# Novel Techniques for Memory

- Speculative Memory Cloaking
- Speculative Memory Bypassing
- Transient Value Cache

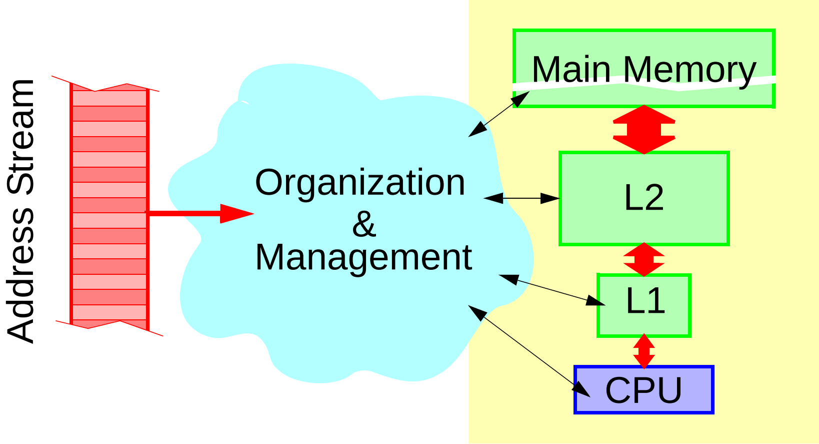




# The "Memory Problem"

**Store & Retrieve Values with:** **1. Low Latency** **2. High Bandwidth**

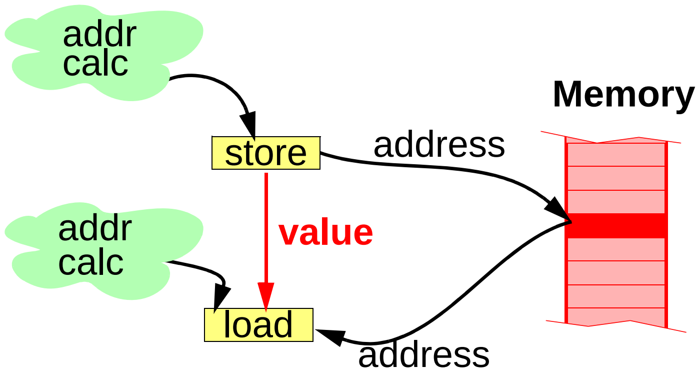
Not all storage can be built this way: Intelligent Mechanisms




**Name-Centric Approach**

**observe and exploit address stream behavior**

## What Purpose Does Memory Serve?




**Memory can be an inter-operation communication mechanism**

**Addresses** -> Communication Channels

**Instructions** -> Communicating Parties

**Values** -> Messages

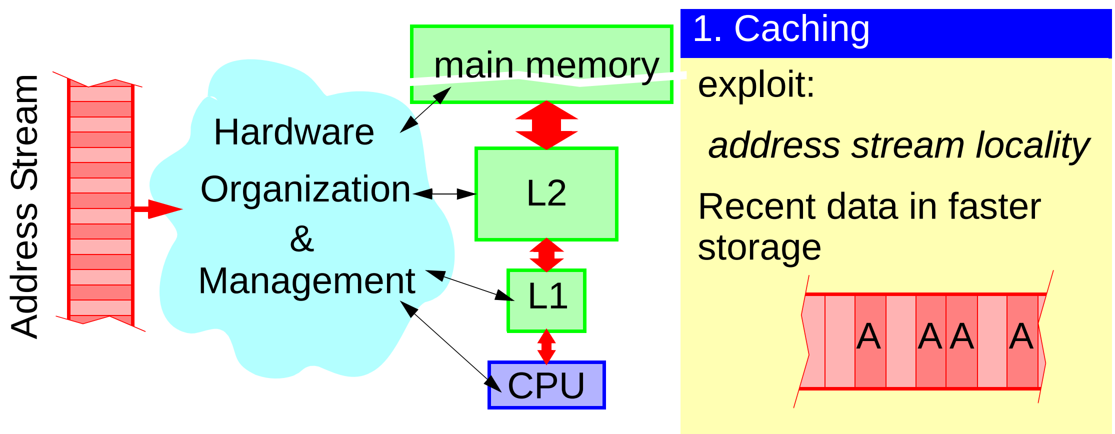
**Address Calculation** -> Channel Selection





## The Name-Centric Approach

- Programs access memory using addresses, i.e., **names**
- Optimize for that: *quick response on address requests*
- How to organize and manage?

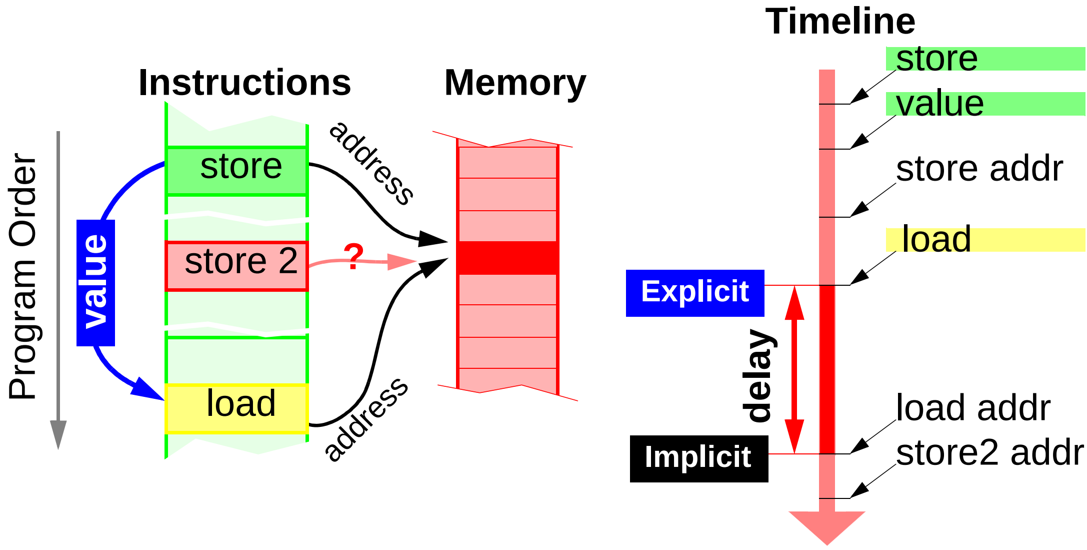



**Exploit Address Stream Behavior**

**Goal: Approximate a Large-Fast Memory**

# Address-Based Memory Concerns

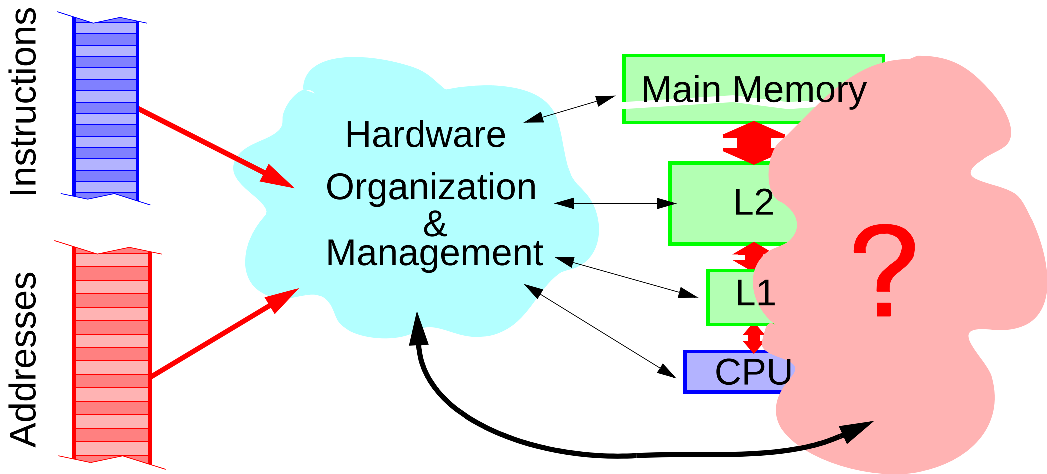
| Implicit | Explicit |
| --- | --- |
| **Delays:** 1. Calculate address<br>2. Establish Dependence | Store - Load: Direct Link<br>**No Delays** |








# The Communication-Conscious Approach

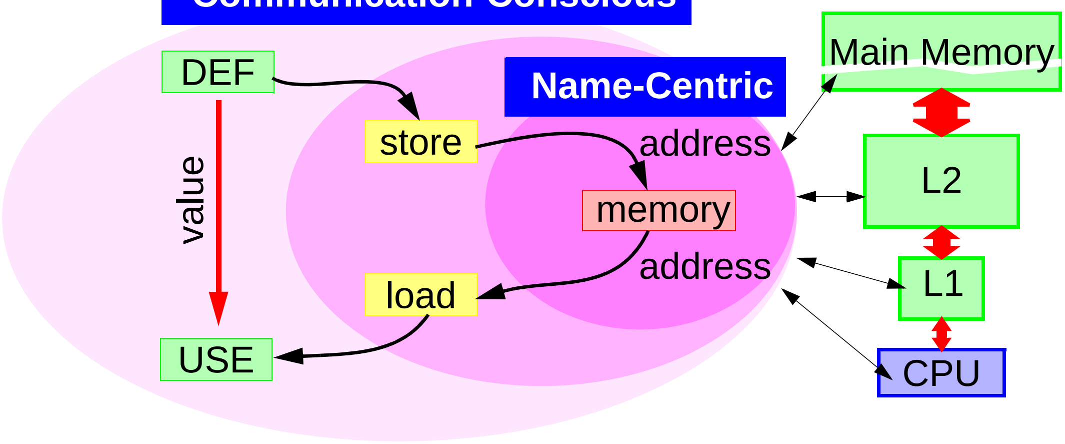



In addition to address stream behavior:

**Expose and Exploit the Inter-Operation Communication**

**Goals:**
1. **Approximate a Large-Fast Memory**
2. **Optimize for the Communication**

# Memory as a Communication Agent




***In Addition to***
*addresses*

**Communication-Conscious Approach**

**Expose the communication**

**Observe and exploit its behavior**





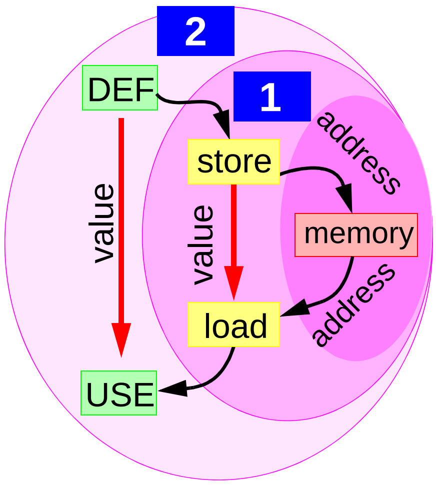
# Communication-Conscious Techniques

Communicating via addresses => inherent delay

**Observe:**
- Many loads get their value from a recent store
- These dependences are **predictable**

**1. Speculative Memory Cloaking**
- Prediction: link *load* - *store*
- pass value
- verify through memory

**2. Speculative Memory Bypassing**
- link ***DEF - USE***




**Communication Latency is Reduced**

# Communication-Conscious Techniques

DCache ports are becoming expensive

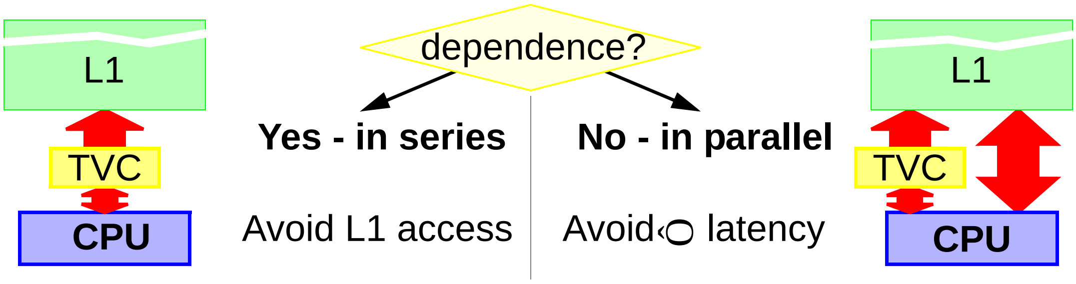
**Observe:**
**A.** Recent stores feed many loads
**B.** Many recent stores are killed

**+** Small cache can service these

**-** Latency for other loads will increase

**C.** A & B / **Dependence Status** is **predictable**

**3. Transient Value Cache**




**L1 DCache Bandwidth/Port Requirements are Reduced**



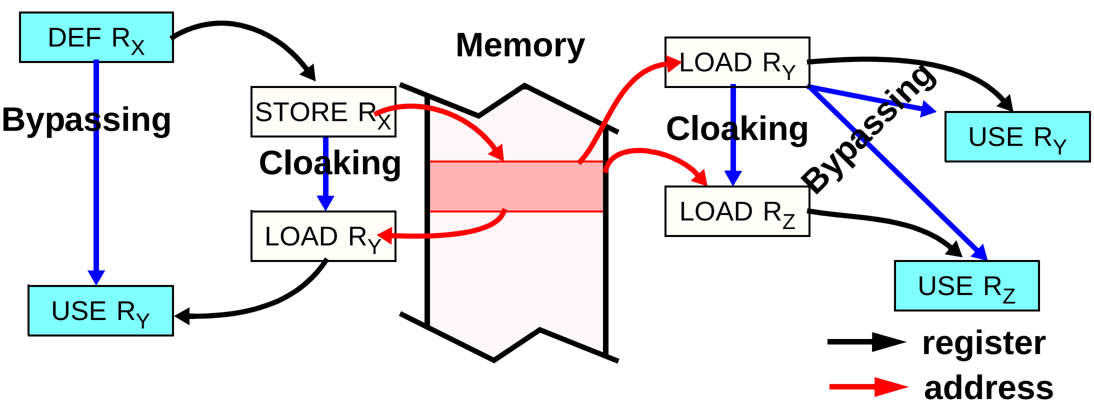


# Speculative Memory Cloaking/Bypassing

**Address-based Memory as: Storage vs. Interface**

**Ask:**
1. What is memory used for?
2. How addresses impact the action?

**Inter-operation Communication** | **Data-Sharing**




**Dynamically Create Direct Links Between Producers/Consumers**

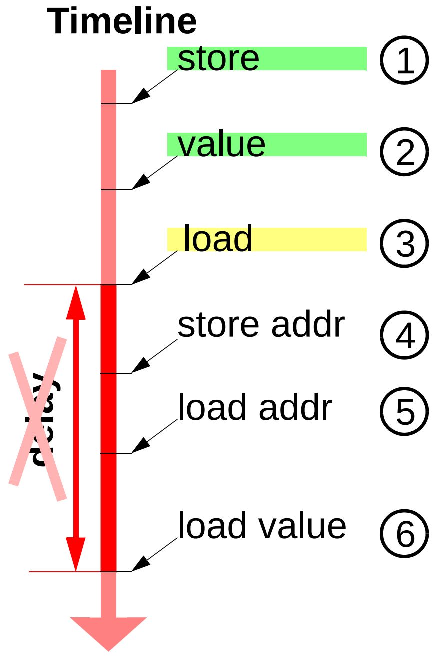
# Speculative Memory Cloaking - Example

1. Store: predict load PC
2. Associate value and load PC
3. Load: predict store

    *Check if value is there*

    *Pass to other instructions*

4. Write value to memory
5. Access memory
6. Verify value, re-execute if



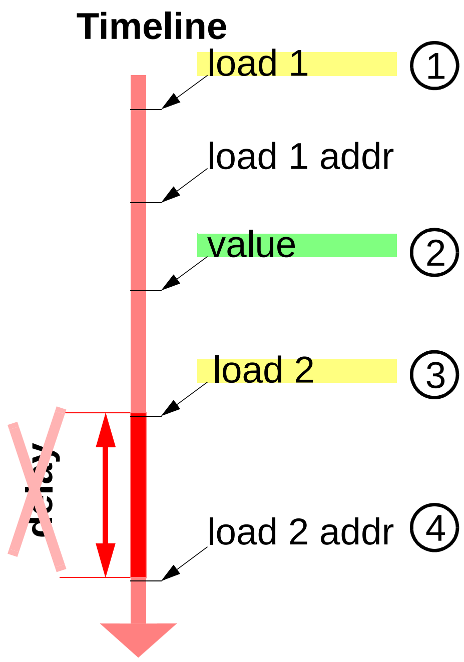

**On-the-fly: Convert implicit communication into explicit**





# Speculative Memory Cloaking - Extension

Could be used for load-to-load dependences



# Why “Cloaking”?

**cloak** *n* *\’klok*

2 : *to alter so as to hide the character of*

3 : *something that conceals*

**“Speculative Memory Renaming”?**

• Already in use in the same context: ARB, LSQ (w/o Speculative)

• **Re-name:** *change the name*

**-** *associate address with a new name*

**-** *Legacy of “Register Renaming”:*

**-** *can go from address to new name*

*synonym and address are NEVER associated*

*can’t determine synonym from address*

*other accesses to the same address can’t locate synonym*



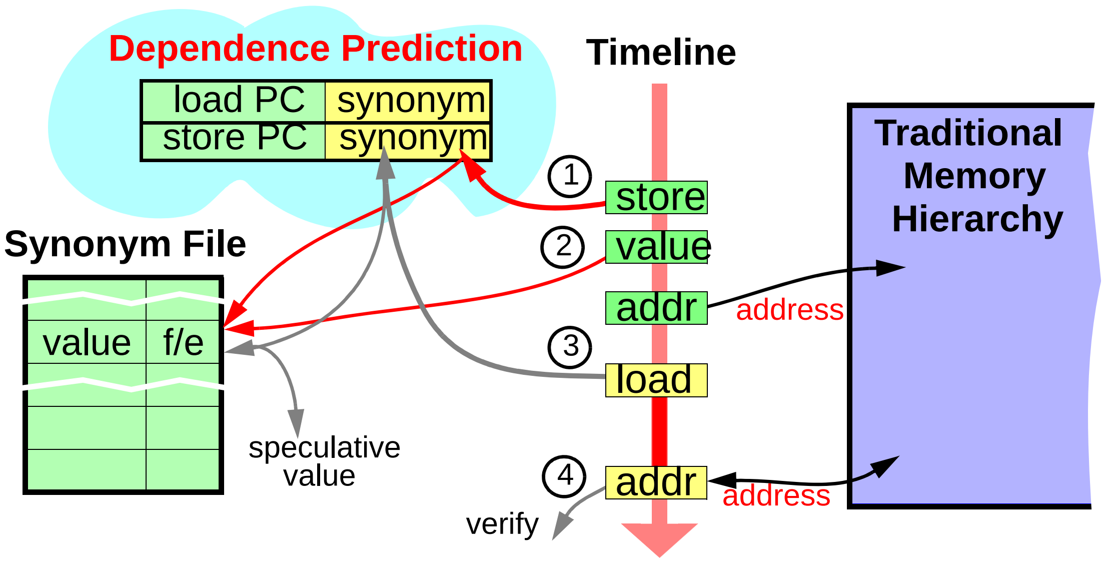


# Speculative Memory Cloaking

**Dynamically & Transparently convert implicit into explicit**

• **Dependence prediction** => direct store-load or load-load links

• Speculative and has to be **eventually** verified



# Predicting RAW and RAR Dependences

1. Build Dependence history: Dependence Detection Table

**Record:** (*store PC*, *address*) or (load PC, *address*)
**Loads:** (*load PC*, *address*)
=> (*store PC, load PC*)
=> (*load PC, load PC*)

2. Use history to predict forthcoming dependences
   assign synonyms to detected dependences
   use synonym to locate value

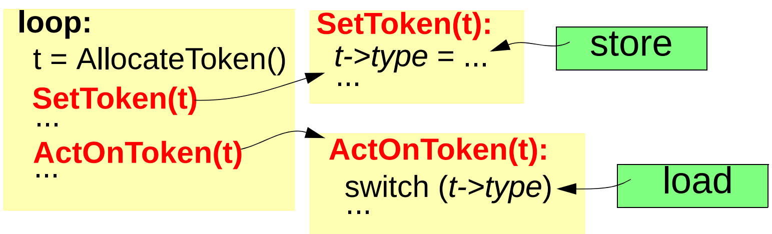


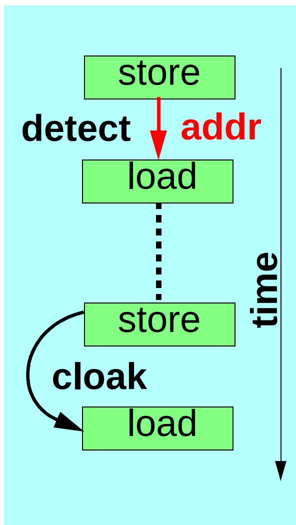

# An Implementation

Support Structures:

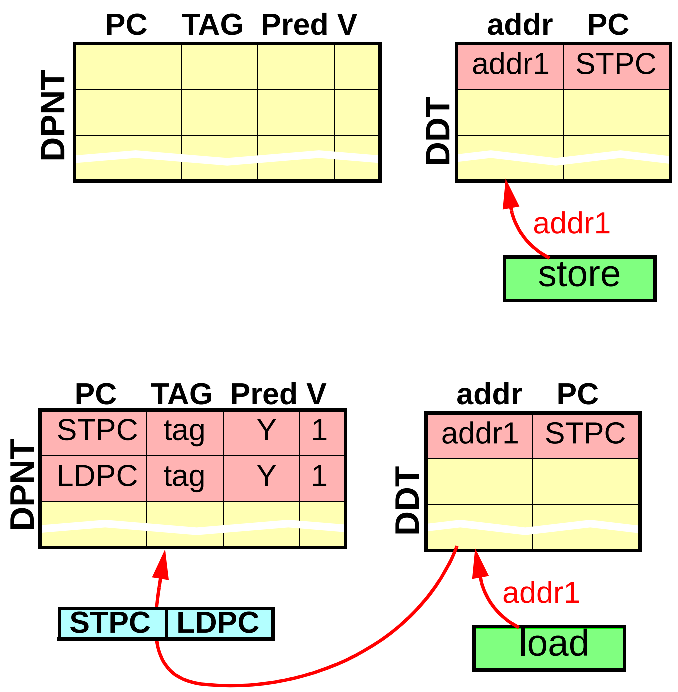
1. **Dependence Detection Table DDT**
2. **Dependence Prediction and Naming Table DPNT**
3. **Synonym File SF**

Example:



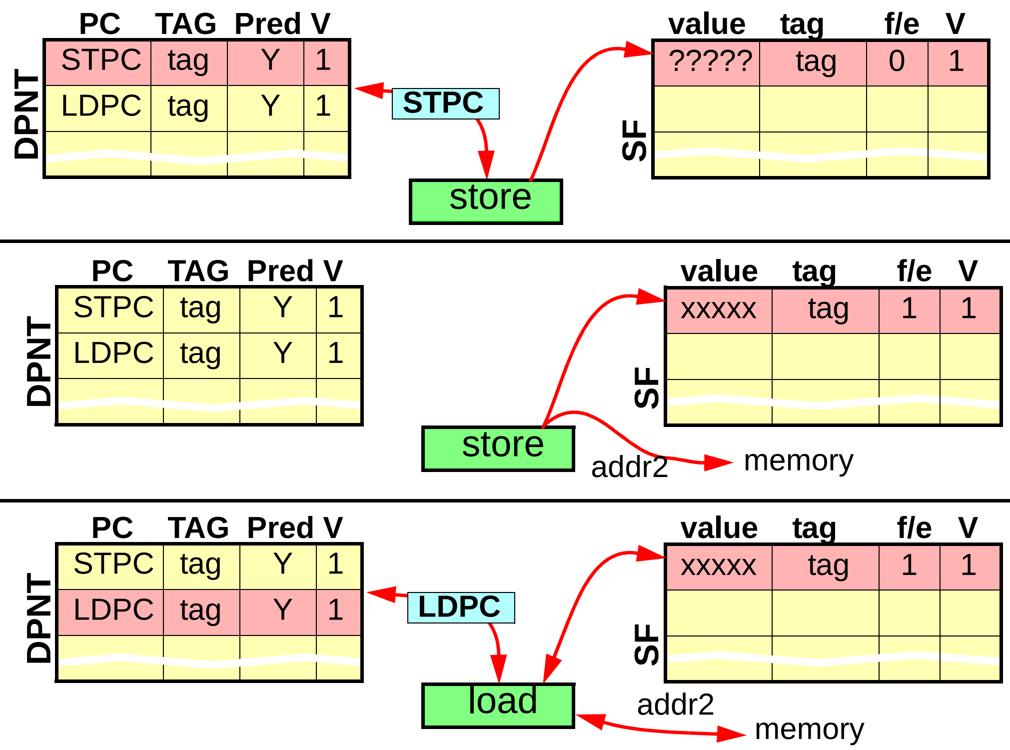
# An implementation - Example

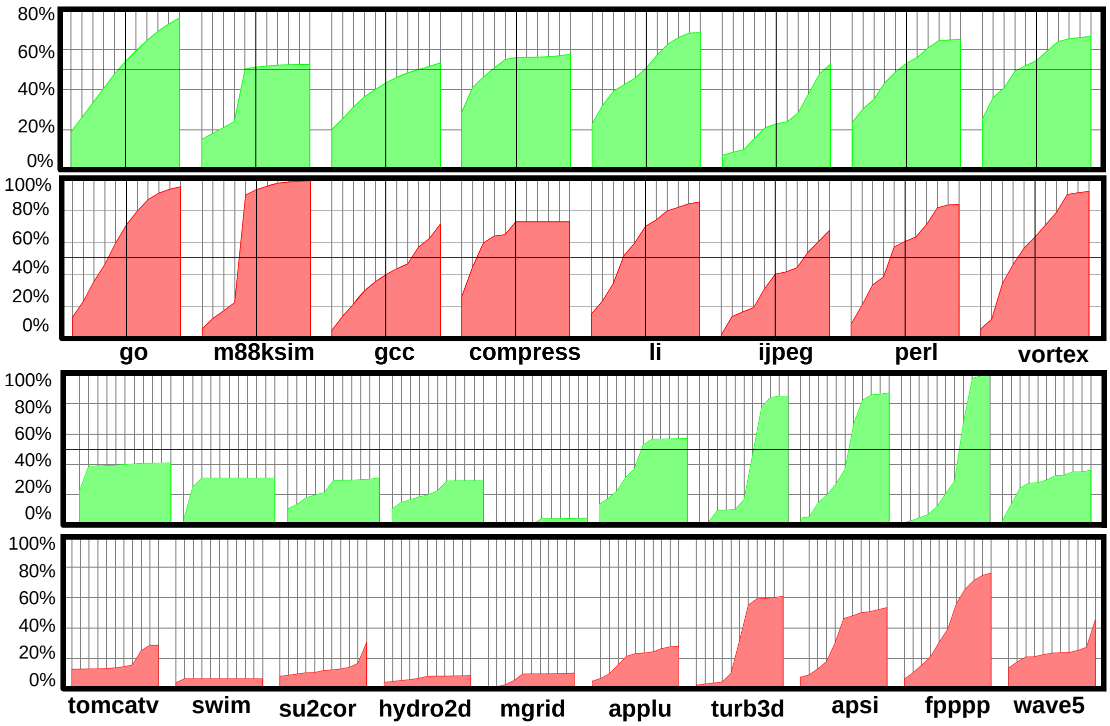







# An implementation - Example



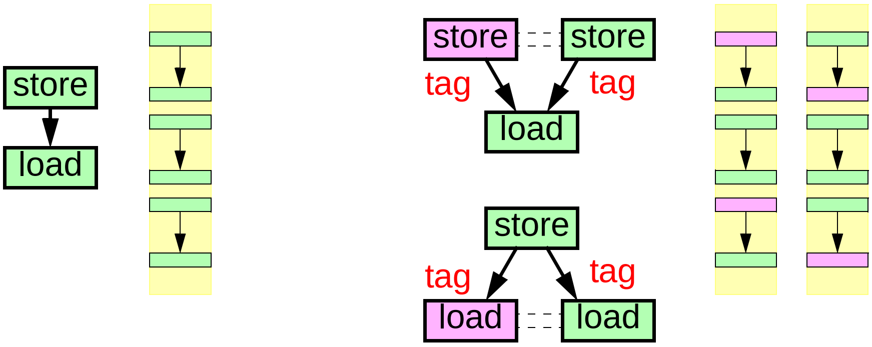







# Predicting Dependences - Synonym Generation

**1-on-1 straightforward** **N-to-N is common**




Break into steps:

**1. Predict dependence status (existence)**

**2. Figure out with who / synonym**

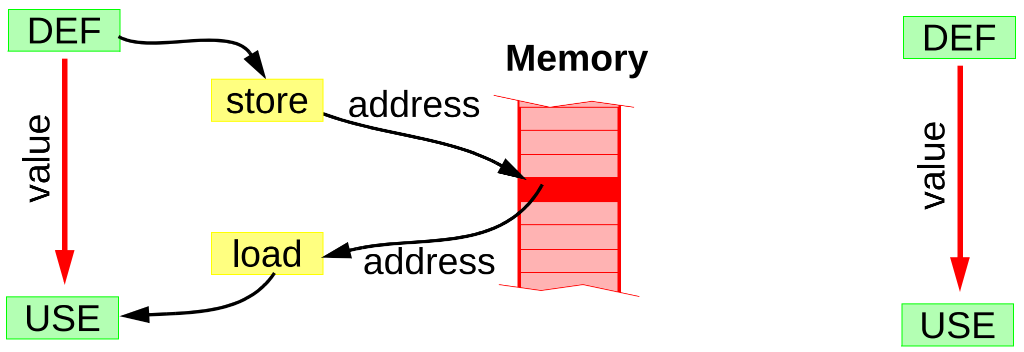
dependences w/ common parties same synonym

execution path determines which is the right one

# Speculative Memory Bypassing

**Observation:** Store and Load are used to just pass values

**Take store & load off the communication path**




- DEF-store-load-USE must be in the instruction window

  Larger windows: higher potential coverage

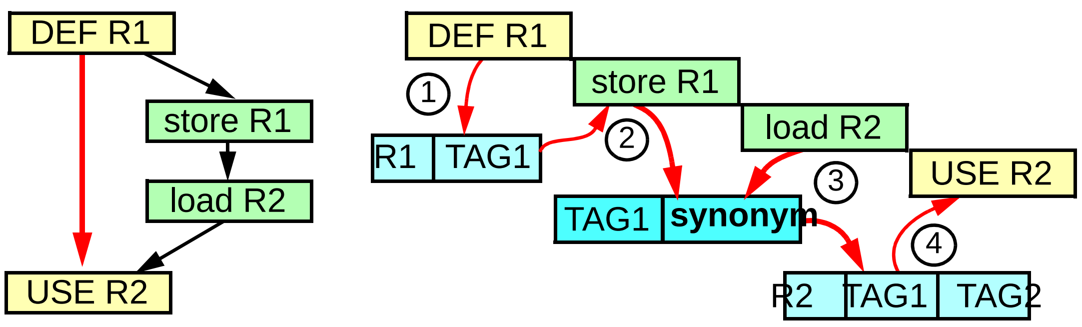
- Extents over multiple store-load dependences





# Speculative Memory Bypassing

**Observe:** Store and Load are used to just pass values




- Extents over multiple store-load dependences
- DEF and USE **must co-exist** in the instruction window

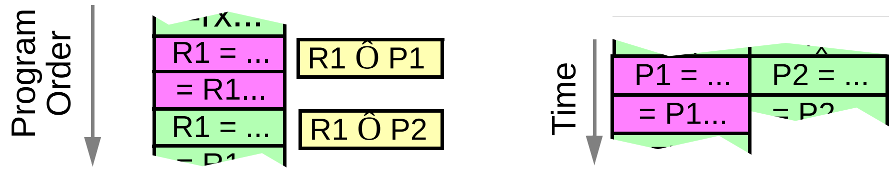
**Takes load-store or loads off the communication path**

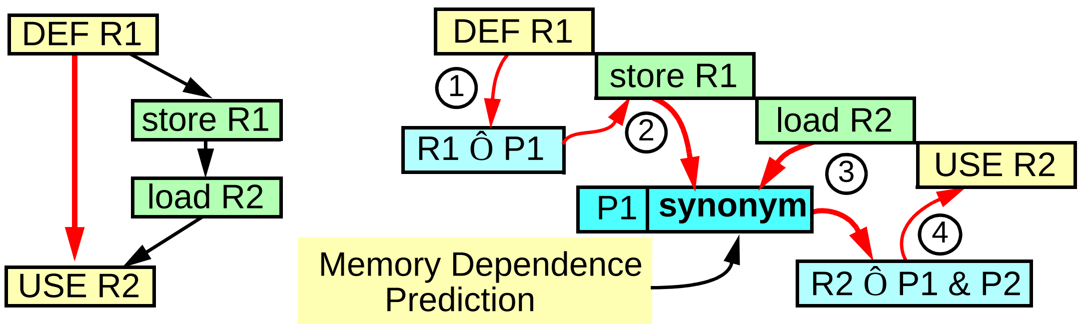
# Speculative Memory Bypassing

**Observation:** Store and Load are used to just pass values

**Take store & load off the communication path**

- Straightforward extension to **Register Renaming**








# Evaluation Roadmap

- Detecting Dependences
- Cloaking Coverage
- Cloaking Mispeculation Rate
- Performance
  1. Squash Invalidation
  2. Selective Invalidation

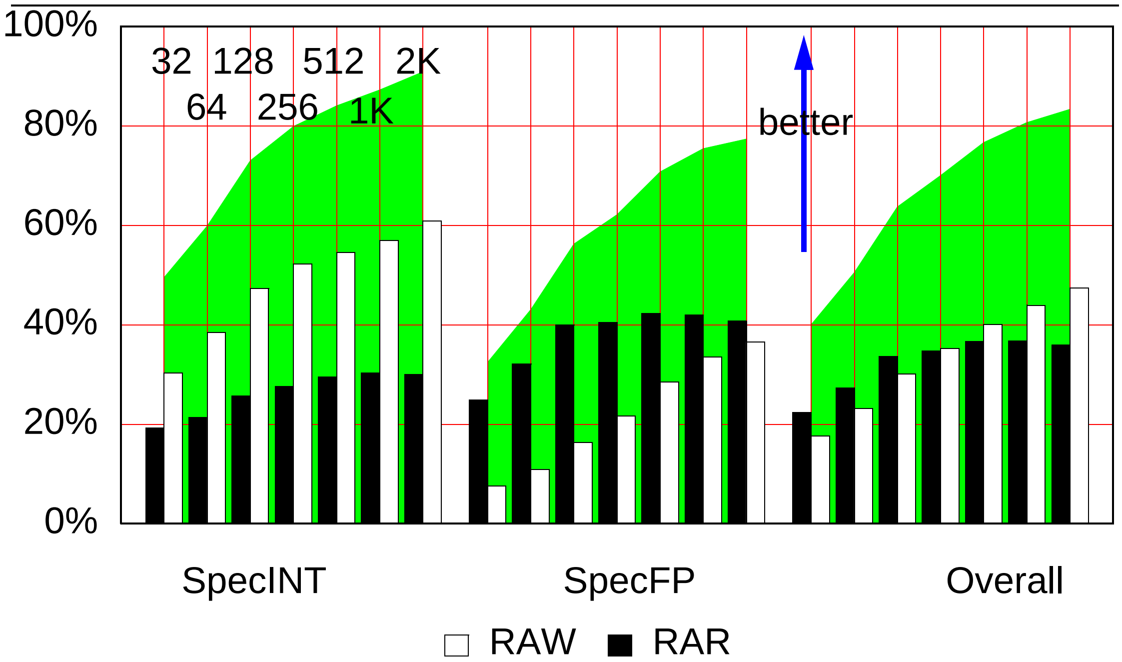
Base Machine:

8-way superscalar 4 load/store ports

128-entry window + 128 entry ABS w/ 1 cycle latency

Naive Memory Dependence Speculation

# Detecting Dependences

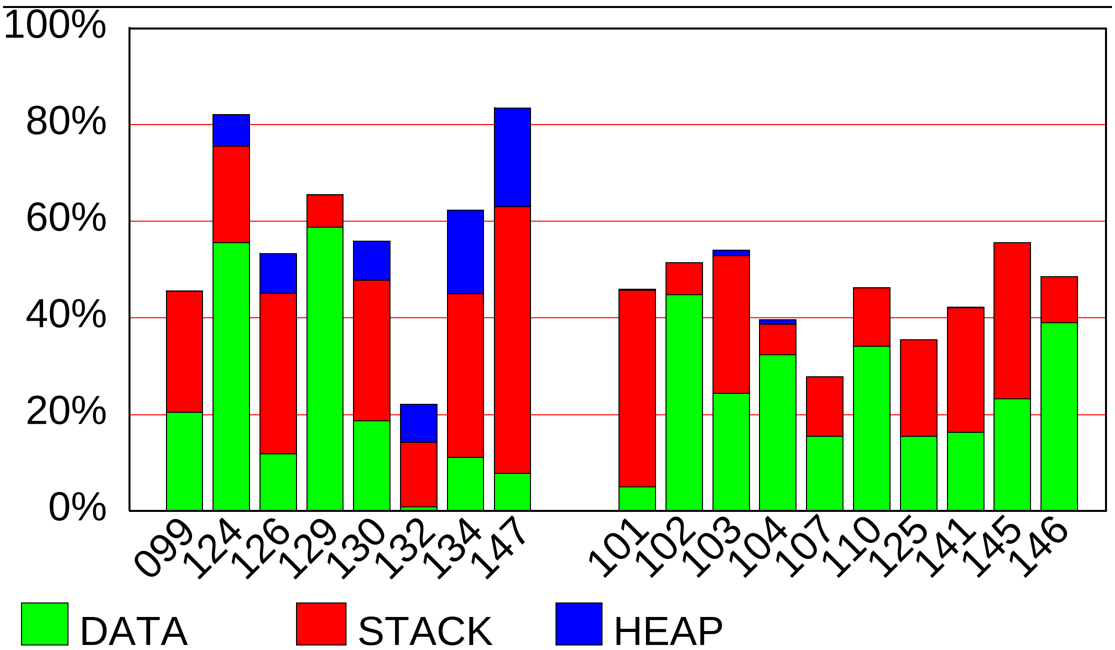



128-entries: Captures Dependences for ~65% of all loads



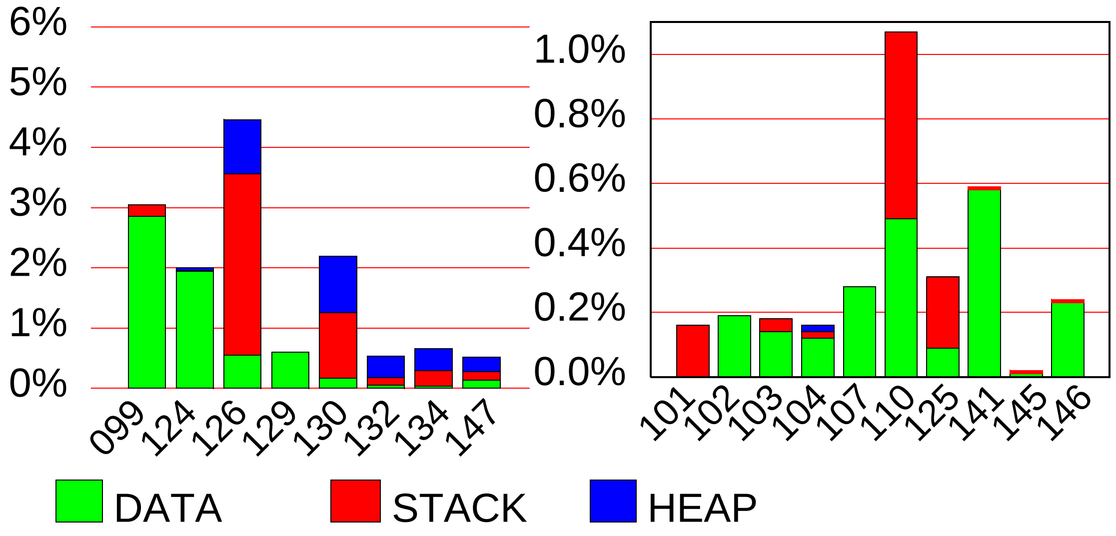


# Cloaking Coverage




Most Dependences Correctly Predicted

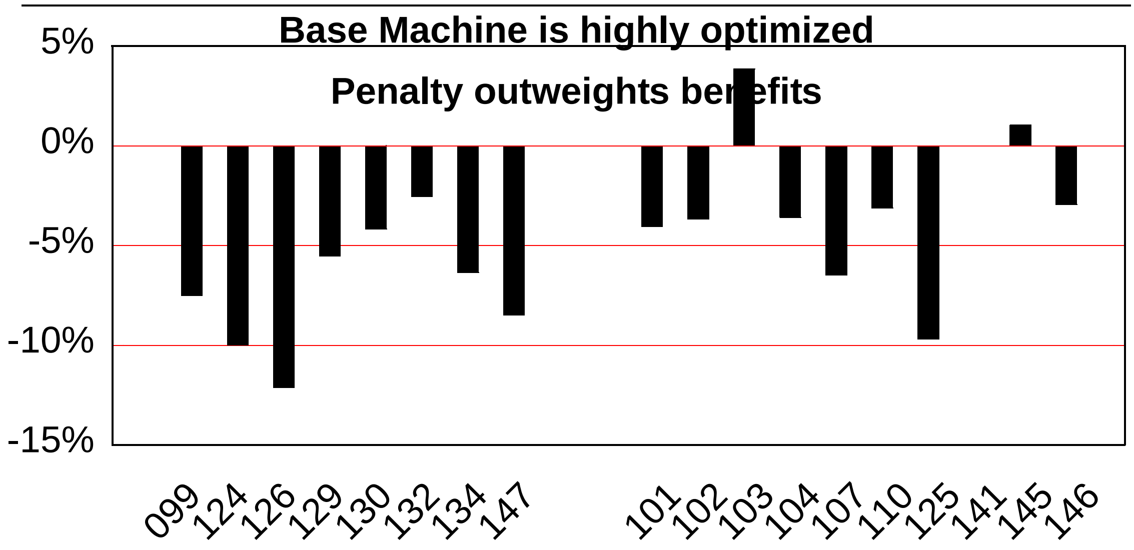
## Cloaking - Misspeculation Rates



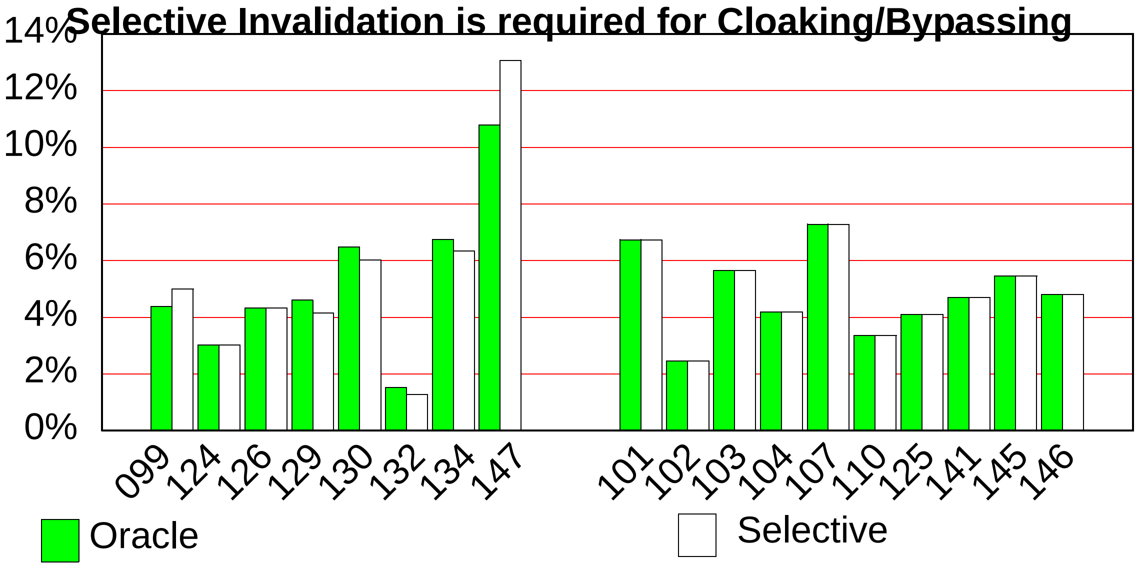





## Performance - Squash Invalidation



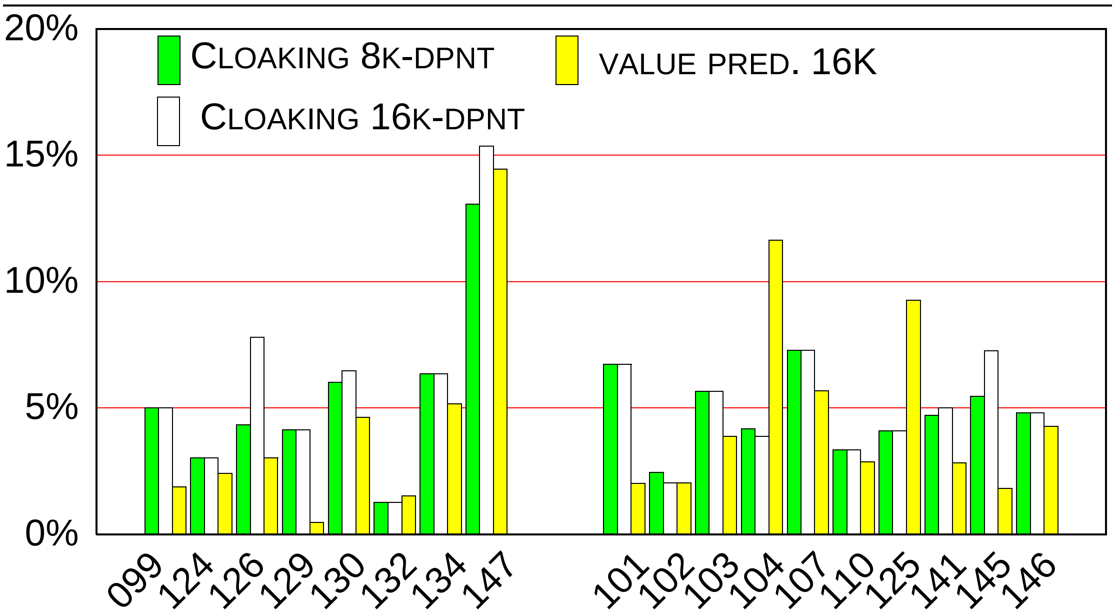
# Performance - Selective Invalidation



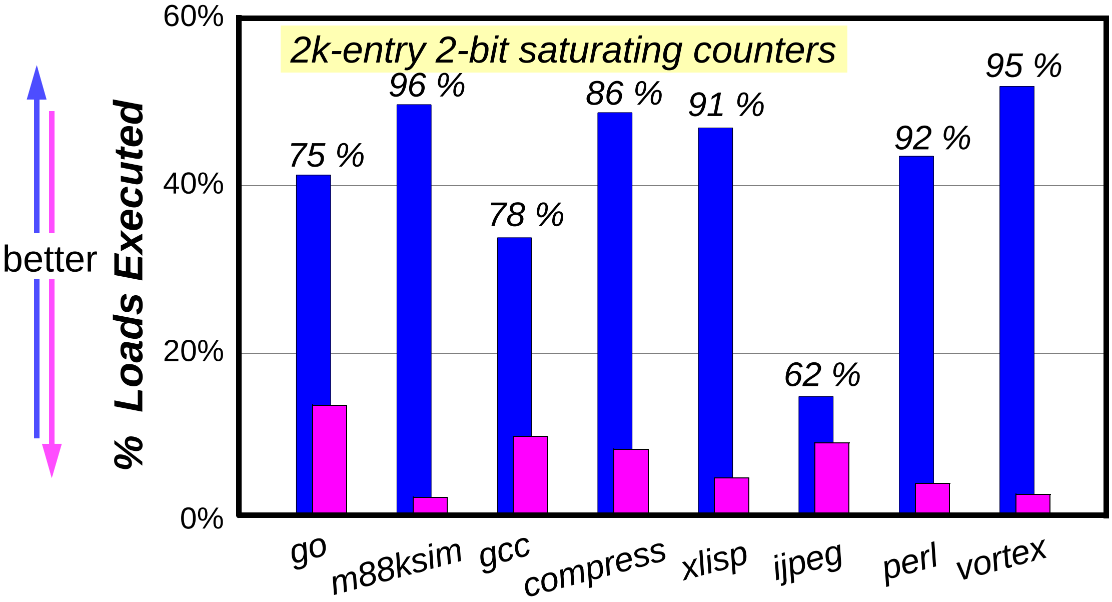





# Cloaking and Load Value Prediction



## Cloaking - Dynamic Loads Serviced



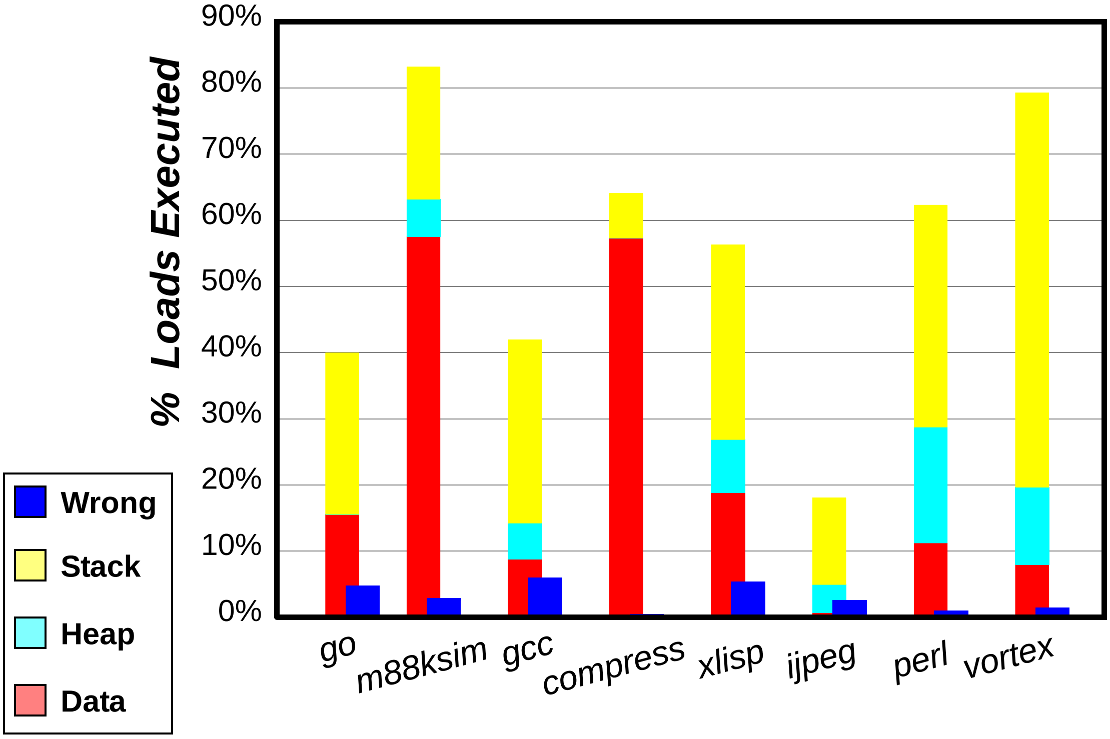

1. First-cut implementation

*2.* Cloaking Extension for Read-after-Read

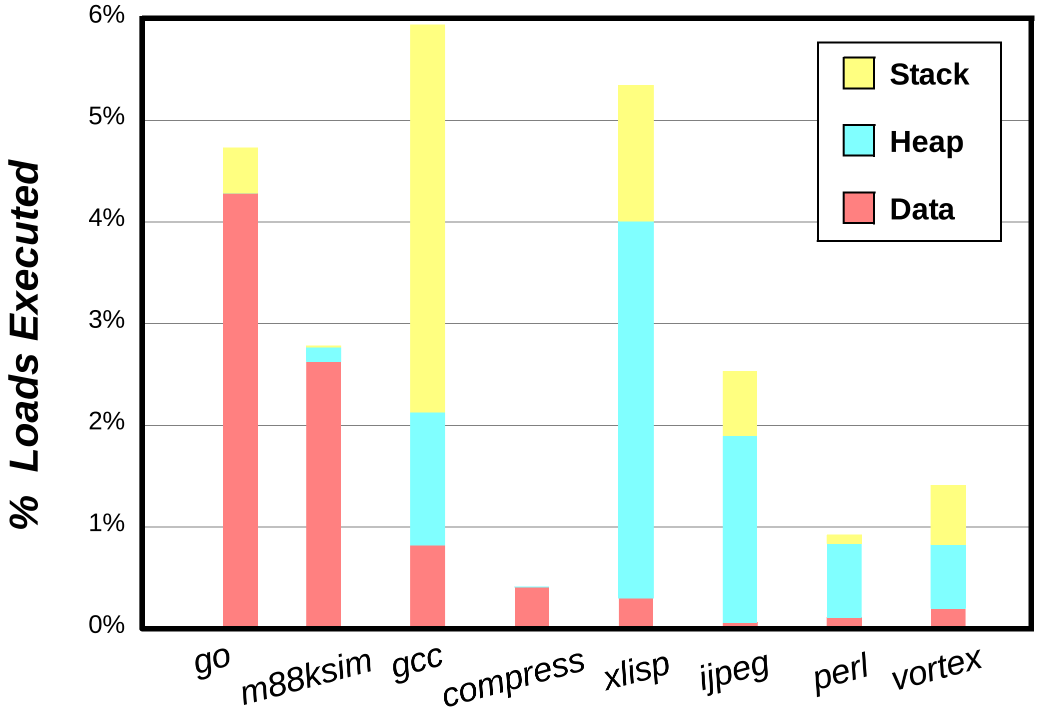




## Cloaking - Prediction Breakdown



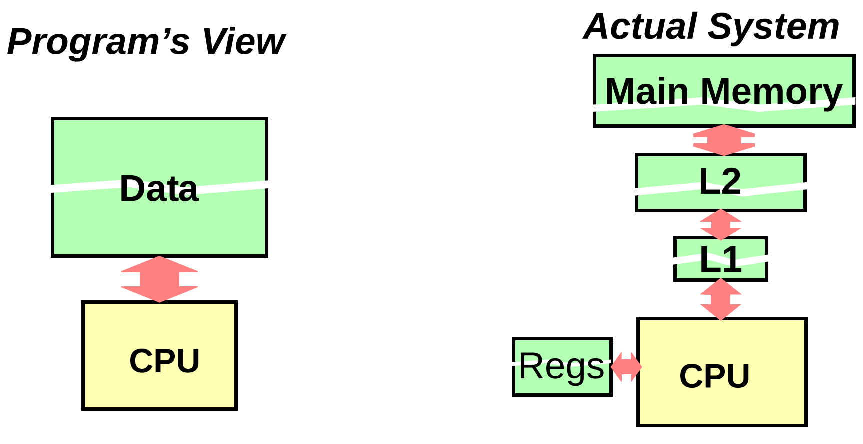
# Cloaking - Misprediction Breakdown





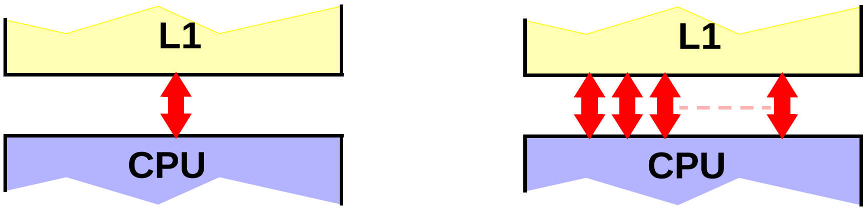



# Back to the Memory System

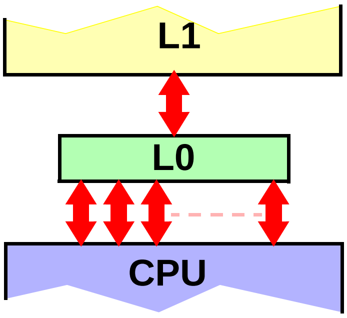


# Increasing Memory Bandwidth

- Parallelism => more bandwidth => more L1 ports




- A very small cache is easier to multi-port




- But, it increases the latency for all accesses that miss in it





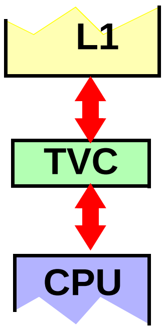
# Transient Value Cache

**Adaptive Placement via Memory Dependence Prediction**

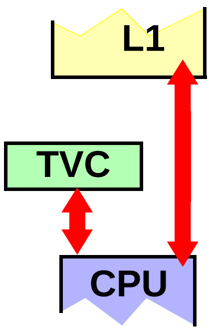
**load:** reads a value from a recent store?

**store:** will be killed by a close-by store?

**Predict Dependence**




**Predict No Dependence**




**A very small cache captures ~ 55% of all memory accesses**

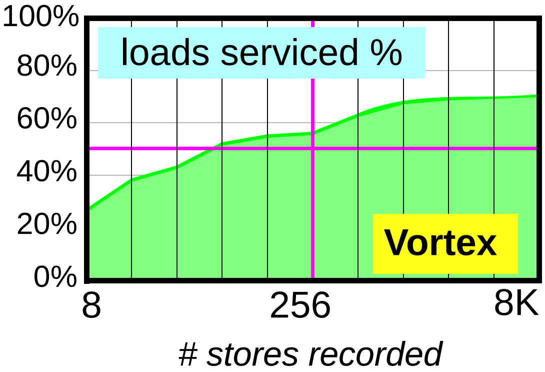
**Without adding latency to all other accesses**

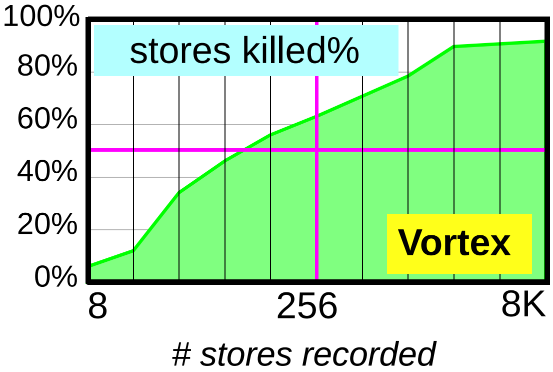
# Transient Value Cache

Data Cache ports: becoming expensive but more are needed

**Observations:** Many loads get their value from a recent store

Many stored values are quickly killed




A 256-FA word cache can service ~50% of loads, ~60% of stores

+ **Hit:** No need to consume L1 ports

- **Miss:** Latency increases



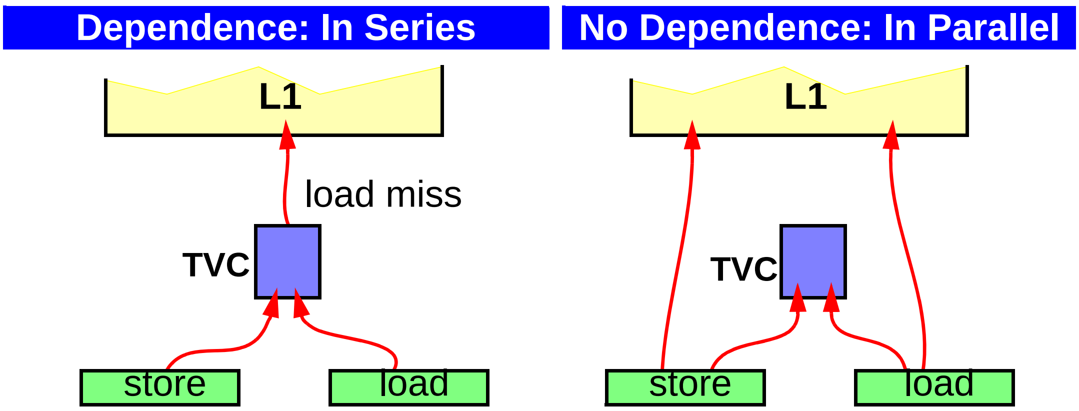


# Transient Value Cache

**Adaptive Placement via Memory Dependence Prediction**

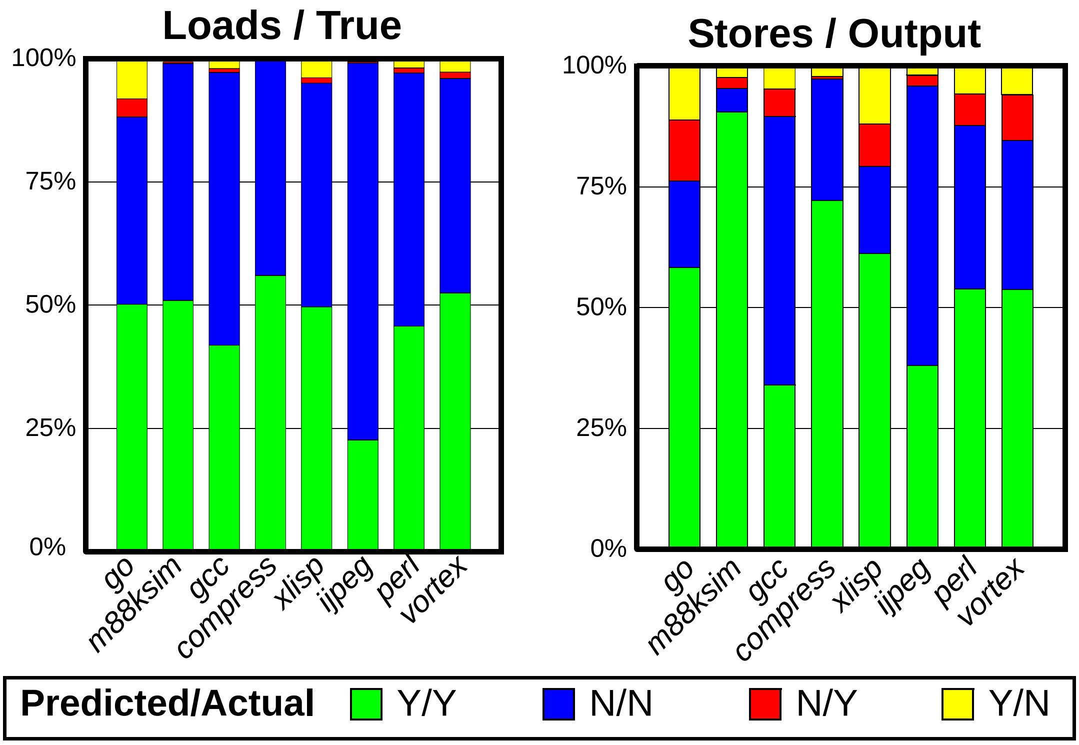
**load:** reads a value from a recent store?

**store:** will be killed by a close-by store?




- Accuracy: Loads > 90%, miss ~%2 max %5, Stores ~ 80%
- Can be combined w/ the Detection Table needed for Cloaking
- Extension to handle Read-*after*-Read (load-load)

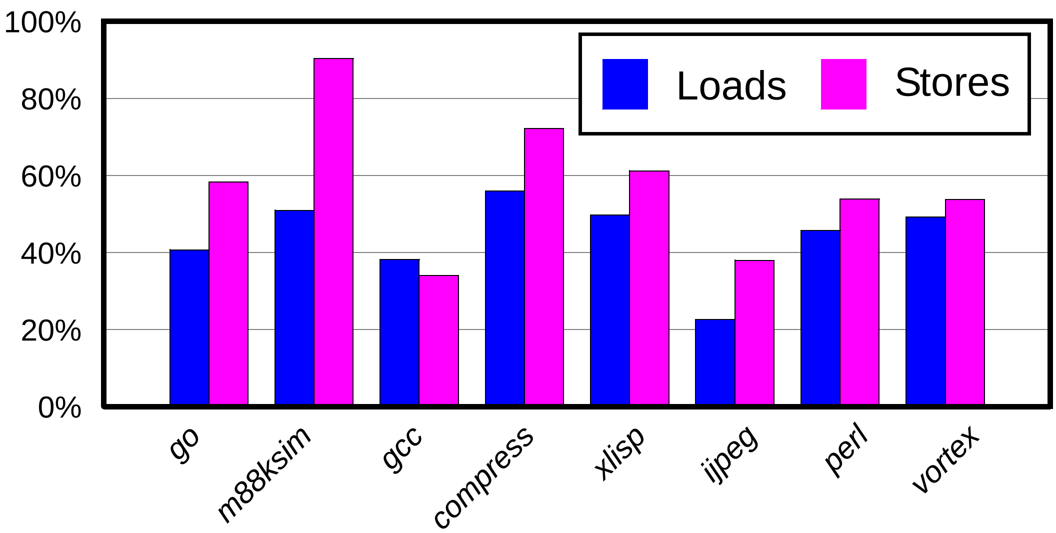
# True/Output Dependence Prediction








# TVC - Reduction in Accesses




**Pessimistic model: last 256 stores - not last 256 addresses**

## Evaluation Parameters

8-way superscalar

64 inst. window

16 entry write buffer

32K data cache/2-way SA/8-way interleaved/16 cycle miss

Same instruction cache

4 memory ports

Perfect disambiguation: cloaking can be used for synchronization.

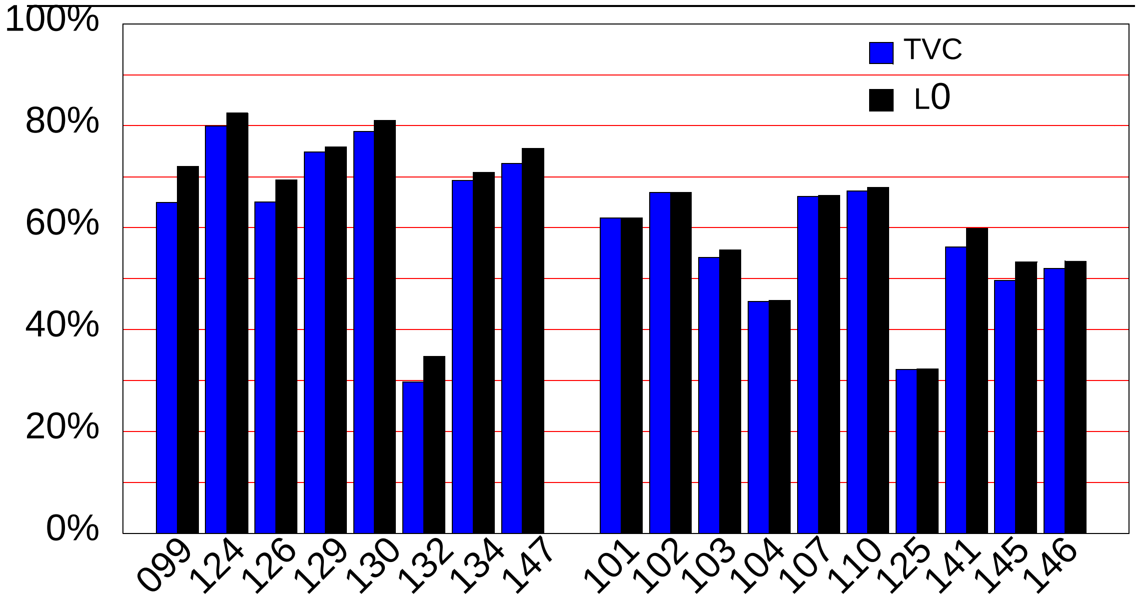
**Mechanism**

- Perfect prediction over last 256-stores
    - *must see dep. at least once*
- 256-word FA Synonym File
- 256-word fully associative TVC/8-ports

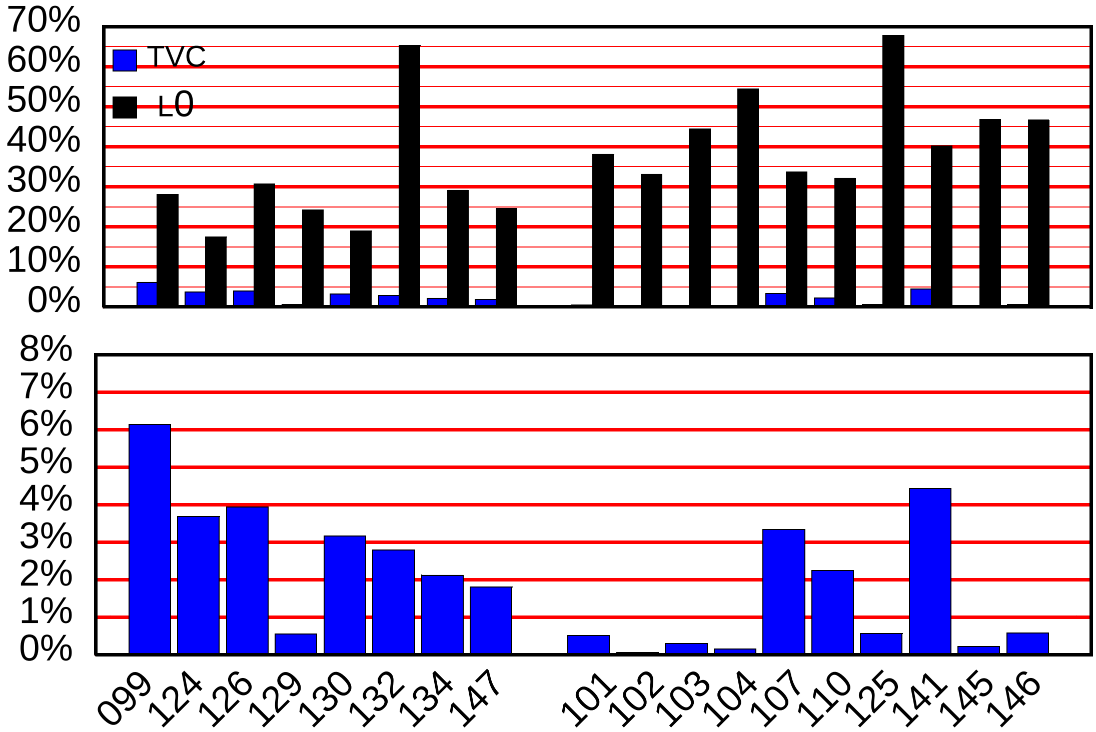




## TVC vs. L0 - Hit Rates



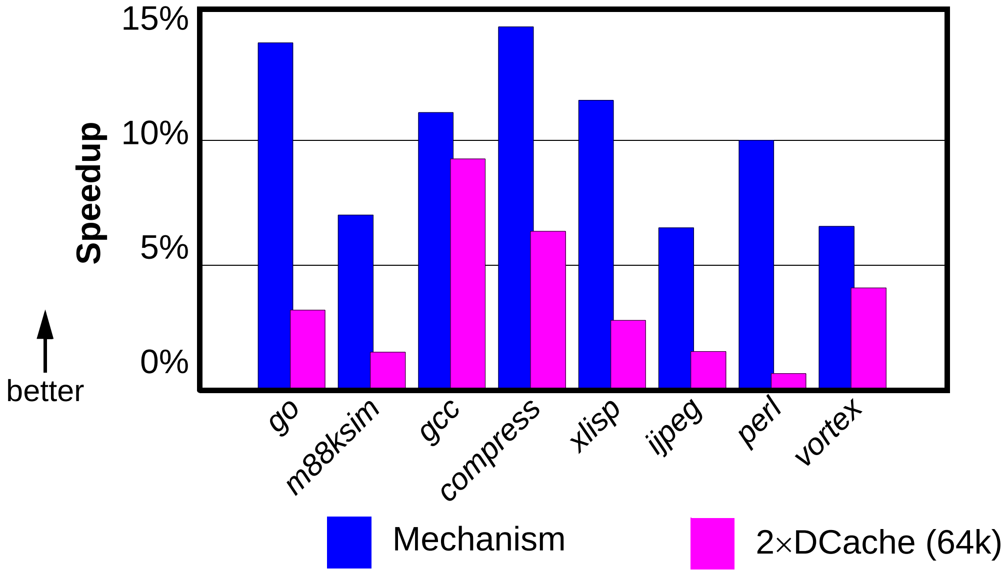
# TVC vs. L0 - "Miss" Rates








# Performance




**There is a point where:**

**Is better to allocate real-estate for our mechanisms**

# More Performance