

# Microprocessor Vulnerability, Continued

Vince Weaver  
CS717

February 20, 2005

## Source

- Mukherjee, Weaver, Emer, Reinhardt and Austin. “A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor”. MICRO 2003.

# Introduction

- Error rate of a device due to **Single Event Updates** depends on particle flux encountered, and on circuit characteristics.
- Particle flux depends on the environment. Denver has 3 to 5 times higher neutron flux than sea level.
- As sizes shrink, the smaller area makes events less likely, but the small size makes a strike more likely to cause an event. These effects apparently cancel for SRAM and latches, so at a given altitude error rate *per latch or bit* will remain roughly constant.
- Without error correction the exponential Moore's Law leads to exponential increase in error rates.

# Preventative Techniques

- Radiation-hardened circuit design
- Localized error detection and correction
- Architectural Redundancy
- Unfortunately the above all cause penalties in performance, power, die size, and design time.
- Inadequate protection may make a product useless, too much may make it uncompetitive.

# Architectural Vulnerability Factor

- Not all events actually change outcome of a program. Therefore estimates based on raw fault-rates overly pessimistic.
- Propose “Architectural Vulnerability Factor” (AVF). This is the probability that a fault in the processor will result in a visible error to the output.
- For example. AVF of branch predictor is 0%, but the AVF for committed PC is effectively 100%.
- Overall error rate is product of fault rate and AVF.
- This can be used to prioritize fault protection design.

# Architecturally Correct Execution

- The paper estimates AVF by tracking the subset of processor state bits required for “Architecturally Correct Execution” (ACE).
- A fault in an ACE-bit will cause a visible error in absence of error correction.
- Assuming cells have equal fault rates, AVF of an on-chip structure is the average of the AVF of the storage cells holding ACE bits.
- This metric is beneficial, because instead of tracking which bits matter or not all the time, you can calculate the average number of ACE bits (this assumes uniformly and randomly distributed faults, which cosmic rays tend to be).

## Error types

- Vendors typically express error likeliness via Mean Time Before Failures (MTBF)
- Errors further subclassified as undetected or detected. Undetected referred to as Silent Data Corruption (SDC), Detected referred to in this paper as Detected Unrecoverable Errors (DUE).
- Note: Detected *recoverable* errors are not errors at all.

## MTBF Examples

- For Power4, IBM aims for 1000 years system MTBF for SDC, 25 years for DUE resulting in system crash, 10 years for DUE resulting in application crash.
- Note that processor MTBF must be significantly higher than system MTBF especially on multi-processor systems.

## Paper Focus

- This paper focuses on the issue of Silent Data Corruption.
- Adding error correction converts SDC errors to DUE errors.

## FIT metric

- Another metric is FIT (Error In Time? Maybe they mean Failure In Time)
- FIT is inversely related to MTBF.
- One FIT is one failure in a billion hours.
- 1000 years MTBF equals 114 FIT ( $\frac{10^9}{24 \times 365 \times 1000}$ )
- A zero error rate corresponds to 0 FIT and infinite MTBF.
- Designers usually work with FIT because it is additive, unlike MTBF.

## Calculating FIT for a Chip

- First designers use complex models to compute FIT for each device (RAM, latches, logic-gates).
- Effective FIT rate is product of raw circuit FIT and the Vulnerability Factor (the probability a fault will result in an observable error)
- Overall FIT rate is determined by adding up the effective FIT rate of all the structures on the chip.

## Typical FIT rates

- Current predictions show FIT rate for latches and SRAM cells vary between 0.001-0.01 FIT/bit at sea level.
- This FIT rate is predicted to stay the same for the next few generations unless CPUs drastically reduce supply voltage.
- Logic gates contribute a negligible amount to FIT compared to the contribution of latches/SRAM
- This paper only looks at the FIT rate due to latches/SRAM.

# Vulnerability Factors

- Effective FIT/bit rate influenced by many vulnerability factors (also known as *derating factors* or *soft error sensitivity factors*)
- Example: when a latch is accepting data, a strike on a stored bit may not cause an error. If latch is accepting data 50% of the time, this results in a *timing vulnerability factor* of 50%. For simplicity paper assumes timing vulnerability factor already incorporated in raw device fault rate.
- Architectural Vulnerability Factor (AVF) expresses the probability a visible system error will occur given a bit flip. Prior studies show 1%-10% AVF for latches, 0%-100% across the whole range of architectural state bits.

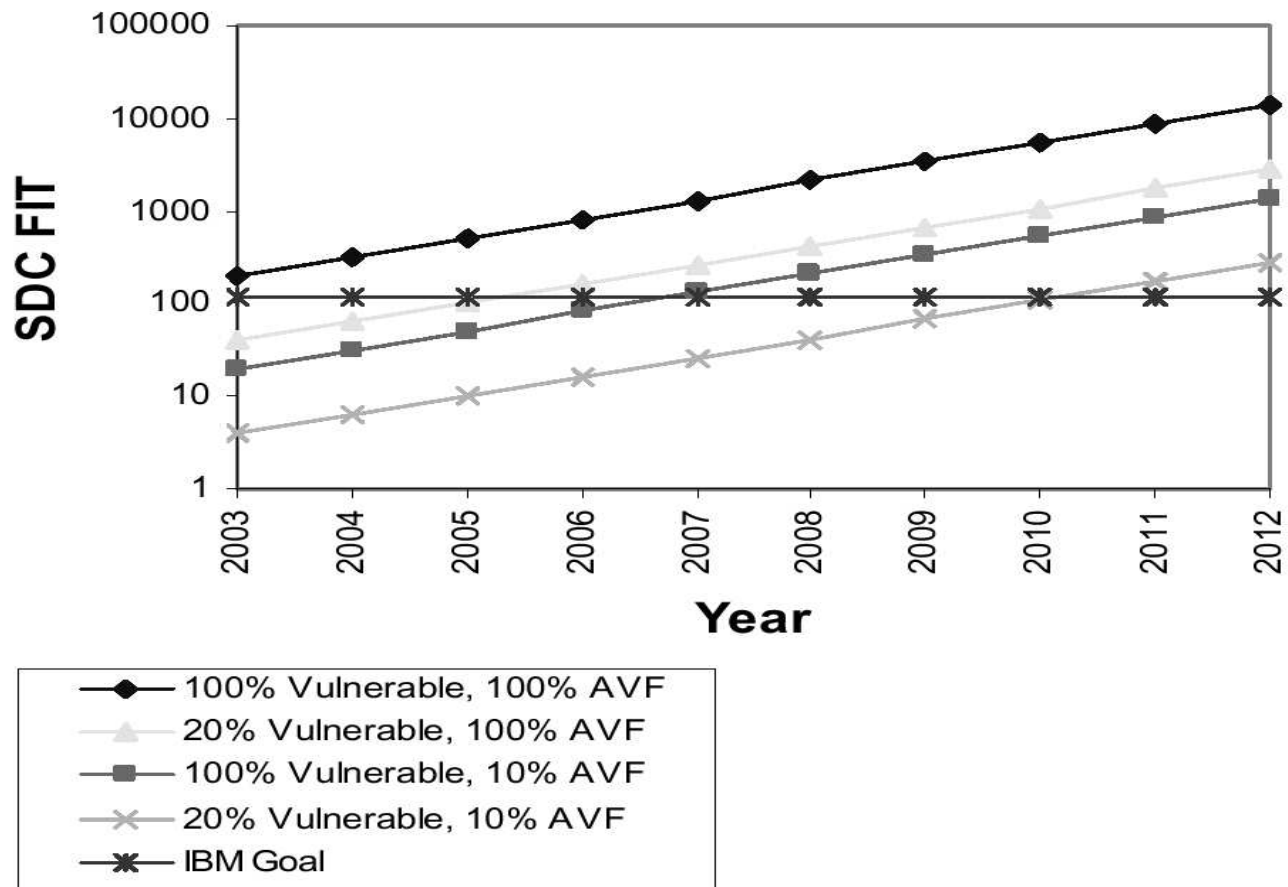


Figure 1: Impact of AVF on SDC FIT

# Determining ACE

- Determining which bits lead to observable error is tricky.
- If all the user sees is screen output, as long as that output is right it is correct.
- If program is run under debugger, that's a different story as formerly unobservable behavior is made visible.
- Under multi-processor systems, different results can occur due to races. This paper only investigates a uniprocessor system.
- For this paper, consider visible behavior as values that make it to I/O device. Don't really track it that far, but they do track past register and memory read/writes.
- Want conservative estimate. Bits are considered ACE unless proved otherwise.

## Microarchitectural Un-ACE Bits

Processor state bits that cannot influence committed instruction path are called “microarchitectural un-ACE bits” .

- *Idle or Invalid State*: Often bits are idle or do not have valid information.
- *Mis-speculated State*: Speculative operations that will later be found incorrect are un-ACE. (Branch prediction, etc).
- *Predictor Structures*: Faults here will possibly cause a misprediction, and will impact performance, but not affect performance.
- *Ex-ACE State*: ACE bits become non-ACE after their last use. Includes register values after last use, instructions waiting in case of pipeline flush that are never needed, etc.

## Architectural Un-ACE Bits

Architectural un-ACE bits are those that affect correct-path instruction execution, but in ways that do not impact output.

- *NOP instructions*: on spec2k Alpha 10% of instructions are NOPs. On ia64, 26% NOPs. Only bits that would convert a NOP to a non-NOP would be ACE.
- *Performance-enhancing instructions*: for example, prefetches. A wrong prefetch will slow a program but not cause incorrect output. On Alpha 0.3% of spec2k is prefetches.

## Architectural Un-ACE Bits (continued)

- *Predicated-false instructions*: On predicated architectures you can have a predicate bit that says if a register is true, to execute the instruction. The predicate register is obviously ACE but if the predicate register is false than the instructions are all non-ACE. Their tests found about 7% of dynamic instructions predicated false.

## Architectural Un-ACE Bits (continued)

- *Dynamically Dead Instructions*: Instructions whose results are never used.

Two groups: First-level Dynamically Dead (FDD) and Transitive Dynamically Dead (TDD). TDD only use as inputs the results of FDD. Under spec2k on Alpha 9% FDD and 3% TDD via registers, 14% if you track both registers and memory.

This paper, numbers higher than previous papers, possibly due to aggressive compiler optimizations which increase fraction of dead instructions.

They conservatively count all opcode and destination register bits as ACE, as changing those could make instruction no longer dead.

## Architectural Un-ACE Bits (continued)

- *Logical Masking*: Often in a chain of instructions some bits will be masked off, thus their previous state is not ACE.

Most logical matching comes from compares versus zero, where any non-zero value will serve.

Another example is 32 bit calculations on 64 bit machines where the top-bits can be considered un-ACE (not true on the Alpha)

- *Others* There are other possible scenarios for un-ACE bits but they are not investigated in this paper.

## Computing AVF

- The AVF for an unprotected storage cell is the percentage of time it contains an ACE bit.
- Although originally defined in terms of storage cells, the rest of the paper will compute AVF for a whole hardware structure.
- The AVF for a whole structure is the average AVF for all bits in the structure, assuming they all have the same structure and raw FIT rate.
- The AVF is equal to:  $\frac{\text{avg number of ACE bits resident in cycle}}{\text{total bits in the structure}}$
- The above can be re-written:  $\frac{\sum \text{residency (in cycles) of all ACE bits}}{\text{total number of bits in structure} \times \text{total execution cycles}}$

## Computing AVF Using Little's Law

- Little's Law can be expressed as  $N = B \times L$  where  $N$ =average bits in a box,  $B$ =average bandwidth per cycle into the box, and  $L$ =average latency of an individual bit through the box.
- Thus we can express the AVF formula as: 
$$\frac{B_{ace} \times L_{ace}}{\text{total numbers of bits in hardware structure}}$$
- In many cases it is possible to compute the bandwidth of ACE bits into a structure and average residence cycles of ACE instructions using hardware performance counters, allowing AVF estimation without a simulation model.

## Computing AVF using a Performance Model

- In this paper AVF was calculated for the Instruction Queue and Execution Units using the Asim performance model framework.
- To compute need:
  - △ sum of all residence cycles of all ACE bits
  - △ total execution cycles for which we observe ACE residence time
  - △ total number of bits in a hardware structure
- All of these can be computed with a performance model

# The Computation

AVF algorithm divided in three parts:

- Record the residence time to instruction in each structure as it flows down the pipeline.
- Before the instruction disappears from the machine either by squash or commit, update the structures it flowed through with info on residency time, whether it committed, etc. Also figure out if the instruction was dynamically dead or masked in some way.
- At the end of the simulation, use the information gathered to compute AVF.

## Determining if instruction is Dynamically Dead

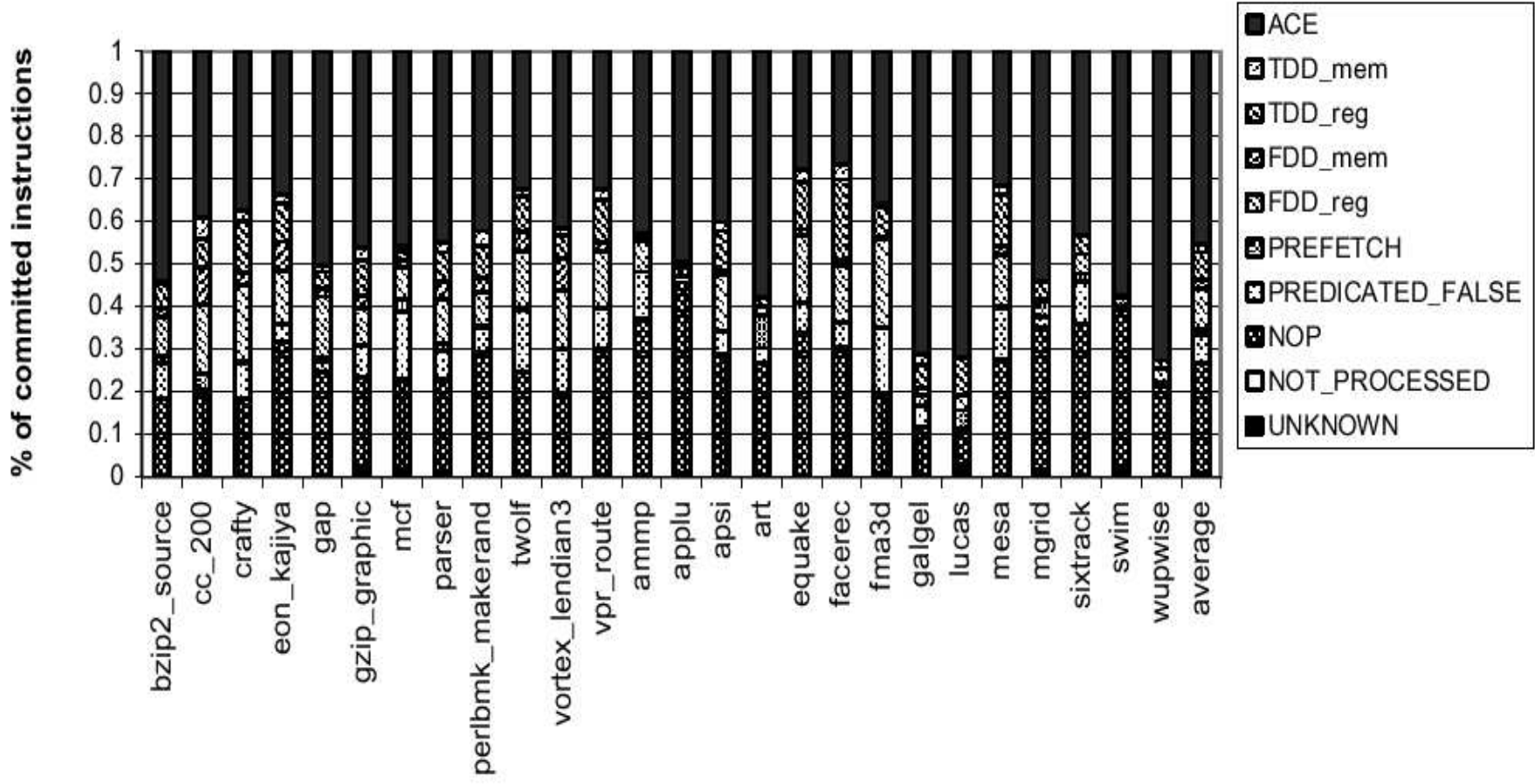
- To determine if instruction is dynamically dead, must keep track of the future of an instruction's result.
- When an instruction commits, it is entered into the analysis window.
- They determined keeping an analysis window of 40,000 instructions seemed to cover most cases for determining DD properly.
- They computed FDD, TDD, and logical masking. Basically kept linked-list of producers and consumers and determined deadness using that.
- They wrote a number of custom assembly benchmarks to determine that their DD detection code was working properly.

# Implementation

- Simulated an Itanium2-like ia64 processors scaled to current technology.
- Processor was modeled in detail with Asim performance model framework.
- Red Hat Linux 7.2 was modeled via OS simulation front-end.
- Spec2k used for analysis, compiled using Intel electron compiler.
- Used SimPoint to fast-forward analysis. Results shown for first SimPoint, using 100 million instructions (including NOPs).

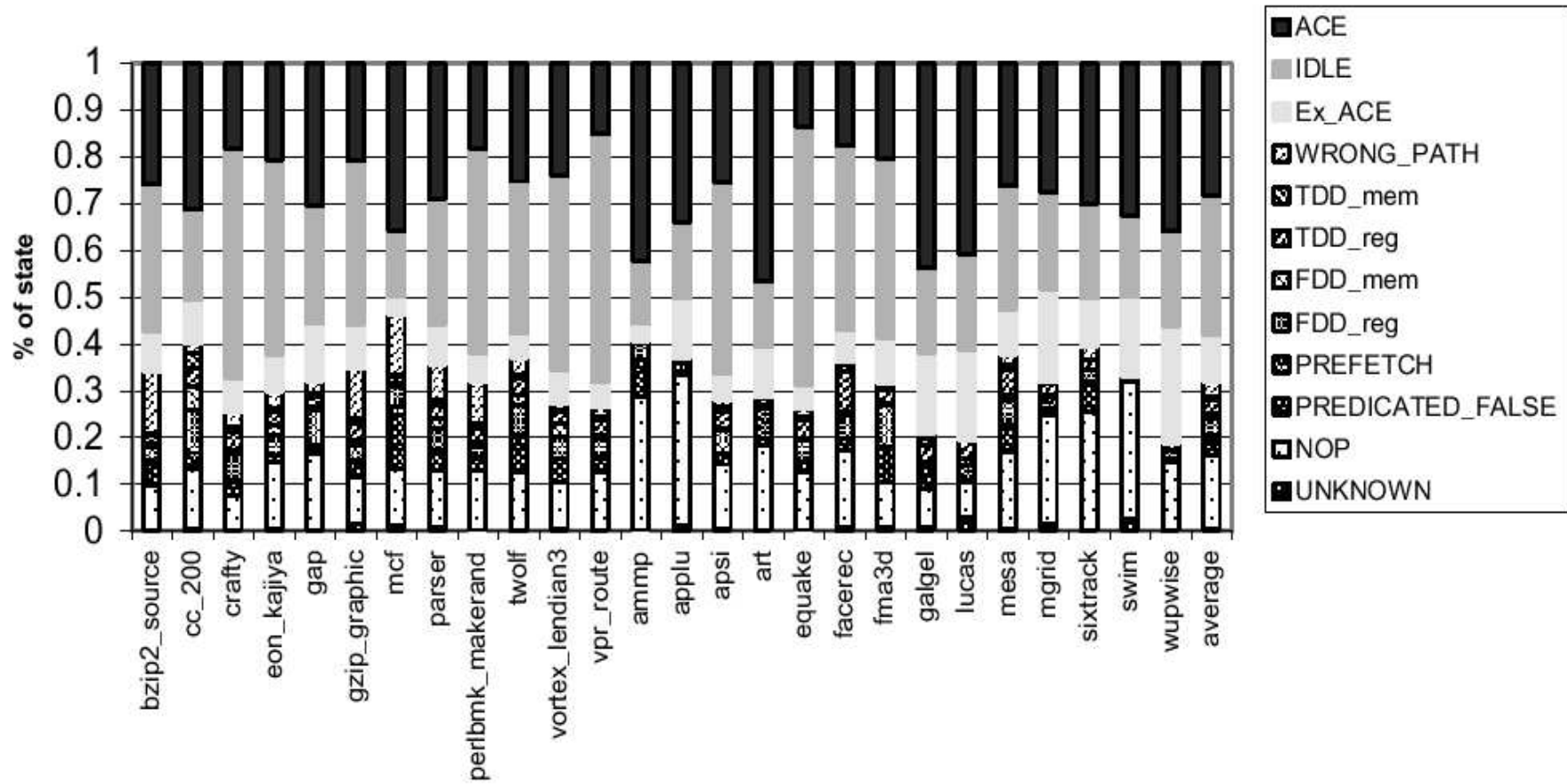
# Results 1

- On average get about 45% ACE instructions
- The rest are un-ACE. Some contain ACE bits but conservative.
- UNKNOWN means unable to determine if live or dead.
- NOT\_PROCESSED is the last 20,000 instructions. The above 2 only account for 1%.
- NOPs are 26%, predicated false are 6.7%, prefetch are 1.5%.



## AVF for Instruction Queue

- On average a storage cell contains an ACE bit 28% of the time.
- A cell is on average idle 30% of the cycles and contains an un-ACE bit 42% of the cycles.
- Across the benchmarks the AVF number ranges from 14% to 25%.
- Floating point programs have higher AVFs compared to integer programs (31% versus 25%). Floating point programs have more long-latency instructions and few branch mispredictions so use instruction queue more effectively.



## Results

- Using Little's Law, average AVF is 19% which is 9% lower than other results. This can be attributed to the ACE bits of un-ACE instructions.
- If had done Little's Law analysis at the bit-level rather than instruction level would have matched the 28% results.

## Little's Law Table

Integer Benchmarks	ACE IPC	ACE Latency (cycles)	AVF	# ACE Inst	Floating Point Benchmarks	ACE IPC	ACE Latency (cycles)	# ACE Inst	AVF
bzip2-source	0.55	22	12	19%	ampp	0.23	92	21	33%
cc-200	0.57	18	10	16%	applu	0.82	21	18	27%
crafty	0.37	15	6	9%	apsi	0.31	31	9	15%
eon-kajiya	0.36	20	7	11%	art-110	0.68	37	25	40%
gap	0.78	17	13	21%	equake	0.26	12	3	5%
gzip-graphic	0.60	13	8	12%	facerec	0.41	7	3	5%
mcf	0.25	68	17	26%	fma3d	0.59	11	7	10%
parser	0.49	24	12	19%	galgel	1.10	21	23	35%
perlbmk-makerand	0.38	17	7	10%	lucas	1.23	17	21	33%
twolf	0.30	27	8	13%	mesa	0.47	16	8	12%
vortex_lendian3	0.42	22	9	15%	mgrid	1.28	10	13	21%
vpr-route	0.35	12	4	7%	sixtrack	0.66	20	13	21%
					swim	1.08	16	17	27%
					wupwise	1.60	13	20	31%
average	0.45	23	9	15%	average	0.77	23	14	23%

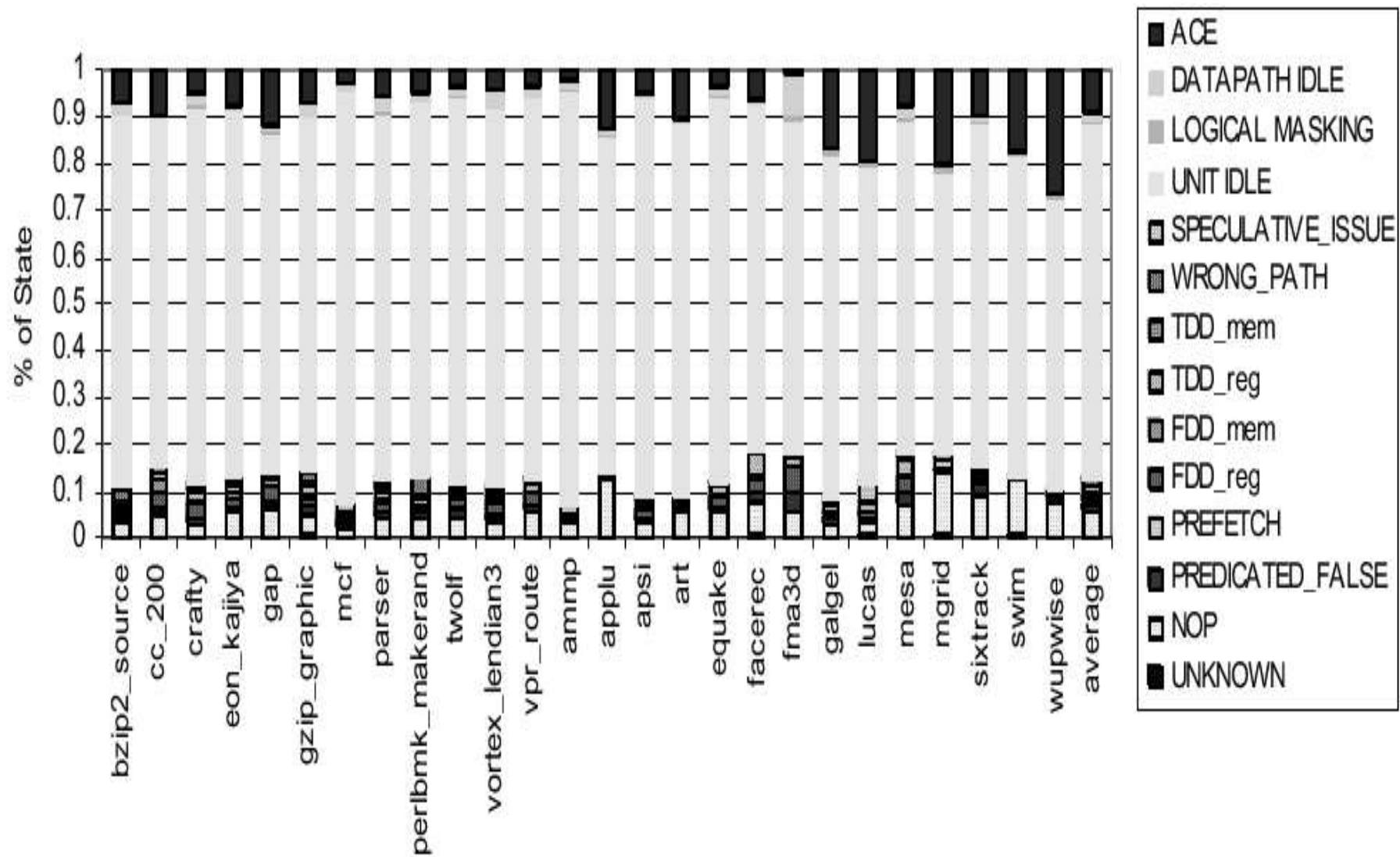
## AVFs for the Execution Units

- Have a six-issue machine with four integer pipes, two floating point pipes, and integer multiply performed in the fp unit.
- Assume the execution units are 50% control latches and 50% datapath latches.
- The execution units spend 11% of cycles processing ACE instructions (a range of 4% to 27%). Thus the average AVF of the execution unit is 11%

## AVFs for the Execution Units (Continued)

The execution unit AVF is significantly less than the instruction queue. For three reasons:

- Instructions wait in IQ for long time.
- Speculatively issued instructions might have to be replayed many times, but only the last one “counts” towards being ACE.
- FP units idle during integer codes, so un-ACE.



## AVFs for the Execution Units

- Can further reduce AVF by doing masking of datapath latches. Only bits with important info counted. This can reduce by another 1.5%.

## Related Work

- Kim and Somani did fault injection on Sun's picoJava RTL ( Register Transfer Level) model.
- Wand and Patel injected faults into RTL model of Alpha 21164 and reported AVF of less than 10% for pipeline latches. RTL models are more real-world than the performance model Asim.

## Related Work/ RTL

This papers claim improvements over statistical injection with RTL models.

- Statistical injection requires large amount of data for statistical significance.
- ACE analysis provides better determination if faults impact processor outcome.
- Gives better breakdown of why the AVF's happen, not just number
- Need an RTL model which isn't always available.

## Conclusion

- Per-structure AVF estimates should help designers estimate FIT rate early in design cycle.
- The estimates can help engineers find specific areas to fix if the part does not meet the FIT rate.
- Overall FIT rate of a chip can be lowered by adding more error protection with AVF estimates as a guide.