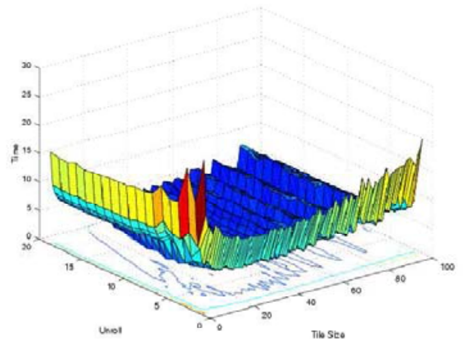
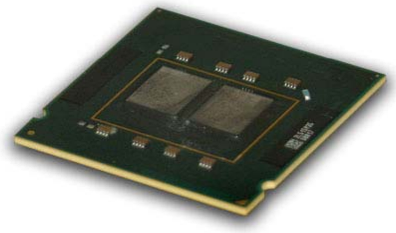


Iterative feedback-directed compilation

Grigori Fursin

Alchemy group, INRIA Saclay, France



My background

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

  Program iterative optimizations and performance prediction

- *Postdoctoral researcher at INRIA Futurs, France (2004 - 2007)*
- *Research scientist at INRIA Saclay, France (2007 …)*

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

- *Main collaborations:*

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

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?

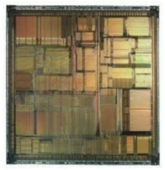
Motivation

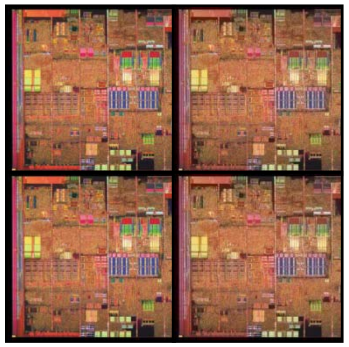
Current innovations in science and industry demand ever-increasing computing resources while placing strict requirements on ***system performance, power consumption, size, response, reliability, portability and design time***.

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



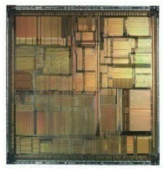


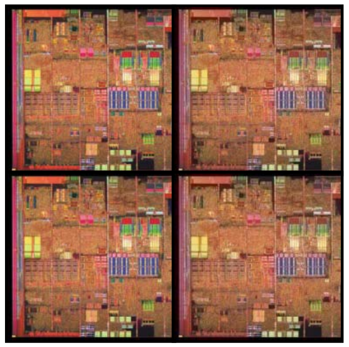
dramatically increased optimization time

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 exit - 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

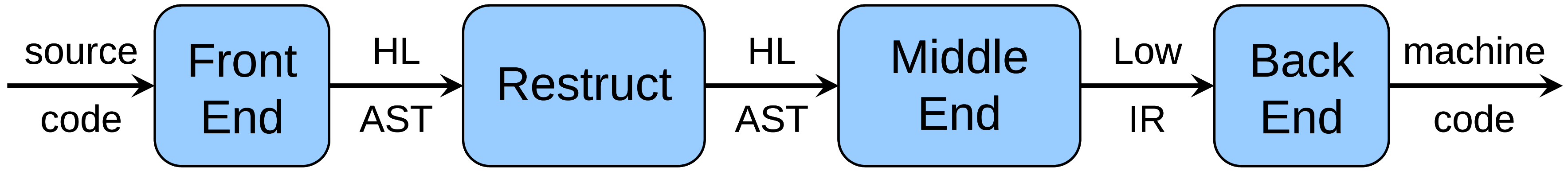
- PathScale Compilers

  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)

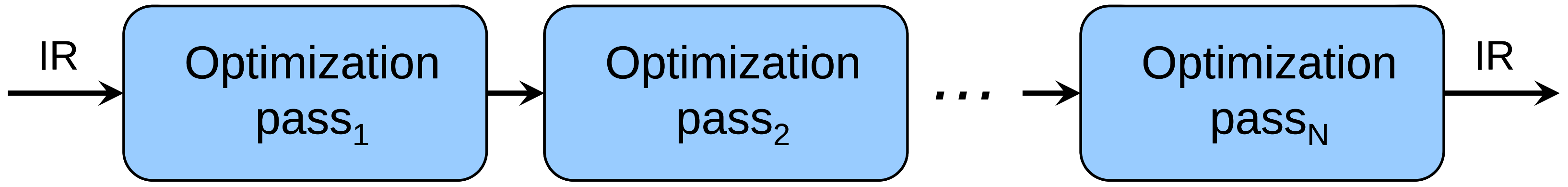
- Restructurer and Middle End performs machine independent optimizations including “source-to-source transformations” and outputs a Lower Level Intermediate Representation (IR)

    - 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

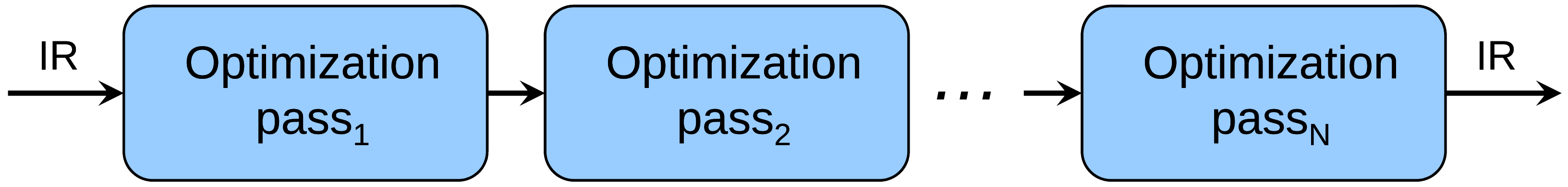
Optimization passes can be often turned on and off using compiler command line flags

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

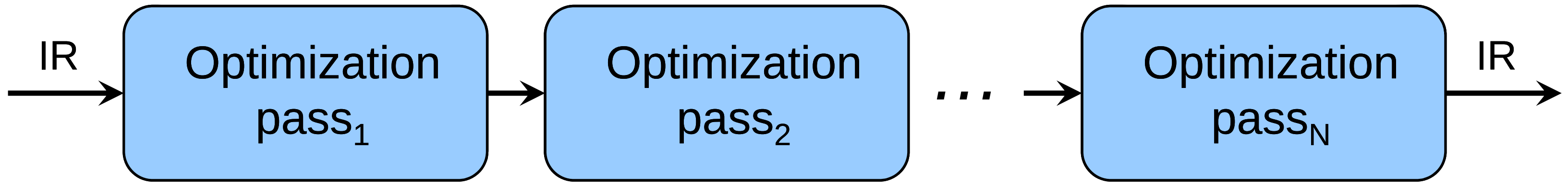
Optimizer structure



Is this working well?

(DEMO$_1$)

Optimizer structure

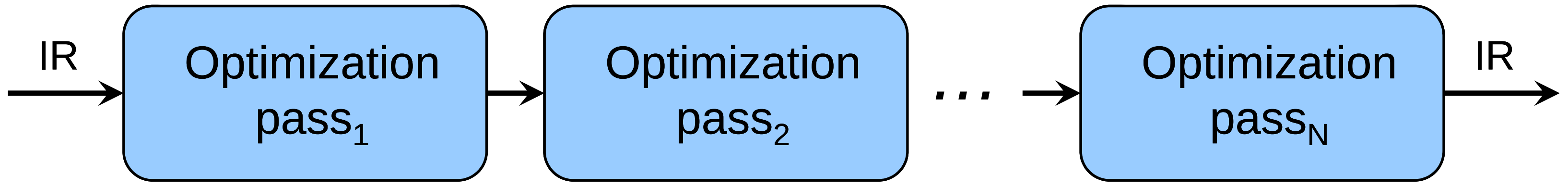


Matmul benchmark and GCC 4.2.x compiler:

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



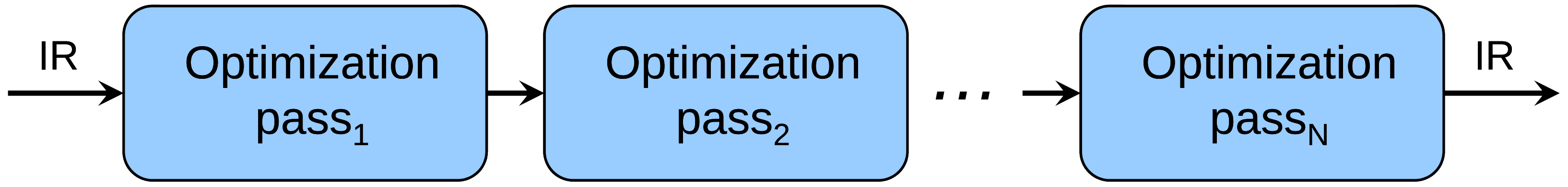
Matmul benchmark and GCC 4.2.x compiler:

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!

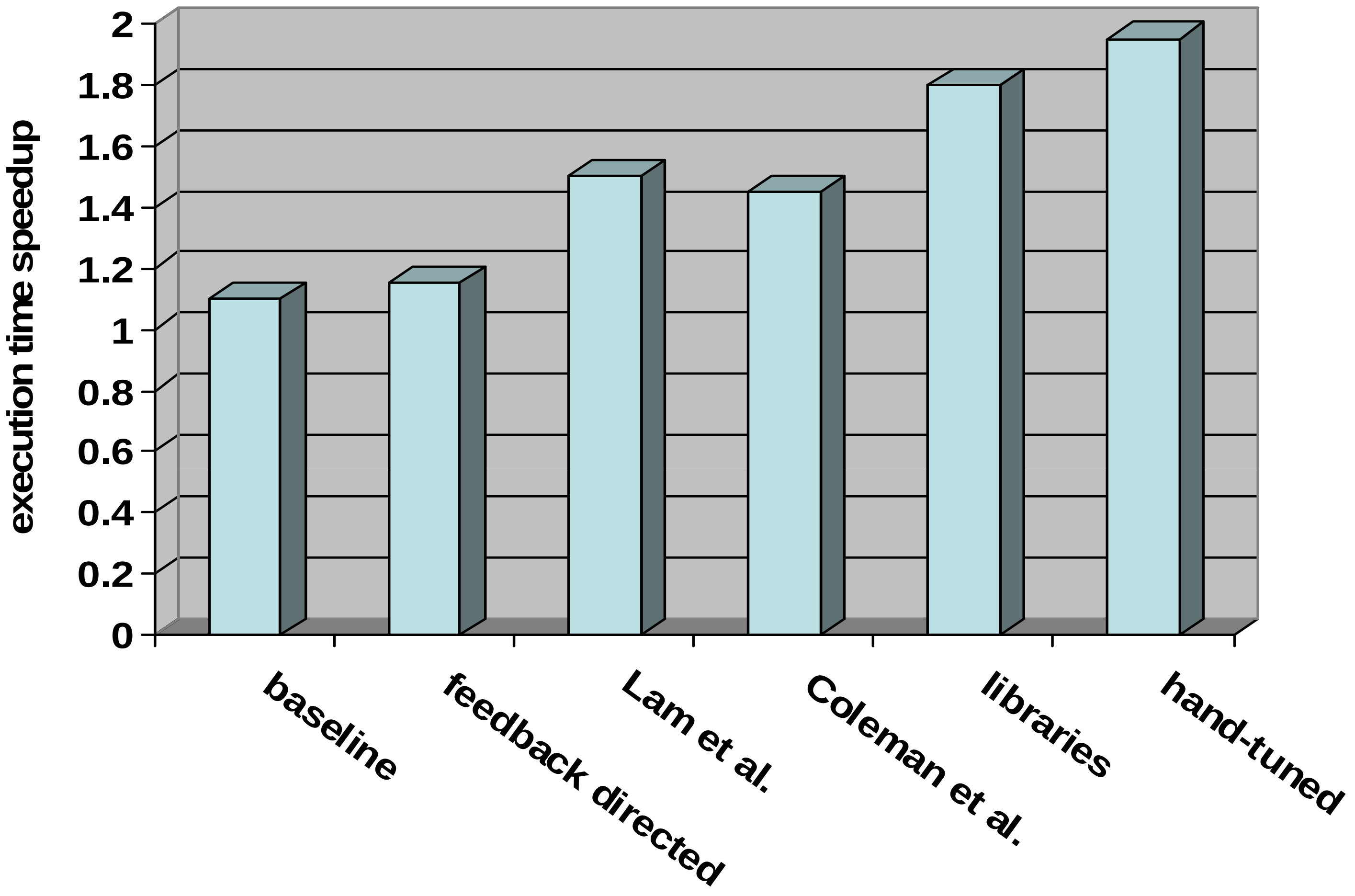
2) gcc -O3 -funroll-loops matmul.c [matrix size 3x3]

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];
```

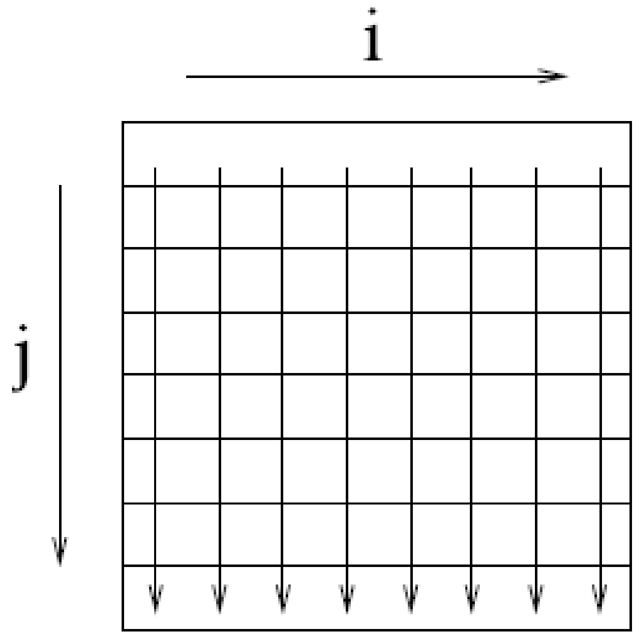
This code traverses the array column-wise

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

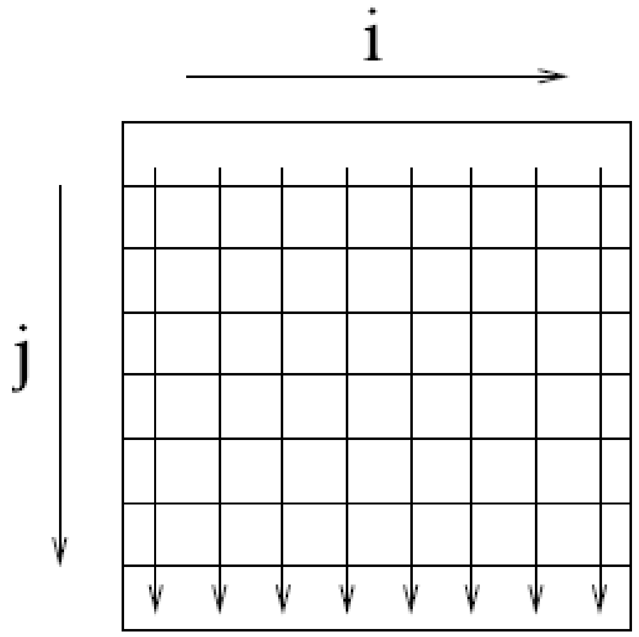
```
for (j = 0; j<n; j++)
for (i = 0; i<n; i++)
    a[j][i] + b[i];
```

- 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)         ⎬ loop body replicated
   S2(i+1)         ⎪ u times
   …               ⎪
   S1(i+u-1)       ⎪
   S2(i+u-1)       ⎪
   …               ⎪
end do             ⎭
do j = i, n        ⎫
   S1(j)           ⎪ processing all
   S2(j)           ⎬ remaining
   …               ⎪ elements
end do             ⎭
```

Which unrolling factor to choose?

Challenges

Some other transformations: Loop Tiling

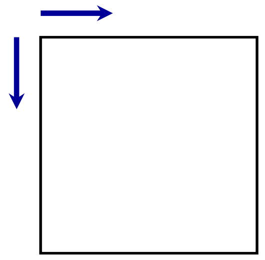
original loop nest:

```
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
```

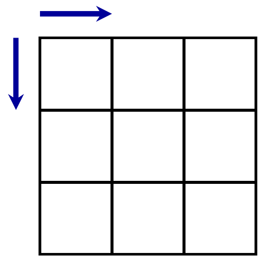
transformed loop nest:

```
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 as 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.

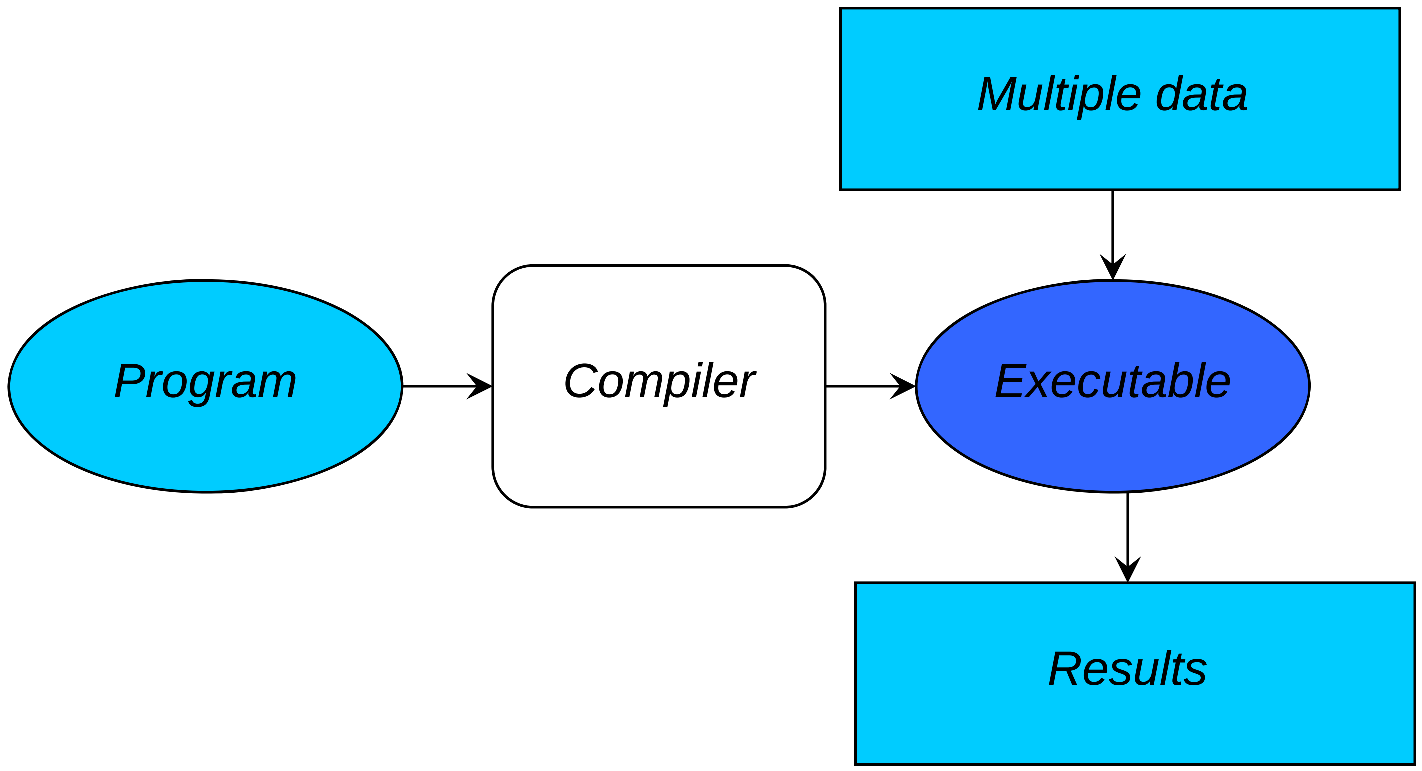
• Allows specialization to run-time data – what are pros and cons?

Feedback directed compilation

Off-line vs on-line compilation

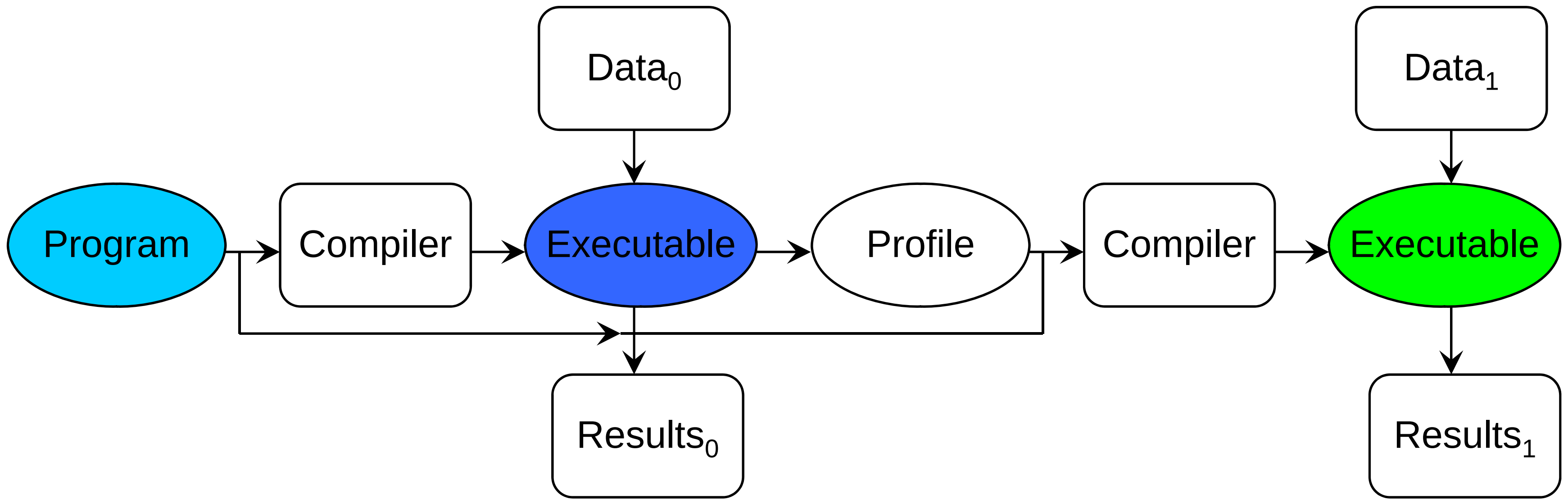
- Profile directed compilation is one example of off-line optimization
- Information is gathered and utilized before the “production” run
- On-line schemes gather information and dynamically change program as it runs.
- Off-line schemes work on basis that costs incurred at compile-time are outweighed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

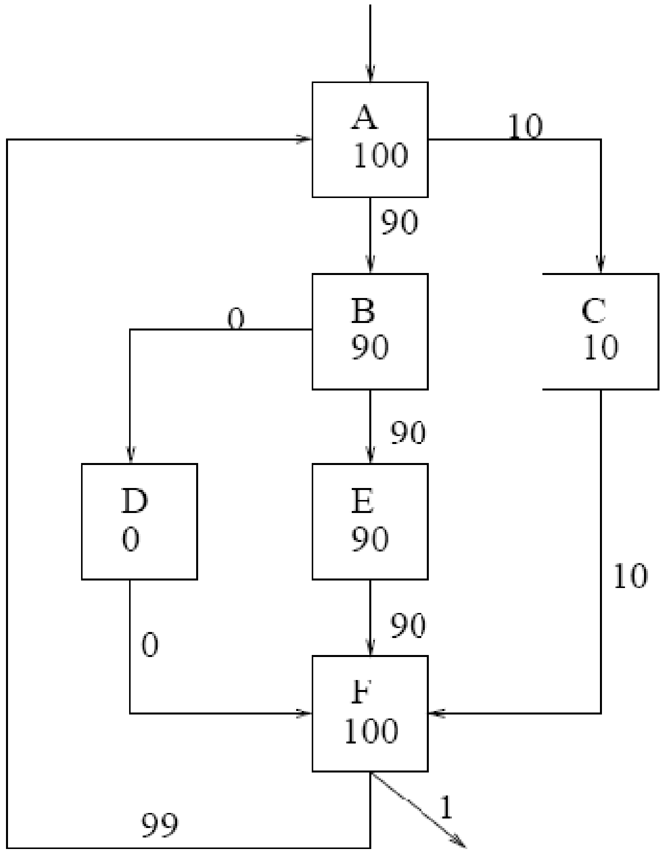
PDC for classic optimization

- 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

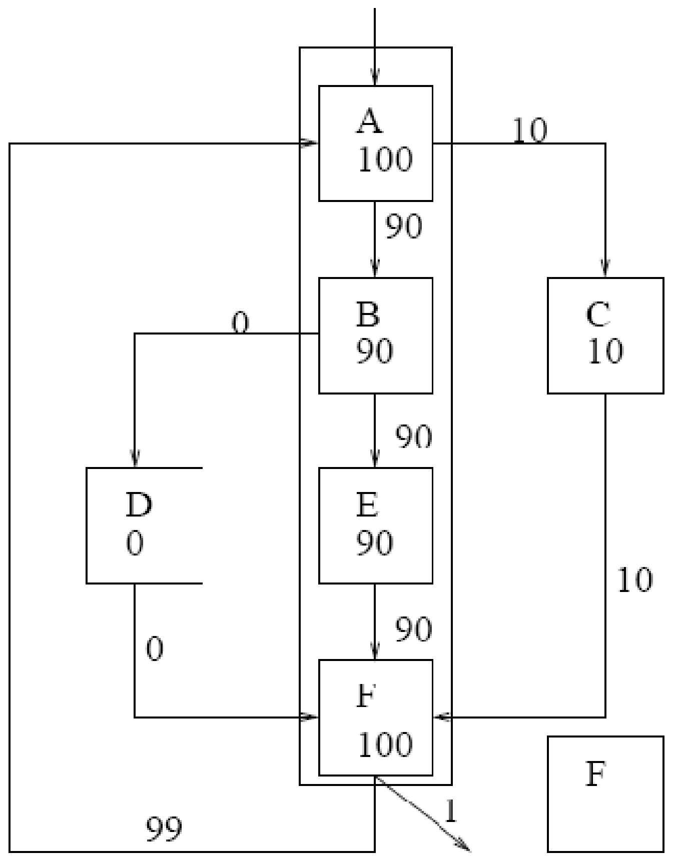
- Sequence of basic blocks
- Frequency of execution on edges and nodes
- Primarily ABEF
- Other entry/exit control-flow prevents merging
- Super-block -frequently executed path
- Merge and tidy-up
- Optimize larger unit



Feedback directed compilation

PDC example 1

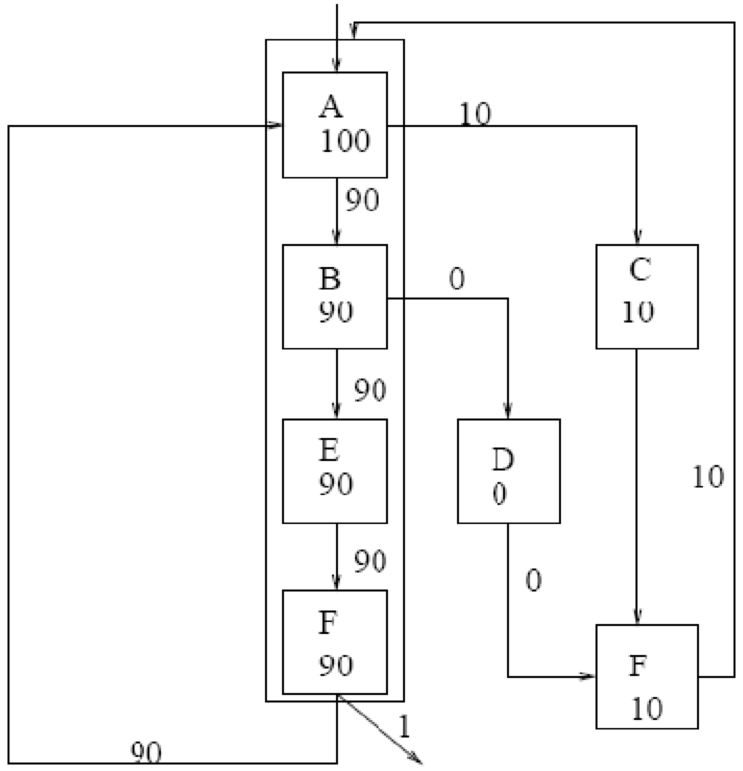
- Selecting the trace
- Start at most frequent block
- Add blocks on most frequent successors
- Repeat on other nodes
- Done in both control-flow directions
- Do on remaining nodes



Feedback directed compilation

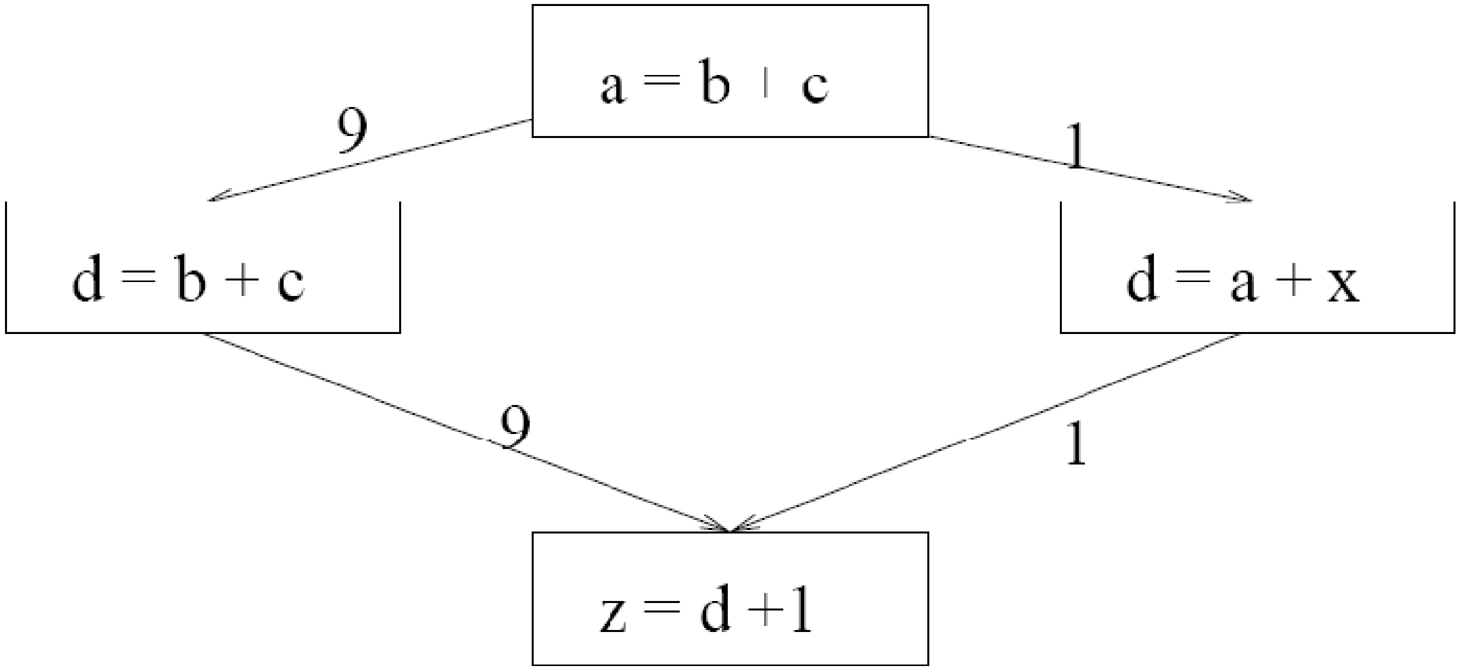
PDC example 1

- 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

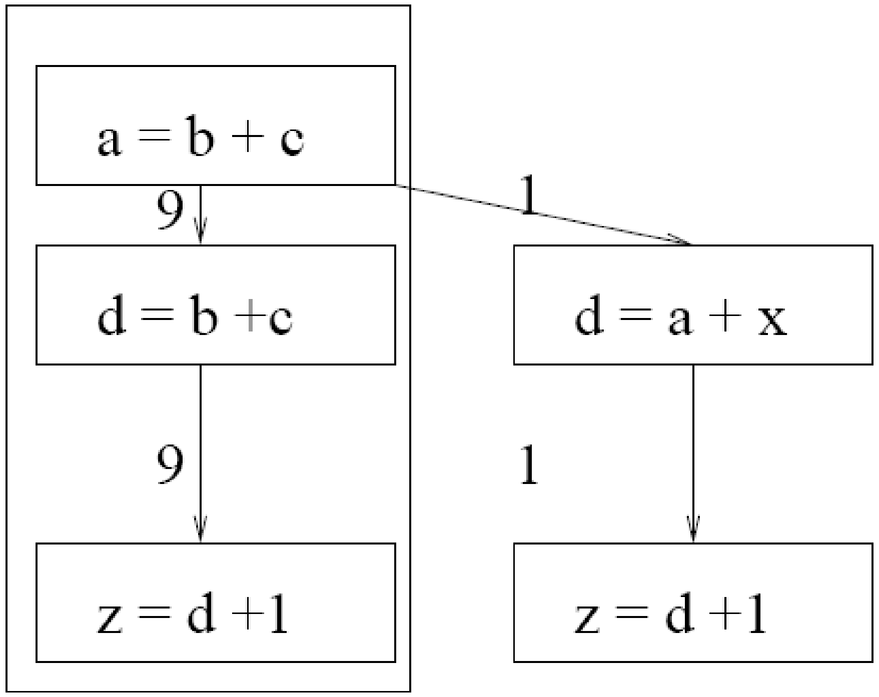
PDC example 2



Common b + c on frequently taken path

Feedback directed compilation

PDC example 2

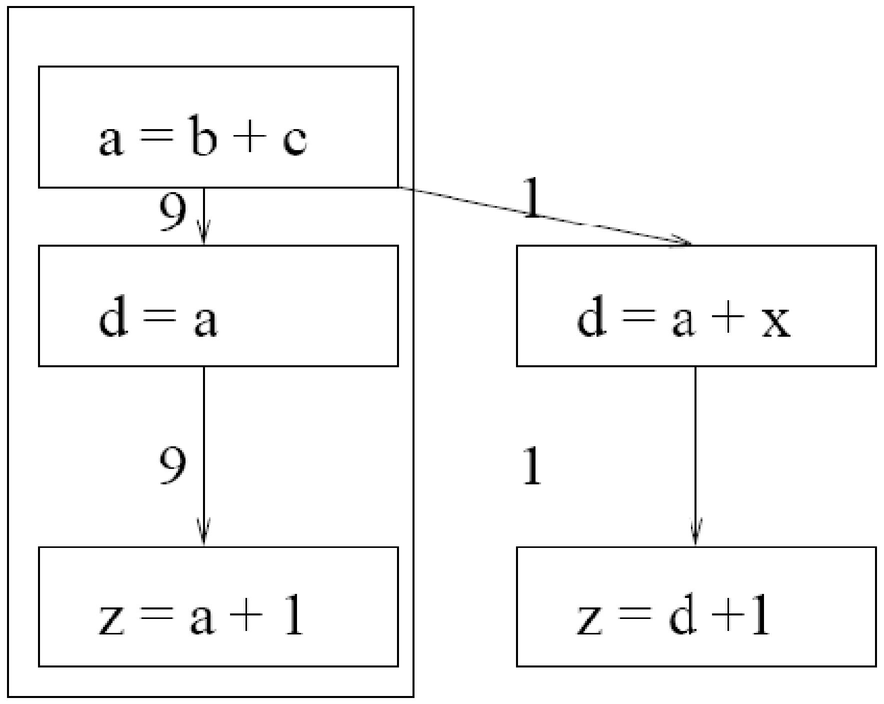


Replicate first node on main path with external incoming edge

Now separate paths

Feedback directed compilation

PDC example 2



Applying CSE eliminates redundant computation at cost of additional code

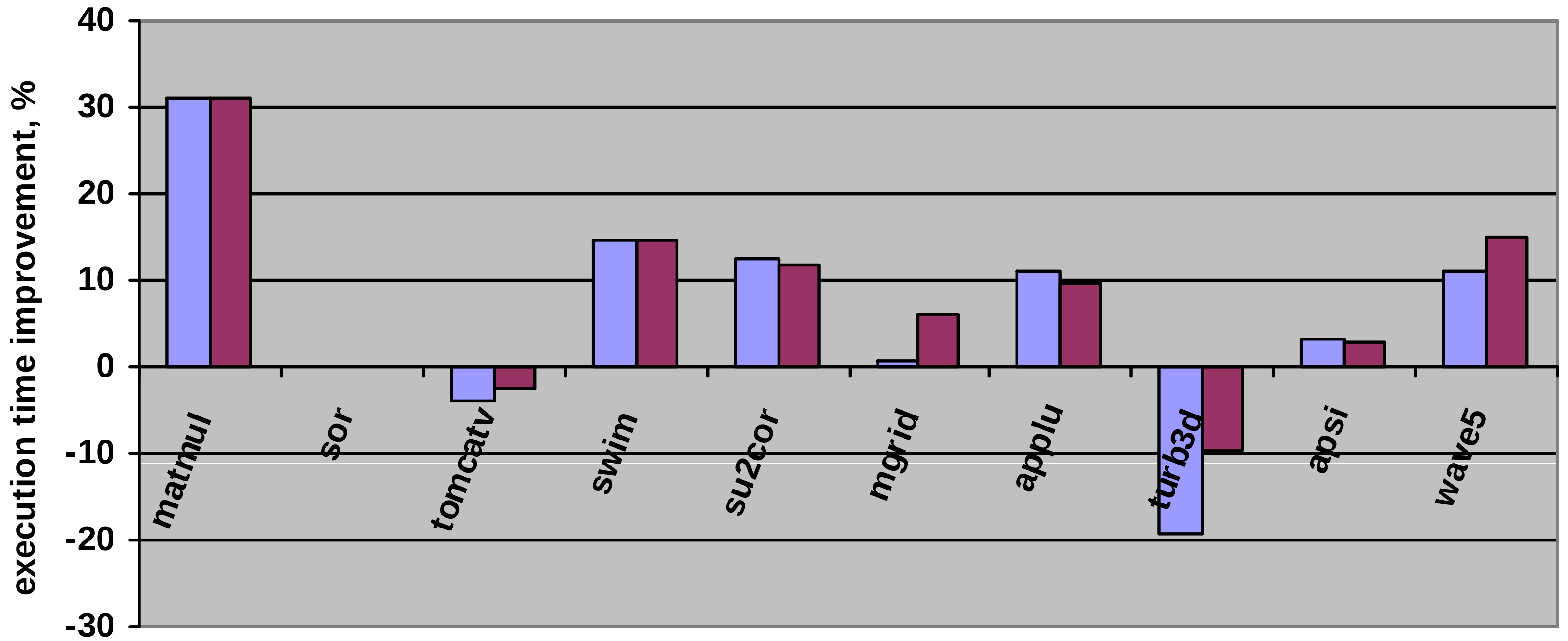
Feedback directed compilation

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)

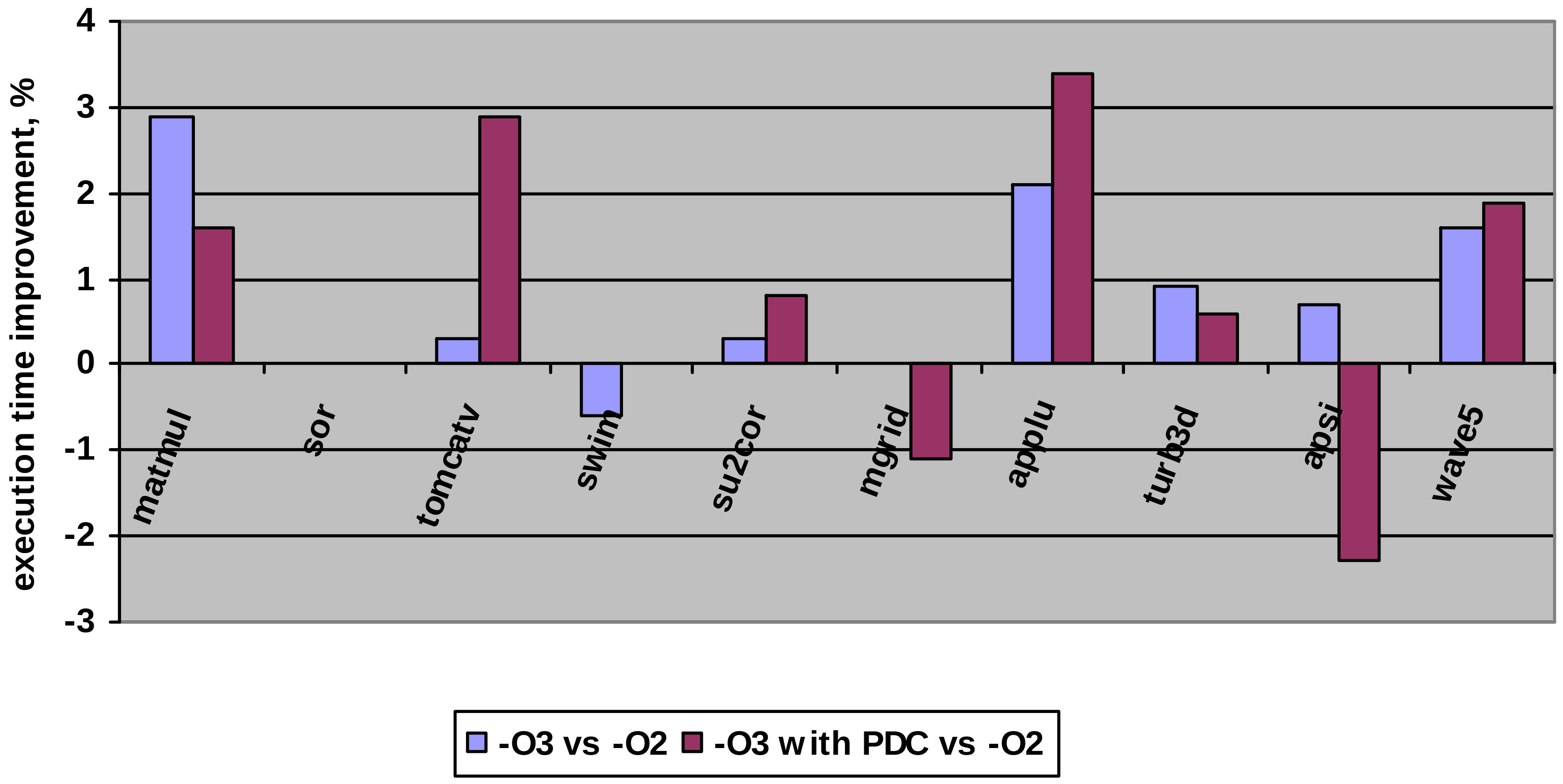


-O3 vs -O2	-O3 with PDC vs -O2

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

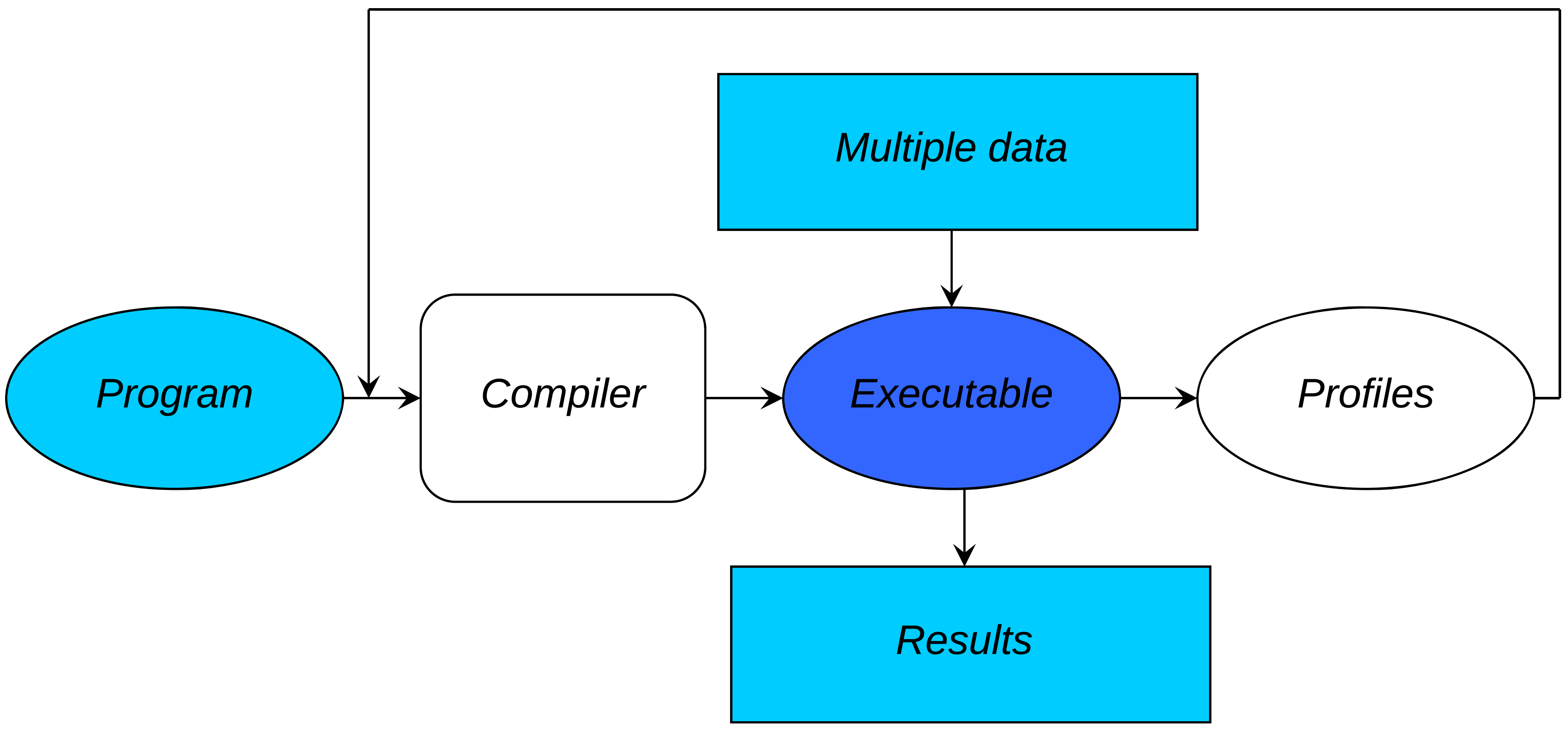
Beyond PDC

- 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

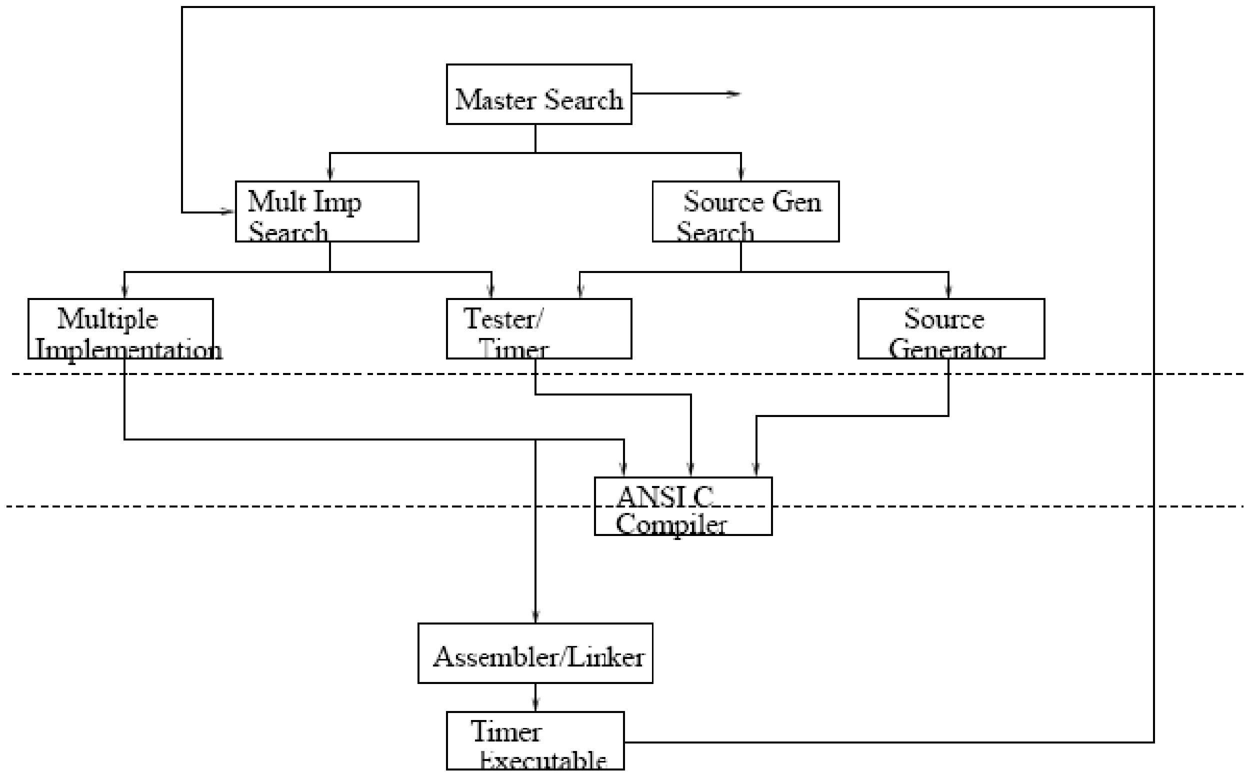
Feedback directed compilation

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

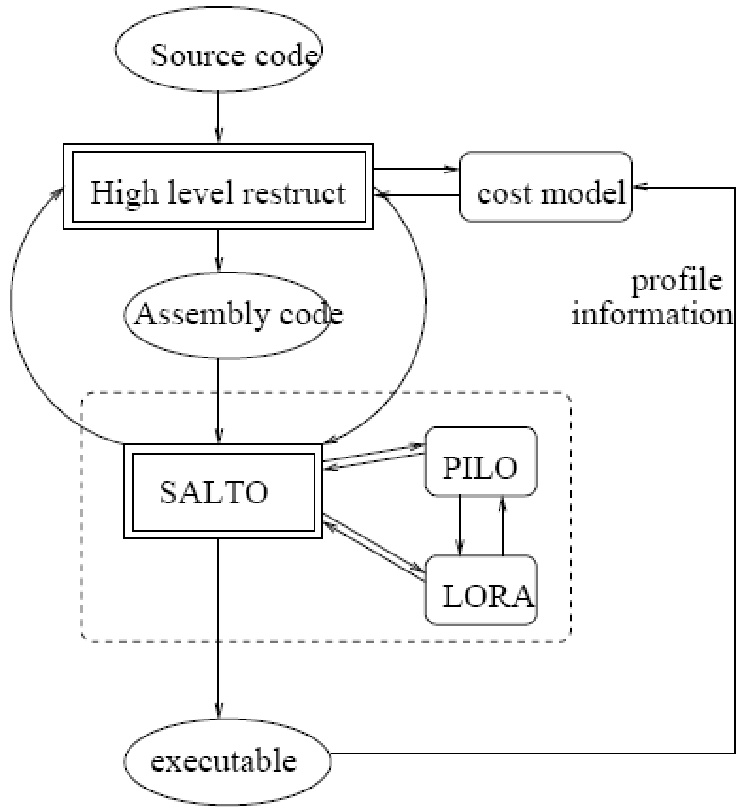
Iterative compilation

- 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

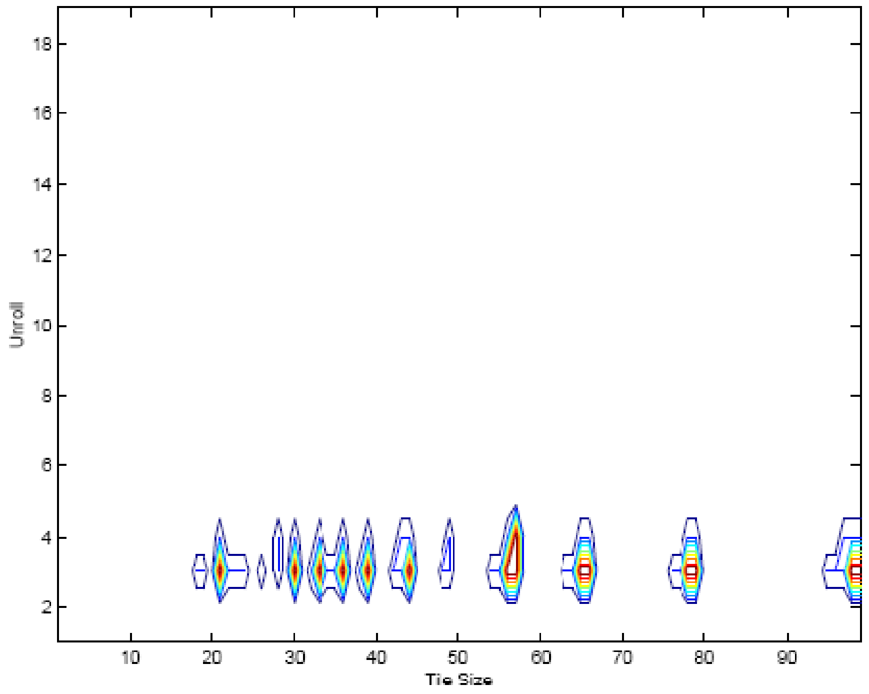
OCEANS

- 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

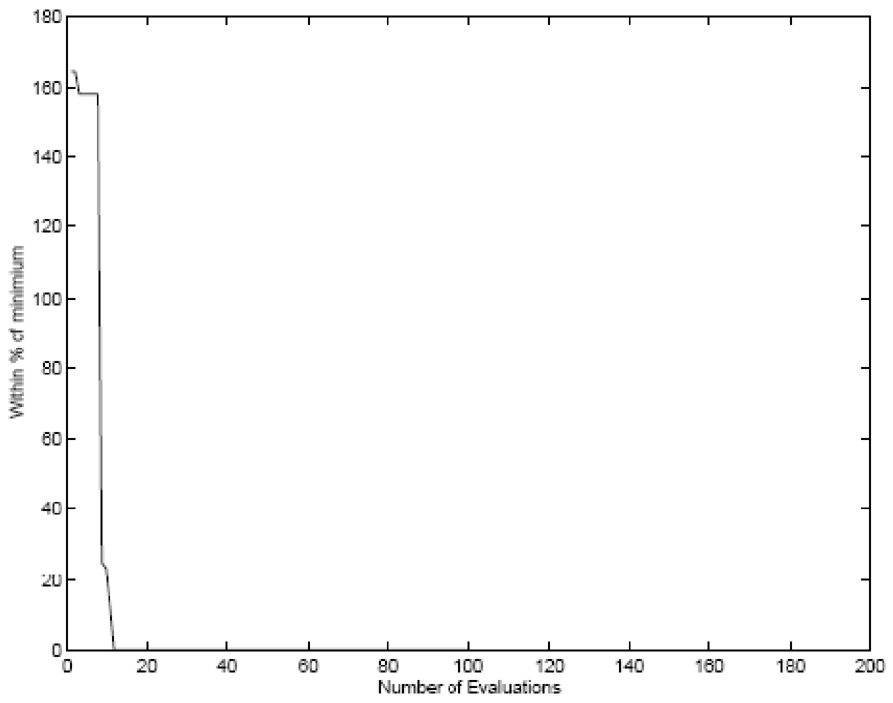


Minimum at: Unroll=3, Tile size=57

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

Feedback directed compilation

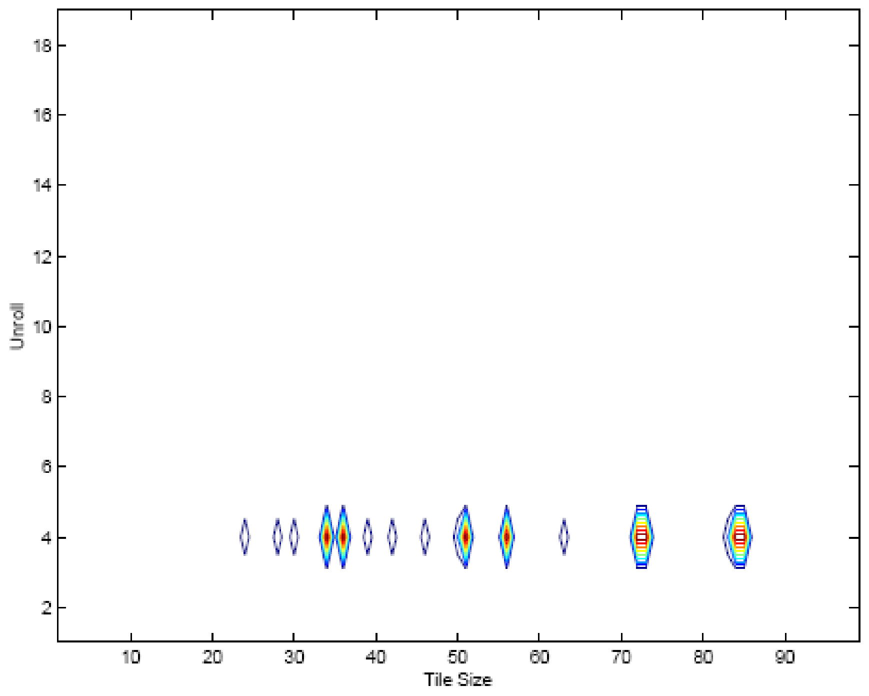
matrix multiply, N=400, UltraSparc, random search



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

Feedback directed compilation

matrix multiply, N=512, Alpha, exhaustive search

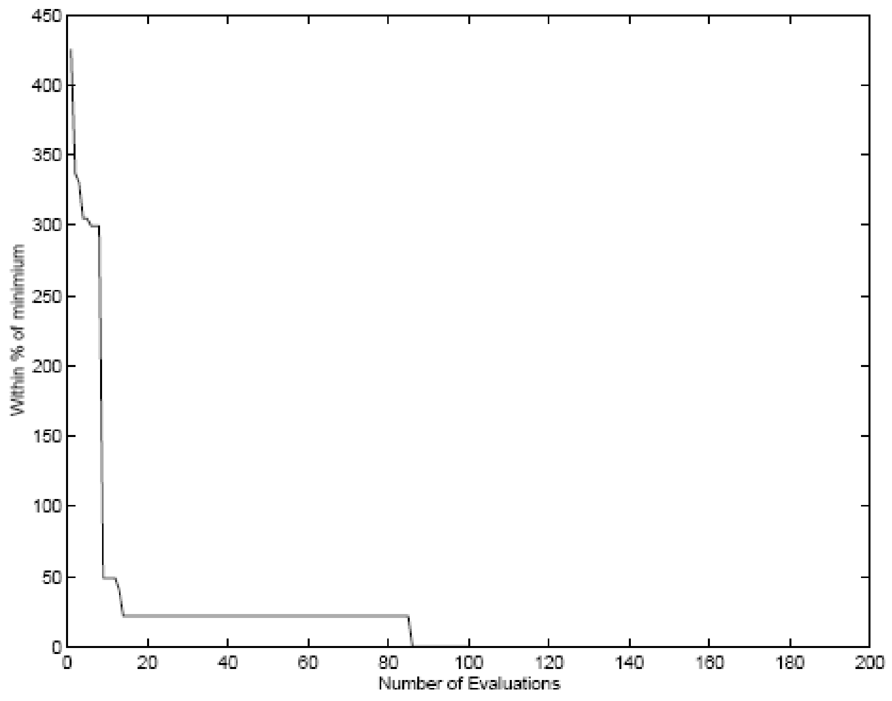


Minimum at: Unroll=4, Tile size=85

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

Feedback directed compilation

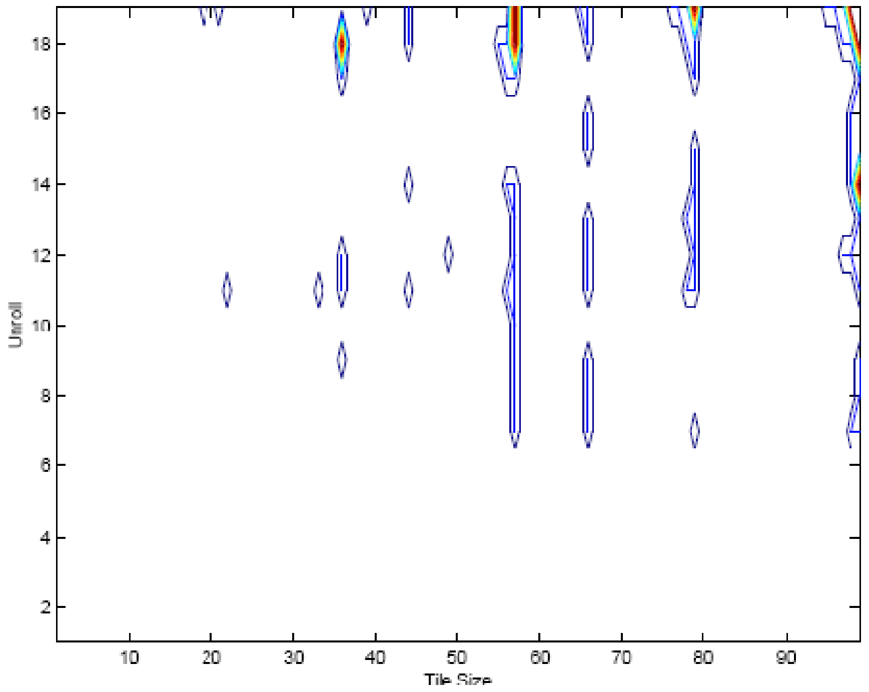
matrix multiply, N=512, Alpha, random search



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

Feedback directed compilation

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

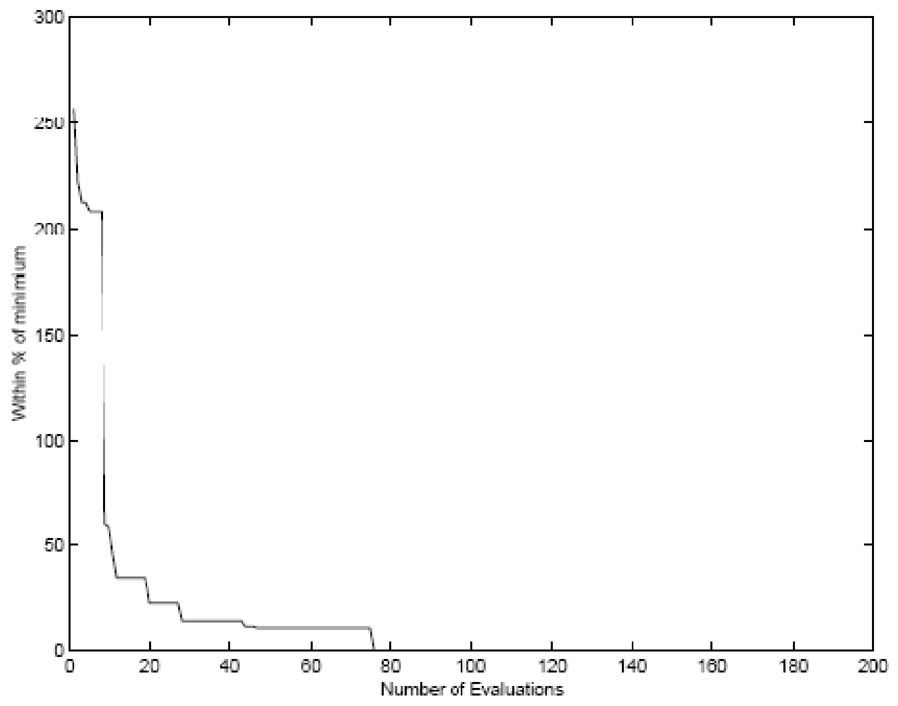


Minimum at: Unroll=19, Tile size=57

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

Feedback directed compilation

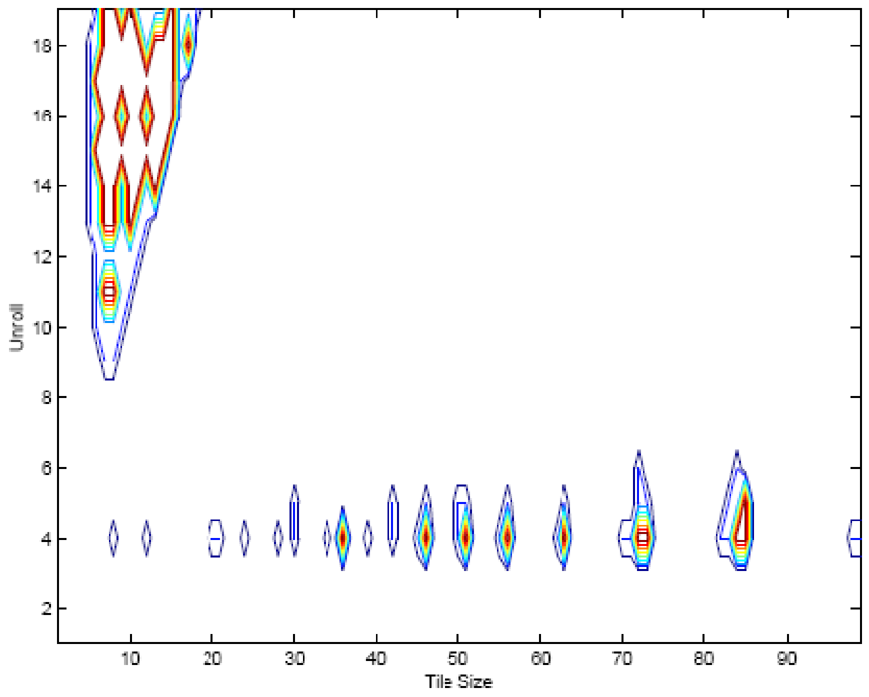
matrix multiply, N=400, Pentium Pro, random search



50 steps: within 10.5%

Feedback directed compilation

matrix multiply, N=512, R10000, exhaustive search

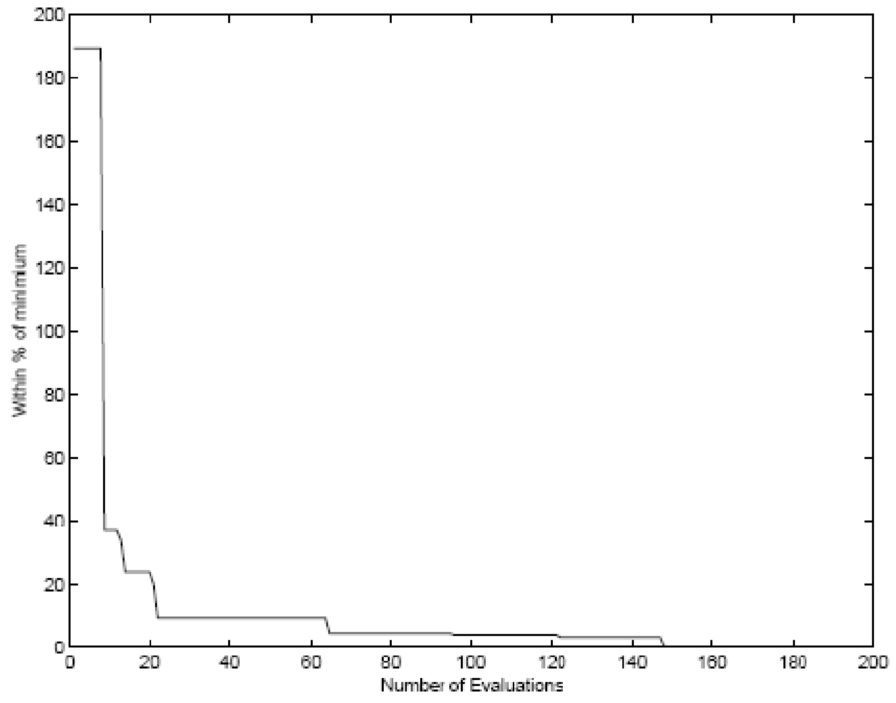


Minimum at: Unroll=4, Tile size=85

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

Feedback directed compilation

matrix multiply, N=512, R10000, random search



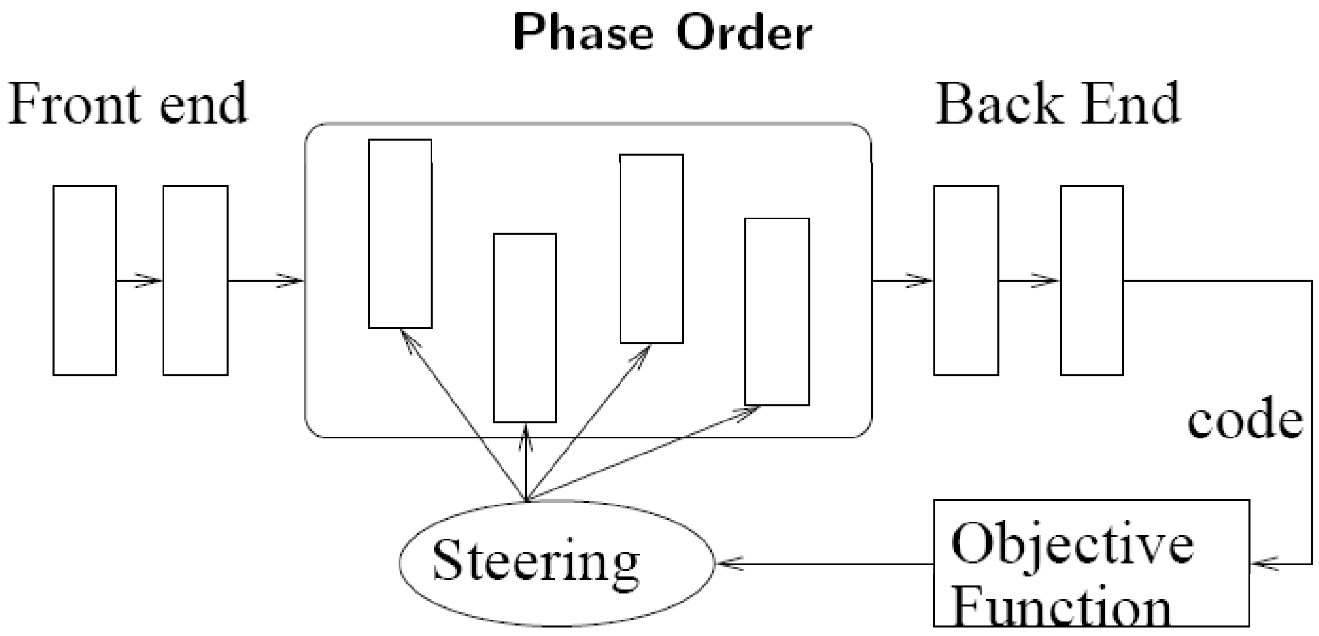
50 steps: within 4.9%

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

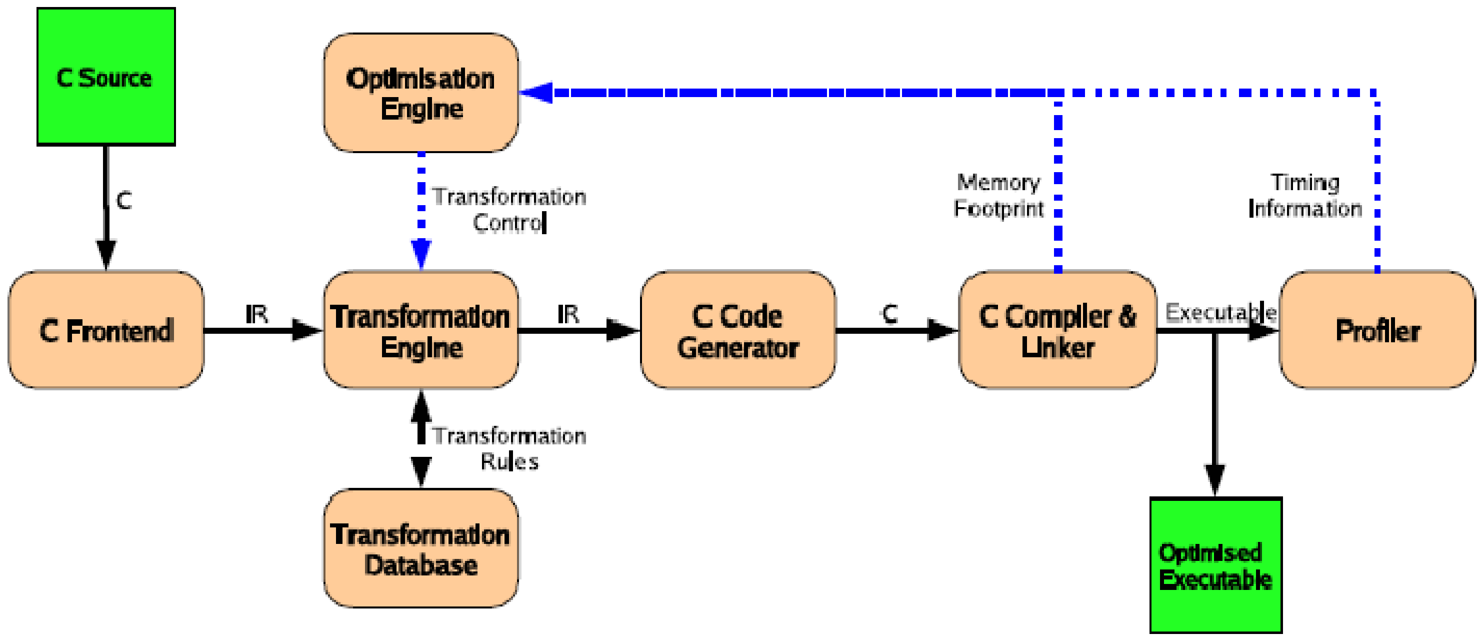
Feedback directed compilation

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

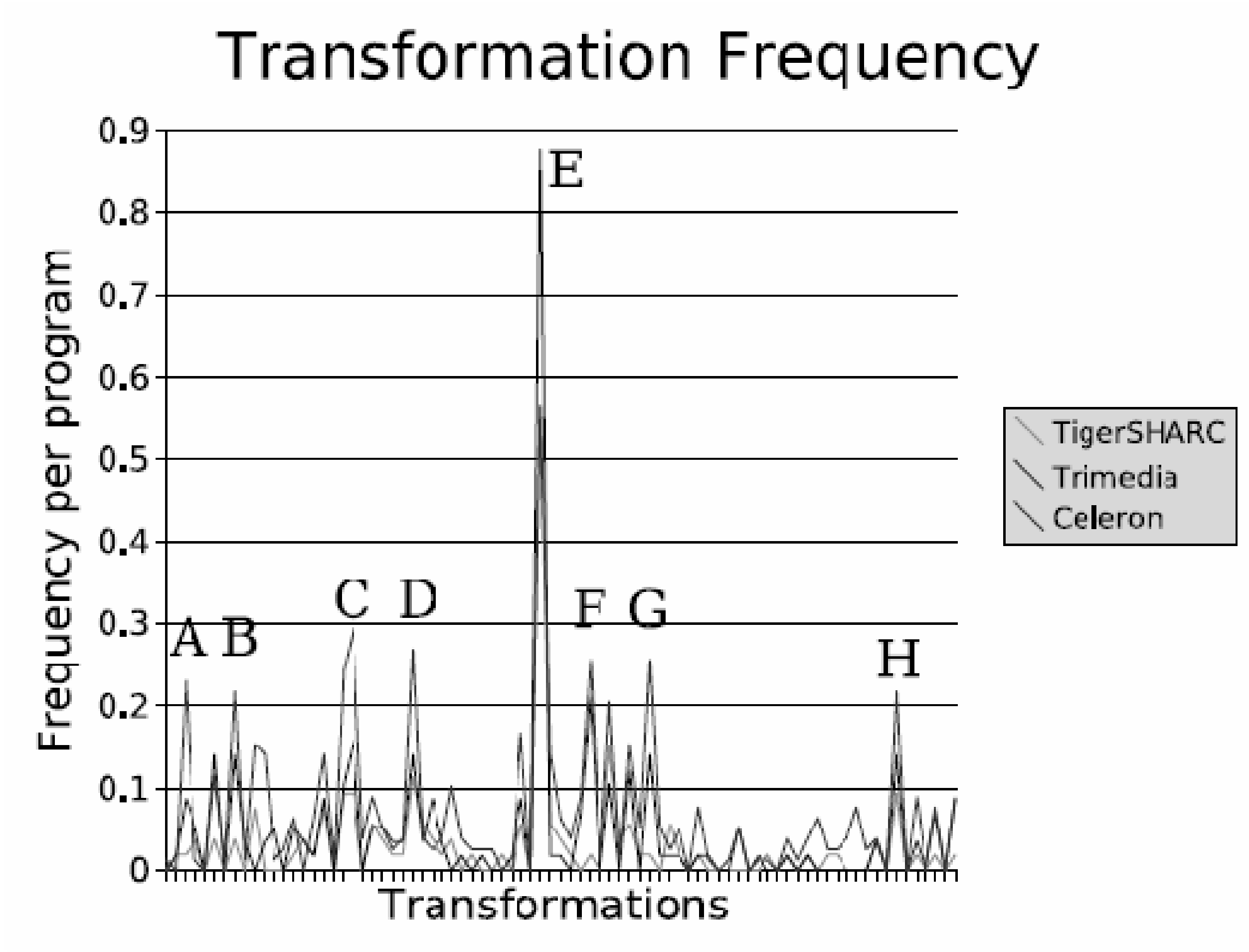
Feedback directed compilation

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)

http://gcc-ici.sourceforge.net

- 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

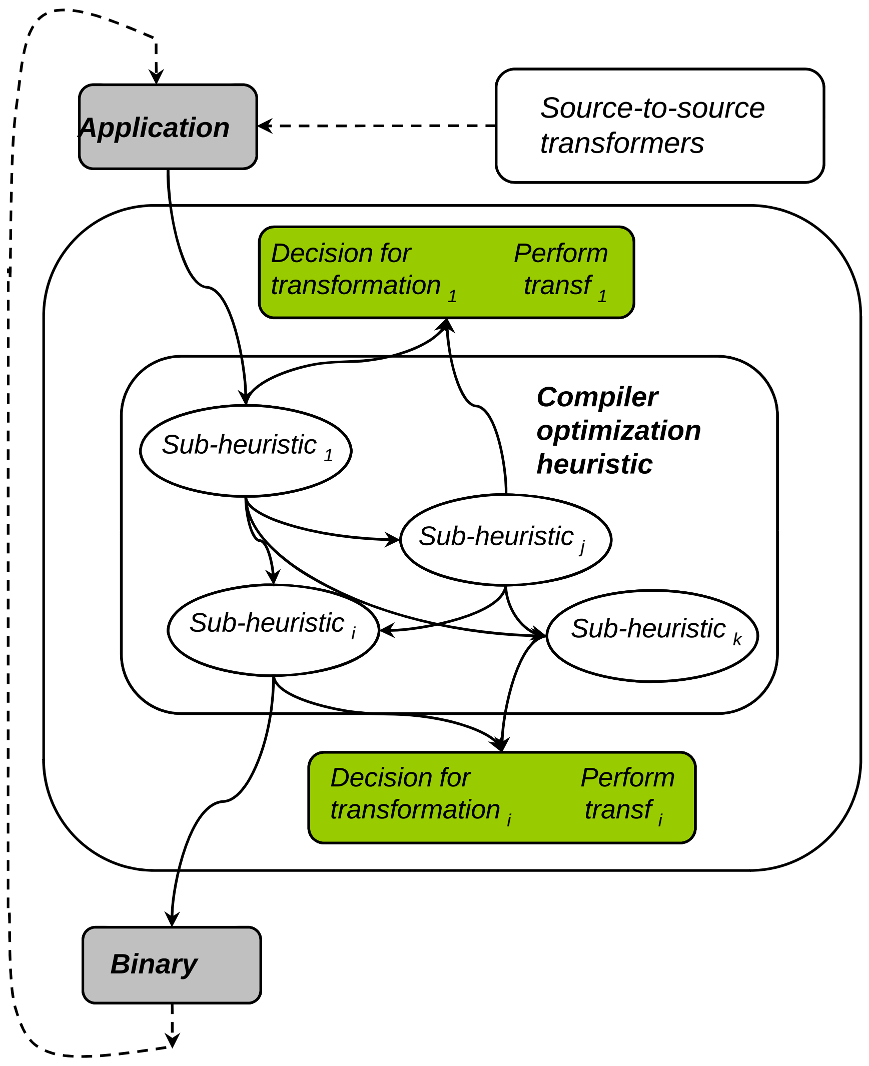
- Enable iterative recompilation algorithm to apply sequences of transformations

- 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

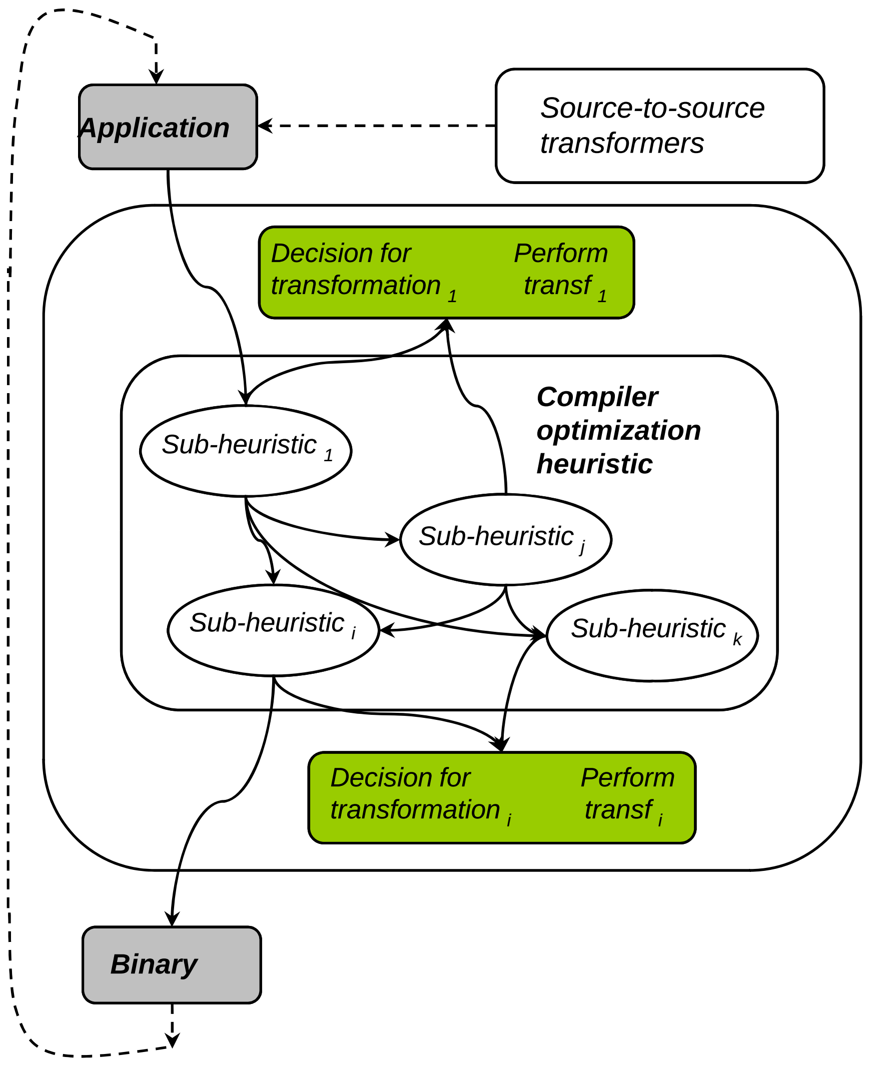
Feedback directed compilation

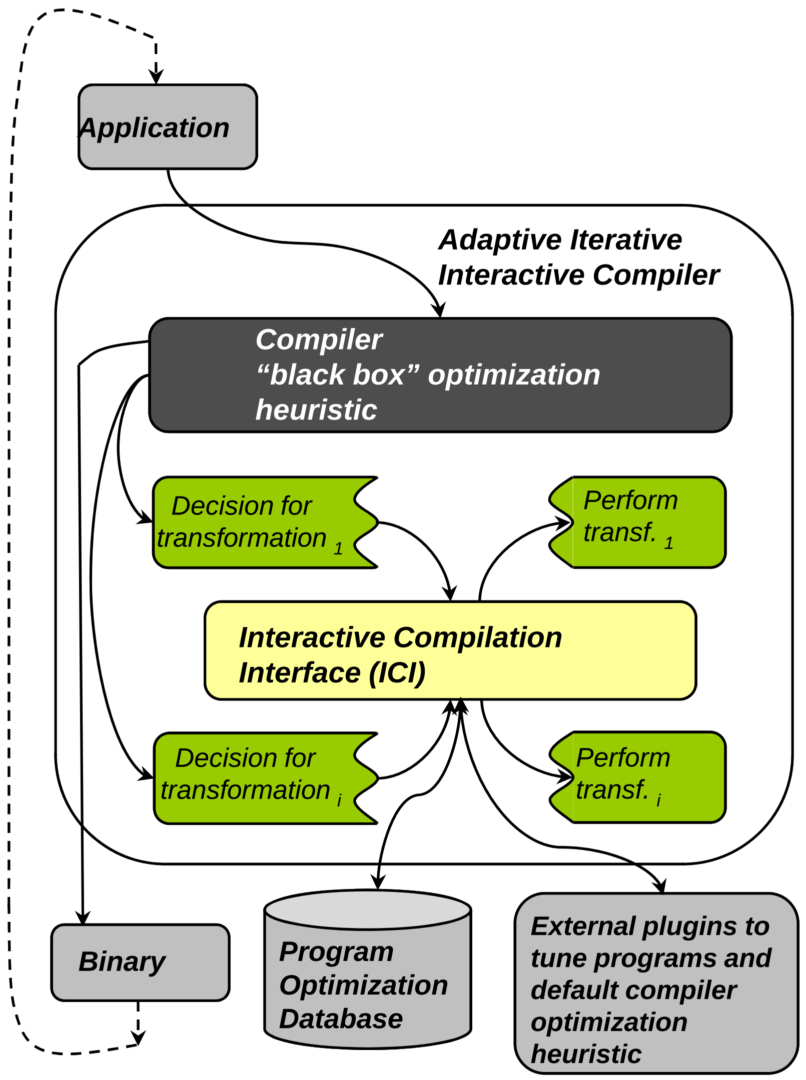
Interactive Compilation Interface



Feedback directed compilation

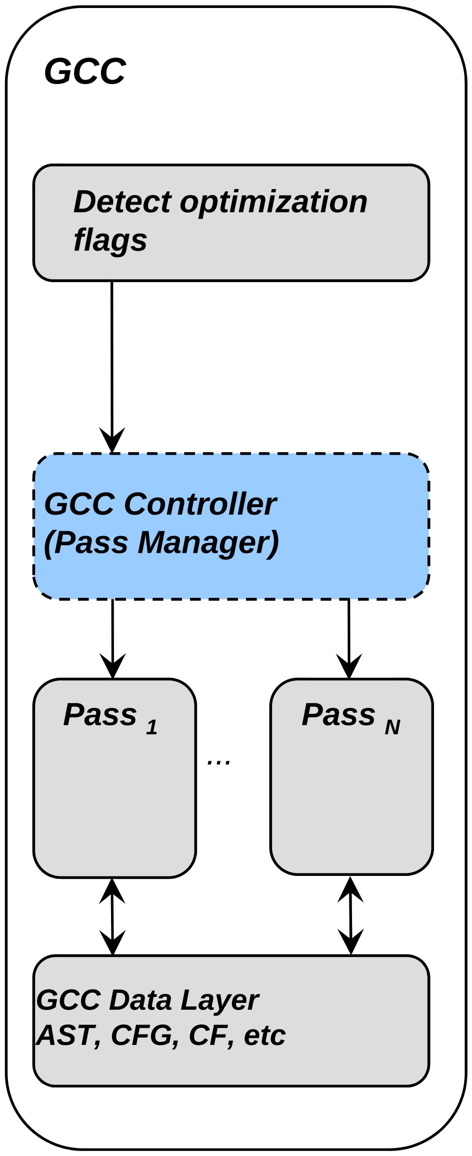
Interactive Compilation Interface



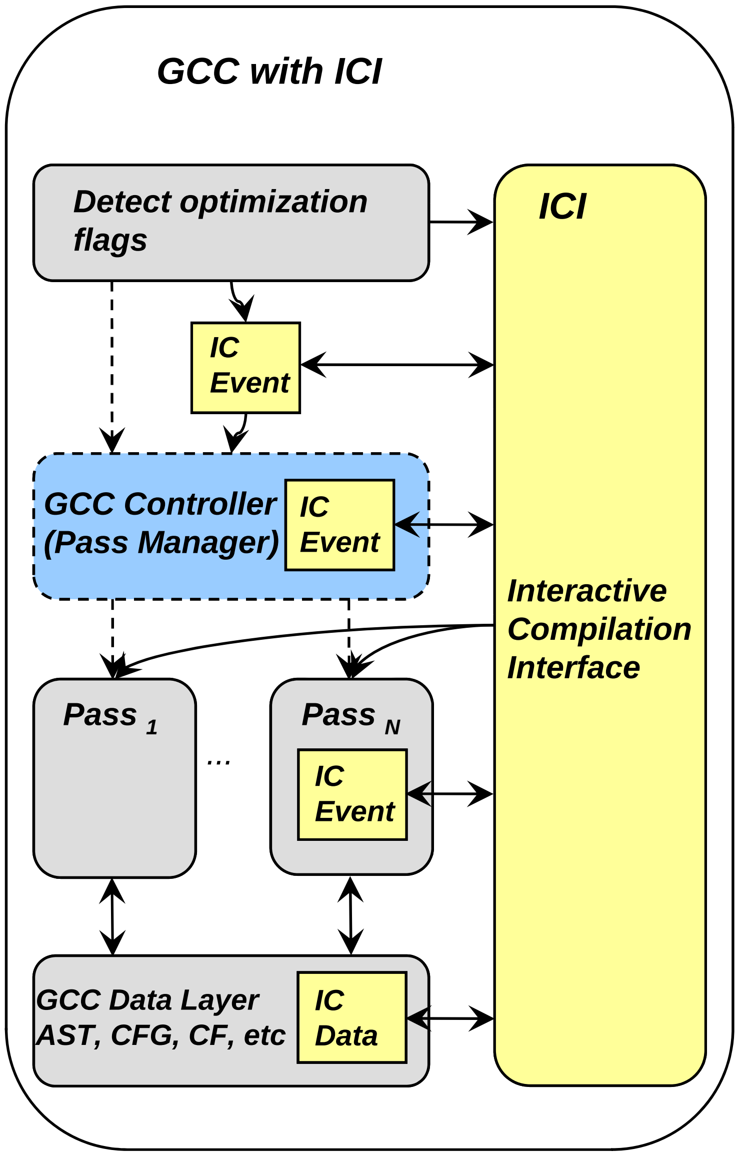


Application
Adaptive Iterative Interactive Compiler
Compiler "black box" optimization heuristic
Decision for transformation $_1$
Perform transf. $_1$
Interactive Compilation Interface (ICI)
Decision for transformation $_i$
Perform transf. $_i$
Binary
Program Optimization Database
External plugins to tune programs and default compiler optimization heuristic

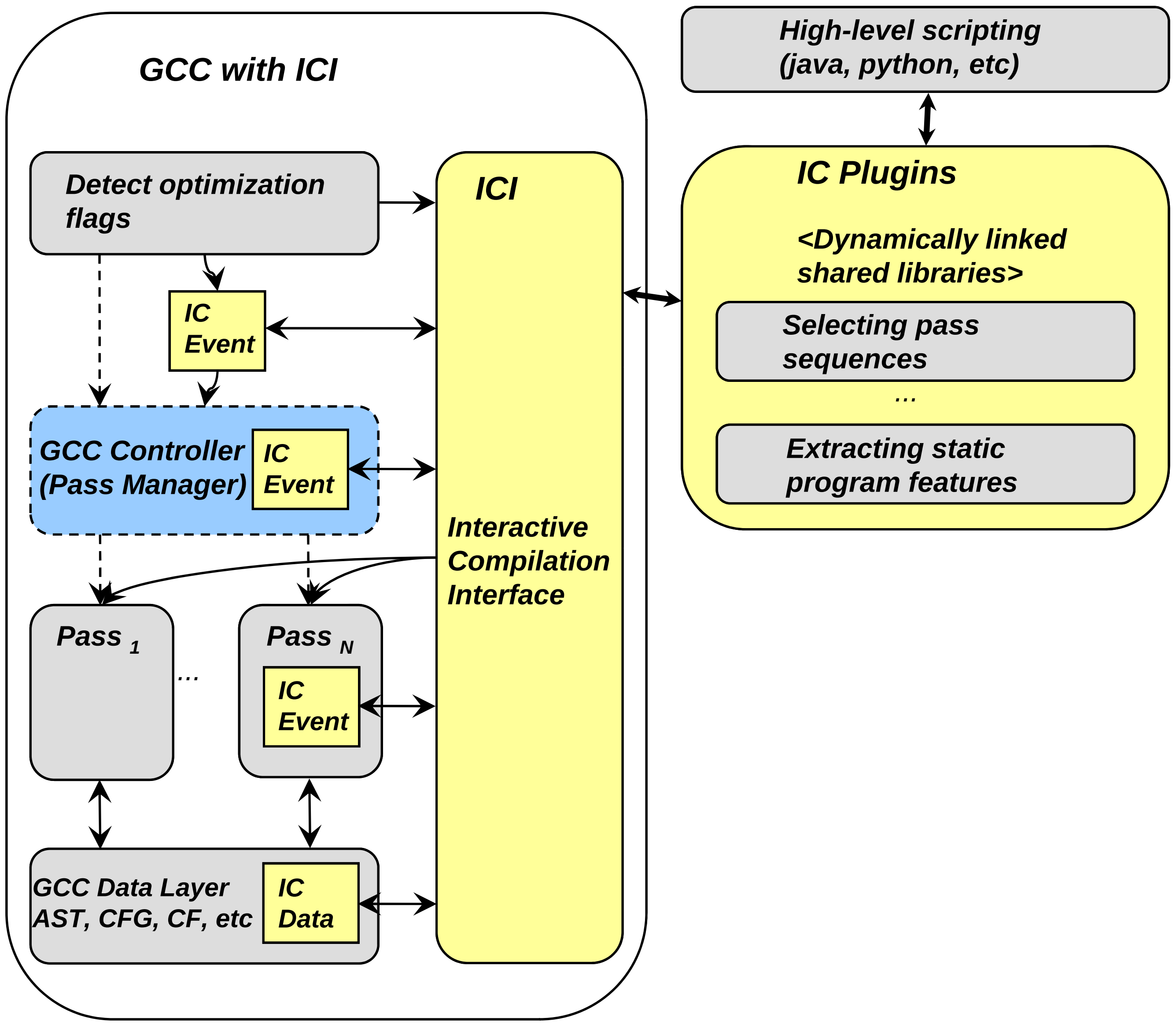
Feedback directed compilation


GCC
Detect optimization flags
GCC Controller (Pass Manager)
Pass 1
…
Pass N
GCC Data Layer AST, CFG, CF, etc

Feedback directed compilation

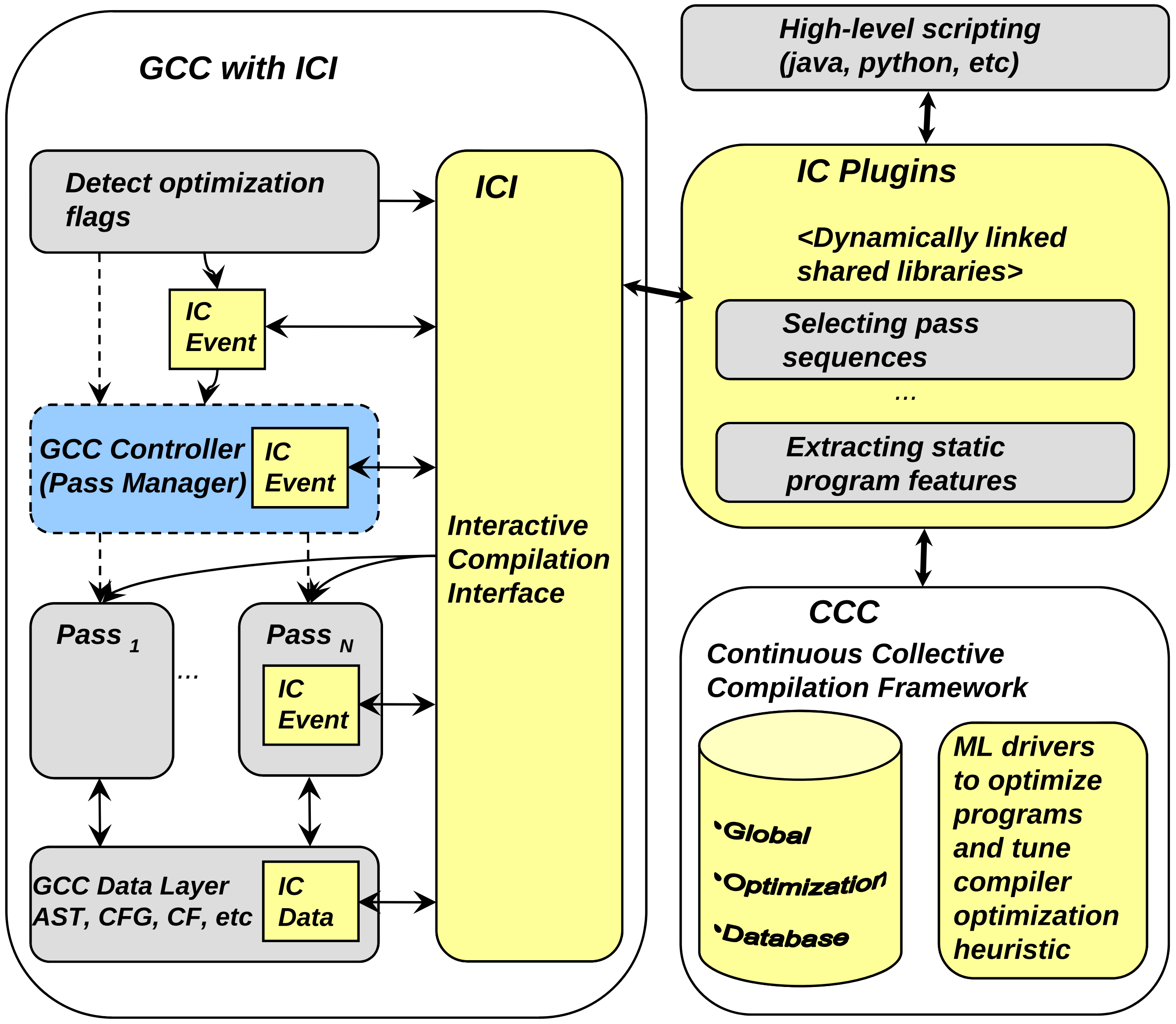

GCC with ICI
Detect optimization flags
ICI
IC Event
GCC Controller (Pass Manager)
IC Event
Interactive Compilation Interface
Pass 1
...
Pass N
IC Event
GCC Data Layer AST, CFG, CF, etc
IC Data

Feedback directed compilation



GCC with ICI
Detect optimization flags
IC Event
GCC Controller (Pass Manager)
IC Event
Pass 1
…
Pass N
IC Event
GCC Data Layer AST, CFG, CF, etc
IC Data
ICI
Interactive Compilation Interface
High-level scripting (java, python, etc)
IC Plugins
<Dynamically linked shared libraries>
Selecting pass sequences
…
Extracting static program features

Feedback directed compilation



GCC with ICI
Detect optimization flags
IC Event
GCC Controller (Pass Manager)
IC Event
Pass 1
…
Pass N
IC Event
GCC Data Layer AST, CFG, CF, etc
IC Data
ICI
Interactive Compilation Interface
High-level scripting (java, python, etc)
IC Plugins
<Dynamically linked shared libraries>
Selecting pass sequences
…
Extracting static program features
CCC
Continuous Collective Compilation Framework
Global Optimization Database
ML drivers to optimize programs and tune compiler optimization heuristic

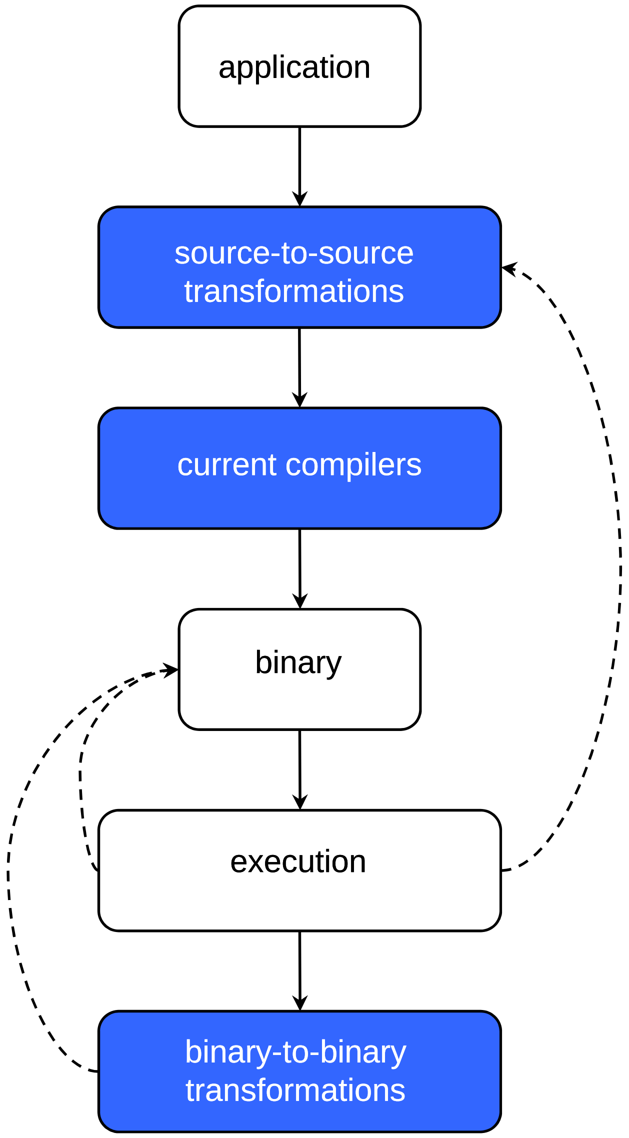
Feedback directed compilation

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);
 }
```

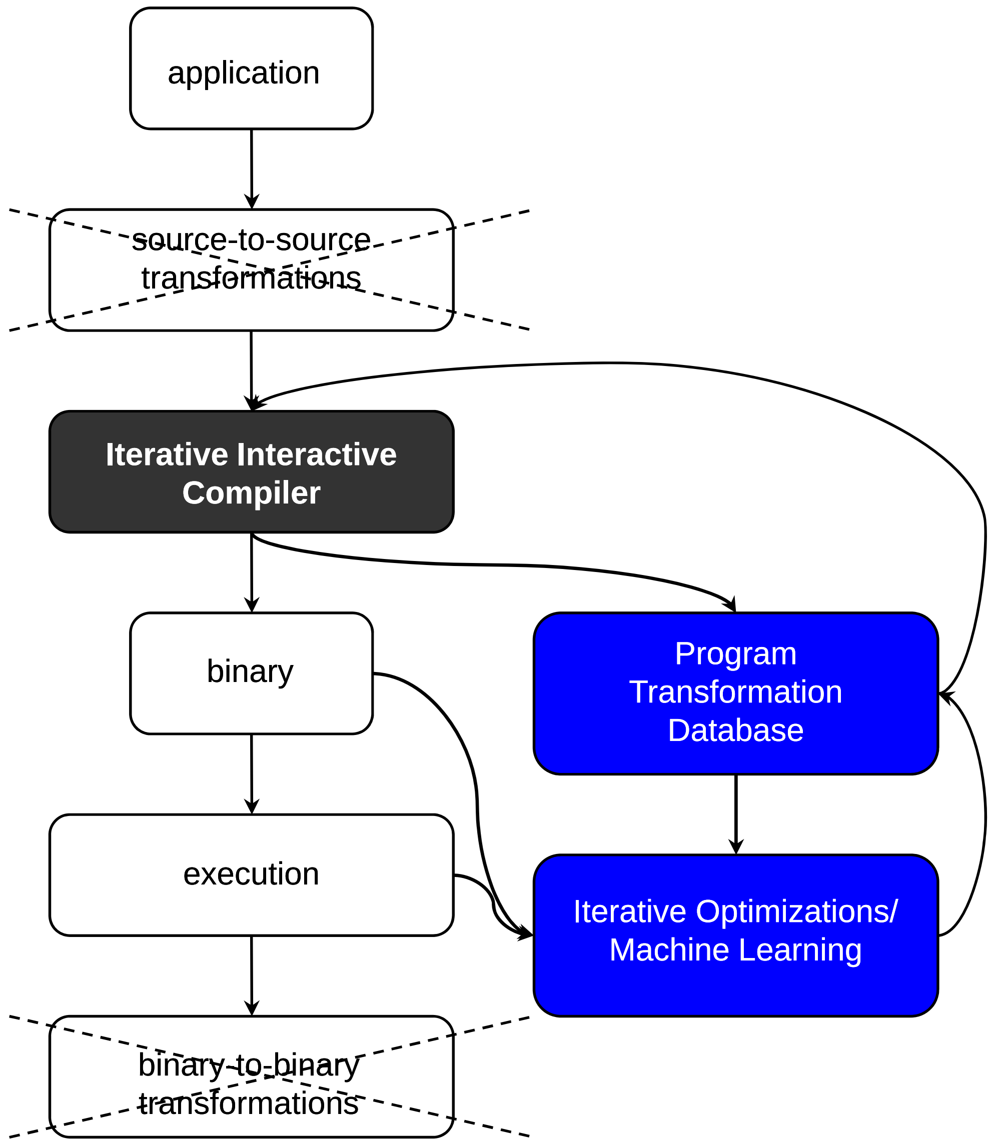
Feedback directed compilation

Continuous Compilation


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

Feedback directed compilation

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

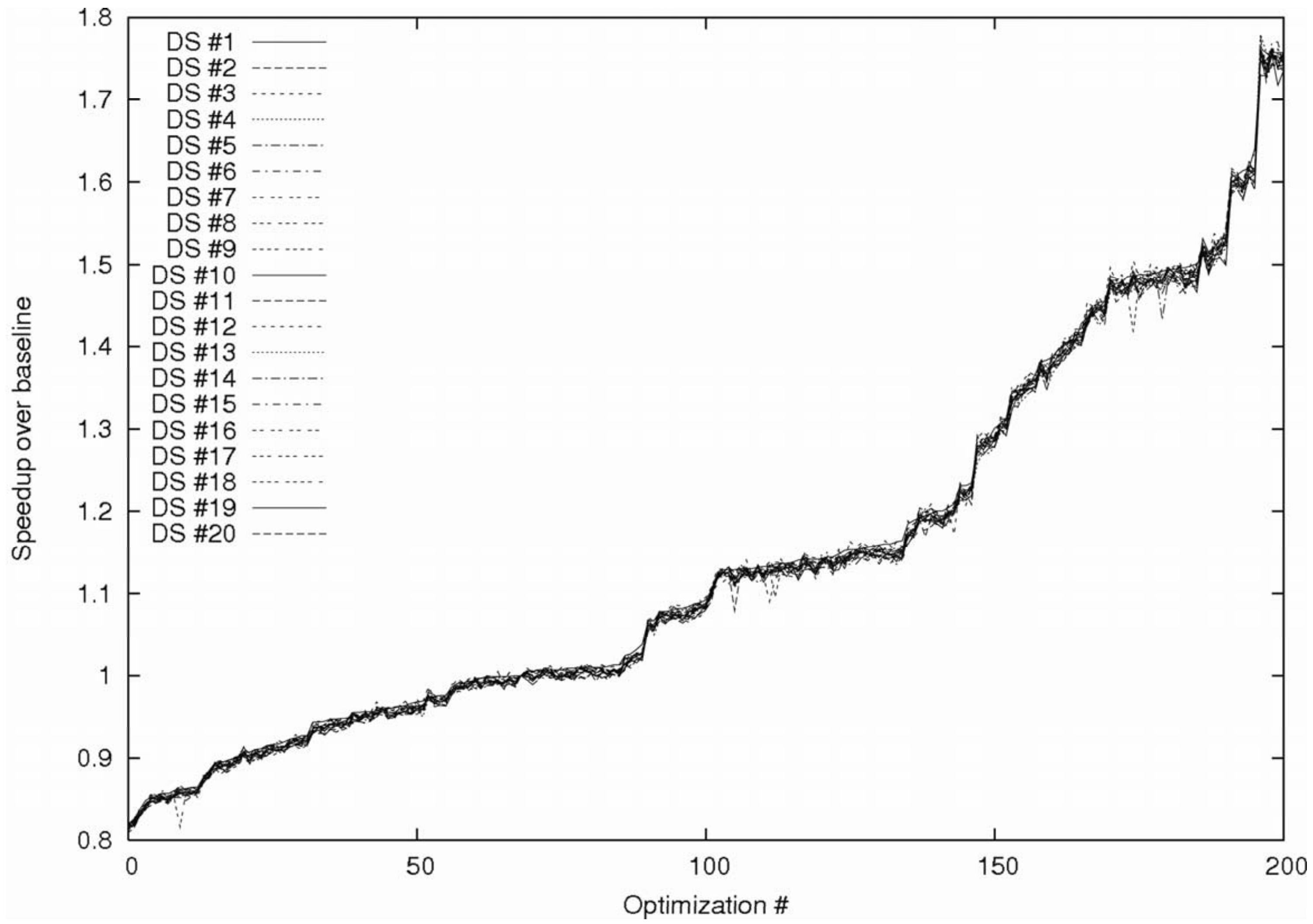
MiDataSets for MiBench – 20 per program

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

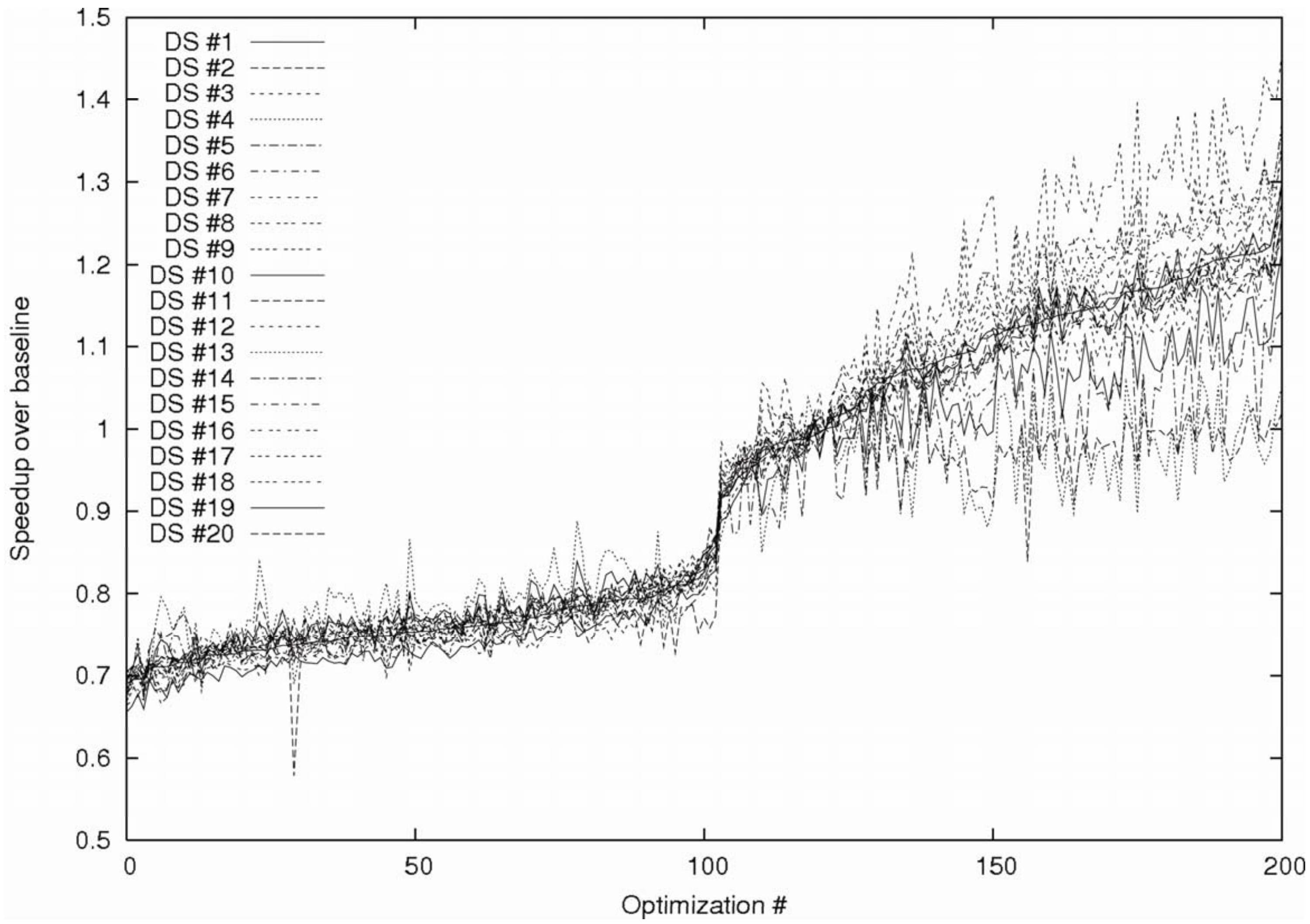
Development website: *http://midatasets.sourceforge.net*

Feedback directed compilation



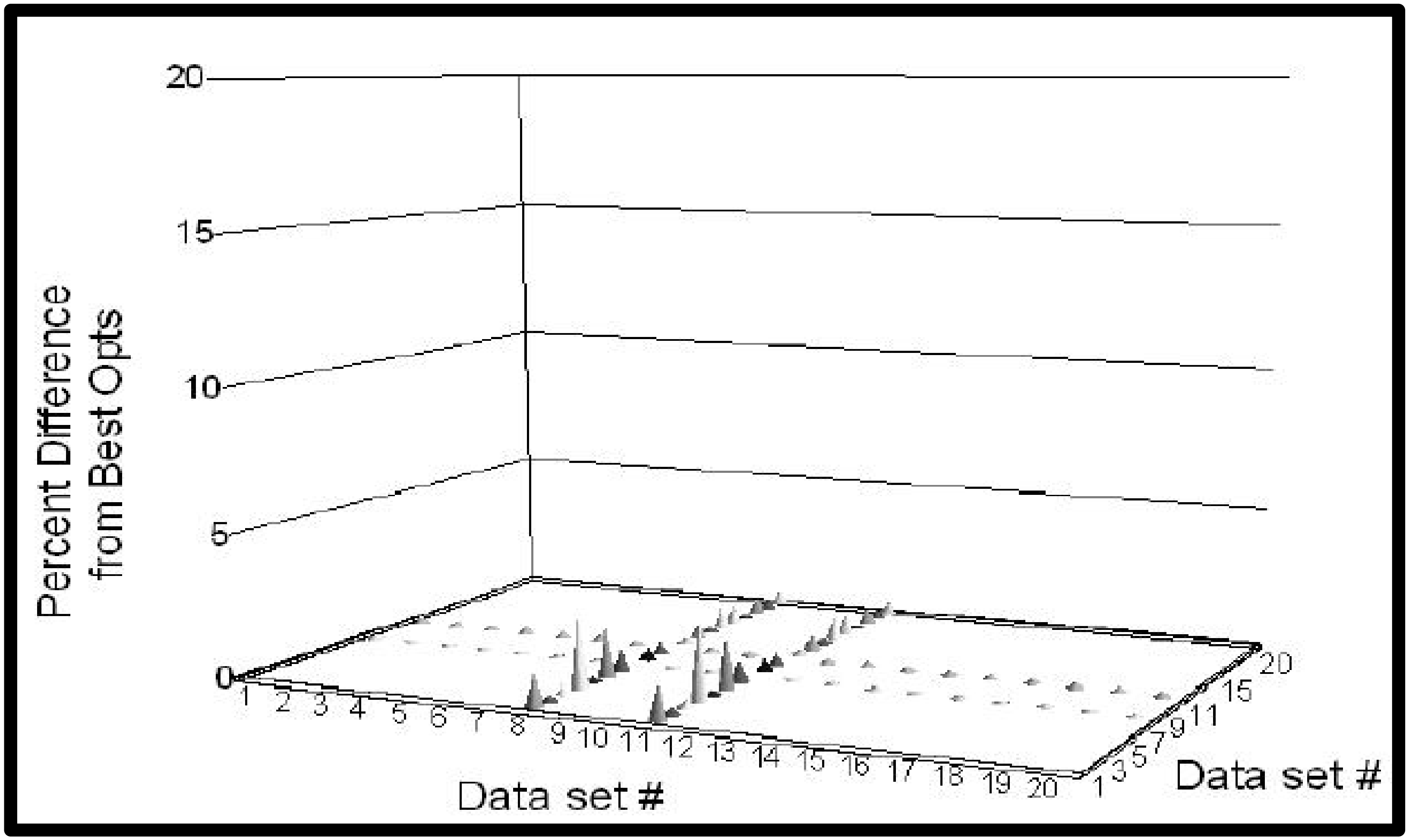
Data sets reactions to optimizations (dijkstra).

Feedback directed compilation



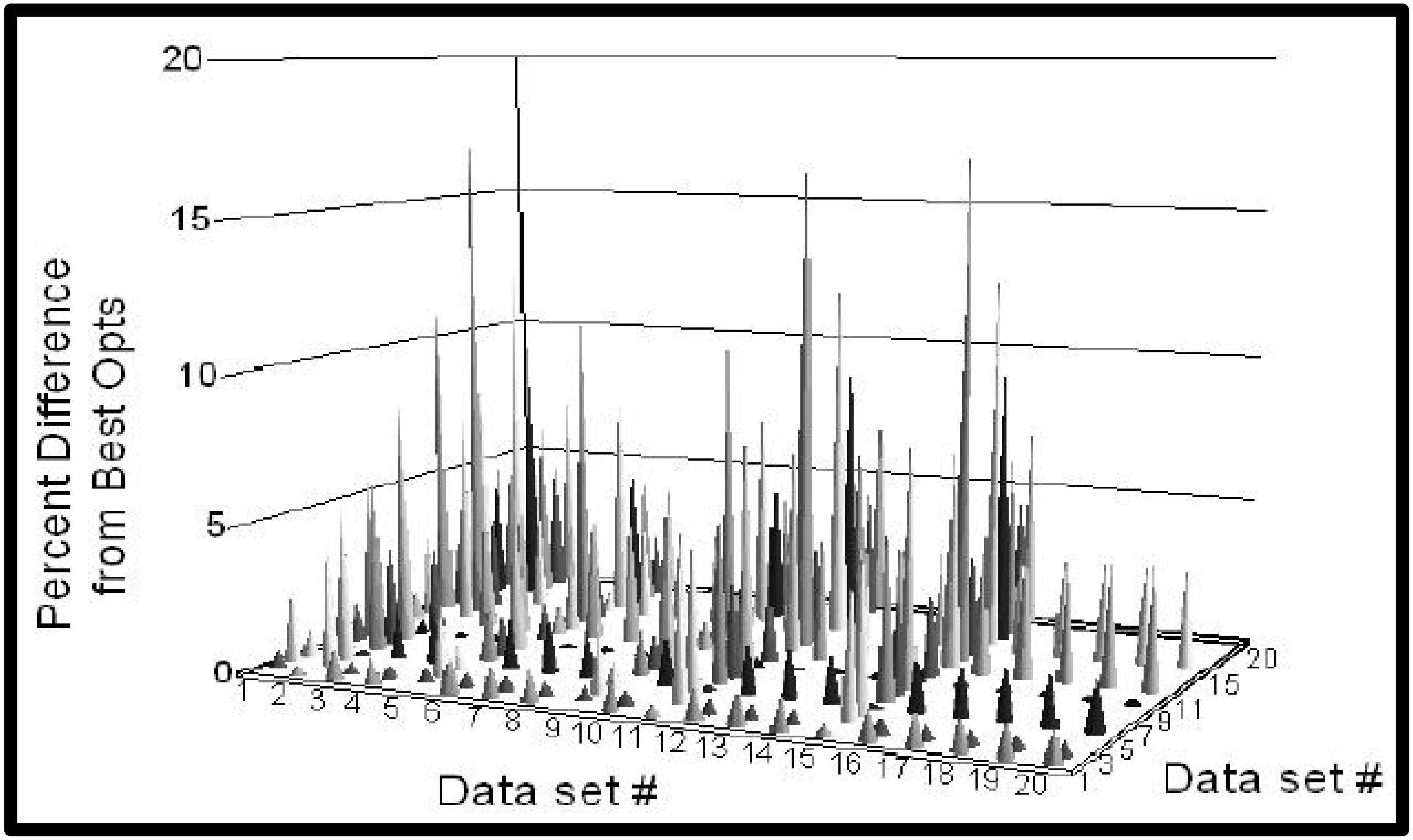
Data sets reactions to optimizations (jpeg decode).

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

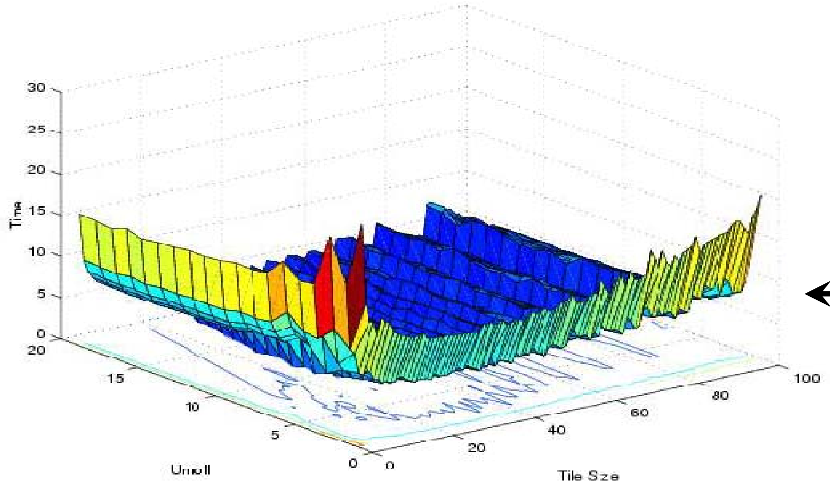
Feedback directed compilation

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.

Conclusions

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

Next lecture

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…

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