Profile-Guided Post-Link Stride Prefetching

C-K Luk, Robert Muth, Harish Patil, Richard Weiss†
P. Geoffrey Lowney, Robert Cohn

Massachusetts Microprocessor Design Center
Intel Corporation

†Department of Computer Science
Smith College

This work was done while the authors were with Compaq

M²DC

intel.

Software-Based Data Prefetching

- A promising method to hide memory latency
  - Most modern processors support data-prefetch instructions
  - Rely on compilers to automatically insert prefetches
- Compiler-based approach works quite well, *but* has 2 limitations:
  - **Targets memory references with statically-known strides**
    - Few in codes with pointers, sparse matrices, malloc()
  - **Needs source code for recompilation**
    - Source code may not be available
      - Legacy codes, libraries, commercial codes
    - Recompilation may not be desired

Our Approach

- Profile-Guided Post-Link Stride Prefetching
  - Profiles address strides that are unknown to compilers
  - Inserts stride prefetches into executables directly
- Substantial performance gains on Alpha
  - 3%-56% speedups in 11 SPEC2000 benchmarks
- Incorporated into Compaq Unix tools
  - Pixie and Spike

M²DC

intel

Outline

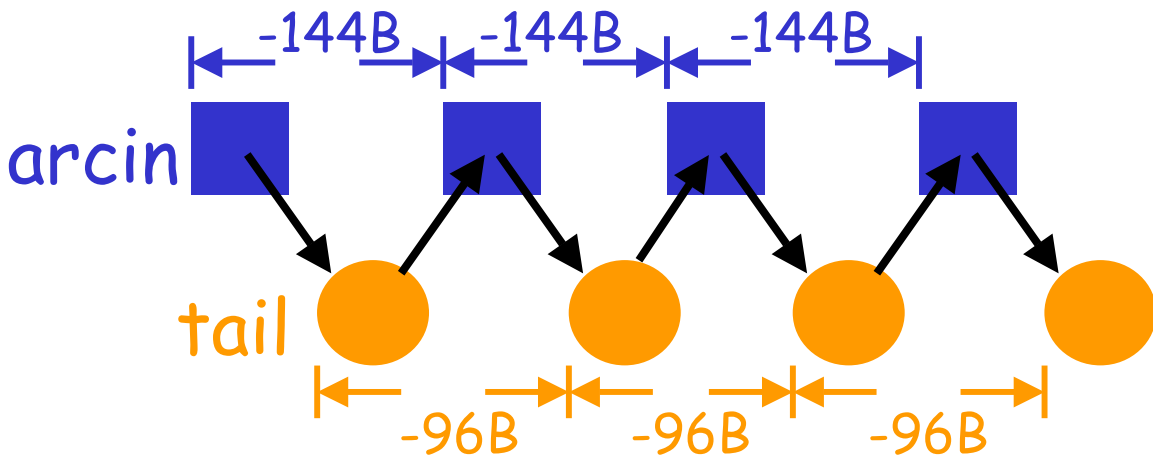
- Case Studies
- Algorithm
- Experimental Results
- Conclusions

M²DC

intel

Address Strides in SPEC2K MCF

```
arc* arcin;
node* tail;
...
while (arcin) {
   tail = arcin->tail;
   ...
   arcin = tail->mark;
}
```



- Account for 26% of the total stall time
- The list structures are sequentially allocated and remain unchanged then

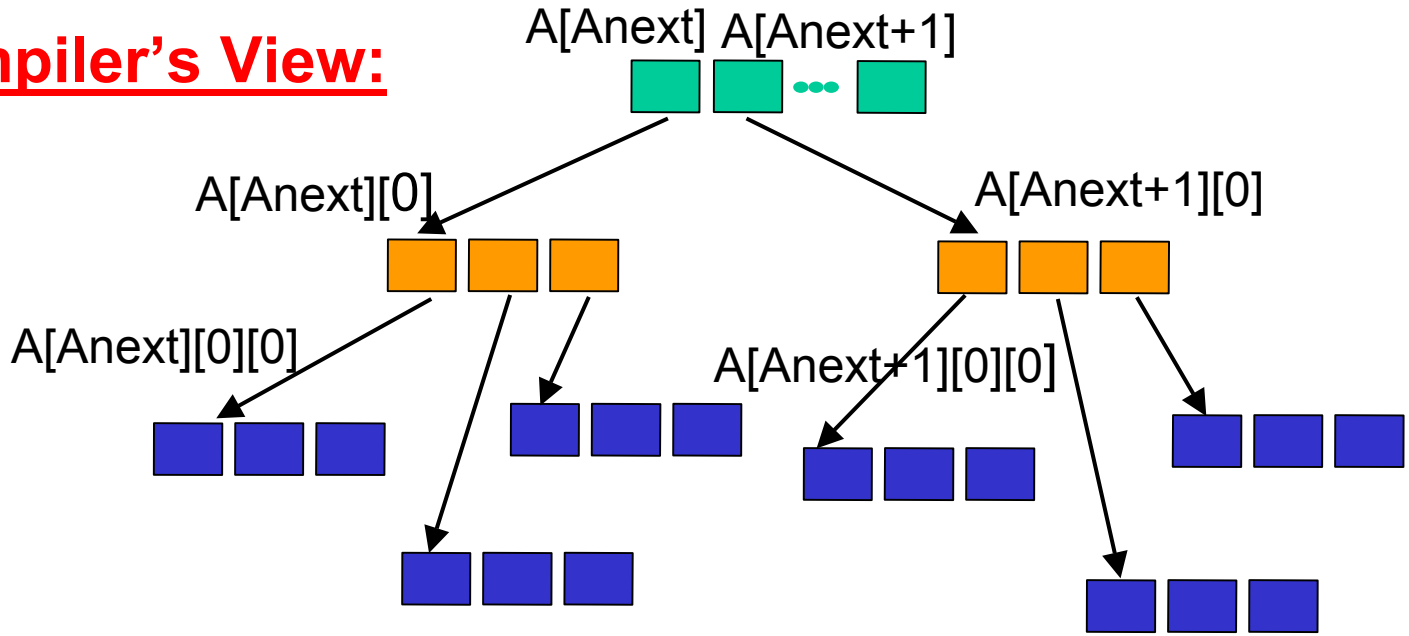
SPEC2K EQUAKE

```
for (i=0; i < nodes; i++) {
        Anext = Aindex[i];
        Alast = Aindex[i + 1];
        …
        Anext++;
        while (Anext < Alast) {
                …
                sum0 +=  A[Anext][0][0] * v[col][0] +
                         A[Anext][0][1] * v[col][1] +
                         A[Anext][0][2] * v[col][2];
                sum1 +=  A[Anext][1][0] * v[col][0] +
                         A[Anext][1][1] * v[col][1] +
                         A[Anext][1][2] * v[col][2];
                sum2 +=  A[Anext][2][0] * v[col][0] +
                         A[Anext][2][1] * v[col][1] +
                         A[Anext][2][2] * v[col][2];
                …
                Anext++;
        }
        …
}
```

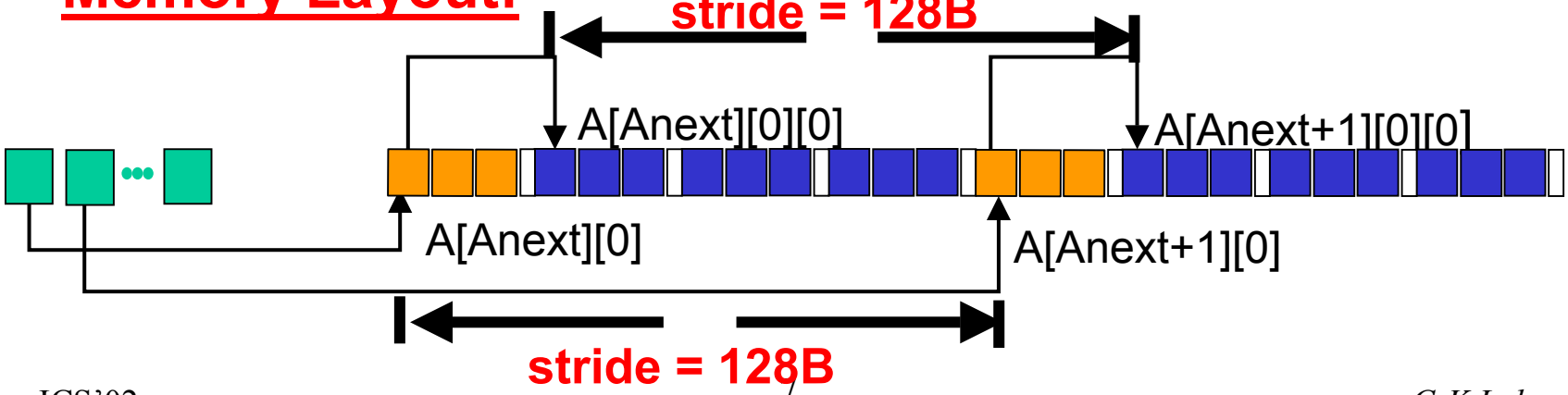
70% of total stall time

Address Strides in EQUAKE

Compiler's View:



Memory Layout:



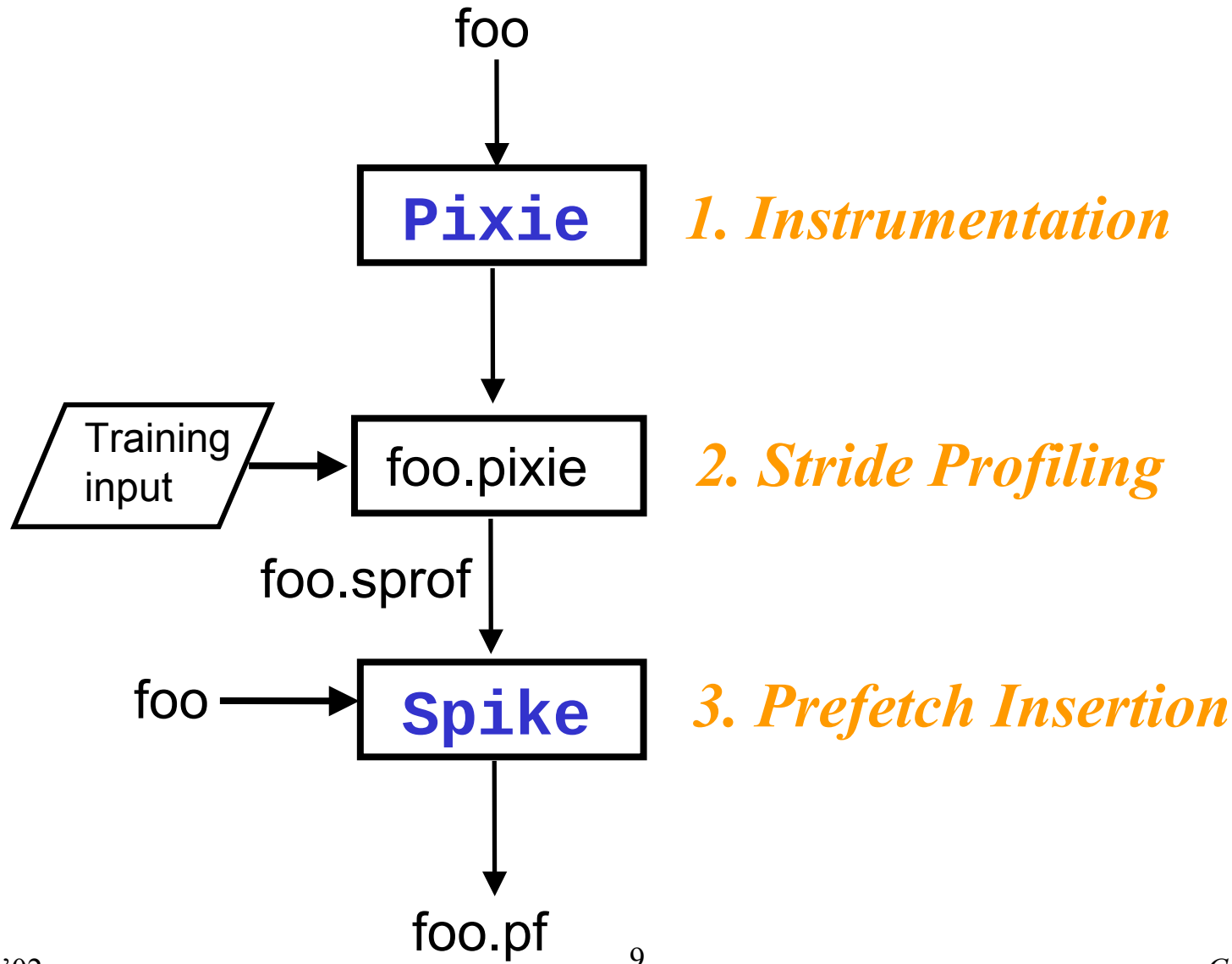
Outline

- Case Studies
- Algorithm
- Experimental Results
- Conclusions

M²DC

intel

Algorithm: 3 Major Steps



Step 1: Instrumentation

Two Issues:

- ***Which** loads do not need to be instrumented?*
  - Scalar (i.e. base registers are either $gp or $sp)
  - Compiler prefetched (i.e. loads with static strides)

- ***Where** should instrumentation be placed?*
  - At the points where base registers are defined

M²DC

intel

Instrumentation (Example)

Naive Instrumentation

```
R1 <- R3 + R4 ;
R2 <- R5 + R6;
// R1 and R2 aren't redefined here
StrideProfile(R1 + 24);
R3 <- load 24(R1);
StrideProfile(R2 + 64);
R4 <- load 64(R2);
StrideProfile(R1 + 48);
R5 <- load 48(R1);
StrideProfile(R2 + 96);
R6 <- load 96(R2);
```

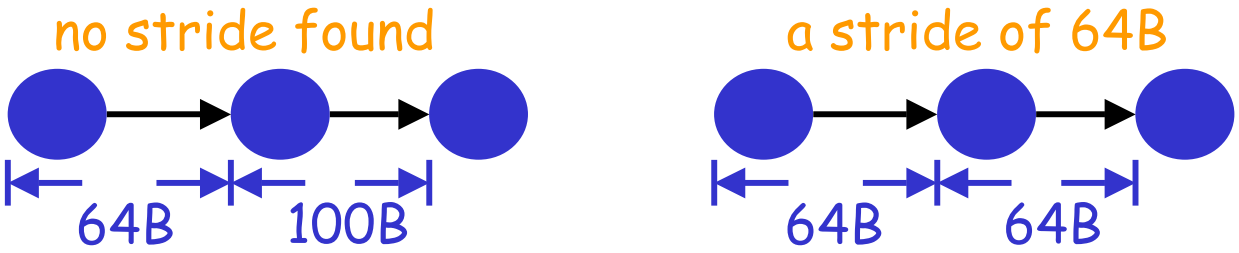
Optimized Instrumentation

```
R1 <- R3 + R4 ;
R2 <- R5 + R6;
StrideProfile(R1, R2);
// R1 and R2 aren't redefined here
R3 <- load 24(R1);
R4 <- load 64(R2);
R5 <- load 48(R1);
R6 <- load 96(R2);
```

☞ ***Reduce profiling overhead (both code size and runtime)***

Step 2: Stride Profiling

- A stride is recognized if it occurs twice in a row



  - Up to 10 strides recorded per load
  - The average **run-length** of each stride is also recorded
    - # consecutive instances that share the same stride

- Complete vs. sampled profiling
  - Complete: profile the entire execution (accurate but slower)
  - Sampled: profile part of the execution (faster but less accurate)
    - First *N* instances
    - Periodic

Step 3: Prefetch Insertion

Three Tasks:

- Choosing from Multiple Strides
- Computing Prefetching Distances
- Prefetch Minimization

M²DC

intel

Choosing from Multiple Strides

- Distribution of strides for each static load:
  - SPECINT: The most frequent stride happens **~90%** of time
  - SPECFP: The most frequent stride happens **~70%** of time

- Two possible approaches:
  - Computing strides at run-time
    + adaptive
    - expensive (especially when spilling is introduced)
  - **Selecting the most frequent stride**

    ☞ ***Inexpensive yet achieves most of the benefits***

M²DC

intel

Computing Prefetching Distances

- Number of loop iterations to fetch ahead

```
while (cur) {
   cur = cur->next;
   prefetch(cur + stride*D);
   /* other work to do */
}
```

$$D = \left\lceil \frac{\text{Latency(cur->next)} \times \text{IPC}}{\text{Length of the loop body}} \right\rceil$$

- Also take the stride's run-length ***R*** into account
  - Observation: first ***D*** instances are not prefetched
    => If $R \leq D$, we want a smaller prefetching distance (e.g., $\frac{R}{2}$)

Prefetch Minimization

Goal:

- Remove prefetches of lines that have been prefetched

Example:

Before Minimization

```
R1 <- R1 + 1024;
prefetch 16(R1);
prefetch 64(R1);
if (…) then
   prefetch 32(R1); …
else
   prefetch 1024(R1); …
endif
prefetch 72(R1);
prefetch 118(R1); …
```

After Minimization

```
R1 <- R1 + 1024;
prefetch 16(R1); //Beginning a 3-line span
prefetch 80(R1); // 1 line from the beginning
if (…) then
   // prefetch 32(R1) combined into the span
else
   prefetch 1024(R1); …
endif
// prefetch 72(R1) combined into the span
prefetch 118(R1); …
```

Outline

- Case Studies
- Algorithm
- Experimental Results
- Conclusions

M²DC

intel

Experimental Framework

Machine: **a DS20E Alpha Workstation**

- 667-MHz 21264 processor
- Memory hierarchy:
  - 64KB D-cache, 8MB L2-cache, 2GB memory
  - Miss latencies: 12+ cycles to L2, 80+ cycles to memory

Benchmarks: **entire SPEC2000 suite**

- Profiled on “training”, run on “reference”

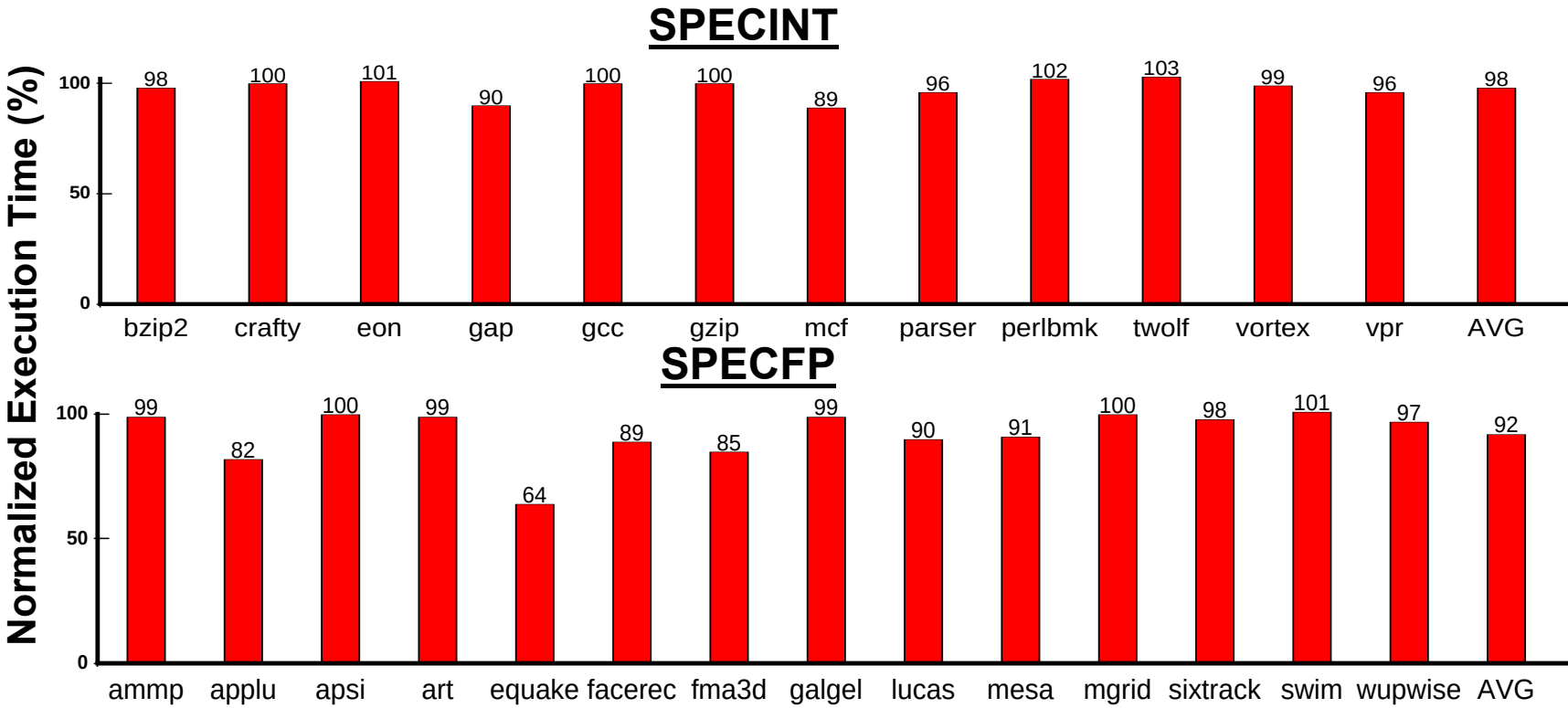
Baseline: **Compaq C & Fortran Compilers “–O5”**

- Implemented classic array-based prefetching

M²DC

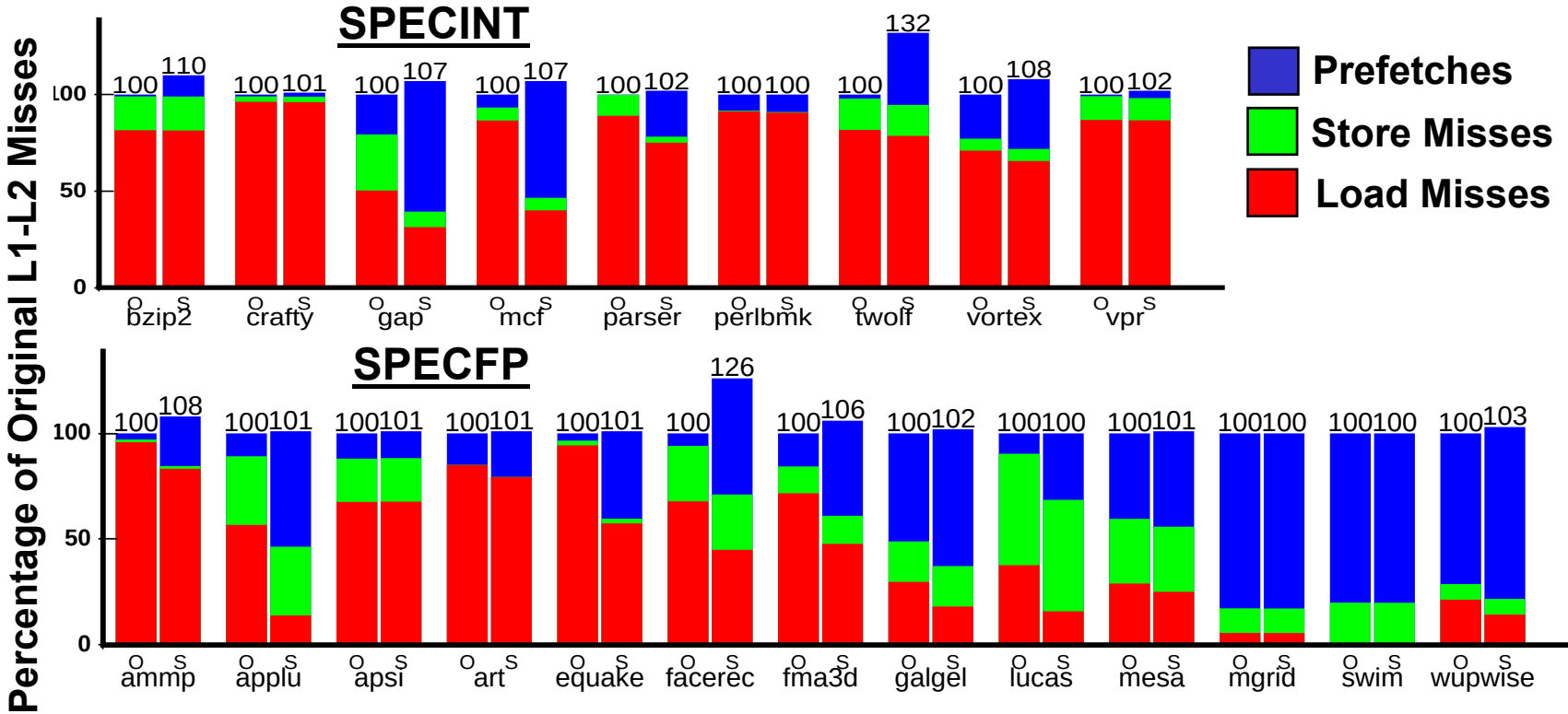
intel

Performance of Stride Prefetching



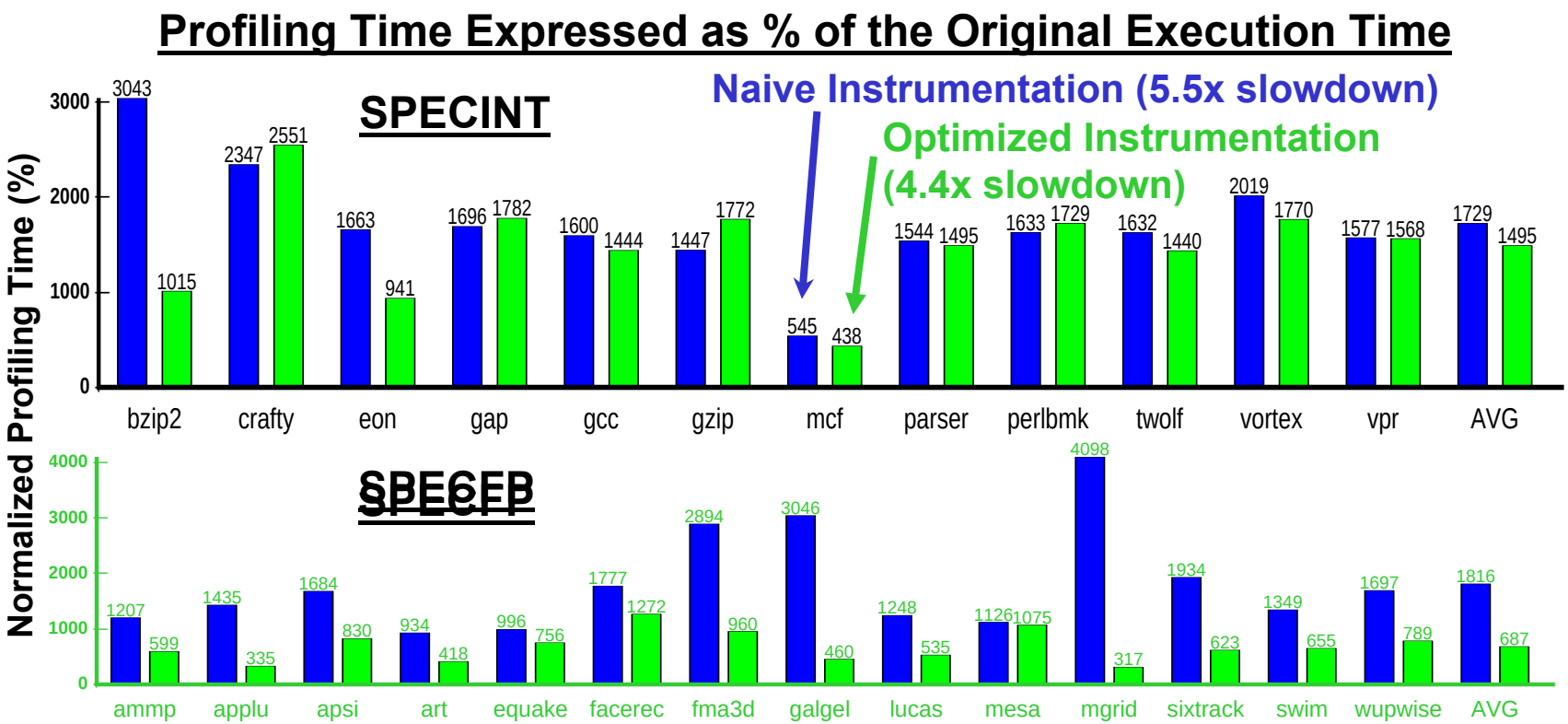
- **Substantial speedups:**
  - 3% - 56% speedups in 11 benchmarks
  - ≥10% speedups in 8 benchmarks
- Only mild slowdowns:
  - 1% - 3% in 4 benchmarks

Where Do the Gains Come From?



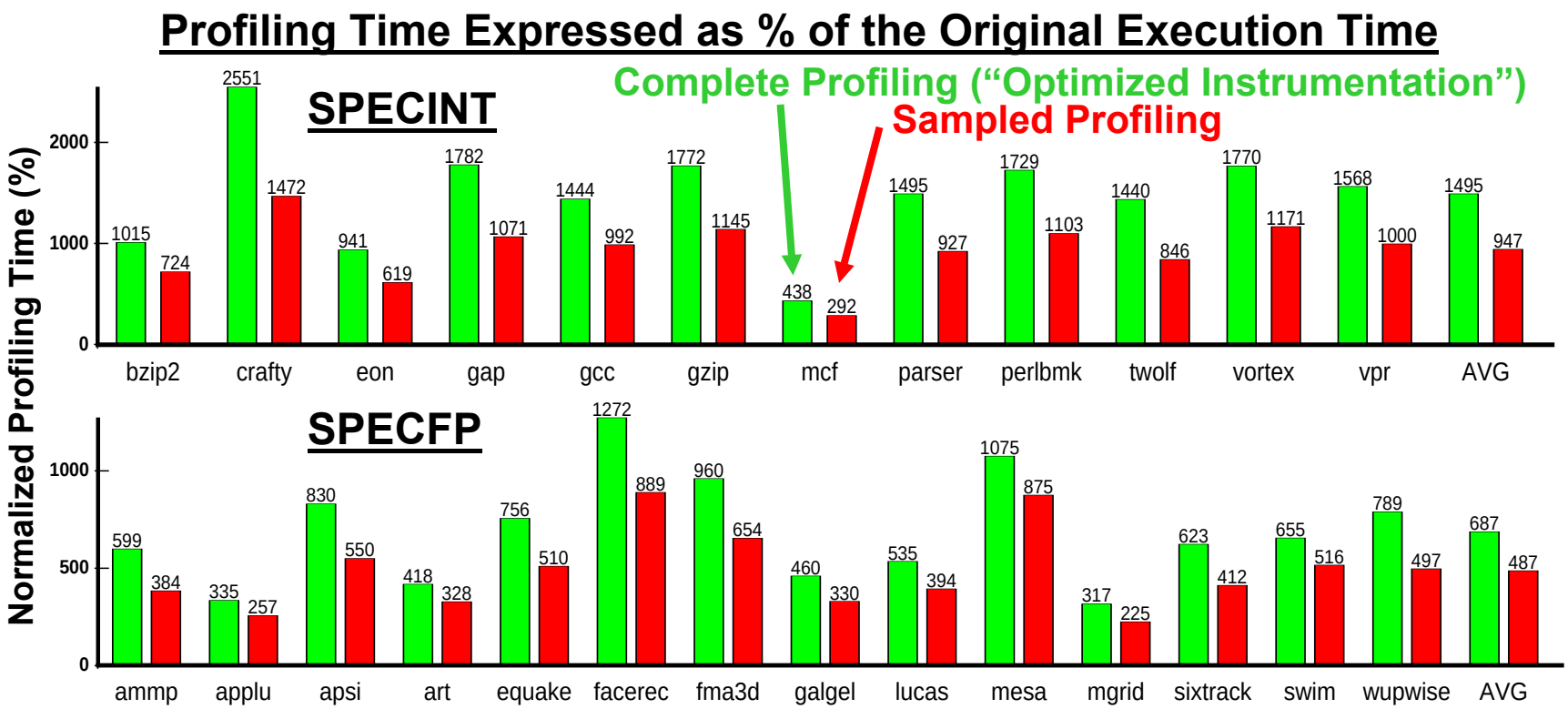
- Wanted: load misses converted into prefetches
  - Significant in gap, mcf, applu, equake, facerec, fma3d, lucas
- Don't want too many extra misses
  - A problem in twolf and facerec

Overhead of Stride Profiling



- Similar overhead as other sw value profilers
- Optimized instrumentation is quite helpful
  - Reduces the overhead by two-thirds for SPECFP

Effectiveness of Sampled Profiling



- Sampling speeds profiling up by ~50%
- Almost no performance degradation

Conclusions

- **Broaden the scope of software prefetching:**
  - Use profiling to detect statically unknown address strides
  - Apply prefetching without source code and recompilation
- **Significant performance gains:**
  - 3% - 56% speedups in 11 SPEC2K benchmarks on Alpha
- **Profiling overhead can be largely reduced:**
  - By instrumentation heuristics, sampling
- **Incorporated into Compaq Unix tools**

M²DC

intel

M²DC

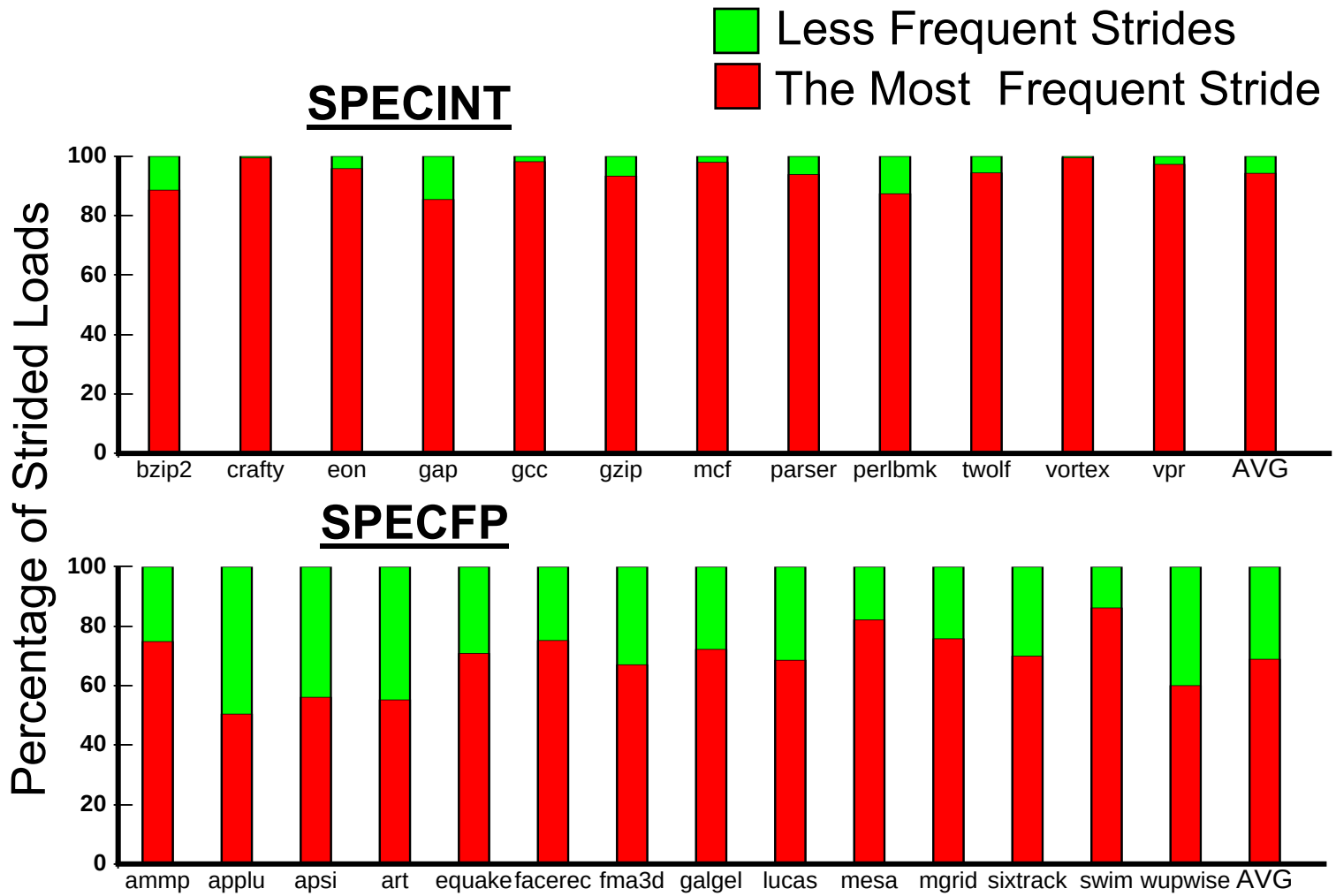
intel

☞ ***Try it out yourself!***

http://www.tru64unix.compaq.com/spike

Backups

Distribution of Strides



Handling Multiple Strides

(a) Original

```
R2 <- load 16(R1)
```

(b) Choosing the Most Frequent Stride

```
// k = most_frequent_stride * prefetching_distance + 16
prefetch k(R1);
R2 <- load 16(R1);
```

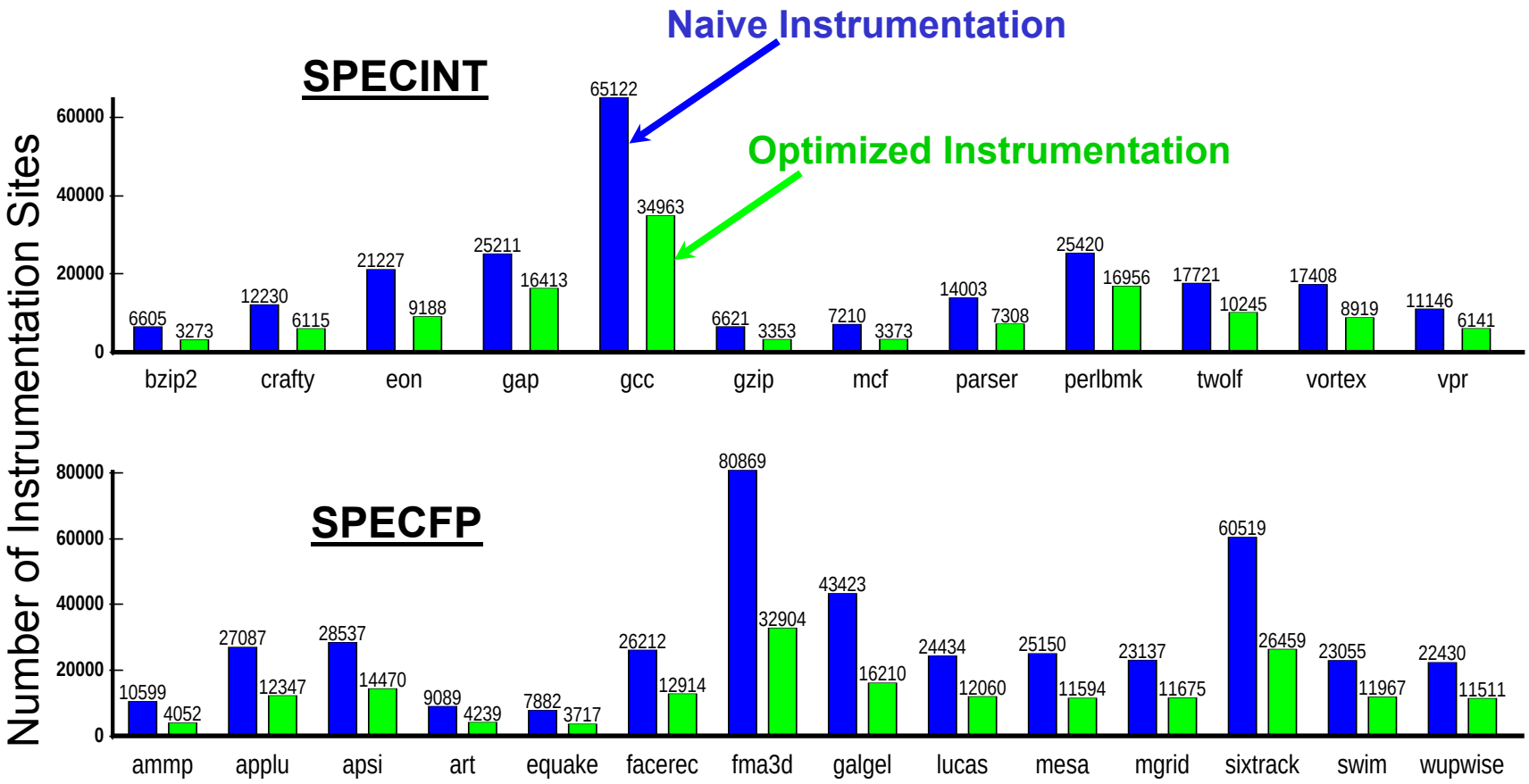
(c) Run-time Stride Detection

```
t <- R1 - t; // t has been holding the previous value of R1.
             // So, (R1 - t) is the stride
t <- prefetching_distance * t;
t <- R1 + t;
prefetch 16(t);
t <- R1; // save R1 for computing the next stride
R2 <- load 16(R1);
```

DS20E's Memory Hierarchy

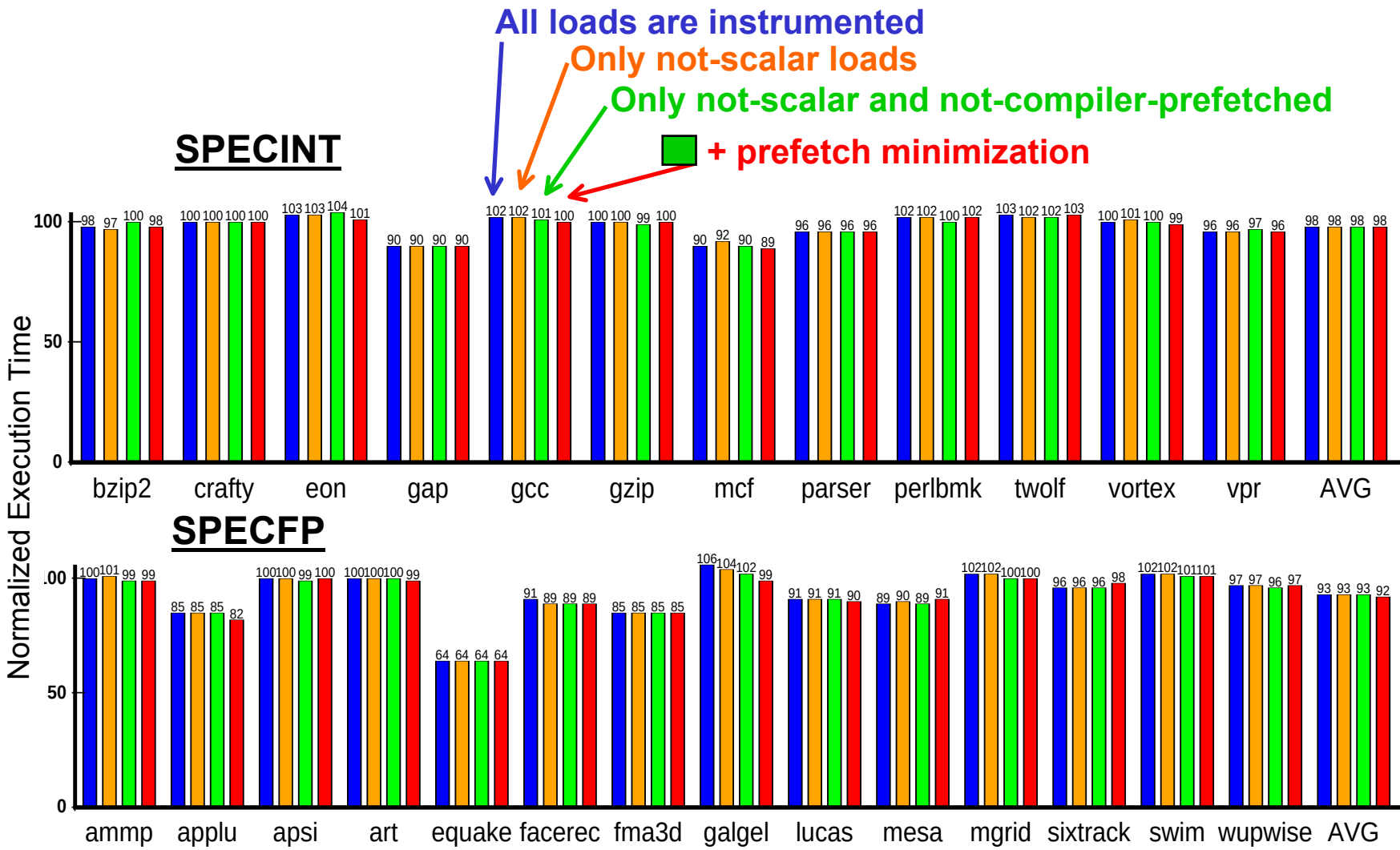
Line Size	64 bytes
I-Cache	64 KB, 2-way set-associative
D-Cache	64 KB, 2-way set-associative
Memory Parallelism	32 in-flight loads, 32 in-flight stores, 8 in-flight cache block fills, 8 cache victims
Off-Chip, Unified L2-Cache	8MB, direct-mapped
L1-to-L2 Miss Latency	12+ cycles
L1-to-Memory Miss Latency	80+ cycles
L1-to-L2 Bandwidth	6.9 GB/sec
L2-to-Memory Bandwidth	2.6 GB/sec

Number of Instrumentation Sites

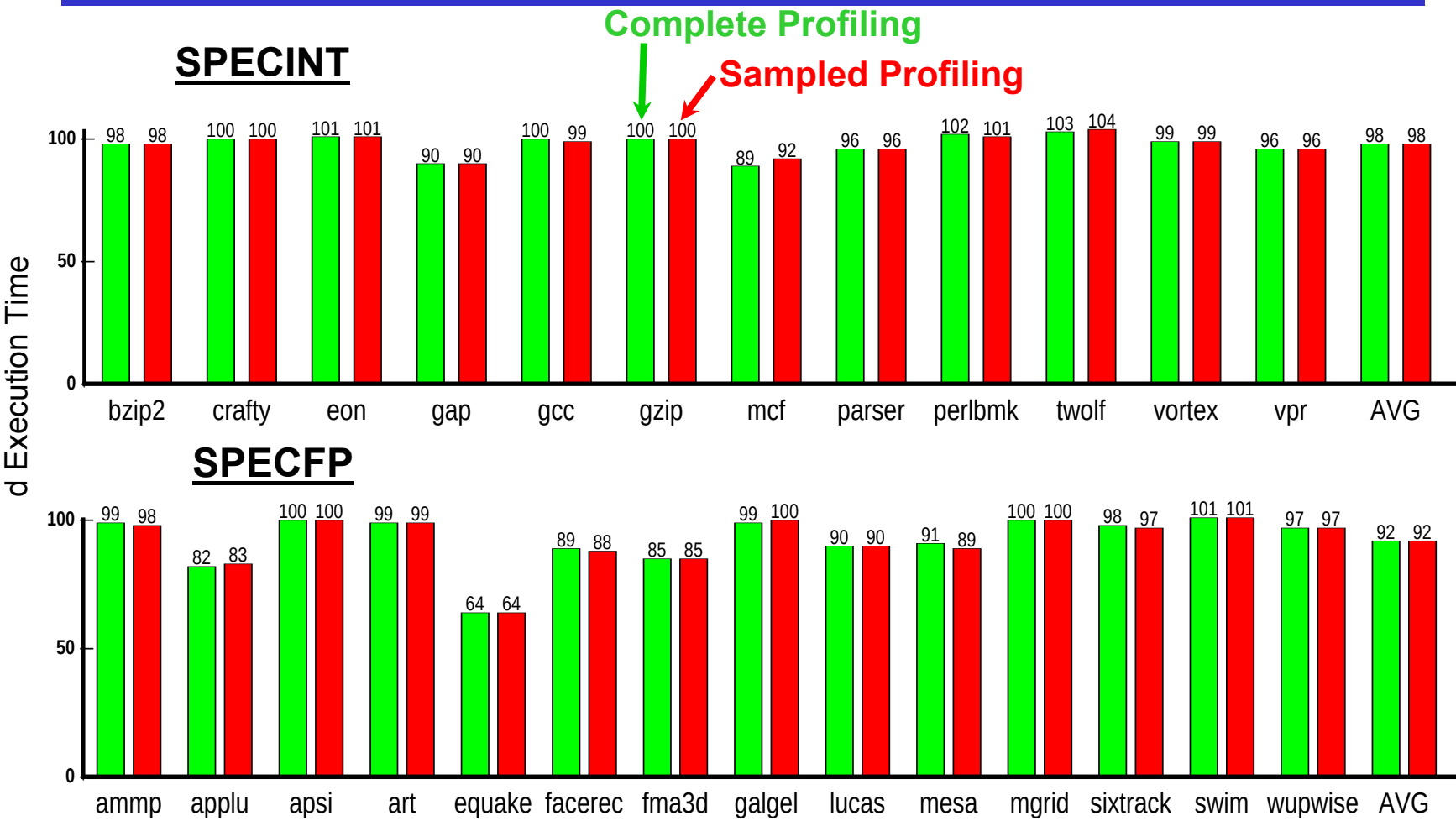


- Reduced by a half or more in most cases

Instrumentation Heuristics and Prefetch Minimization



Performance of Sampled Profiling



- Almost no performance degradation

HW vs. SW Stride Prefetching

Hardware Stride Prefetching

- Chen and Baer's scheme:
  - Computes strides at runtime using special hardware
  - Has a table that associates strides with loads

- **Advantages of HW stride prefetching:**
  - Requires no recompilation or post-link optimizations
  - No instruction overhead
  - Adaptive
- **Advantages of SW stride prefetching:**
  - No limit on how many static loads can be remembered
  - More programmable (e.g., choosing prefetching distances)
  - Applicable to most existing machines

Other Software Stride Prefetching

- Compiler-Inserted Stride Prefetching
  - Stoutchinin *et al*'s approach:
    - Focuses on prefetching recurrent pointer updates
    - Does not use profiling
  - Wu *et al*'s approach:
    - Does stride profiling at the source level
    - Reduces profiling overhead by code transformation
- Post-Link Stride Prefetching
  - Barnes *et al*'s approach:
    - Uses cache simulation to guide prefetch insertion