



In Search of Near-Optimal Optimization Phase Orderings

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



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 (LCTES '06)



Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Analyzing the DAG of function instances
- Conclusions



Outline

- **Experimental framework**
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Analyzing the DAG of Function instances
- 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.



Benchmarks

- Used two programs from each of the six MiBench categories, 244 total functions.

Category	Program	Description
auto	bitcount	test processor bit manipulation abilities
	qsort	sort strings using the quicksort sorting algorithm
network	dijkstra	Dijkstra's shortest path algorithm
	patricia	construct patricia trie for IP traffic
telecomm	fft	fast fourier transform
	adpcm	compress 16-bit linear PCM samples to 4-bit
consumer	jpeg	image compression / decompression
	tiff2bw	convert color <i>tiff</i> image to b & w image
security	sha	secure hash algorithm
	blowfish	symmetric block cipher with variable length key
office	stringsearch	searches for given words in phrases
	ispell	fast spelling checker



Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Analyzing the DAG of function instances
- 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^{16} different phase orders per function.



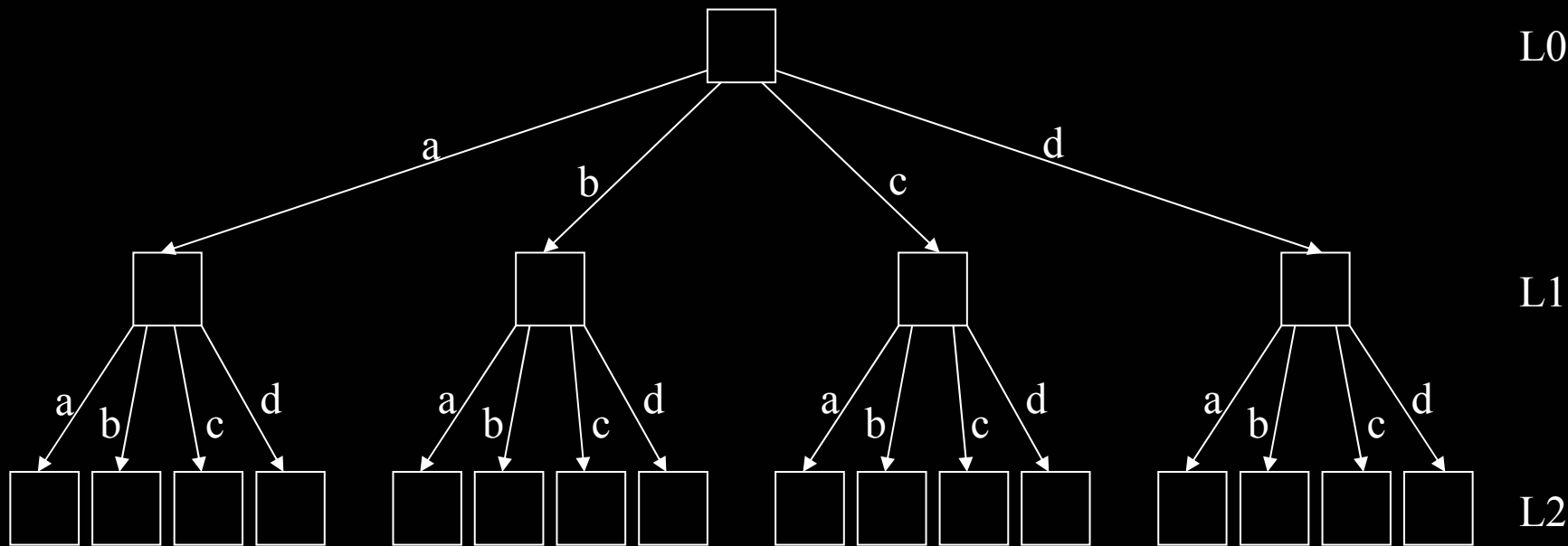
Re-stating the Phase Ordering Problem

- Many different orderings of optimization phases produce the same code.
- 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 any combination of optimization phases.



Naive Optimization Phase Order Space

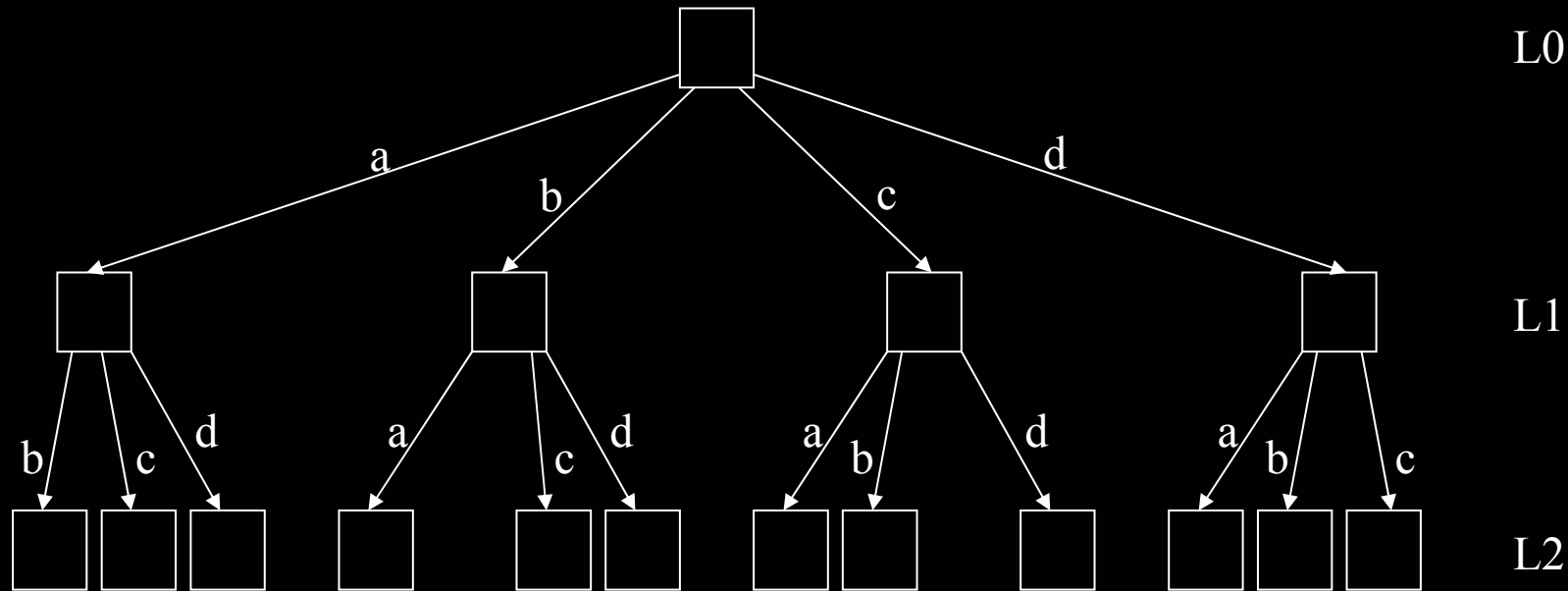
- All combinations of optimization phase sequences are attempted.





Eliminating Consecutively Applied Phases

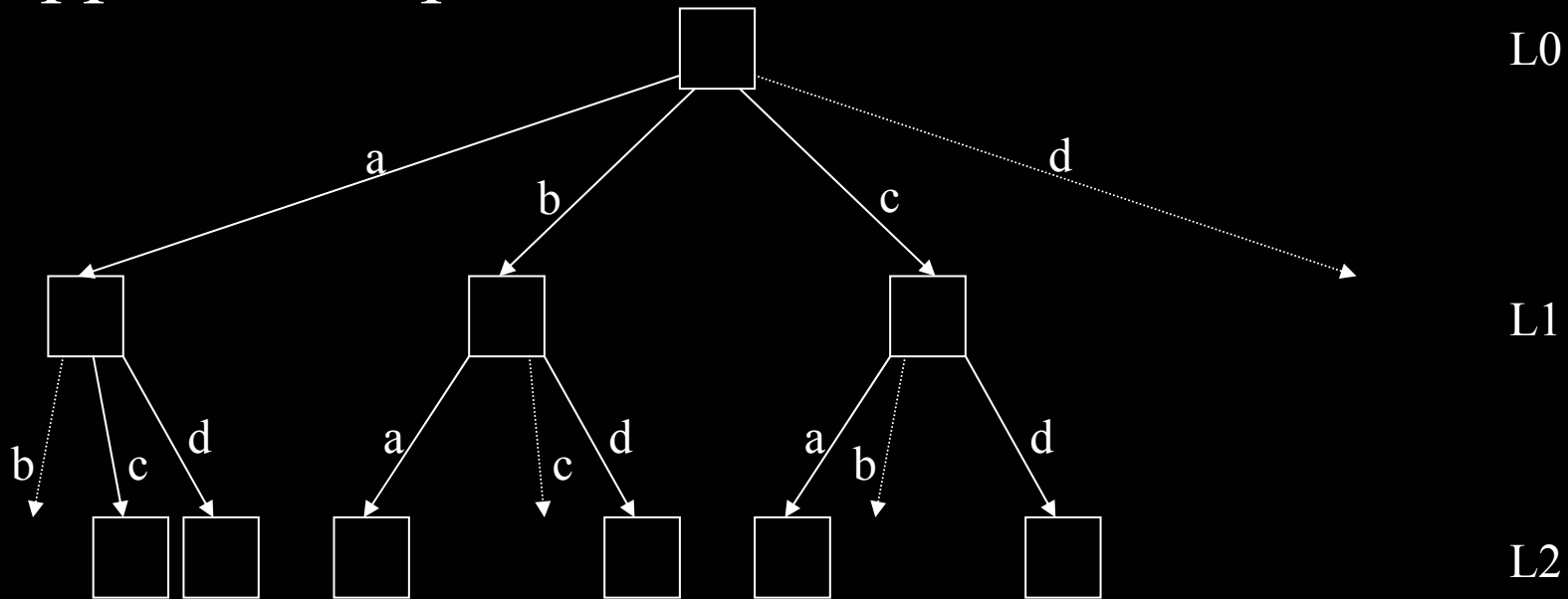
- A phase just applied in our compiler cannot be immediately active again.





Eliminating Dormant Phases

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





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;
```



Detecting Equivalent Function Instances

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

Source Code

```
r[10]=0;
r[12]=HI[a];
r[12]=r[12]+LO[a];
r[1]=r[12];
r[9]=4000+r[12];
```

L3

```
r[8]=M[r[1]];
r[10]=r[10]+r[8];
r[1]=r[1]+4;
IC=r[1]?r[9];
PC=IC<0, L3;
```

**Register Allocation
before Code Motion**

```
r[11]=0;
r[10]=HI[a];
r[10]=r[10]+LO[a];
r[1]=r[10];
r[9]=4000+r[10];
```

L5

```
r[8]=M[r[1]];
r[11]=r[11]+r[8];
r[1]=r[1]+4;
IC=r[1]?r[9];
PC=IC<0, L5;
```

**Code Motion before
Register Allocation**

```
r[32]=0;
r[33]=HI[a];
r[33]=r[33]+LO[a];
r[34]=r[33];
r[35]=4000+r[33];
```

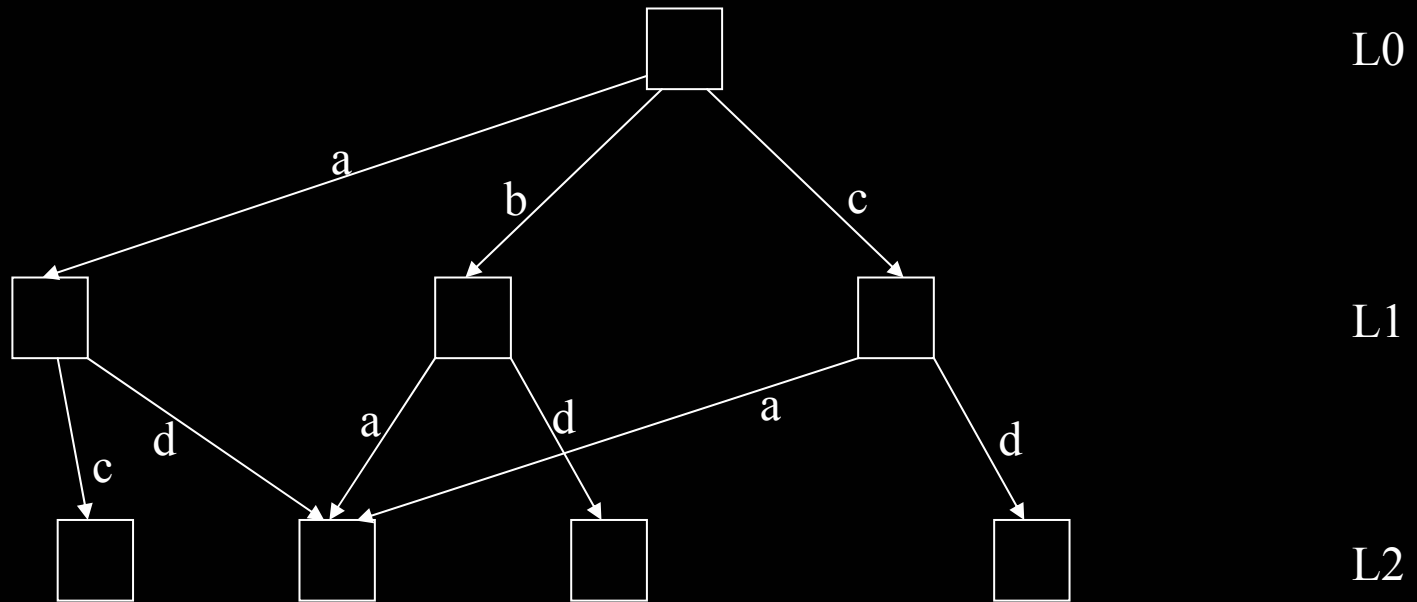
L01

```
r[36]=M[r[34]];
r[32]=r[32]+r[36];
r[34]=r[34]+4;
IC=r[34]?r[35];
PC=IC<0, L01;
```

**After Mapping
Registers**

Resulting Search Space

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





Techniques to Make Searches Faster

- Kept a copy of the program representation of the unoptimized function instance in memory to avoid repeated disk accesses.
- Enumerated the space in a depth-first order keeping in memory the program representation after each active phase to reduce number of phases applied.
- Reduced search time by at least a factor of 5 to 10.
- Able to completely enumerate 234 out of 244 functions in our benchmark suite, most in a few minutes, and the remainder in a few hours.



Search Space Static Statistics

Function	Insts	Blk	Loop	Instances	Len	CF	Leaves
main(t)	1,275	110	6	2,882,021	29	389	15,164
parse_sw...(j)	1,228	144	1	180,762	20	53	2,057
askmode(i)	942	84	3	232,453	24	108	475
skiptoword(i)	901	144	3	439,994	22	103	2,834
start_in...(j)	795	50	1	8,521	16	45	80
treeinit(i)	666	59	0	8,940	15	22	240
pfx_list...(i)	640	59	2	1,269,638	44	136	4,660
main(f)	624	35	5	2,789,903	33	122	4,214
sha_tran...(h)	541	25	6	548,812	32	98	5,262
initckch(i)	536	48	2	1,075,278	32	32	4,988
main(p)	483	26	1	14,510	15	10	178
pat_insert(p)	469	41	4	1,088,108	25	71	3,021
....
average	234.3	21.7	1.2	174,574.8	16.1	47.4	813.4



Outline

- Experimental framework
- Exhaustive phase order space enumeration
- **Accurately determining dynamic performance**
- Analyzing the DAG of function instances
- Conclusions



Finding the Best Dynamic Function Instance

- On average, there were 174,575 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.
- Is it possible to infer dynamic performance of one function instance from another?



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* =
$$\Sigma (\text{static cycles} * \text{block frequency})$$

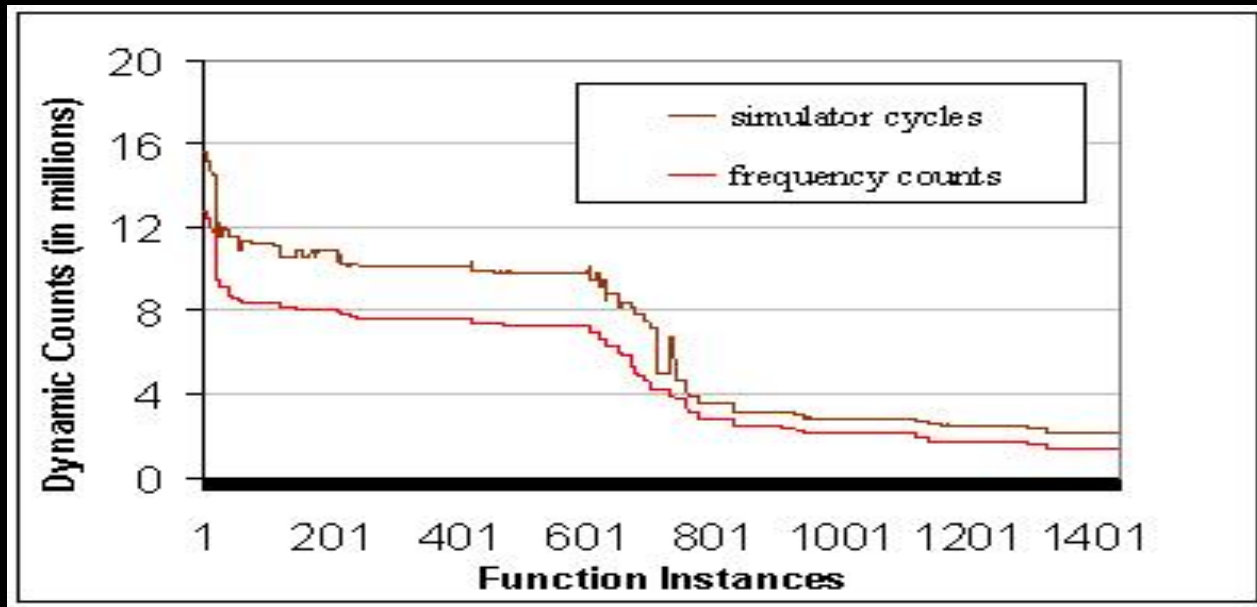


Dynamic Frequency Statistics

Function	Instances	CF	Leaves	% from optimal	
				Batch	Worst
main(t)	2,882,021	389	15,164	0.0	84.3
parse_sw...(j)	180,762	53	2,057	6.7	64.8
askmode(i)	232,453	108	475	8.4	56.2
skiptoword(i)	439,994	103	2,834	6.1	49.6
start_in...(j)	8,521	45	80	1.7	28.4
treeinit(i)	8,940	22	240	0.0	3.4
pfx_list...(i)	1,269,638	136	4,660	4.3	78.6
main(f)	2,789,903	122	4,214	7.5	46.1
sha_tran...(h)	548,812	98	5,262	9.6	133.4
initckch(i)	1,075,278	32	4,988	0.0	108.4
main(p)	14,510	10	178	7.7	13.1
pat_insert(p)	1,088,108	71	3,021	0.0	126.4
....
average	174,574.8	47.4	813.4	4.8	65.4



Correlation - Dynamic Frequency Measures & Processor Cycles



- For leaf function instances
 - Pearson's correlation coefficient – 0.96
 - performing 4.38 executions will get within 98% of optimal, and 21 within 99.6% of optimal



Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- **Analyzing the DAG of function instances**
- Conclusions



Analyzing the DAG of Function Instances

- Can analyze the DAG of function instances to learn properties about the optimization phase order space.
- Can be used to improve
 - Conventional compilation
 - Searches for effective optimization phase orderings



Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Analyzing the DAG of function instances
- **Conclusions**



Conclusions

- First work to show that the optimization phase order space can often be completely enumerated (at least for the phases in our compiler).
- Demonstrated how a near-optimal phase ordering can be obtained in a short period of time.
- Used phase interaction information from the enumerated phase order space to achieve a much faster compiler that still generated comparable code.