
Implementing an Experimental VLIW Compiler

M. Moudgill
IBM T.J. Watson Research Center

The VLIW gang (over time)

Kemal Ebcioglu

Erik Altman

Shyh-Kwei Chen

Norm Cohen

Dick Goldberg

Brian Hall

Rene Miranda

Jaime Moreno

Peter Oden

Arkady Polyak

Balaram Sinharoy

Bryce Cogswell

Induprakas Kodukula

Vladimir Kotlyar

No compilers need apply...

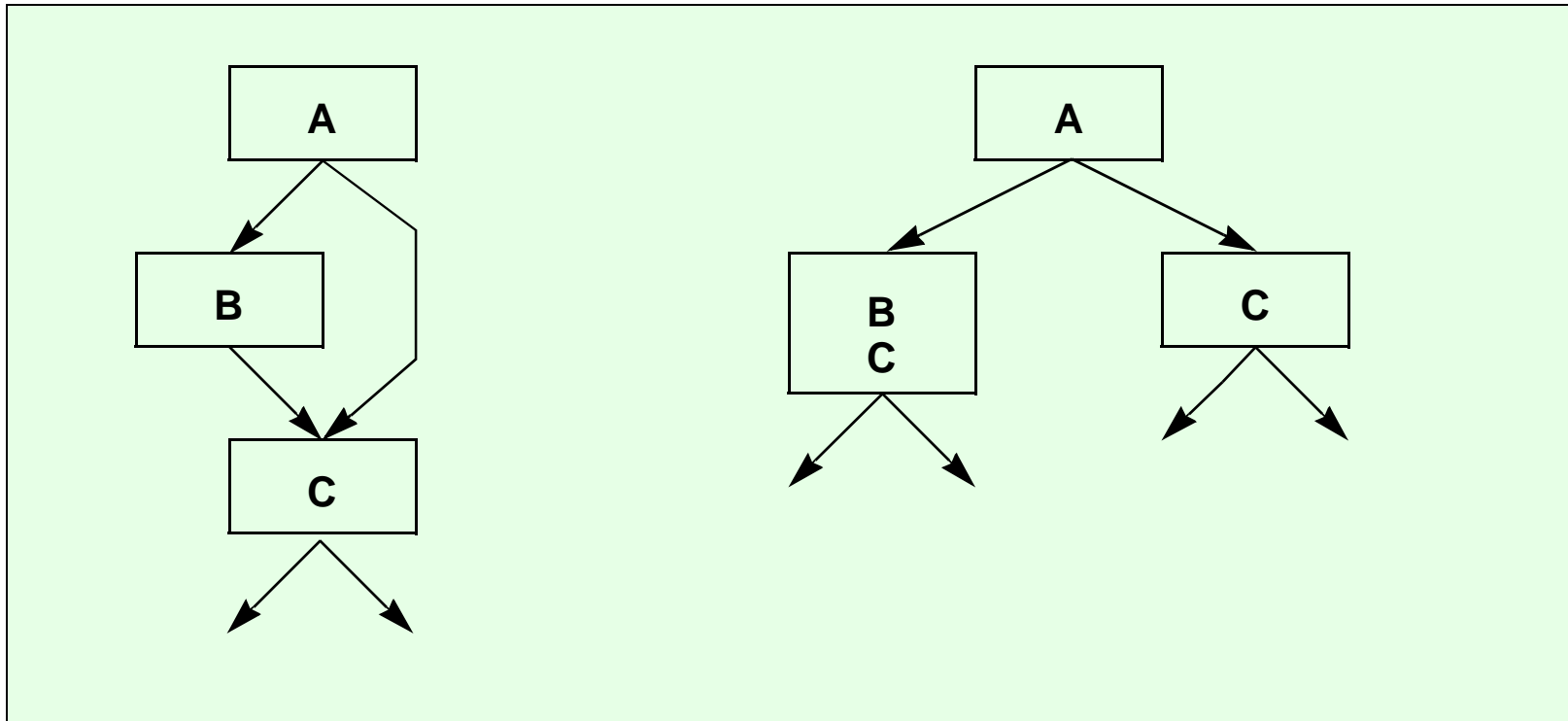
Branch prediction

- Table sizes
- 2bit/1bit Counters
- gshare
- gselect
- ...

But ... existing traces only!

A compiler could change the numbers....

“super-blocking”



With new instructions (and a compiler) ...

Branch elimination

```
    branch pred, C
B:   r3 = r4 + 1
C:   ...
```

- **Conditional-Move**

```
    r13 = r4 + 1
    r3 = cmove pred, r13
C:   ...
```

- **Predicated Execution**

```
    r3 = r4 + 1 if pred
C:   ...
```

- **“Idioms”**

The processor recognizes:

```
    branch pred, PC+2
```

as predicating next instruction

The ultimate in compiler/architecture co-dependence

VLIW

- Issues many instructions per cycle
- More bits per “instruction” word
 - Larger register files
 - More instruction types
- Shorter cycle time

Great *potential* performance

But ... needs a “magic” compiler!

Requirements for an ILP architecture sandbox

Compiler

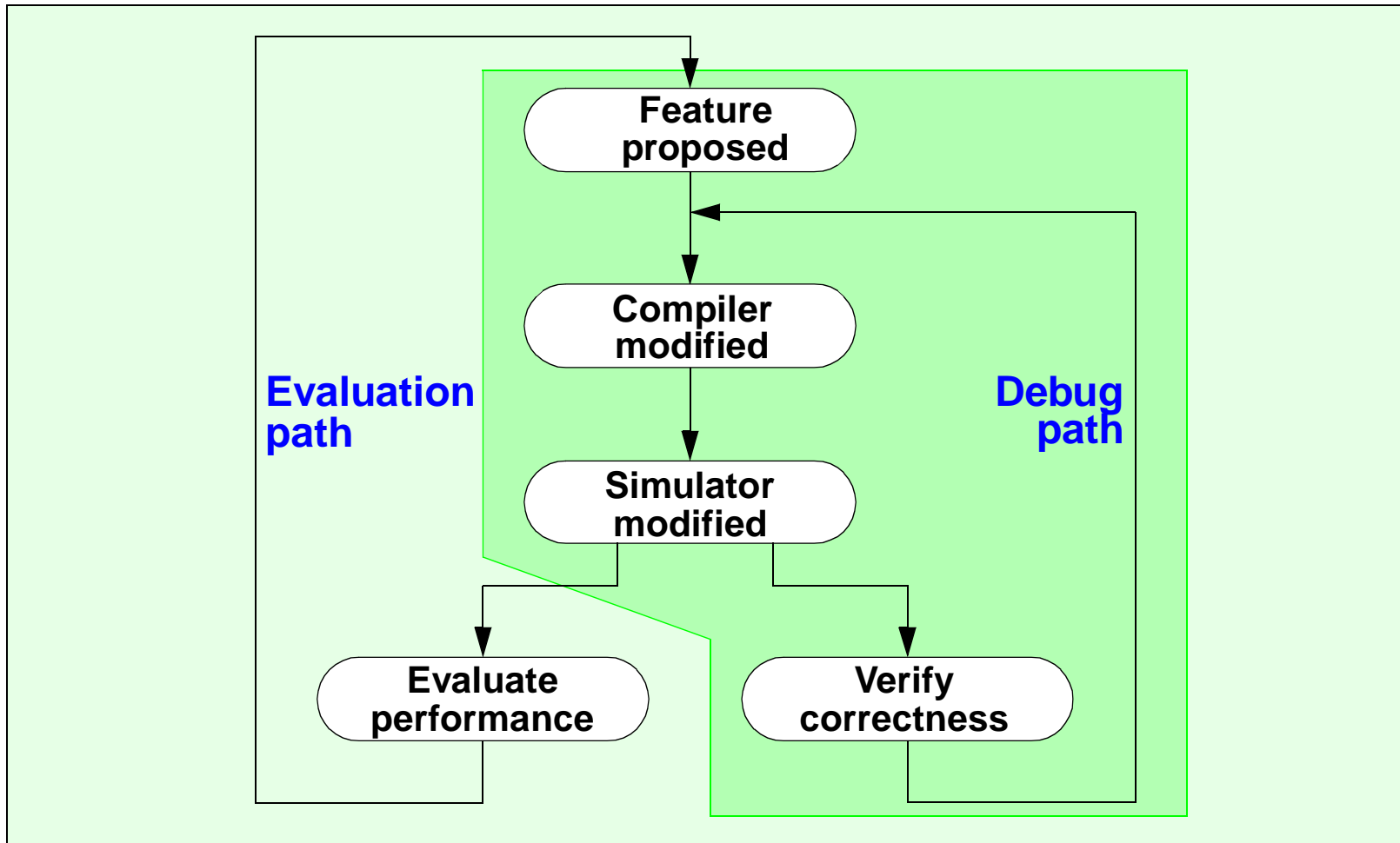
- Rapidly write and debug cutting-edge optimizations
- Painlessly support changes in the architecture

Simulator

- Support changes in the architecture
- Execute target architecture code rapidly

Iterative simulation/evaluation process

Built *around* the architecture/compiler interaction



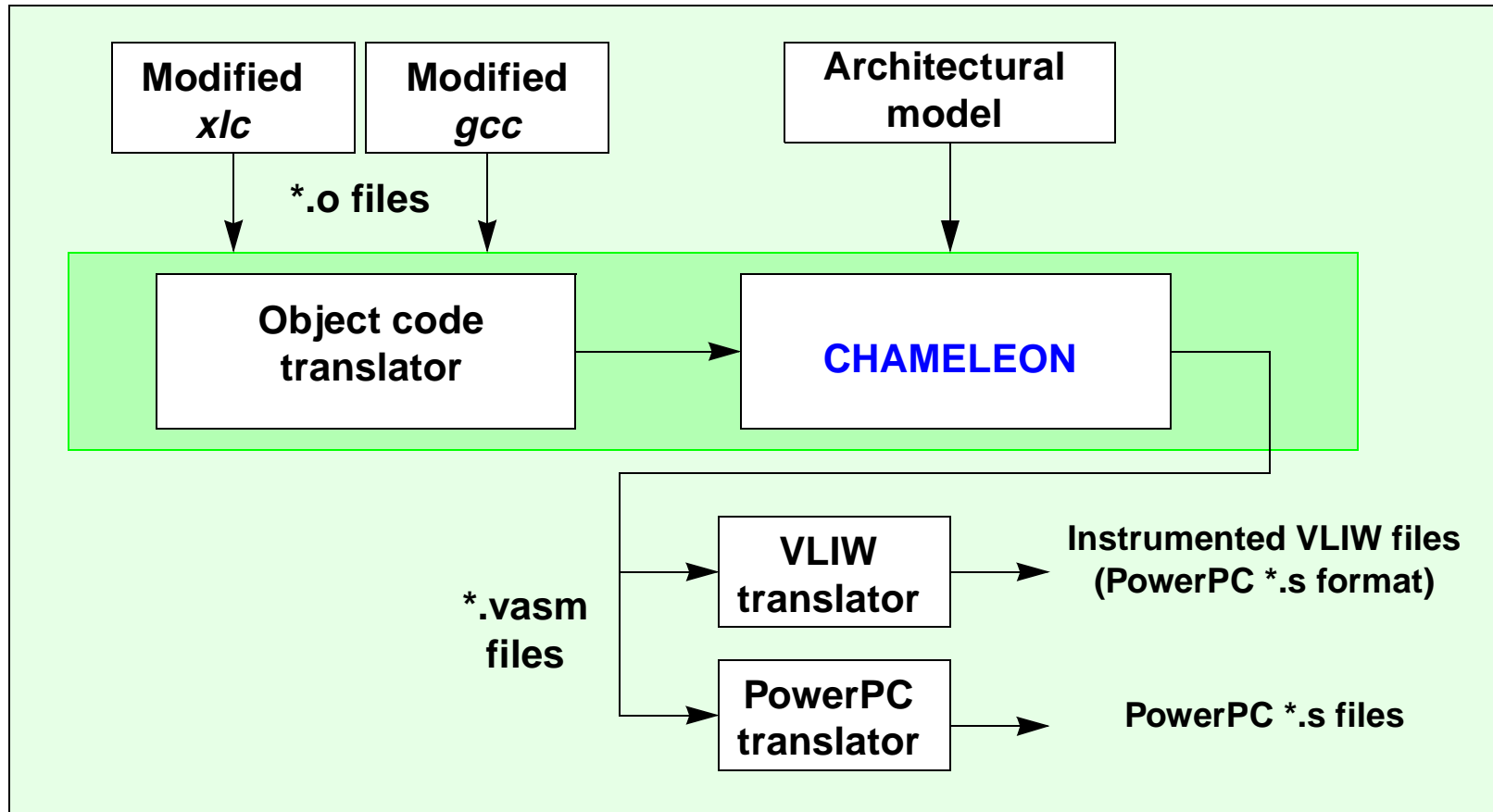
Problems with existing compilers...

- Internal data-structures make writing global optimizations difficult
- Internal data-structures make writing dependence-based optimizations difficult
- Difficult to modify without introducing unrelated bugs
- Existing optimizations geared towards low-ILP machines
- Difficult to support different architectures
- Availability

The CHAMELEON compiler

Designed to support ILP research and evaluate trade-offs

Extensively parameterized (architectural model)



Modified xlc/gcc: make object code translation easier

Implementation of CHAMELEON

Based on dependence-flow graph DFG [Pingali et al. 91, Johnson 94]

- integrated data dependences/control flow information
- uniform representation of memory (flow, anti, output) dependences
- SSA/reverse-SSA
- executable representation
- checkable

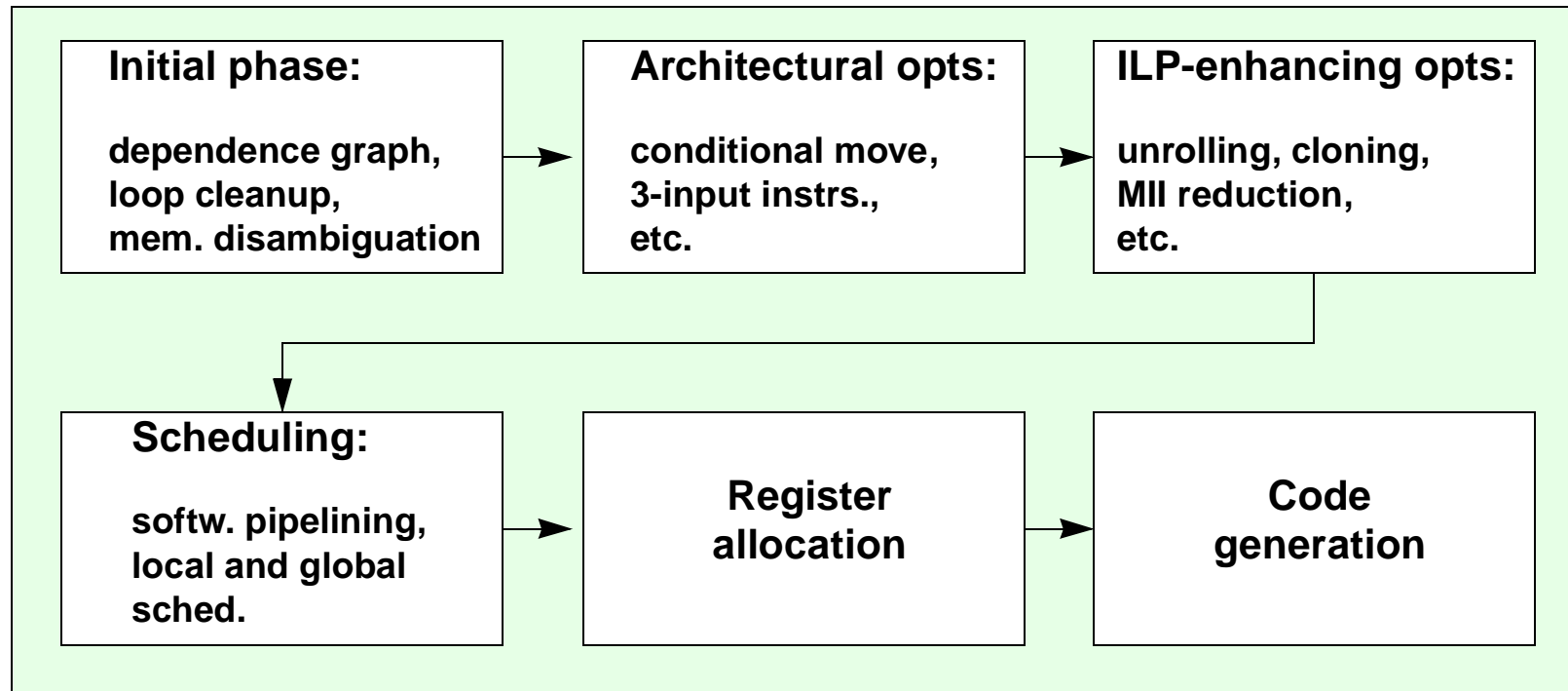
Optimizations

- stand-alone
- table/property-driven
- global analysis
- incremental transformation

Extensive debugging support

Optimizations in CHAMELEON

Traditional (applied throughout the compilation process)



Variant of Ball-Larus heuristics (no profile-directed feedback)
No interprocedural analysis or inlining

Phases in CHAMELEON

General

- Dead code elimination
- Synthetic frequency
- Memory alias analysis
- Interval/SESE analysis

Traditional

- Constant propagation
- Redundant if
- Reassociation
- Load/store elimination

Loop

- Unrolling
- Peeling
- Do-Loop unrolling
- Invariant/counter
- MII-reduction

Architecture

- Cache prefetch
- Load/verify
- Conditional-move
- 3-1 instructions

Scheduling

- Multi-path software pipelining
- Tree scheduling
- Spill scheduling

Register allocation

Peephole compaction

Architectural modifications

Resource model: specified at run-time

- Register file size
- Functional-unit number and mix
- Issue restrictions

Gross architectural changes: requires rebuild

- Instructions
- Register file types

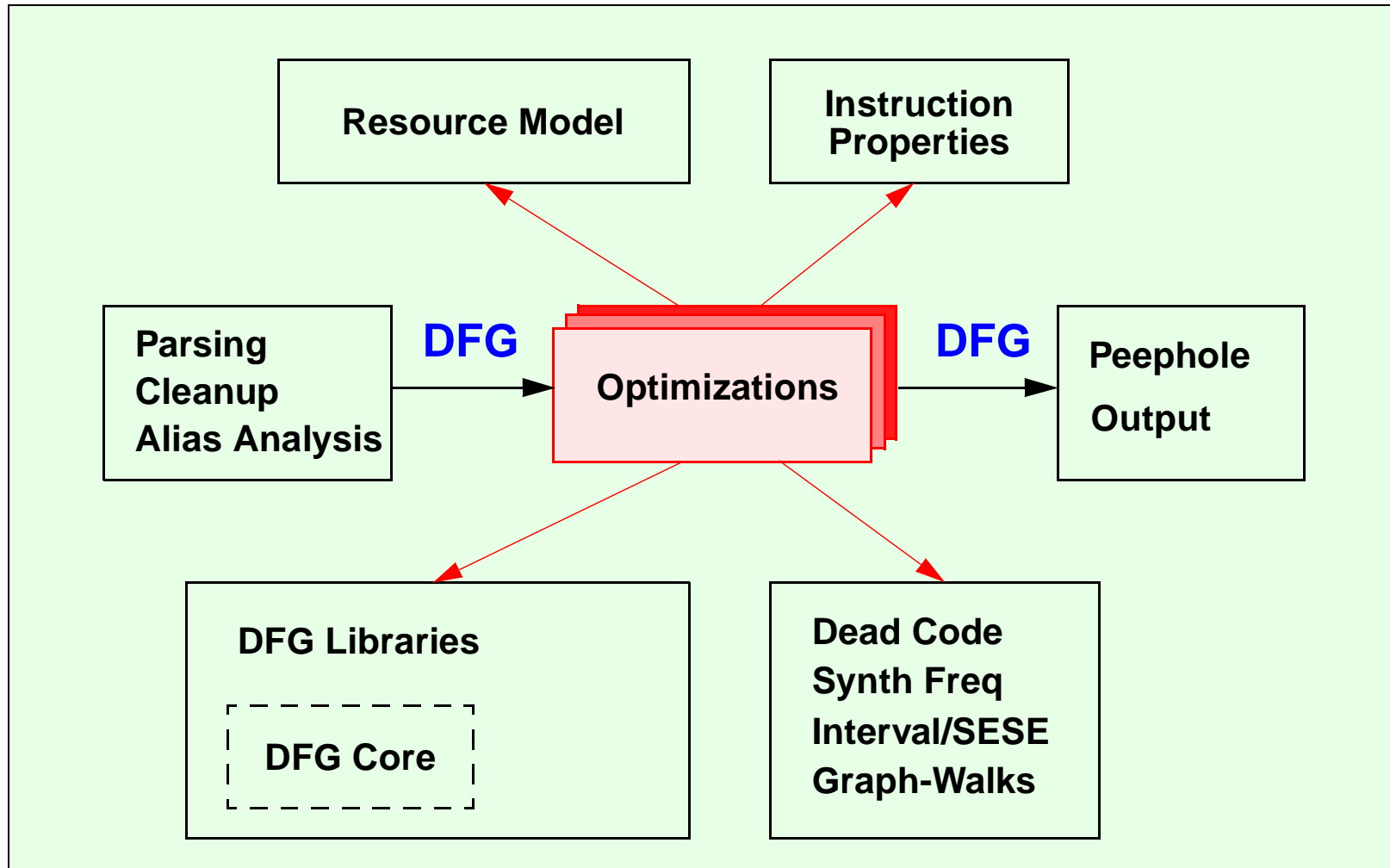
Example: adding instructions

- Update `Opcode.table` entry
- Add constant evaluation routine (if needed)
- Parsing/phases/output accept new instruction
- Some optimizations can take advantage of new instructions

Table entry for instruction “Add”

```
op      ADD
in      GPR GPR_or_CST
out     GPR
eval    add_eval_opcode
assoc
commute
add
ppc_record
@end
```

Writing Optimizations



Optimization highlights

Use information readily available

- Data dependence
- Integrated control/data dependences
- Interval and/or SESE
- Alias

Architectural abstraction

- Resource model
- Instruction properties

Abstract interpretation

- Consequence of executable DFG

Transformation

- DFG to DFG
- Transformation libraries (eg, instruction speculation, loop-unrolling).
- Incremental transformation

A trivial example...

Swap inputs to commutative operation if first input is constant

```
FOR_BLOCKS_FUNC(func, block)
  FOR_OPS_BLOCK(block, op)
    if( is_commute_op(op) &&
        is_const_use( use_op(op, 0) ) ) {
      use0 = use_op(op, 0);
      use1 = use_op(op, 1);
      swap_use2use( use0, op, use1, op );
    }
  END_OPS_BLOCK
END_BLOCKS_FUNC
```

DFG Core

DFG Library

Property Table

CHAMELEON makes writing optimizations easy...

Unordered

- Optimizations are decoupled (i.e. stand-alone)
- Optimizations are permutable

Analysis

- Advanced information (e.g. data-dependence) available
- Global information (e.g. function-wide/interval-wide)
- Abstract-interpretation based
- Dependence-pattern based

Transformation

- Support library for most idioms
- One major data-structure to be manipulated

CHAMELEON makes debugging easy ...

Automatic checking

- Output of transformation is DFG, hence can be validated.

Inconsistency

- Only one major data-structure, no room for inconsistent state.

Isolation

- Optimizations can be turned off, since decoupled.
- Transformations in an optimization can be turned off, since incremental.

CHAMELEON's good points...

Comes with cutting-edge optimizations

Can be used as-is to study a variety of architectural issues

Easy to add architectural modifications

Easy to write cutting-edge optimizations

Easy to debug optimizations

CHAMELEON: resource hog....

Memory

as much as 256M (reload.c in gcc)

- consequence of extensive dependence information
- consequence of whole function optimization
- 20-30% because of implementation mistakes

CPU

minutes - hours CPU time

- consequence of whole function optimizations
- combined with $O(n^2)$ algorithms
- low priority

Missing features

Computed-goto/switch/jump-tables

- Currently implementes using ifs
- Support for jump-tables in form of multi-way branches
- Not used, not tested, will probably break

Multiple entry points

- FORTRAN ENTRY statement needs to be implemented indirectly
- Similarly for exception handling in C++

Learning curve

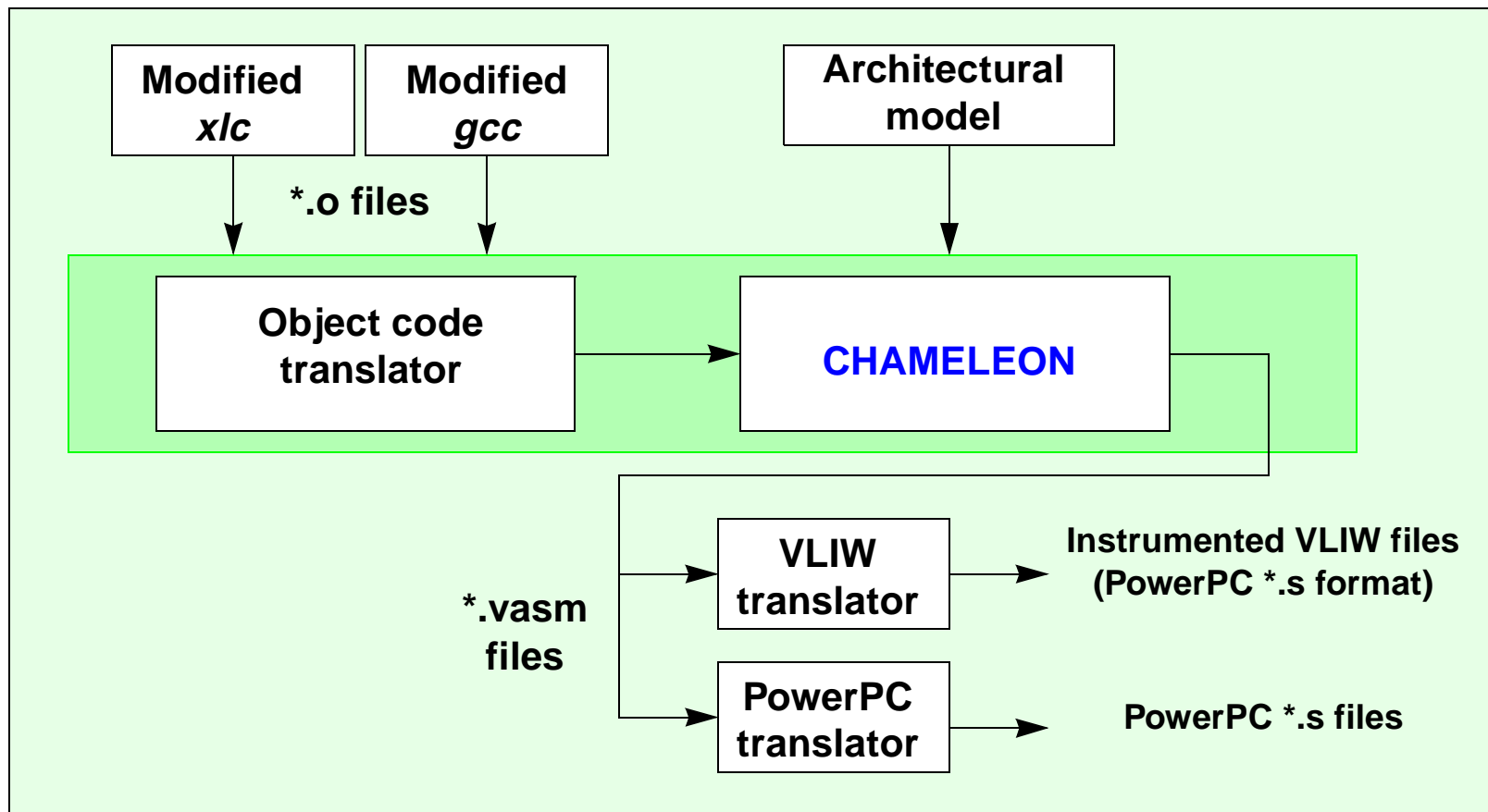
Advanced knowledge base required

- Data dependence
- DFG
- SSA/reverse SSA
- Abstract interpretation

Documentation under development

CHAMELEON: AIX/PowerPC centric

- object-code translator reads xcoff format
- back-ends produce POWER/PowerPC assembler
- back-ends assume AIX calling conventions
- source code written against AIX libraries



Current status: stability

xlc/VLIW

- SPECint95
- SPECint92
- assorted utilities
- xlc test suite

xlc/PowerPC

- SPECint92
- assorted utilities
- xlc test suite

gcc/PowerPC

- some SPECint92
- assorted utilities
- xlc test suite

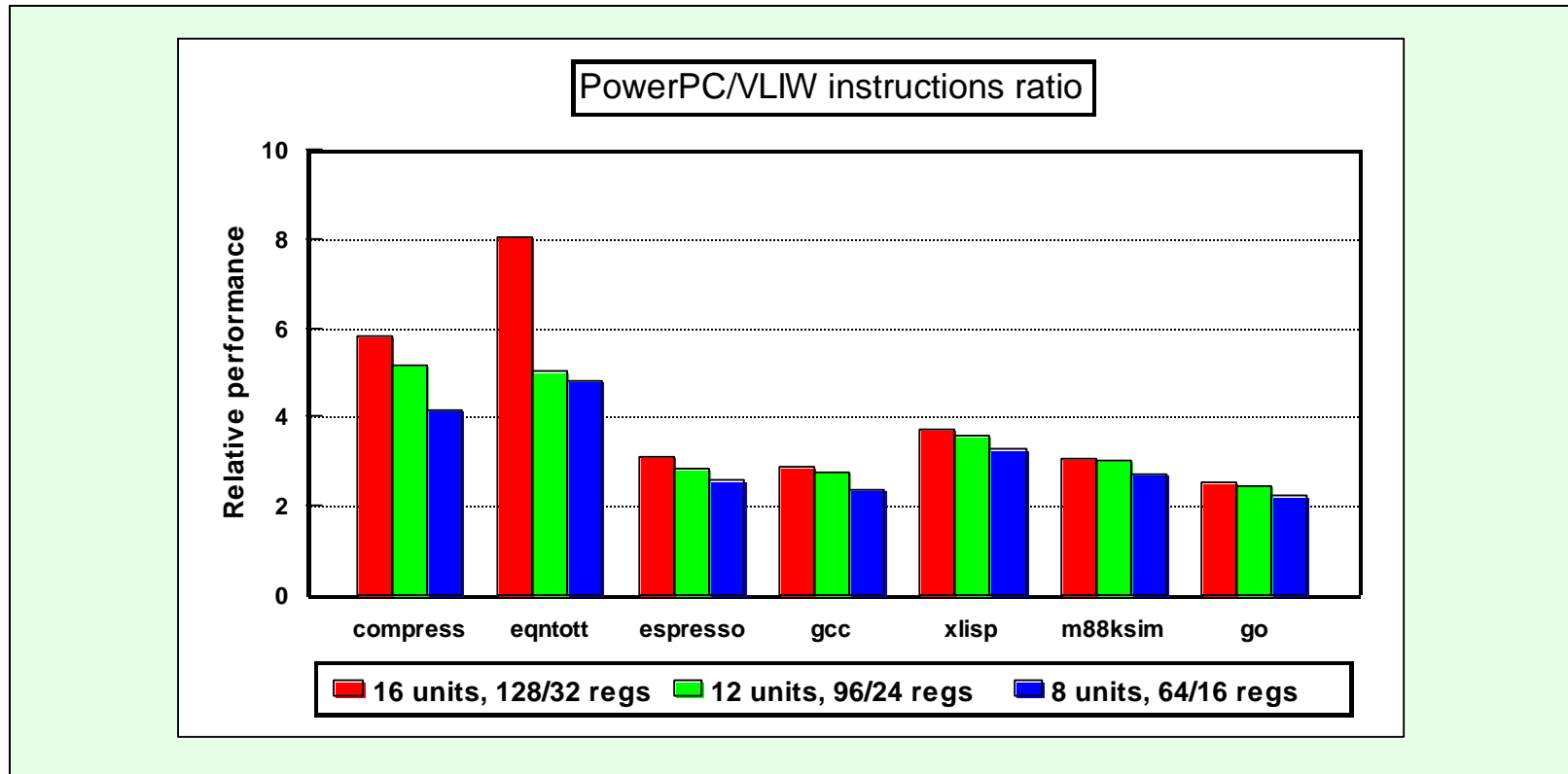
gcc/VLIW

- some SPECint92
- assorted utilities
- xlc test suite

Current status: performance

Instruction-level parallelism, larger register set

Compiler invoked with parameters file describing target architecture



ILP computed with respect to sequential code optimized with *x/c*

Future directions

Short term

- Stabilize gcc/VLIW and gcc/PowerPC
- Complete documentation

Longer term

- Add support for sub-function sized code regions
 - Fix compile-time
 - Decrease memory usage
- Stabilize multi-way branches
- Fix memory-consuming representation mistakes

Summary

CHAMELEON is an optimizing ILP compiler

- designed to support experimentation
 - architecture
 - compiler optimizations

Available to members of the academic community

Obtaining CHAMELEON

Contact

- **Mayan Moudgill** (mayan@watson.ibm.com)
- **Kemal Ebcioğlu** (kemal@watson.ibm.com)

What you get

- **Source code**
- **gcc only (no xlc front-end)**
- **non-commercial use only**