Eliminating Virtual Function Calls in C++ Programs

Gerald Aigner
Urs Hölzle
UC, Santa Barbara

ECOOP'96

http://www.cs.ucsb.edu/oocsb

Outline

Direct Cost of Virtual Function Calls

(cf. Karel Driesen's OOPSLA'96 paper)

Optimization and Its Effects

reduction of virtual fn calls
overall execution time
code size
I-cache misses

Conclusions

Implementation of Virtual Function Calls

Virtual Function Table (VFT)

a table of virtual function pointers

a VFT per class

a VFT pointer per object

multiple inheritance

VFT ← objectAddr[VFToffset]	x	
delta ← VFT[deltaOffset]		
selectorAddr← VFT[selectorOffset]	x	
objectAddr← objectAddr + delta		
call selectorAddr	x	x

cost * frequency

Direct Cost of Virtual Function Calls

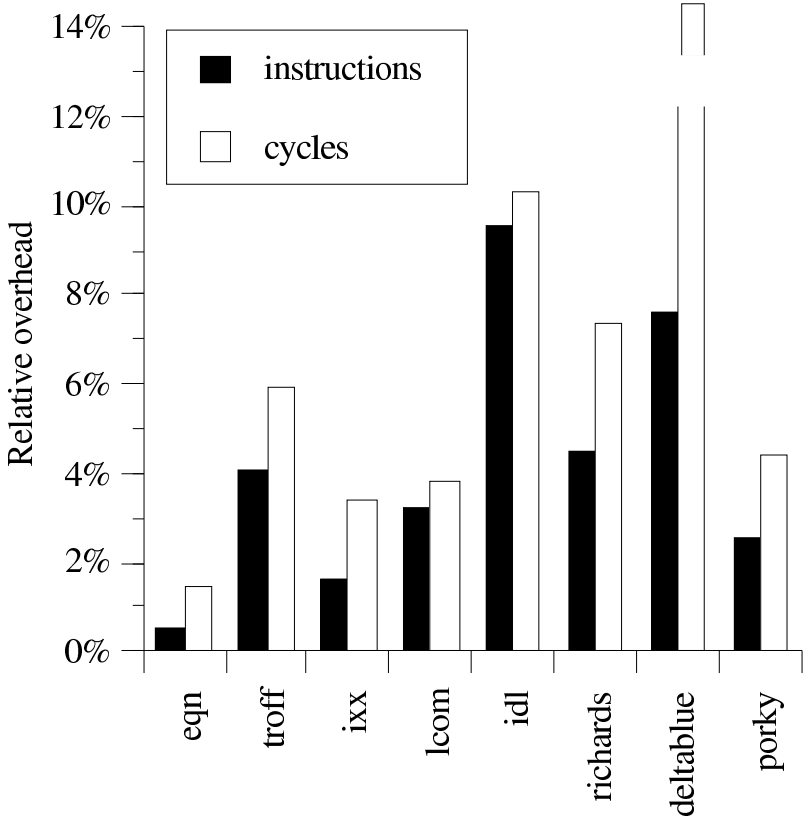


Figure 7. Direct cost of standard VFT dispatch (unmodified benchmarks)

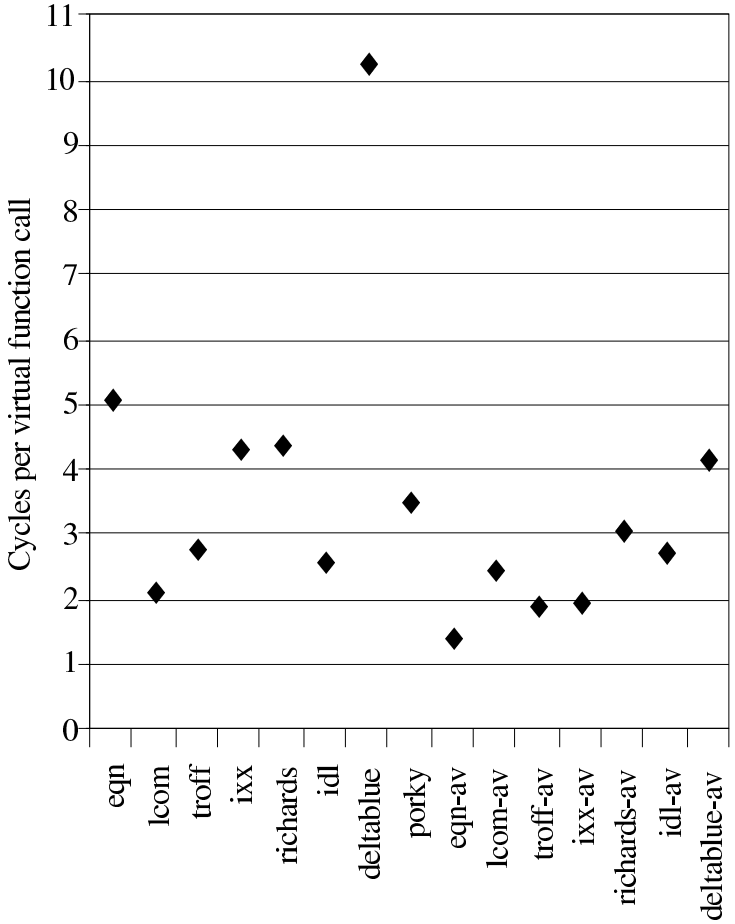


Figure 18. Cycles per dispatch

Summary

* a median of 5.2% additional cycles

* a median of 3.7% additional instr's

* cycle cost/instr cost varies substantially

* cost insensitive to branch penalty

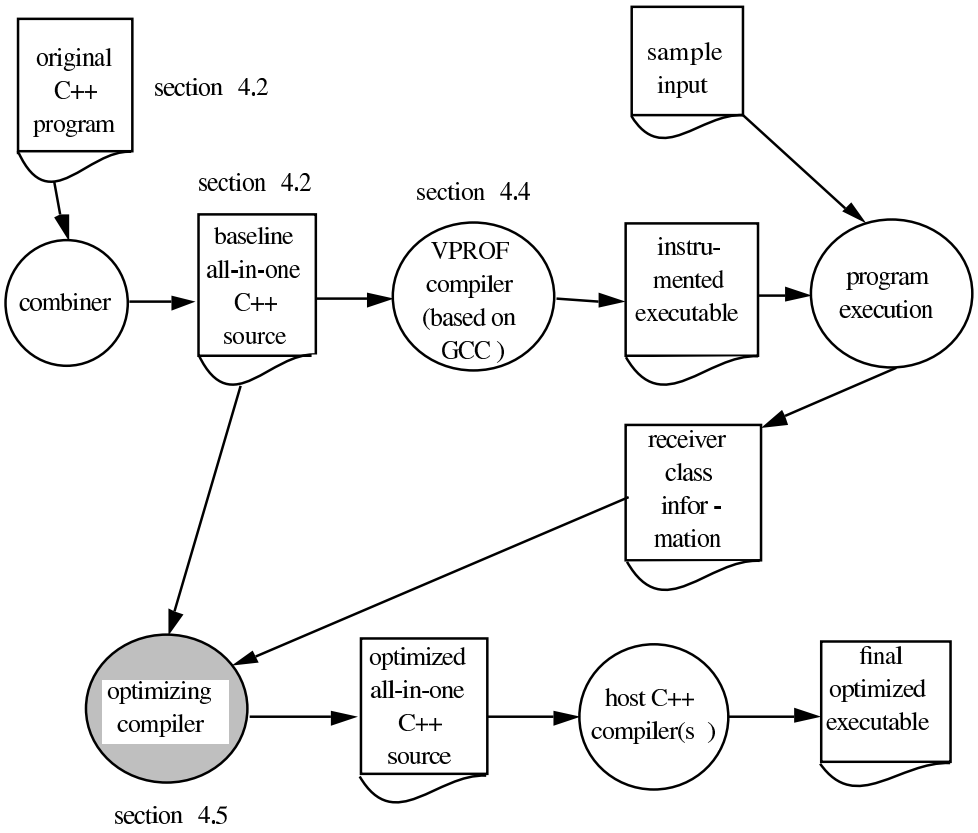
* cost proportional to branch misprediction
  --- a median of 65% prediction rate
  --- some can be predicted well 90%
  --- some simply cannot be predicted well

* cost sensitive to load latency, issue width

* **5-10% performance improvement for best dispatch mechanism**

* **relative dispatch overhead increases moderately in the future**

Implementation of Optimization



Figur e 1. Overview of optimization process

* source-to-source
* type test and "inline" annotation
* annotate " inline" (per call site) if
    --- receiver classes are "hot" (40%)
    --- call sites are "hot" (0.1%)

*** at most one case per send**
*** back-end always inlines**

Benchmarks

program		lines of code	
name	description	original	baseline
deltablue	incremental dataflow constraint solver	1,000	1,400
eqn	type-setting program for mathematical equations	8,300	10,800
idl	SunSoft's IDL compiler (version 1.3) using the demonstration back end which exercises the front end but produces no translated output.	13,900	25,900
ixx	IDL parser generating C++ stubs, distributed as part of the Fresco library (which is part of X11R6). Although it performs a function similar to IDL, the program was developed independently and is structured differently.	11,600	11,900
lcom	optimizing compiler for a hardware description language developed at the University of Guelph.	14,100	16,200
porky	back-end optimizer that is part of the Stanford SUIF compiler system	22,900	41,100
richards	simple operating system simulator	500	1,100
troff	GNU groff version 1.09, a batch-style text formatting program	19,200	21,500

Table 3. Benchmark programs

Name	Description	Style	K lines of code	# of indirect branches	instr. / indirect	cond. / indirect	virtual %	switch %	indirect %	1 target %	2 targets %	> 2 targets %	active branches	
													99 %	100 %
idl	IDL compiler[a]	OO	14	1,883,641	47	6	93.2	3.2	3.6	97.1	0.1	2.8	70	543
jhm	JHM[b] 6-12M	OO	15	6,000,000	47	5	93.6	1.2	5.2	58.7	1.4	39.9	34	155
self	Self-93 VM: 5-6M	OO	77	1,000,000	56	7	76.0	4.4	19.6	40.1	31.6	28.3	848	1855
xlisp	SPEC95	C	55	6,000,000	69	11	0.0	0.1	99.9	38.9	9.0	52.1	4	13
troff	GNU groff 1.09	OO	19	1,110,592	90	13	73.7	12.5	13.8	41.9	13.6	44.5	61	161
lcom	HDL[c] compiler	OO	14	1,737,751	97	10	63.2	36.8	0.0	33.5	54.0	12.5	87	328
AVG-100: instr/indirect < 100			24	2,955,331	68	9	66.6	9.7	23.7	51.7	18.3	30.0	184	509
perl	SPEC95	C	21	300,000	113	17	0.0	31.7	68.3	41.2	0.0	58.8	7	24
porky	scalar optimizer[d]	OO	23	5,392,890	138	19	70.6	23.8	5.6	15.6	8.1	76.3	89	285
ixx	IDL parser[e]	OO	12	212,035	139	18	46.5	52.2	1.3	37.1	6.4	56.5	91	203
edg	C++ front end	C	114	548,893	149	23	0.0	62.4	37.6	7.9	29.6	62.5	186	350
eqn	equation typesetter	OO	8	296,425	159	25	33.8	66.2	0.0	4.2	37.8	58.0	58	114
gcc	SPEC95	C	131	864,838	176	31	0.0	31.5	68.5	0.8	1.7	97.5	95	166
beta	BETA compiler	OO	72	1,005,995	188	23	0.0	2.3	97.7	18.7	28.1	53.2	135	376

The Effects of Compiler Optimization

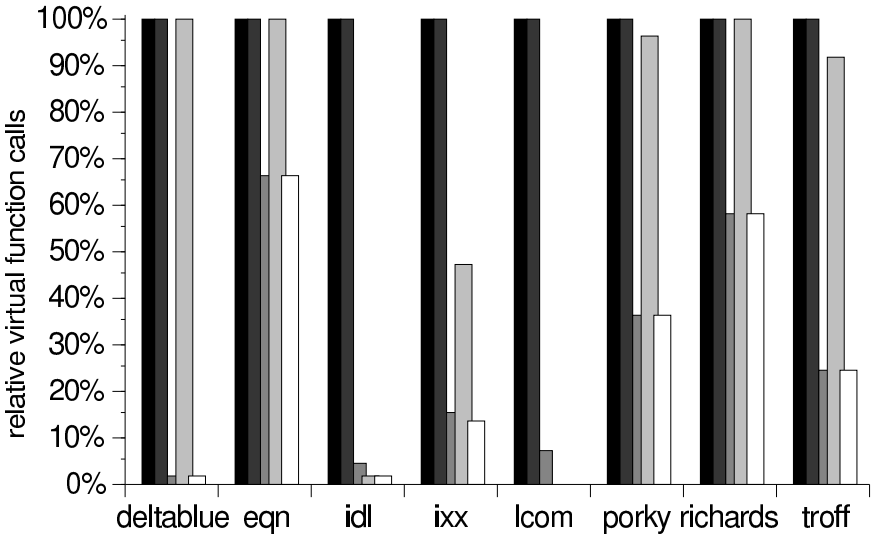


Figure 3. Virtual function calls

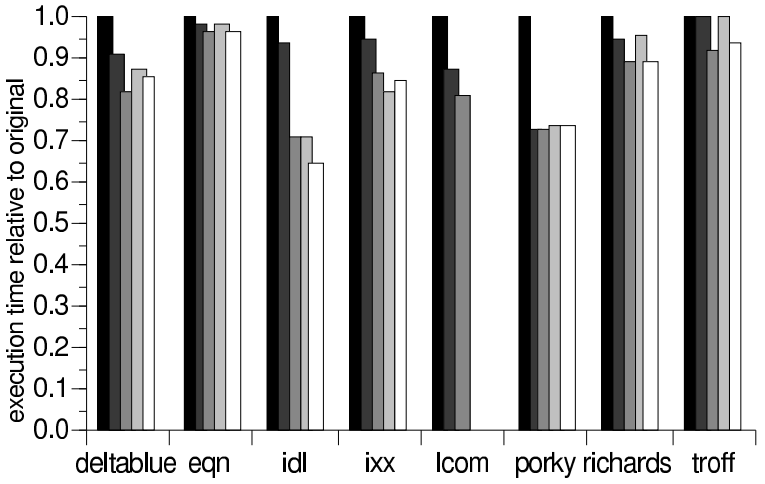


Figure 5. Execution time of optimized programs

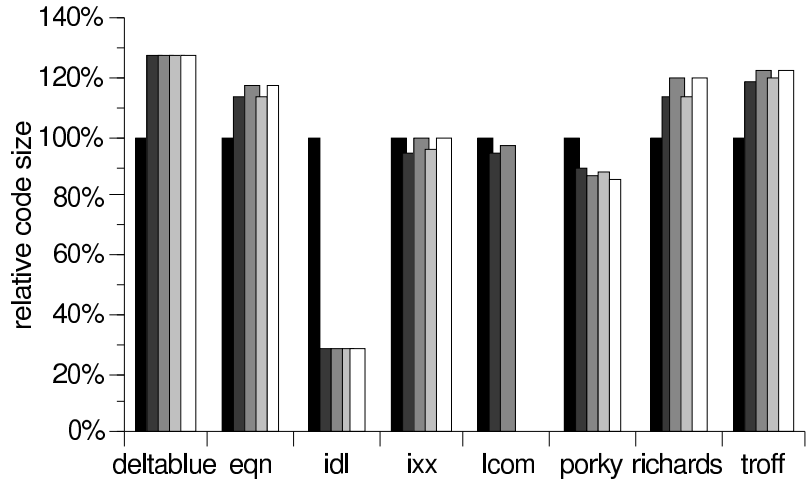


Figure 8. Code size

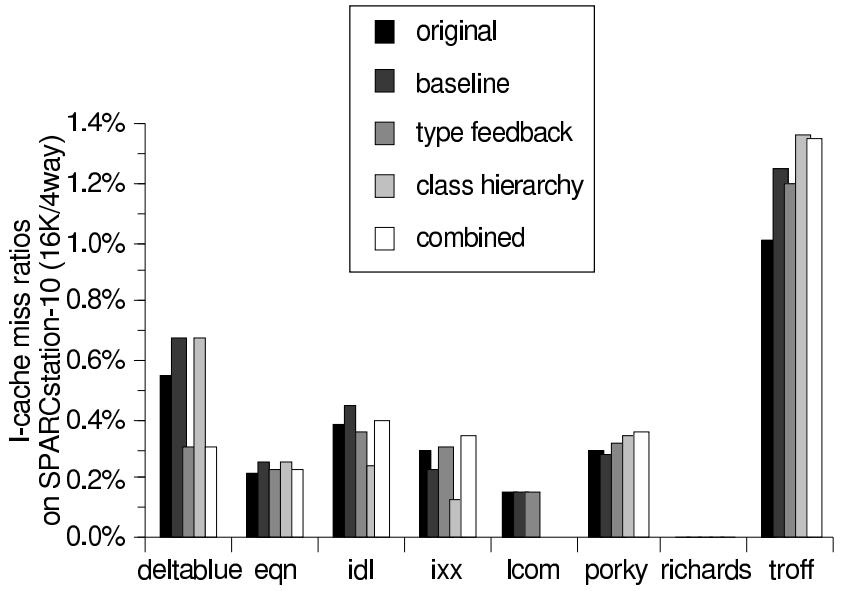


Figure 10. Instruction cache misses

Summary

* (virtual fn call reduction)
  --- TF/comb effective (80%), CHA less
  many programs do not have enough monomorphic calls to be optimized,
  but have few targets !!

* (performance impact)
  --- TF/comb best (18% speedup)
  --- baseline faster than original
  lower bound !

* (program size)
  --- TF/comb barely increase (8%)
  "hot" call sites work!

* (I-cache misses)
  --- increase small (10%)
  misses sensitive to code placement, insensitive to optimization

All-Virtual Version

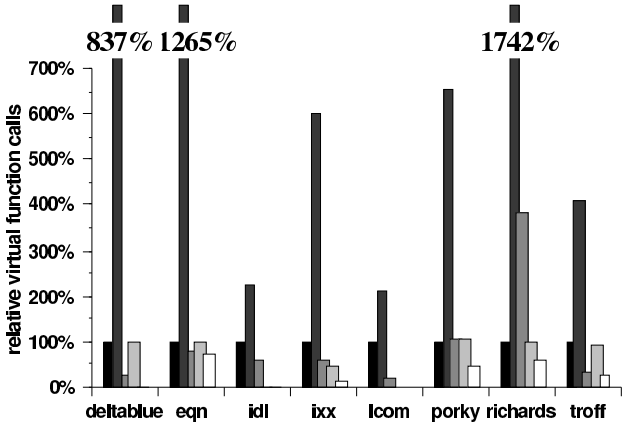


Figure 4. Virtual function calls of "allvirtual" programs

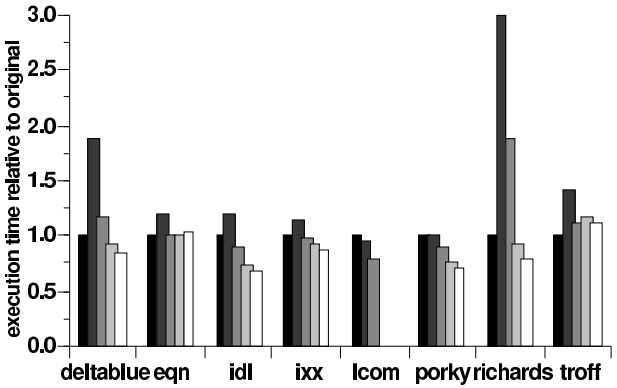


Figure 7. Execution time of "allvirtual" programs

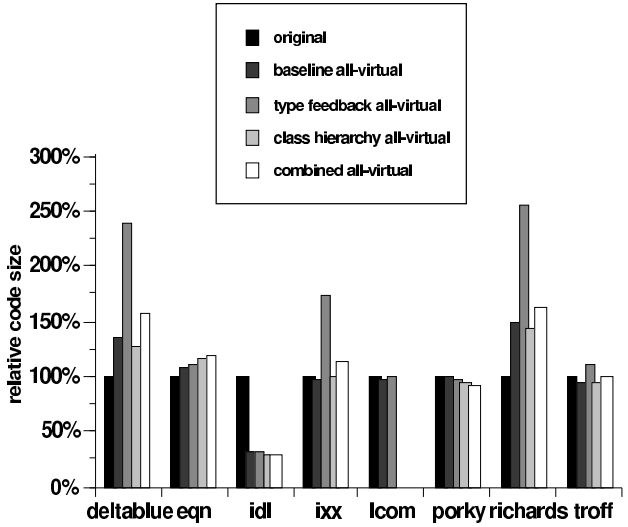


Figure 9. Code size of "allvirtual" programs

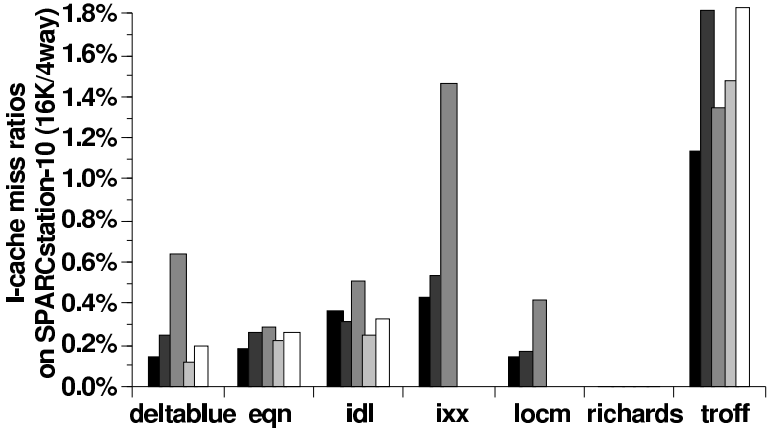


Figure 11. Instruction cache misses of "all-virtual" programs

Summary (All-Virtual vs Unmodified)

* (virtual fn call reduction)
  --- calls increases 5 times
  --- TF (88% vs 80%), **CHA (92% vs 4%)**

* (performance impact)
  --- 26% vs 18%

* (program size)
  --- 11% vs 8%

* (I-cache misses)
  --- similar

How hot is really "hot" ?

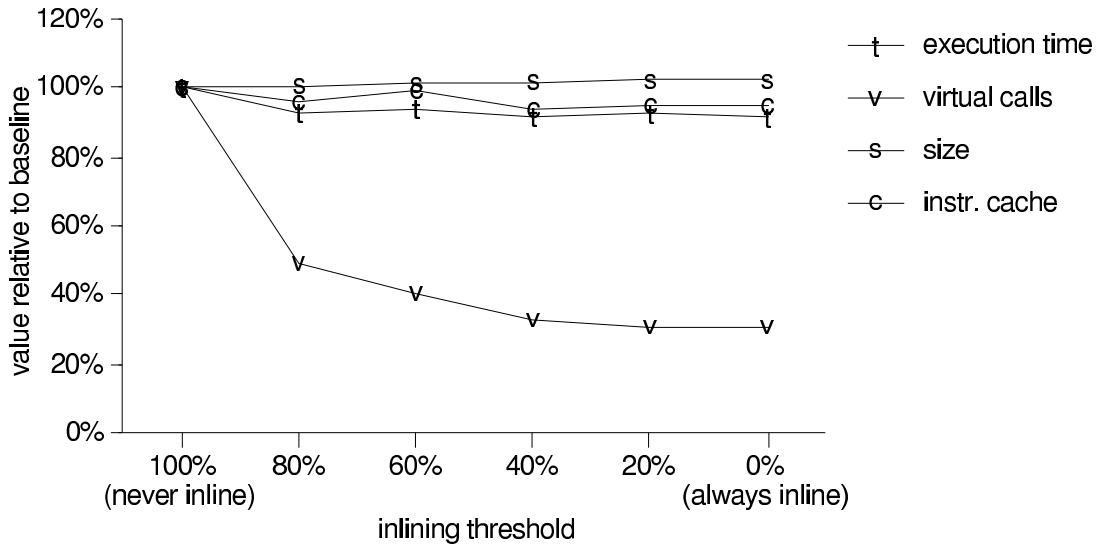


Figure 11. Performance characteristics as a function of inlining threshold (averages over all programs)

"hot" calls:

default is 40%

"hot" call sites:

0.1% : code size increase 11%
0.0%: 23%
(all-virtual) 144%

Conclusions

TF effective (18% speedup, 80% reduction)

CHA is ineffective in reducing v.f. calls

Inlining doesn’t increase code size too much

Results underestimate performance gains

Programmers use v.f. more liberally

Relative cost of v.f. calls will increase

V.F. call reduction becomes critical