Eliminating Dynamic Computation Redundancy Using An Integrated Architecture and Compilation Framework

Daniel A. Connors

IMPACT Compiler Research Group

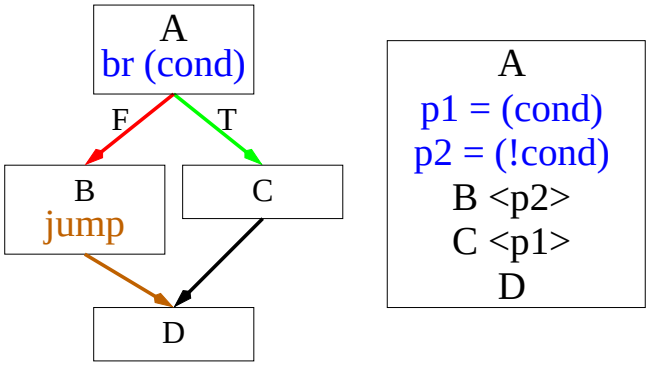
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Instruction-Level Parallelism Overview

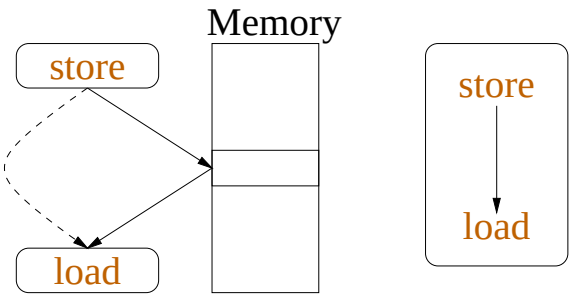
- Instruction-Level Parallelism (ILP) is a cost effective way to extract performance from programs.
  - ILP concurrently executes independent instructions.
  - Expect future processors to execute 8 to 12 instruction per cycle. (i.e. IA-64 (Merced), Alpha 21364)
- Processors are dependent on the ability of compilers to expose ILP.
  - Exposing large amounts of ILP requires advanced global analysis and optimization.
  - Current state-of-the-art compilers cannot expose this level of ILP.
  - Diminishing performance returns in using available silicon for ILP.
- Ultimately performance becomes limited by dependences of programs, not machine resources in ILP paradigm.
- Growing availability of silicon resources.

Dependences Limit the Potential ILP

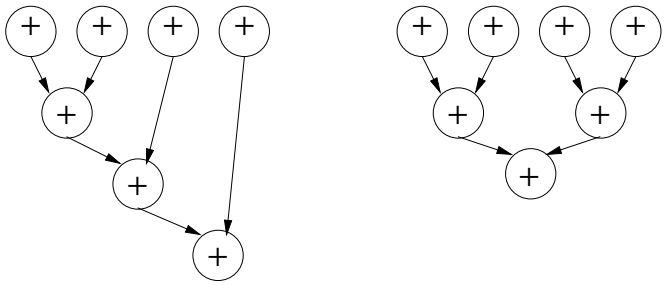
Control Dependences



Memory Dependences



Data Dependences



- Dataflow limits ILP by imposing serialization constraints on operations.

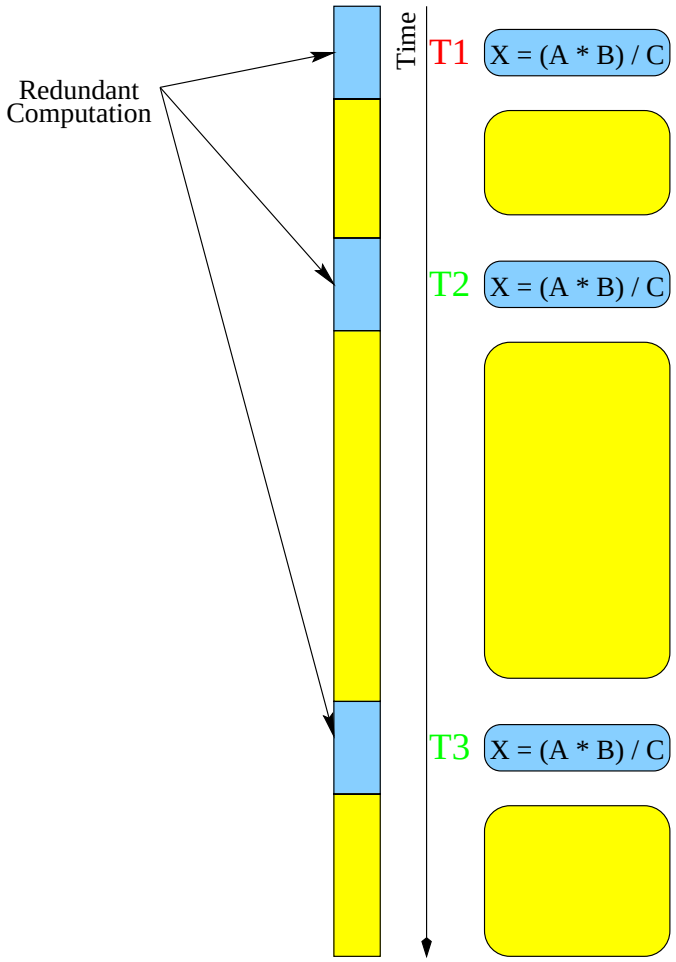
Compilation/Software Distribution Trends

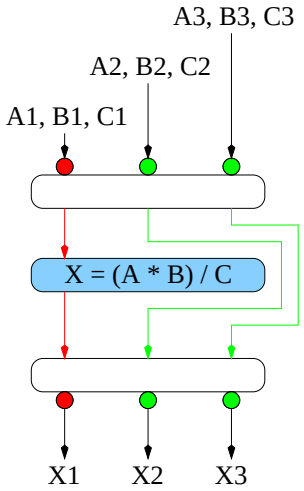
- Future models of software distribution.
  - Dynamically Linked Library (DLL) model.
  - Downloadable updates.
  - Downloadable net applications.
- Barriers to traditional compilation.
  - Invariant or semi-invariant nature of software.
  - Ultimately performance is sacrificed for maintainability.
- Domains of applications with run-time constants and semi-invariants.
  - Interpreters (program being interpreted is run-time constant).
  - Simulators (circuit or architecture is run-time constant).
  - Graphics processing (scene or viewing parameters are semi-invariant).

New Design Challenges

- Challenges:
  - Increase performance by overcoming dataflow limitation.
  - Achieve high performance in the presence of barriers to optimization.
  - Effectively allocate silicon technology.
- 40% of executed statements are dynamically invariant. [Lipasti & Shen]
- Reuse previous computation results.
  - Dataflow limit can be surpassed for sequences of operations otherwise redundantly executed.
- Two forms of redundancy
  - Static redundancy: repetition with the complete assertion that the computations are redundant on all executions.
  - Dynamic redundancy: repetition occurring over a temporal set of computation definitions.

Dynamically Redundant Computation

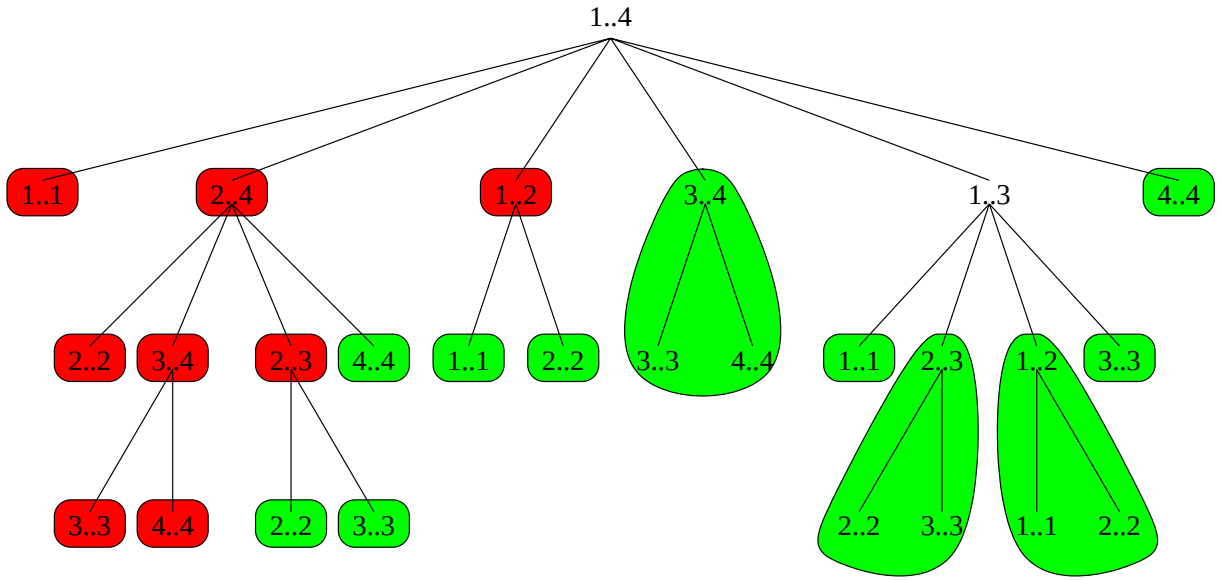




- Existing compilers cannot effectively restructure programs to exploit dynamic redundancy.
- Program execution is composed of *Repetition* and *Variant Computation*.
- Computation results can be recomputed or re-used.

Dynamic Programming Principles

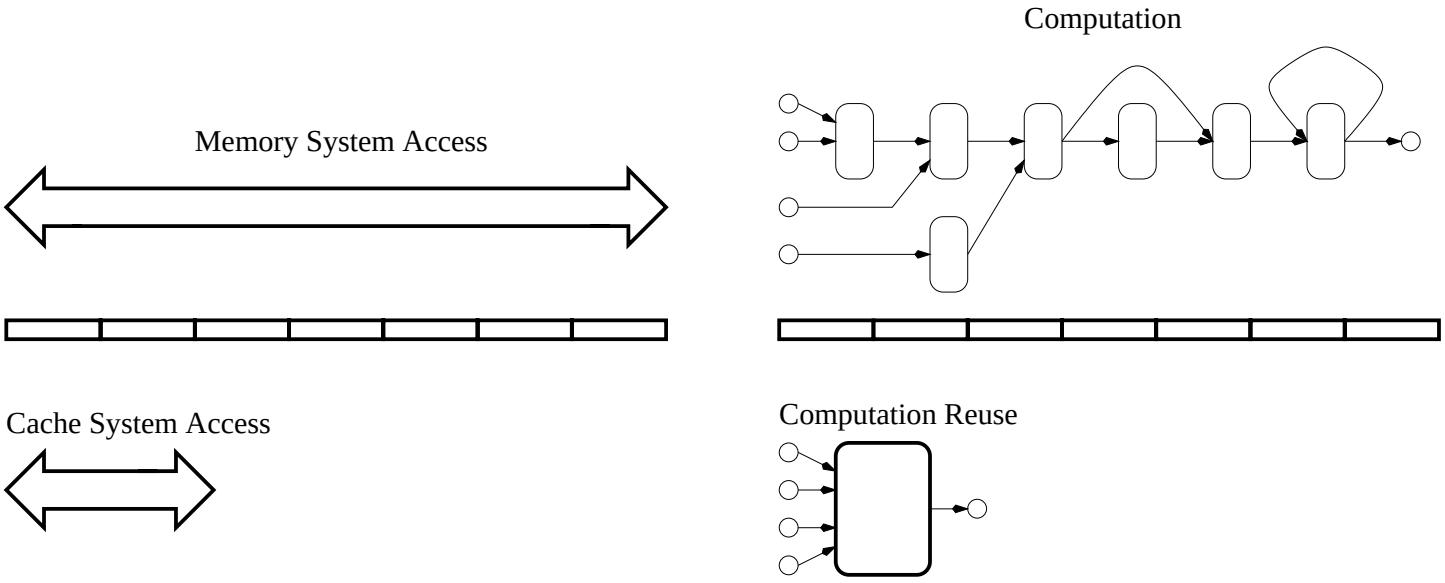
- Element of dynamic programming: Overlapping subproblems.
- Algorithms are characterized by subproblems.
  - Optimal matrix multiply ordering : $(A_1(A_2(A_3A_4)))), (A_1((A_2A_3)A_4)),$ $((A_1A_2)(A_3A_4)), ((A_1(A_2A_3))A_4), ((A_1A_2)A_3)A_4).$
  - Matrix multiply $[p \times q] \cdot [q \times r]$



- Take advantage of structured subproblems in programs.

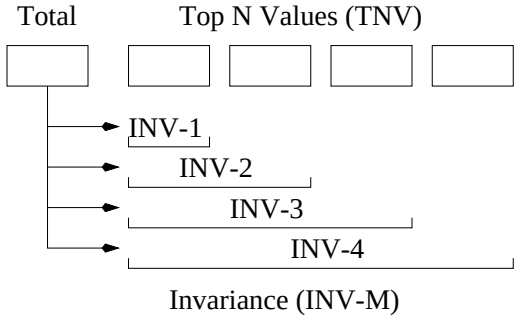
Locality

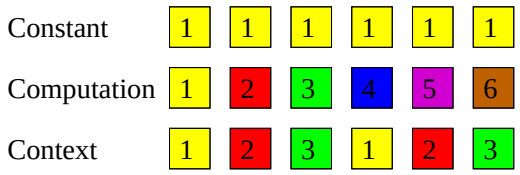
- Machines and compilers regularly exploit the property of locality of memory references. [Cache Memory]
  - Temporal - recently accessed items tend to be accessed in near future.
  - Spatial - references near one another tend to be close together in time.
- Controlled by dynamic program behavior information.

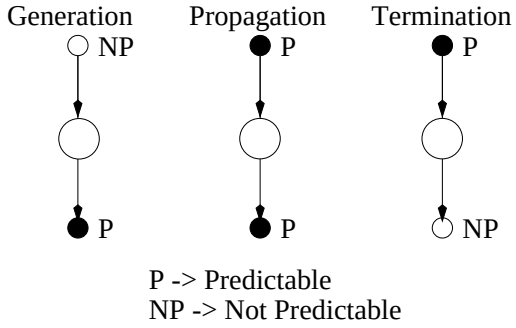


Redundancy Explanation

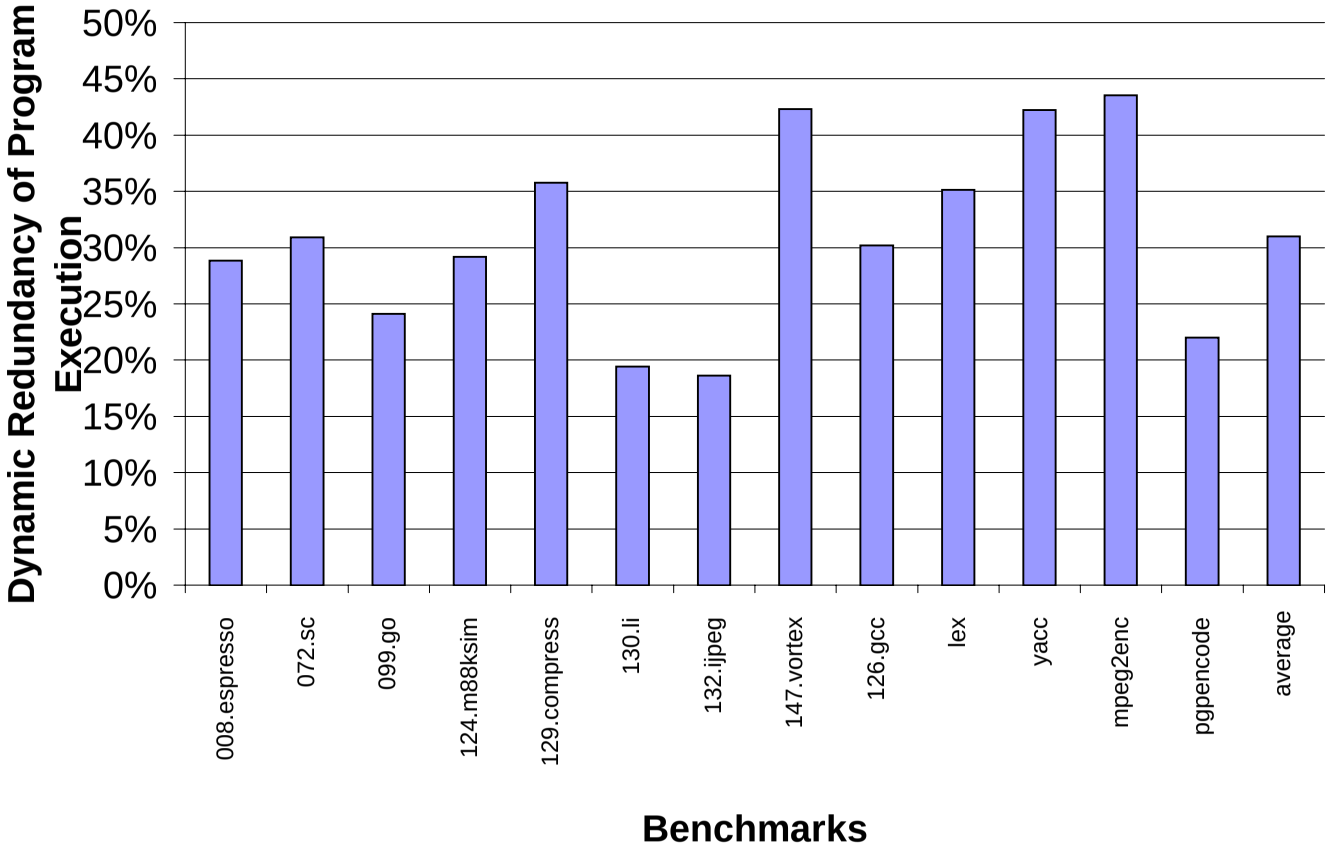
- Sources for redundancy
  - Natural : Data redundancy, quantization, etc.
  - Barriers to compiler optimization :
    - Memory alias resolution, interprocedural optimization.
    - Register spill code, linkage conventions, virtual function calls.
- Techniques for understanding redundancy in programs.
  - Value profiling. [Calder]
  - Predictability of data values. [Sazeides, Gabbay]
  - Dynamic Prediction Graph (DPG) representation. [Sazeides]







Instruction-Level Redundancy



- Four previous unique values, least recently used (LRU) replacement.

Redundancy Elimination - Hardware Techniques

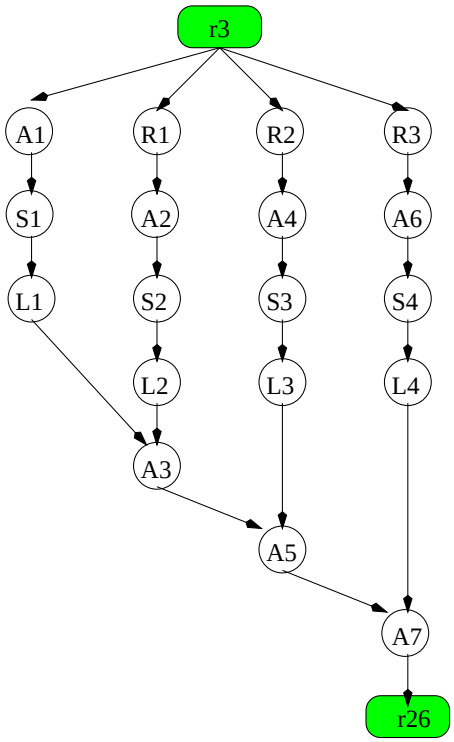
- Dynamic reuse mechanisms
  - Instruction reuse. [Sodani]
  - Block-level reuse. [Huang]
  - Trace-level reuse
    and removal of redundant computation. [Gonzalez, Molina, Tubella]
  - Results cache. [Richardson, Oberman]
- Value prediction
  - Load value prediction. [Lipasti]
  - Value predictive microarchitecture. [Sodani]
- **PROBLEM: These techniques exploit limited opportunities.**
  - Detect, understand, and exploit opportunities.
  - Storage capacity, reusable computation detection, and re-entrance.
  - 6-21% Value prediction/Reuse improvement. [Lipasti, Sodani, Huang]

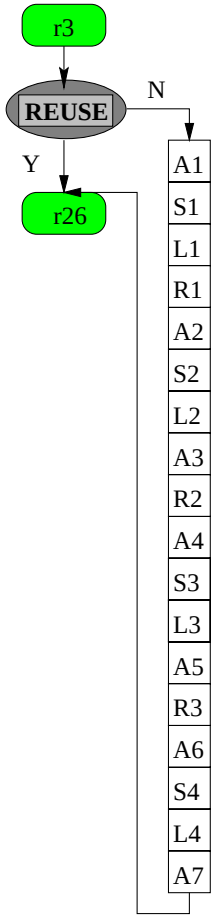
Region-level Computation Reuse

- Population count (*008.espresso*)

```
#define count_ones(v)\
   (bit_count[v & 255] + bit_count[(v >> 8) & 255]\
   + bit_count[(v >> 16) & 255] + bit_count[(v >> 24) & 255])

 int bit_count[256] = {
 0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,
 1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
 1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
 2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
 1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
 2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
 2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
 3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,4,5,5,6,5,6,6,7,5,6,6,7,6,7,7,8
};
```

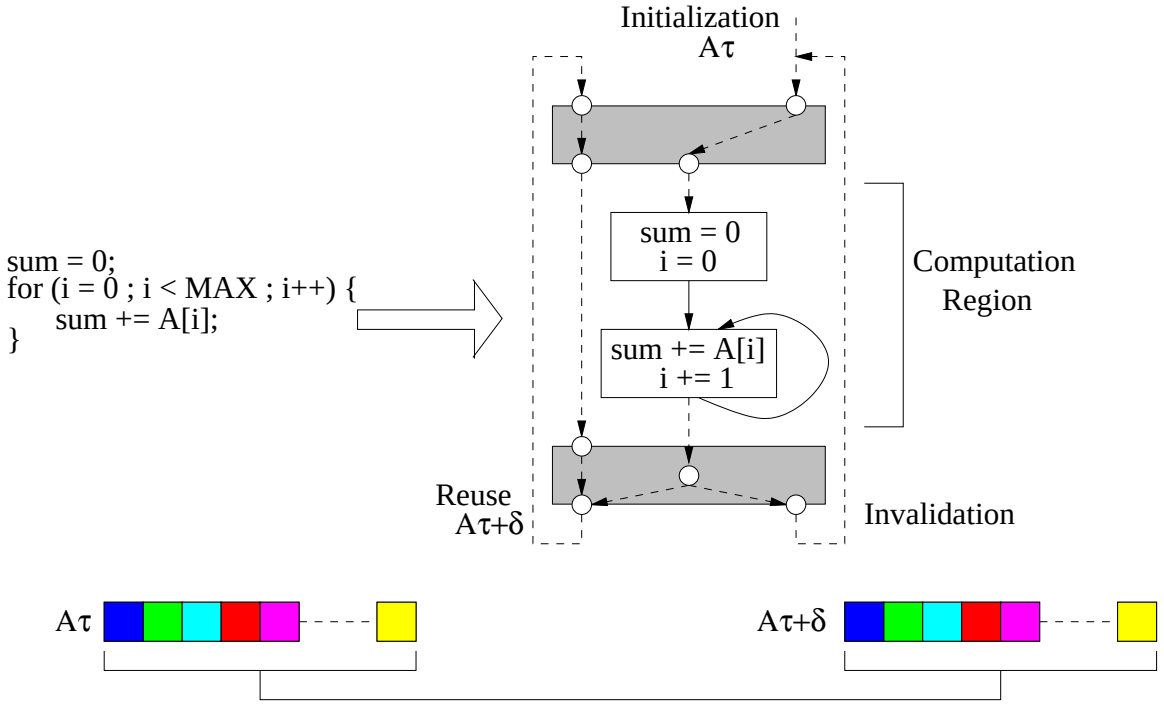




- Code sequence implements a mapping.
- Instruction and data cache logic.
  - Attempt to reuse mechanics of function.
- Compiler/Hardware solutions.

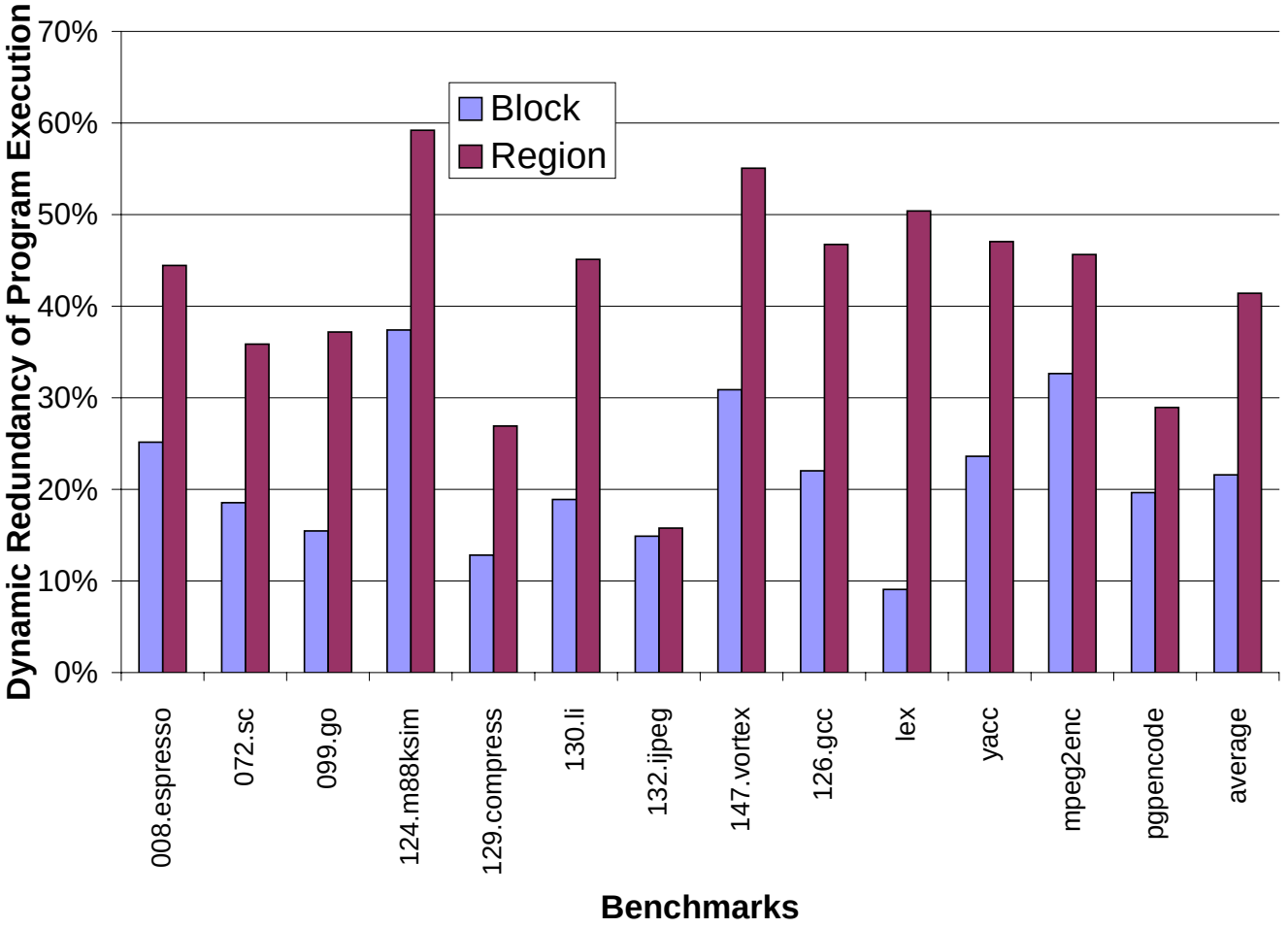
Region-level Computation Reuse

- Exploit large sequences of instructions.
- Desirable to remove loop's computation when the resulting sum is identically computed with a previous invocation.



- Exploit redundancy in domains with hard-to-detect repetition.
- Hardware cost of detecting memory modification.

Program Execution Exploited by Reuse



- Using a dynamic profiling model, gather reusable traces.
- Region : cyclic and acyclic paths of instructions.

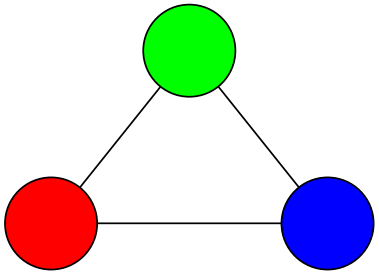
Redundancy Elimination - Compiler Techniques

- Techniques addressing redundancy at compile-time.
  - Subexpression elimination, loop invariant code removal, conditional branch elimination, and constant propagation. [Aho, Sethi, and Ullman]
  - Partial Redundancy Elimination (PRE). [Bodik]
  - Code specialization. [Calder]
- Techniques addressing redundancy at run-time
  - Software memoization. [Richardson]
  - Tree Machine (TM). [Harbison]
- **PROBLEM: These techniques do not address issues at both compile-time and run-time.**

Goal of Work

- **Elimination of Redundant Computation.**
- Exploit each form of redundancy using an appropriate method.
  - Hardware, Compiler, and Integrated Compiler-Architecture.

INTEGRATED COMPILER-ARCHITECTURE

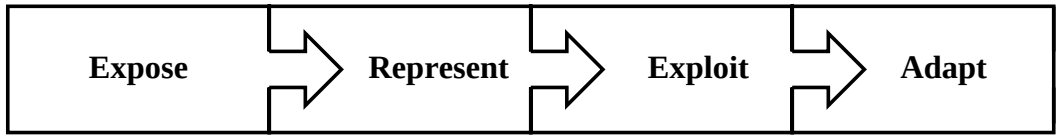


COMPILER HARDWARE

- Integrated Compiler/Architecture : Compiler-Directed Reuse
- Compiler : Code Specialization and Reformulation
- Hardware : General Exploitation of Redundancy

Integrated Compiler and Architecture Approach

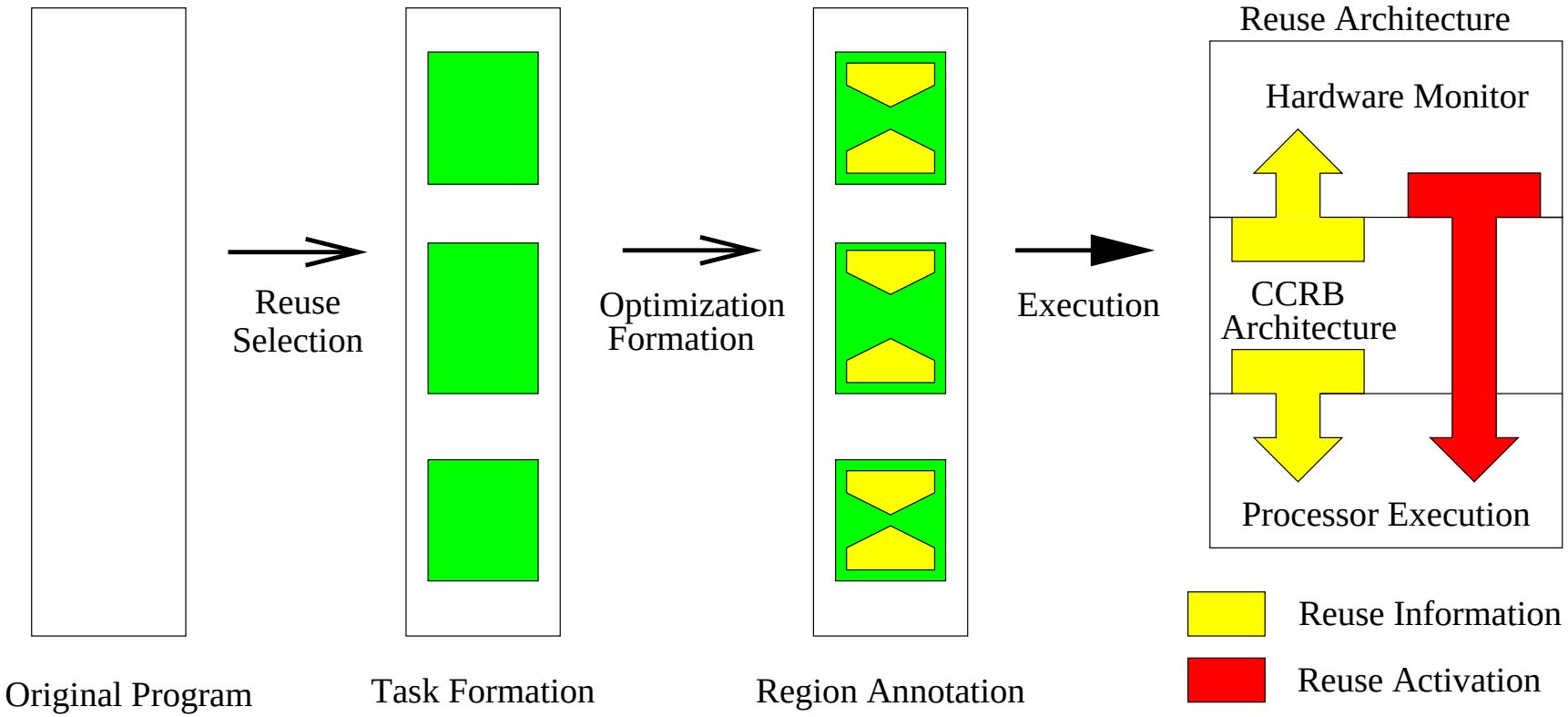
- Develop integrated compiler and architecture approach to eliminate large amounts of dynamically redundant computation.
- Use instruction-set architecture as interface to communicate key information on scope, content, and management of regions of reusable computation.



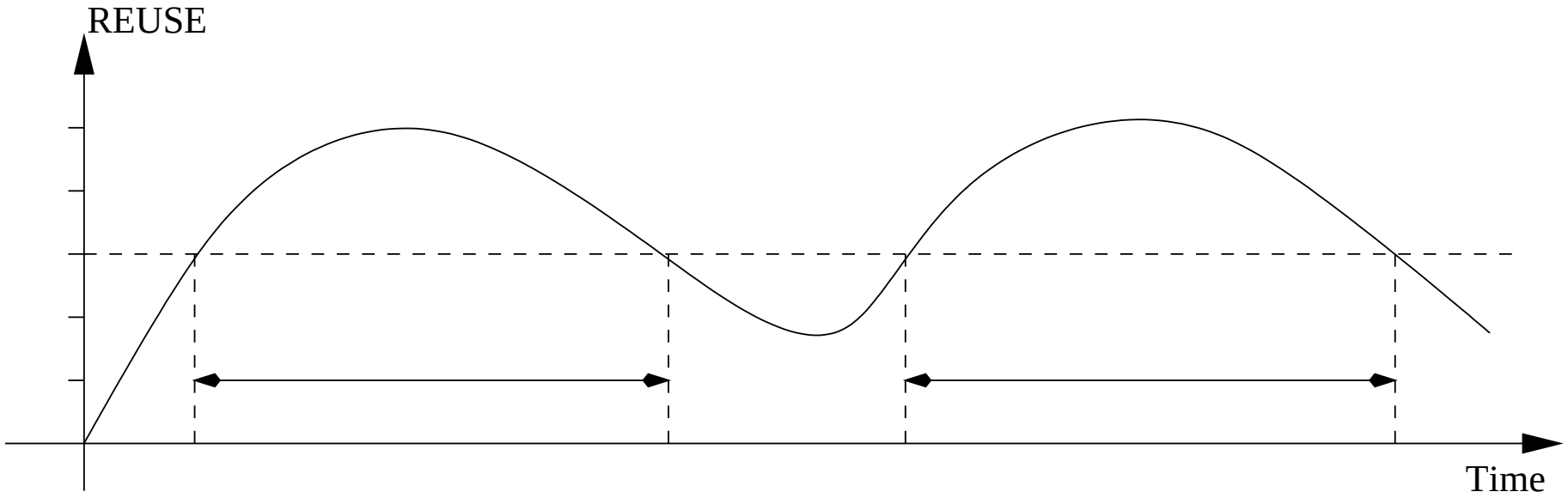
- Compiler exploitation
  - (+) Better scope
  - (+) Representation (analysis)
  - (-) Contention for resources
  - (-) Poor adaptability
- Hardware/Dynamic exploitation
  - (+) Hardware monitoring
  - (+) Run-time adaptability
  - (-) Substantial hardware cost
  - (-) Scope of solution

Architecture Framework

- Compiler provides information to architecture about reuse opportunities.
- Hardware activates reuse opportunities.

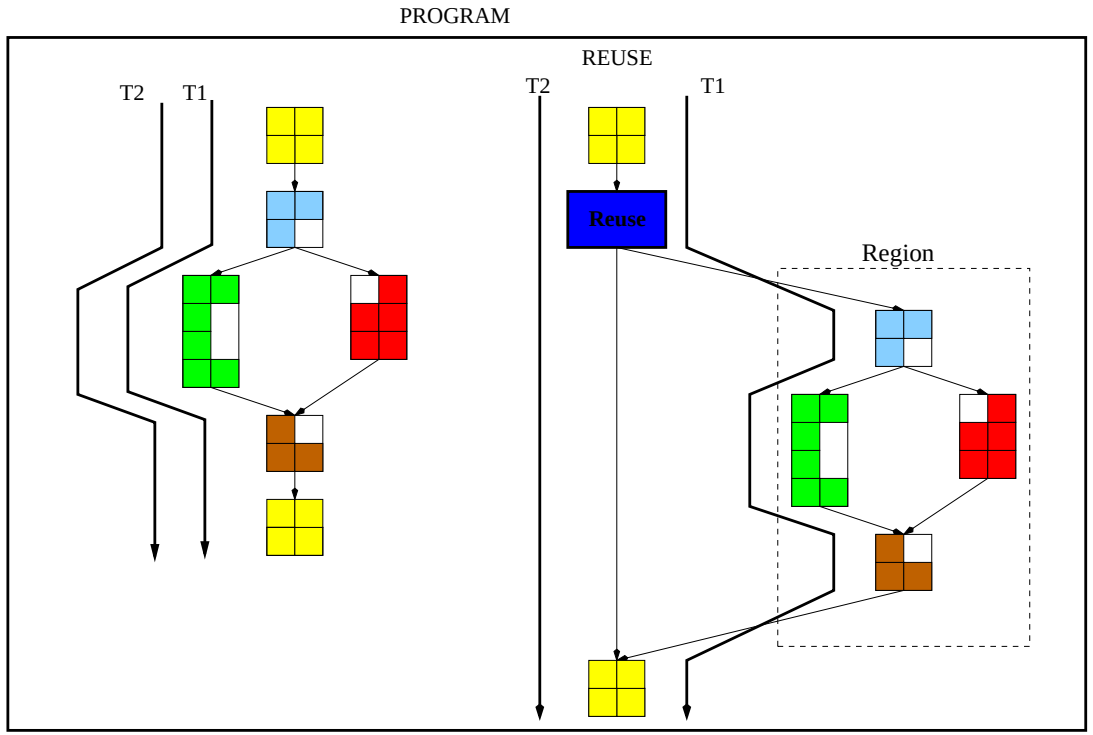


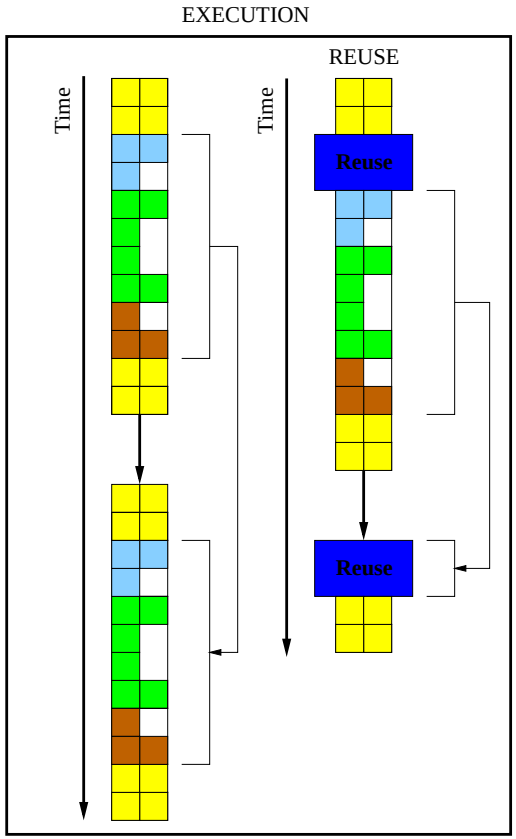
Reuse Template and Hardware Monitoring



- Run-time monitoring and hot spot detection. [Merten, Smith]
  - Large portion of silicon used for non-functional logic.
  - Improve hit rate and utilization of resources.
- Dynamic compilation.
  - Partial evaluation templates. [Consel and Noel]
  - Templates for semi-invariant code sequences. [Auslander]
- Compiler-directed triggering (input, memory validation).

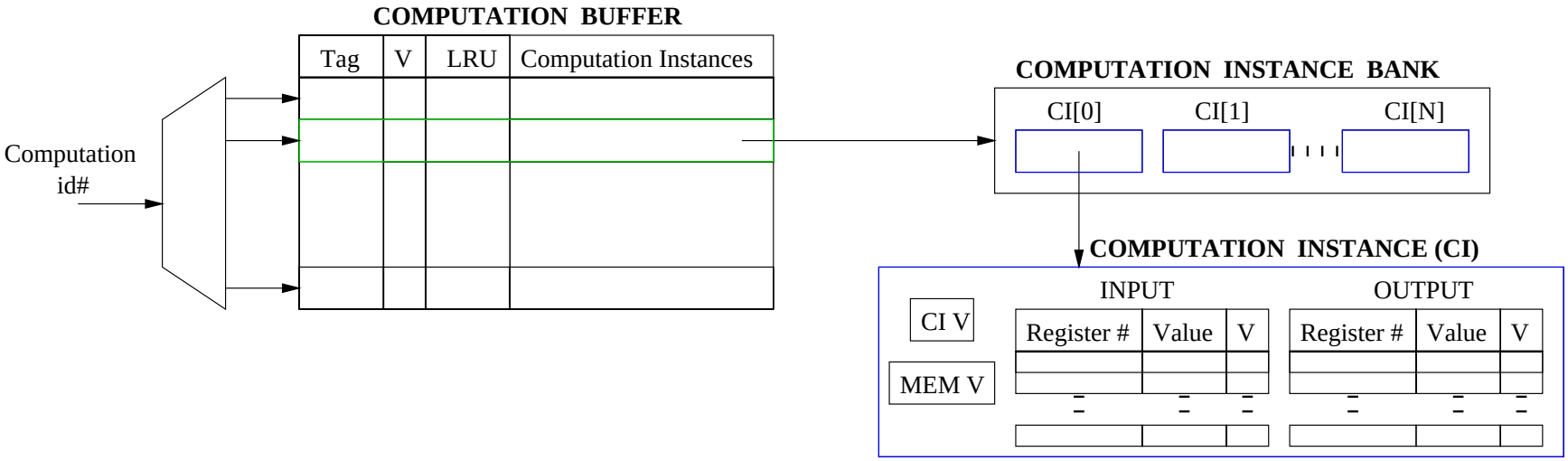
Memoization Mode





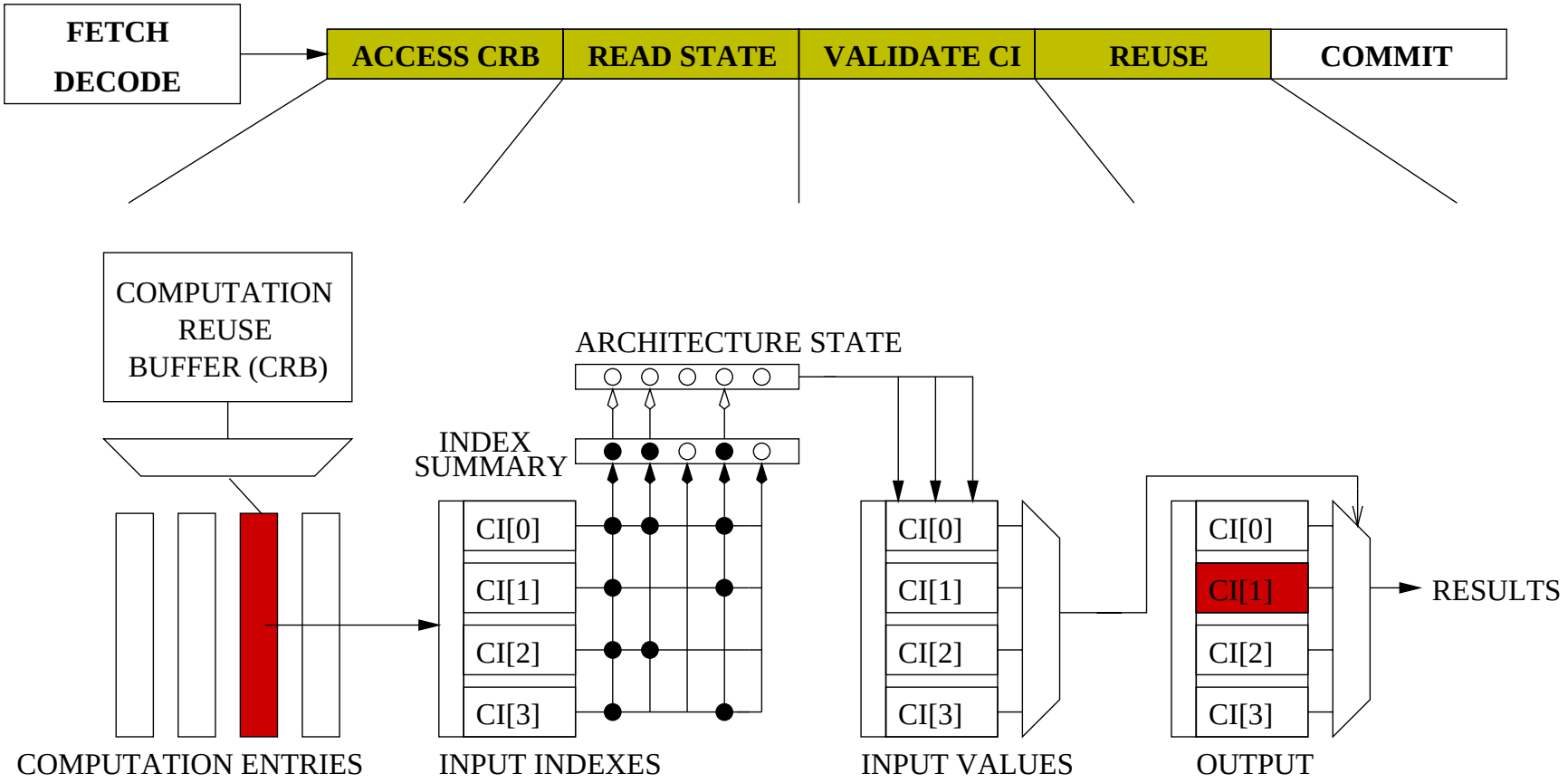
- Reuse Interface (Instruction Set Architecture)
  - Data flow (live-out) : instruction extensions
  - Control flow region : control extensions
  - Memory reuse : invalidation instruction
  - Reuse : reuse instruction

Proposed Architecture Mechanism - Compiler-Directed Reuse Buffer

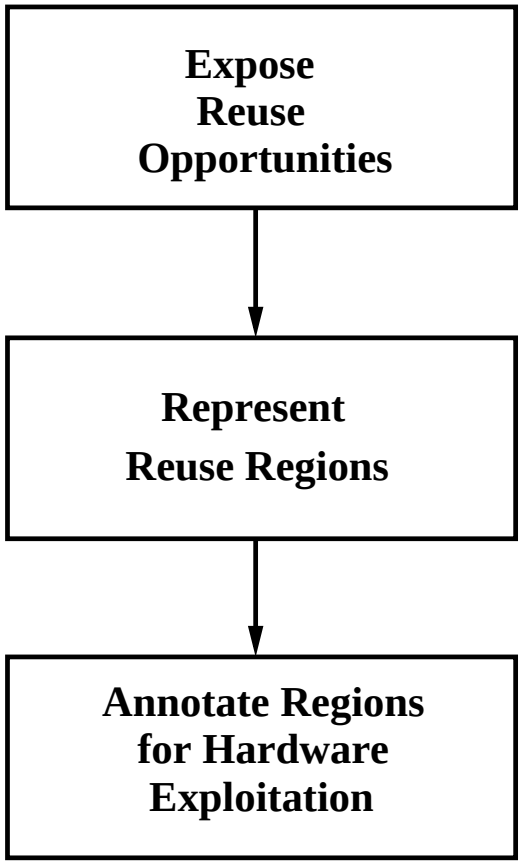


- Computation entry- referenced by computation identifier
- Computation instance - mapping of inputs→outputs
- Computation equivalence - hardware/compiler solution

CCR Microarchitecture



Current Compilation Framework

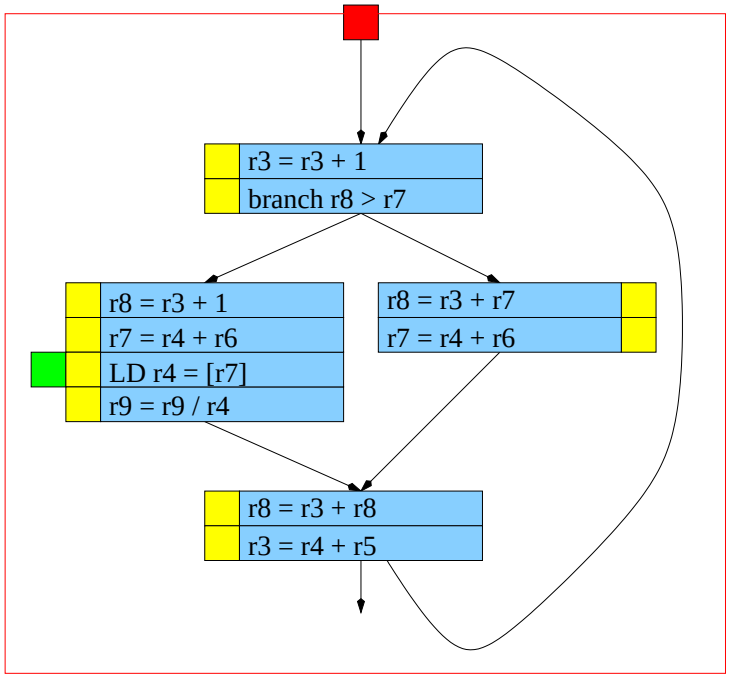


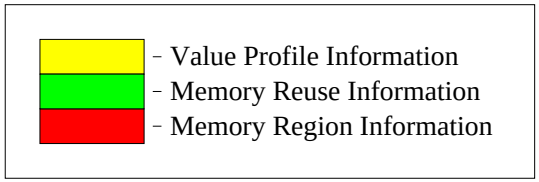
- Deterministic computation
- Reuse analysis (value profiling)
- Value flow analysis
- Static formation of reuse tasks

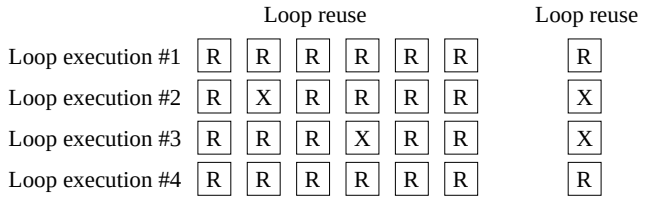
- Data flow analysis
- Control flow analysis

- Optimization
- Control flow restructuring
- Memory validation

Reuse Profiling System (RPS)







- Estimate reuse behavior
- Value profiling
  - Invariance INV-TNV
  - Predictability
  - MRV (most recent value)
  - TBR (time between reuse)
- Memory reuse profiling
  - Load reuse
- Memory region profiling
  - Consider reuse/invalidation for every load.

Deterministic Computation Regions

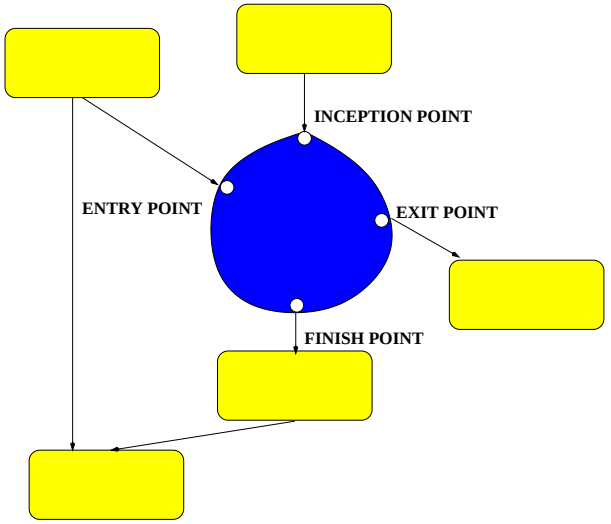
- A *deterministic computation region* is an arbitrary, connected subgraph of the program control flow graph that can be analyzed to determine the location of all input operands that affect the region's computation.
- Stateless (SL) regions are paths of computation based only on register operands and not on memory state.
- Memory dependent (MD) regions are paths of computation based on both register operands and memory state, with the requirement that the memory dependence be either completely determined at compile time.

Inception Point Starting point for memoization mode and location for reuse instruction.

Finish Point Ending point for successful memoization mode.

Exit Point Side exit from computation region and termination of memoization mode. No reuse along paths from inception to exit point.

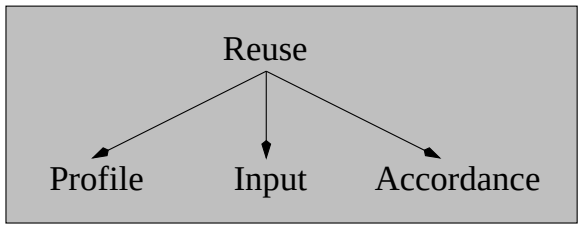
Entry Point Side entrance to reusable region that is not involved with reuse or memoization of computation.



Reusable Region Selection

- A *reusable computation region* is a set of control blocks in which reuse opportunities exist from a start node to an end node.
  - Side entrance/exit may exist (no reuse opportunities).
- Primary contribution: designate path boundary to reuse architecture.
- Reusable computation regions are formed heuristically.
- Profile

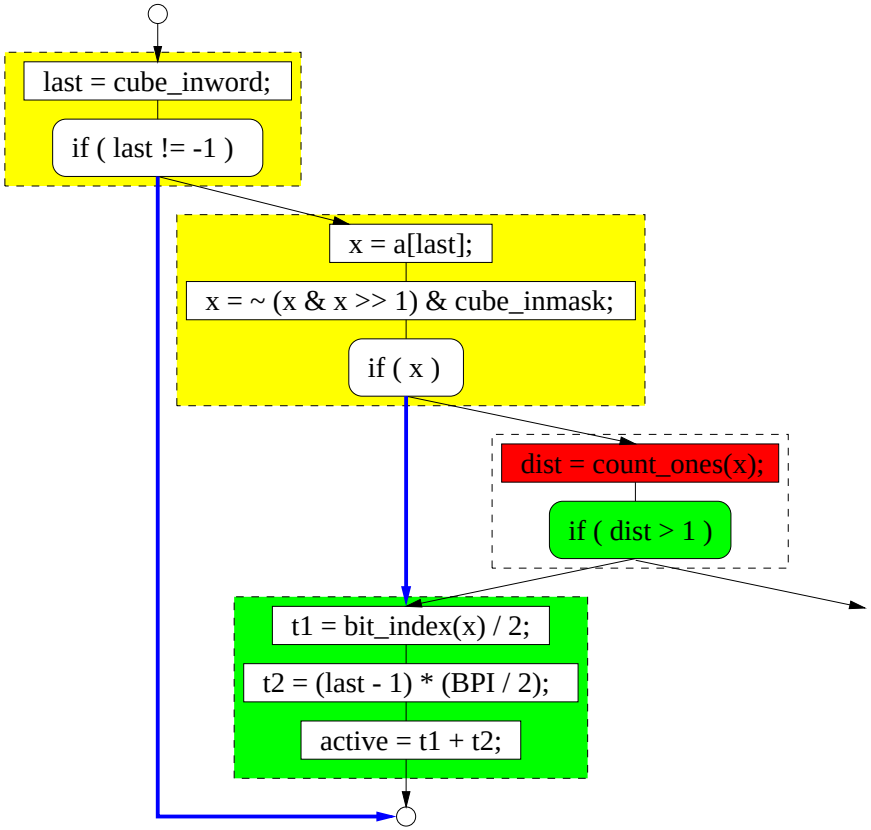
$$Reuse(i) = \left(\frac{ValueInv[k](i)}{Exec(i)} \geq R\right)$$
$$MemReuse(i) = \left(\frac{Valid(i)}{Exec(i)} \geq R_m\right)$$
$R = .65$ and $R_m = .65$



- Input characteristics
- Accordance
  - Adding new memory relations
- Overall decision - minimal path height and estimated success.

Region Formation Example

- Example region formation for *_cactive* in the benchmark *008.espresso*:
  - Formation process: cyclic region formation, acyclic region formation, and region transformations.

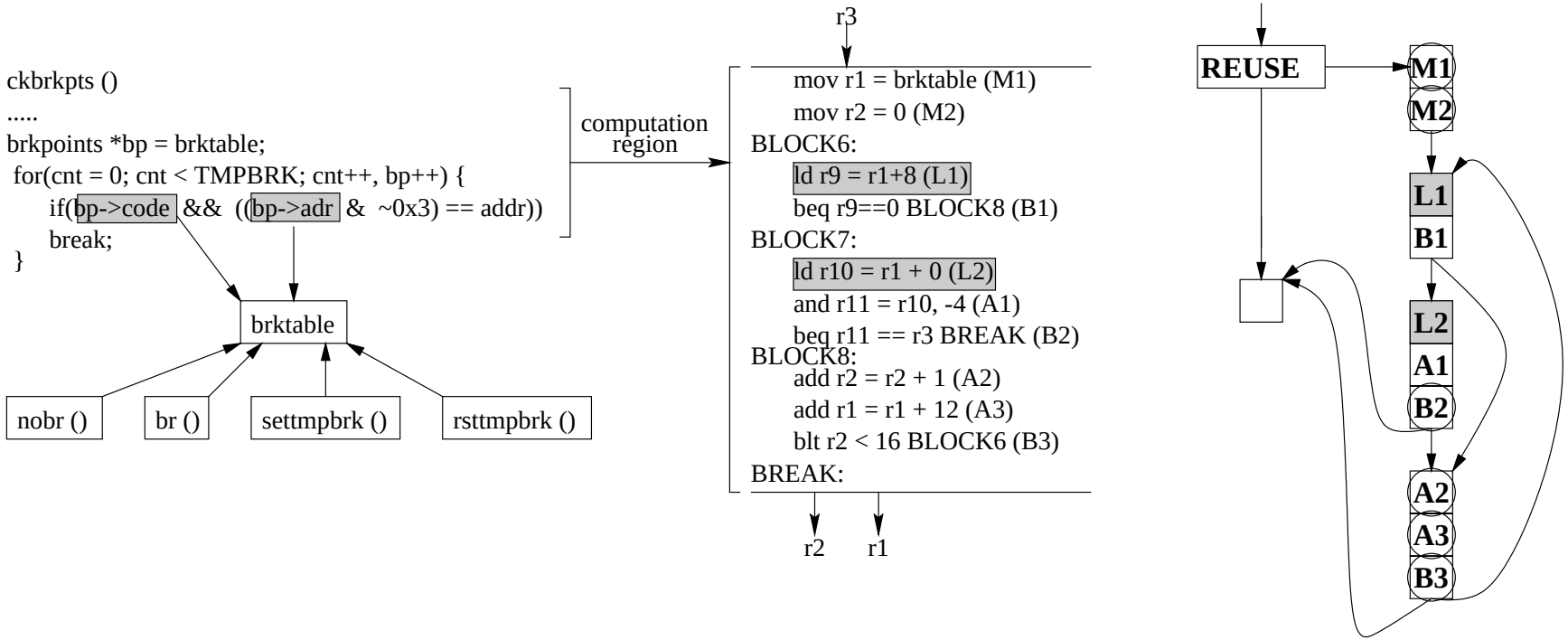


Acyclic region formation steps:

- Step 1: Reuse seed selection, ordered by reuse potential.
- Step 2: Successor formation.
- Step 3: Predecessor formation.
- Step 4: Subordinate path formation.

Memory Dependent Cyclic (MD_C) Region Example

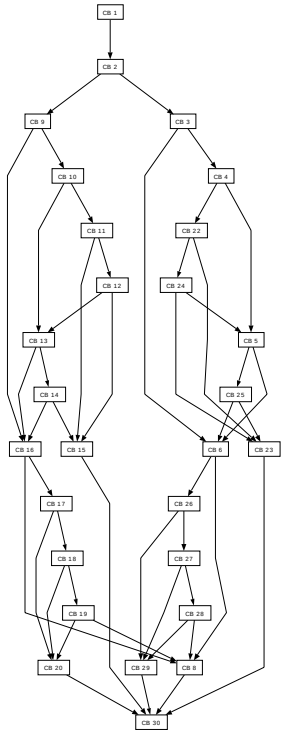
- In the function *_ckbrkpts* in the benchmark *124.m88ksim*:
  - Reduces execution time from 423,342 cycles → 58,392
  - Improves instruction/data cache and branch prediction performance.



Stateless Acyclic (SL_A) Region Example

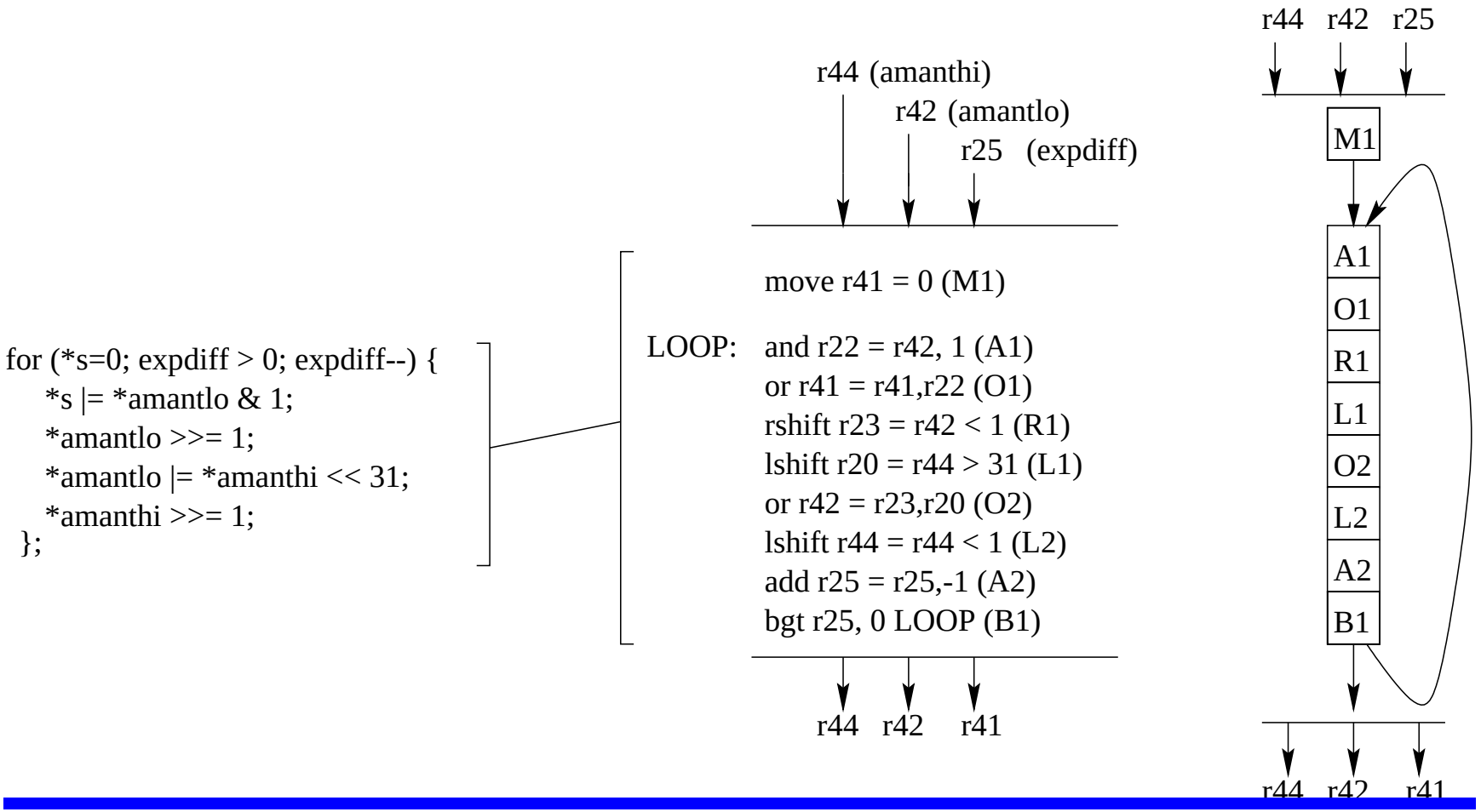
- In the function *_round* in the benchmark *124.m88ksim*:
  - Minimum of 5 branches each invocation of region.
  - Transforms branch intensive code with low ILP to simple lookup.
  - 80% of region execution captured with 4 computation 6-input CI.

```
if (sign) {
      if ((rnd == RN) && ((g && (r || s)) || (l && g)))
        return(1);
      else if ((rnd == RM) && (g || r || s))
         return(1);
 }
else
      if ((rnd == RN) && ((g && (r || s)) || (l && g)))
        return(1);
      else if ((rnd == RP) && (g || r || s))
         return(1);
 }
```

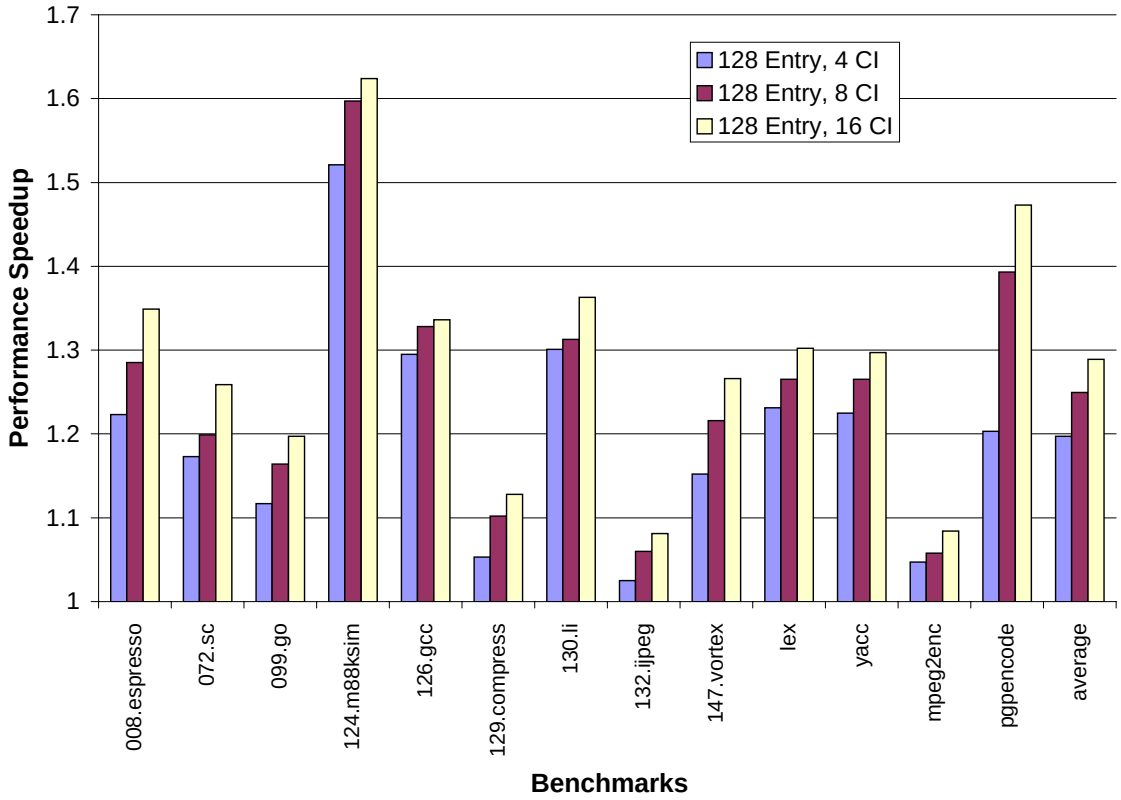


Stateless Cyclic (SL_C) Region Example

- In the function *_alignd* in the benchmark *124.m88ksim*:
  - Eliminates loop execution.

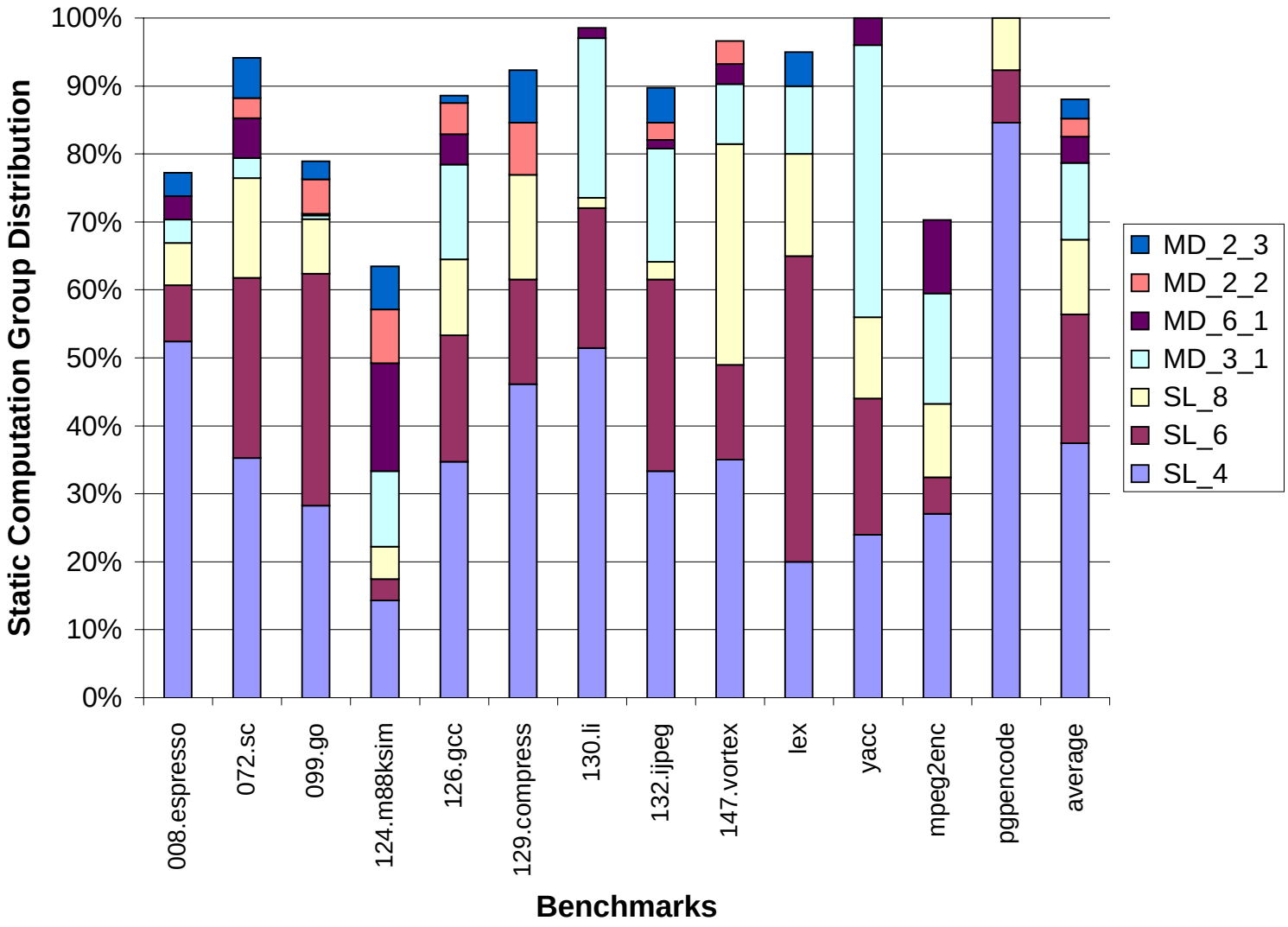


Performance

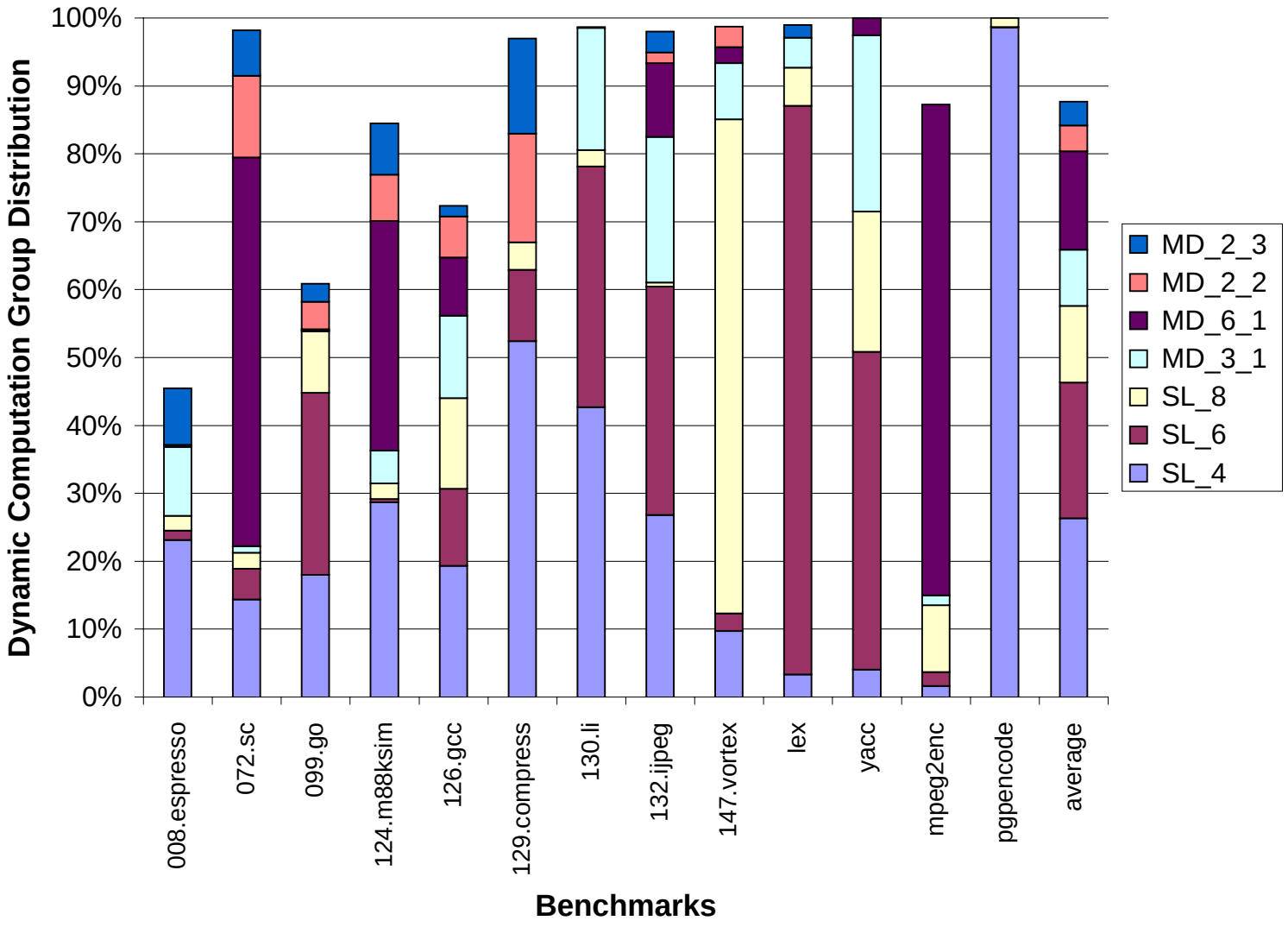


- 32K Data Cache, 32K Instruction Cache, and 4096-entry 2-bit BTB.
- 6-issue: 1 branch, 4 integer, 2 memory, and 1 float.
- Three cycle execution time for successful reuse and eight cycle penalty.

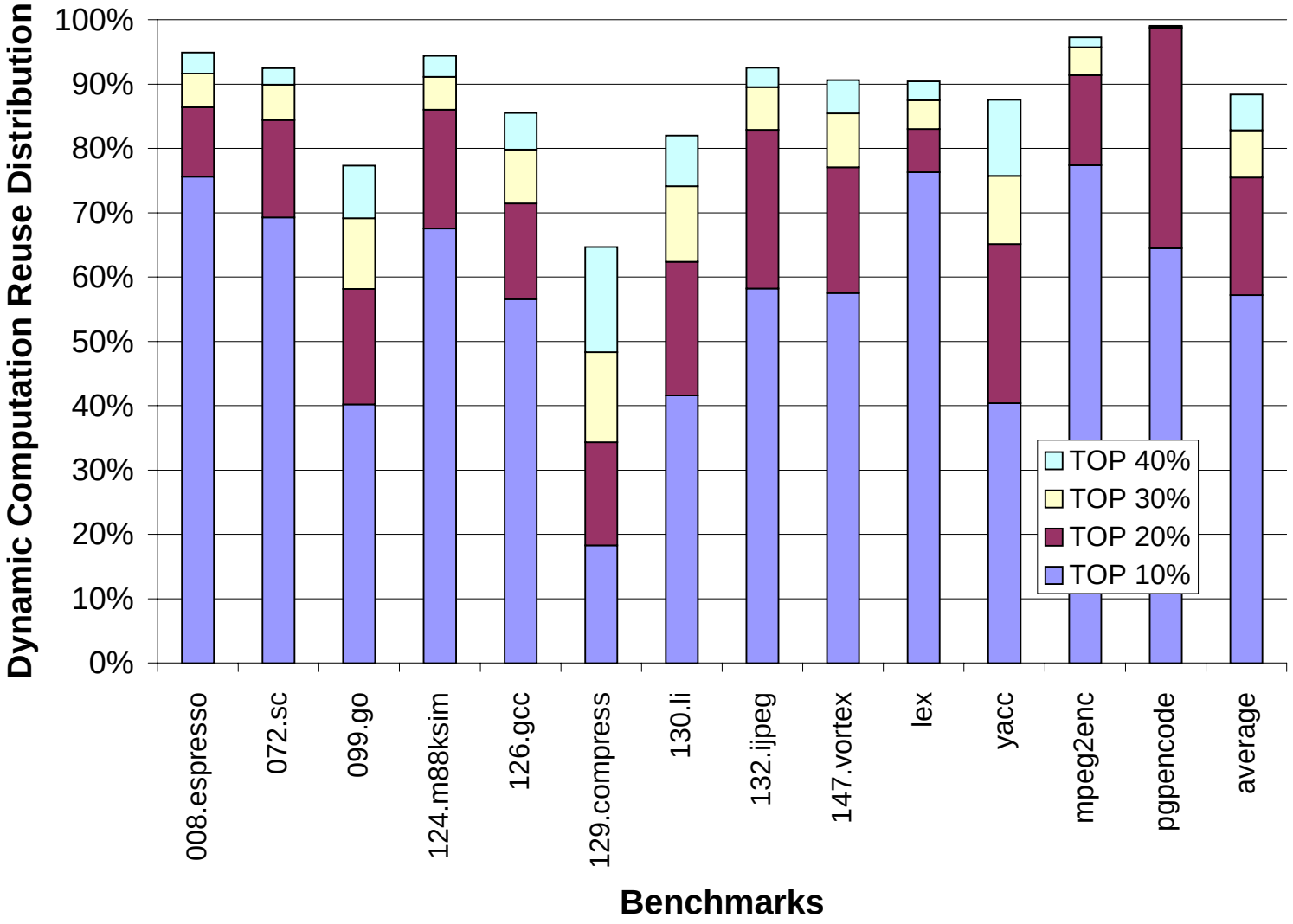
Computation Group Static Distribution



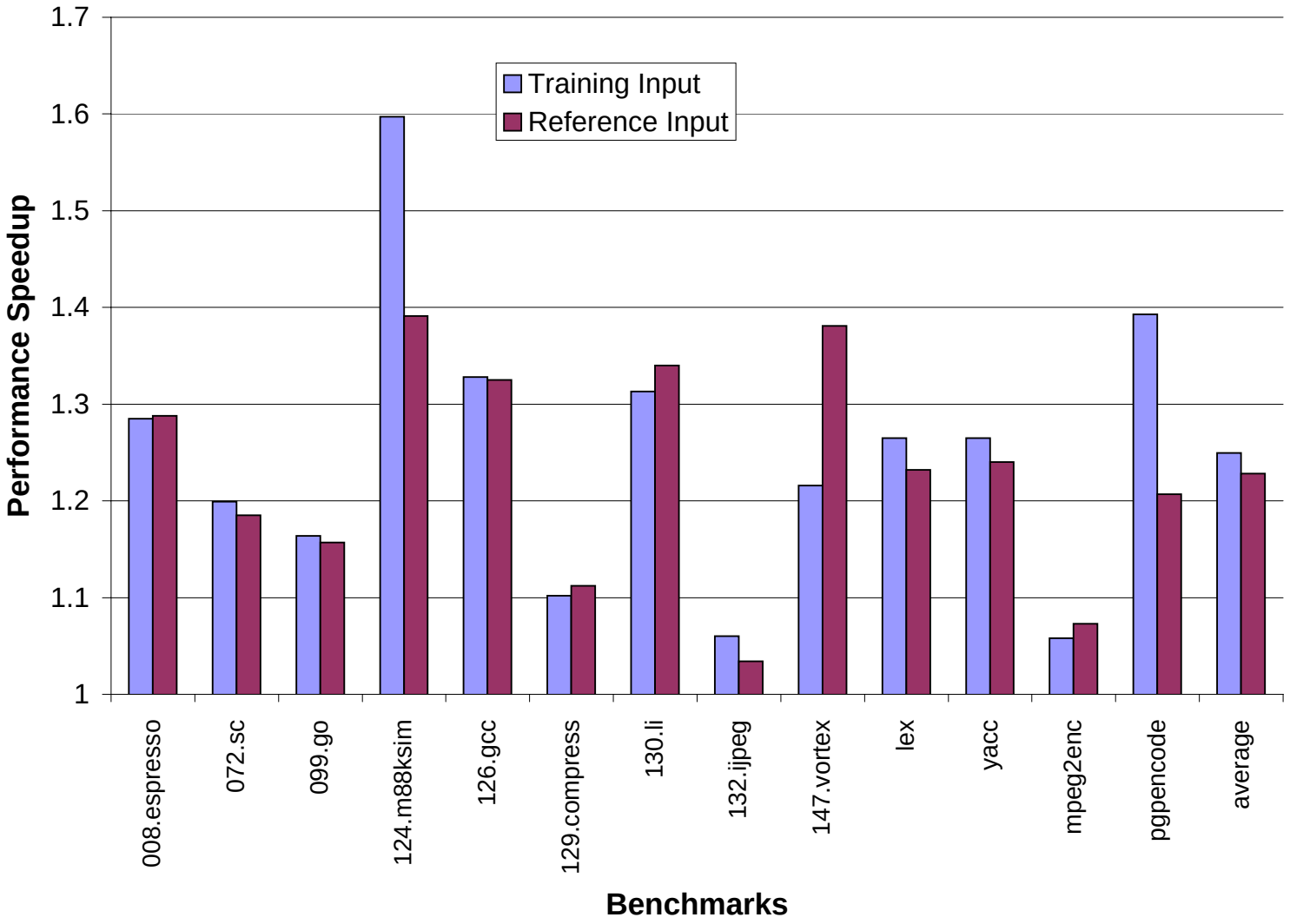
Computation Group Dynamic Distribution



Computation Dynamic Distribution

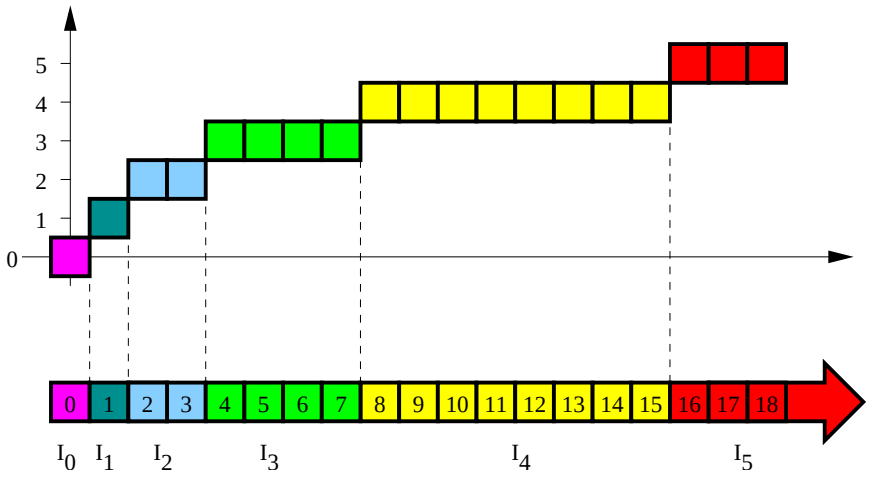


Cross Profile Performance

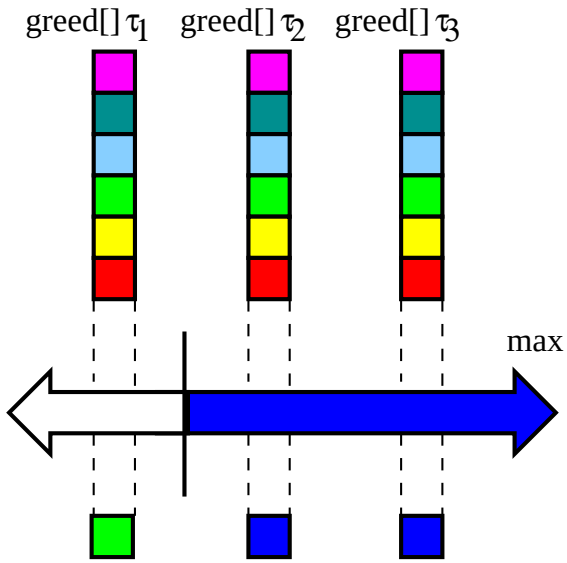


Future Reuse Architecture Development

- Range sensitivity - mapping of range of input values to output result.
  - Range locality of values.
  - Many to one mapping.



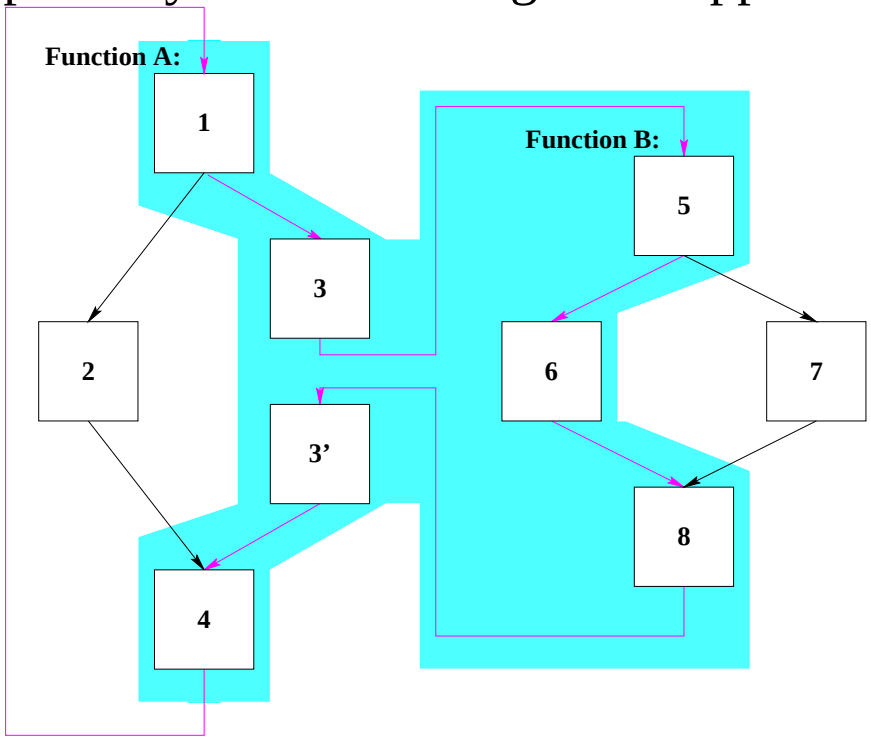

```
for (i=0; i<nstate; i++) {
    if (greed[i] >= MAX) {
        max = greed[i];
        maxi = i;
    }
}
```



- Storing reuse information in the cache hierarchy.
- Value speculation to hide latency of reuse validation.

Future Compiler Development

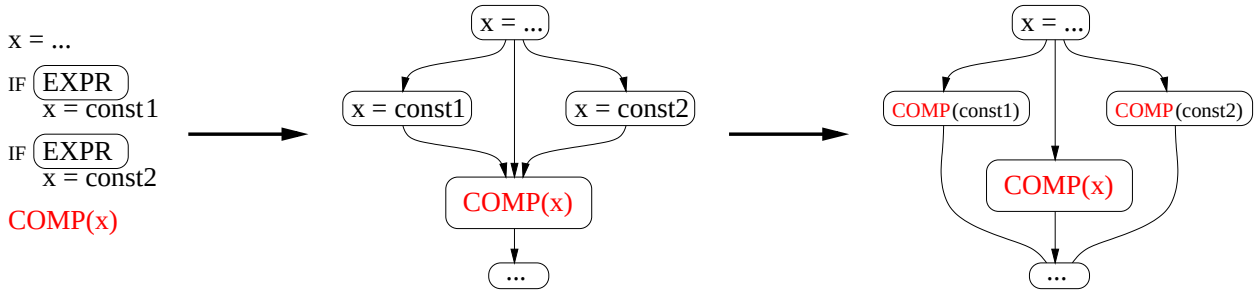
- Static compiler analysis for reusable computation formation.
- Program callgraph analysis for directing reuse opportunities.



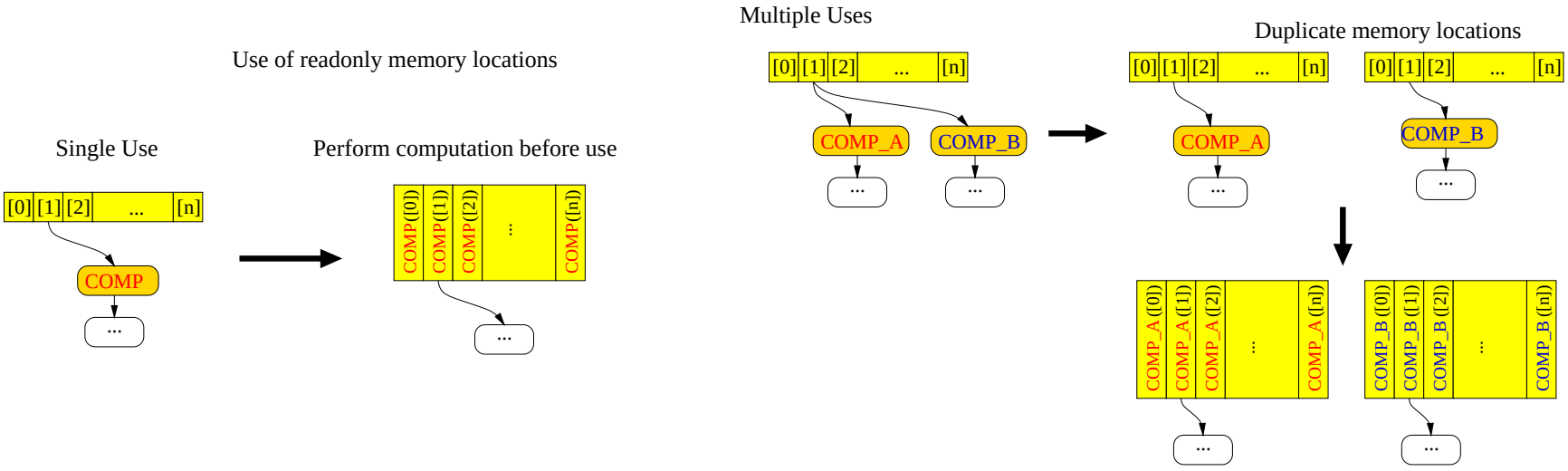
- Evaluation of library codes.
- Apply reuse to anonymous memory structures.

New Compiler Techniques for Eliminating Redundancy

- Value flow analysis.

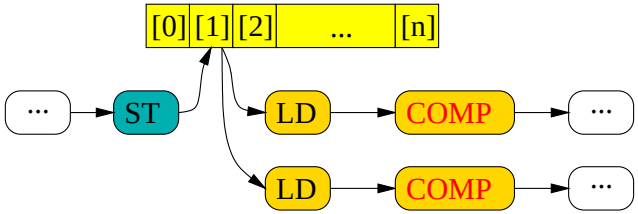


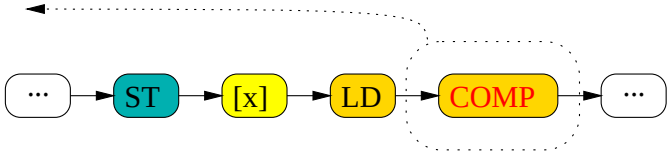
- Variable connection framework with expression analysis.



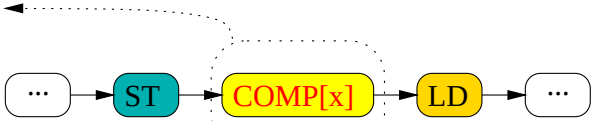
Eliminating Dynamic Redundancy Using the VCF

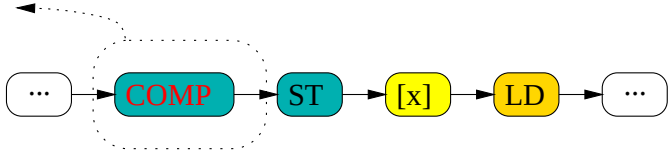
Use of a dynamic memory location

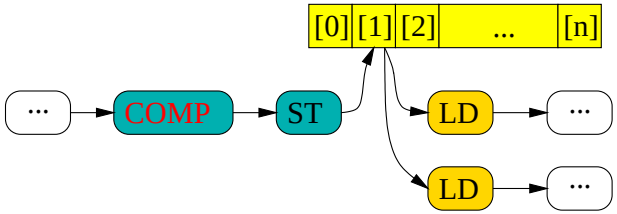




Pull computation before store into memory



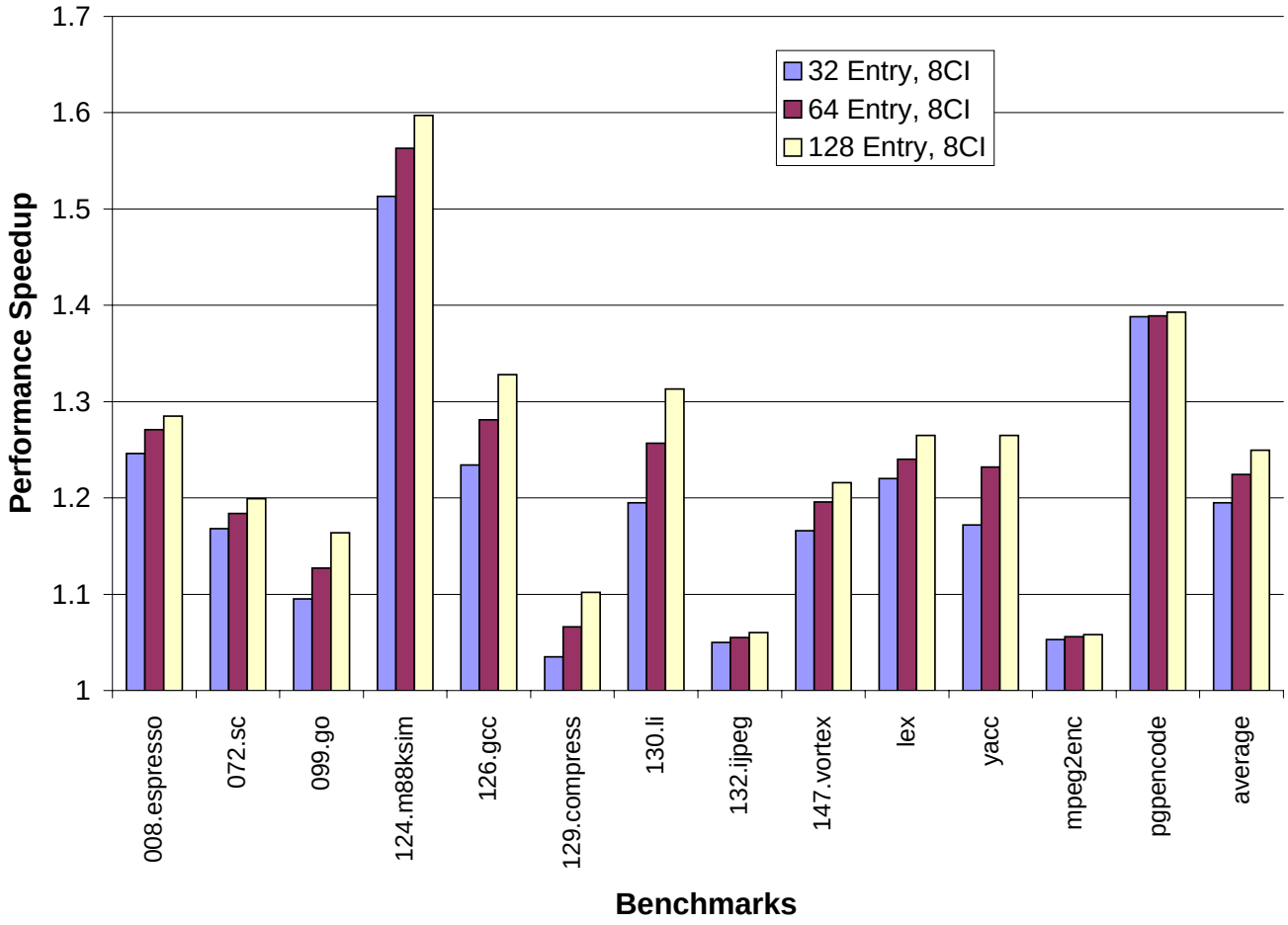




Conclusions

- Traditional computation reuse techniques have not been able to dramatically exploit redundancy.
- The combination of new compiler framework techniques and the integrated compiler/hardware framework (CCR Architecture) shows promise for achieving high levels of performance.
- Combination of profiling information and dataflow analysis establishes an initial method of detecting the significant subproblems revisited within general purpose programs.

Performance (Varying Computation Table Size)

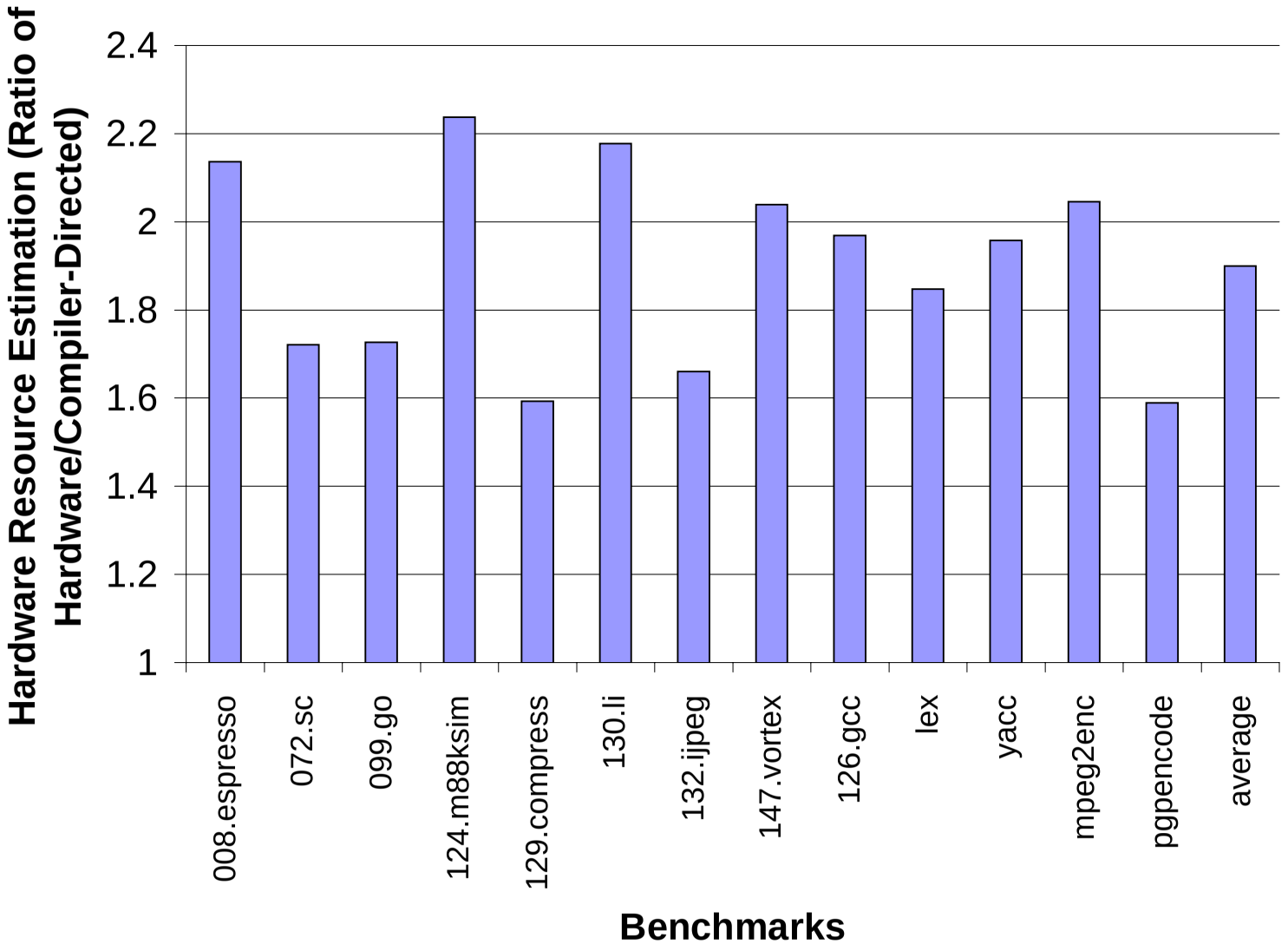


Application Statistics

Benchmark	Reuse Opportunities
008.espresso	148
072.sc	70
099.go	440
124.m88ksim	128
129.compress	36
130.li	57
132.ijpeg	60
147.vortex	192
126.gcc	1764
lex	51
yacc	69
mpeg2enc	83
pgpencode	51

- 50% stateless computations.
- 10% cyclic computations.

Potential Hardware



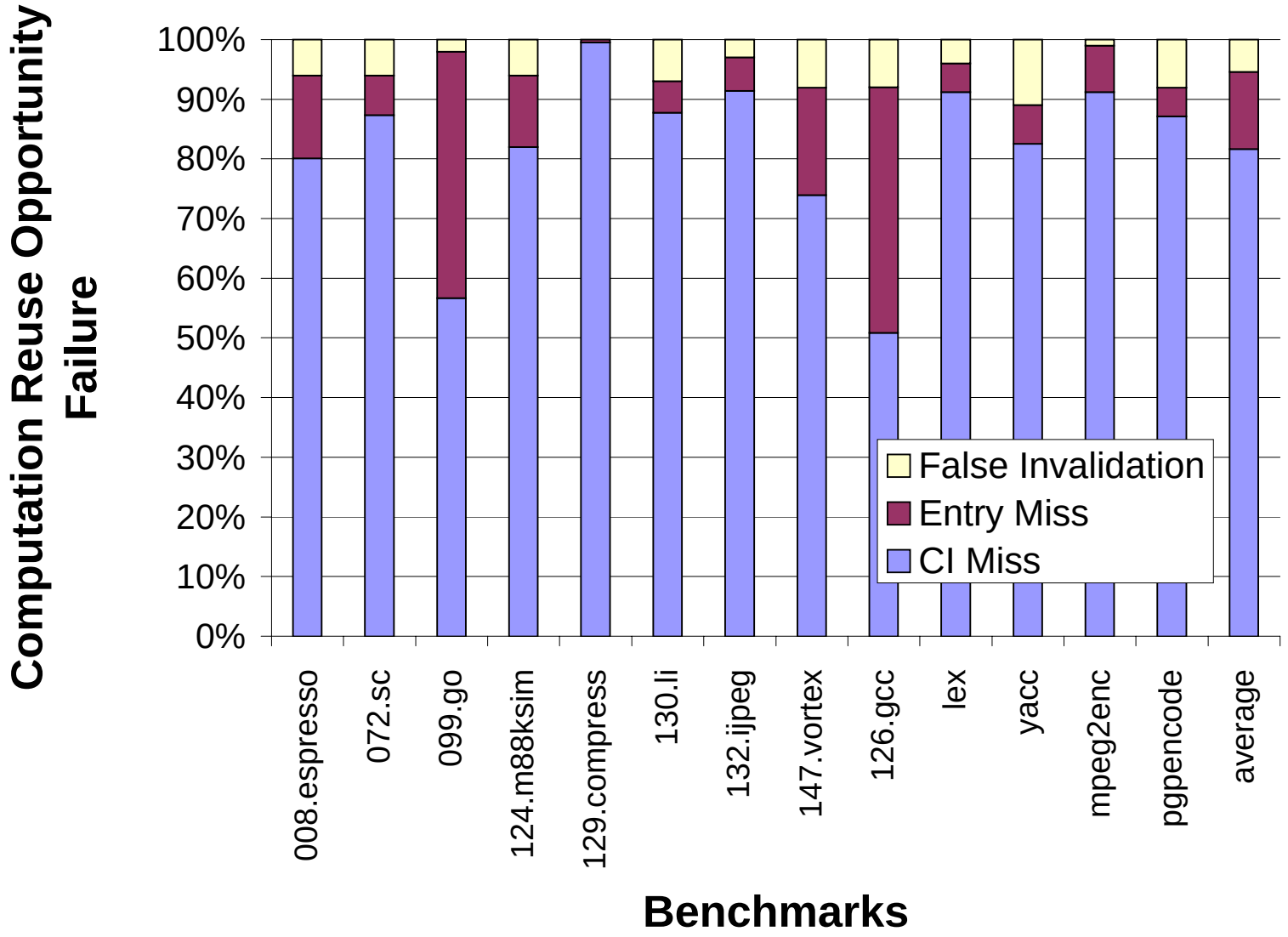
Hardware Mechanism - Bit Count

Scheme	Cost Expression	Cost
CCRB	$N(Entry + CI(CI\_COST + Num\_Input(Input\_Cost) + Num\_Output(Output\_Cost)))$	338944

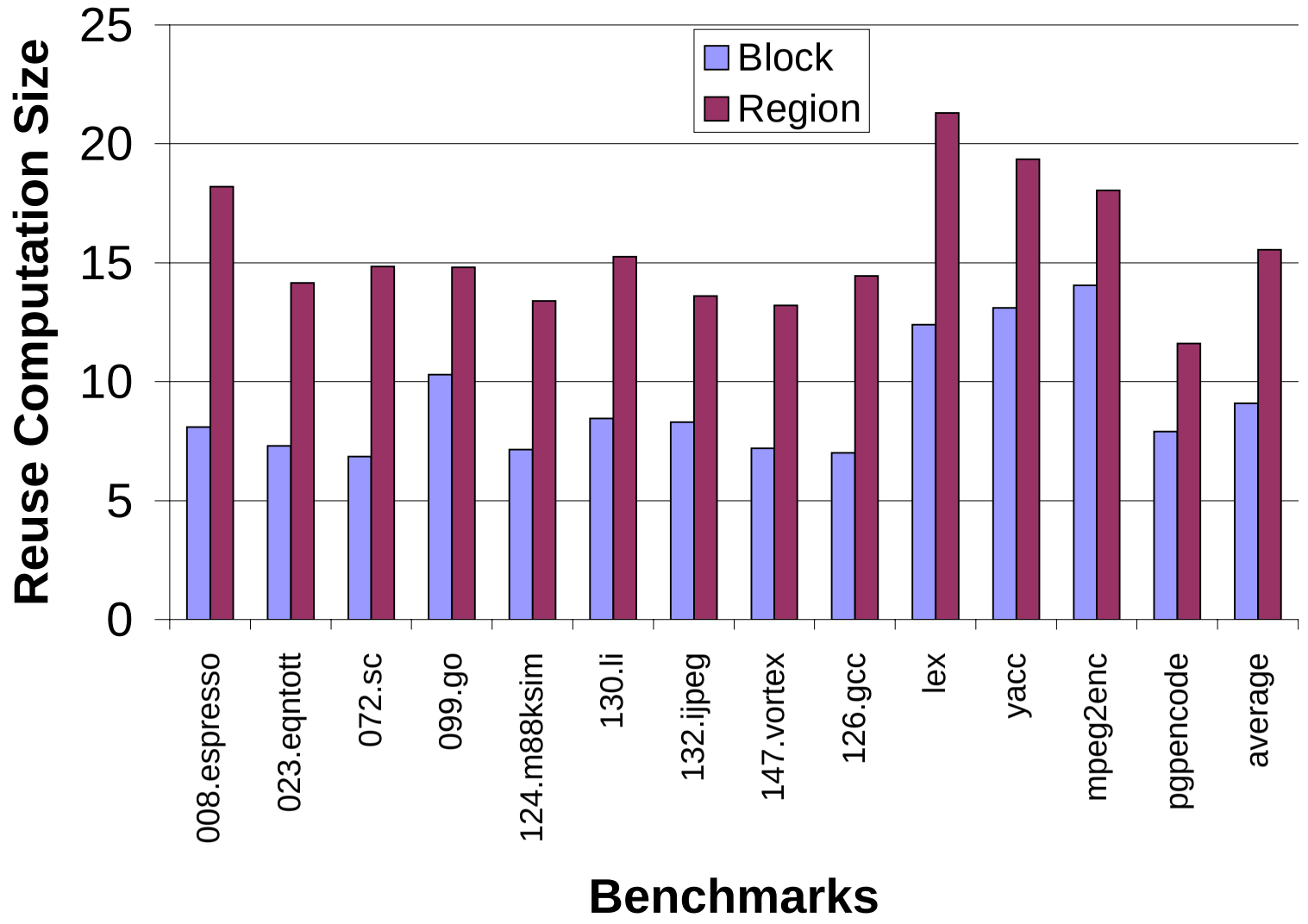
Var	Definition
N	number of computation entries
$Entry$	size of computation entry information
CI	number of computation instances per entry
CI_COST	size of computation instance information
Num_Input	number of input registers in computation instance
$Input\_Cost$	size of input register information (value, reg index, valid)
Num_Output	number of output registers in computation instance
$Output\_Cost$	size of output register information (value, reg index, valid)

$N = 64$, $Entry = 128$, $CI = 8$, $CI\_COST = 5$, $Num\_Input = 16$,
$Input\_Cost = 40$, $Num\_Output = 8$, $Output\_Cost = 40$.

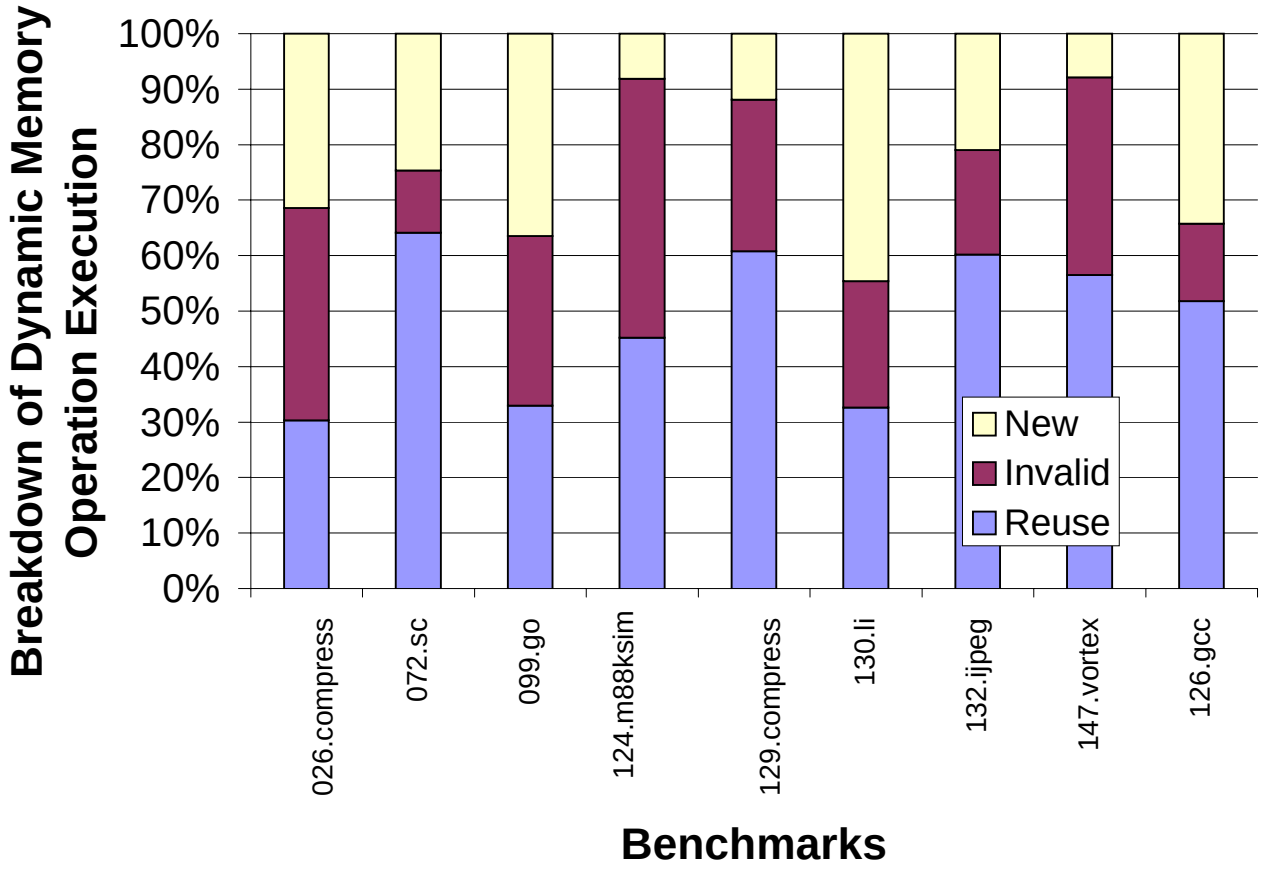
Miss Breakdown



Potential Reuse Sizes (Acyclic)



Memory Reuse Execution



Computation Dynamic Class