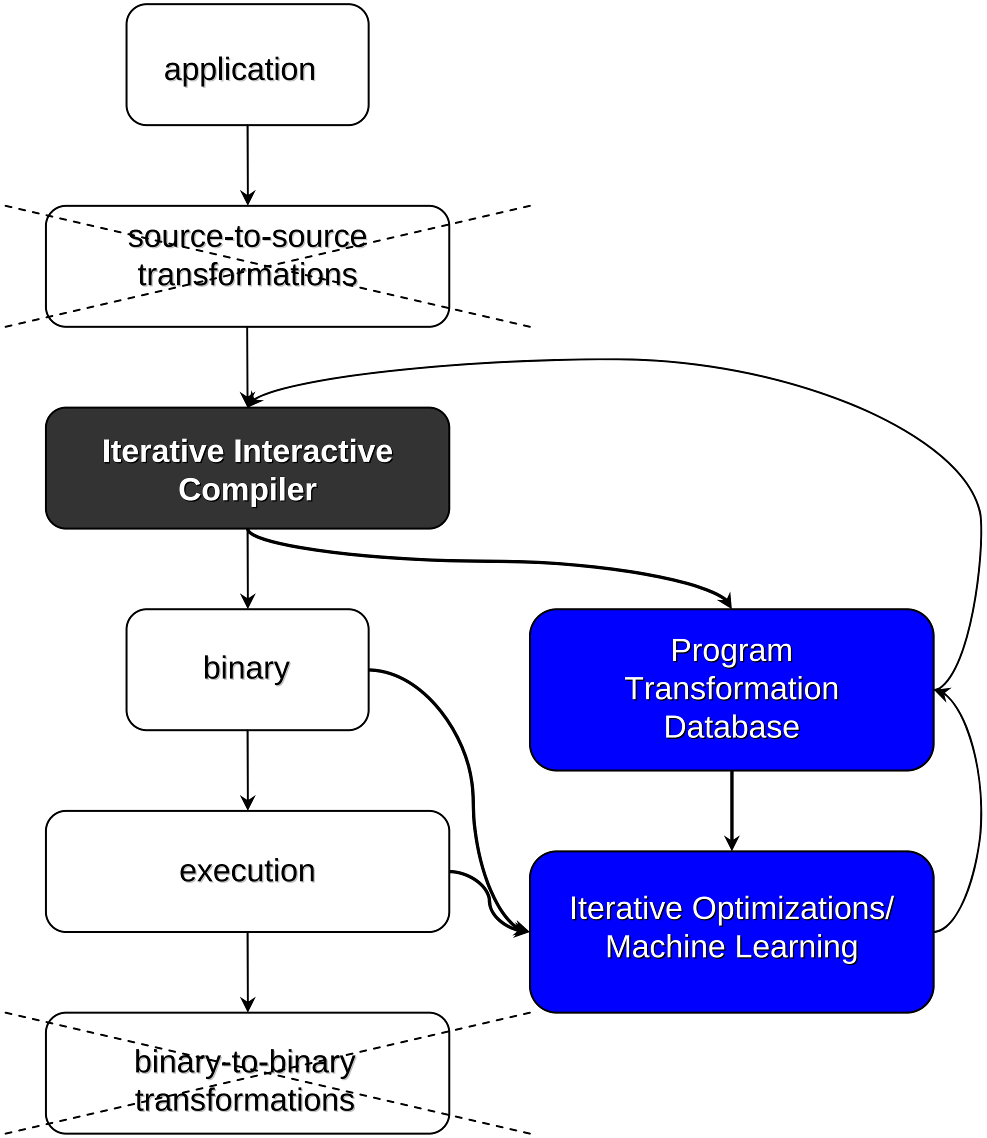
Feedback directed compilation

Interactive Continuous Compilation


application
source-to-source transformations
current compilers
binary
execution
binary-to-binary transformations

Feedback directed compilation

Interactive Continuous Compilation


application
~~source-to-source transformations~~
Iterative Interactive Compiler
binary
Program Transformation Database
execution
Iterative Optimizations/ Machine Learning
~~binary-to-binary transformations~~

Development Websites:

http://gcc-ici.sourceforge.net

http://pathscale-ici.sourceforge.net

http://open64-ici.sourceforge.net

http://gcc-ccc.sourceforge.net

Feedback directed compilation

Evaluating iterative compilation with multiple datasets

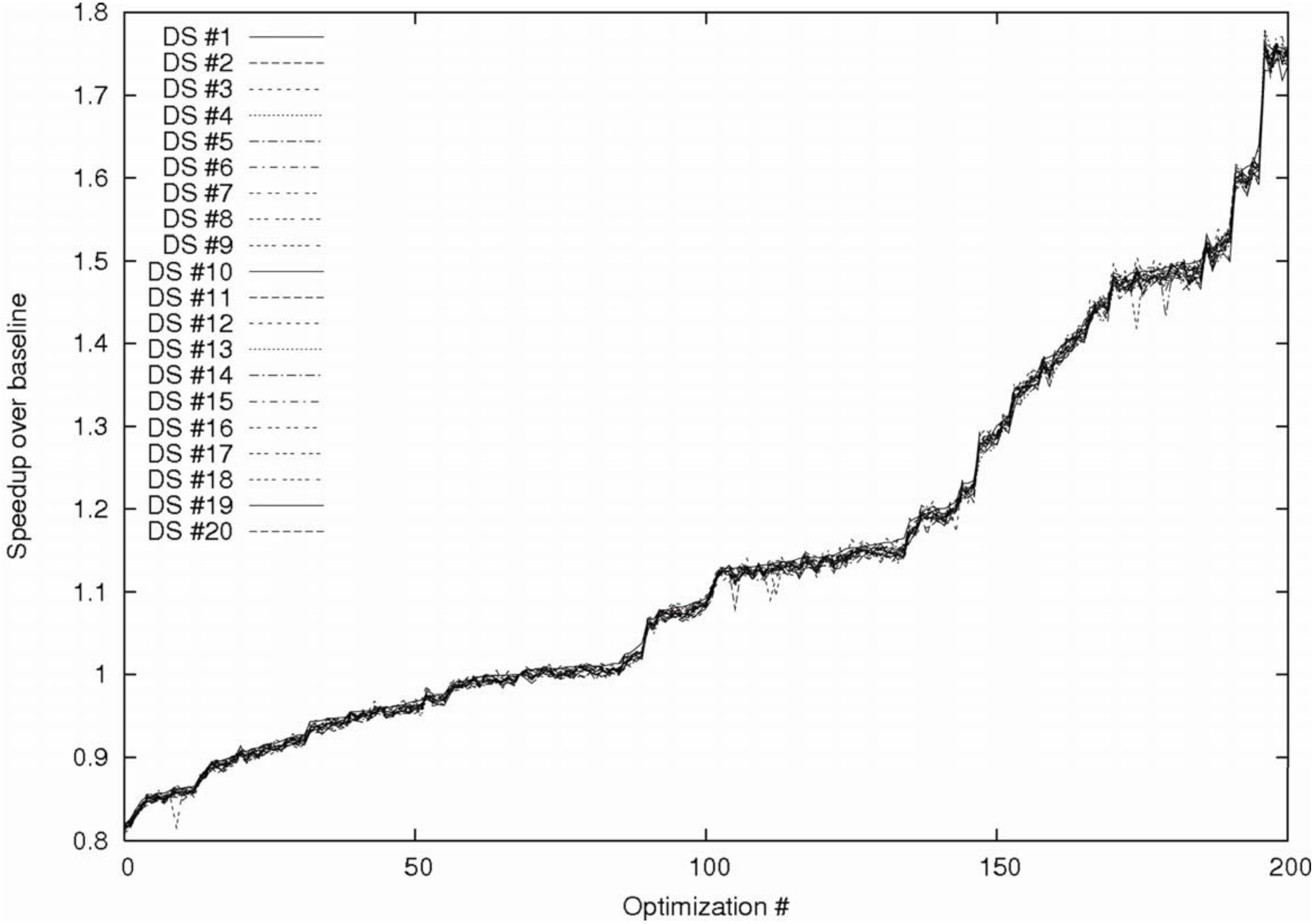
MiDataSets for MiBench – 20 per program

Iterative search for best compiler flags using PathScale compiler suite

Grigori Fursin, John Cavazos, Michael O'Boyle and Olivier Temam. MiDataSets: Creating The Conditions For A More Realistic Evaluation of Iterative Optimization. Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007), Ghent, Belgium, January 2007

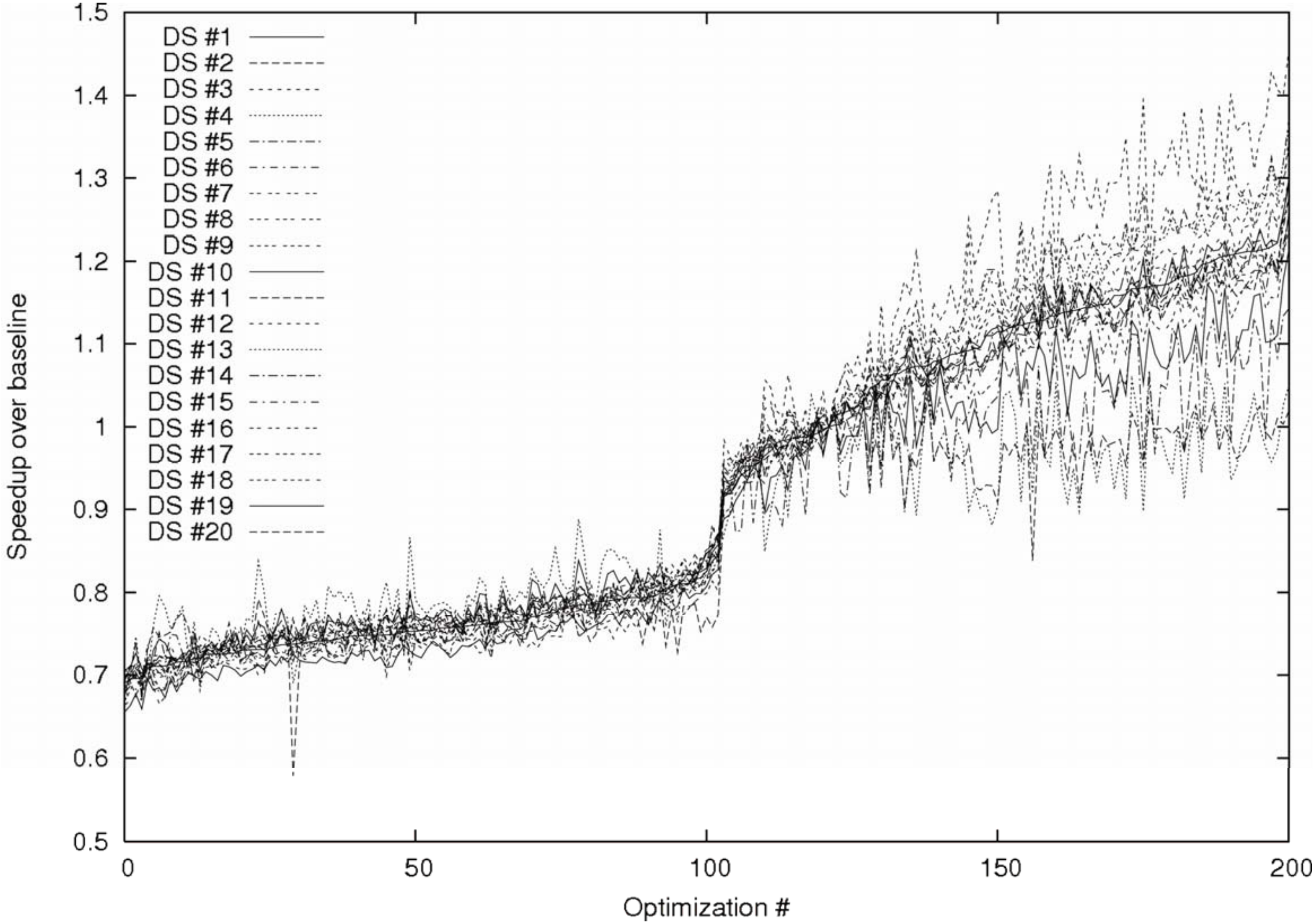
Development website: *http://midatasets.sourceforge.net*

Feedback directed compilation



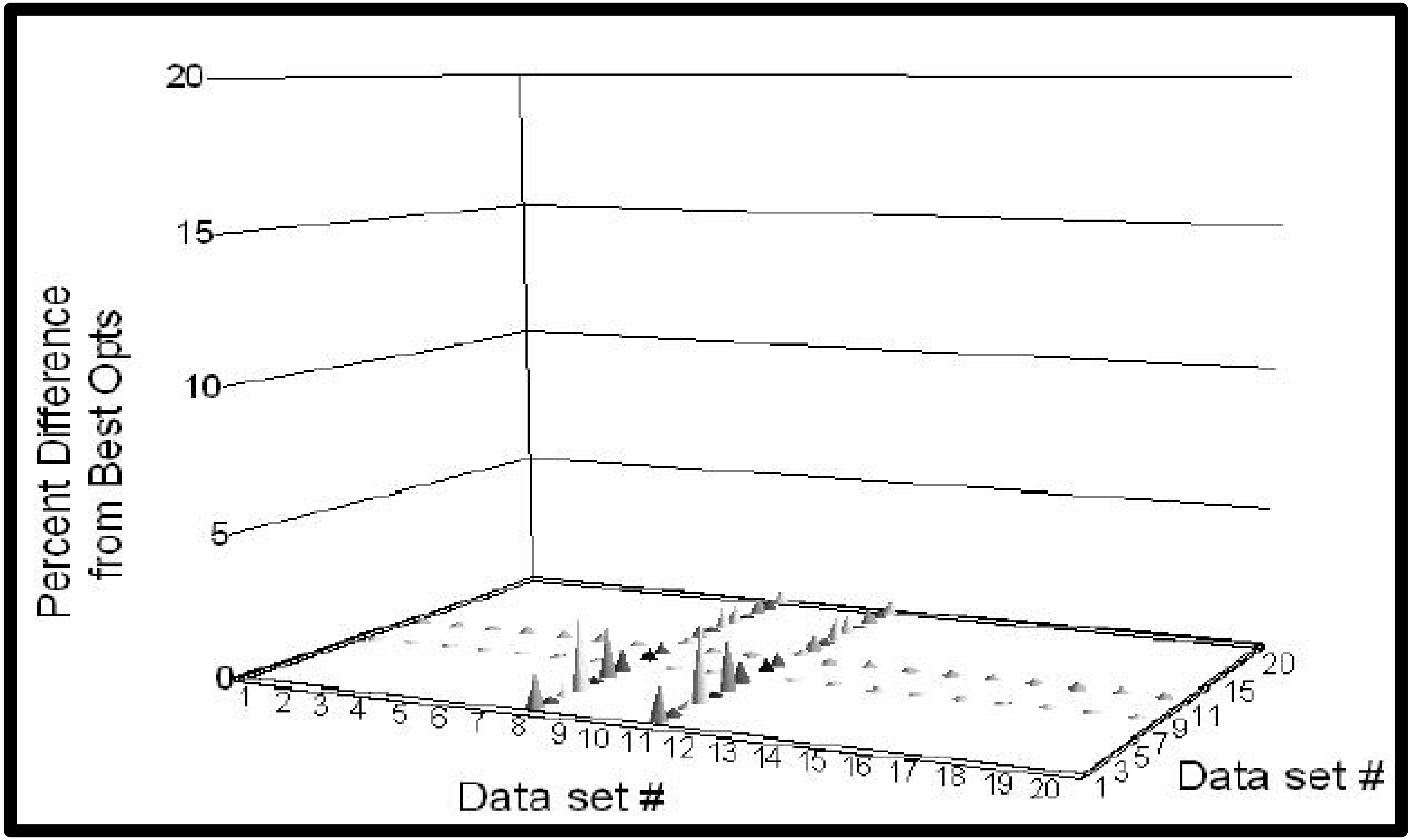
Data sets reactions to optimizations (dijkstra).

Feedback directed compilation



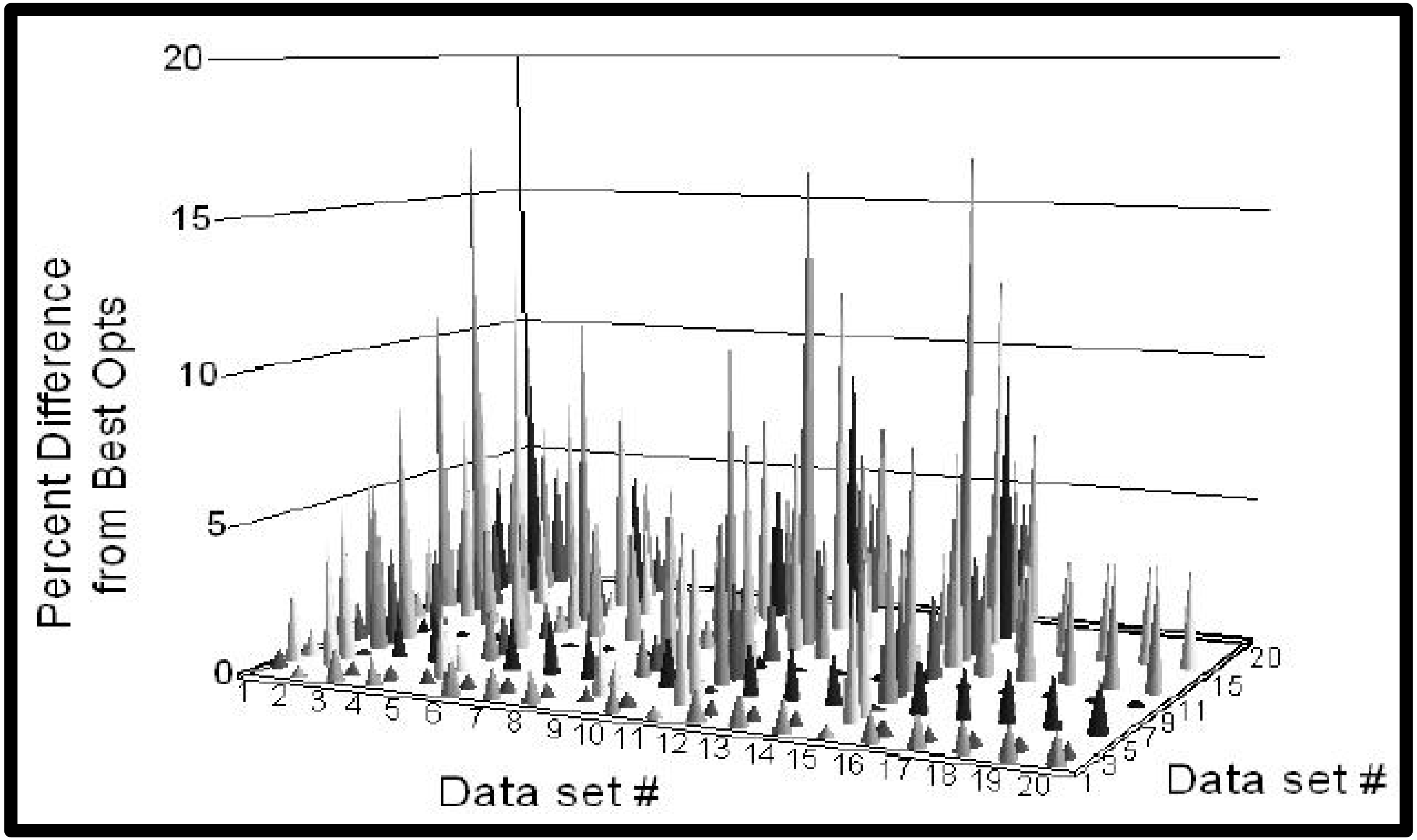
Data sets reactions to optimizations (jpeg decode).

Feedback directed compilation



Variation of best optimizations across programs (SHA)

Feedback directed compilation



Variation of best optimizations across programs (SUSAN Corners)

Feedback directed compilation

Search speed

- The main problem is optimization space size and speed to solution
- Many use a cut down transformation space - but this just imposes ad hoc non portable bias
- Need to have large interesting transformation space. Orthogonal - no repetition. SUIF is ad hoc. UTF framework from Shun et al 2004 very systematic but doesn’t cover everything
- Build search techniques to find good points quickly

Feedback directed compilation

Using models

- Obvious approach is to use cheap static modes to help reduce number of runs
- Difficulty is to balance savings gained by model against hardwiring strategy
- Wolfe and Mayadan generate many versions of a program and check against an internal cache models rather than generate the best by construction
- Although more successful doesn't address problem of processor complexity. No real feedback (Pugh A* search). Cannot adapt
- Knijnenburg et al PACT 2000 use simple cache models as filters. Used to eliminate bad options rather than as substitute for feedback. Obtained significant speed up

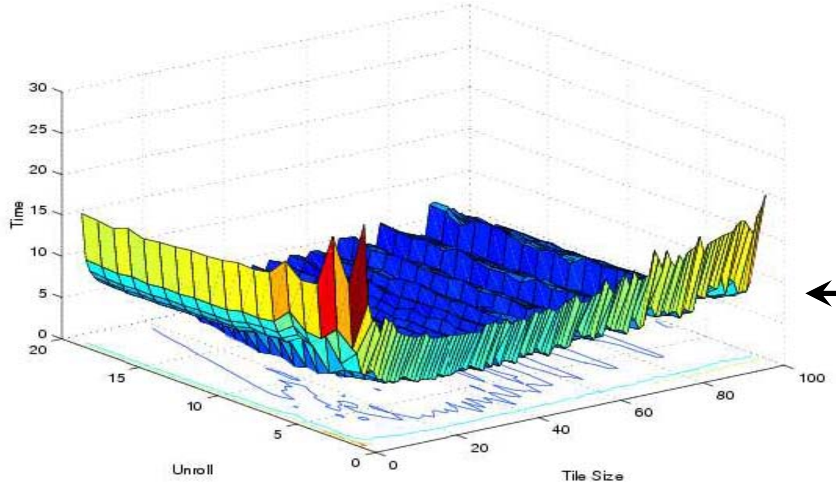
Feedback directed compilation

Search space

- Understanding the shape or structure of search space is vital to determining good ways to search it
- Unfortunately little agreement
- Vuduc '99 shows that minima dramatically vary across processor
- Cooper shows that reasonable minima are very near any given point
- However, our recent work shows that it strongly depends on scenario. Rich space on a TriMedia while golf green on the TI. Should use structure to aid search
- Vuduc uses distribution of good points as stopping criteria
- Fursin use upper bound of performance as guide.

Feedback directed compilation

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

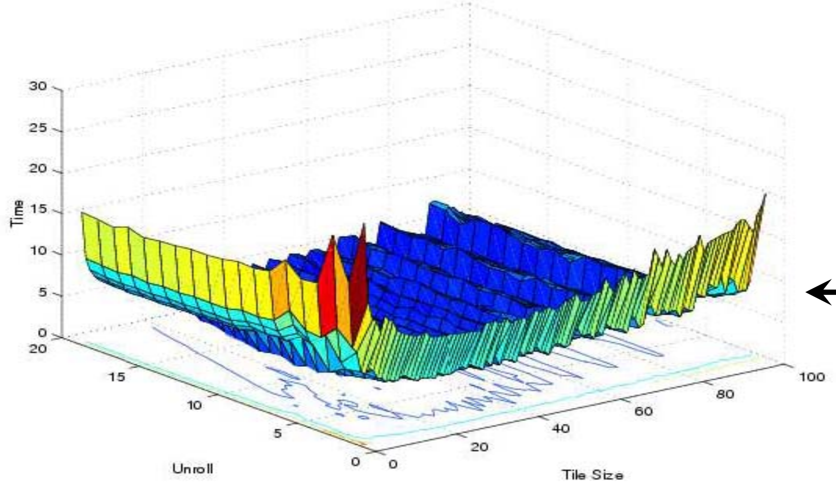
Feedback directed compilation

Next will focus on

dynamic compilation/optimization approaches to adapt to different programs behavior at run-time and machine learning to speed up iterative search…

Reminder

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



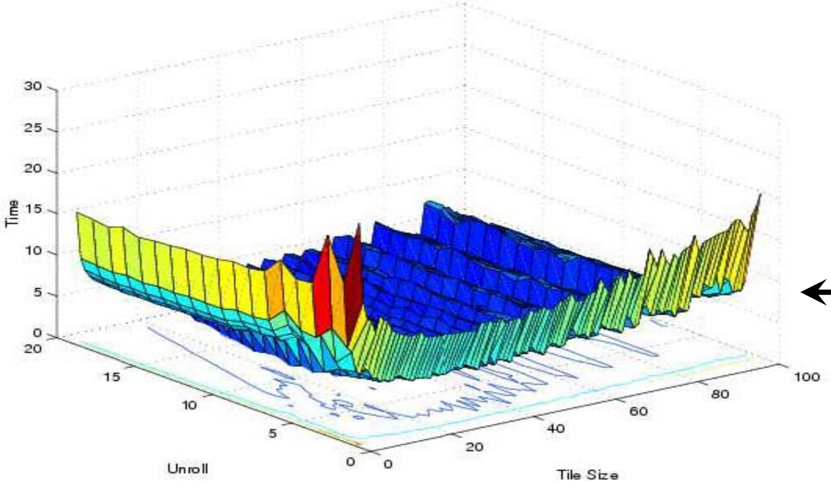
Finding a good solution may be long and non-trivial

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}

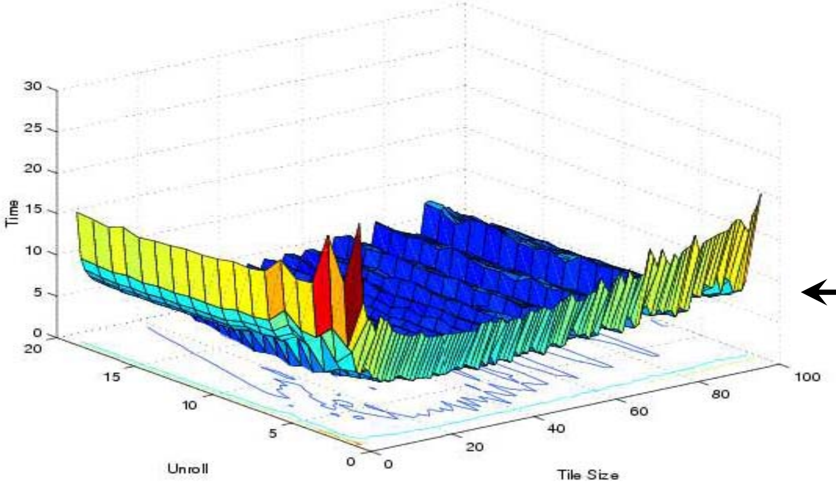
Recent technique - iterative compilation: learn program behavior across executions

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 more fundamentally has runtime type checking of objects

- Smalltalk in the 80s deferred compilation to runtime to reduce the amount of compilation otherwise required in the 00 setting

- Java applications are compiled into bytecode and 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

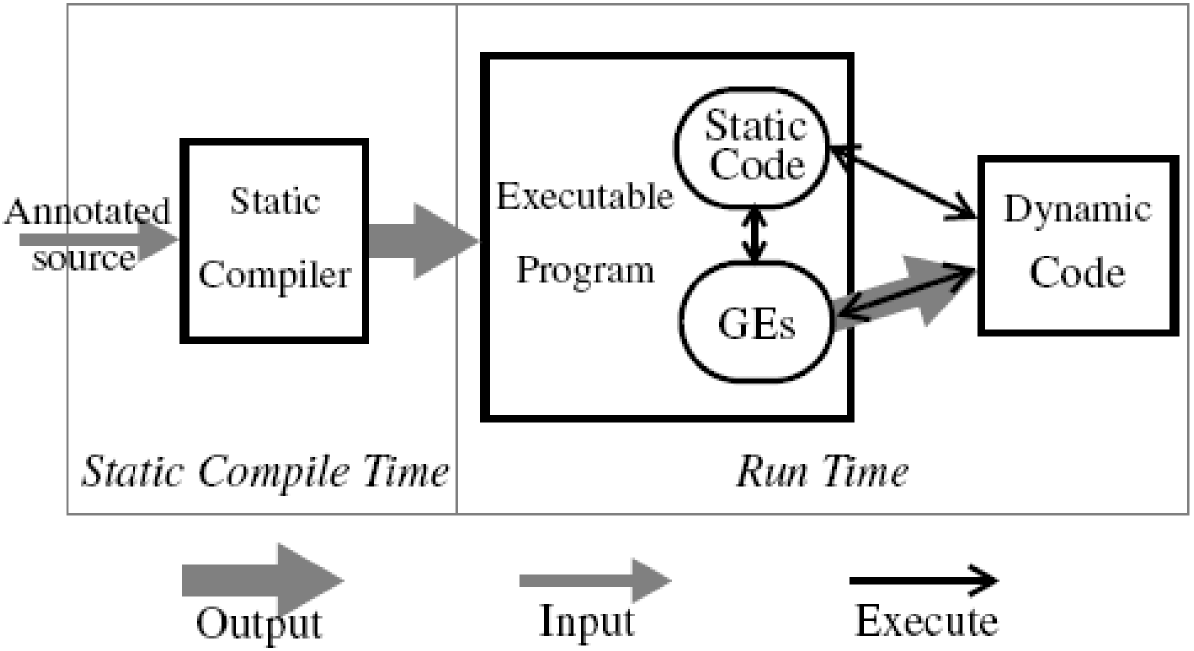
- The user annotates the program defining where there may be opportunities for runtime specialization. Marks variables and memory locations that are static within a particular scope

- 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 speedup and a range 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

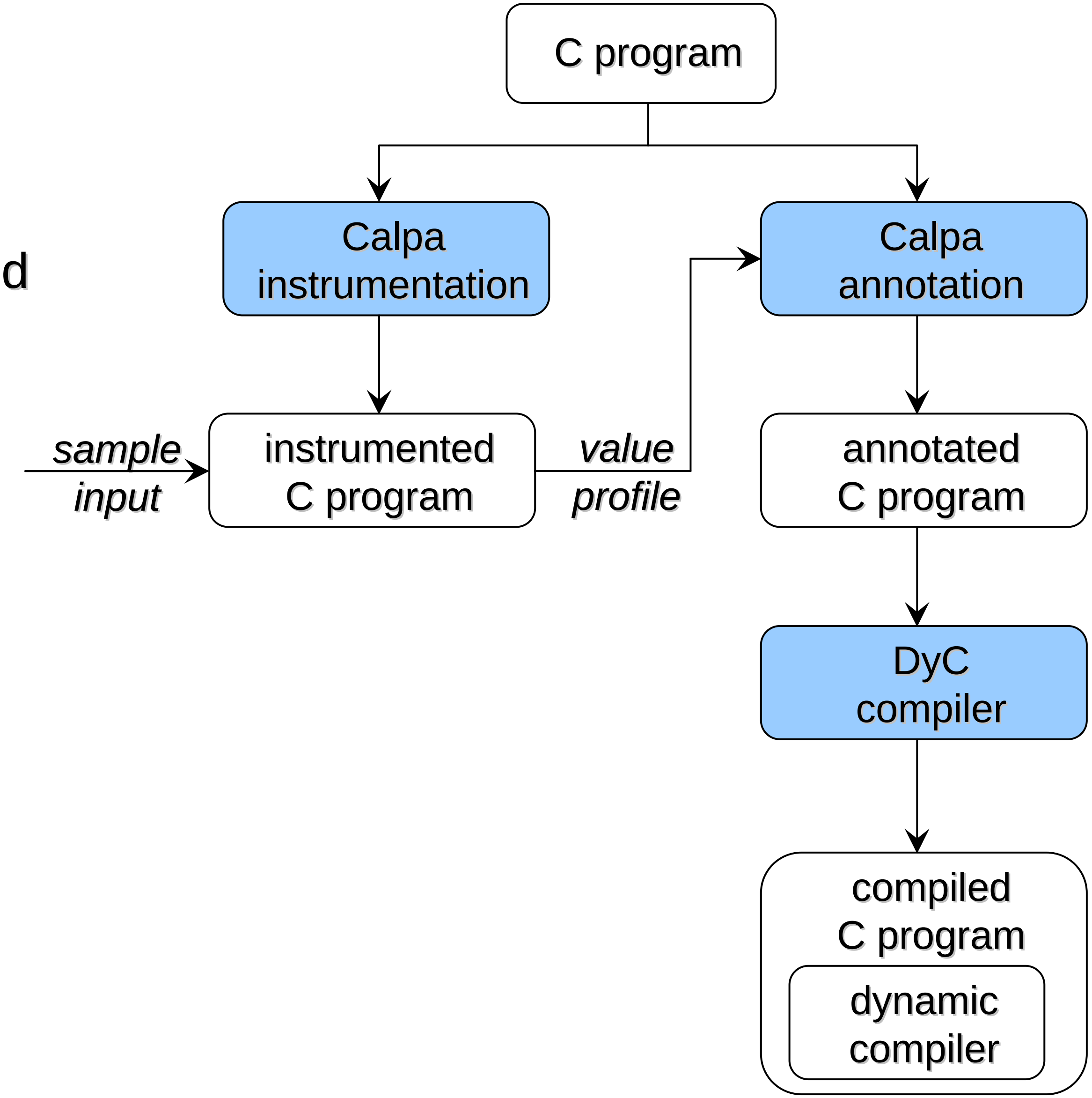
- For example if a variable is multiplied by another variable which is known to be constant in a particular scope, then if this is equal to 0 or 1 then cheaper code maybe generated

- If this 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 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 optimization

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

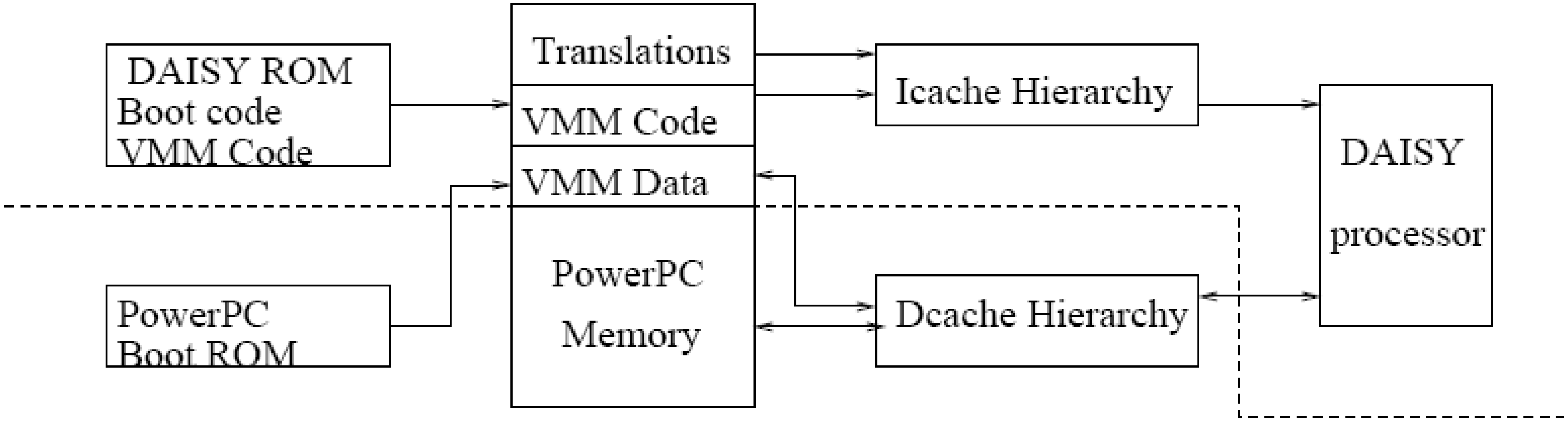
- Dynamic translation opens up opportunities for dynamic optimization.

- 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

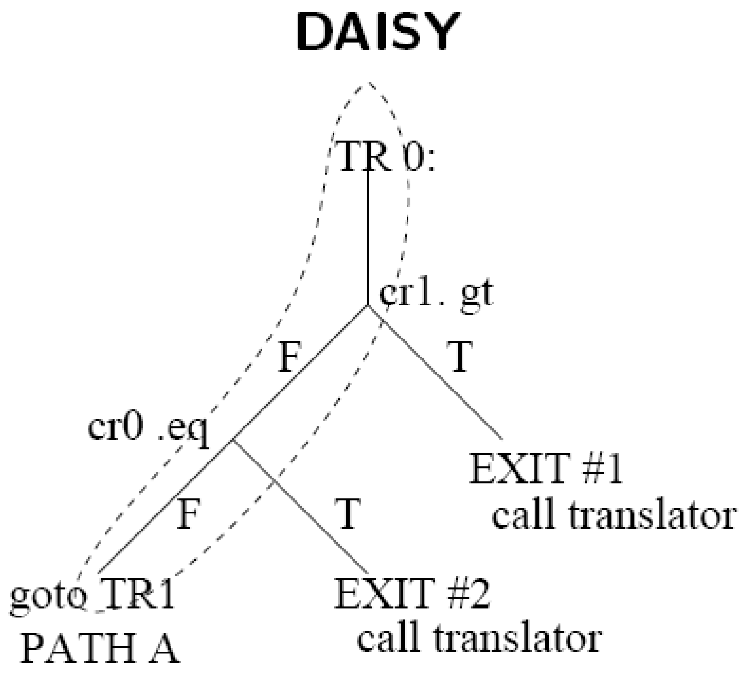


- Power PC code runs without modification
- DAISY specific additions separated by dotted line
- Initially interpret PowerPC instructions and then compile after hitting threshold
- Then schedule and save instruction in cache (2-4k). Untaken branches are translated as (unused) calls to the binary translator

DAISY example

- Here the group is expanded to contain two conditionals

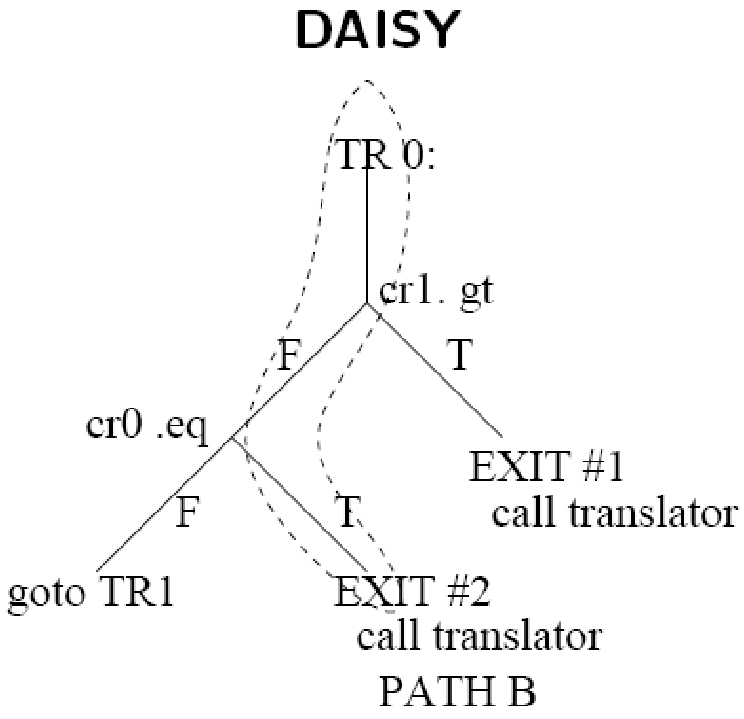
- Path A is encountered and translated



DAISY example

- When Path B is encountered for the first time

- Translator is called



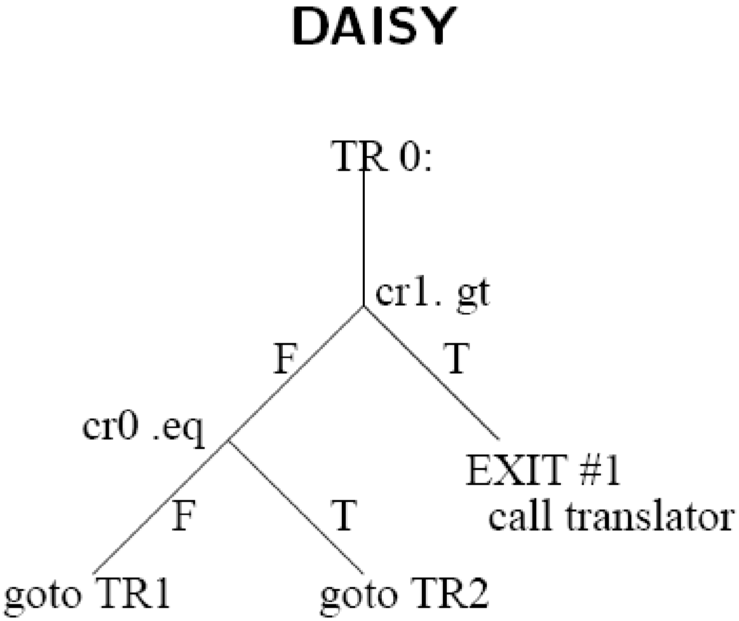
DAISY example

- 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 way of improving performance from a shipped binary overtime

- 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 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

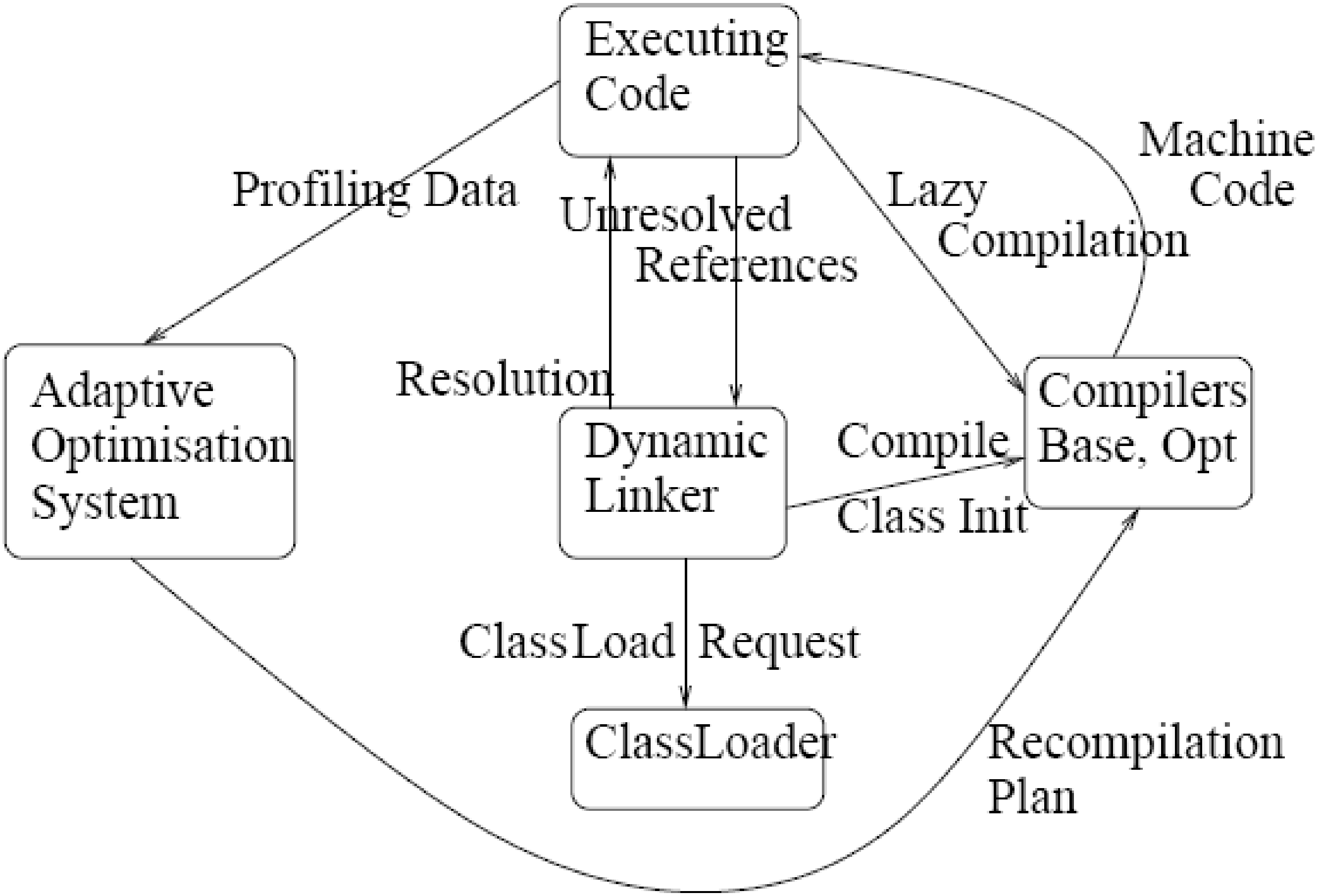
- Normally done on a per method basis

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

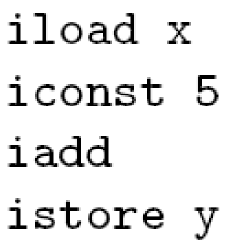
- Well known robust research compiler freely available

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

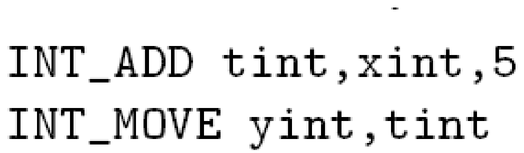
Jikes structure



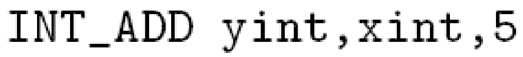
Jikes example



```
iload x
iconst 5
iadd
istore y
```



```
INT_ADD tint,xint,5
INT_MOVE yint,tint
```



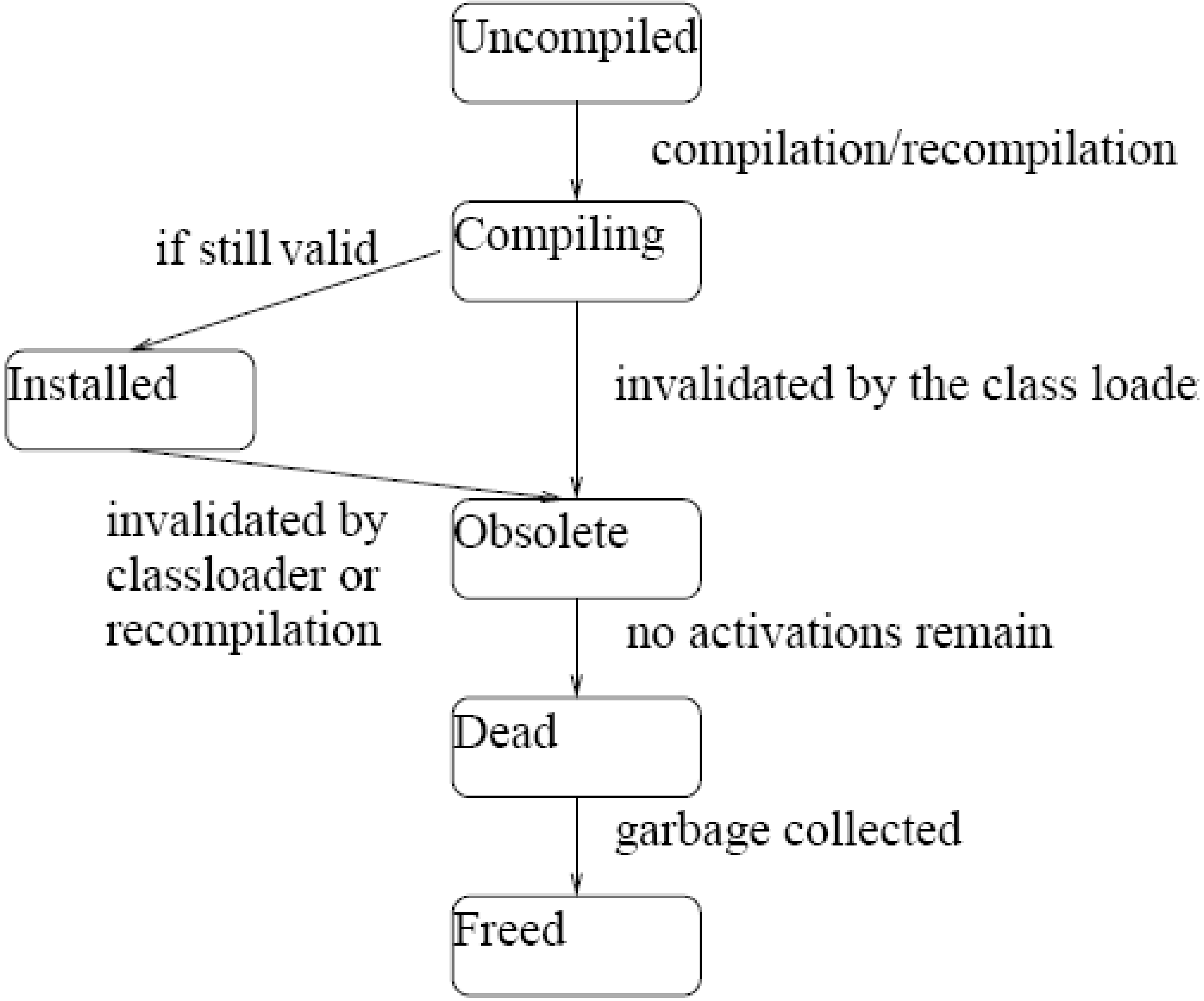
```
INT_ADD yint,xint,5
```

- 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

- Online iterative compilation main contribution

- Only works for scientific programs with relatively static behavior

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?

- 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)

- HiPEAC cluster funding to "Explore optimization techniques and runtime code selection mechanisms for heterogeneous systems" for 18 months starting from September, 2006. Collaboration with STMicro, IBM, UPC

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!

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:

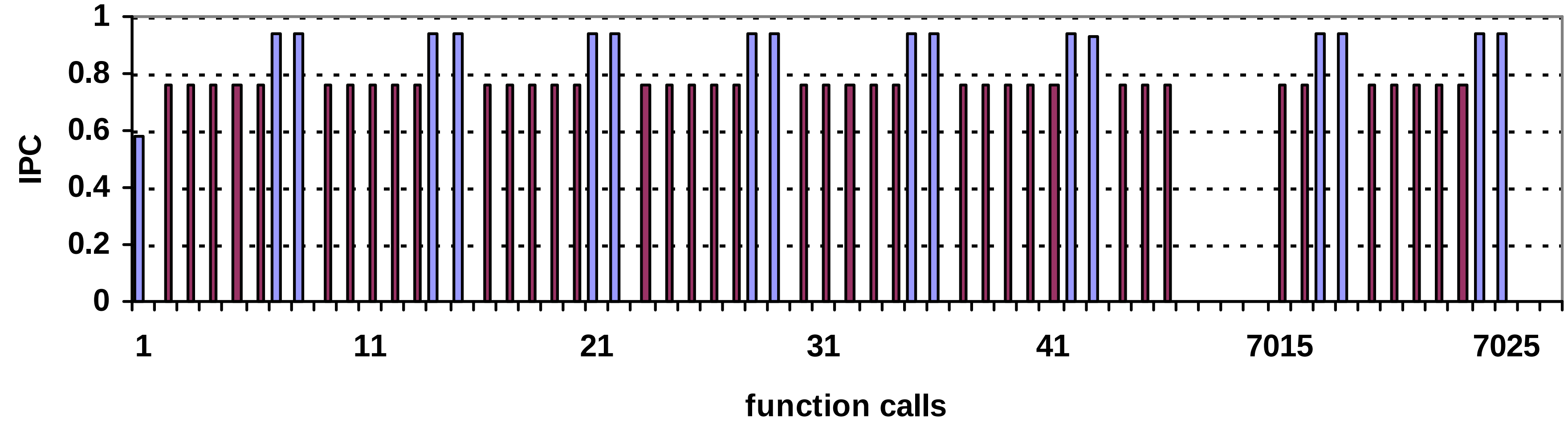
typically functions or loops

Run-time program behavior

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

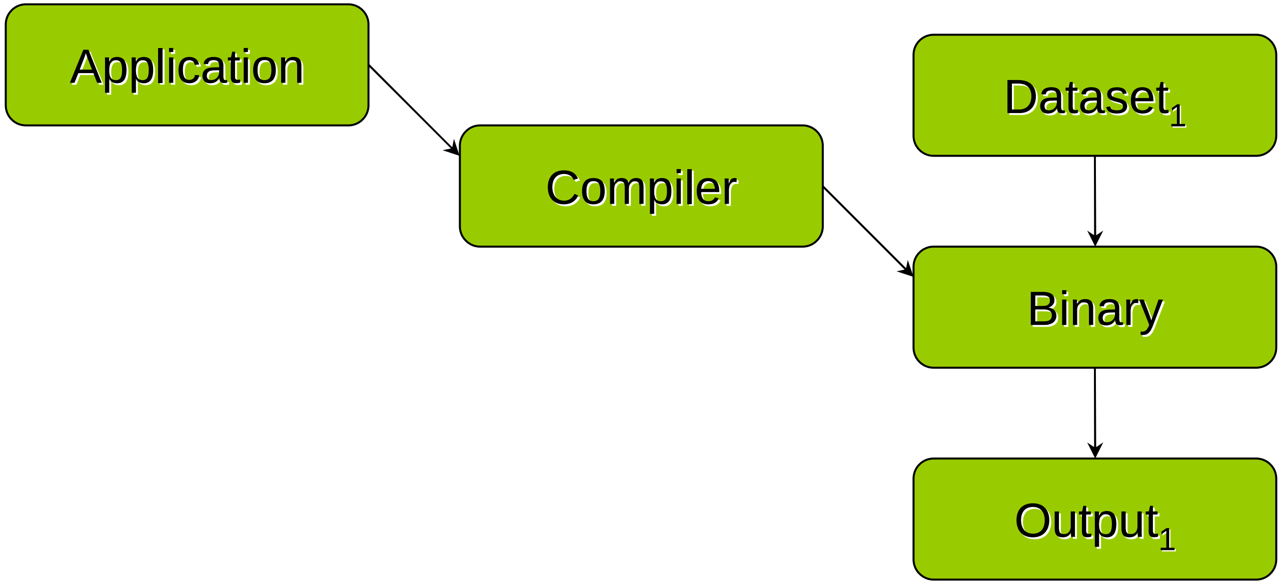
typically functions or loops

IPC for subroutine resid of benchmark mgrid across calls



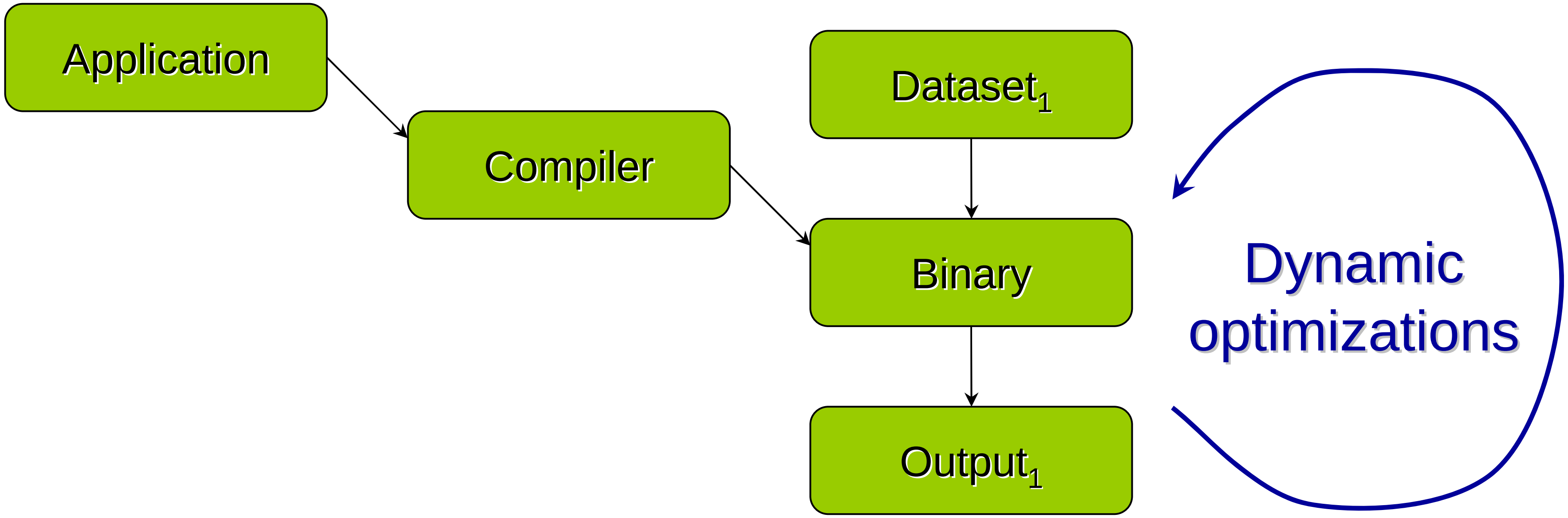
Current methods

Some existing solutions:



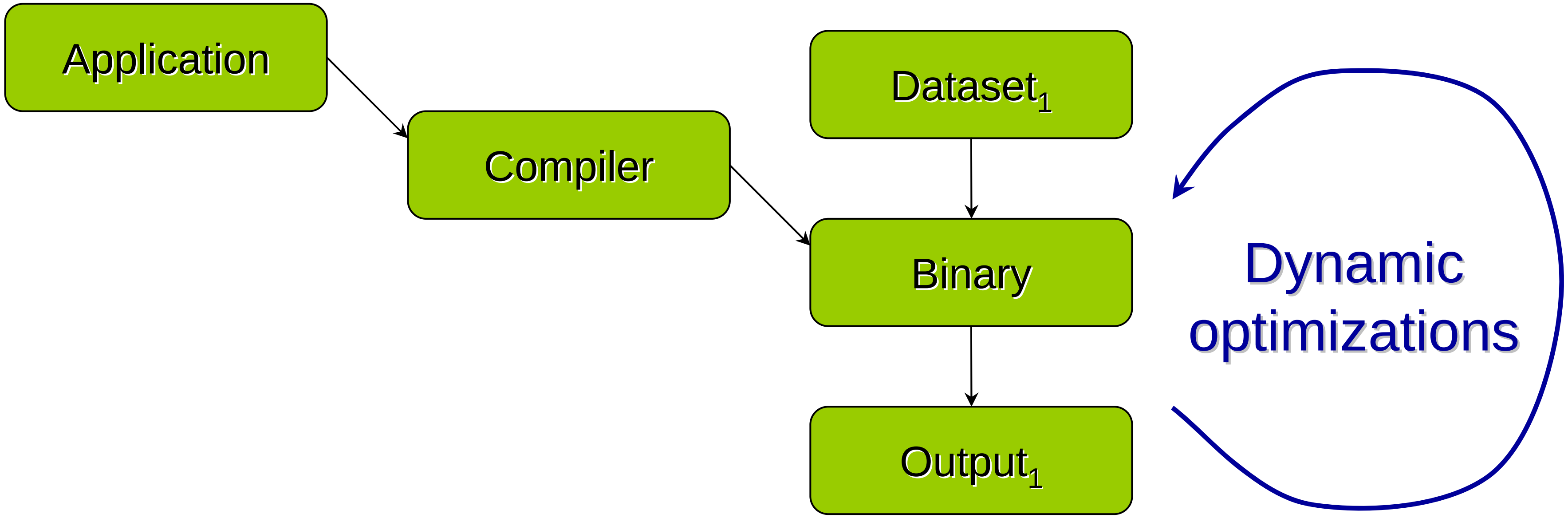
Current methods

Some existing solutions:



Current methods

Some existing solutions:

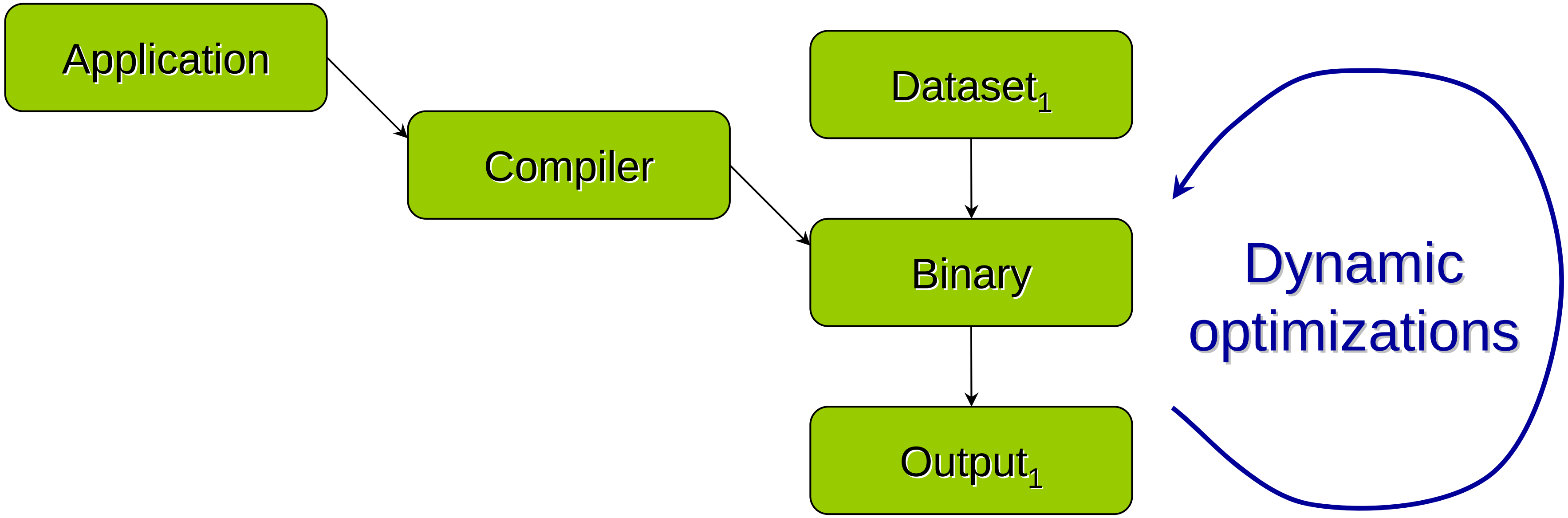


Pros: *run-time information,*

potentially more than one dataset

Current methods

Some existing solutions:



Pros: *run-time information,*

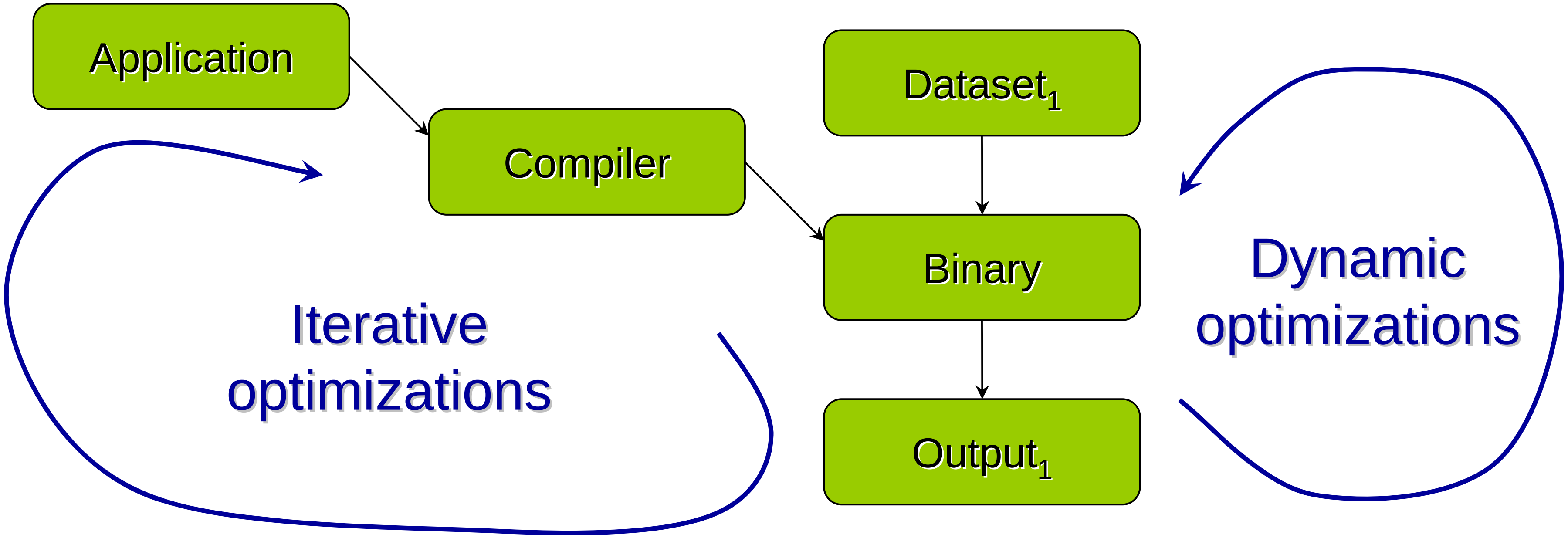
potentially more than one dataset

Cons: *restrictions on optimization time,*

simple optimizations

Current methods

Some existing solutions:



Pros: *run-time information,*

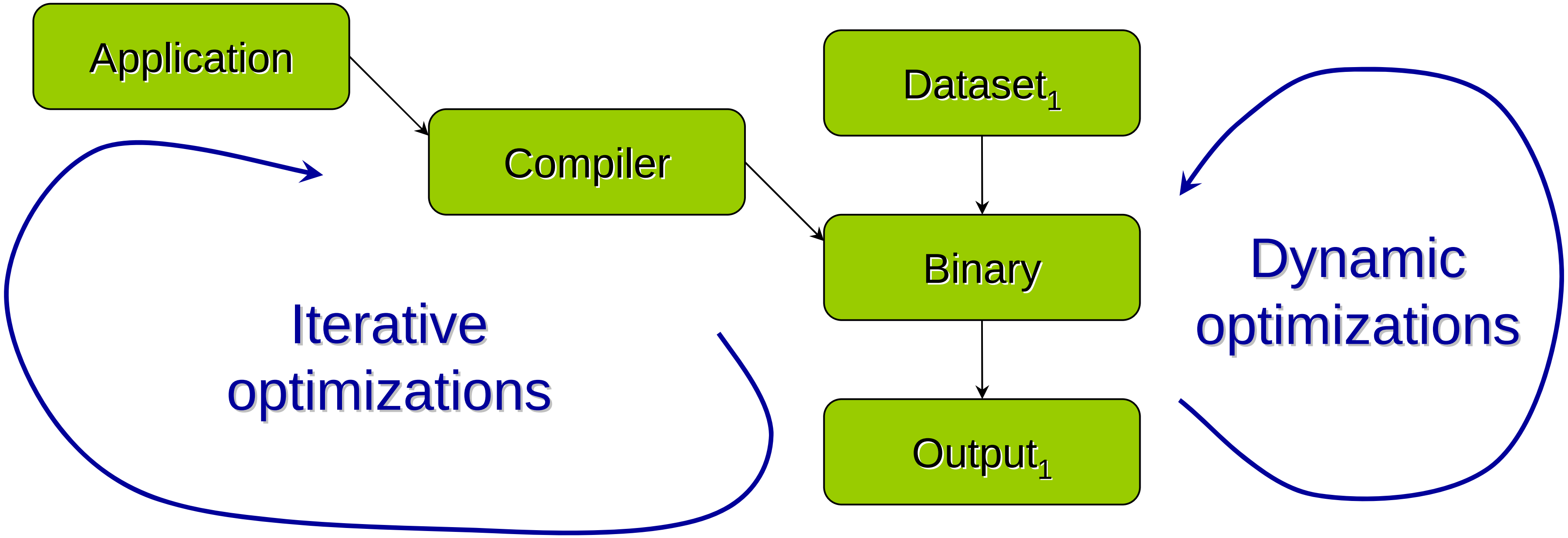
potentially more than one dataset

Cons: *restrictions on optimization time,*

simple optimizations

Current methods

Some existing solutions:



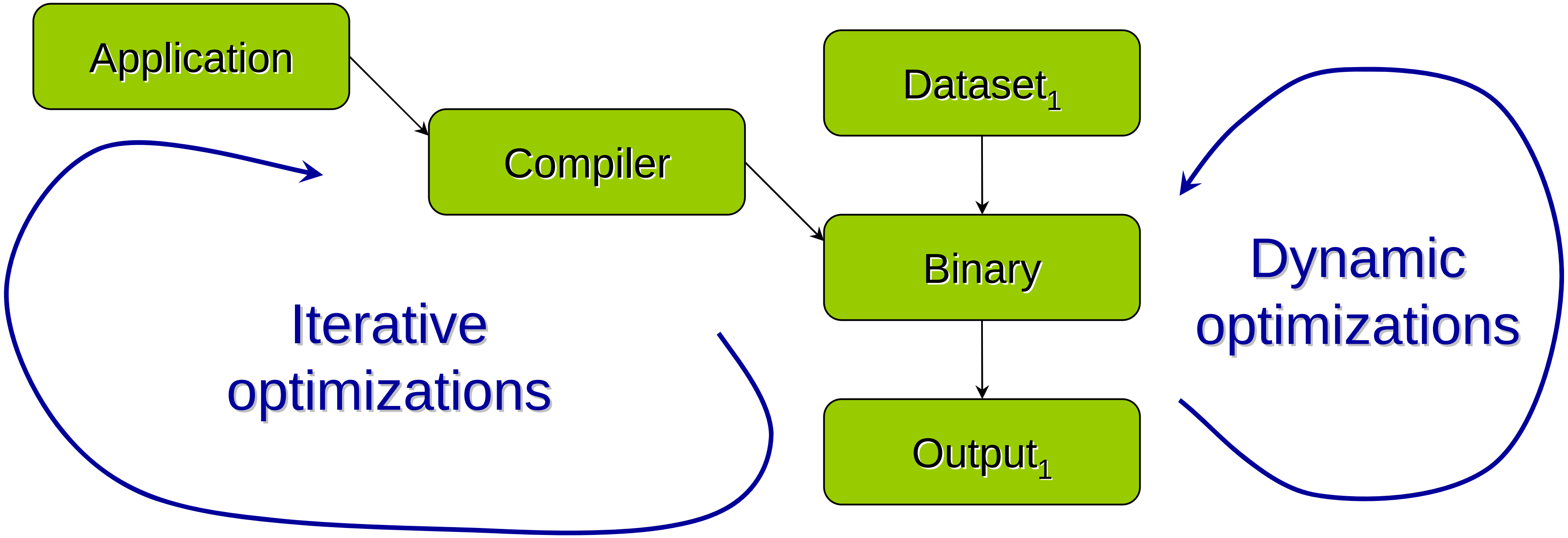
Pros: *powerful transformation space exploration*

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*

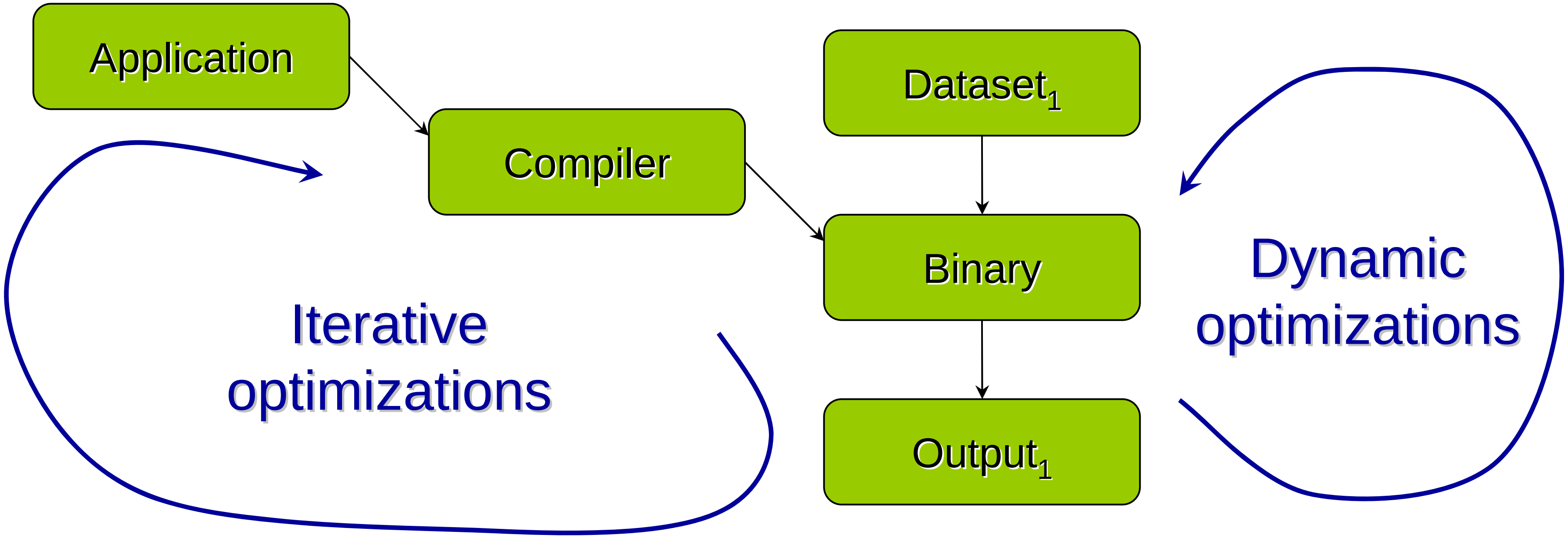
Cons: *slow, one dataset*

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

Cons: *restrictions on optimization time, simple optimizations*

Current methods

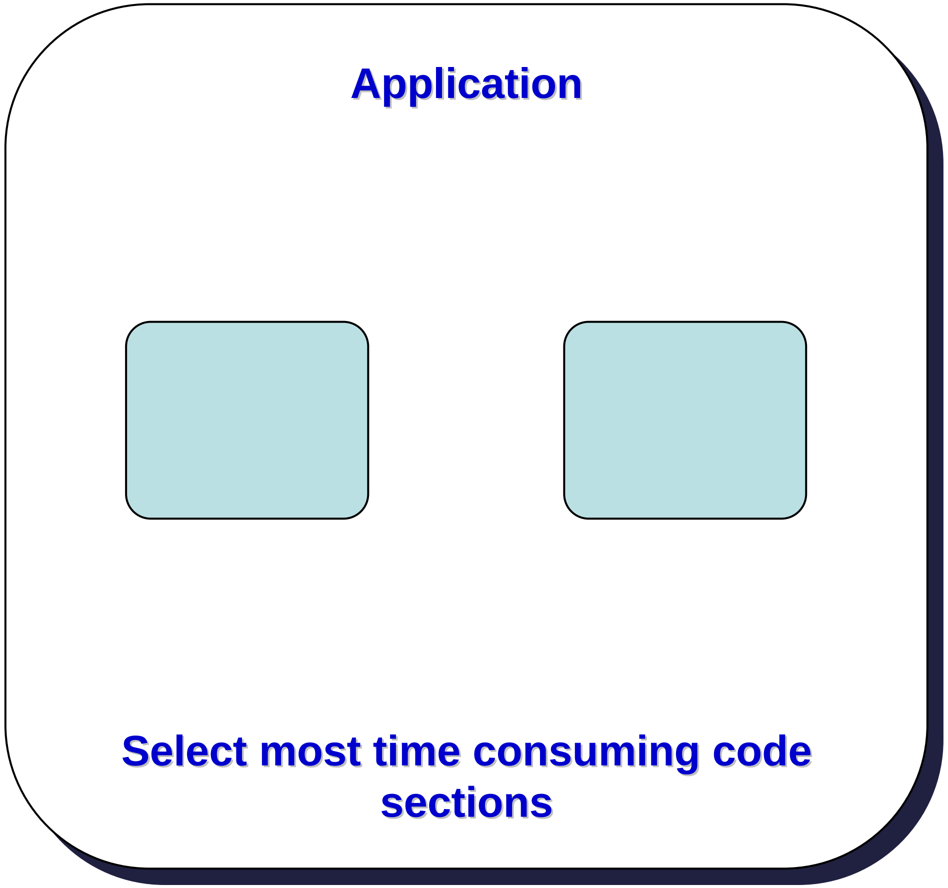
Can we combine both?



Combination of

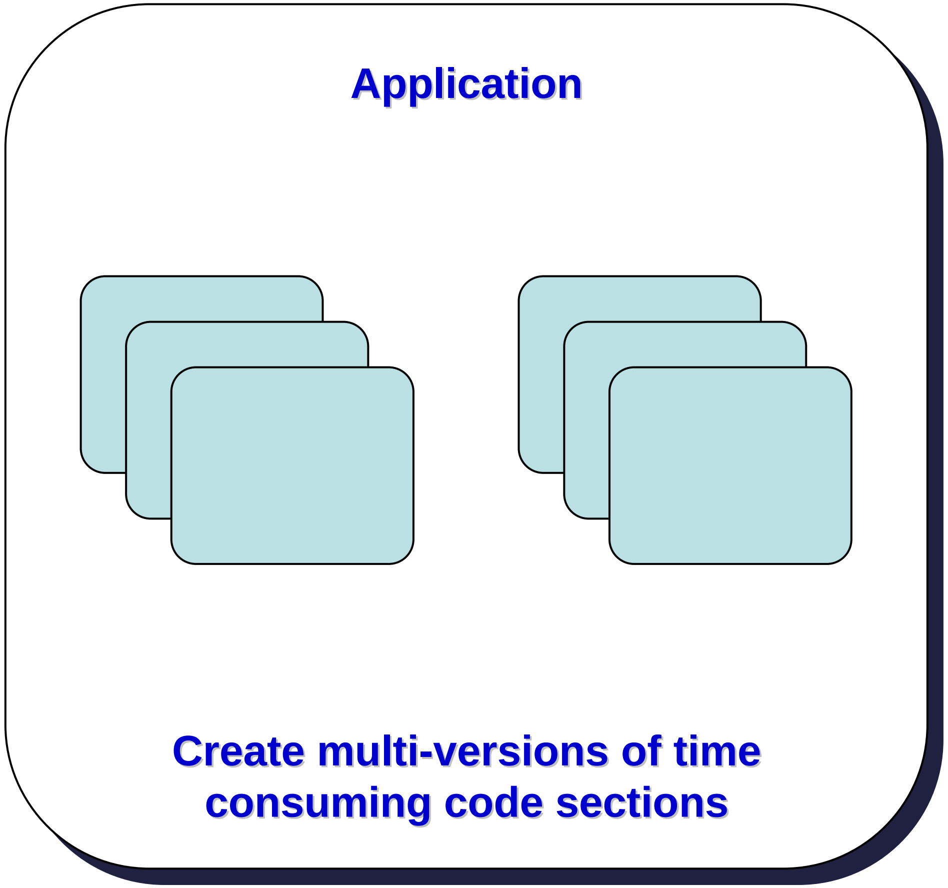
powerful transformation space exploration,
run-time information
self-adaptable code

Our approach: static multiversioning



Select most time consuming code sections

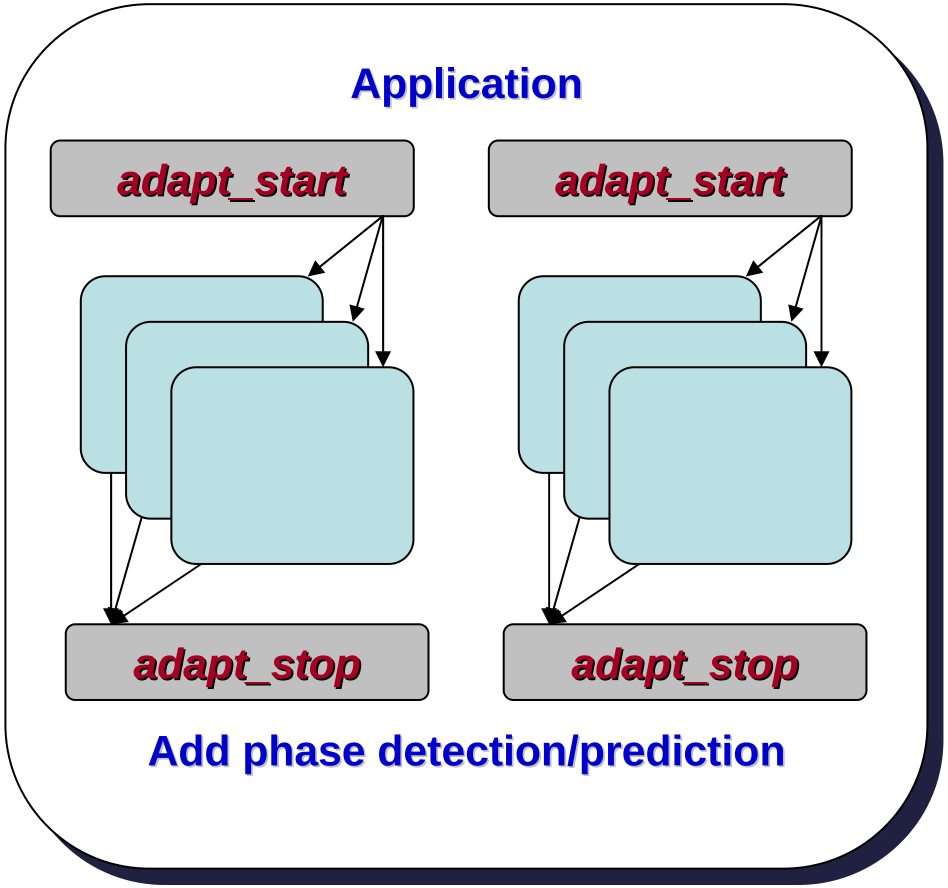
Our approach: static multiversioning



Application

Create multi-versions of time consuming code sections

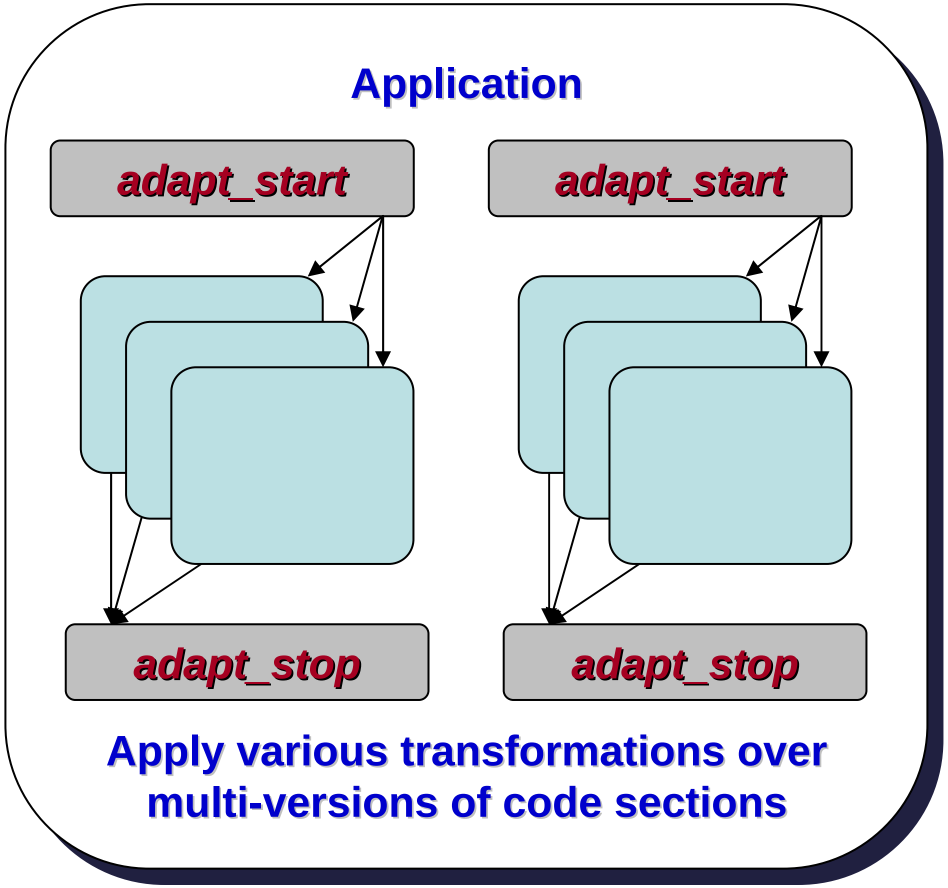
Our approach: static multiversioning



Add phase detection/prediction

Our approach: static multiversioning

Transformations



Application

adapt_start

adapt_start

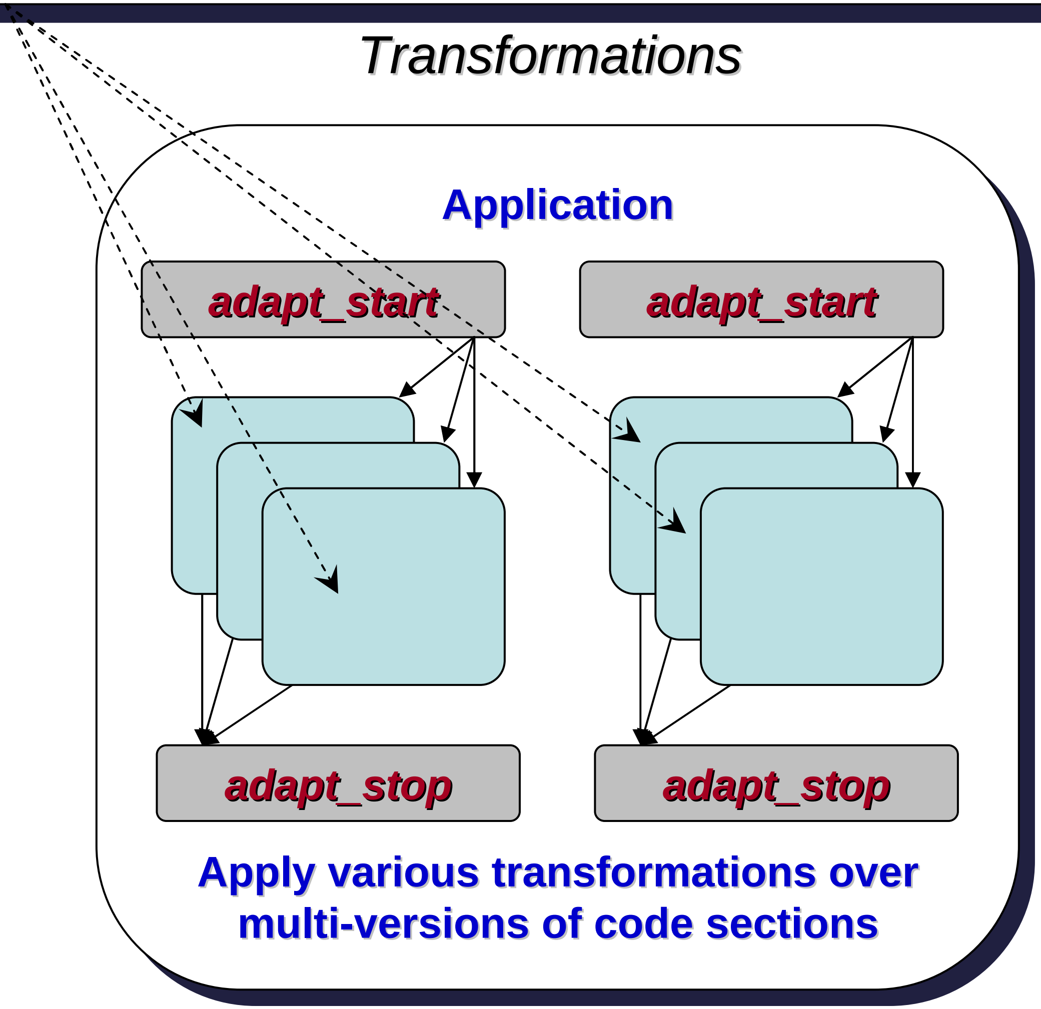
adapt_stop

adapt_stop

Apply various transformations over multi-versions of code sections

Our approach: static multiversioning

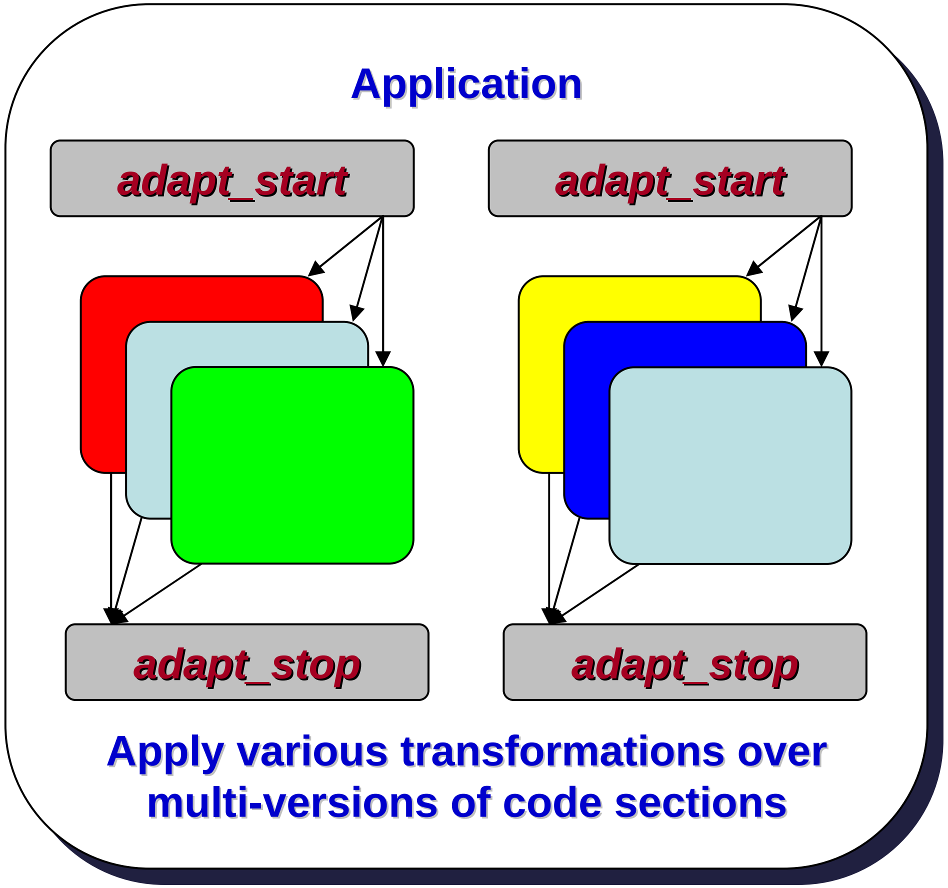
Fine-grain internal compiler (PathScale, Open64, ORC, gcc) transformations using Interactive Compilation Interface (ICI)



Apply various transformations over multi-versions of code sections

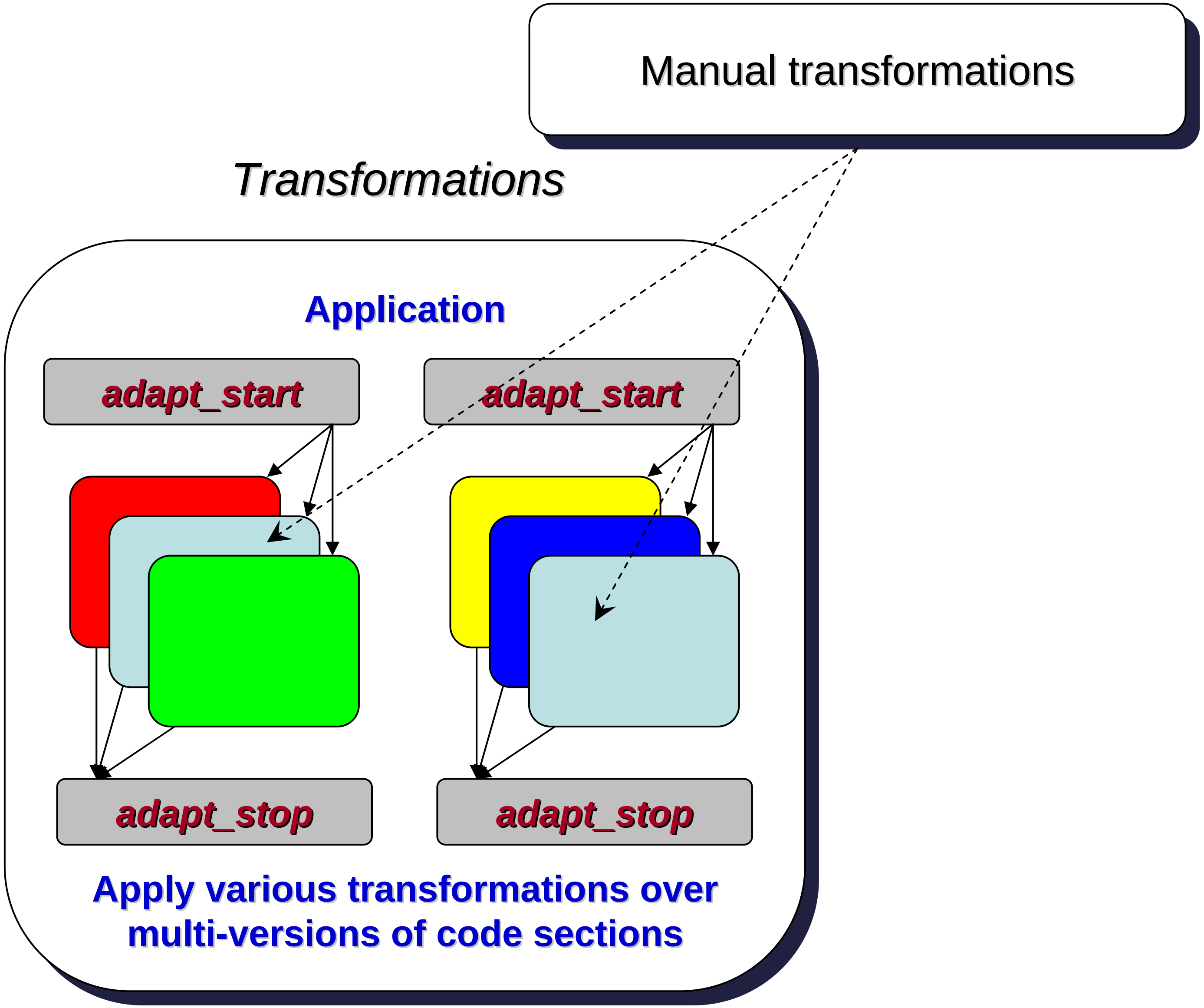
Our approach: static multiversioning

Transformations



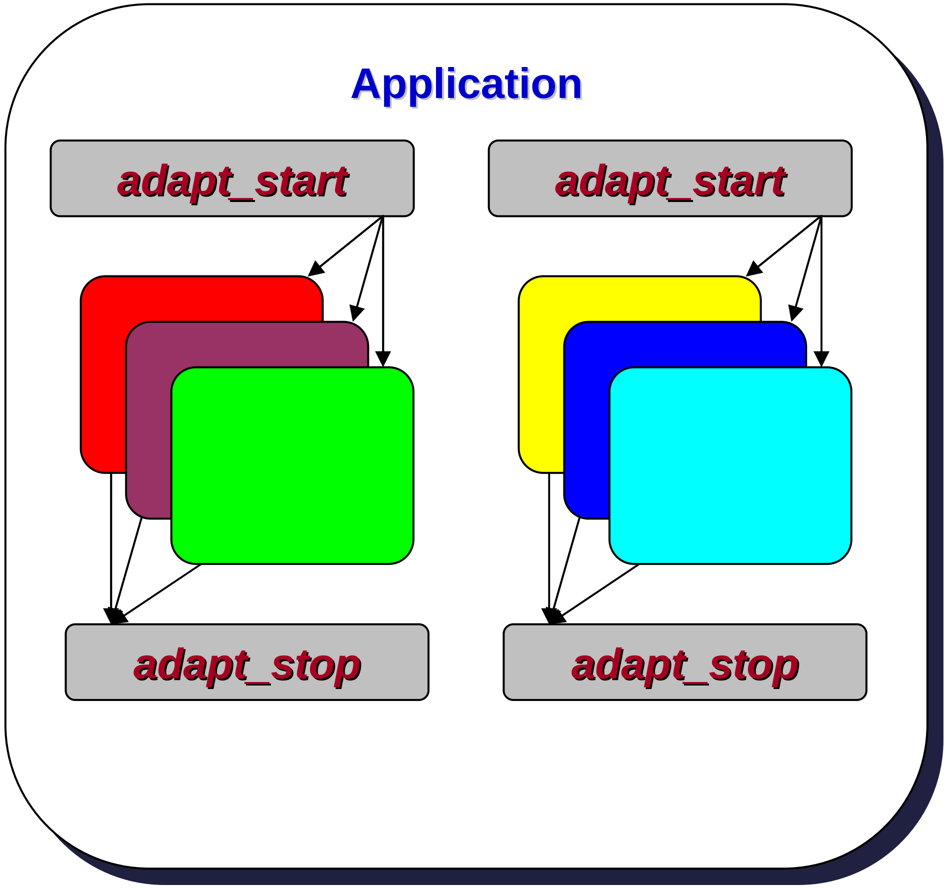
Apply various transformations over multi-versions of code sections

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

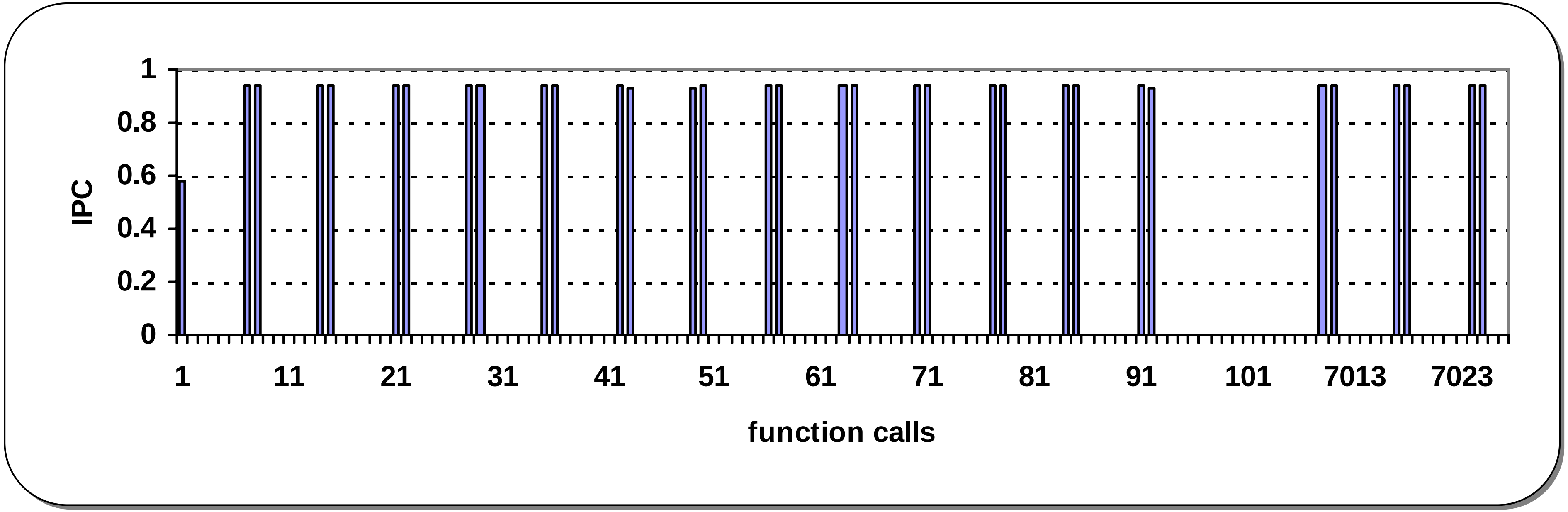
Depends on program behaviour

Programs with regular behavior

Programs with irregular behavior

Adaptation for regular behaviour

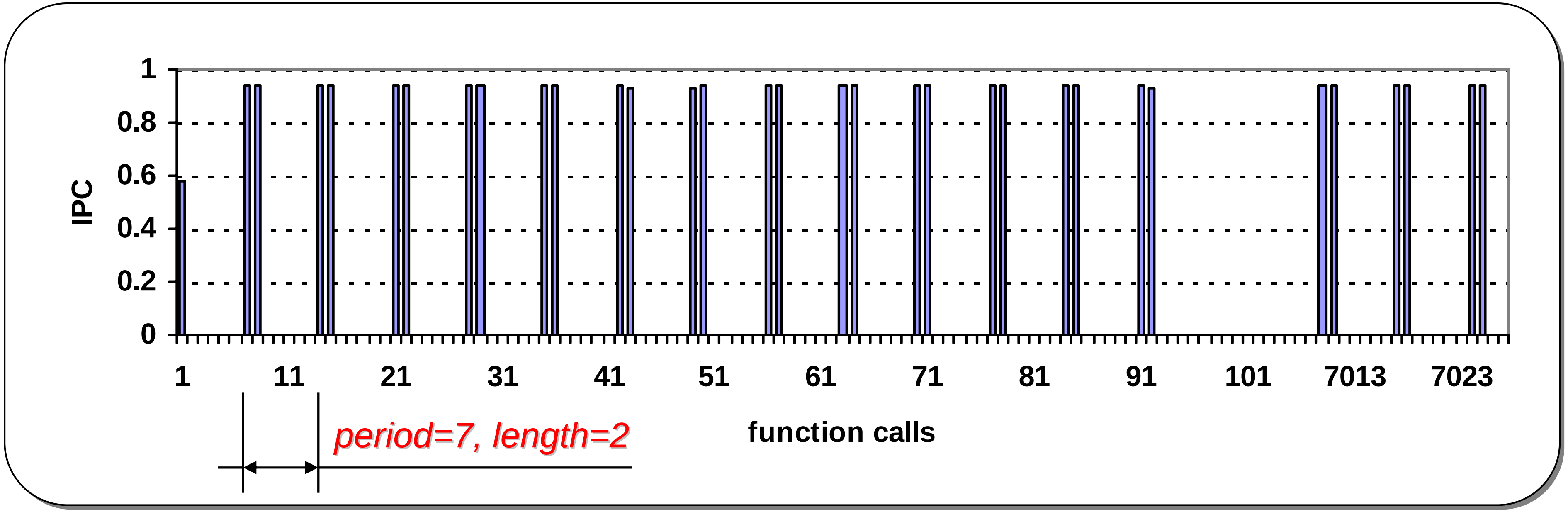
IPC for subroutine resid of benchmark mgrid across calls



- 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

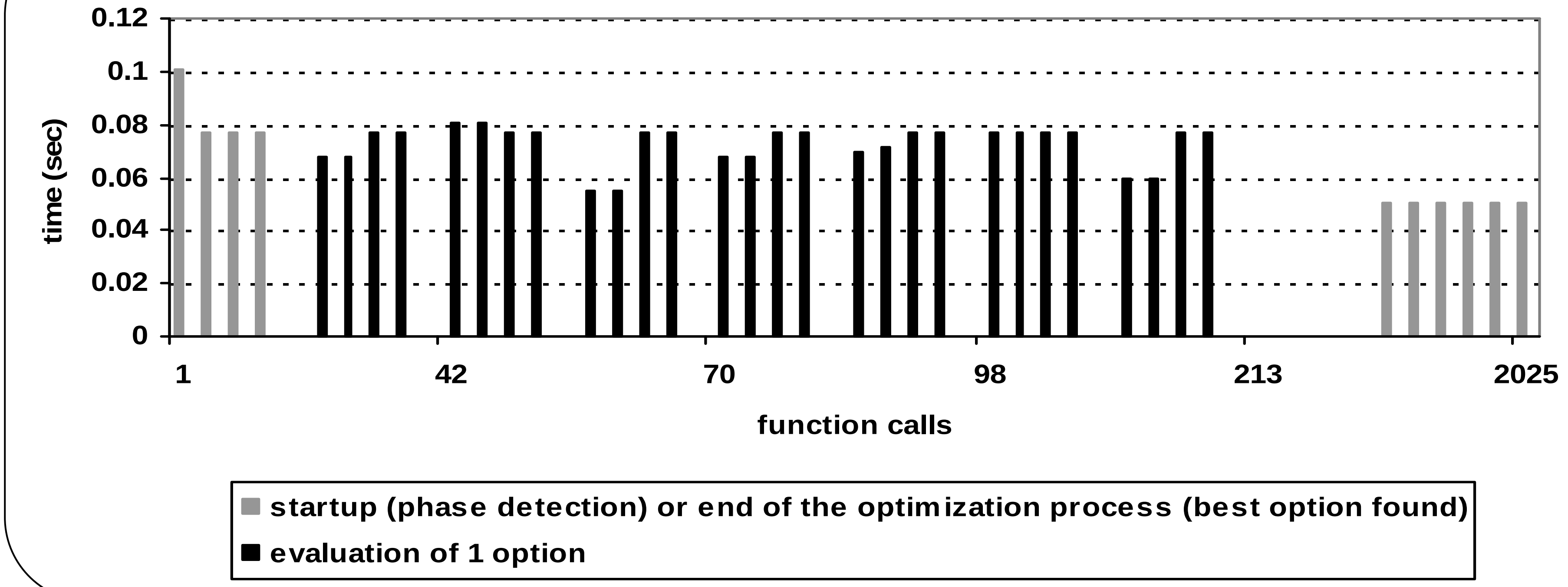


- Detect regular (stable) patterns of behaviour (phases) - we define stability as 3 consecutive or periodic executions with the same IPC

- 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



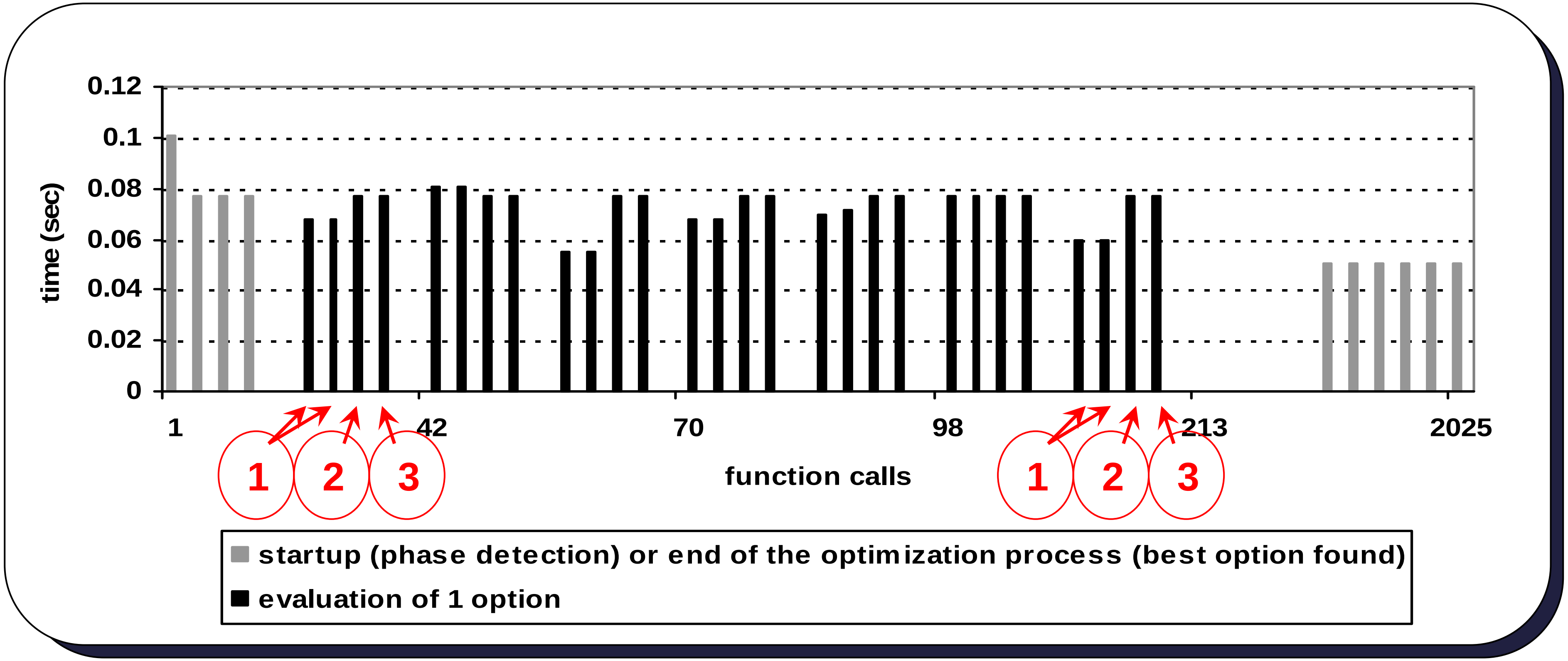
1) Consider new code version evaluated after 2 consecutive executions of the code section with the same performance

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

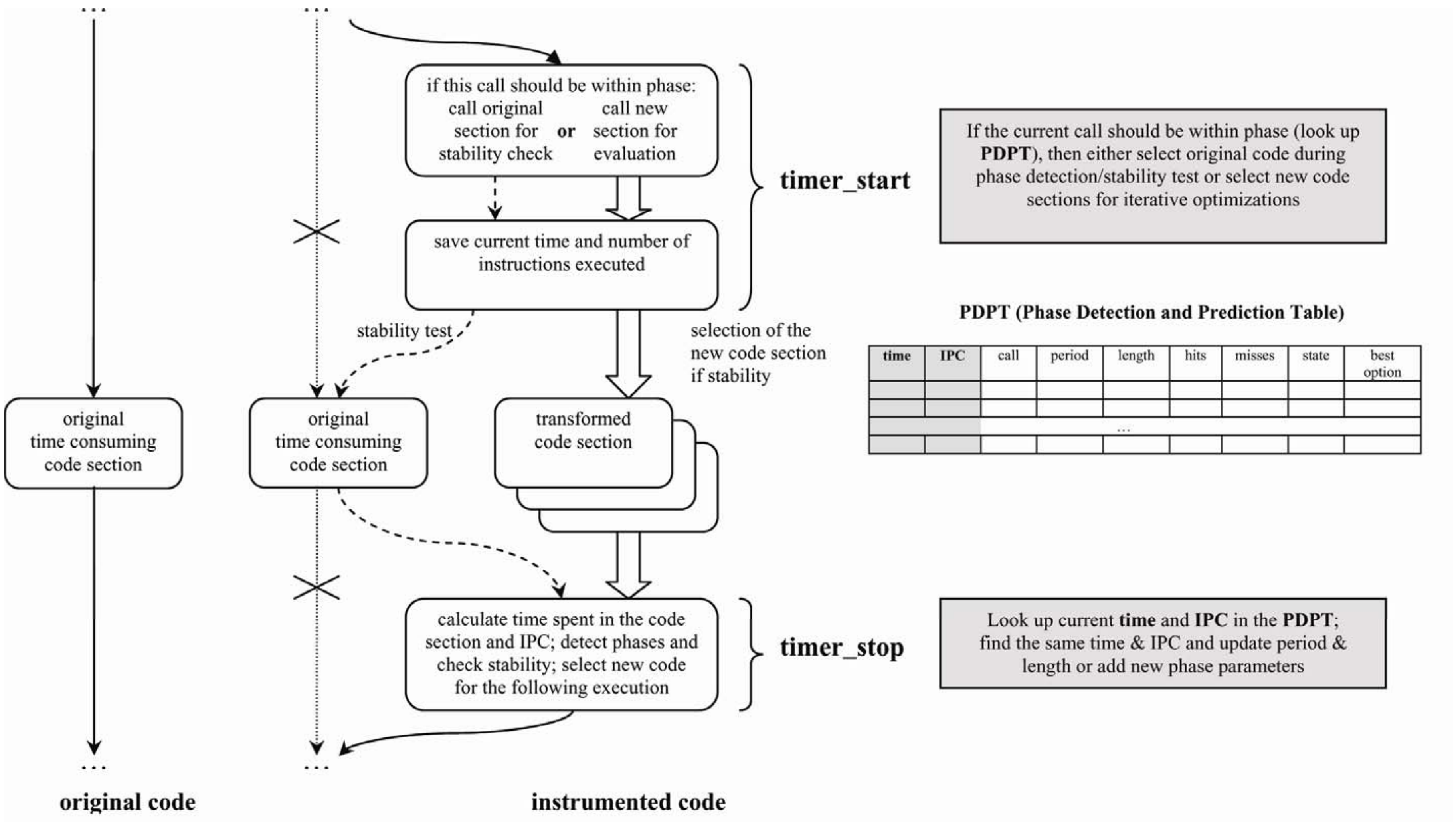


1) Consider new code version evaluated after 2 consecutive executions of the code section with the same performance

2) Ignore one next execution to avoid transitional effects

3) Check baseline performance (to verify stability prediction)

Adaptation for regular behaviour


...
...
if this call should be within phase:
call original section for stability check
or
call new section for evaluation
save current time and number of instructions executed
stability test
selection of the new code section if stability
original time consuming code section
original time consuming code section
transformed code section
calculate time spent in the code section and IPC; detect phases and check stability; select new code for the following execution
timer_start
timer_stop
...
...
original code
instrumented code

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

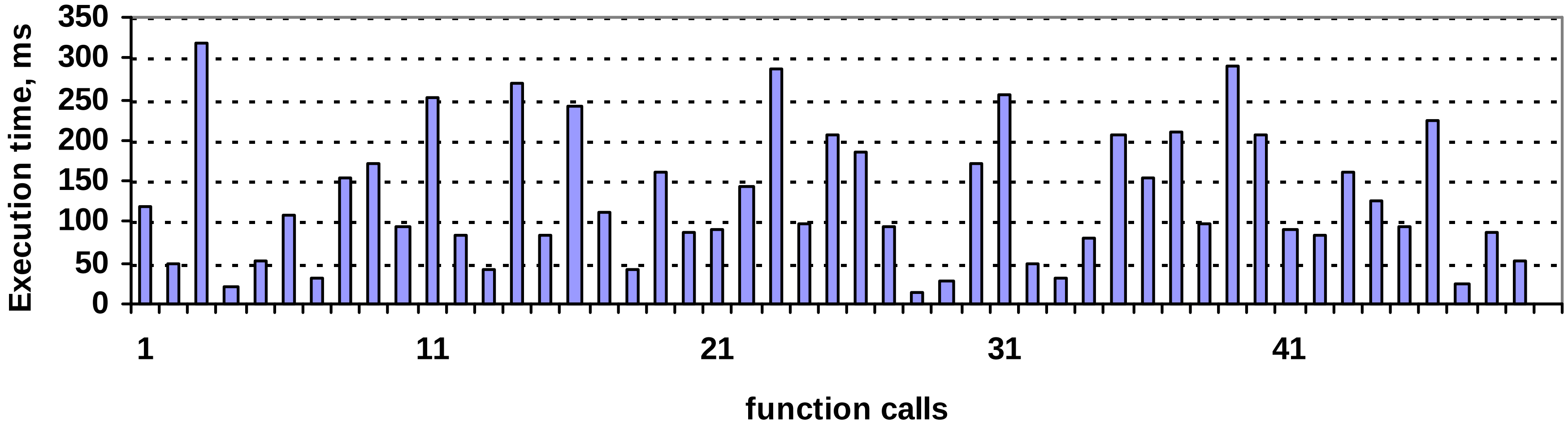
PDPT (Phase Detection and Prediction Table)

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

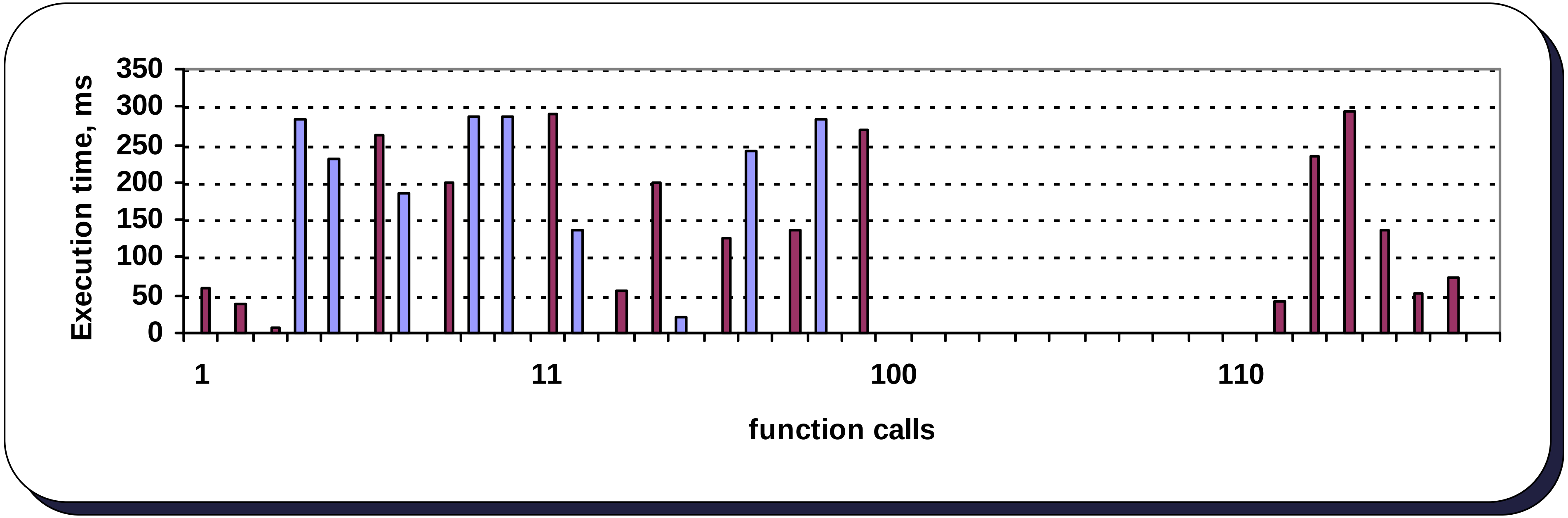
Adaptation for irregular behaviour

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



Adaptation for irregular behaviour

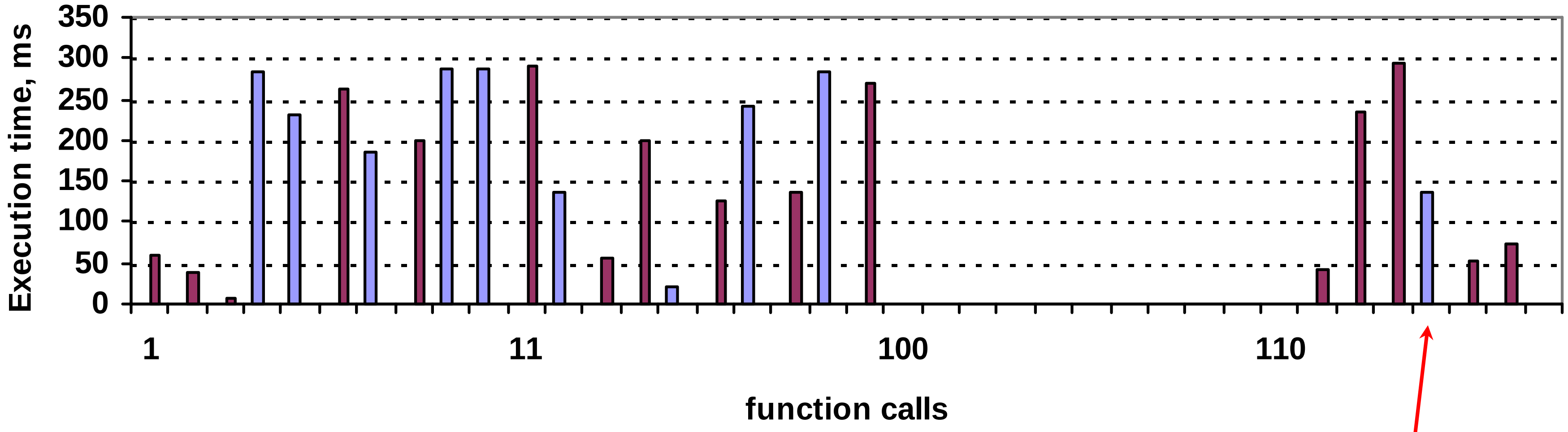
Execution time for library subroutine matmul (with 2 different versions)



- 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

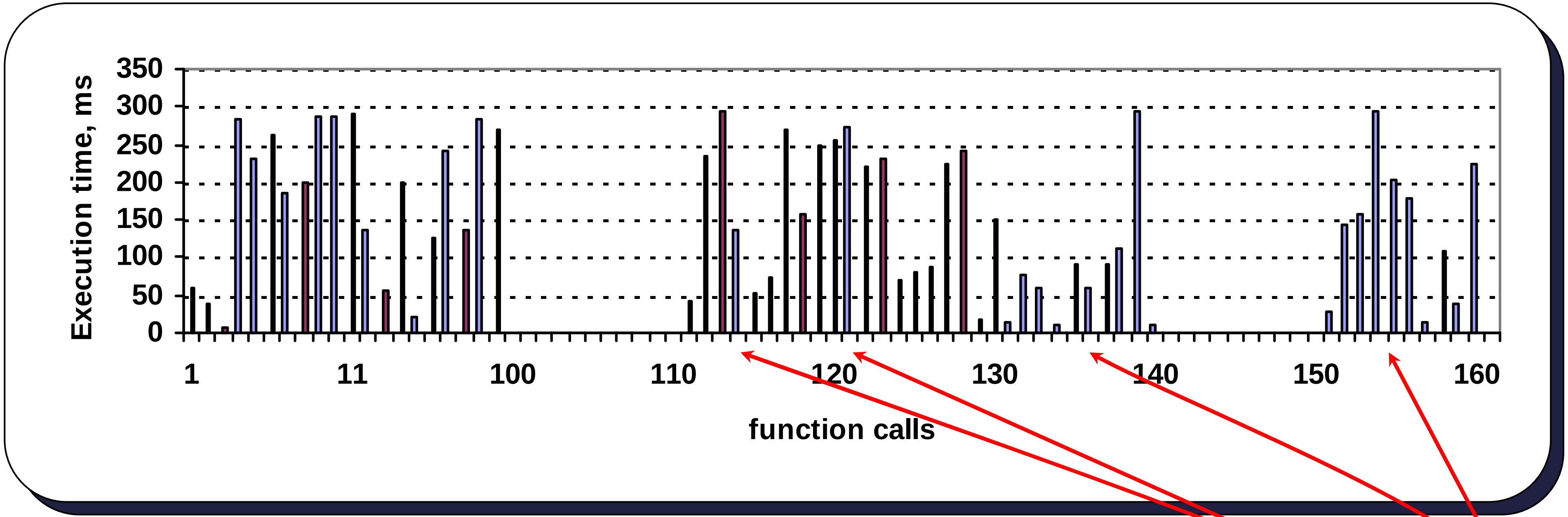
Execution time for library subroutine matmul (with 2 different versions)



- 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

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

Continuous Optimization Framework

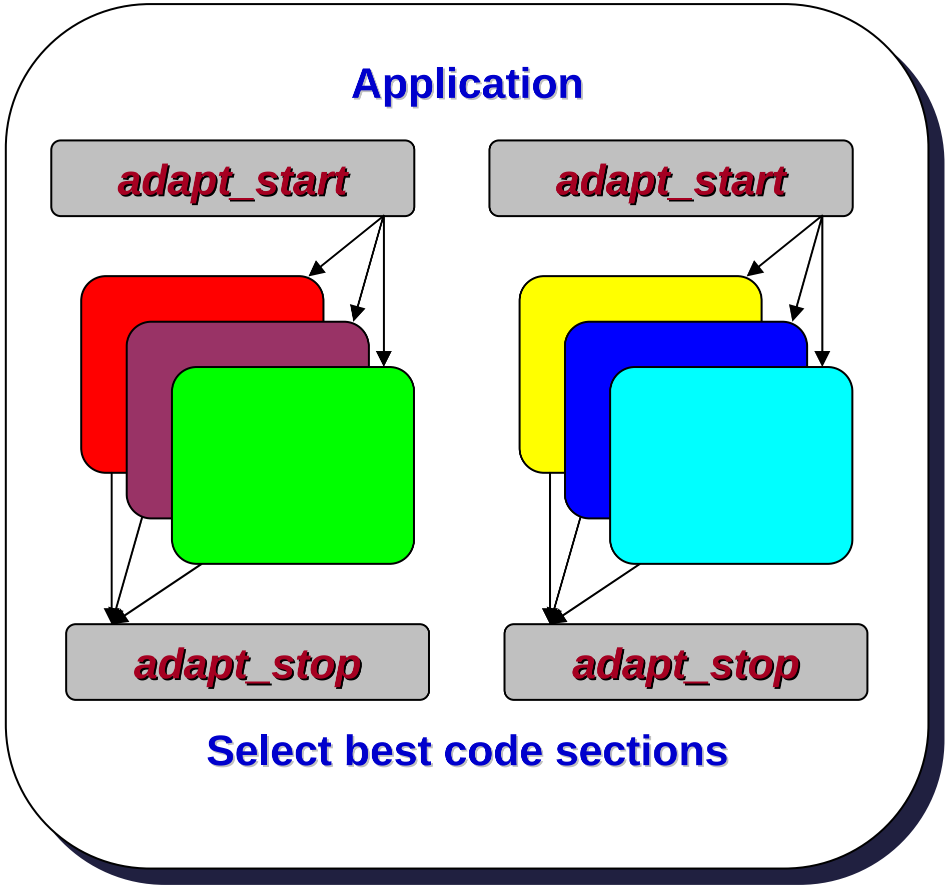
sequence of evaluations: speedups $s_1, s_2, \ldots s_n$

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

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

Continuously monitor the variance to detect convergence across executions

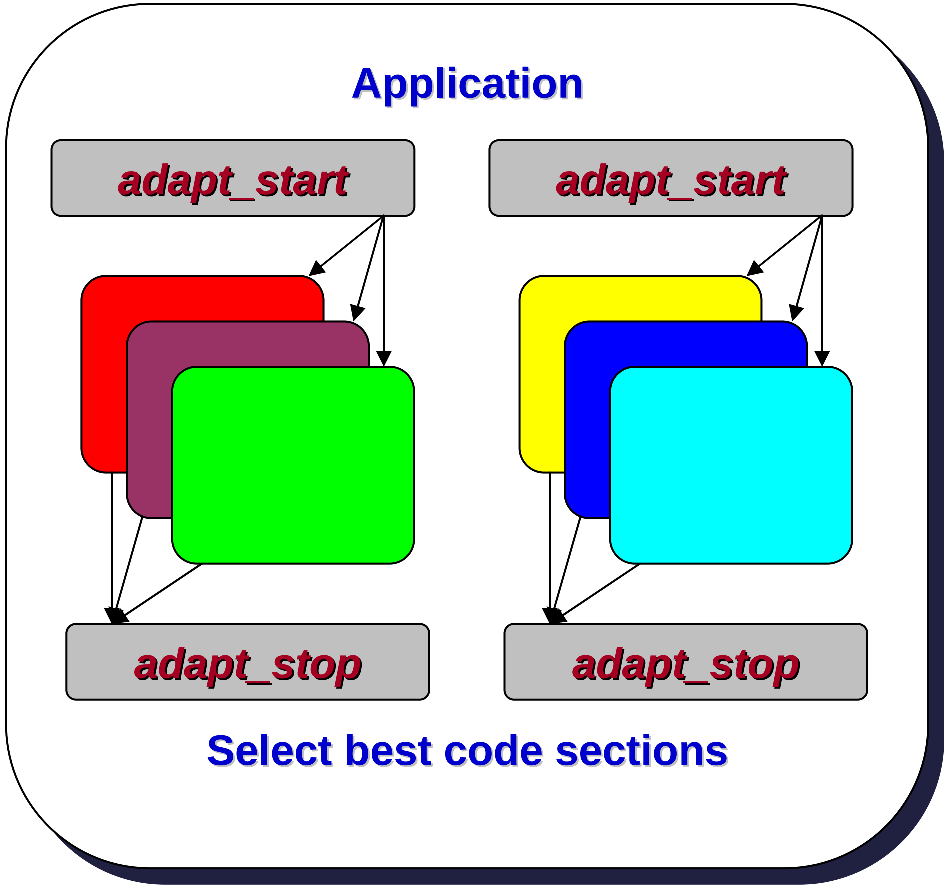
Removing adaptation overhead



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

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

Removing adaptation overhead

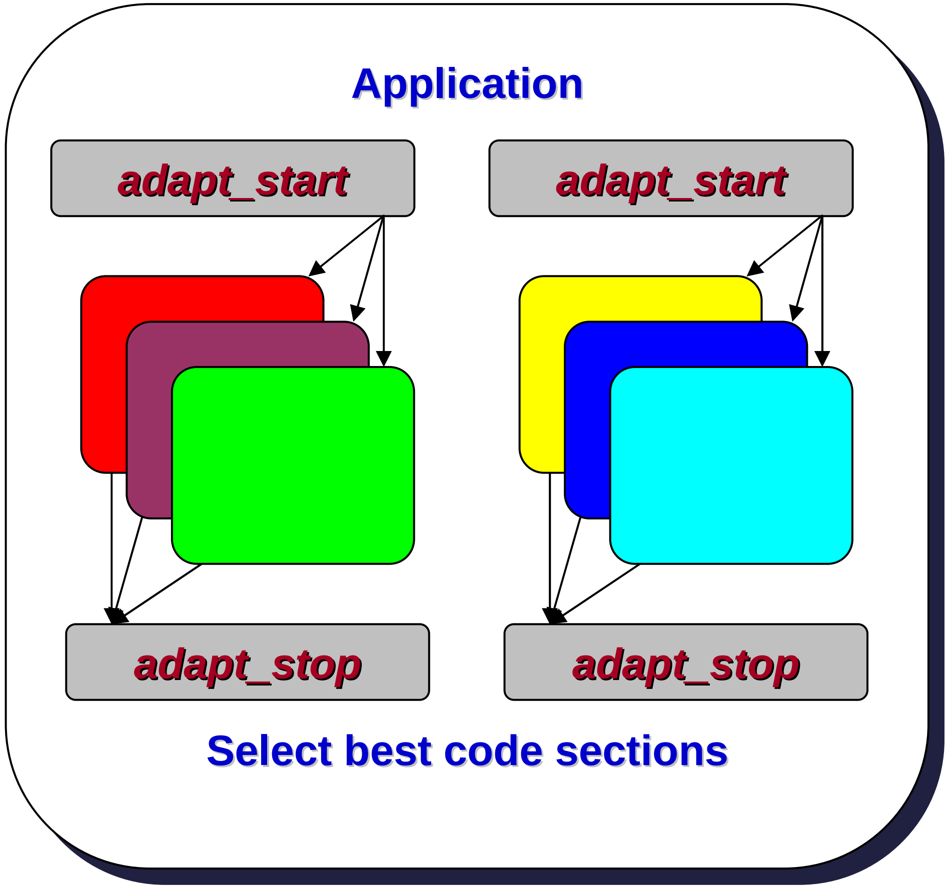


Calls to adaptation routines are not direct but through array of functions:

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:

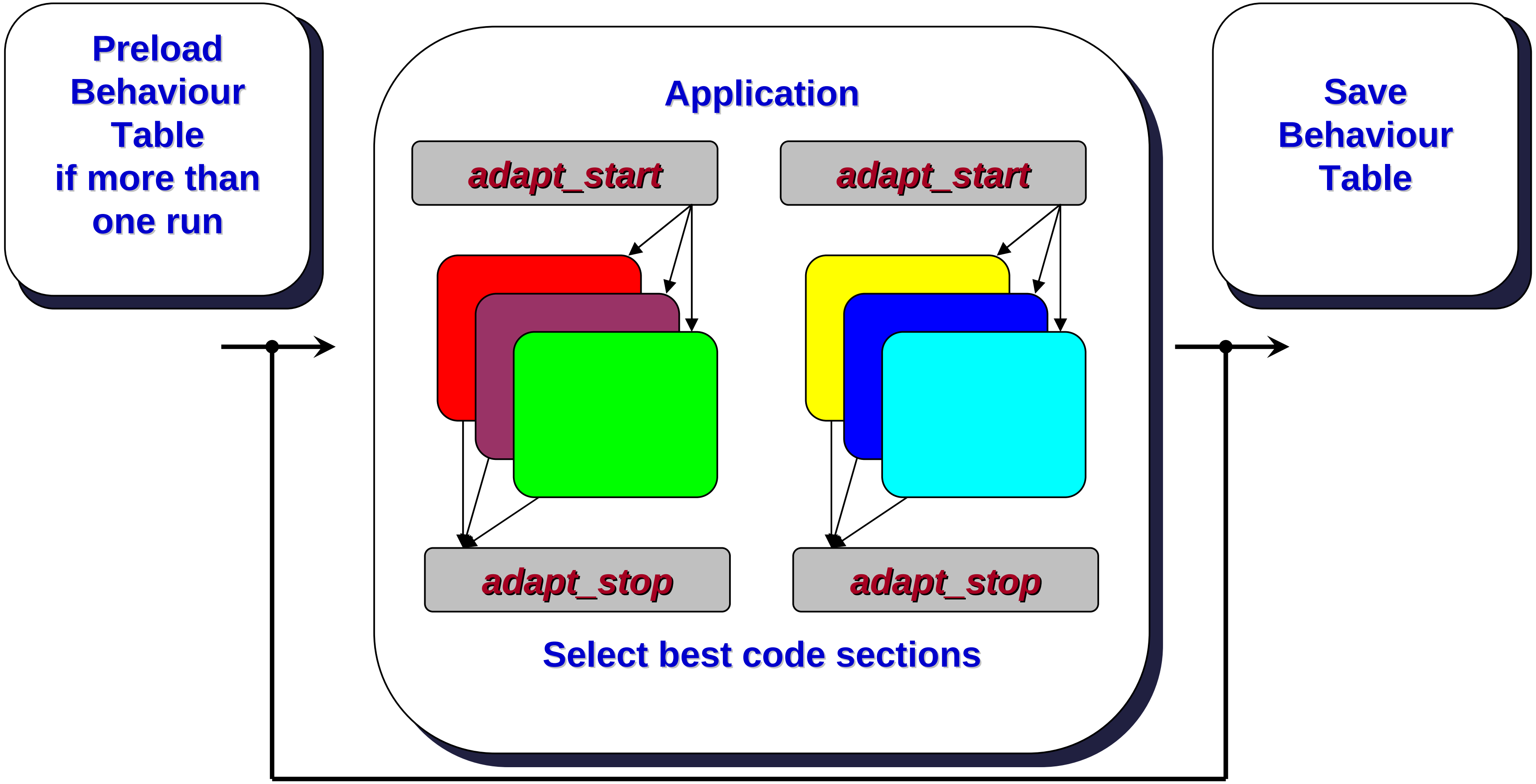
static void (*call1[..])();
static void (*call2[..])();

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:



Preload Behaviour Table if more than one run

Application

adapt_start ***adapt_start***

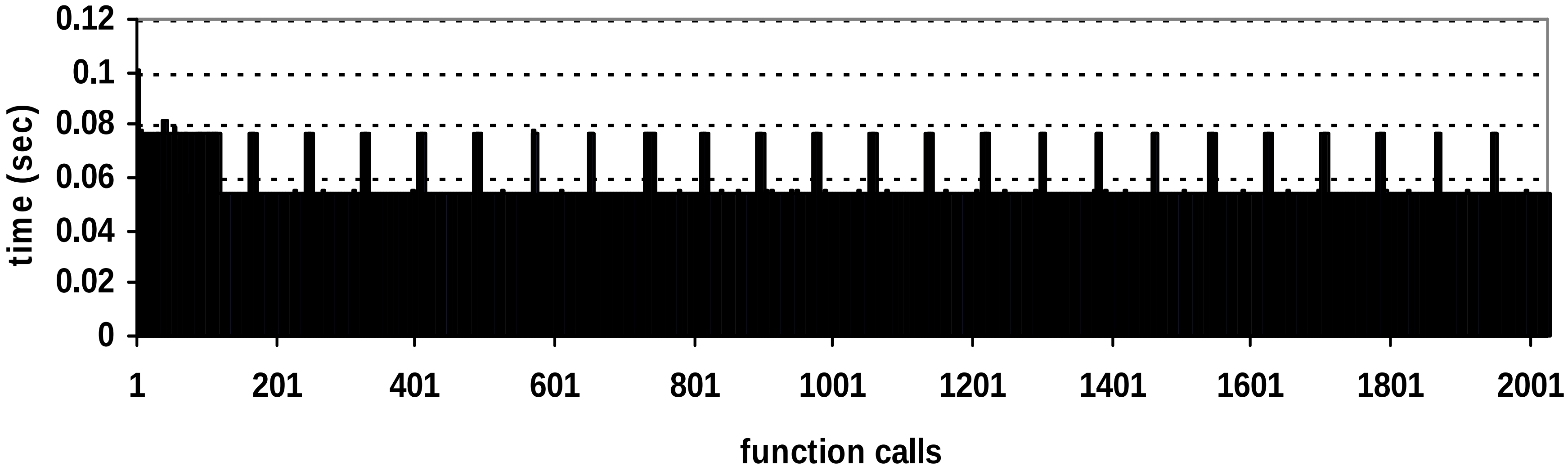
adapt_stop ***adapt_stop***

Select best code sections

Save Behaviour Table

Continuous optimization and adaptation

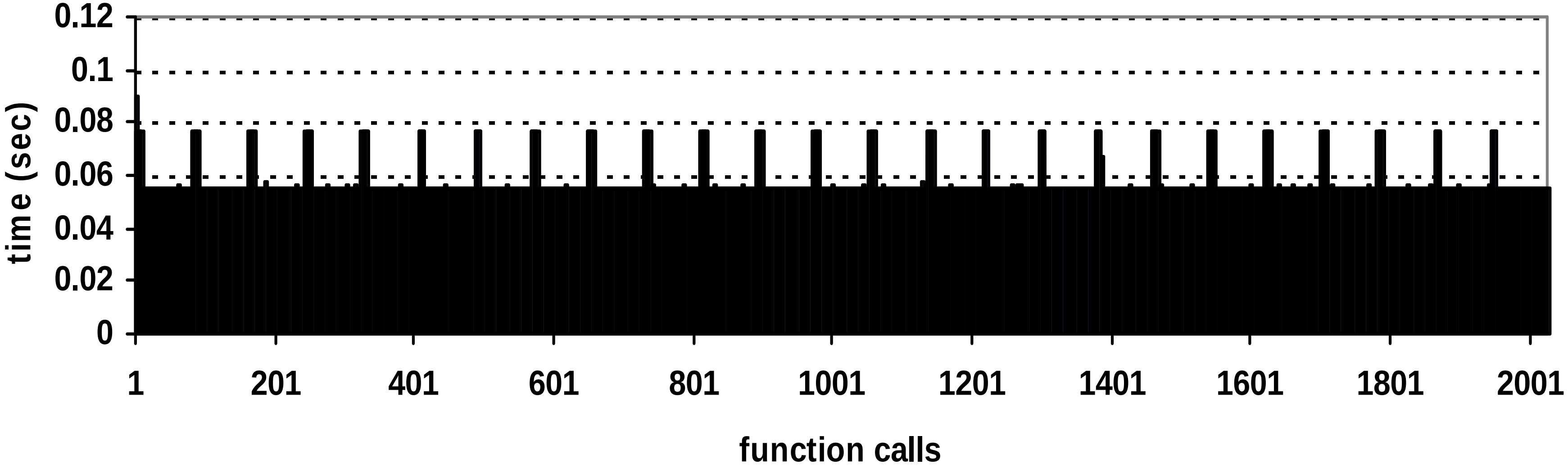
Execution times for subroutine resid of benchmark mgrid across calls



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:	10.5 s.	7.5 s.	
profile:			susan_edges0: 3.7 s. susan_edges1: 2.5 s.

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;
- 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, with easy debugging

Conclusions

However:

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

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 *process of designing a compiler*

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

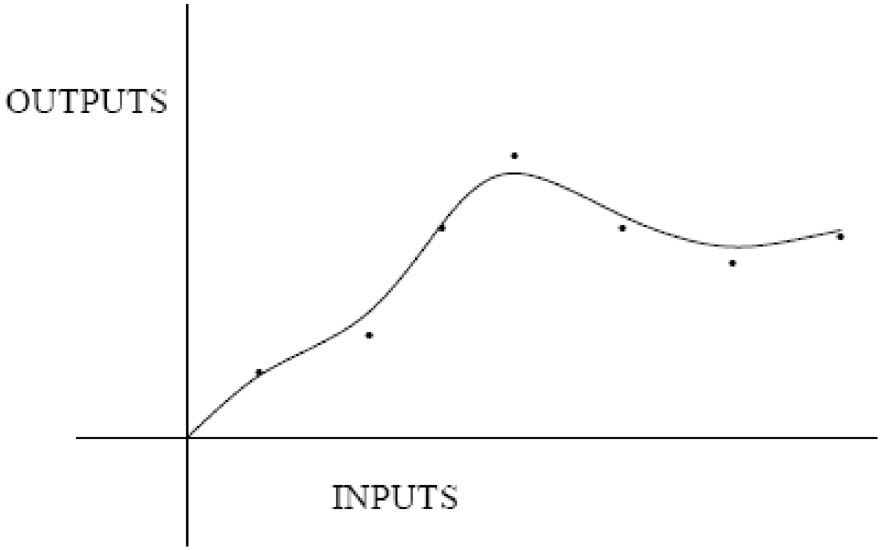
- Iterative compilation goes beyond this with no a priori knowledge but is not suitable for general compilations and does not adapt to changing data

- 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?

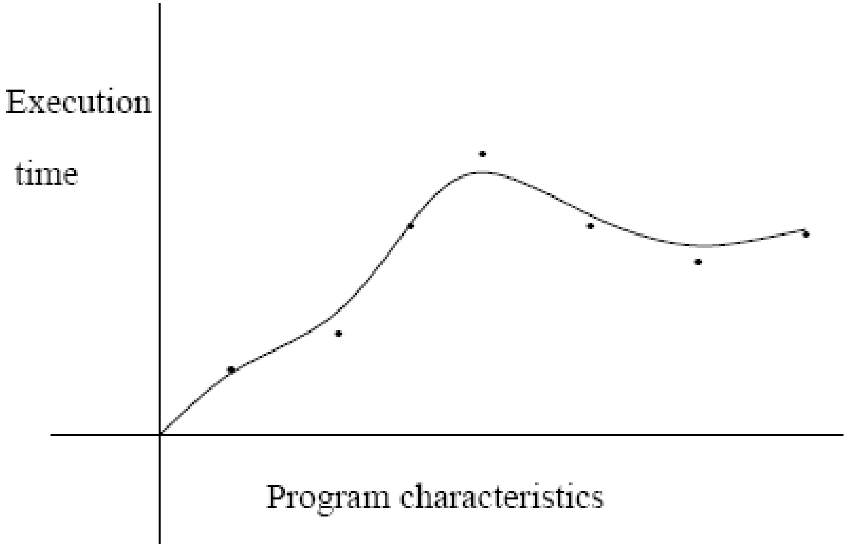
- In a very simplistic sense machine learning can be considered as sophisticated form of curve fitting

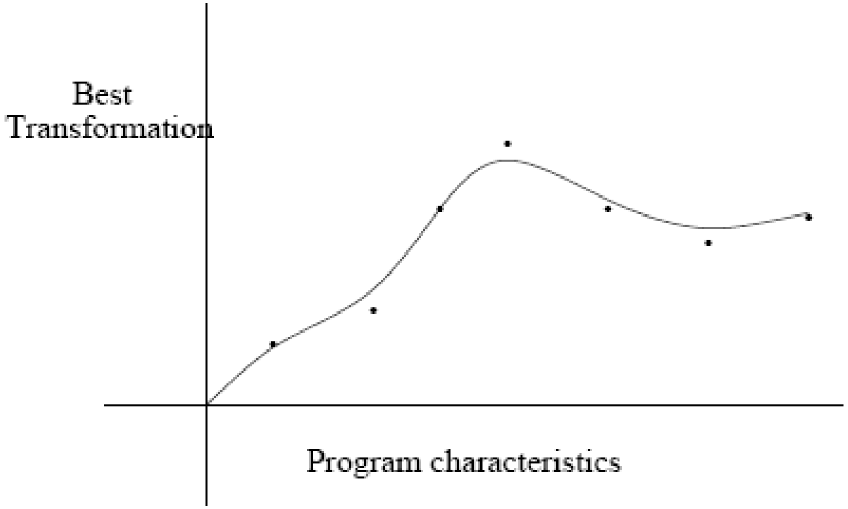


Machine learning

- The inputs are characteristics of the program and processor. Outputs, the optimization function we are interested in, execution time power or code size

- Theoretically predict future behavior and find the best optimization





Global optimization and predictive modeling

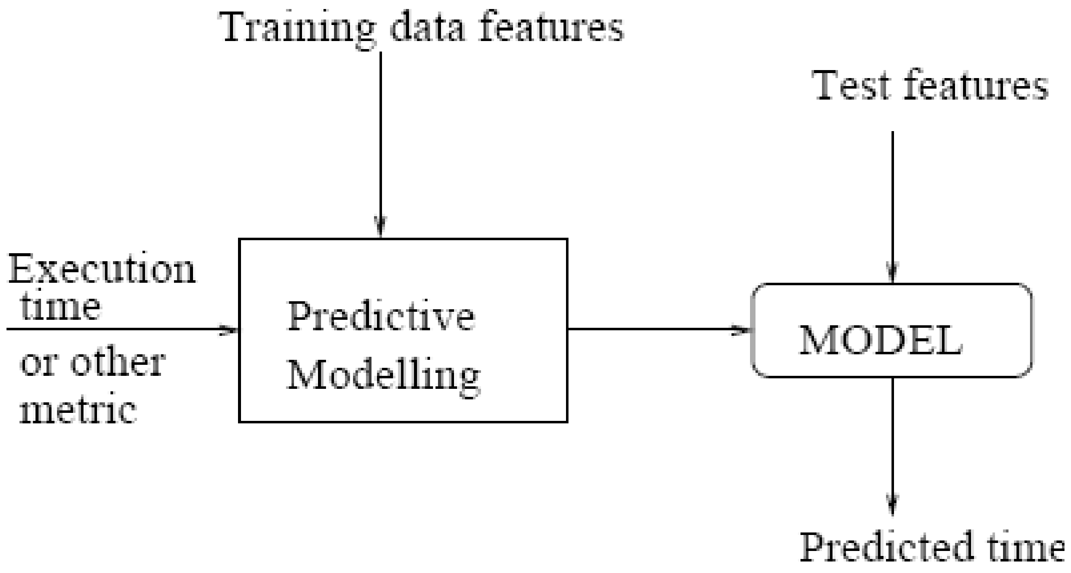
- For our purposes it is possible to consider machine learning as *global optimization* and *predictive modeling*

- *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

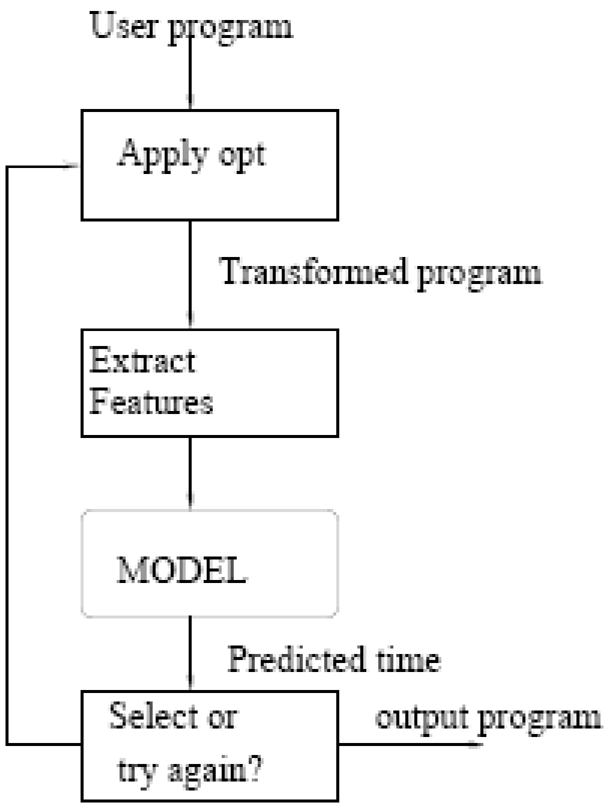


- Predictive modeling techniques all have the property that they try to learn a model that describes the correlation between inputs and outputs

- 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

- Nobody has done this yet! Feature selection and accuracy are main problems!

Training data

- Crucial to this working 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!

- 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 the easily used at compile time. Extract the features of the loop and see which section they belong too

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

Learning to unroll

```
do i = 2, 100

   a(i) = a(i) + a(i−1) + a(i+1)

enddo
```

features

statements	1
aritmetic op	2
iterations	99
array access	4
resuses	3
ifs	0

- 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

Learning to inline

- Inlining is the number one optimization in JIT compilers. Many papers from IBM on adaptive algorithms to get it right in Jikes

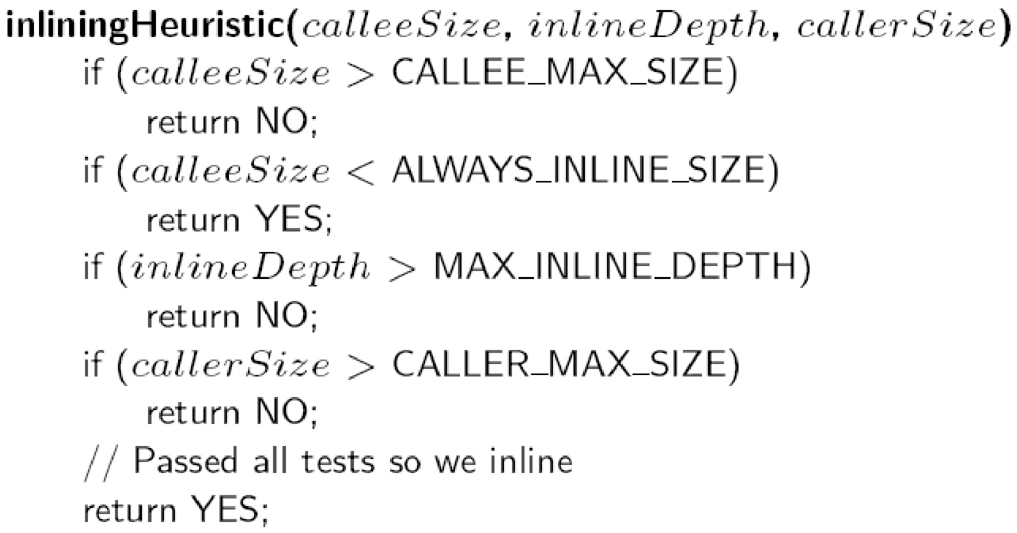
- Can we use machine learning to improve this highly tuned heuristic? Tough problem. Similar to meta-optimization goal

- Cavazos (2005) looked at automatically determining inline heuristics under different *scenarios*

- Opt vs Adapt - different user compiler options. Total time vs run time vs a balance - compile time is part of runtime

- x86 vs PPC - can the strategy port across platform

Learning a heuristic

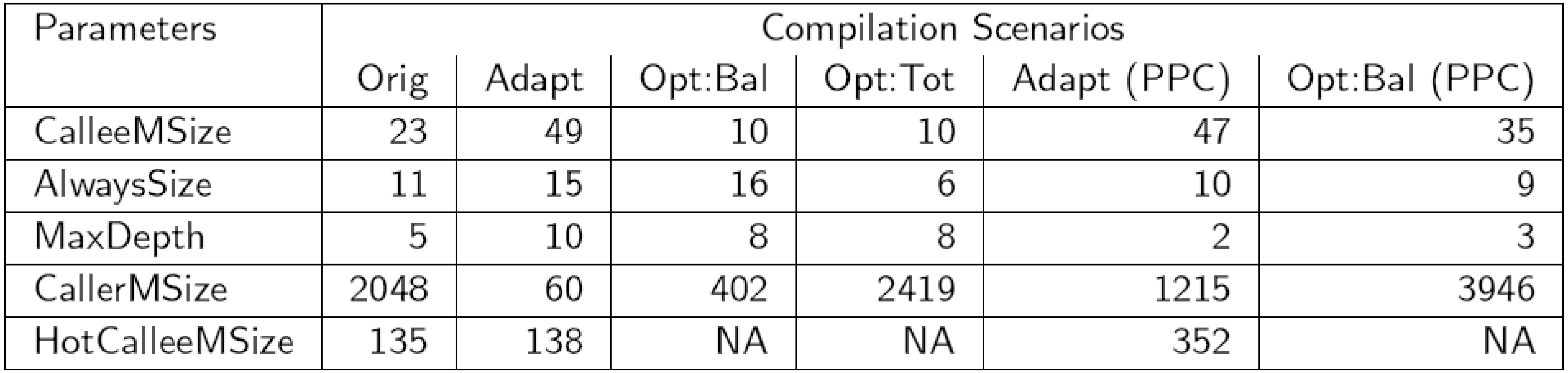


```
inliningHeuristic($calleeSize$, $inlineDepth$, $callerSize$)
    if ($calleeSize >$ CALLEE_MAX_SIZE)
        return NO;
    if ($calleeSize <$ ALWAYS_INLINE_SIZE)
        return YES;
    if ($inlineDepth >$ MAX_INLINE_DEPTH)
        return NO;
    if ($callerSize >$ CALLER_MAX_SIZE)
        return NO;
    // Passed all tests so we inline
    return YES;
```

- Focus on tuning parameters of an existing heuristic rather than generating a new one from scratch

- Features are *dynamic*. Learn off-line and applied heuristic on-line

Parameters found



Parameters	Compilation Scenarios					
	Orig	Adapt	Opt:Bal	Opt:Tot	Adapt (PPC)	Opt:Bal (PPC)
CalleeMSize	23	49	10	10	47	35
AlwaysSize	11	15	16	6	10	9
MaxDepth	5	10	8	8	2	3
CallerMSize	2048	60	402	2419	1215	3946
HotCalleeMSize	135	138	NA	NA	352	NA

- Considerable variation across scenario
- For instance on x86, Bal and Total similar except for the CallerMaxSize
- A priori these values could not be predetermined

Learning to inline

- Initially tried rule induction - failed miserably. Not clear at this stage why

- Difficult to determine whether optimization has impact

- Next used a genetic algorithm to find a good heuristic

- For each scenario asked the GA to find the best geometric mean over the training set. Using search for learning

- Training set used - Specjvm98, test set - DaCapo including Specjbb

- Focused learning on choosing the right numeric parameters of a fixed heuristic

- Applied this to a test set comparing against IBM heuristic

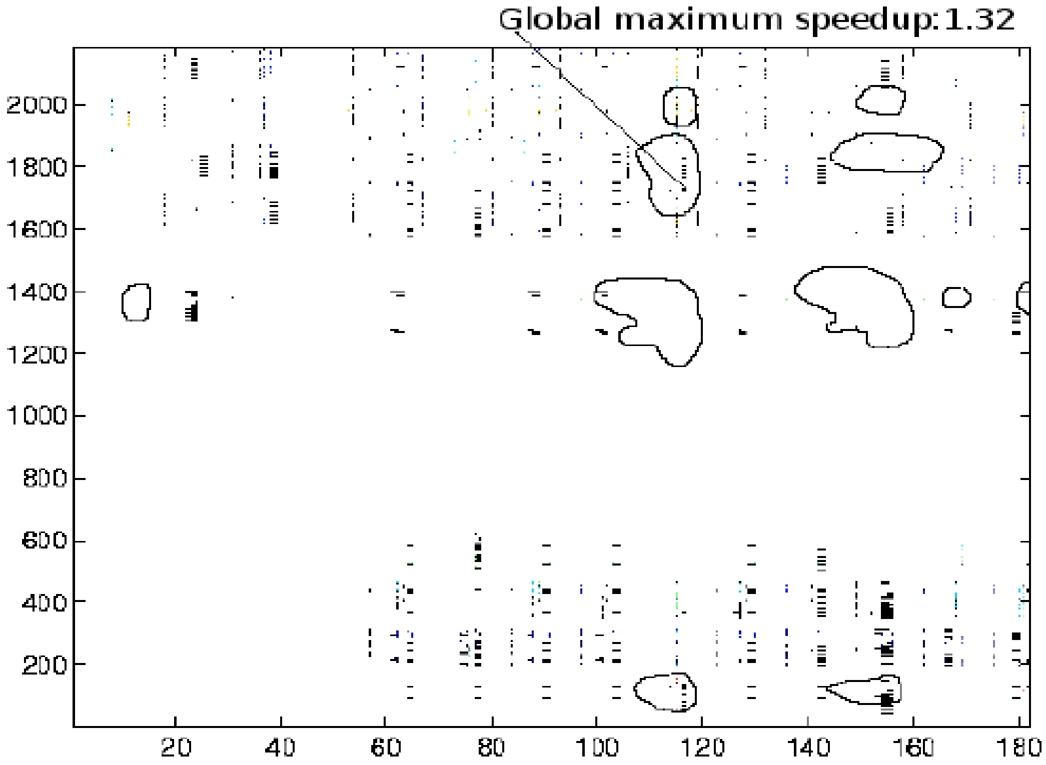
More general approaches?

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

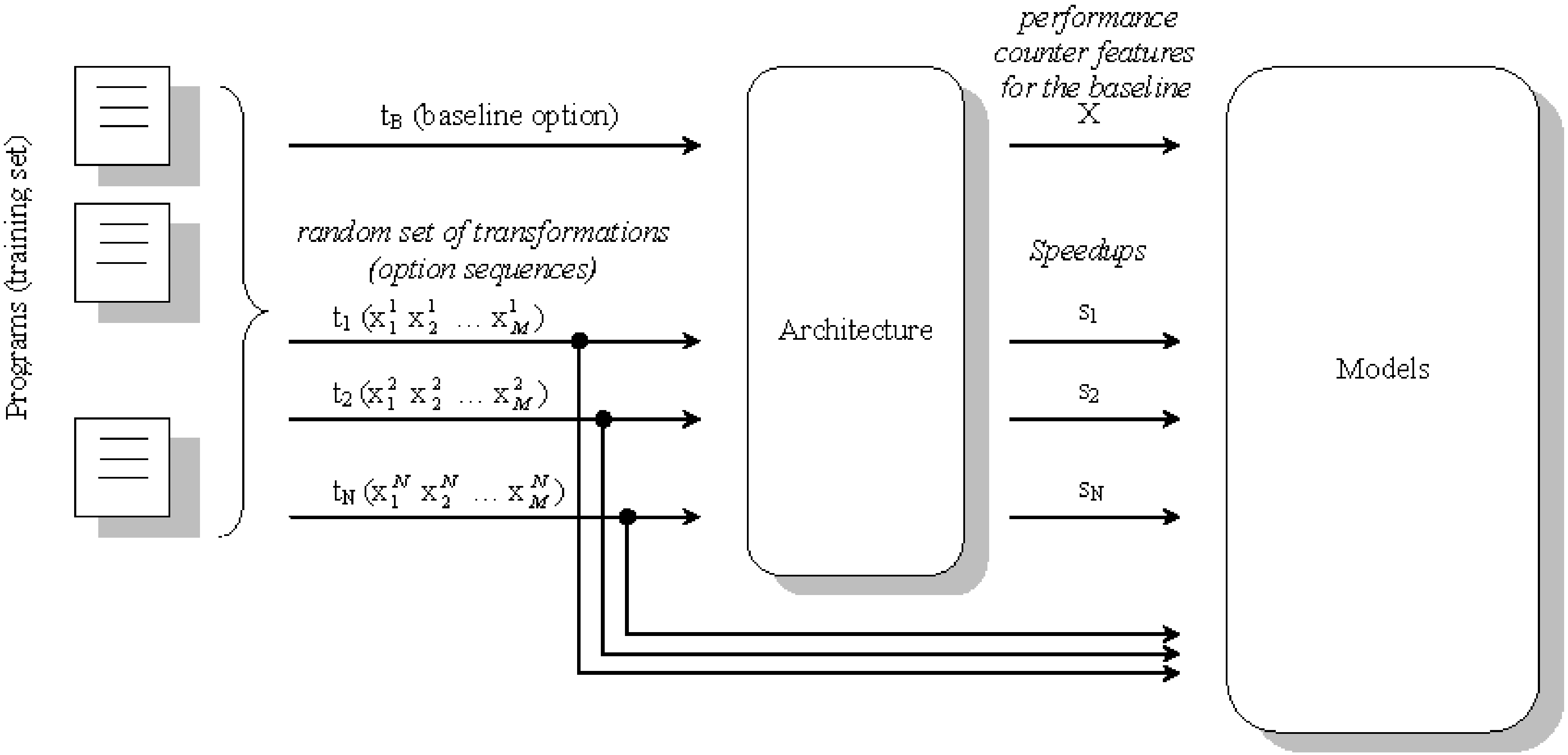
Performance counters provide a compact summary of a program's dynamic behavior

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

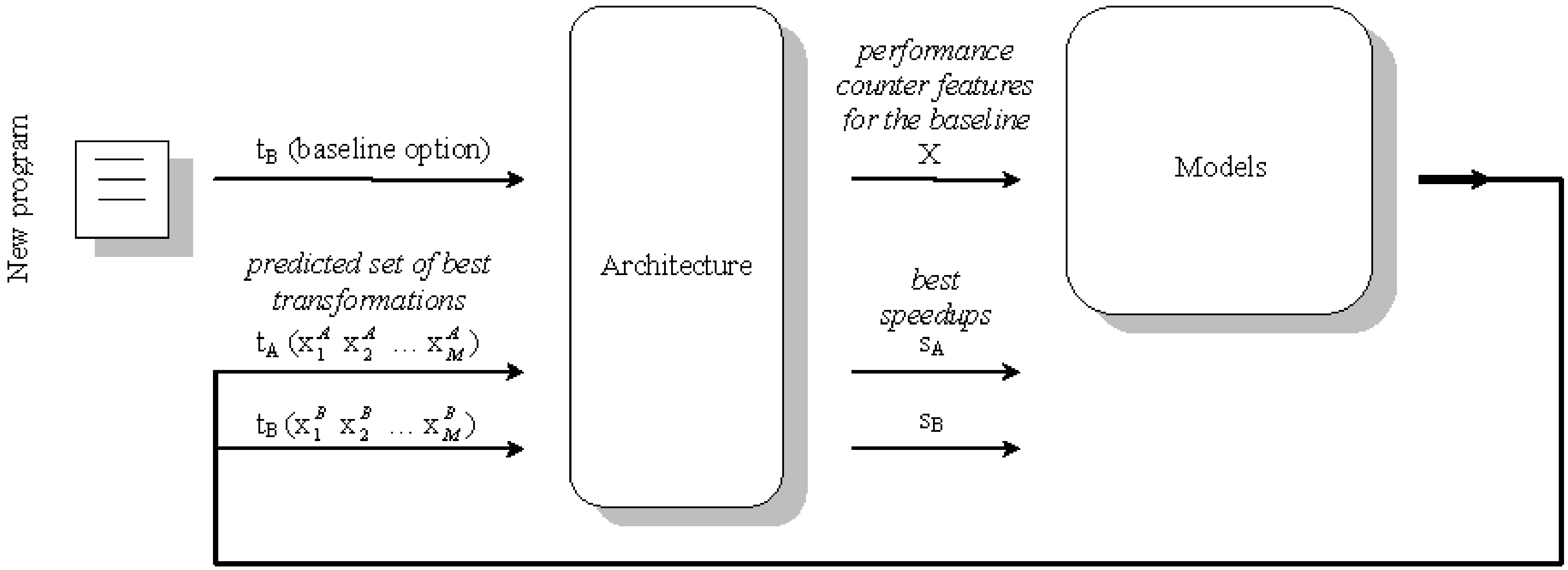
Predictive modeling using logistic regression



(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* → *t* by fitting a probabilistic model to the training set.

General optimizations

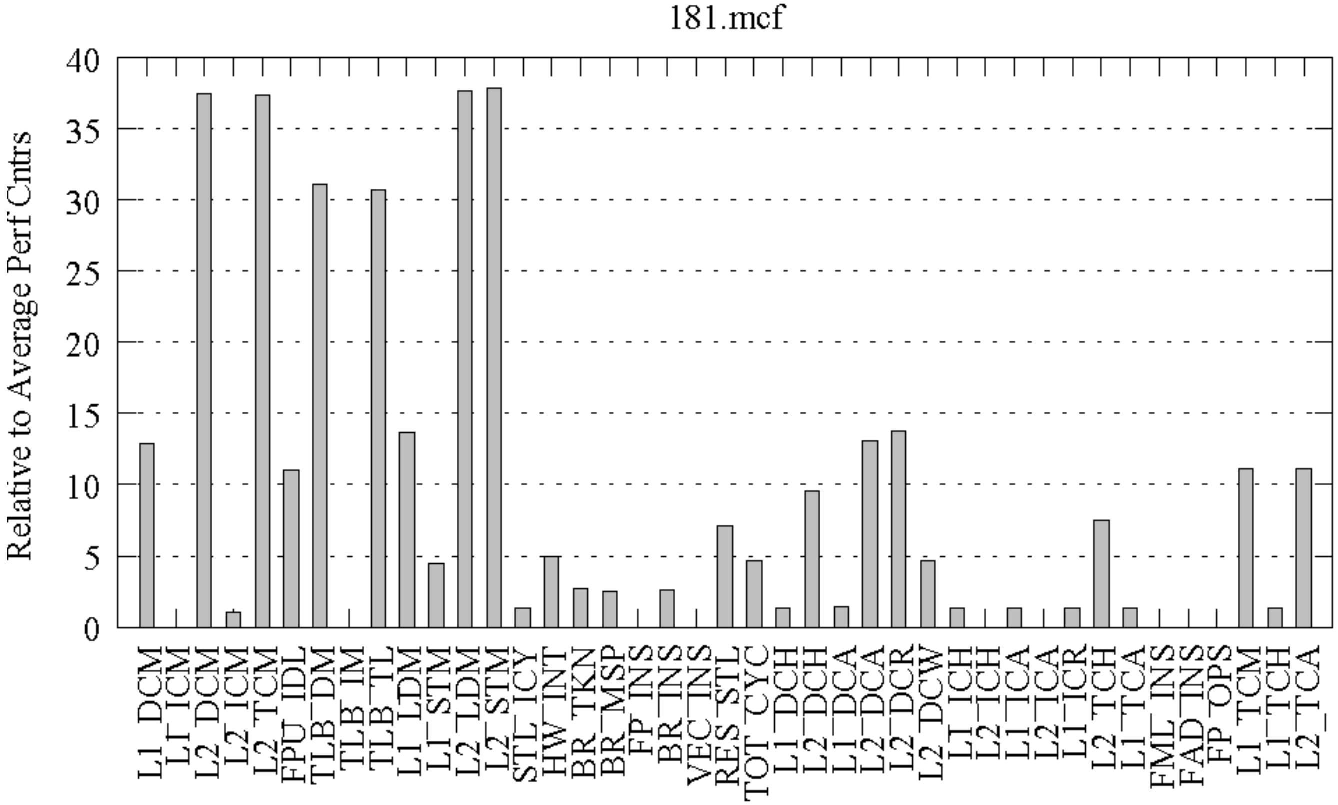
Using models


New program
t_B (baseline option)
predicted set of best transformations
t_A $(x_1^A\ x_2^A\ \dots\ x_M^A)$
t_B $(x_1^B\ x_2^B\ \dots\ x_M^B)$
Architecture
performance counter features for the baseline
X
best speedups
s_A
s_B
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.

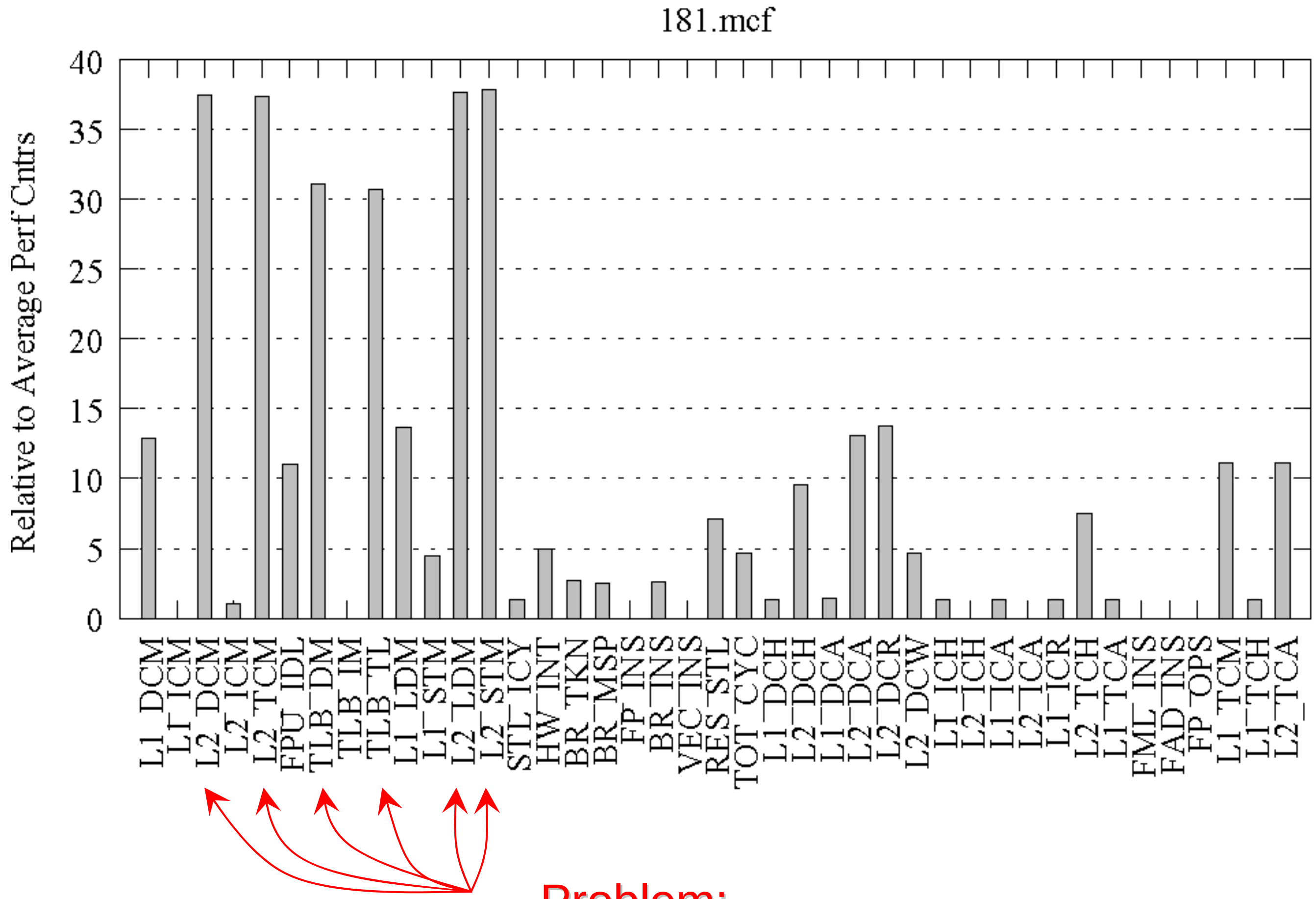
Dynamic characterization of programs

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)

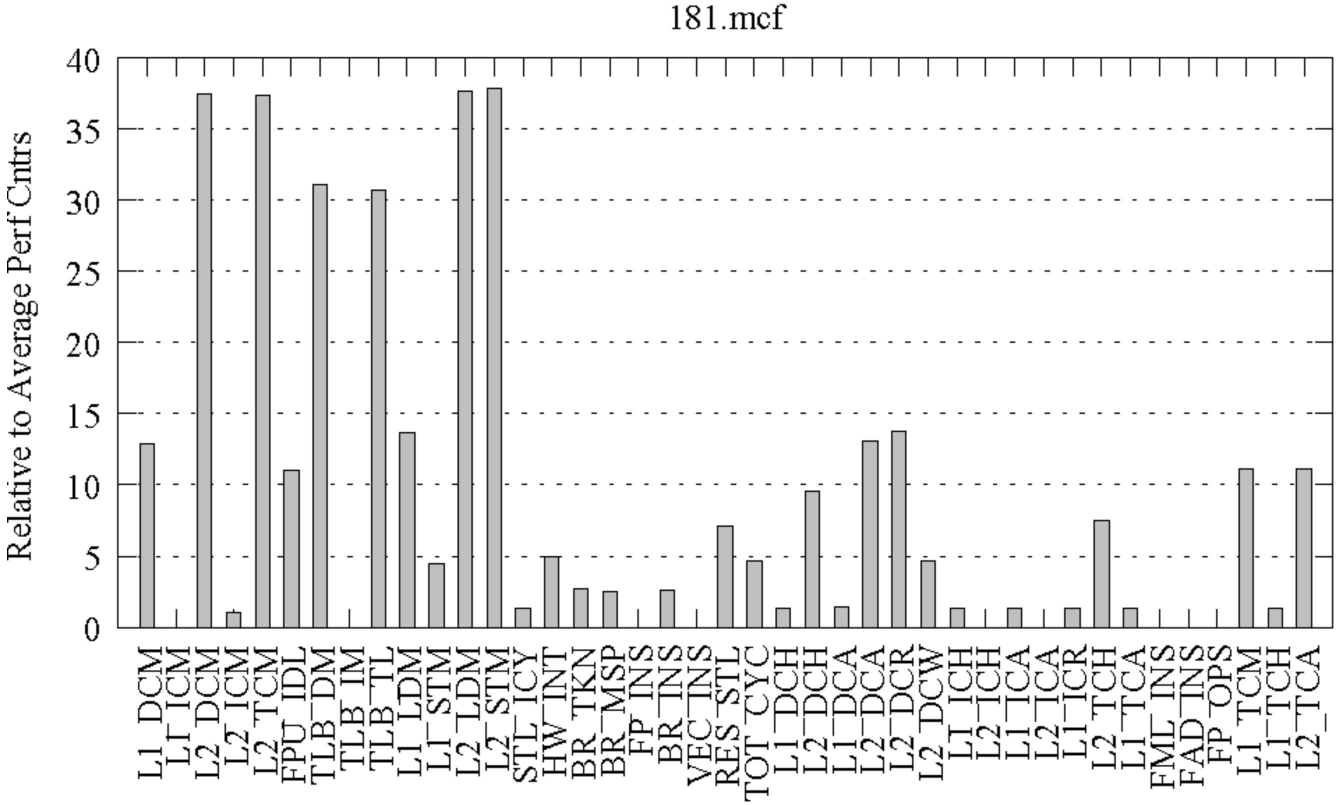


Problem:

greater number of memory accesses per instruction than average

Dynamic characterization of programs

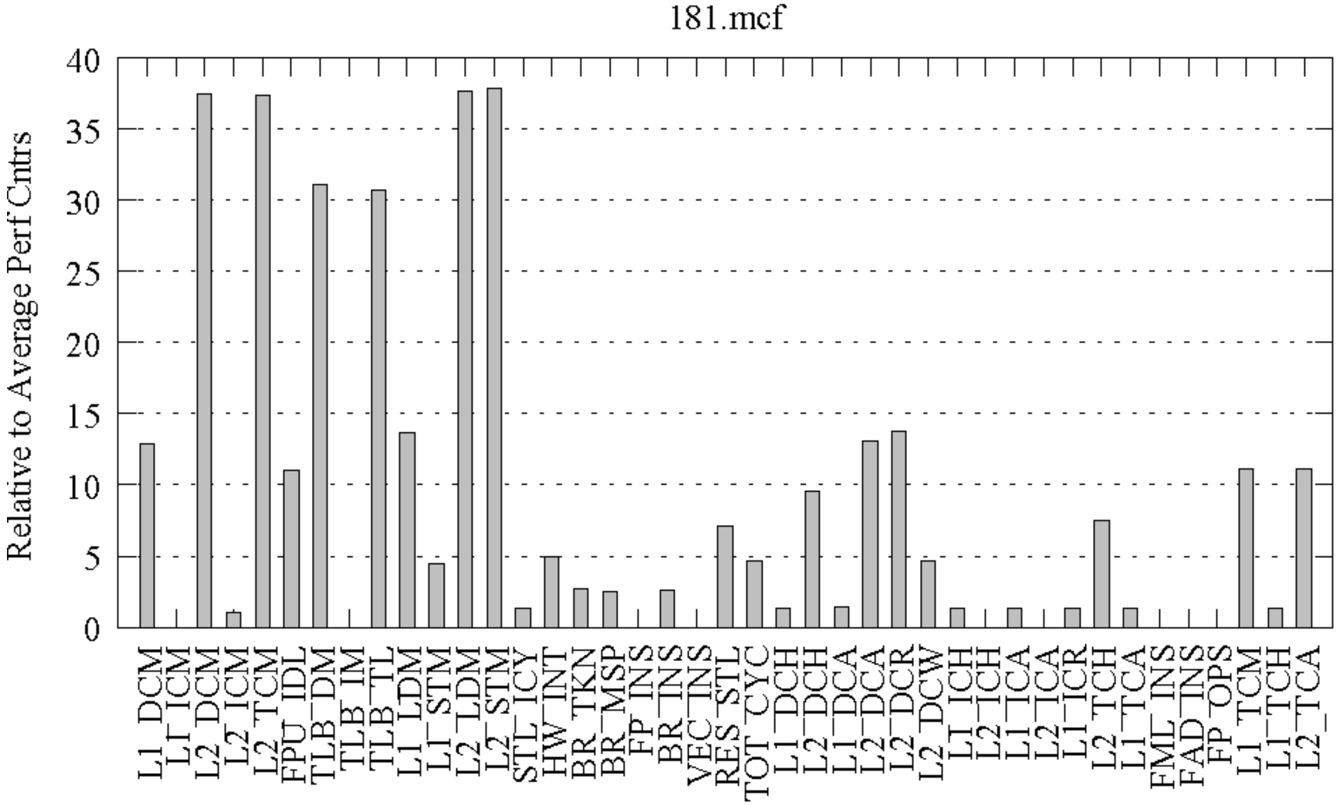
Performance counter values for 181.mcf compiled with -O0 relative to the average values for the entire set of benchmark suite (SPECFP,SPECINT, MiBench, Polyhedron)



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

Dynamic characterization of programs

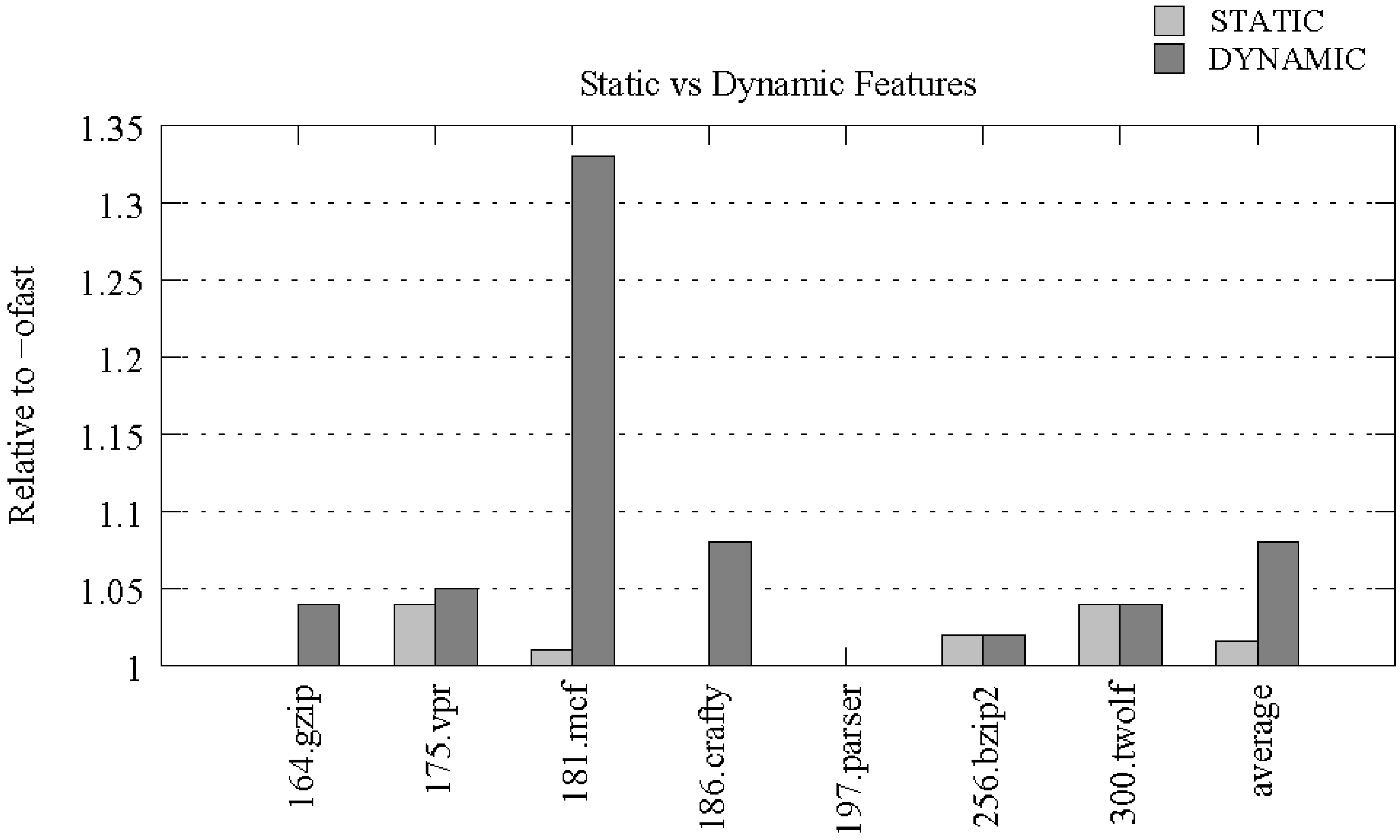
Performance counter values for 181.mcf compiled with -O0 relative to the average values for the entire set of benchmark suite (SPECFP,SPECINT, MiBench, Polyhedron)



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 !

Experimental Results



Performance of SPEC INT 2000 Benchmarks using static code features and dynamic features

Machine learning for DSE

Speeding up Architecture Design Space Exploration

Problems:

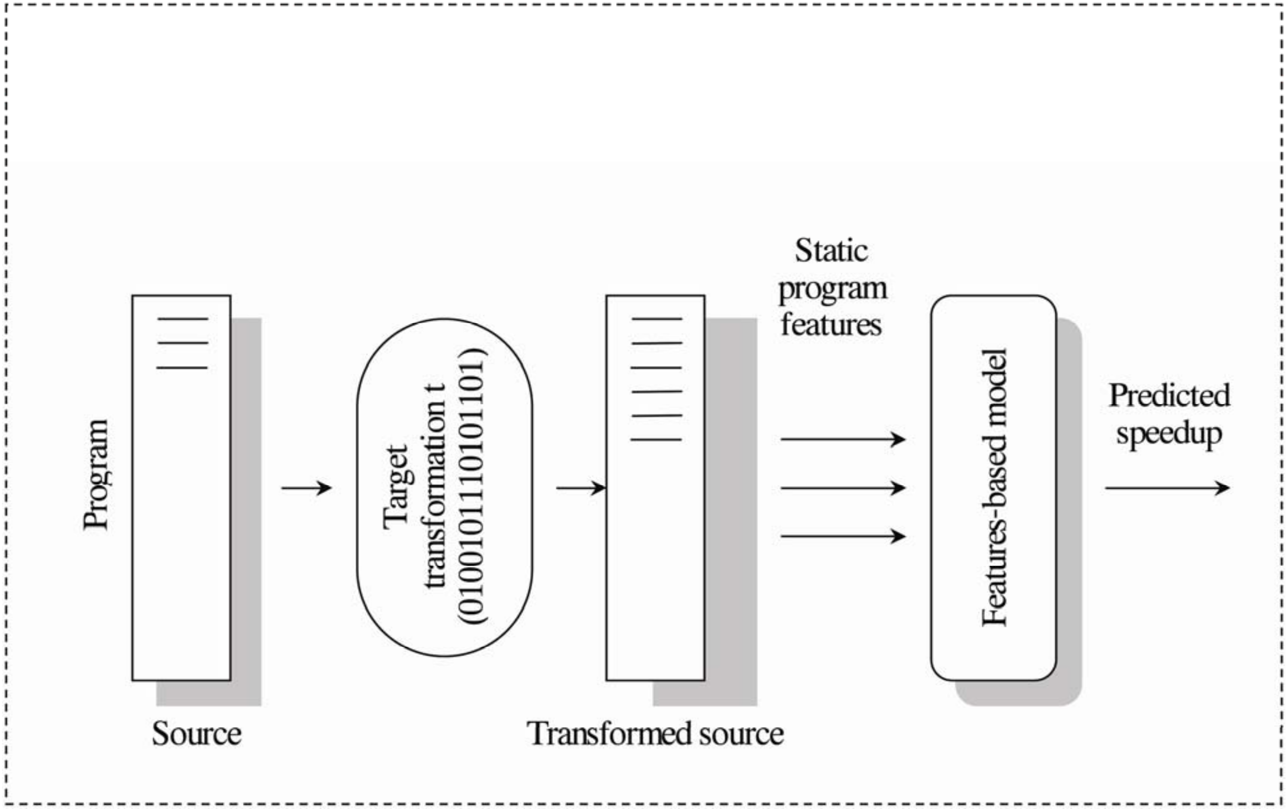
- Developing an optimizing compiler for new architecture is difficult particularly when only simulator is available
- Tuning such compiler requires many runs
- Simulators are orders of magnitude slower than real processors
- Therefore compiler tuning is highly restricted

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

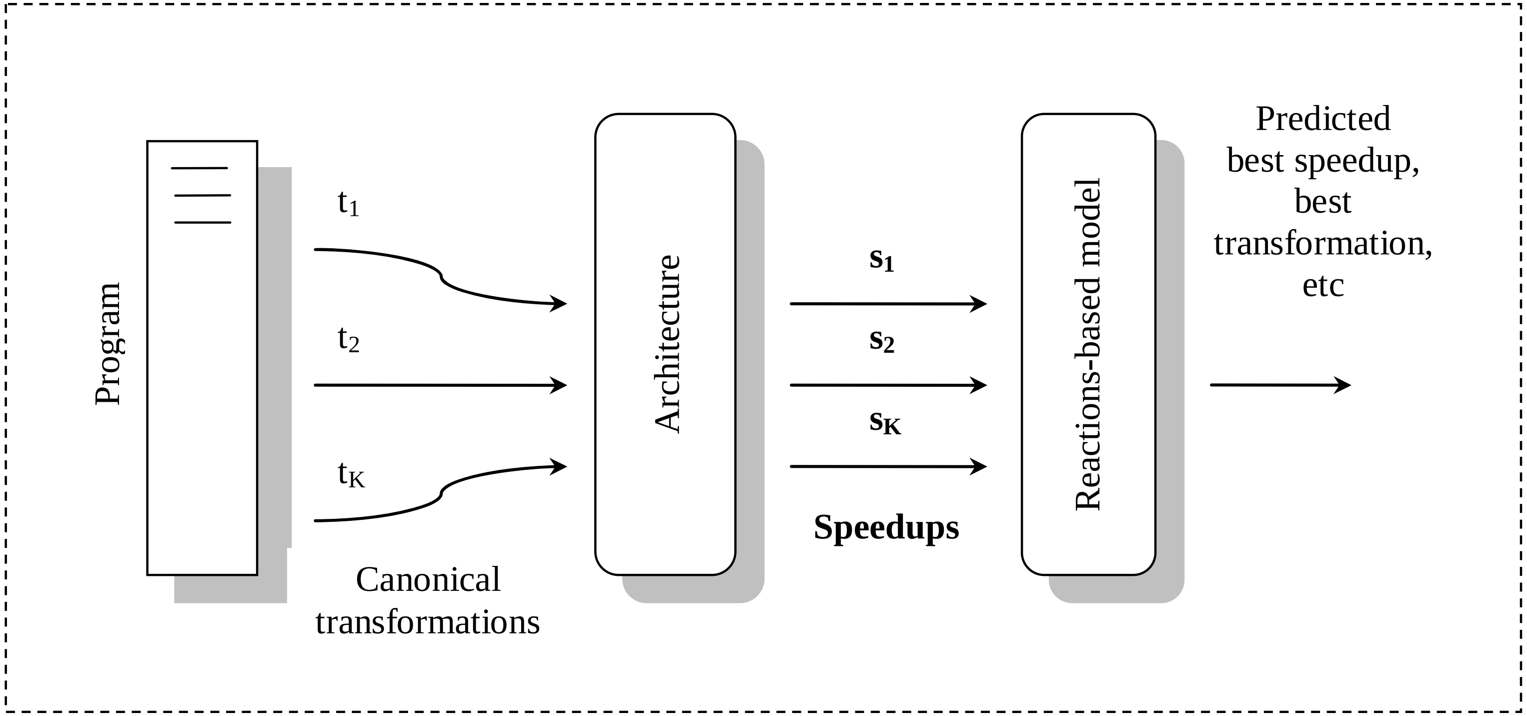


Features-based model

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

Output: *program speedup*

Machine learning for DSE

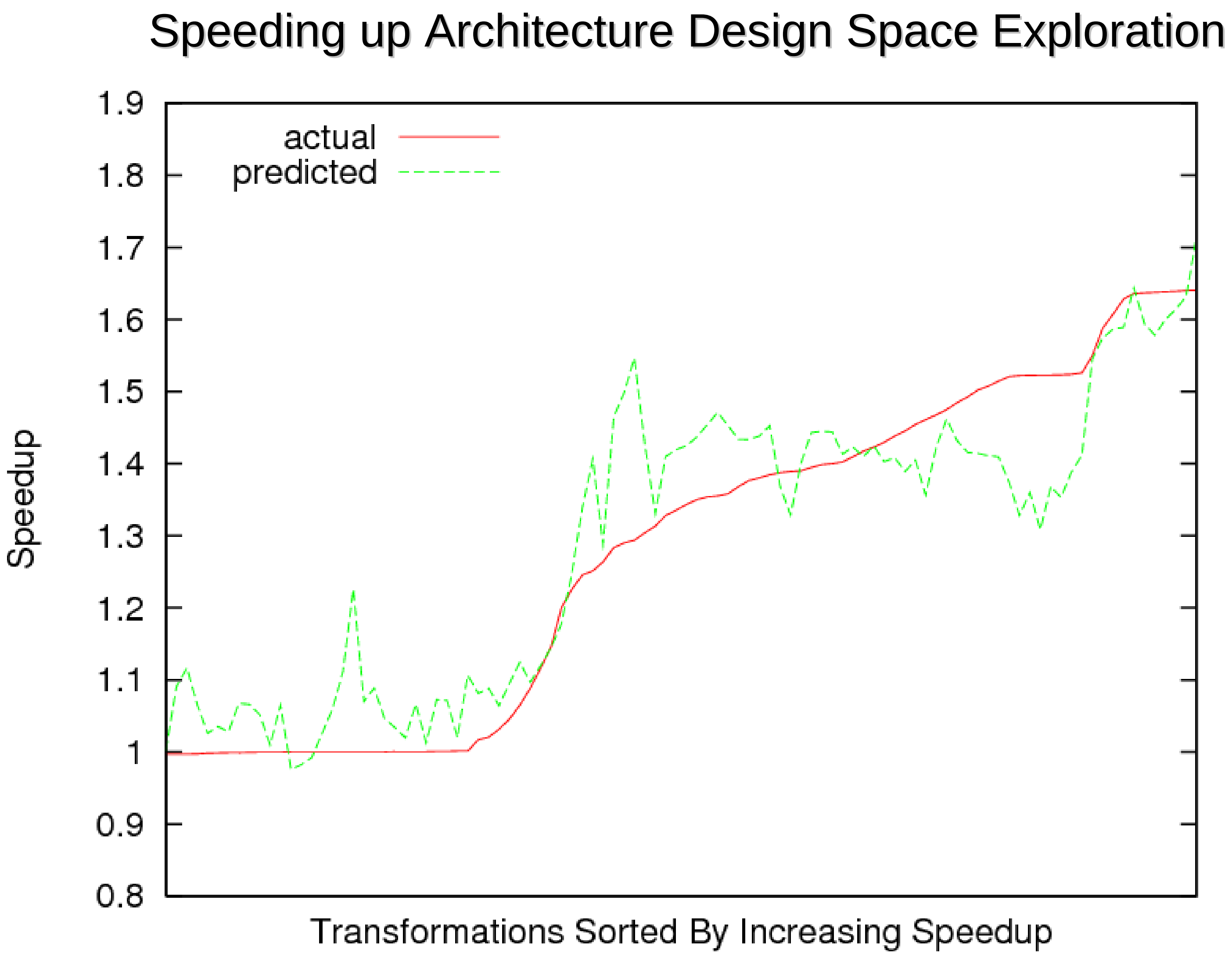


Reactions-based model

Input: *speedups on canonical transformation sequences*

Output: *transformation sequence speedup*

Machine learning for DSE



Reliable performance model 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

- 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

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

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