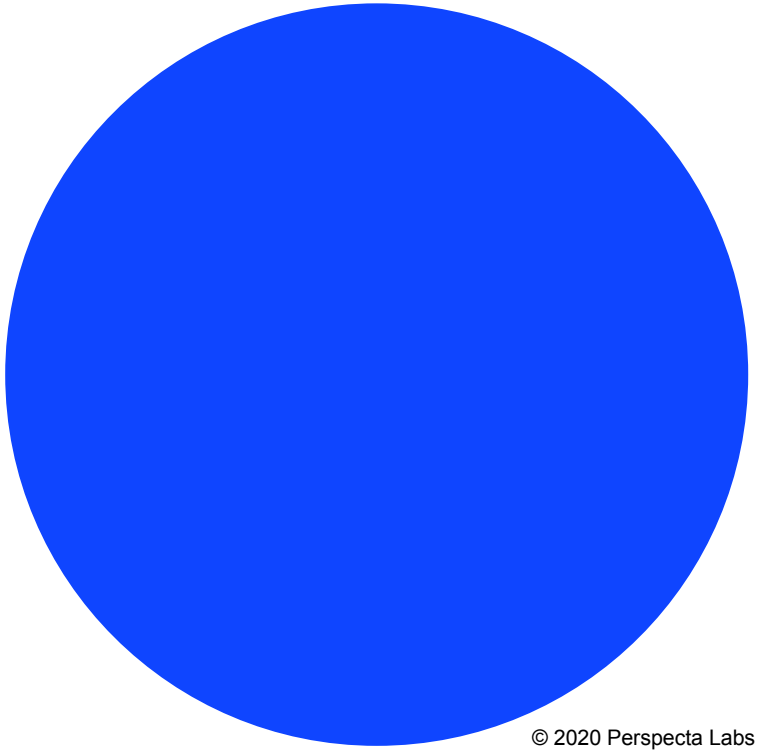
Weird Circuits in CPU Microarchitectures

IEEE LangSec Workshop
May 21, 2020 (Invited Talk)

Presenter:
Thomas S. Benjamin
tbenjamin@perspectalabs.com

Weird Circuits Team Members:
Thomas S. Benjamin [1], Dmitry Evtyushkin [2], Abhrajit Ghosh [1],
Jeffery Eitel [1], Jesse Elwell [1]

1) Perspecta Labs 2) William & Mary



Overview

- Microarchitectural Weird Machines - Weird Circuits
- Construction and Evaluation of Weird Circuits
- Future Work

Microarchitectural Weird Machines – Architecture vs Microarchitecture

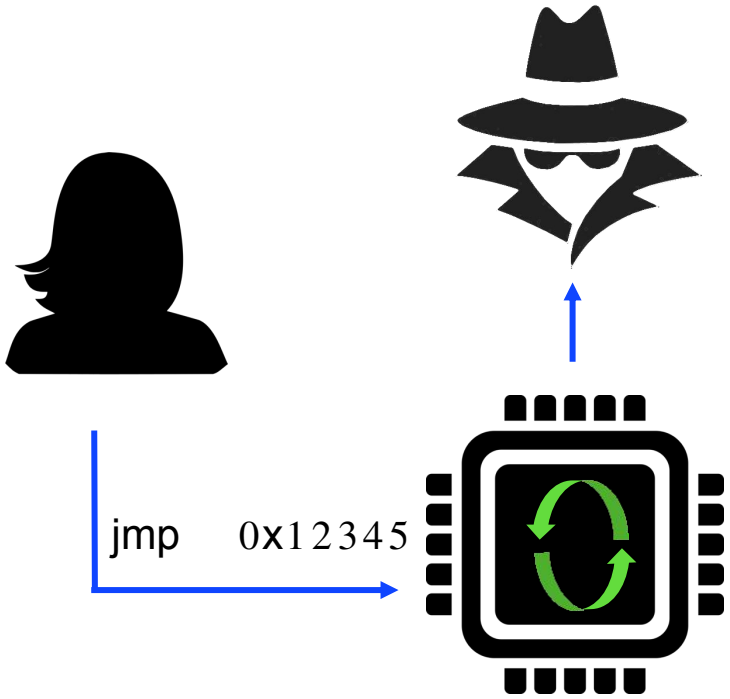
- Two abstract layers for existing CPUs
  - Architectural Layer: Instruction set, registers, IP, addressable memory
  - Microarchitectural Layer: Caches, Prefetchers, Branch predictors, Speculative execution
- Architectural Layer:
  - Well documented
  - Formally specifiable
- Microarchitectural Layer:
  - Realization of Architectural layer
  - **Complex** performance optimization mechanisms
  - Internal data structures and activities (e.g. load data into cache)
  - Little information on internal structure other than textbook-level descriptions
  - Transparent to programs other than execution time

Weird Machines – Related Work

- ELF weird machine (Shapiro et al)
  - Turing complete computation via ELF header manipulation
- Page fault weird machine (Bangert et al)
  - Turing complete computation via IA32 interrupt handling mechanisms
- Both the above are theoretically observable by an architectural reference monitor developed by following CPU specifications
  - Proposed Weird Circuits are based on ***microarchitectural state that is not observable*** via such a reference monitor

Attacking Microarchitecture

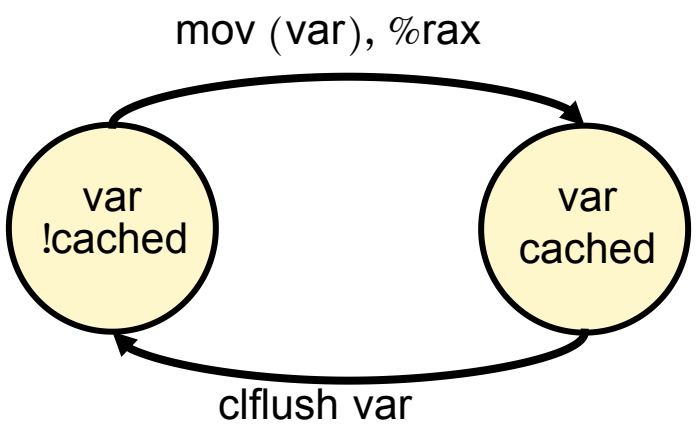
- µarch artifacts have been used to construct well known data leakage attacks:
  - Side channels, Spectre, Meltdown, etc.



- **In side channel attack:**
  - Actions of victim cause changes in **state of complex µarch components**
  - This state also affects behavior of target system for attacker
- Programs can ***implicitly*** manipulate µarch components
- Possible to formally **define an interface** for an abstract computer:
  - write to µarch components (set state)
  - read from µarch components (probe state)
  - perform computation (state transitions)

Weird Registers (WReg)

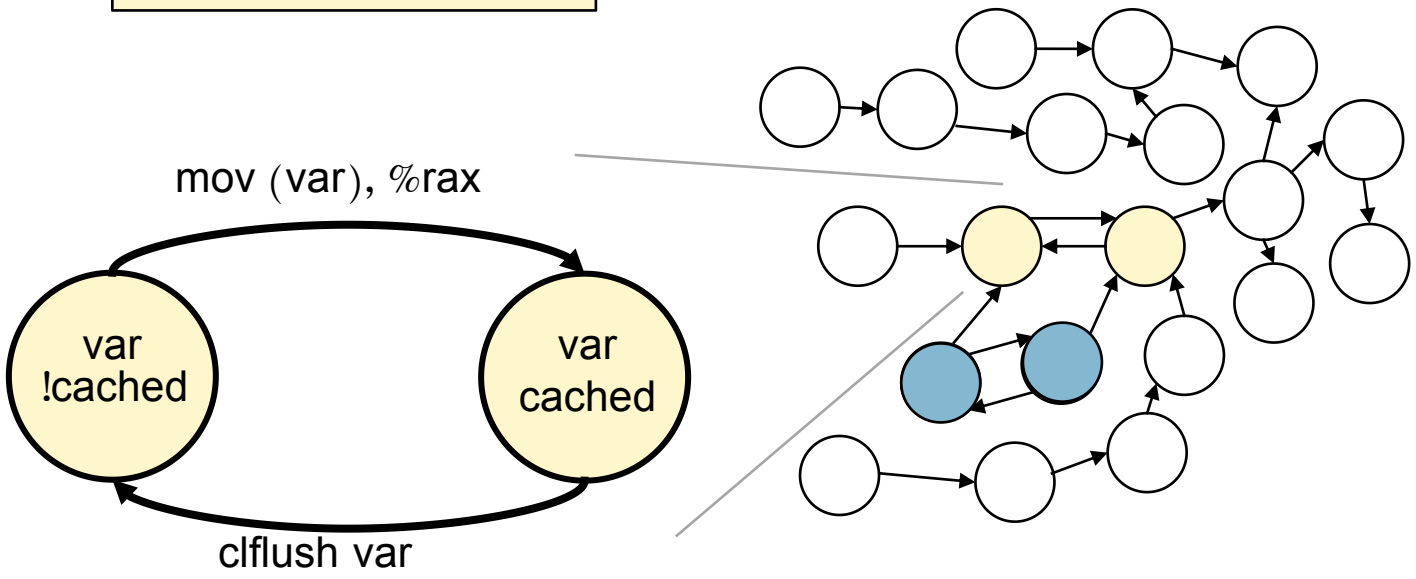
```
const int var = 42;
```



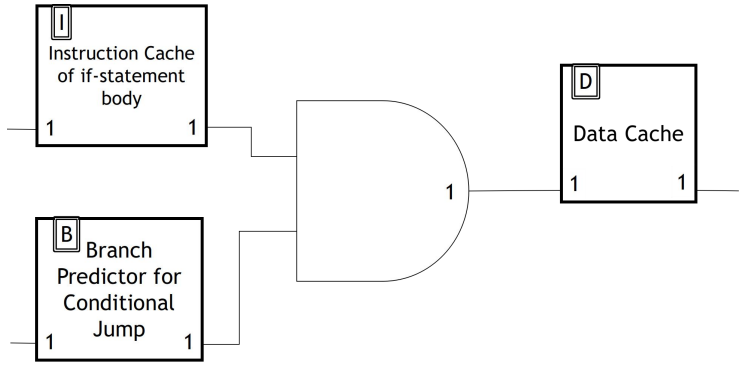
- Weird Registers (**WReg**) are abstractions to describe μarch artifacts with >1 state manipulable via programmatic inputs
- Example - Data Cache (**DC WReg**)
  - Two states: cached, non-cached
  - Interface for writing: load or flush data
  - Interface for reading: time access to variable
    - Cached → low latency
    - Non-cached → high latency
- The DC WReg implements basic **storage covert channel** functionality
- Actions on the Instruction Set Architecture (ISA) level have created state that is not visible at the architectual level
- This state can be read through timed execution at the cost of destroying the state.

Formally Describing Weird Circuits

```
const int var = 42;
```



- Any real-world machine can be viewed as a FSM where each state defines states of all its arch and μarch components
- Interaction of component states is leveraged to construct basic computational functionality
- For example, BPU, Data and Instruction Cache WRegs results in a state transition graph that implements an OR and AND binary gates



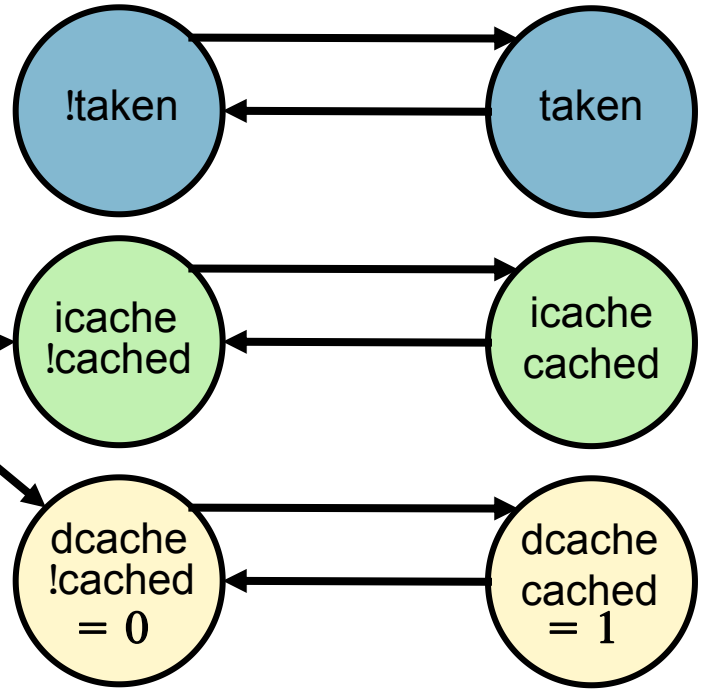
Weird Gates (WGate)

- Weird Gate (**WGate**) – Programmatic construct that reads >= 1 WReg and writes output into a WReg
- The Weird Gates we will discuss in this presentation use the Data Cache WReg (DC WReg) described in the previous slide as well as
  - The Branch Predictor (BP) WReg
    - By repeatedly evaluating a branch with the same conditional value we can train the BP into one of two states (taken or not taken).
    - Whether or not some chosen code will be speculatively executed can depend on the training of the BP
      - E.g. loading the data in a DC WReg
  - The Instruction Cache (IC) WReg
    - Whether or not some chosen code will be speculatively executed can also depend on whether it is in the instruction cache
- **Weird Circuit** is a collection of WGates connected to perform an operation

An example Weird Gate – Weird AND (W_AND)

AND Weird Gate code:

```
cmp    0, var1
je not_taken
    mov var2, %rax
    nop
    nop
    nop
not_taken:
nop
```



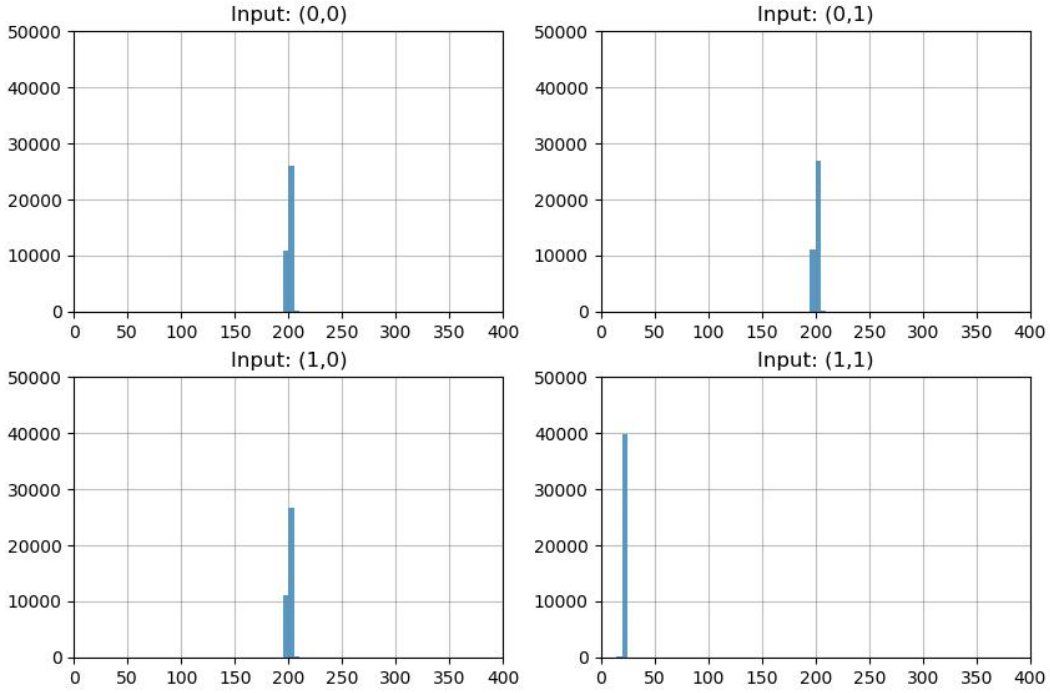
- AND Gate is built with 2 input WReg and 1 output WReg:
  - Input 1: Branch Predictor (BP)
  - Input 2: Instraction Cache
  - Output: Data Cache
- Operation:
  - var2 becomes **cached** only when **BP is trained to be non-taken AND code is in instruction cache**
- Utilizes: Speculative execution
- **Not visible at architecture level!**

Weird AND Output: Reading the Data Cache WReg

- In this experiment we executed the Weird AND gate from the previous slide 160,000 times with inputs I (IC WReg input) and B (BP WReg input) chosen uniformly at random.
- We observed the output of each Weird Computation by timing the load of the data in the DC WReg. All times in this presentation are measured in CPU cycles
- The table on the right shows the distribution of the load times for each possible (I, B) input pair
- As we expect, the load time for D is significantly faster when both the I input and B input are 1 – Speculative execution has placed D in cache

I Input	B Input	Min Time	Q1 Time	Median Time	Q3 Time	Max Time	Standard Deviation	Mean Time
0	0	174	198.0	**198**	200.0	48600	322	237.2
0	1	176	198.0	**198**	200.0	2506	109	220.4
1	0	176	198.0	**198**	200.0	3140	106	218.2
1	1	18	20.0	**20**	22.0	32264	161	22.0

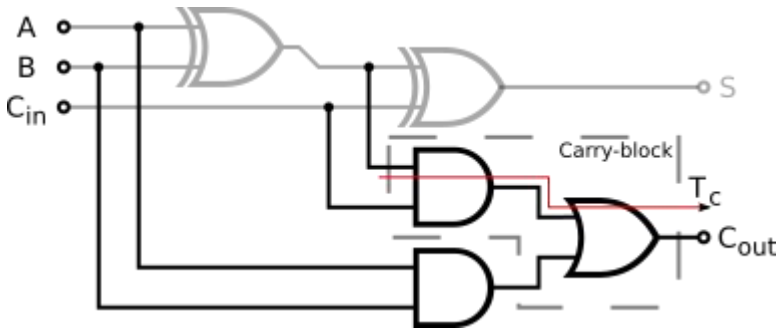
Load Times (CPU Cycles) For Data in DC WReg



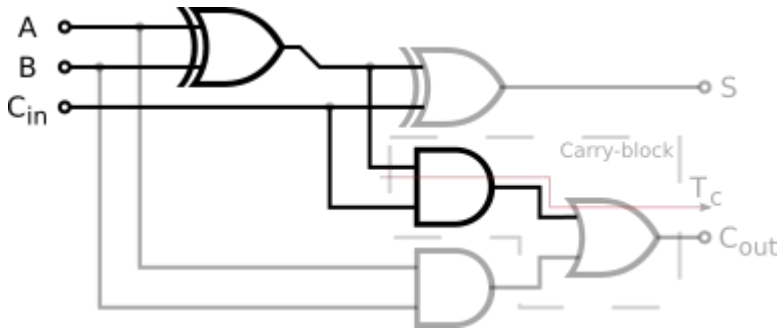
Histogram of Load Times (CPU Cycles) For Data in DC WReg

Simple and Compound Weird Circuits

- Using the IC WReg, BP WReg and DC WReg we have successfully constructed stable WGates that are Weird versions of the following simple gates:
  - AND, OR, NOT, NAND, NOR, XOR
- If we permit visible intermediate states we can construct arbitrary Weird Circuits because NAND and NOR are universally composable
- Using architecturally visible intermediate states we have constructed Weird Circuits such as The Full Adder Weird Circuit (W_ADD)
- We are making circuits like a Full Adder progressively “Weirder” by replacing these architecturally visible intermediate states with more complex discrete Weird Circuits
  - Example: W_AND_AND_OR
  - Example: (A ^ B) & C



Two ANDS and an OR in a single Weird Circuit



(A ^ B) & C in a single Weird Circuit

Reliability and More Complex Calculation

- Many parameters with tradeoffs between accuracy and run time
  - e.g. number of training rounds for Branch Predictor
  - Must be determined empirically on a given series of CPU
- In a recent experiment we performed 100,000 executions of each of the following gates with parameters chosen to balance speed and accuracy with the following results

Gate Type	W_AND	W_OR	W_DC_OR	W_NAND	W_AND_AND_OR
Accuracy	0.99972	0.98256	0.93612	0.99996	0.99746

- Above accuraccy sufficient for error tolerant computation
- Cryptographic algorithms such as SHA-1: By design single-bit errors cause completely incorrect output
- Two-tiered redundency for greater reliability
  - Each gate run a set of *k* times. Median timing value from that set compared against threshold to get logic value
  - That process repeated *n* times and final logic value is majority vote
- Additional tradeoffs between architectual visibility, run time, and reliability
  - Intermediate values visible in memory

More Complex Calculation - SHA-1 Results

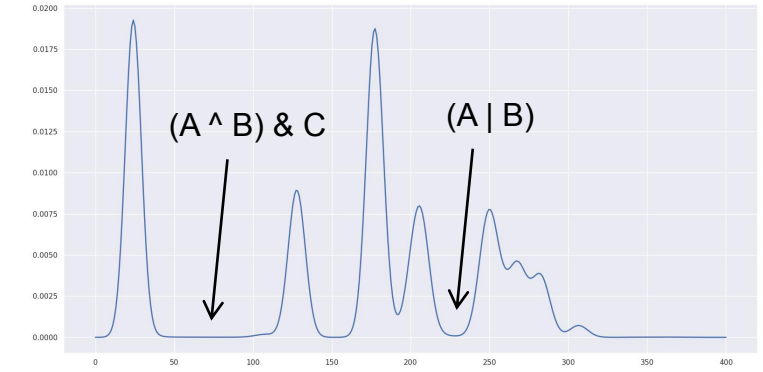
- We have implemented SHA-1 entirely with Weird Circuits
  - Most intermediate values architecturally visible
  - All actual SHA-1 computation is Weird: e.g. there are no calls to the CPU's ADD instruction during additions, and no calls to the CPU's XOR during the algorithm's XOR operations
- This experiment shows results for a single round of SHA-1 execution
  - Success rates vary with params chosen. In this experiment series we ran SHA-1 4 times using our default params, all of the runs succeeded.
  - This execution completed in 26 minutes and 44 seconds
  - The timing set size (see prev. slide) for each gate was approx 10, and vote group size approx 5
  - The following accuracies are after median timing selection but before vote.

Gate	Correct / Total	Accuracy
W_AND	9600 / 9600	1
W_OR	7680 / 7680	1
W_NAND	379008 / 379008	1
W_AND_AND_OR	907869 / 907872	0.999997

Accuracy of individual WGates during a SHA-1 Execution - Timing Set Size = 10

Future Work

- Investigate how many independent WGates can be constructed and composed using available microarchitectural resources of a given CPU
  - Example resources: number of branch predictors, size of instruction and data caches
- Investigate noise reduction techniques
- Explore additional microarchitectural structures
  - Re-Order Buffer (ROB), Translation Lookaside Buffer (TLB), ALU based, many more
- How to interpret and formally model multi-level logic?
  - Some Weird Circuits have gate timing distributions that are not bi-model
  - Simultaneously perform the equivalent of several traditional gates



Probability Distribution of Timings for a Three Input Weird Circuit

QUESTIONS?