Post-link Analysis and Optimization

Yousef Shajrawi
IBM Haifa Research Lab
Work Mail: yousefs@il.ibm.com
Personal Mail: yousef@NoTo.MS

overview, popular tools and examples

Table of Content

Introduction/Motivations

Free (as in Freedom) tools

Free (as in Beer) tools

Post-link optimizations examples

What is post-link analysis and optimization?

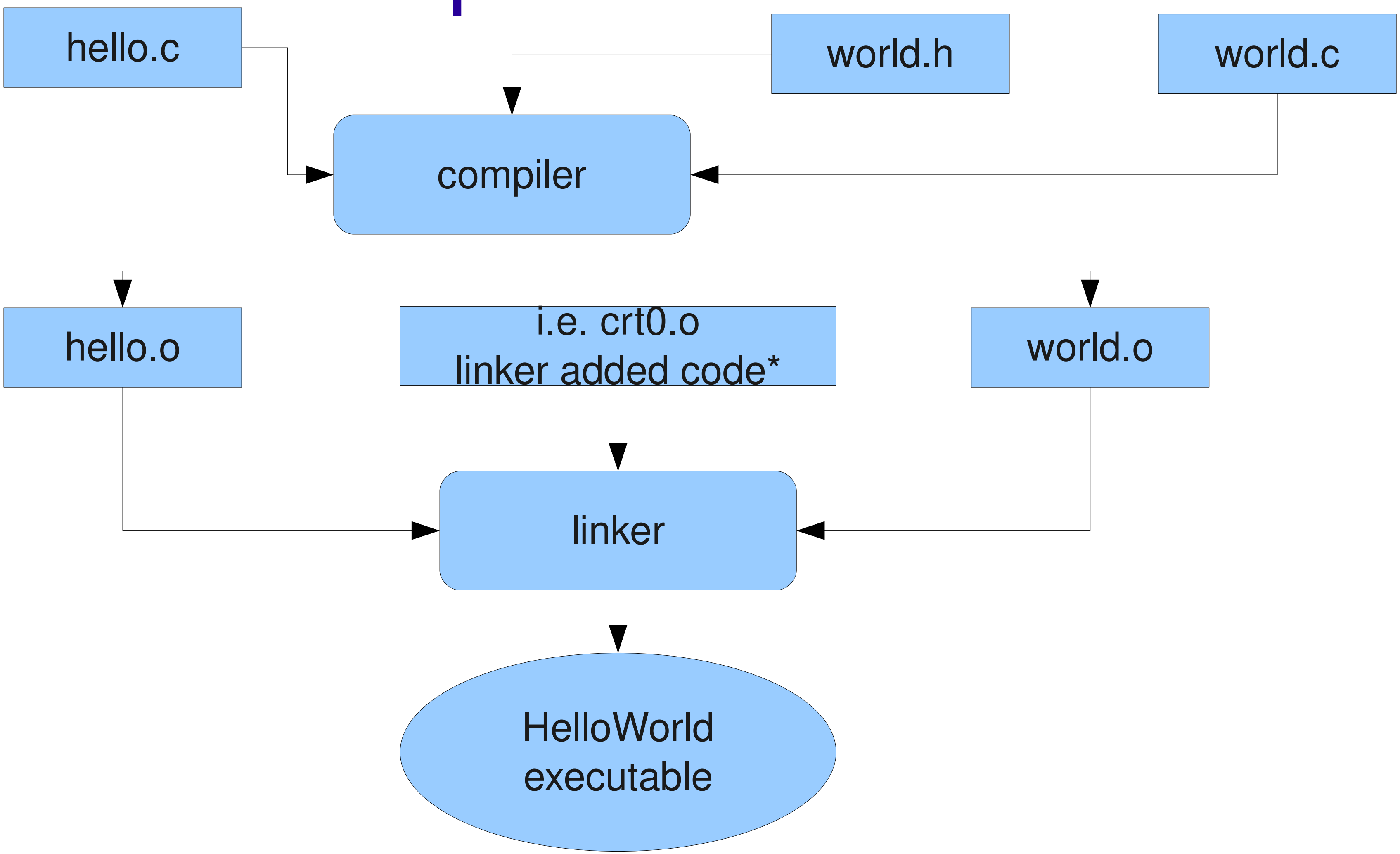
When compiling some program, the compiler turns the source code into 'objects' containing machine code

An optimizing compiler can run different transformations and optimizations to the source of each of these 'objects' to produce a faster/better 'object' (for example, instruction scheduling)

What is post-link analysis and optimization?

- When the compiler finishes producing the 'objects' of a given program we need to 'link' them together to produce a single library or executable binary
- That's the job of the 'linker' that combines the objects produced by the compiler
- The linker doesn't typically run any optimizations on the output file (for example, doing instruction scheduling for the entire program)
  - the GCC community are now working on a linktime optimization framework

What is post-link analysis and optimization?



* start up code and linkage code

What is post-link analysis and optimization?

Here, we are discussing the process of doing analysis and/or optimizations after the linker has finished its job (that is, doing them on the output file), In addition we do optimization that changes the code to something completely new

We are at an advantage of being able to work on all the objects at once and on the output binary directly

We are at a disadvantage of not having the vast knowledge the compiler had such as aliasing information (knowing if separate memory references point to the same location)

What is it good for? - motivation

Producing an 'optimized' binary file that runs 'faster'

Collecting accurate profiling information / frequency statistics

Knowing which static and dynamic data have been accessed

Program verification and Code coverage

working on optimized binary while any changes done during compile time may change the generated code

...Many More!

Free (as in Freedom) tools

- Unfortunately, F/OSS is lacking on this front
- There's no F/OSS post link optimizer for the ELF file format (the one used, among other, by the GNU/Linux OS)
- Post-link analyzers lack certain features compared to Free (as in Beer) offerings

Free (as in Freedom) tools

The SOLAR Project from the university of Arizona aims at developing link-time and run-time code optimizations for Intel's architectures

http://www.cs.arizona.edu/solar/

This work started in the PLTO Link-Time Optimizer

Alto is a free Link-time Code Optimizatier, but only for Alpha/DEC :-(

http://www.cs.arizona.edu/projects/alto/

PIN

Tool for the dynamic instrumentation of programs

Functionality similar to the popular ATOM toolkit for Compaq's Tru64 Unix on Alpha, i.e. arbitrary code (written in C or C++) can be injected at arbitrary places in the executable

Does not instrument an executable statically by rewriting it, but rather adds the code dynamically while the executable is running.

We will Focus on another tool, Valgrind

Valgrind
http://valgrind.org/

GPLed (version 2) instrumentation framework for building dynamic analysis tools which provides various debugging and profiling tools such as Memcheck

Translates the program into IR (Intermediate Representation) which is given for the 'tools' for transformations before being turned back into machine code for the CPU to run

Valgrind

Requires debugging information in the binary

Works best with -O0 (no compiler optimizations)

The 'binary' we want to investigate will runs 10s of times slower than its native speed

Supports x86, AMD64, PPC32 and PPC64 architectures

Valgrind Tools - Memcheck

- The most popular valgrind tool
- A memory checking tool for common memory errors such as:
  - Use of uninitialized values/memory
  - Memory leaks
  - Reading/Writing freed memory or off the end of malloc'd blocks

Valgrind Tools - Cachegrind

Does cache and branch simulations of the program

Can collect statistics about L1/L2 write/read misses

Detects mis predicted conditional branches

Detects mis predicted indirect branch's targets

Valgrind Tools - Callgrind

- A profiling tool that can construct a call graph for a program's run
- Collects the following data:
  - number of instructions executed and their relationship to source lines
  - caller/callee relationship between functions and the numbers of such calls

Valgrind Tools - Others

- Helgrind: tool for detecting synchronization errors in multi threaded code. (such as race conditions and deadlocks)
- Massif: a heap profiling tool
  - Can measure the size of the program's stack(s)

Free (as in Beer) tools

- Post-link optimizers can improve the performance of the program by 10s of %
- Some tools can work on any binary even if has been aggressively optimized by the compiler and has no debugging information
- There's such tools for every major architecture
- We'll be taking a closer look at the tools produced at the IBM Haifa Research Lab

FDPR-Pro

http://www.alphaworks.ibm.com/tech/fdprpro

A feedback-based post-link optimization tool

Collects information on the behavior of the program while the program is used for some typical workload, and then creating a new version of the program that is optimized for that workload

performs global optimizations at the level of the entire executable

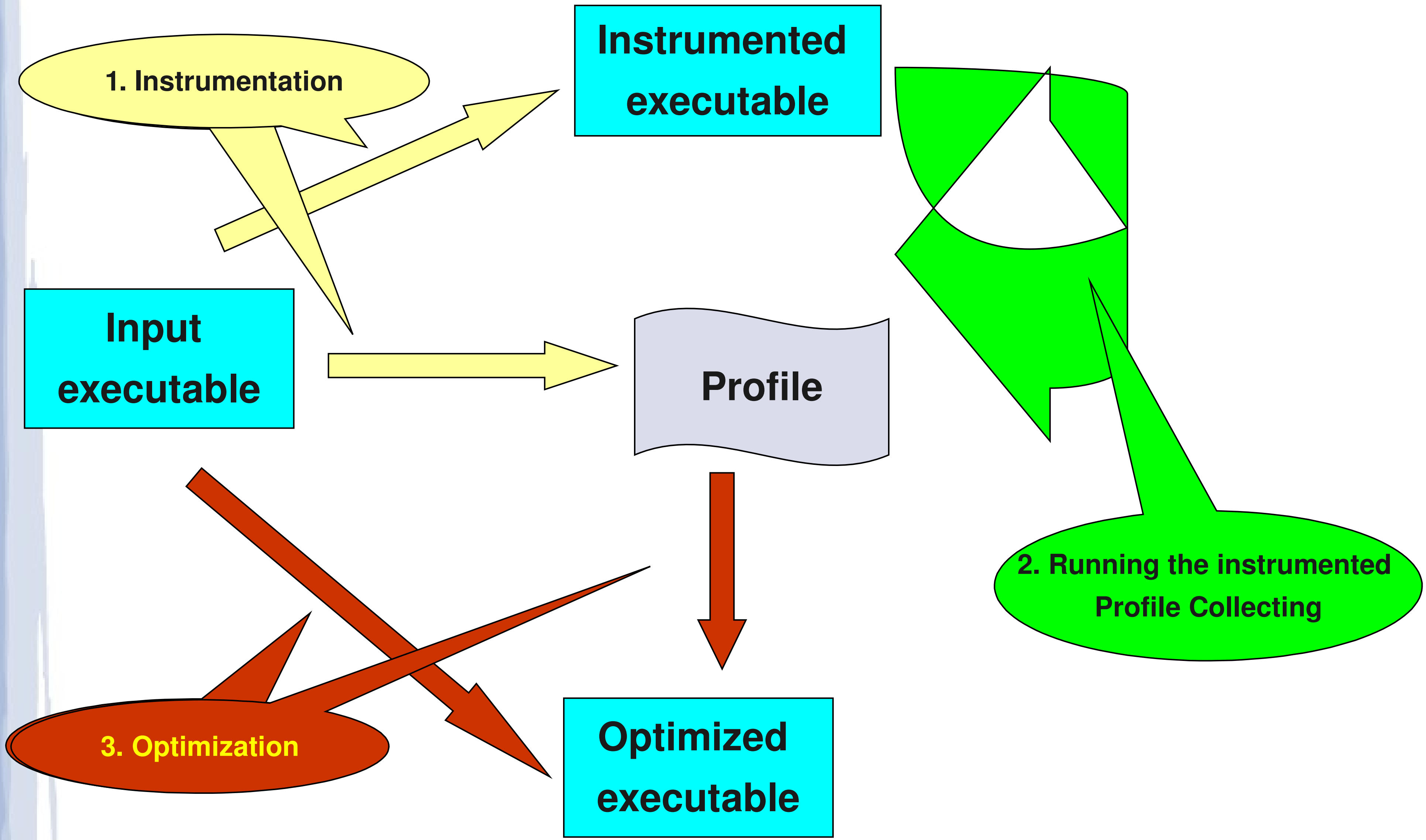
FDPR-Pro

- Since the executable to be optimized by FDPR-Pro will not be re-linked, the compiler and linker conventions do not need to be preserved, thus allowing aggressive optimizations that are not available to optimizing compilers
  - It Improves code and static data locality
    - Reduces cache miss rate
    - Improves branch prediction rate

FDPR-Pro Collecting profiling (Training)

- In this phase the user runs the instrumented executable
- The user runs it with a usual invocation command, the same way he would run the original executable
- fdprpro does not run in this phase
- The user should choose representative workload in order to receive good optimization results

FDPR-Pro Operation



FDPR-Pro

Running FDPR-Pro from Command Line – Typical Example

```
> fdprpro –a instr myexe –f myexe.prof –o myexe.instr

> myexe.instr

> fdprpro –a opt myexe –f myexe.prof –o myexe.fdpr
```

FDPR-Pro
Optimization Phase

- The are 5 levels of optimization, -O is the basic one, -O5 is the most aggressive
- basic optimizations include:
  - Code Reordering
  - NOOP removal
  - Branch Prediction Bit Setting

FDPR-Pro Code Reordering

Reduce the number of I-cache misses

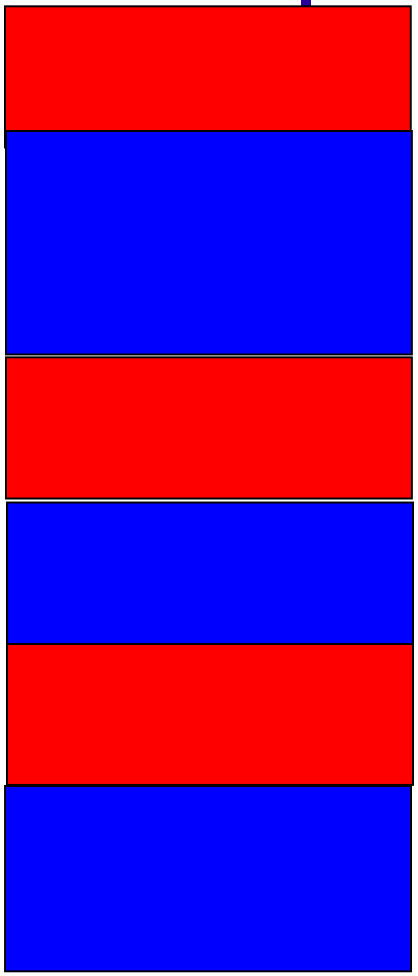
Reduce the number of I-TLB misses

Reduce the number of page faults

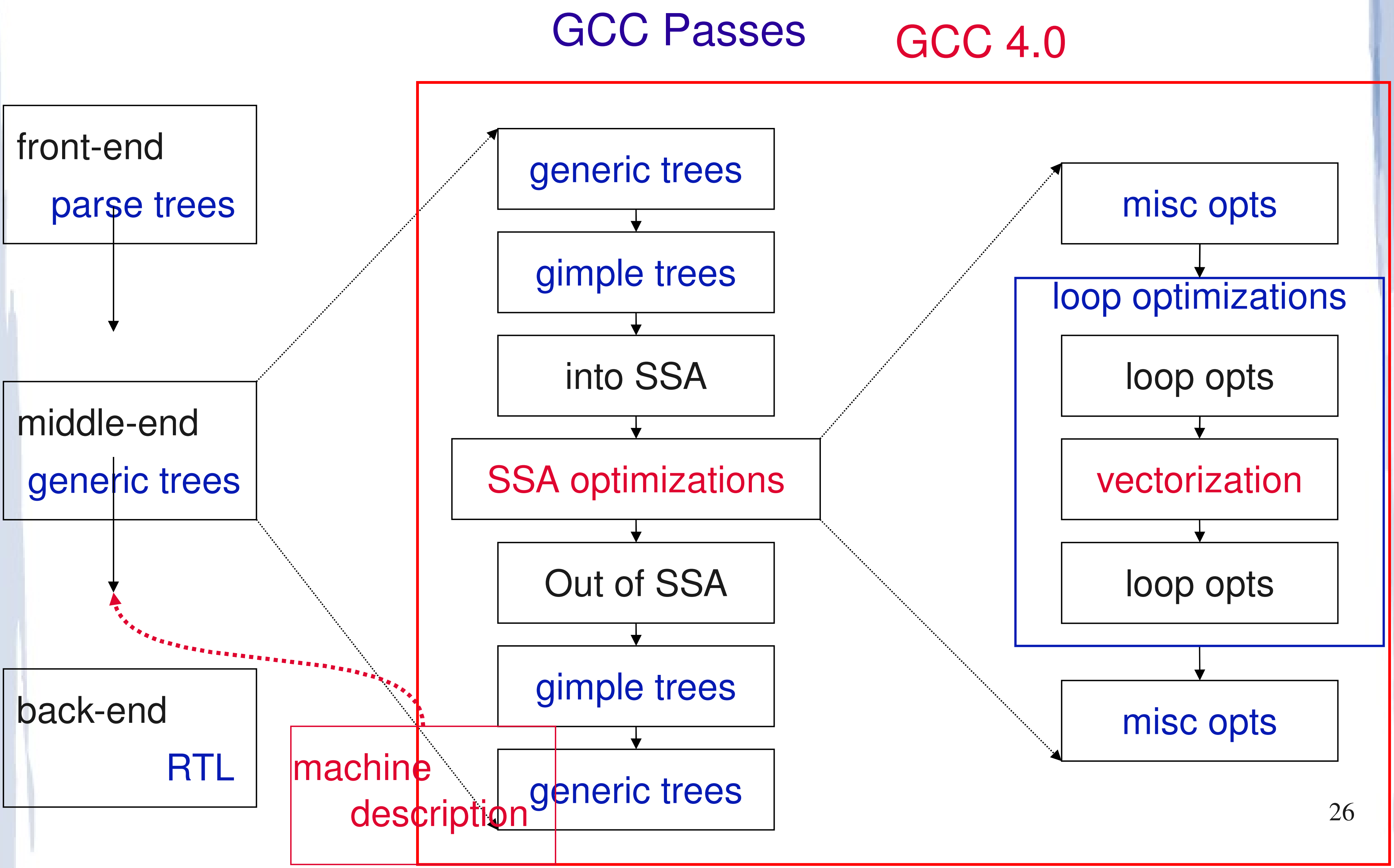
Reduce the branch penalty

Improve branch prediction

Code Reordering – The basic FDPR-Pro optimization



High Level Representation



FDPR-Pro
High Level Representation (HLR)

HLR is not (just) a layer for optimizations

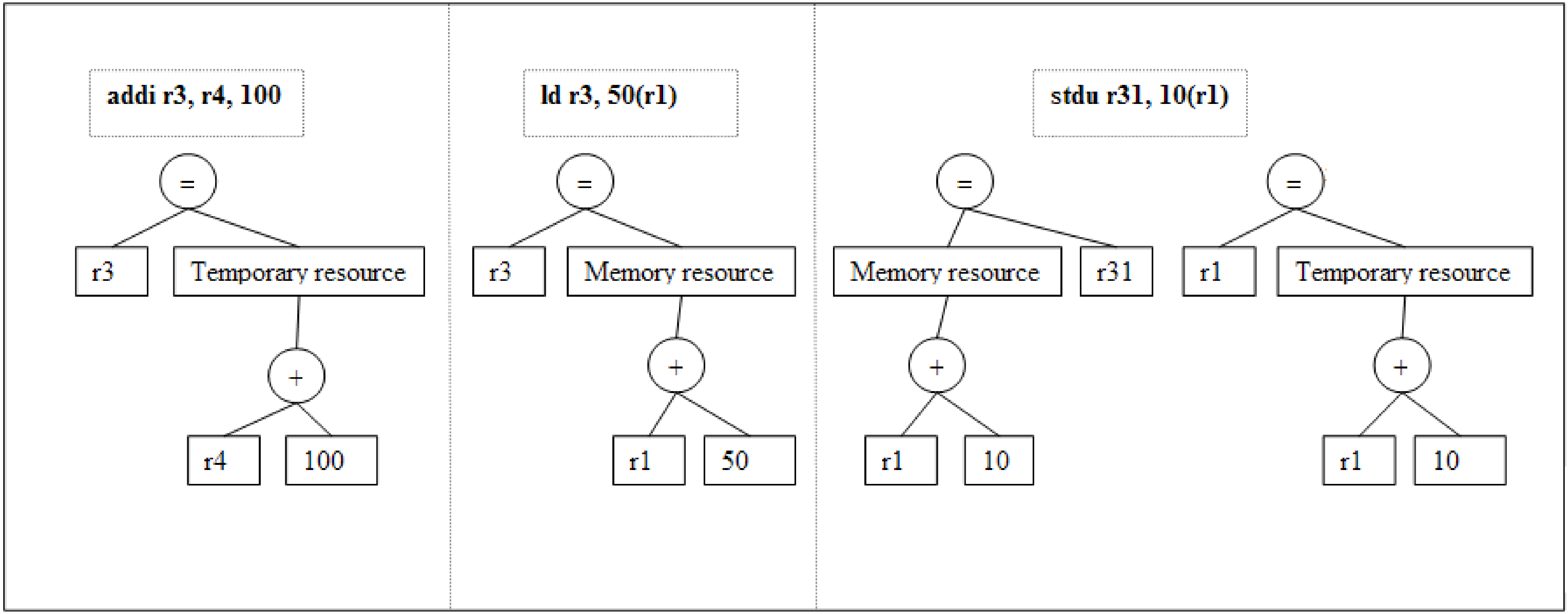
- Platform independent layer for data flow analysis
- Serves in the analysis of Binaries
- Development of cross platform branch table analysis

FDPR-Pro
High Level Representation

Includes

- AbsAsm
  - Similar to RTL (register transfer language, an IR close to assembly language) in compilers
  - Support aliasing for memory resources and register alias sets
  - Extendable to support SSA (static single assignment form, IR in which every variable is assigned exactly once) - using virtual registers
- PartialCFG (Partial Control Flow Graph)
  - Encapsulated calling convention and ABI information
  - Not restricted to single procedure

Abstract assembly



Abstract assembly (continued)

Machine independent representation

Well suited for calculating constant values

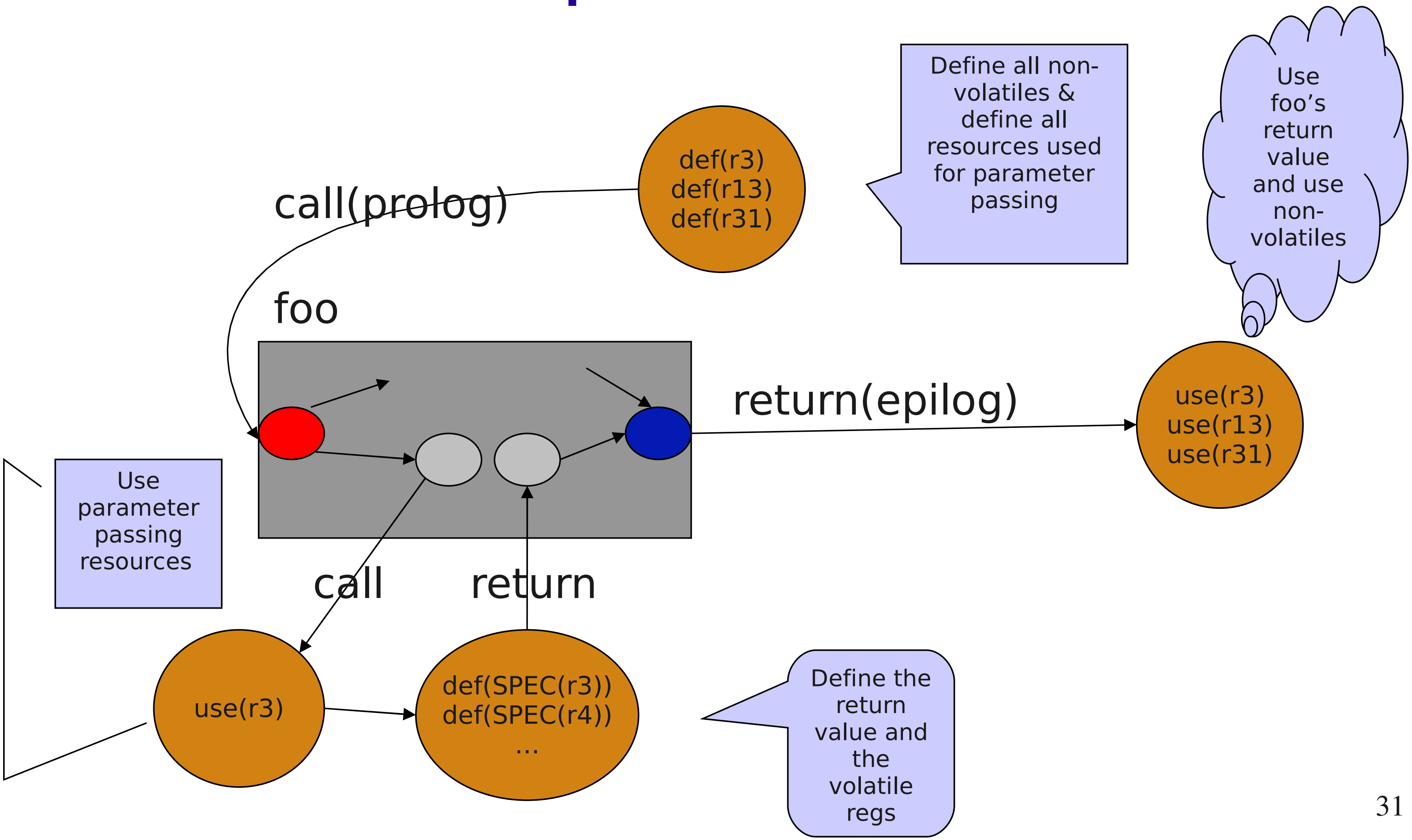
Virtual instructions

- def/use instructions which are used to specify calling ABIs.
- future use can also include *phi* functions for SSA-form

Polymorphic instructions

- By replacing resources in an instruction the instruction may change all-together
- For instance a *load* instruction may change to a *move* instruction
- Support caching

PCFG representation



FDPR-Pro HLR Pros

- a cross platform frame work for data flow optimizations/analysis on binary executables
- Optimizations written over HLR increase performance by:
  - Operating on inlined functions in their new context (more on that later)
  - Operating on scopes larger than single functions
  - makes development of new optimizations easier

Code Analyzer

http://www.alphaworks.ibm.com/tech/vpa

An Eclipse (a platform containing, extensible framework and great IDE: eclipse.org) based plugin that can display feedback-directed performance information about a given application

Based on FDPR-Pro performance tools (its engine for analysis and instrumentation)

Displays assembly instructions, BBs (basic blocks), functions, CSECT (Control Section, unified group of code/data) modules, control flow graph, hot (high execution count) loops, call graph, and annotated code.

Code Analyzer

Able to read in profiling information generated by the tprof (reads tprof/oprofile through the ETM/OPM formats) or FDPR-Pro

Can map given assembly (or machine) code back to its source code, when source files are available.

Can instrument executables or shared libraries in order to collect accurate frequency statistics

Supports a variety of binary file formats: XCoff, Elf file formats containing Power PC code, jita2n files, ox lst files (AS400) and z/OS LM files.

Part of VPA - Visual Performance Analyzer

CodeAnalyzer

Provides several views of the input binary

Assembly instructions

Basic blocks

Procedures

CSECT modules

Control flow graph

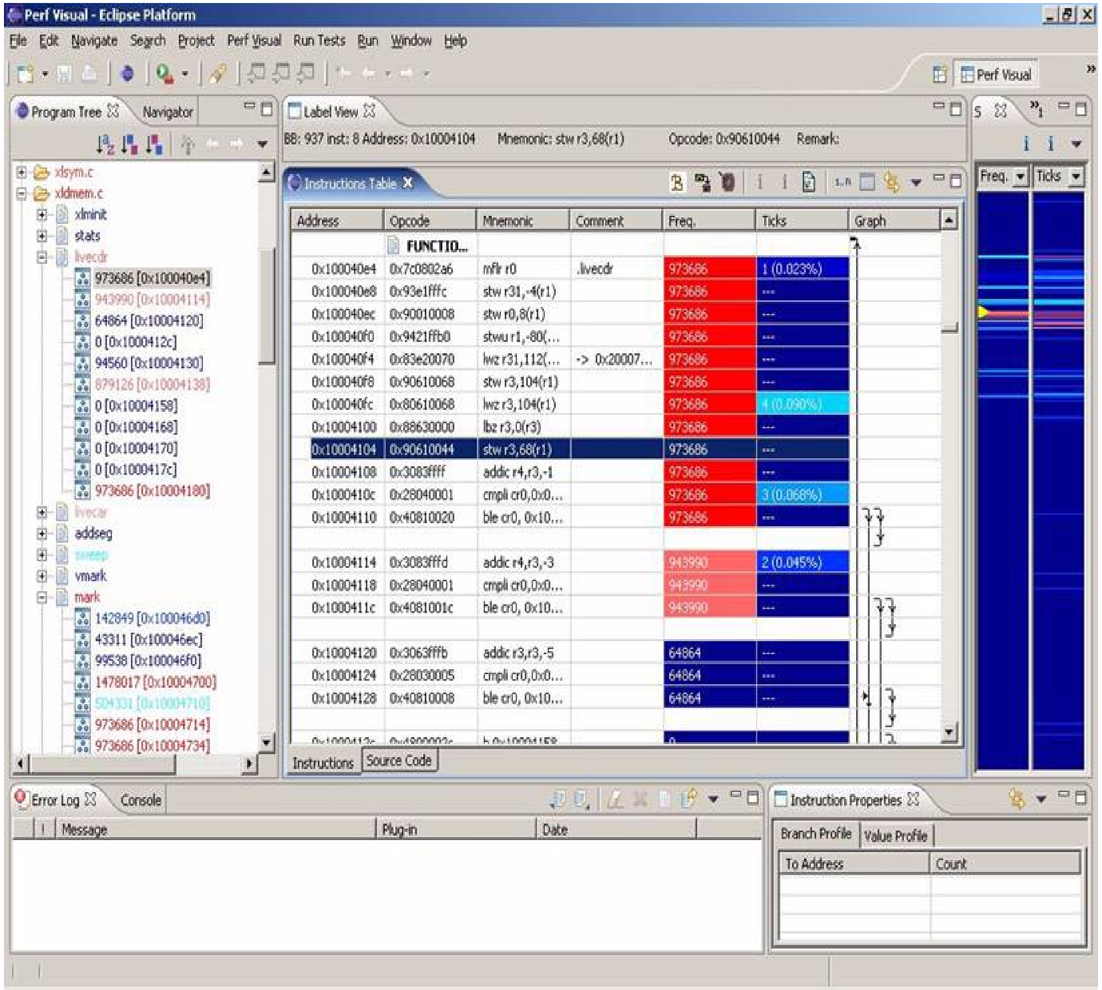
Hot loops

Call graph

Annotated source code

Dispatch group formation

Pipeline slots and functional units



Code Analyzer Features

Showing Disassembly of The Program

Program tree of an EXE

Colors Indicating Hotness of Code/Function...

Grouping Info

Performance comments

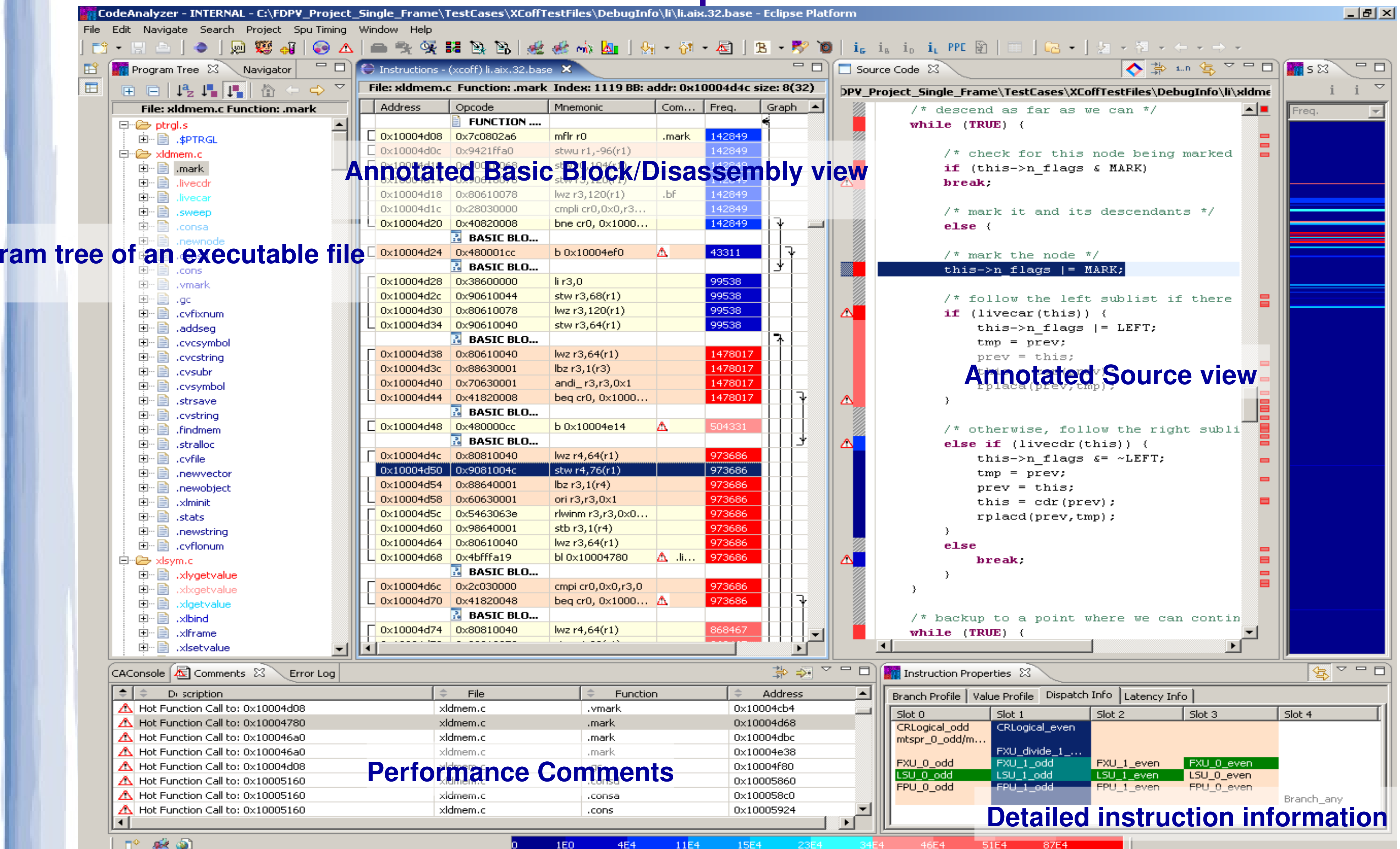
Statistics about the program

bidirectional mapping between source code and assembly

Editing Mode (changing instructions...)

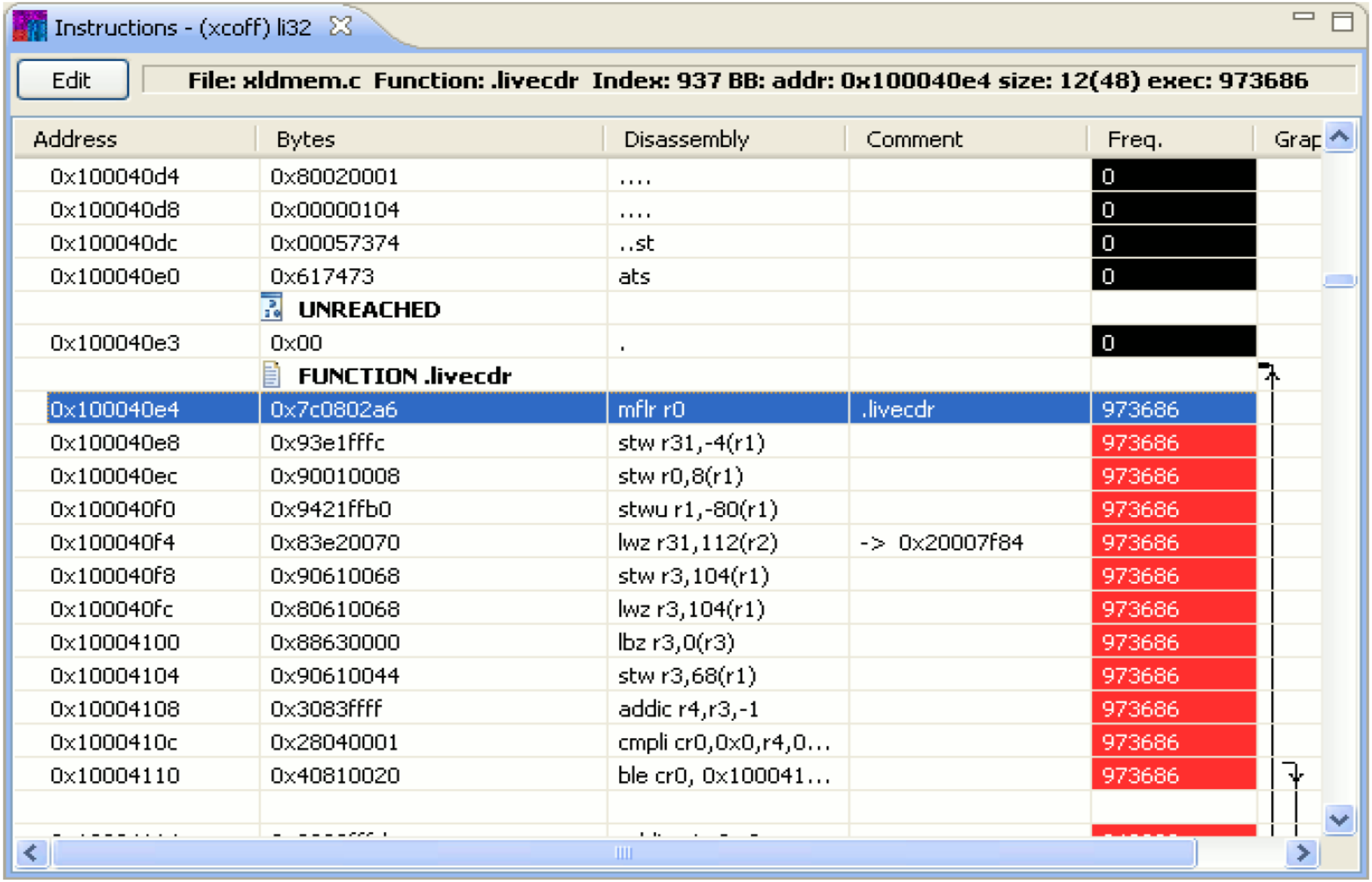
Code Analyzer

Sample View



Code Analyzer

Instruction Editor



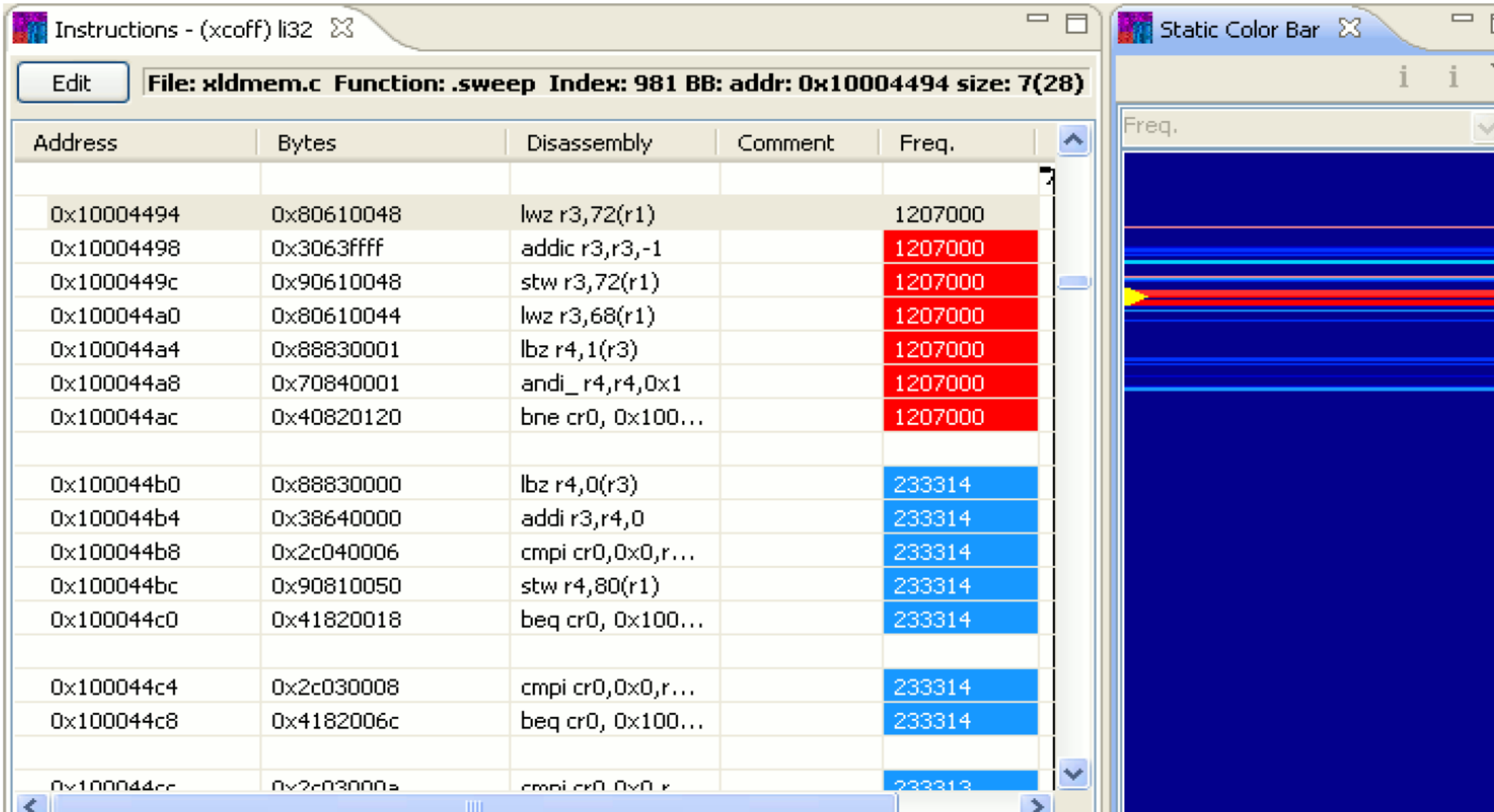
Instructions - (xcoff) li32

Edit | File: xldmem.c Function: .livecdr Index: 937 BB: addr: 0x100040e4 size: 12(48) exec: 973686

Address	Bytes	Disassembly	Comment	Freq.	Grap
0x100040d4	0x80020001	....		0	
0x100040d8	0x00000104	....		0	
0x100040dc	0x00057374	..st		0	
0x100040e0	0x617473	ats		0	
	UNREACHED				
0x100040e3	0x00	.		0	
	FUNCTION .livecdr				
0x100040e4	0x7c0802a6	mflr r0	.livecdr	973686	
0x100040e8	0x93e1fffc	stw r31,-4(r1)		973686	
0x100040ec	0x90010008	stw r0,8(r1)		973686	
0x100040f0	0x9421ffb0	stwu r1,-80(r1)		973686	
0x100040f4	0x83e20070	lwz r31,112(r2)	-> 0x20007f84	973686	
0x100040f8	0x90610068	stw r3,104(r1)		973686	
0x100040fc	0x80610068	lwz r3,104(r1)		973686	
0x10004100	0x88630000	lbz r3,0(r3)		973686	
0x10004104	0x90610044	stw r3,68(r1)		973686	
0x10004108	0x3083ffff	addic r4,r3,-1		973686	
0x1000410c	0x28040001	cmpli cr0,0x0,r4,0...		973686	
0x10004110	0x40810020	ble cr0, 0x100041...		973686	

Code Analyzer

overview of frequency distribution of BBs and instructions



Code Analyzer

Comments View

used to display the comments (which can help you edit the code, investigate various performance problems ...etc) collected by loaded profile file. It provides the file, function and address of the instruction which is tagged with specific comments.

Comments

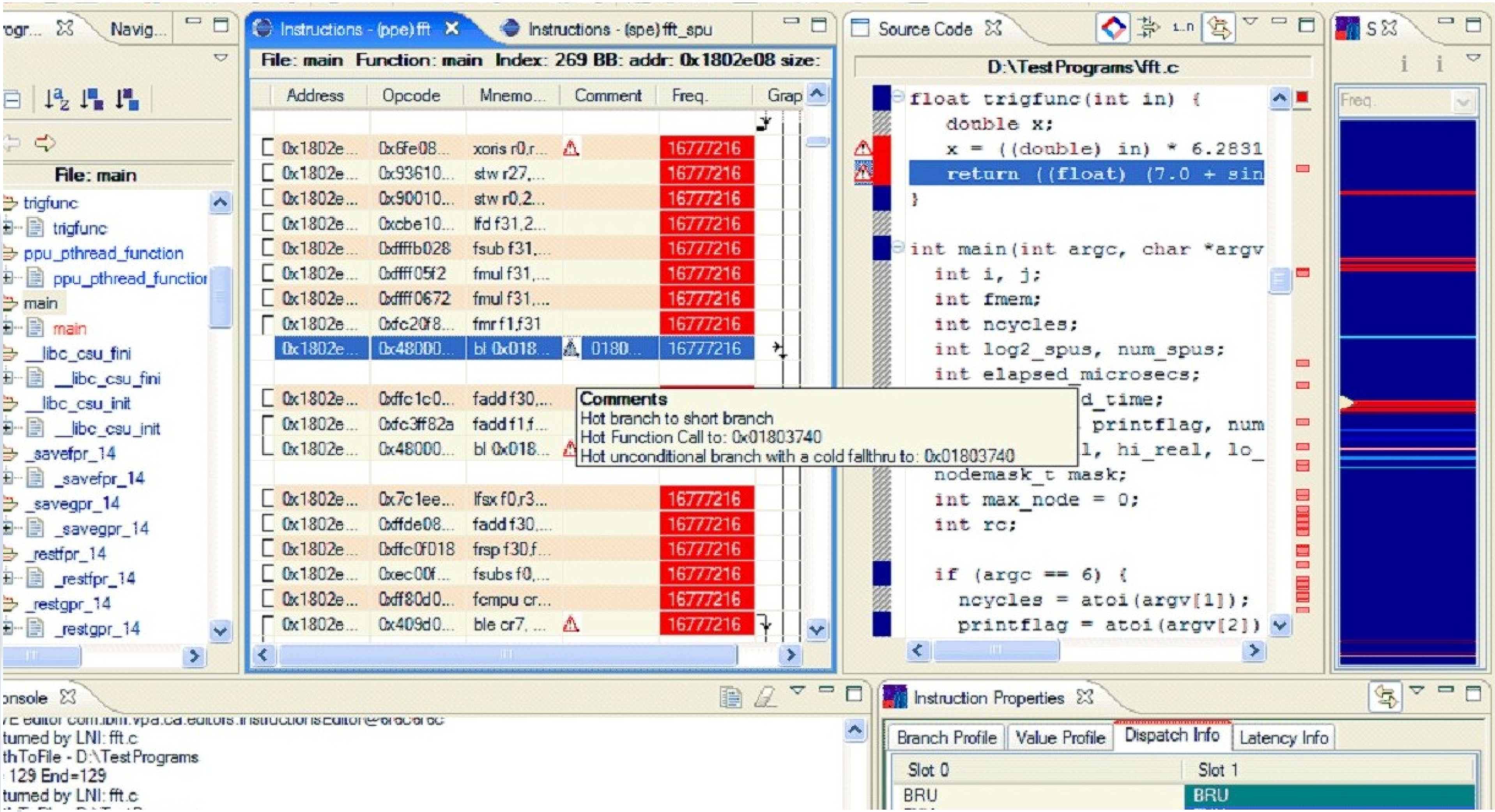
Top 50 comments that match the filter

	Heat	Description	File	Function	Address
⚠	105004	Hot Function Call to: 0x10003248	xlsubr.c	.xlevarg	0x1000712c
⚠	94788	Hot Function Call to: 0x10007240	xlsubr.c	.xlmatch	0x1000719c
⚠	397326	Hot TOC Load Instruction: 0x10002468	xleval.c	.xlsave	0x10002468
⚠	397326	Hot TOC Load Instruction: 0x1000248c	xleval.c	.xlsave	0x1000248c
⚠	715660	Hot TOC Load Instruction: 0x100024b0	xleval.c	.xlsave	0x100024b0
⚠	715660	Hot TOC Load Instruction: 0x100024b8	xleval.c	.xlsave	0x100024b8
⚠	715660	Hot TOC Load Instruction: 0x100024d8	xleval.c	.xlsave	0x100024d8
⚠	100680	Hot TOC Load Instruction: 0x10002c1c	xleval.c	.xlevlist	0x10002c1c
⚠	132574	Hot TOC Load Instruction: 0x10002e60	xleval.c	.evform	0x10002e60

Code Analyzer

Comments View

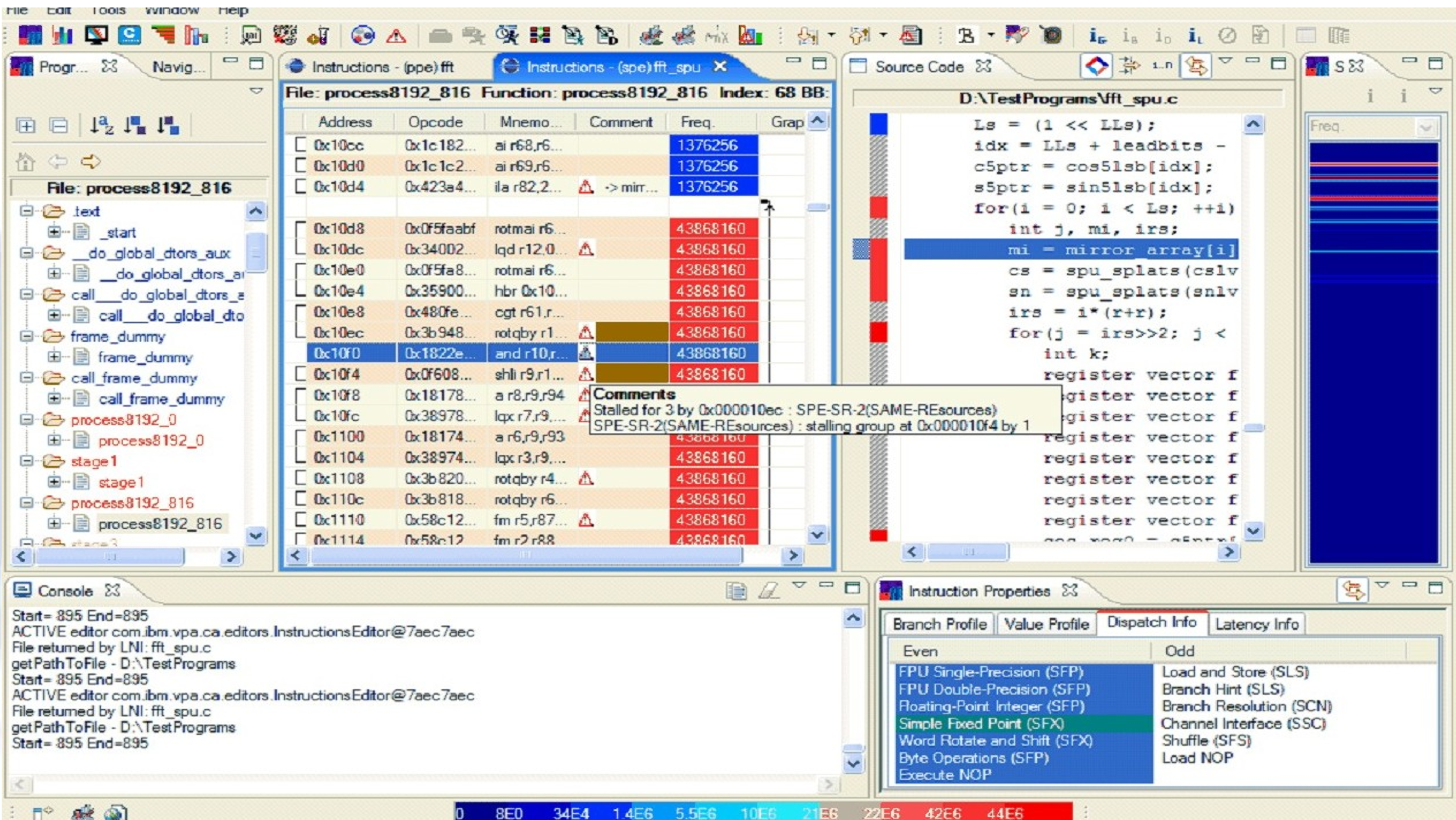
Cell PPU (Power Processor Unit) FFT code with performance comments:



Code Analyzer

Comments View

Cell SPU (Synergistic Processing Unit) Pipeline Stalls



Code Analyzer

Grouping – instructions grouped in Power5

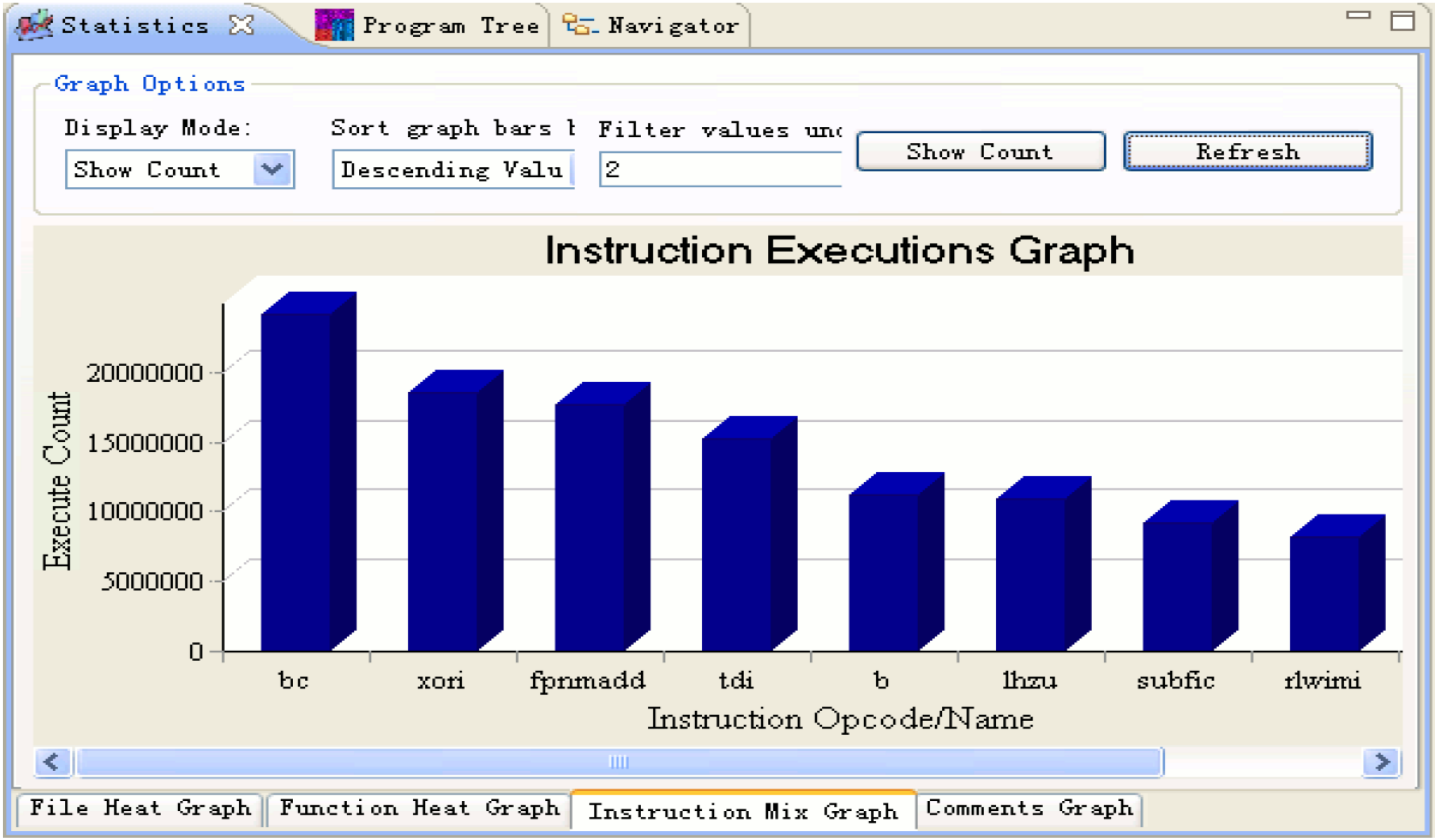
Instructions - (xcoff) li32

Edit | **File: xlisp.c Function: .main Index: 24 BB: addr: 0x1000033c size: 5(20) exec: 1**

Address	Bytes	Disassembly	Comment	Freq
0x1000033c	0x38600001	li r3,1		1
0x10000340	0x9061015c	stw r3,348(r1)		1
0x10000344	0x80810198	lwz r4,408(r1)		1
0x10000348	0x7c032000	cmp cr0,0x0,r3,r4		1
0x1000034c	0x40800050	bge cr0, 0x10000...		1
0x10000350	0x8061019c	lwz r3,412(r1)		1
0x10000354	0x8081015c	lwz r4,348(r1)		1
0x10000358	0x5484103a	rlwinm r4,r4,0x2,...		1
0x1000035c	0x7c63202e	lwzx r3,r3,r4		1
0x10000360	0x38800001	li r4,1		1
0x10000364	0x38a00000	li r5,0		1
0x10000368	0x48008c71	bl 0x10008fd8	.xlload	1
0x1000036c	0x60000000	ori r0,r0,0x0		0
0x10000370	0x2c030000	cmpi cr0,0x0,r3,0		0
0x10000374	0x40820010	bne cr0, 0x10000...		0

Code Analyzer

Graph View example



Code Analyzer

Cell example: inserting branch hint

	BASIC BLOCK			
0x6a0	0x340b0082	lqd r2,704(r1)		1000980
0x6a4	0x04000103	ori r3,r2,0		1000980
0x6a8	0x340a8082	lqd r2,672(r1)		1000980
0x6ac	0x4800c102	cgt r2,r2,r3		1000980
0x6b0	0x217fe782	brnz r2,0x5ec		1000980
	BASIC BLOCK			
0x6b4	0x33936782	lqr r2,0xa1f0		
0x6b8	0x3f810102	rotqbyi r2,r2,4		980
0x6bc	0x04000103	ori r3,r2,0		980

Comments
Insert branch hint instruction to frequently taken target

Branch hint bits in the Cell SPE are the equivalent of the branch prediction bits that are used on the PPE. However, with the lack of profiling information, the compiler cannot always determine if a hint bit is needed. Hint bits on the SPE processor usually have more impact on the performance with compare to the PPE processor, as there is no hardware in the SPE to support branch prediction.

BProber
http://www.alphaworks.ibm.com/tech/bprober

- Framework for binary level instrumentation
  - Profiling
  - Program monitoring
  - Program verification and coverage
  - Program patching
- No need for changing source code or recompile
- Supports very large programs, which may exceed 32MB of code
- Handles both 32-bit and 64-bit program files, compiled with aggressive optimization options, including profile-based and linker optimizations

BProber – Features

Enable user's "plug-in"

Built-in Code-Coverage

Built-in profiling

BProber – User’s “plug-in”

Enables the user to execute his own instrumentation code in designated locations

Specific address <INSTR_ADDR……>

Prolog/Epilog of a function <INSTR_PROC….. >

User’s instrumentation code (stub) is written in high level language and compiled into shared library

The shared library is linked to the executable

Call to the stub are inserted in the program

Overhead due to environment preservation before/after the call

Reducing overhead with gated instrumentation

Controlling/Reducing saved environment using specific flags

BProber
Enhancing User's "plug-in" Ability

Additional Directives

Compound directives on where to insert stubs

<ALL_BB ….>

<ALL_PROC ….>

…..

Enabling Gated instrumentation – limiting number of times the stub is calls and reducing overhead

<GATED_INSTR_....>

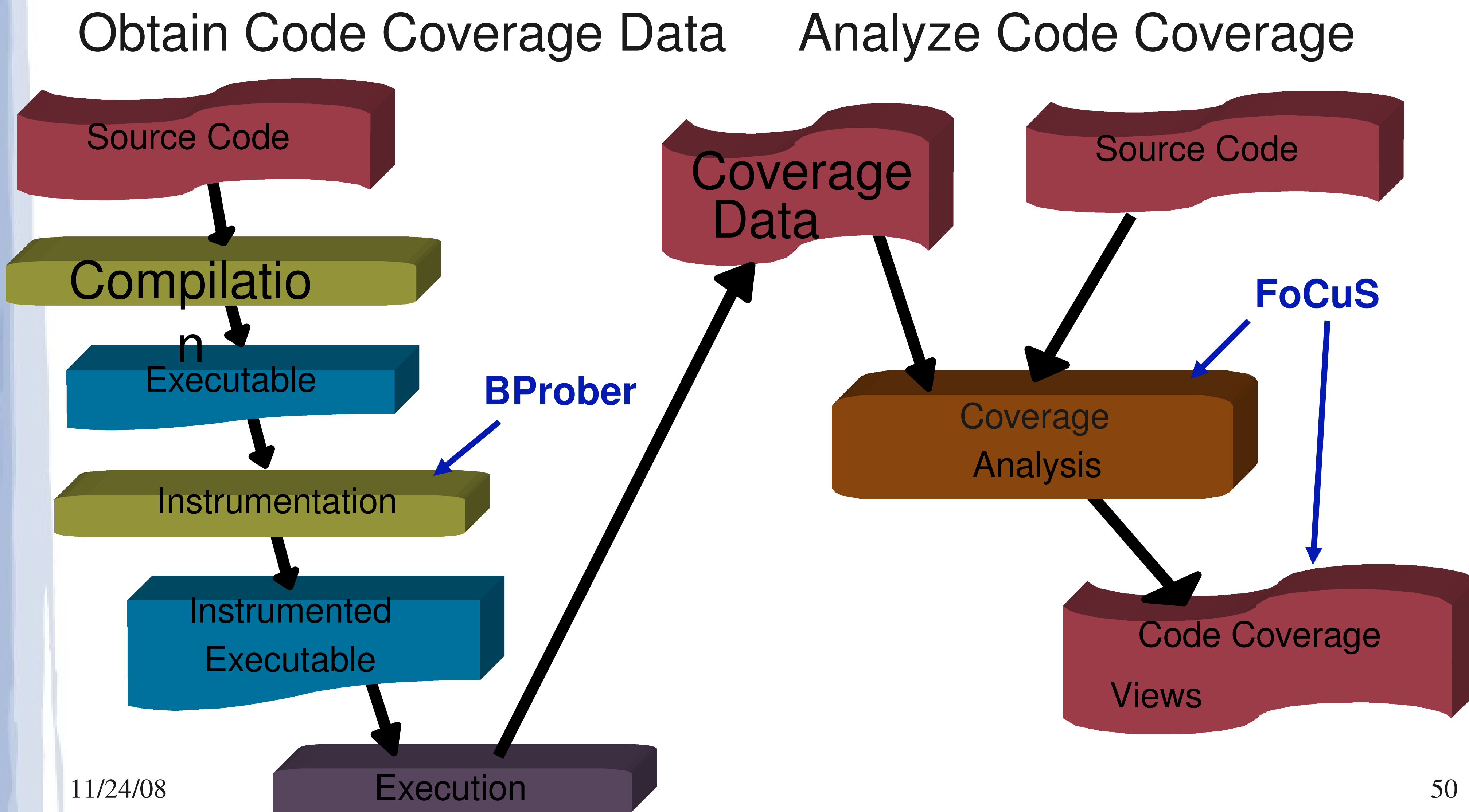
……

Predefined stubs

Performance Monitoring stubs for AIX

Tracing stubs (on the work)

BProber – Built-in Code Coverage



BProber – Built-in Code Coverage

Function level coverage or finer grain of basic block coverage

Map to source code (when debug information available)

Filtering of functions to reduce overhead

Customized coverage – fine grain BB coverage on specific functions

Enables very low (5%) overhead (experimental)

Using self modifying code

BProber

Example of the FoCuS coverage display

```
    memset(logmsg,0x20,sizeof(logmsg));                                     //@f5a
    sprintf(logmsg, "CCA v%s (%s) Started...", CCA_APP_VERS, CCA_APP_DATE);  //@f5a
    LOG_EVENT(LOG_INFO+LOG_USER, brief, reqID, logmsg);                      //@f5c//@n40c//@n74c
  #endif                                                                  //@n59a

    // printf("\n\nCCA started\n");
    for (j = 0; j <  MAXTHREADS; j++)                                     /*@n46c*/
    //   for (j = 0; j < 1; j++)                                          /*@n46c*/
    {
      pthread_join(thread_id[j],NULL);                                    /*@n46c*/
    }                                                                     /*@n46c*/

  #ifdef S390                                                             //@n59a
    memset(logmsg,0x20,sizeof(logmsg));                                   //@n60a
    sprintf(logmsg,"z-CCA Ending. \n");                                   //@n60a
    LOG_EVENT(LOG_INFO+LOG_USER,brief, reqID,logmsg);                              //@n60a//@n74c
    return 0;
  #else
    memset(logmsg,0x20,sizeof(logmsg));                                   //@n60a
    sprintf(logmsg,"CCA Ending. \n");                                     //@n60a
    LOG_EVENT(LOG_INFO+LOG_USER,brief, reqID,logmsg);                              //@n60a//@n74c
    return 0;
  #endif

  }


  /*--------------------------------------------------------------------------*/
  /*                              CCAM_STARTUP                                  */
  /*--------------------------------------------------------------------------*/
  /*                                                                            */
  /* Perform initialization required by the CCA Manager.  This includes the     */
```

Covered   partialy covered   not covered

BProber – Built-in profiling

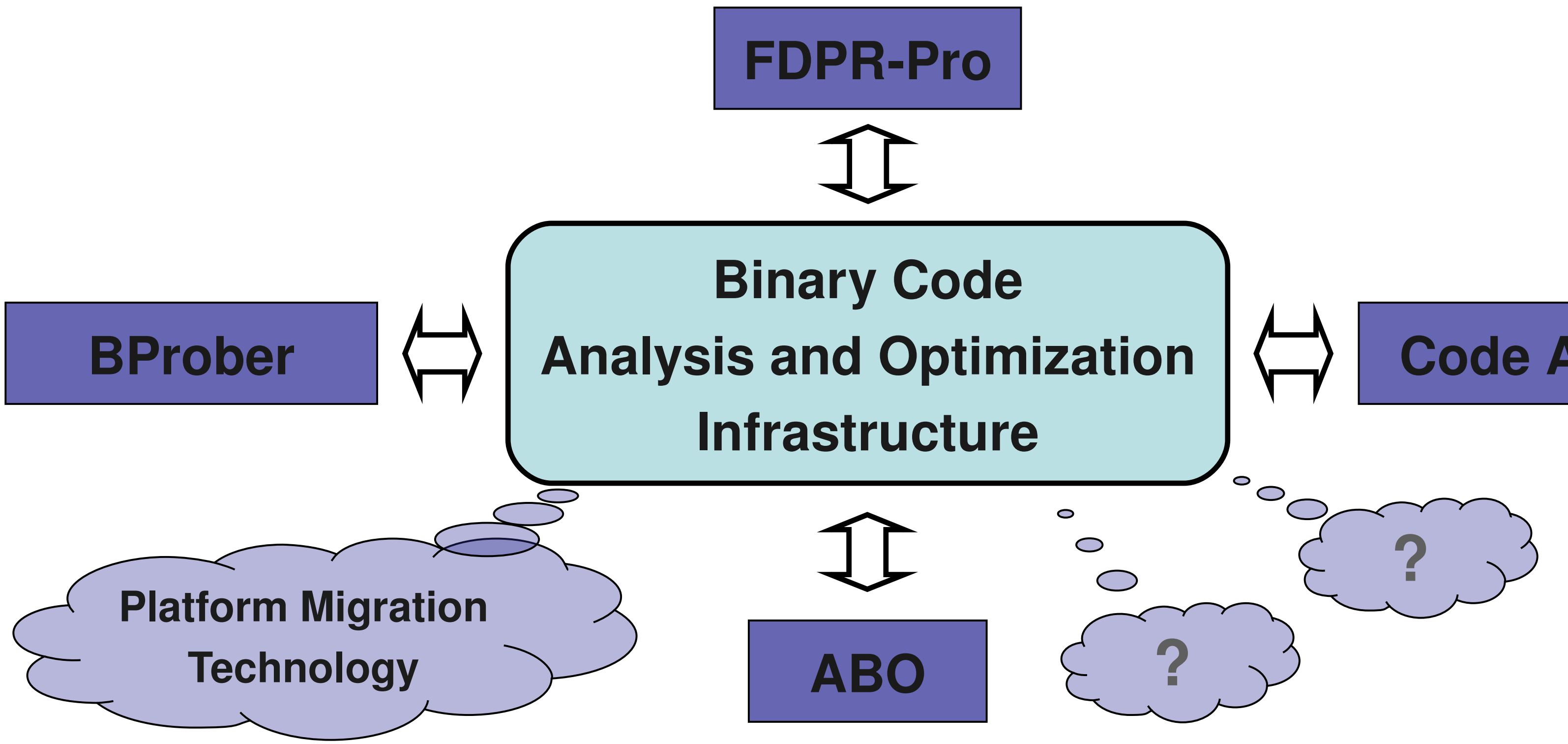
Edge Profiling at the Basic Block level

Register value profiling

Integrated display of profile with assembly code

Profile can be used in Code Analyzer for performance analysis and in FDPR-Pro for performance optimization

Overview, IBM's tools



Post-link optimizations examples
The Light Weight Approach

- Based on feedback information
- Requires only local information for each procedure
  - Immediate callers
  - Call sites
  - Immediate callees
- Scaleability
  - Short completion time
  - Single path
  - Simple data structures

Light Weight Optimizations

- Inter - Procedural Optimizations
  - Killed Register
- Intra - Procedural Optimizations
  - Non-used Caller-Saved Register

Killed Registers Optimization

```
     call foo
     ...
     call foo
     R27 <- 7
     ...
foo:
     save R27
     ...
     add R27,3
     ...
```

```
     call foo
     ...
     call foo_opt
     R27 <- 7
     ...
foo_opt:
     ...
     add R27,3

foo:
     save R27
     call foo_opt
     restore R27
```

Using Renaming to Enable it

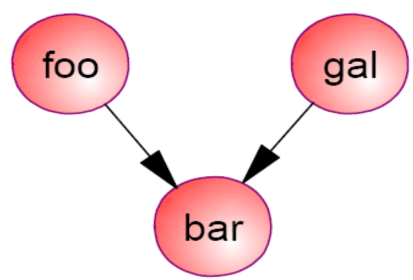
```
    call foo
    ...
    call foo
    R27 <- 7
    ...
foo:
    save R28
    ...
    add R28,3
    ...
```

```
    call foo
    ...
    call foo
    R27 <- 7
    ...
foo:
    save R27
    ...
    add R27,3
    ...
```

Reducing its code size

```
foo:
    ...
    call bar
    R29 <- 7
    ...
gal:
    ...
    call bar
    R28 <- 12
    ...
bar:
    save R28
    save R29
    ...
    restore R29
    restore R28
```

```
foo:
    ...
    store R28
    call bar_opt
    restore R28
    R29 <- 7
    ...
gal:
    ...
    store R29
    call bar_opt
    restore R29
    R28 <- 12
    ...
bar_opt:
    ...
bar:
    save R28
    save R29
    call bar_opt
    restore R29
    restore R28
```



Reducing the code size even more

```
foo:
        ...
        call bar
R29     R28 <- 7
        ...
gal:
        ...
        call bar
        R28 <- 12
        ...
bar:
        save R28
        save R29
        ...
        restore R29
        restore R28
```

→

```
foo:
        ...
        call bar_opt
        R28 <- 7
        ...
gal:
        ...
        call bar_opt
        R28 <- 12
        ...
bar_opt:
        save R29
        ...
        restore R29
bar:
        save R28
        call bar
        restore R28
```

Non-used Caller-Saved Register

Volatile Registers Optimization

```
foo:
      save R29
      save R30
      save LR
      ...
      call bar
      ...
      add R29,34
      ...
```

→

```
foo:
      save R30
      ...
      save LR
      save R4
      call bar
      restore R4
      restore LR
      ...
      add R4,34
      ...
```

* R4 is not used in foo
* foo contains only cold calls

Function Inlining

Pros:

- Instructions reduction in execution path
- Additional optimization opportunities after inline (Constant propagation, Scheduling...)
- Reducing branch penaltiesCall
  - On return – indirect branch, requires target prediction
  - Call

Pros:
New potential after inlining

- Plain Inlining is currently one of the most significant optimizations in FDPR-Pro
- Gives potential:
  - Copy+Constant propagation potential
    - parameter passing and return value (will also reduce register pressure).
    - A weighted mean of around 20% (tested on some selected benchmarks) of the parameters passed to function are either constants or copied registers.
  - Code motion from callee to caller or vise versa
    - Shrink wrapping, partial redundancy elimination, loop invariant code motion
    - Code can be moved, usually from the hot inlined function to the caller which is usually colder
      - for example a loop calling an inlined function
  - Dead code elimination
  - Register Reallocation
  - …

Function Inlining

Cons:

- Increasing code size
  - Physical limitation (embedded systems)
  - Duplication of hot code that can increase cache conflicts

heuristics for Inlining

- Size (small function, inlined traces fits to L1 line)
- Single/dominate call
- Path Based Selective Inlining (ILB: ICache Loop Blocking)
  - For more info see: Aggressive Function Inlining: Preventing Loop Blockings in the Instruction Cache

Synergy of Code Reordering and Inlining

Code reordering rearranges basic blocks in consecutive hot chains, removing part of the Icache conflicts caused by aggressive inlining

- Relocating inlined cold code

Function inlining creates better opportunities for code reordering by extending its scope across function calls

- Enables to have larger traces of BB

Inlining Performance Result

Comparing 4 inline methods with ILB:

all - all executed functions that were somewhat hot

hot - all functions that are above the average heat

dominant - call that execute than 80% of calls to the function.

small - only small size functions

Implemented with IBM FDPR-Pro - a postlink optimization tool.

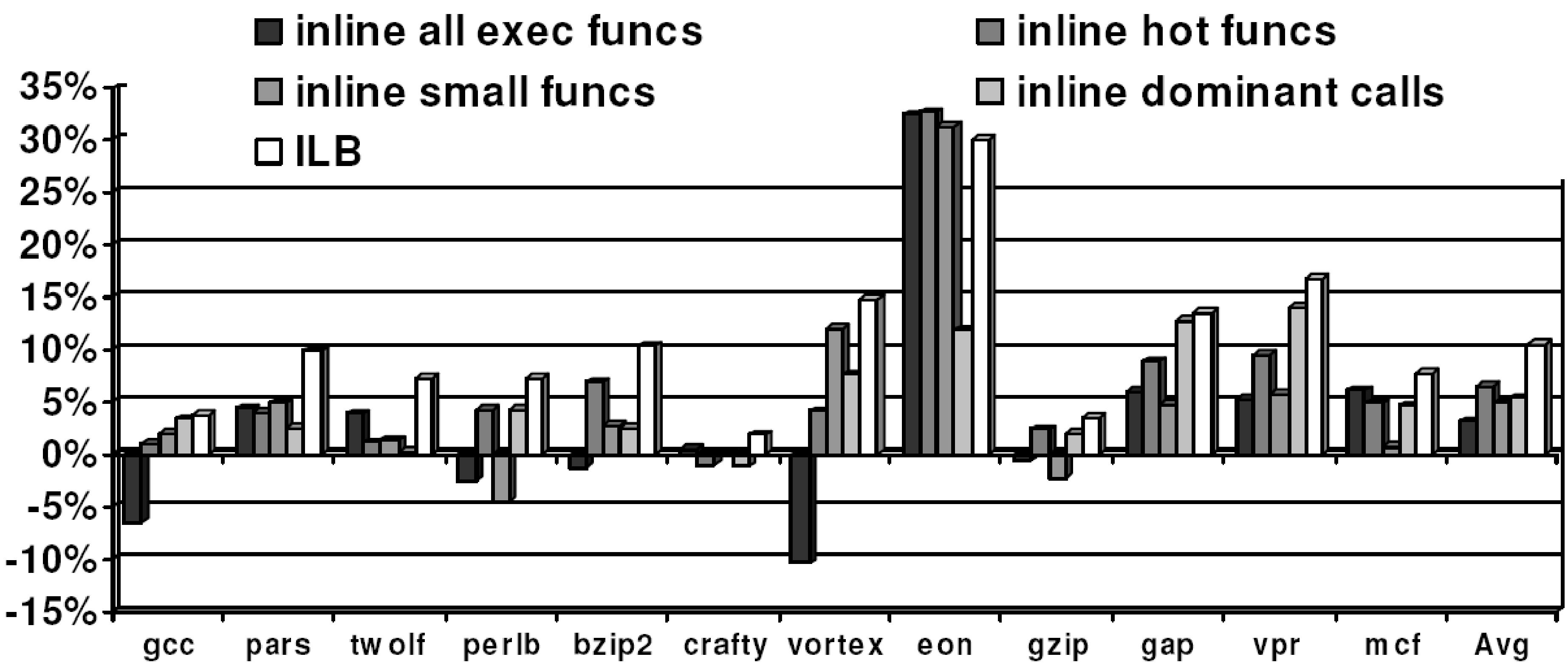
SPEC CINT2000 using train profile and ref measurments

Hardware

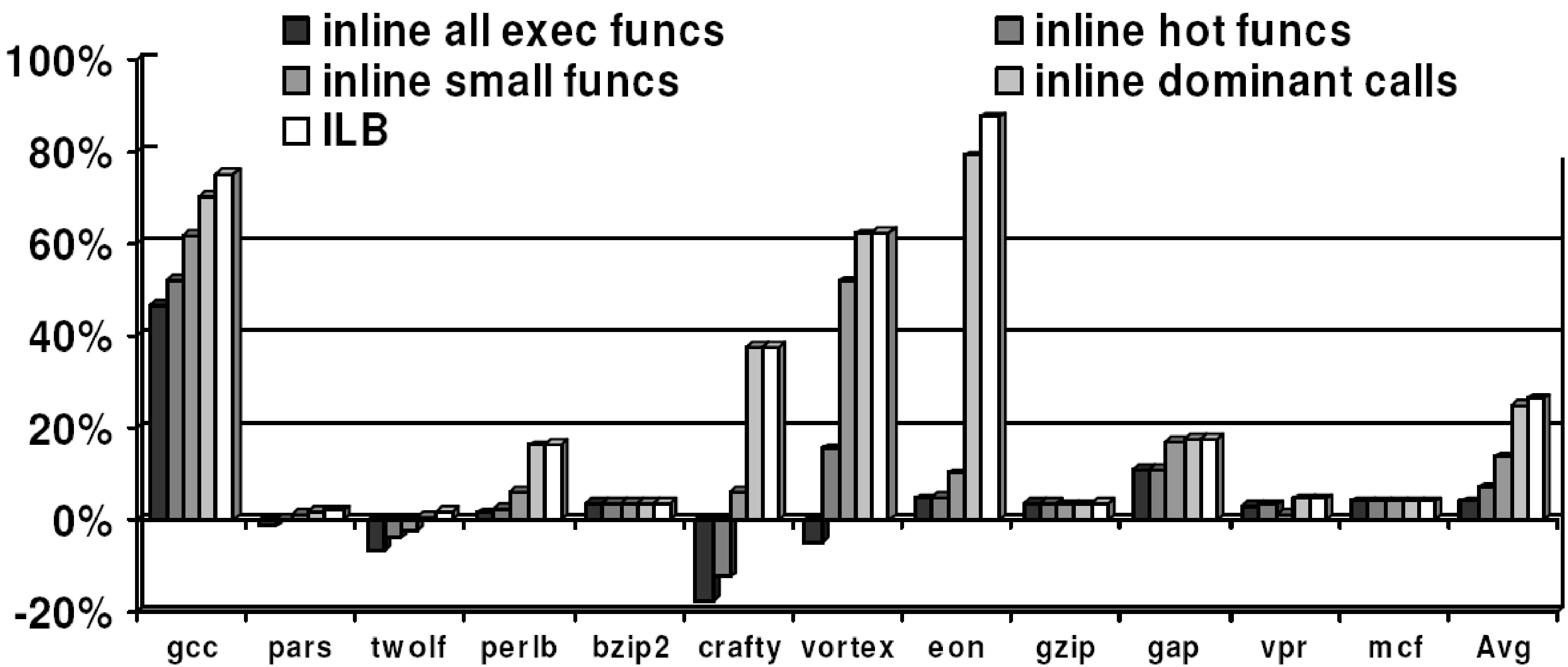
IBM Power4

AMCC 440GX

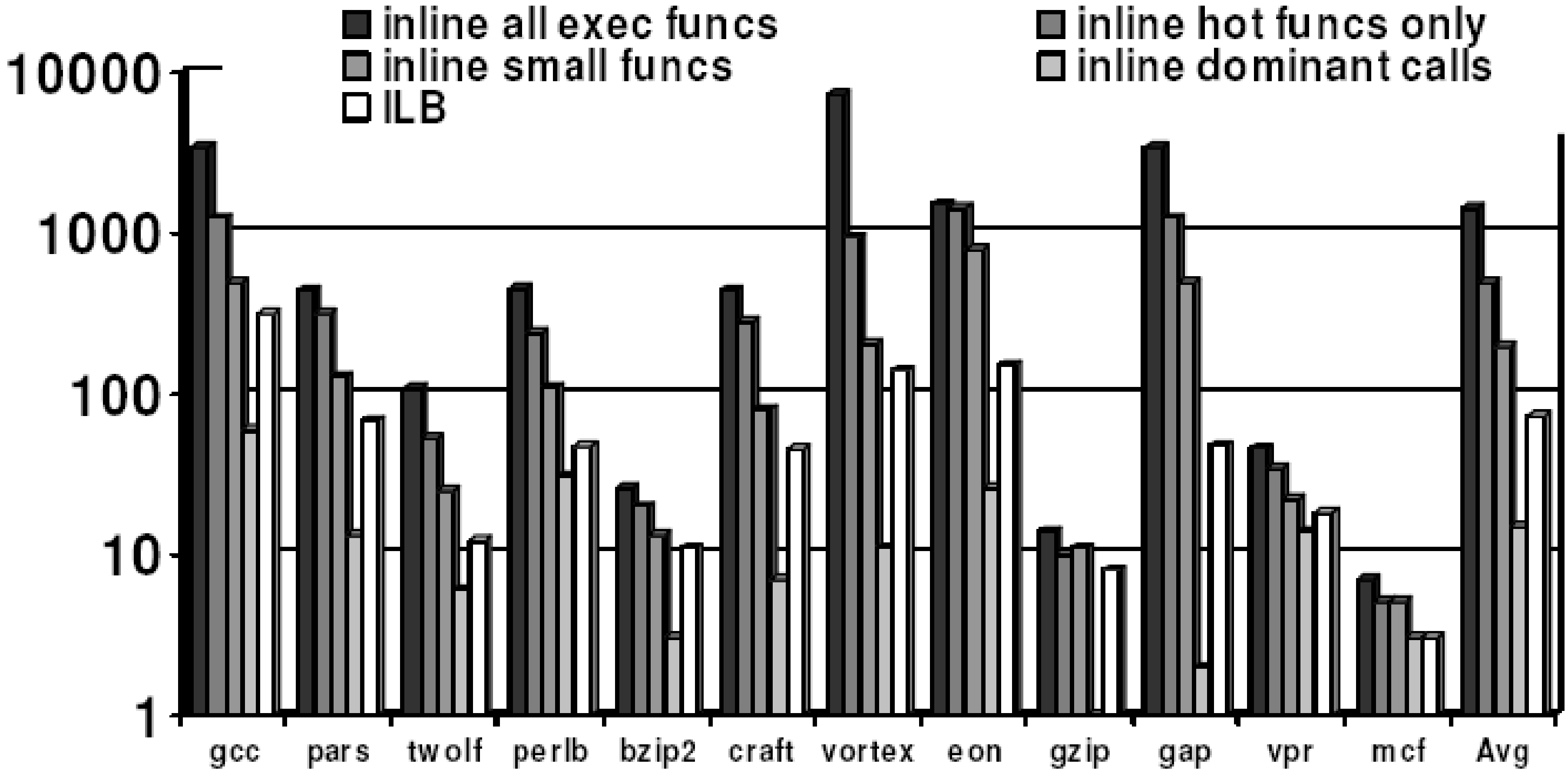
Performance Results – Power4



Performance Results – 440GX



Number of Inlined Functions



Summary

Post-Link Optimizations can give us huge performance gains

Post-Link Analysis gives us much more anylzing options and permits us to investigate, among others, linker code and compiler-optimized code

F/OSS is still lacking on this front, although valgrind and the SOLAR project sounds promising

Special Thanks

This Talk was made possible by the material, slides, optimization-implementation and guidance of IBM's PAOT Team, thanks everyone :-)

I'd like to give thanks to following people for their help in preparing these slides (in no particular order):

- Omer Boehm
- Gad Haber
- Moshe Klausner
- Marcel Zalmanovici