Florida State University

In Search of Near-Optimal Optimization Phase Orderings

Prasad A. Kulkarni
David B. Whalley
Gary S. Tyson
Jack W. Davidson

Florida State University

Optimization Phase Ordering

- Optimizing compilers apply several optimization phases to improve the performance of applications.
- Optimization phases interact with each other.
- Determining the order of applying optimization phases to obtain the best performance has been a long standing problem in compilers.

Exhaustive Phase Order Evaluation

- Determine the performance of all possible orderings of optimization phases.
- Exhaustive phase order evaluation involves
  - generating all distinct function instances that can be produced by changing optimization phase orderings (CGO ’06)
  - determining the dynamic performance of each distinct function instance for each function

Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Correlation between dynamic frequency measures and processor cycles
- Genetic algorithm performance results
- Future work and conclusions

Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Correlation between dynamic frequency measures and processor cycles
- Genetic algorithm performance results
- Future work and conclusions

Experimental Framework

- We used the VPO compilation system
  - established compiler framework, started development in 1988
  - comparable performance to gcc –O2
- VPO performs all transformations on a single representation (RTLs), so it is possible to perform most phases in an arbitrary order.
- Experiments use all the 15 available optimization phases in VPO.
- Target architecture was the StrongARM SA-100 processor.

Florida State University

Disclaimers

- Instruction scheduling and predication not included.
- VPO does not contain optimization phases normally associated with compiler front ends
  - no memory hierarchy optimizations
  - no inlining or other interprocedural optimizations
- Did not vary how phases are applied.
- Did not include optimizations that require profile data.

Benchmarks

- Used one program from each of the six MiBench categories.
- Total of 111 functions.

Category	Program	Description
auto	bitcount	test processor bit manipulation abilities
network	dijkstra	Dijkstra's shortest path algorithm
telecomm	fft	fast fourier transform
consumer	jpeg	image compression / decompression
security	sha	secure hash algorithm
office	stringsearch	searches for given words in phrases

Florida State University

Outline

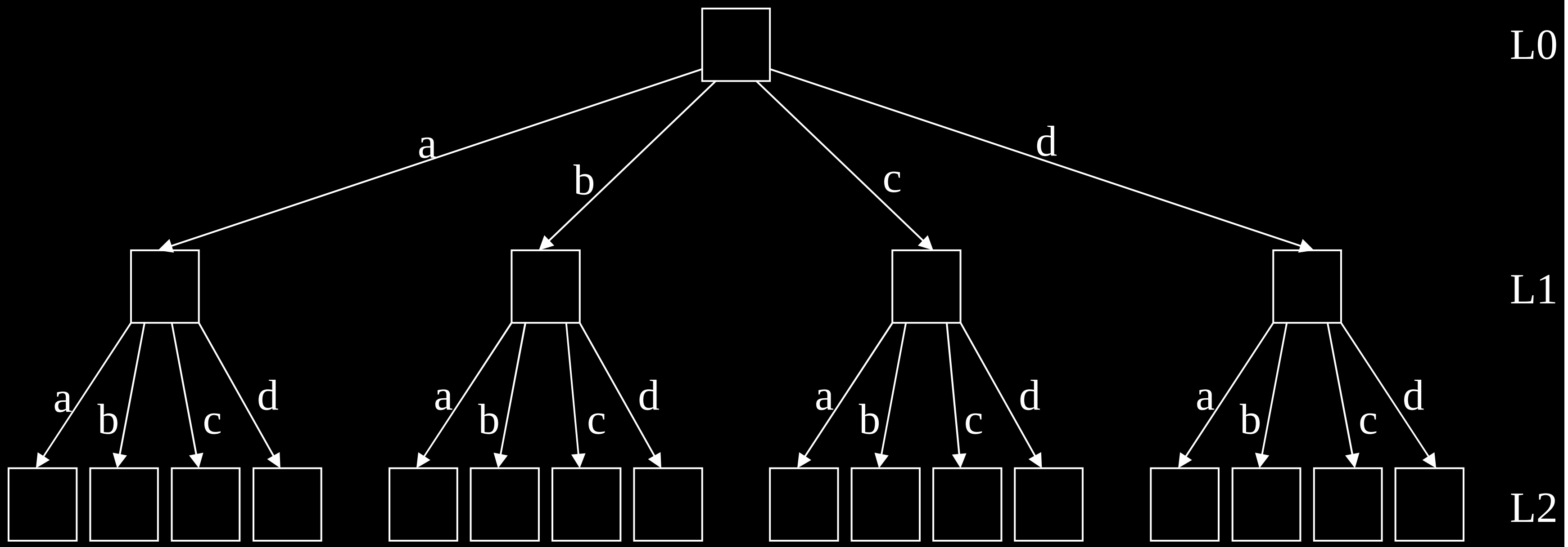
- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Correlation between dynamic frequency measures and processor cycles
- Genetic algorithm performance results
- Future work and conclusions

Exhaustive Phase Order Enumeration

- Exhaustive enumeration is difficult
  - compilers typically contain many different optimization phases
  - optimizations may be successful multiple times for each function / program
- On average, we would need to evaluate 15^{12} different phase orders per function.

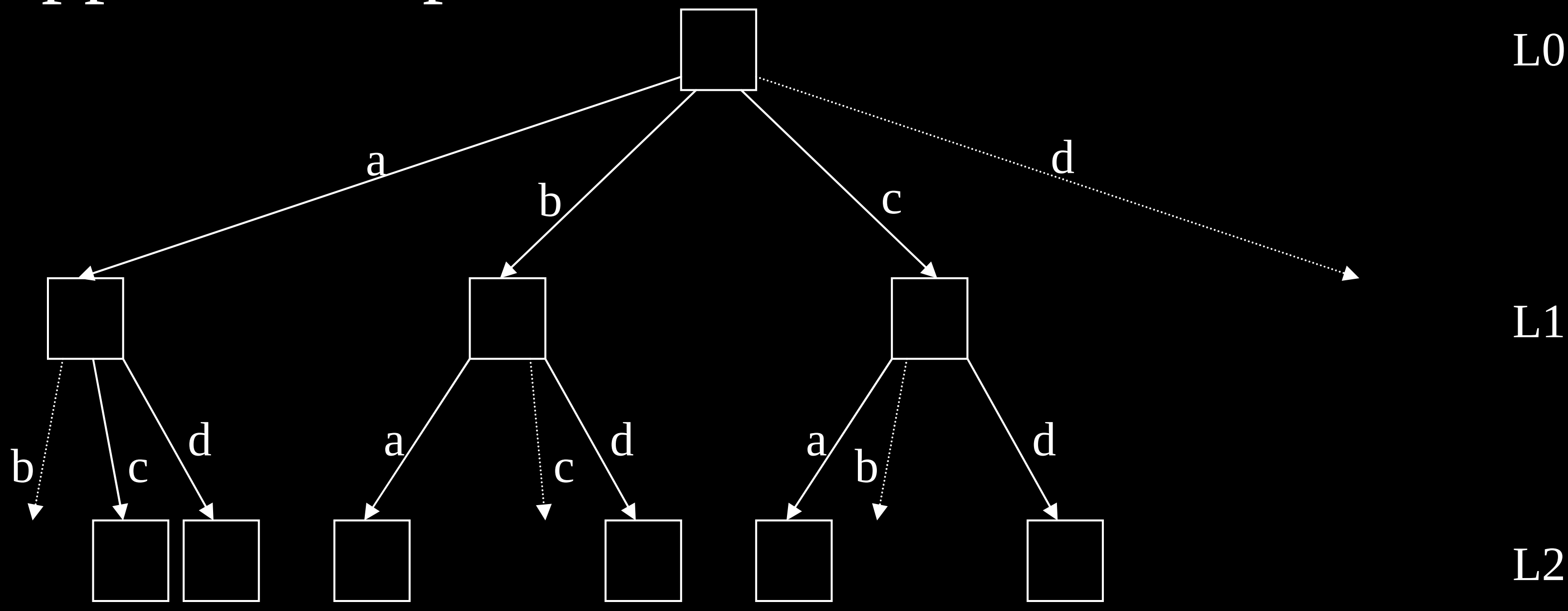
Naive Optimization Phase Order Space

- All combinations of optimization phase sequences are attempted.



Eliminating Dormant Phases

- Get feedback from the compiler indicating if any transformations were successfully applied in a phase.



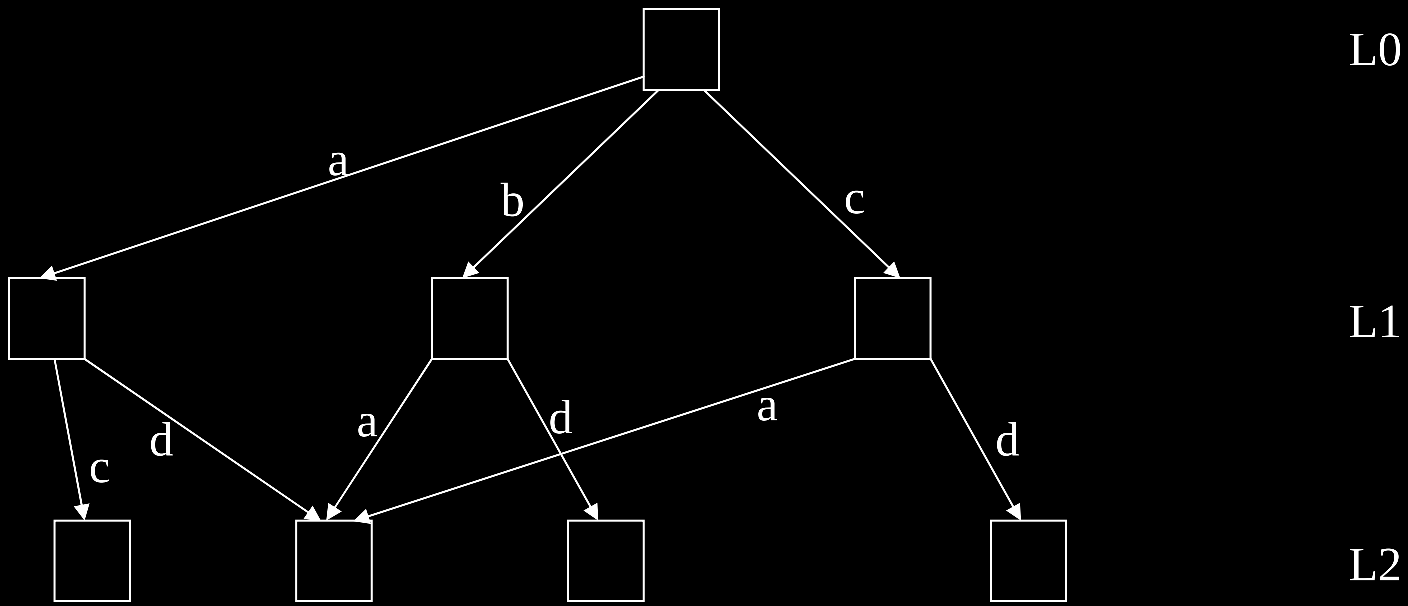
Florida State University

Identical / Equivalent Function Instances

- Some optimization phases are independent
  - example: branch chaining and register allocation
- Different phase sequences can produce the same code.
- Two function instances can be identical except for register numbers or basic block numbers used.

Resulting Search Space

- Merging equivalent function instances transforms the tree to a DAG.



Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Correlation between dynamic frequency measures and processor cycles
- Genetic algorithm performance results
- Future work and conclusions

Florida State University

Finding the Best Dynamic Function Instance

- On average, there were over 25,000 distinct function instances for each studied function.
- Executing all distinct function instances would be too time consuming.
- Many embedded development environments use simulation instead of direct execution.
- Use data obtained from a few executions to estimate the performance of all remaining function instances.

Quickly Obtaining Dynamic Frequency Measures

- Two different instances of the same function having identical control-flow graphs will execute each block the same number of times.
- Statically estimate the number of cycles required to execute each basic block.
- *dynamic frequency measure =*

  *Σ (static cycles * block frequency)*

Dynamic Frequency Statistics

Function	Insts.	CF	Leaf	% from optimal	
				Batch	Worst
AR_btbl...(b)	40	88	2	0.00	4.55
BW_btbl...(b)	56	198	4	0.00	4.00
bit_count.(b)	155	72	4	1.40	1.40
bit_shifter(b)	147	82	3	0.00	3.96
bitcount(b)	86	63	10	2.40	4.33
main(b)	92834	45	171	8.33	233.31
ntbl_bitcnt(b)	253	50	20	18.69	18.69
ntbl_bit…(b)	48	33	8	4.09	4.68
dequeue(d)	102	59	14	0.00	12.00
dijkstra(d)	86370	44	1168	0.04	51.12
enqueue(d)	570	40	9	0.20	4.49
main(d)	8566	30	143	4.29	75.32
….	….	….	….	….	….
average	**25362.6**	**27.5**	**182.8**	**4.60**	**47.64**

Outline

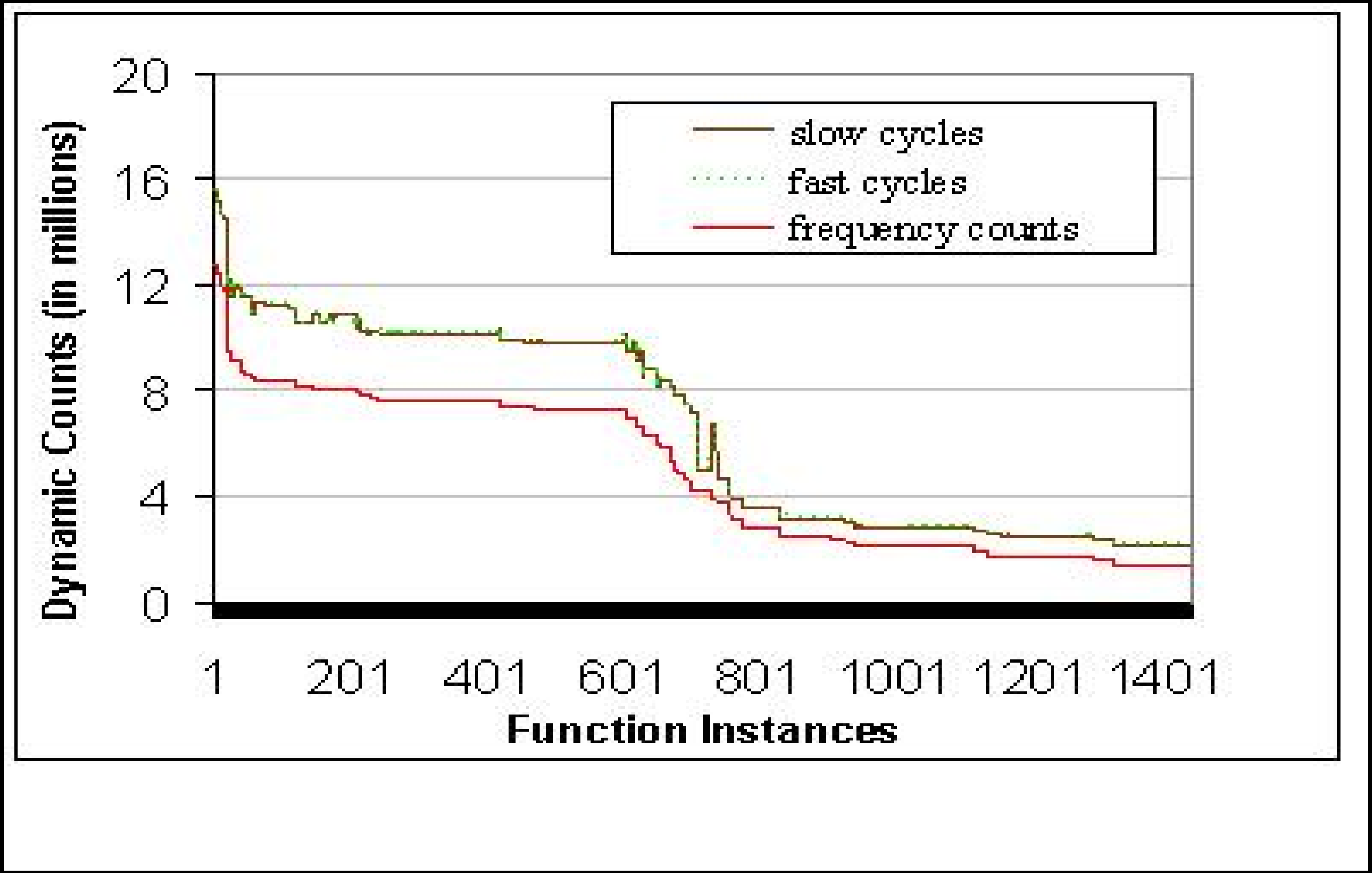
- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Correlation between dynamic frequency measures and processor cycles
- Genetic algorithm performance results
- Future work and conclusions

Florida State University

Cycle level Simulation

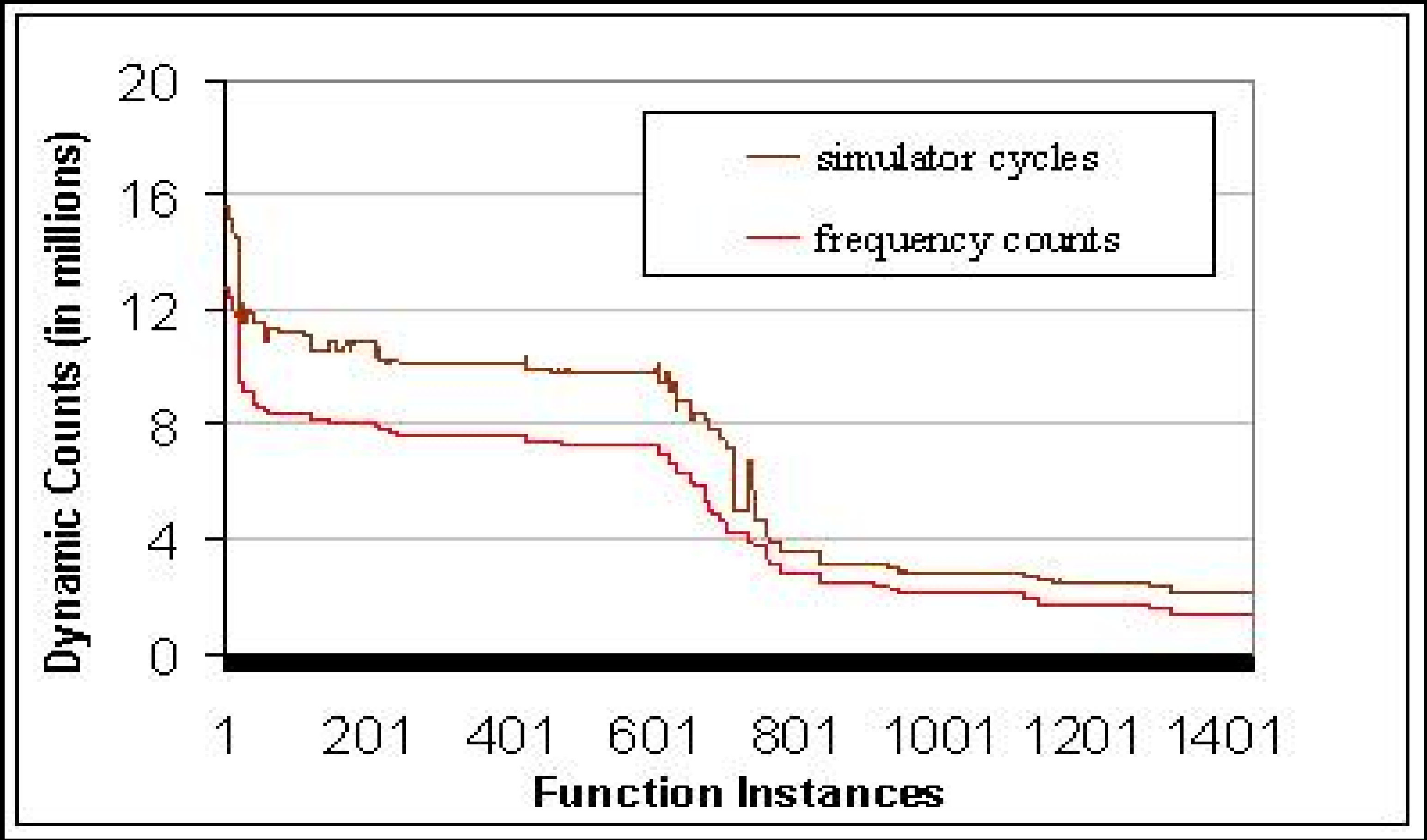
- *SimpleScalar* toolset includes several different simulators
  - *sim-uop* - functional simulator, relatively fast, provides only dynamic instruction counts
  - *sim-outorder* – cycle accurate simulator, much slower, also model microarchitecture
- Extended *sim-outorder* to switch to a functional mode when not in the function of interest.

Complete Function Correlation



Florida State University

Complete Function Correlation

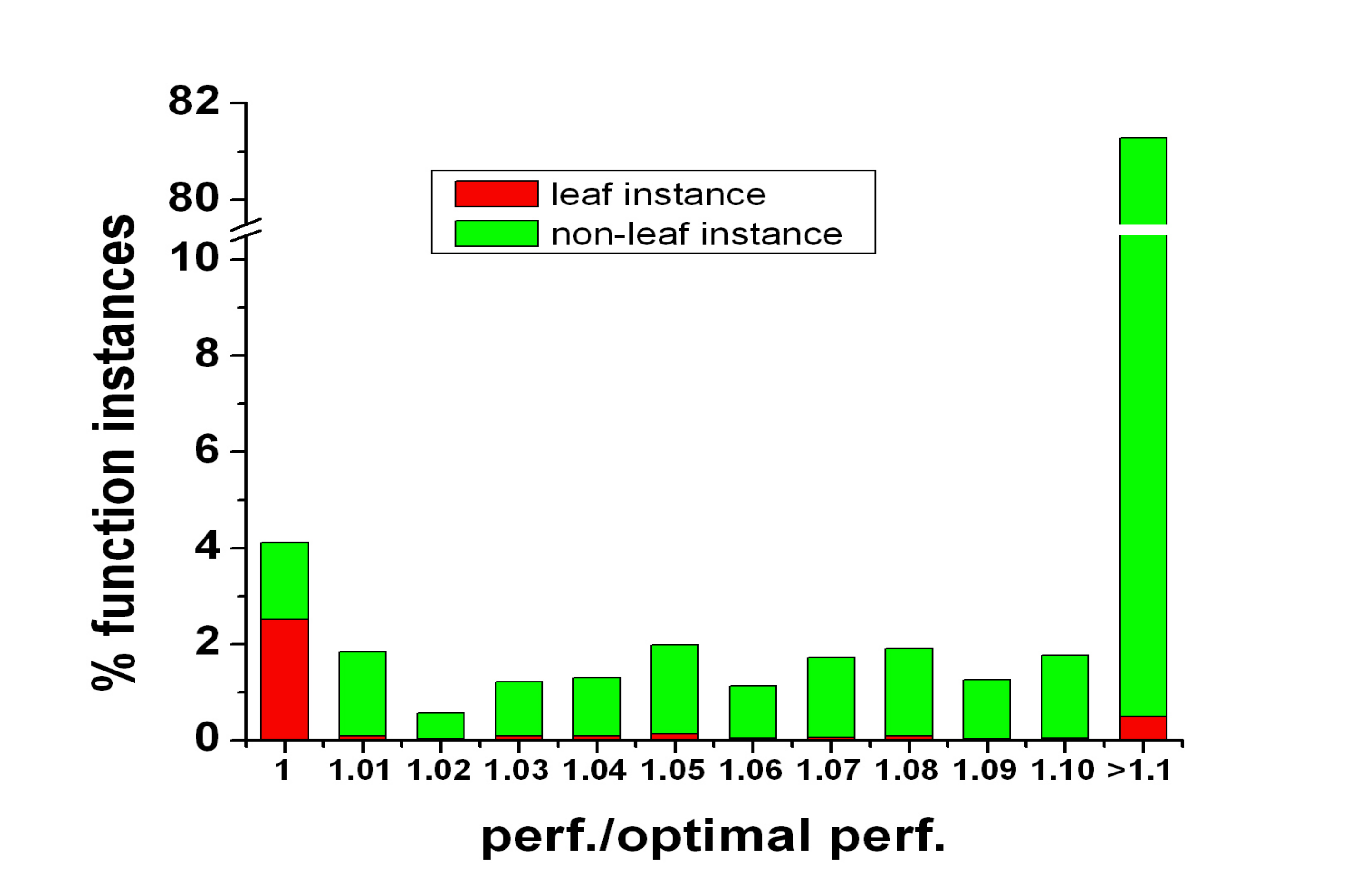


Florida State University

Leaf Function Correlation

- Leaf function instances are generated from optimization sequences when no additional phases can be successfully applied.
- On average there are only about 183 leaf function instances, as compared to over 25,000 total instances.
- Leaf function instances represent possible code that can be generated from an iterative compiler when the phase order is varied.

Leaf versus Nonleaf Performance



Leaf Function Correlation Statistics

- Pearson's correlation coefficient
  - $$Pcorr = \frac{\Sigma xy - (\Sigma x \Sigma y)/n}{sqrt((\Sigma x^2 - (\Sigma x)^2/n) * (\Sigma y^2 - (\Sigma y)^2/n))}$$
- $$Lcorr = \frac{\text{cycle count for best leaf}}{\text{cy. cnt for leaf with best dynamic freq count}}$$

Leaf Function Correlation Statistics (cont…)

Function	Pcorr	Lcorr 0%		Lcorr 1%	
		Ratio	Leaves	Ratio	Leaves
AR_btbl...(b)	1.00	1.00	1	1.00	1
BW_btbl...(b)	1.00	1.00	2	1.00	2
bit_count.(b)	1.00	1.00	2	1.00	2
bit_shifter(b)	1.00	1.00	2	1.00	2
bitcount(b)	0.89	0.92	1	0.92	1
main(b)	1.00	1.00	6	1.00	23
ntbl_bitcnt(b)	1.00	0.95	2	0.95	2
ntbl_bit…(b)	0.99	1.00	2	1.00	2
dequeue(d)	0.99	1.00	6	1.00	6
dijkstra(d)	1.00	0.97	4	1.00	269
enqueue(d)	1.00	1.00	2	1.00	4
main(d)	0.98	1.00	4	1.00	4
….	….	….	….	….	….
average	**0.96**	**0.98**	**4.38**	**0.996**	**21**

Florida State University

Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Correlation between dynamic frequency measures and processor cycles
- Genetic algorithm performance evaluation
- Future work and conclusions

Florida State University

Genetic Algorithm Properties

- *Genes* are phases, *chromosomes* are sequences.
- There are 20 chromosomes per generation.
- *Crossover* is used to replace 4 poorly performing chromosomes per generation.
- All, except the best sequence and the 4 newly generated sequences are subject to *mutation*.
- We modified our GA to use phase *enabling* and *disabling* relationships during the mutation phase of the GA.

Florida State University

GA Evaluation Results

Function	Original GA		Modified GA	
	Opt	Diff	Opt	Difff
AR_btbl...(b)	Y	0.00	Y	0.00
BW_btbl...(b)	Y	0.00	Y	0.00
bit_count.(b)	Y	0.00	Y	0.00
bit_shifter(b)	Y	0.00	Y	0.00
bitcount(b)	Y	0.00	Y	0.00
main(b)	Y	0.00	Y	0.00
ntbl_bitcnt(b)	N	6.55	Y	0.00
ntbl_bit…(b)	Y	0.00	Y	0.00
dequeue(d)	Y	0.00	Y	0.00
dijkstra(d)	Y	0.00	Y	0.00
enqueue(d)	Y	0.00	Y	0.00
main(d)	N	3.96	Y	0.00
….	….	….	….	….
average	**0.87**	**0.51**	**0.97**	**0.02**

Florida State University

Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Correlation between dynamic frequency measures and processor cycles
- Genetic algorithm performance evaluation
- Future work and conclusions

Florida State University

Future Work

- Find more equivalent performing function instances to further reduce the phase order space.
- Study effect of limiting scope of phases so that the most deeply nested loops of a function are optimized first.
- Improve conventional compilation speed and performance.

Florida State University

Conclusions

- We demonstrated how a near-optimal phase ordering can be obtained in a short period of time.
- We showed that our measure of *dynamic frequency counts* correlate extremely well to simulator cycles.
- We also showed how the enumerated space can be used to evaluate the effectiveness of heuristic phase order search algorithms.

Optimization Space Properties

- Phase ordering problem can be made more manageable by exploiting certain properties of the optimization search space
  - optimization phases might not apply any transformations
  - many optimization phases are independent
- Thus, many different orderings of optimization phases produce the same code.

Re-stating the Phase Ordering Problem

- Rather than considering all attempted phase sequences, the phase ordering problem can be addressed by enumerating all distinct *function instances* that can be produced by combination of optimization phases.
- We were able to exhaustively enumerate 109 out of 111 functions, in a few minutes for most.

Detecting Identical Function Instances

- Some optimization phases are independent
  - example: branch chaining & register allocation
- Different phase sequences can produce the same code

```
r[2] = 1;
r[3] = r[4] + r[2];

⇒instruction selection
r[3] = r[4] + 1;
```

```
r[2] = 1;
r[3] = r[4] + r[2];

⇒constant propagation
r[2] = 1;
r[3] = r[4] + 1;

⇒dead assignment elimination
r[3] = r[4] + 1;
```

Florida State University

VPO Optimization Phases

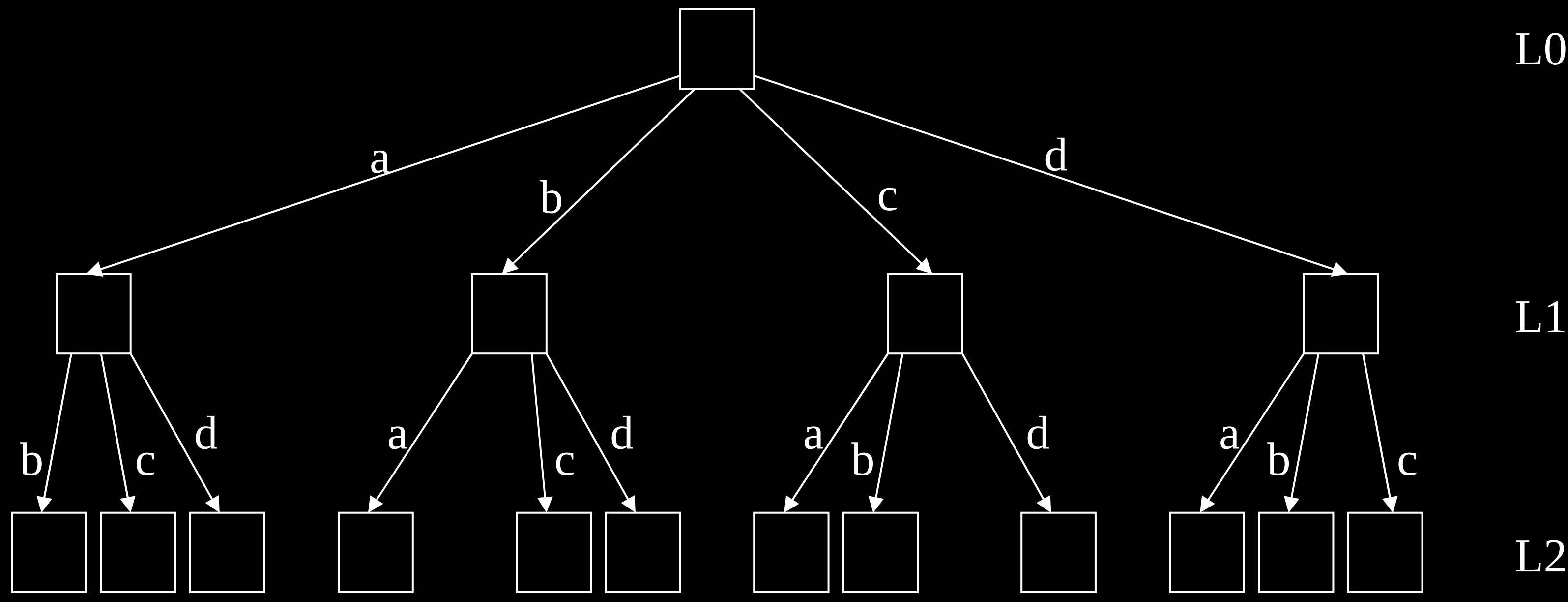
- Register assignment (assigning pseudo registers to hardware registers) is implicitly performed before the first phase that requires it.
- Some phases are applied after the sequence
  - fixing the entry and exit of the function to manage the run-time stack
  - exploiting predication on the ARM
  - performing instruction scheduling

VPO Optimization Phases

ID	Optimization Phase	ID	Optimization Phase
b	branch chaining	l	loop transformations
c	common subexpr. elim.	n	code abstraction
d	remv. unreachable code	o	eval. order determin.
g	loop unrolling	q	strength reduction
h	dead assignment elim.	r	reverse branches
i	block reordering	s	instruction selection
j	minimize loop jumps	u	remv. useless jumps
k	register allocation		

Eliminating Consecutively Applied Phases

- A phase just applied in our compiler cannot be immediately active again.



Florida State University

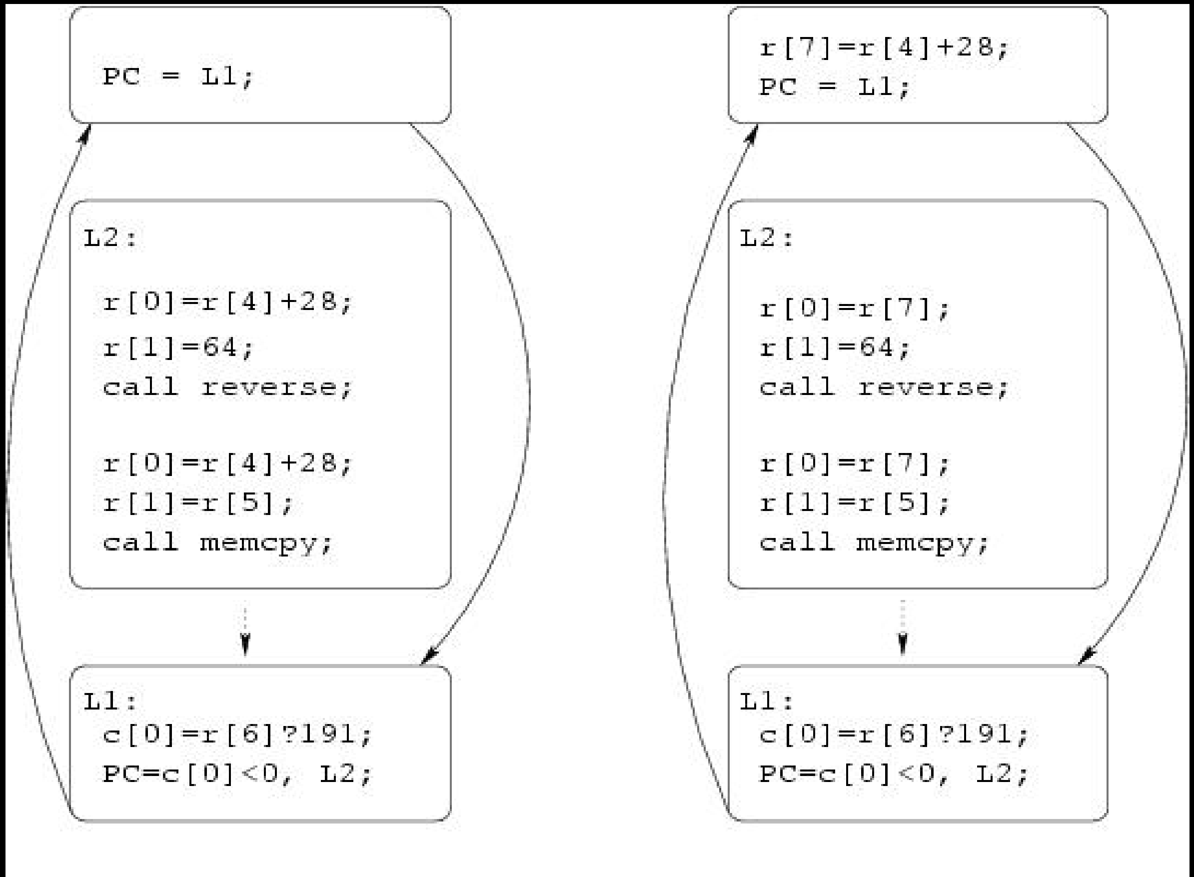
Detecting Equivalent Function Instances

```
sum = 0;
for (i = 0; i < 1000; i++ )
     sum += a [ i ];
```

Source Code

`r[10]=0;`	`r[11]=0;`	`r[32]=0;`
`r[12]=HI[a];`	`r[10]=HI[a];`	`r[33]=HI[a];`
`r[12]=r[12]+LO[a];`	`r[10]=r[10]+LO[a];`	`r[33]=r[33]+LO[a];`
`r[1]=r[12];`	`r[1]=r[10];`	`r[34]=r[33];`
`r[9]=4000+r[12];`	`r[9]=4000+r[10];`	`r[35]=4000+r[33];`
`L3`	`L5`	`L01`
`r[8]=M[r[1]];`	`r[8]=M[r[1]];`	`r[36]=M[r[34]];`
`r[10]=r[10]+r[8];`	`r[11]=r[11]+r[8];`	`r[32]=r[32]+r[36];`
`r[1]=r[1]+4;`	`r[1]=r[1]+4;`	`r[34]=r[34]+4;`
`IC=r[1]?r[9];`	`IC=r[1]?r[9];`	`IC=r[34]?r[35];`
`PC=IC<0,L3;`	`PC=IC<0,L5;`	`PC=IC<0,L01;`
Register Allocation before Code Motion	**Code Motion before Register Allocation**	**After Mapping Registers**

Case when No Leaf is Optimal



(a) Before Code Motion

(b) After Code Motion