

9.03.2000

# CODEGENERATOREN 185.117U (IN ENGLISH)

DAVID GREGG

INSTITUT FÜR COMPUTERSPRACHEN

dave@comlang.tuwien.ac.at

- EXACT TIMES
- EXAM
- NOTES
- PRACTICAL WORK

# TOPICS

## INSTRUCTION LEVEL PARALLELISM (ILP)

- WHAT IS ILP ?
- HOW MUCH ILP IS THERE ?
- SPECULATIVE EXECUTION
- ILP ARCHITECTURES
- INSTRUCTION SCHEDULING
- TRACE SCHEDULING
- SOFTWARE PIPELINING
- IA-64
- OTHER ILP ENHANCING OPTIMIZATIONS
- COMPILING FOR MULTISCALARS

## OTHER TOPICS IN CODE GENERATION

- CODE GENERATOR GENERATORS
- REGISTER ALLOCATION
- INTERPROCEDURAL REGISTER ALLOCATION
- STRUCTURE OF A CODE GENERATOR



# WHAT IS ILP?

PROGRAMS ARE COMPILED FROM HIGH LEVEL LANGUAGE (SUCH AS C, PASCAL) TO MACHINE INSTRUCTIONS.

GOAL OF ILP IS TO EXECUTE SEVERAL INSTRUCTIONS SIMULTANEOUSLY, TO MAKE THE PROGRAM RUN FASTER.

EXAMPLE:

2 WIDE ILP MACHINE

$xsq = xdir * xdir$

$ysq = ydir * ydir$

$xysumsq = xsq + ysq$

$tsq = tdir * tdir$

$vsq = vdir * vdir$

$tvsumsq = tsq + vsq$

$count = count + 1$

$xsq = xdir * xdir$      $ysq = ydir * ydir$

$xysumsq = xsq + ysq$      $tsq = tdir * tdir$

$vsq = vdir * vdir$      $count = count + 1$

$tvsumsq = tsq + vsq$

4 CYCLES INSTEAD OF 7

SOME INSTRUCTIONS ARE INDEPENDANT OF OTHERS.

WE DON'T ALWAYS HAVE TO WAIT FOR ALL PREVIOUS INSTRUCTIONS TO EXECUTE, BEFORE EXECUTING A GIVEN INSTRUCTION.

IF INDEPENDENT INSTRUCTIONS ARE EXECUTED SIMULTANEOUSLY, PROGRAM RUNS FASTER.



# How MUCH ILP IS THERE?

(IN TYPICAL PROGRAMS)

- BIG QUESTION IN LATE 1960's
- MEASURE HOW MUCH SPEEDUP FROM EXECUTING INDEPENDENT OPERATIONS SIMULTANEOUSLY.
- PRESUME WE HAVE INFINITE NUMBER OF PROCESSORS, REGISTERS, MEMORY, ETC.
- WHAT IS THE MOST, THE LIMIT OF ILP ONE CAN GET FROM A TYPICAL PROGRAM.

MANY LIMIT STUDIES WERE CARRIED OUT, AND THEY ALL FOUND (ROUGHLY) THE SAME RESULT :

LIMIT OF ILP SPEEDUP ABOUT 1.5-2.5,  
IE 50 - 150%.

**CONCLUSION :** EVEN WITH UNREALISTIC MACHINES WITH INFINITE RESOURCES, THERE IS VERY LITTLE ILP IN TYPICAL PROGRAMS.



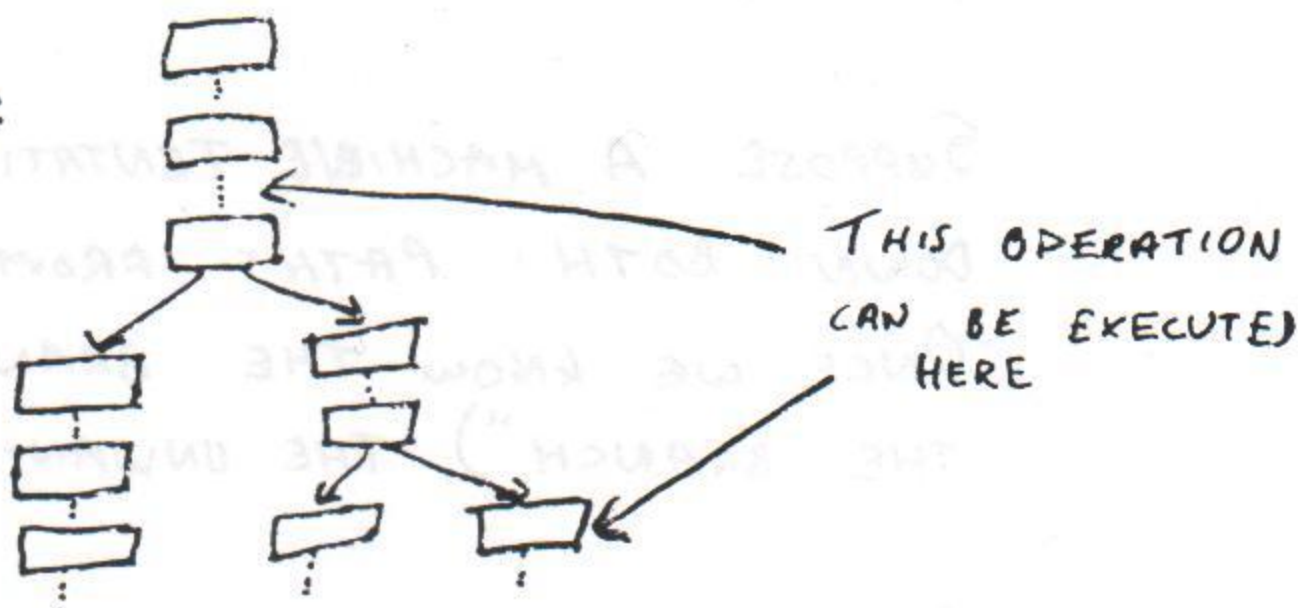
# THE BRANCH PROBLEM

ILP INVOLVES CHANGING THE ORDER IN WHICH INSTRUCTIONS ARE EXECUTED.

BUT: You CAN'T SAFELY MOVE AN INSTRUCTION ABOVE A BRANCH.

YOU DON'T KNOW WHICH WAY THE BRANCH  
WILL GO UNTIL IT IS EXECUTED.

EXAMPLE:



IN TYPICAL C INTEGER CODE, THERE IS A  
BRANCH ABOUT EVERY 5 INSTRUCTIONS.


IN TYPICAL FORTRAN SCIENTIFIC CODE, EVERY 8-9.

THEREFORE, THERE IS VERY LITTLE 1LP  
FROM OVERLAPPING INSTRUCTIONS BETWEEN  
BRANCHES.



# RISEMAN & FOSTER (1972)

WHAT IF WE COULD SOMEHOW IGNORE THE BRANCHES? WHAT IF EACH INSTRUCTION COULD EXECUTE AS SOON AS ALL ITS INPUTS ARE AVAILABLE?

POSSIBLE SPEEDUP OF  51

How could we EFFECTIVELY IGNORE OR "BYPASS" ALL BRANCHES?

SUPPOSE A MACHINE TENTATIVELY BEGINS EXECUTION DOWN BOTH PATHS FROM A CONDITIONAL BRANCH. ONCE WE KNOW THE BRANCH CONDITION ("RESOLVE THE BRANCH"), THE UNWANTED PATH IS DISCARDED.

A MACHINE THAT BYPASSES TWO BRANCHES WILL TENTATIVELY EXECUTE FOUR PATHS, AND THROW AWAY THREE.



A MACHINE THAT BYPASSES  $K$  BRANCHES WILL NEED TO KEEP  $2^K$  PATHS EXECUTING AT ALL TIMES.

A MACHINE THAT BYPASSES ALL BRANCHES HAS  $K = \infty$ .



# SPEEDUP FROM BYPASSING CONDITIONAL BRANCHES

RISEMAN &  
FOSTER  
(1972)

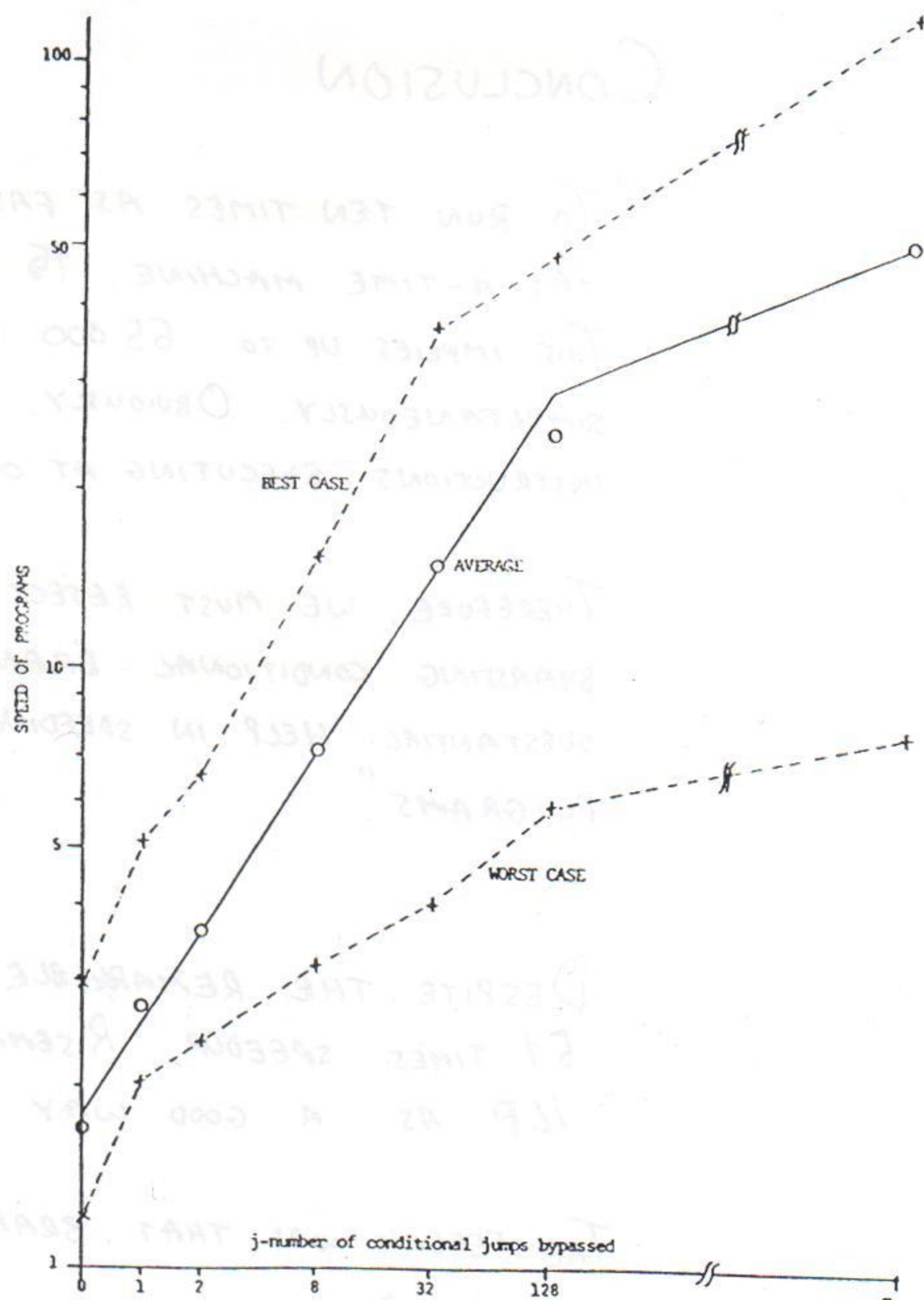


Fig. 1. Average speed as a function of number of conditional jumps that are bypassed—infinite stack machine.

TABLE II  
SPEEDUP OF SEVEN PROGRAMS (IN A MACHINE WITH AN INFINITE STACK) AS A FUNCTION OF THE NUMBER OF CONDITIONAL JUMPS PASSABLE

Program	0 jump	1 jump	2 jumps	8 jumps	32 jumps	128 jumps	∞ jumps
FORTRAN	1.40	2.03	2.38	3.14	4.02	5.86	32.4
COMPASS	1.22	2.10	2.74	4.28	5.55	7.17	27.2
CONCORDANCE	1.53	2.27	3.45	8.50	20.20	47.30	100.3
INTERIT	2.98	5.11	6.60	15.10	36.70	32.70	39.8
EIGENVALUE	1.72	2.40	3.34	6.64	14.20	22.40	29.7
DECALIZE	1.79	2.76	3.44	5.23	6.15	6.53	7.8
BPMOLD	1.43	2.38	3.32	7.56	16.80	43.50	120.5
AVERAGE	1.72	2.72	3.62	7.21	14.8	24.4	51.2

## CONCLUSION

"TO RUN TEN TIMES AS FAST AS A ONE-INSTRUCTION-AT-A-TIME MACHINE, 15 JUMPS MUST BE BYPASSED. THIS IMPLIES UP TO 65 000 PATHS BEING EXPLORED SIMULTANEOUSLY. OBVIOUSLY, A MACHINE WITH 65 000 INSTRUCTIONS EXECUTING AT ONCE IS A BIT IMPRACTICAL.

THEREFORE, WE MUST REJECT THE POSSIBILITY OF BYPASSING CONDITIONAL BRANCHES AS BEING OF SUBSTANTIAL HELP IN SPEEDING UP THE EXECUTION OF PROGRAMS."

DESPITE THE REMARKABLE DISCOVERY OF A 51 TIMES SPEEDUP, RISEMAN & FOSTER REJECTED ILP AS A GOOD WAY TO SPEED UP COMPUTERS.

THE FEELING WAS THAT BRANCHES MAKE ILP INFEASIBLE.



# BRANCH PREDICTION

PROBLEM: 65000 PATHS IS FAR TOO MANY TO CONSIDER

BUT: NOT ALL OF THOSE PATHS ARE EQUALLY LIKELY TO BE FOLLOWED. THE OUTCOME OF BRANCHES IS NOT RANDOM. IT IS HIGHLY PREDICTABLE.

How OFTEN DO BRANCHES TAKE THEIR MAJORITY DIRECTION?

RANGE	PERCENTAGE OF DYNAMIC BRANCHES WITHIN RANGE
99.5% +	40.0
95 - 99%	20.7
90 - 94%	13.8
85 - 89%	2.59
80 - 84%	4.36
75 - 79%	3.85
70 - 74%	4.94
65 - 69%	2.63
60 - 64%	2.67
55 - 59%	1.50
50 - 54%	3.13

SOURCE: HP EXPERIMENTS

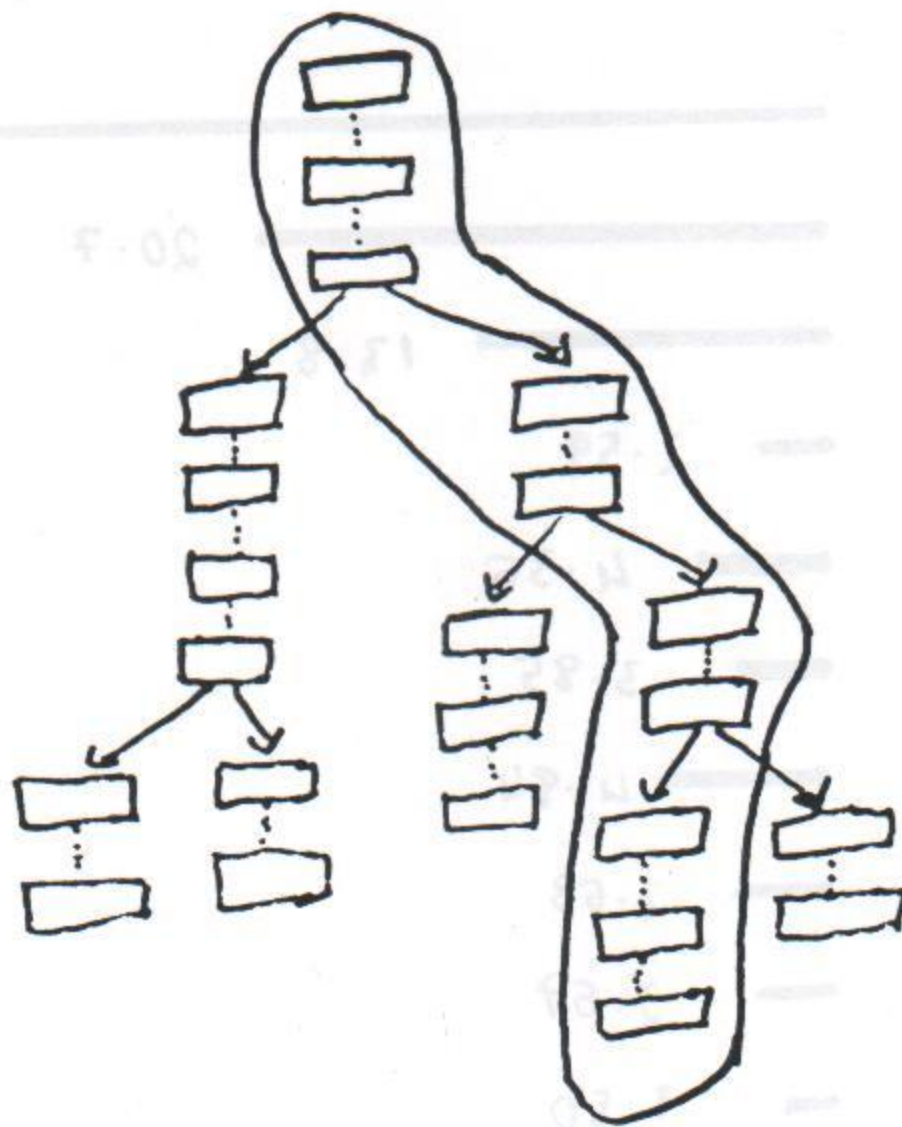


# SINGLE PATH

"TRACE SCHEDULING" (FISHER 1979)

- PREDICT THE DIRECTION OF EACH BRANCH
- FIND THE MOST COMMON PATH
- MACHINE FINDS ILP IN THE MOST COMMON PATH, IGNORING BRANCHES
- IF WE CHOSE THE RIGHT PATH, WE GET A BIG SPEEDUP
- MAKE SURE WE CAN RECOVER IF IT TURNS OUT WE CHOSE THE RIGHT PATH

GAMBLE ON THE MOST COMMON CASE





# WRONG PATH CHOSEN

PROBLEM 1: LOST ILP SLOWDOWN.

THERE IS VERY LITTLE THAT CAN BE DONE ABOUT THIS.

ON AVERAGE, OPTIMISING FOR THE MOST COMMON CASE WILL GIVE GOOD RESULTS, BUT IT MAY CAUSE THE LESS COMMON CASE TO BE EVEN SLOWER.

PROBLEM 2: THE REAL PATH NEEDS THE VALUE IN A VARIABLE X.

THE MOST COMMON PATH OVERWRITES X WITH A NEW VALUE.

THE STANDARD SOLUTION IN THIS CASE IS FOR THE COMMON PATH TO WRITE THE VALUE INTO A TEMPORARY VARIABLE.

THIS CAN BE COPIED TO X WHEN WE ARE SURE WE ARE ON THE RIGHT PATH.

PROBLEM 3: WE EXECUTE AN INSTRUCTION ON THE MOST COMMONLY FOLLOWED PATH.

WE EXECUTE IT BEFORE ALL PREVIOUS BRANCHES HAVE EXECUTED, SO WE ARE NOT YET SURE WE ARE ON THE RIGHT PATH.



THE INSTRUCTION CAUSES AN EXCEPTION.

- DIVIDE BY ZERO
- DEREFERENCING NULL POINTER
- INVALID ADDRESS
- FLOATING POINT EXCEPTION
- PAGE FAULT

WE DON'T KNOW WHETHER OR NOT THE INSTRUCTION SHOULD HAVE BEEN EXECUTED. WHAT SHOULD WE DO WITH THE EXCEPTION?

Option 1: DON'T ALLOW INSTRUCTIONS THAT CAN CAUSE EXCEPTIONS TO EXECUTE SPECULATIVELY.

- GREATLY REDUCES ILP
- MOST LOADS CANNOT EXECUTE SPECULATIVELY

Option 2: IGNORE THE EXCEPTION

- MULTIFLOW TRACE, OPTION ON ALPHA COMPILER
- IGNORING EXCEPTIONS IS EXTREMELY DANGEROUS. PLANES FALL FROM SKY, ETC

Option 3: SERVICE THE EXCEPTION, BUT GET THE EXCEPTION HANDLER TO CHECK IF IT IS VALID

- MANY PROGRAMS CAUSE HUNDREDS OF THOUSANDS OF FALSE EXCEPTIONS
- VERY SLOW



OPTION 4: RECORD THE FACT THAT AN EXCEPTION HAPPENED, BUT DON'T DO ANYTHING UNTIL ALL PREVIOUS BRANCHES HAVE RESOLVED.

THIS IS THE APPROACH USED BY MOST ILP MACHINES.

A SIMILAR PROBLEM ARISES WITH PAGE FAULTS. SHOULD WE SERVICE A SPECULATIVE PAGE FAULT?

ALMOST CERTAINLY NOT, SINCE TO SERVICE A PAGE FAULT, WE HAVE TO STOP THE MACHINE, SWITCH TO THE OPERATING SYSTEM, ETC.

BUT SHOULD WE SERVICE A VERY EXPENSIVE CACHE MISS?

- CACHE MISSES TAKE A LONG TIME, SO WE WANT TO START AS EARLY AS POSSIBLE
- TOO MUCH SPECULATION MAY "POLLUTE" THE CACHE WITH THINGS WE DON'T NEED.



16.03.2000

# RECENT LIMIT STUDY

(THEOBALD ET AL 1992)

LOOKED AT THE EFFECT OF VARIOUS VARIABLES ON THE ILP ACHIEVED.

## DATA DEPENDENCES AND RENAMING

TRUE DEPENDENCE

READ AFTER  
WRITE

1.  $A = b + c$
2.  $z = A * y$

ANTI DEPENDENCE

WRITE AFTER  
READ

1.  $z = A + c$
2.  $A = y * x$

OUTPUT DEPENDENCE  
(or False Dependence)

WRITE AFTER  
WRITE

1.  $A = b + c$
2.  $A = z * y$

TRUE DEPENDENCES COME FROM THE NEED TO PERFORM ONE COMPUTATION BEFORE ANOTHER. THEY ARE (MORE OR LESS) INHERANT IN THE ALGORITHM.

ANTI AND OUTPUT (OR FALSE) DEPENDENCES COME FROM OUR IMPERATIVE PROGRAMMING LANGUAGES. THE SAME VARIABLE IS ASSIGNED TO MORE THAN ONE.

FALSE DEPENDENCES CAN BE REMOVED FROM THE PROGRAM BY HAVING DIFFERENT COPIES OF VARIABLES THAT ARE ASSIGNED TO. RENAMING.



REGISTER RENAMING IS QUITE SIMPLE, IN COMPILER OR IN HARDWARE. THERE IS A SMALL FIXED NUMBER OF REGISTERS. EACH REGISTER CAN BE ACCESSED USING ONLY ONE NAME.

MEMORY RENAMING IS FAR MORE COMPLICATED.

- MEMORY IS FAR BIGGER
- MEMORY CONTAINS STRUCTURES, SUCH AS ARRAYS. HOW DO YOU RENAME ONE ELEMENT OF AN ARRAY WITHOUT COPYING THE WHOLE ARRAY?
- ALIASING - THE SAME MEMORY LOCATION MAY BE ACCESSED BY MORE THAN ONE NAME.

IT'S POSSIBLE TO IMAGINE HOW SOME LIMITED MEMORY RENAMING COULD BE DONE. BUT WITH THE EXCEPTION OF DATA FLOW MACHINES, I KNOW OF NO ALP MACHINES THAT ATTEMPT IT.

THEOBALD ET AL DIVIDE MEMORY RENAMING INTO STACK AND HEAP RENAMING. STACK RENAMING SEEMS MUCH MORE FEASIBLE, SINCE VARIABLES ON THE STACK ARE USUALLY SCALAR (NOT ARRAY) VALUES. ALIASING IS ALSO EASIER TO DETECT ON THE STACK.

RENAMING OF HEAP STRUCTURES, SUCH AS ARRAYS AND LINKED LISTS SEEMS MUCH MORE DIFFICULT.



## MEMORY DISAMBIGUATION

DO TWO NAMES REFER TO THE SAME MEMORY LOCATION?

E.G.        store  
              ⋮  
              load

CAN WE EXECUTE THE LOAD BEFORE THE STORE?  
IE DOES THE STORE WRITE TO THE MEMORY LOCATION THAT THE LOAD READS FROM?

### STATIC MEMORY DISAMBIGUATION

COMPILER TRIES TO DETERMINE WHETHER ADDRESSES ARE THE SAME

- NEVER
- SOMETIMES
- ALWAYS

### RUN TIME MEMORY DISAMBIGUATION

COMPARE THE ADDRESSES USED BY THE LOAD AND STORE AT RUN TIME.

MOST COMMONLY DONE IN HARDWARE.

BUT, COMPILER CAN ADD  
IF LOAD-ADDRESS  $\neq$  STORE-ADDRESS  
THEN.....  
----



# CONTROL BARRIERS

CONTROL FLOW INSTRUCTIONS AFFECT WHICH INSTRUCTIONS ARE EXECUTED NEXT. E.G. BRANCHES, INDIRECT JUMPS, CALLS, RETURNS, ETC.

- EARLY LIMIT STUDIES ASSUMED THAT CONDITIONAL BRANCHES ARE AN ABSOLUTE BARRIER.
- BRANCH PREDICTION CAN BE USED TO AVOID THE EFFECT OF BRANCHES, PROVIDED THE PREDICTION IS CORRECT.

THEOBALD ET AL USE A BRANCH PREDICTION TABLE THAT ASSUMES THE BRANCH ████████ WILL GO THE SAME WAY AS LAST TIME THE BRANCH WAS EXECUTED.

- THEY ALSO DO 'PROCEDURE SEPARATION'. IF TWO DIFFERENT PROCEDURES HAVE NO DATA DEPENDENCES BETWEEN THEM, BRANCH MISPREDICTIONS IN ONE DO NOT AFFECT THE EXECUTION OF THE OTHER.
- ORACLE — CAN PERFECTLY PREDICT ALL BRANCHES. THE RESULTS FOR THIS WILL BE THE SAME AS RISEMAN & FOSTER'S BRANCH BYPASSING MODEL.



# LIMITED INSTRUCTION WINDOW

REAL COMPUTERS OFTEN EXAMINE ONLY A LIMITED NUMBER OF INSTRUCTIONS WHEN LOOKING FOR ONES THAT ARE READY TO EXECUTE. THIS IS THE "PRE" WINDOW IN THE TABLE BELOW.

ANOTHER LIMITATION MIGHT BE TO HAVE A LIMIT ON THE NUMBER OF DIFFERENT CYCLES THAT AN OPERATION MAY BE PLACED IN. THIS IS THE "POST" WINDOW.

THE "PRE" WINDOW, OR SIMPLY WINDOW CORRESPONDS TO LIMITATIONS ON SUPERSCALAR ILP MACHINES.

## MACHINE MODELS EXAMINED

Model Name	Rename Regs.	Rename Memory	Branch Predict	Jump Predict	Proc. Sep.	Window Length	Unroll
Omniscient Oracle	yes	all	perfect	perfect	yes	$\infty$	no
Unrolling Omni. Or.	yes	all	perfect	perfect	yes	$\infty$	yes
Myopic Oracle	yes	all	perfect	perfect	yes	2048 pre	no
Short Oracle	yes	all	perfect	perfect	yes	2048 post	no
Tree Oracle	yes	stack	perfect	perfect	yes	$\infty$	no
Linear Oracle	yes	heap	perfect	perfect	yes	$\infty$	no
Frugal Oracle	yes	disambig.	perfect	perfect	yes	$\infty$	no
Speculative Multithreaded	yes	all	table	table	yes	$\infty$	no
Unrolling Spec. Mult.	yes	all	table	table	yes	$\infty$	yes
Multithreaded	yes	all	none	none	yes	$\infty$	no
Unrolling Multithreaded	yes	all	none	none	yes	$\infty$	yes
Smart Superscalar	yes	all	none	none	no	$\infty$	no
Stupid Superscalar	no	none	none	none	no	$\infty$	no

Table 1: Oracle and Machine Models



Source	Program	Description	Test Case	Useful Operations	Call Dep.	% of ops.		
						FP	Ld	St
DLX suite	tex	Text formatting	man.tex (1 p.)	15,184,459	19	.01	16	18
			draft.tex (11)	108,543,512	23	.04	15	8.0
Indust.	speech	Speech recognit.		551,267,528	12	4.5	14	2.8
SPEC test suite	li	Lisp interpreter	(queens 7)	204,921,097	75103	0	22	8.6
	espresso	Boolean minimization	bca.in	468,808,718	33	<.01	23	2.5
			cps.in	624,184,555	41	<.01	21	3.9
			ti.in	729,302,047	43	<.01	20	4.6
			tial.in	1,190,056,688	30	<.01	22	4.5
	eqntott	Truth-table gen.	int_pri_3.eqn	1,769,805,904	18	0	33	0.7
DLX suite	spice	Analog circuit simulation	small	26,026,482	17	10	30	9.9
			large	1,032,185,404	17	12	32	10
SPEC test suite	tomcatv	Mesh generation	N=33	46,850,826	17	16	46	13
			N=65	187,962,609	17	17	47	13
			N=129	751,099,835	17	17	47	13
			N=257	3,018,222,124	17	17	48	13
	doduc	Hydrocode simulation	tiny	103,371,553	14	14	36	10
			small	522,997,889	14	14	36	10
			ref	3,018,328,348	14	14	36	10
	fpppp	Quantum chemistry	NATOMS=4	276,896,598	14	19	43	10
			NATOMS=6	1,257,591,073	14	19	43	10

Table 2: Benchmarks and test cases used

Benchmark	Omniscient Oracle	Speculative Multithreaded	Multithreaded	Smart Superscalar	Stupid Superscalar
tex (man)	99.1	22.2	5.28	2.04	1.74
tex (draft)	192	12.2	3.04	2.08	1.66
speech	8,104	129	6.30	1.80	1.62
li (queens 7)	1,544	41.3	3.88	1.67	1.43
espresso (bca)	1,224	9.65	1.78	1.47	1.37
espresso (cps)	541	10.8	2.18	1.58	1.43
espresso (ti)	403	11.8	2.31	1.64	1.47
espresso (tial)	371	10.3	2.12	1.59	1.45
eqntott	43,298	300	1.66	1.46	1.43
spice (small)	76.5	27.2	8.18	3.53	2.00
spice (large)	62.8	22.4	6.11	3.90	2.04
tomcatv (N=33)	1,058	51.4	19.4	14.1	2.69
tomcatv (N=65)	2,108	52.6	20.1	18.3	2.76
tomcatv (N=129)	4,197	53.6	19.2	18.7	2.74
tomcatv (N=257)	8,418	54.2	19.2	19.1	2.74
doduc (tiny)	562	158	31.0	4.88	2.33
doduc (small)	617	174	32.8	4.92	2.33
doduc (ref)	621	202	33.2	4.92	2.33
fpppp (NATOMS=4)	4,977	297	52.7	30.1	2.88
fpppp (NATOMS=6)	11,296	311	54.1	30.9	2.88



Benchmark	Omniscient Oracle	Myopic Oracle	% of Omni	Short Oracle	% of Omni
tex (man)	99.1	34.0	34.3	39.4	39.8
tex (draft)	192	62.1	32.3	123	63.8
speech	8,104	90.5	1.12	146	1.80
li (queens 7)	1,544	64.0	4.14	92.6	6.00
espresso (bca)	1,224	35.1	2.86	103	8.42
espresso (cps)	541	59.6	11.0	312	57.7
espresso (ti)	403	66.7	16.5	317	78.6
espresso (tial)	371	55.3	14.9	256	68.9
eqntott	43,298	141	0.32	1,314	3.03
spice (small)	76.5	40.1	52.4	74.0	96.8
spice (large)	62.8	28.8	45.8	61.1	97.3
tomcatv (N=33)	1,058	159	15.0	1,058	100
tomcatv (N=65)	2,108	125	5.93	2,108	100
tomcatv (N=129)	4,197	101	2.40	4,197	100
tomcatv (N=257)	8,418	86.4	1.03	8,418	100
doduc (tiny)	562	103	18.3	550	98.0
doduc (small)	617	104	16.9	614	99.6
doduc (ref)	621	104	16.7	621	99.9
fpppp (NATOMS=4)	4,977	75.6	1.52	3,455	69.4
fpppp (NATOMS=6)	11,296	75.8	0.67	4,976	44.0

Table 4: The effects of finite window sizes

NOTE: TREE ORACLE → STACK RENAMING  
 LINEAR ORACLE → HEAP RENAMING

Benchmark	Omniscient Oracle	Tree Oracle	% of Omni	Linear Oracle	% of Omni	Frugal Oracle	% of Omni
tex (man)	99.1	32.1	32.4	79.2	79.9	32.0	32.3
tex (draft)	192	75.8	39.4	167	86.9	71.5	37.2
speech	8,104	136	1.68	57.1	0.70	53.6	0.66
li (queens 7)	1,544	58.5	3.79	336	21.8	58.3	3.78
espresso (bca)	1,224	126	10.3	906	74.0	124	10.1
espresso (cps)	541	86.9	16.1	523	96.6	86.9	16.0
espresso (ti)	403	84.5	21.0	403	100	84.5	21.0
espresso (tial)	371	138	37.1	370	99.8	138	37.1
eqntott	43,298	1,742	4.02	1,314	3.03	1,314	3.03
spice (small)	76.5	58.8	76.9	27.6	36.1	25.6	33.5
spice (large)	62.8	36.0	57.3	20.1	32.0	19.7	31.4
tomcatv (N=33)	1,058	423	40.0	161	15.2	161	15.2
tomcatv (N=65)	2,108	443	21.0	157	7.45	157	7.45
tomcatv (N=129)	4,197	451	10.7	155	3.69	155	3.69
tomcatv (N=257)	8,418	457	5.43	155	1.84	155	1.84
doduc (tiny)	562	554	98.6	25.8	4.59	25.8	4.58
doduc (small)	617	615	99.7	25.0	4.06	25.0	4.06
doduc (ref)	621	621	99.9	25.0	4.02	25.0	4.02
fpppp (NATOMS=4)	4,977	4,977	100	71.0	1.43	70.9	1.43
fpppp (NATOMS=6)	11,296	11,296	100	72.8	0.65	72.8	0.64

Table 5: The effect of frugal use of memory