# How to Make Loop-level Data Dependence Profiling Fast and Memory Efficient?

Hongtao Yu, Zhiyuan Li

Department of Computer Science
Purdue University

# The bigger picture

- How to retrofit existing application software for multicore processors?
  - Difficulties with programmer writing parallel programs
    - Race conditions
    - Memory consistency model
    - Execution model (process, thread, stream, vector, …)
    - Granularity
  - Difficulties with using compilers to help
    - Alias problems
    - Symbolic analysis
    - Input-dependent behaviors
    - etc

- In both cases, a key issue is to understand the inherent data dependences thoroughly

# Loop-level data dependence

```
while (…) {
…
for (i = 0; i <= 65536; i++)
   ftab[i] = 0;
c1 = block[-1];
for (i = 0; i <= last; i++) {
   c2 = block[i];
   ftab[(c1 << 8) + c2]++;
   c1 = c2;}
for (i = 1; i <= 65536; i++)
   ftab[i] += ftab[i-1];
```

```
c1 = block[0];
for (i = 0; i < last;
i++) {
   c2 = block[i+1];
   j = (c1 << 8) +
c2;
   c1 = c2;
   ftab[j]--;
   zptr[ftab[j]] = i;}

…

   for () {
   … = zptr[..]
} /* end while */
```

- Zptr[] and ftab[] are both privatizable;
- No flow dependence actually exists across the while-loop iterations;
- While loop is a doacross kind of loop w/ significant parallelism
- Current compilers cannot determine yet

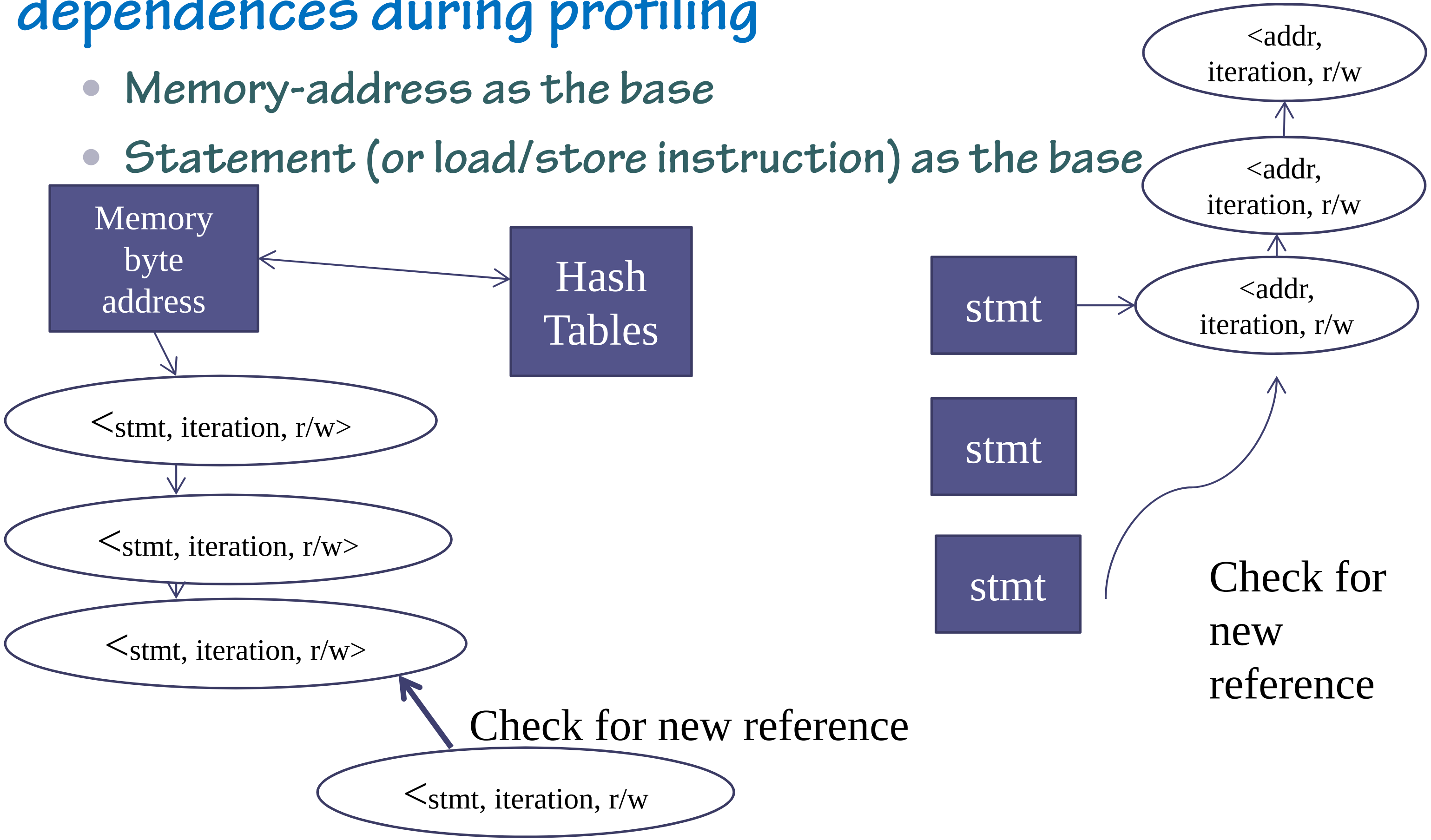
- Data dependence profiling can be an important tool for such understanding
  - Compiler writers can discover areas for improvement
  - Programmers can find code segments to rewrite
  - With speculative parallel execution support, we'll know which loop to speculate
- Unfortunately, when done in a straightforward way, data dependence profiling is *extremely time consuming*
  - Which is perhaps why it has not been widely used in practice
- We look for ways to make it practical
  - Reduce *memory use* (otherwise profiling may not even be feasible)
  - Reduce *time* (by one or two orders of magnitude)

# Techniques to discuss today

- Basic mechanisms
  - Hash tables for memory addresses
  - Regular strides (and limitation)
- Obvious compiler-based techniques to reduce cost
  - Some works but some don't
- "Multi-slicing" (or parallel profiling based on partitioning)
  - Partitioning by alias classes
  - Partitioning by potential dependence edges
  - Input-dependent specialization ("thinned analysis")
- To make profiling much more practical
  - Partial profiling that is sufficient for loop parallelization
- A more efficient mechanism (memory-tags)
  - Need compiler support

# Basic Mechanisms

- Fundamentally there are two ways to check dependences during profiling
  - Memory-address as the base
  - Statement (or load/store instruction) as the base



Memory byte address

Hash Tables

<stmt, iteration, r/w>

<stmt, iteration, r/w>

<stmt, iteration, r/w>

Check for new reference

<stmt, iteration, r/w

stmt

stmt

stmt

<addr, iteration, r/w

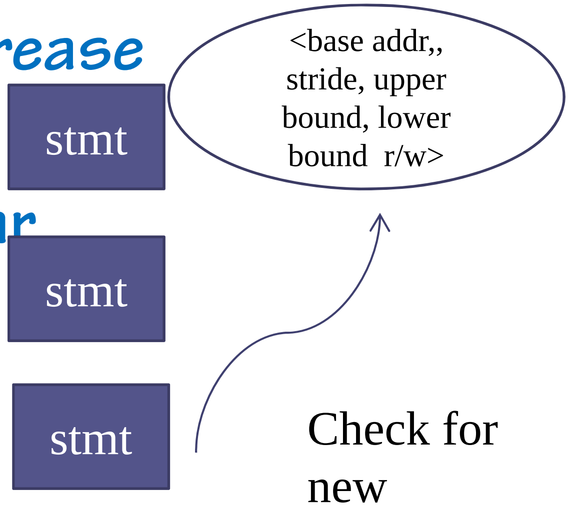
<addr, iteration, r/w

<addr, iteration, r/w

Check for new reference

# Regular strides

- Method called SD3
    - M. Kim, H. Kim, and C.-K. Luk. Sd3: A scalable approach to dynamic data-dependence profiling. In MICRO, 2010
    - When it works, it can speed up profiling by 100x
- Unfortunately, when loop levels increase
    - Strides are often not regular anymore
- When aliasing relationship is unclear
    - It is unsafe to use strides
- Fall back to hash tables



<base addr,, stride, upper bound, lower bound r/w>

stmt

stmt

stmt

Check for new reference

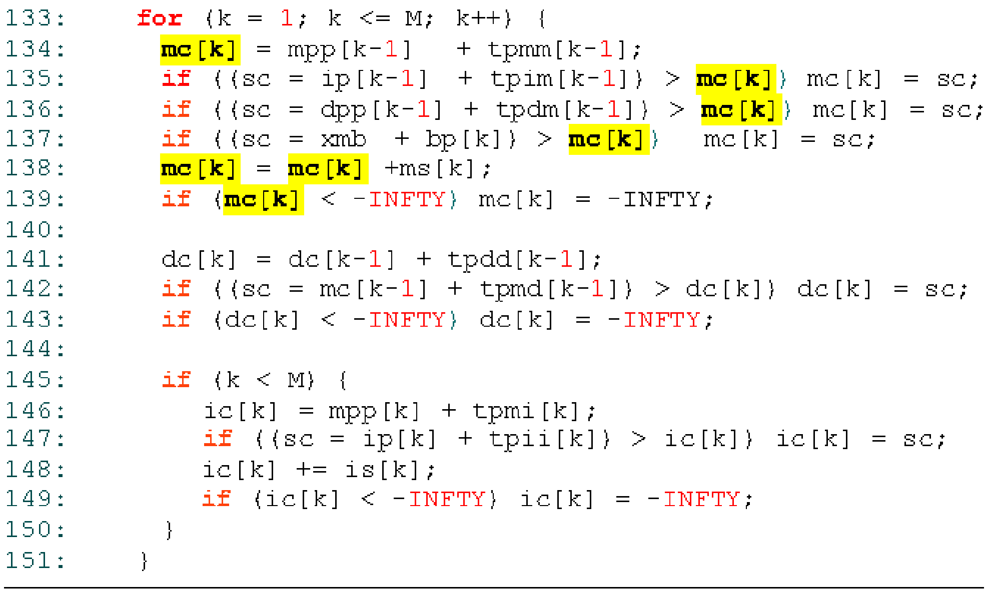
# Compiler-based improvement

- Obviously, if the dependence question for a pair of load/store operations can be answered by the compiler,
  - then it does not need to be checked by profiling
- However, under the (memory-address) hash-table mechanism
  - If a load/store operation has a possible dependence with *any* other operation,
    - then it must be registered at (and checked against) the hash table
- Therefore in C programs, we often end up with registering and checking every memory reference
- Nonetheless, there exists a simple opportunity for improvement

# Equivalence class and its representative for profiling

- Need symbolic analysis to make sure the address is the same in every iteration
- Need control flow information to identify upward-exposed use

A code segment from 456.hmmer

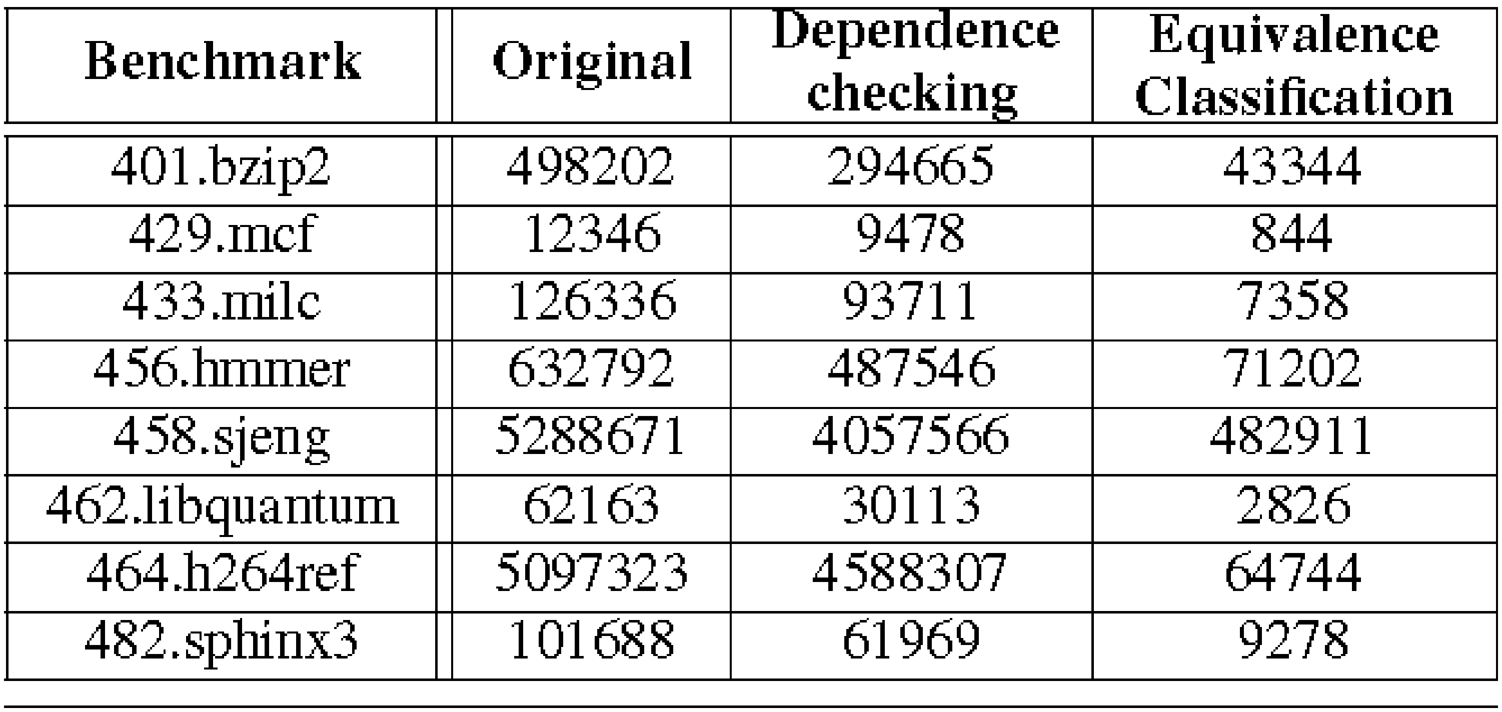


```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1]   + tpmm[k-1];
  if ((sc = ip[k-1]  + tpim[k-1]) > mc[k]) mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
  if ((sc = xmb  + bp[k]) > mc[k])  mc[k] = sc;
  mc[k] = mc[k]  +ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;

  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;

  if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
  }
}
```

# Static count of dependence edges

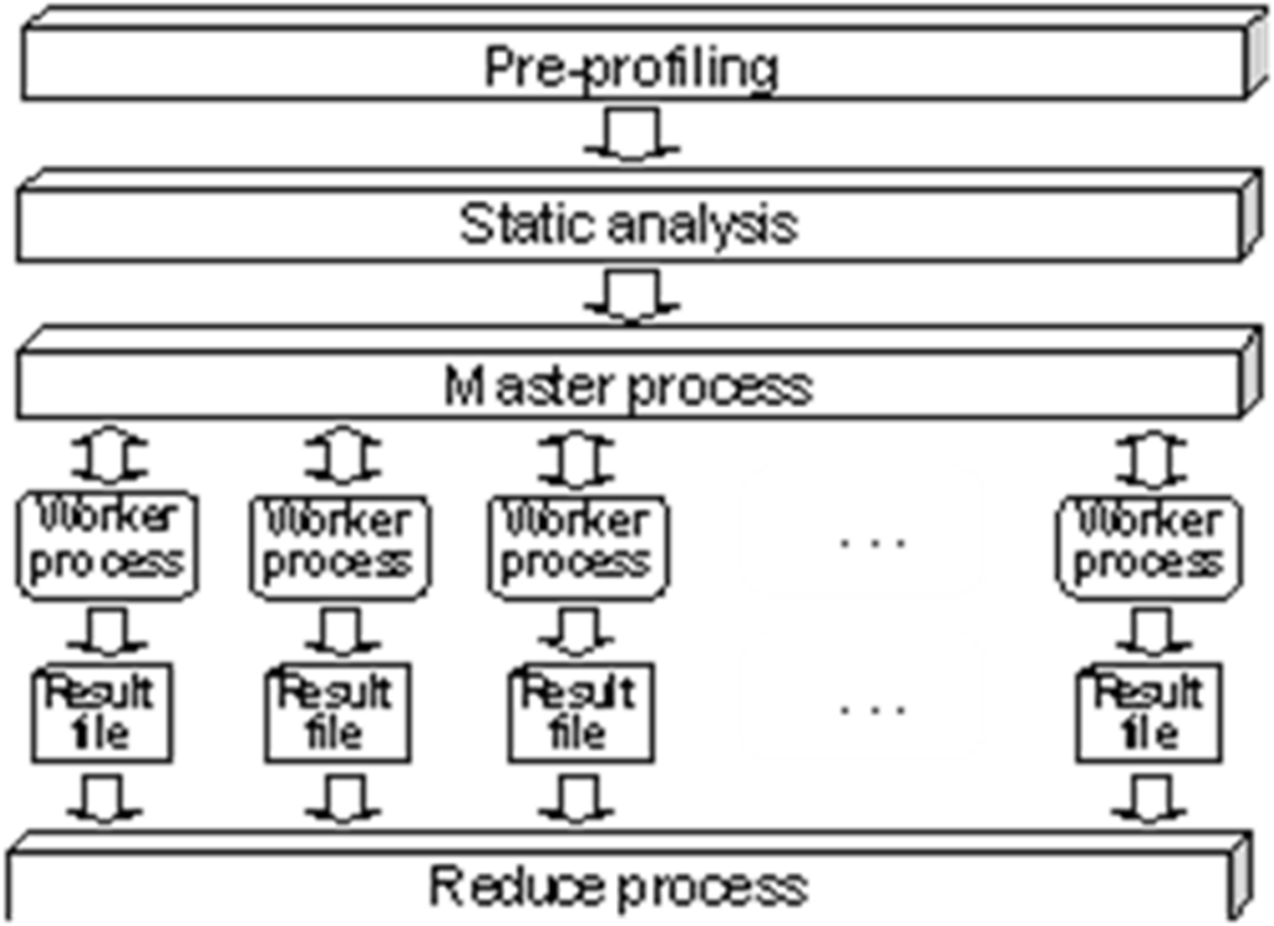


| Benchmark | Original | Dependence checking | Equivalence Classification |
|---|---|---|---|
| 401.bzip2 | 498202 | 294665 | 43344 |
| 429.mcf | 12346 | 9478 | 844 |
| 433.milc | 126336 | 93711 | 7358 |
| 456.hmmer | 632792 | 487546 | 71202 |
| 458.sjeng | 5288671 | 4057566 | 482911 |
| 462.libquantum | 62163 | 30113 | 2826 |
| 464.h264ref | 5097323 | 4588307 | 64744 |
| 482.sphinx3 | 101688 | 61969 | 9278 |

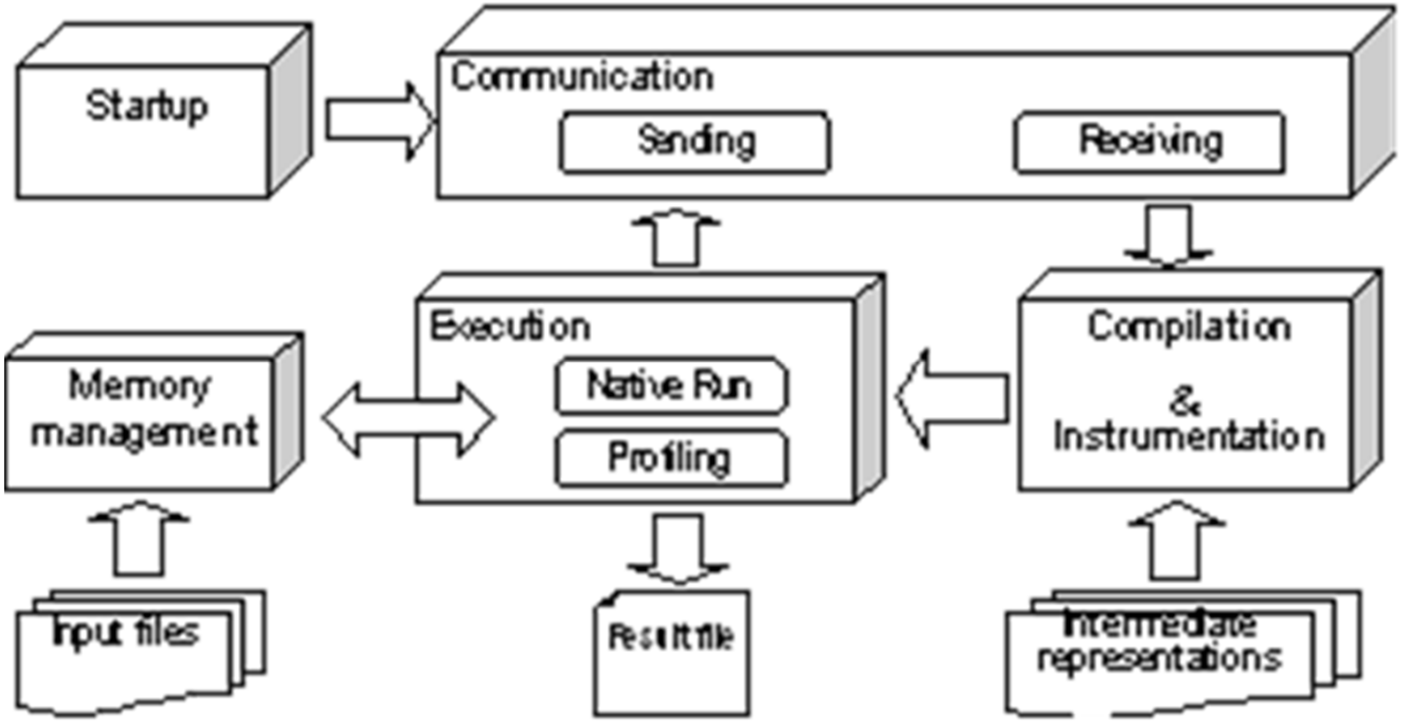
# Parallel profiling

- Basic idea
  - Instrumentation for a subset of potentially dependent memory references
    - Run instrumentation and profiling for different subsets independently
    - Merge the results to produce the final dependence graph
- Overhead due to extra instrumentation and progrma execution
    - Controlling granularity and scheduling carefully for load balance
- Partitioning by alias classes
- Partitioning by potential dependence edges
    - Grouping dependence edges that must be profiled together (to recognize loop-independent dependences)
    - Granularity controlled by available computing resource
    - Dynamic scheduling by work stealing

# A high-level overview of the multi-slicing approach



# Details of worker processes



# Implementation

- In the GCC compiler, version 4.6.
- Experiments with SPECint 2006 benchmark suite
  - 401bzip2, 429.mcf, 433.milc, 445.gobmk, 456.hmmer, 458.sjeng, 462.libquantum, 464.h264ref, and 482.sphinx3.
- "Test" size input

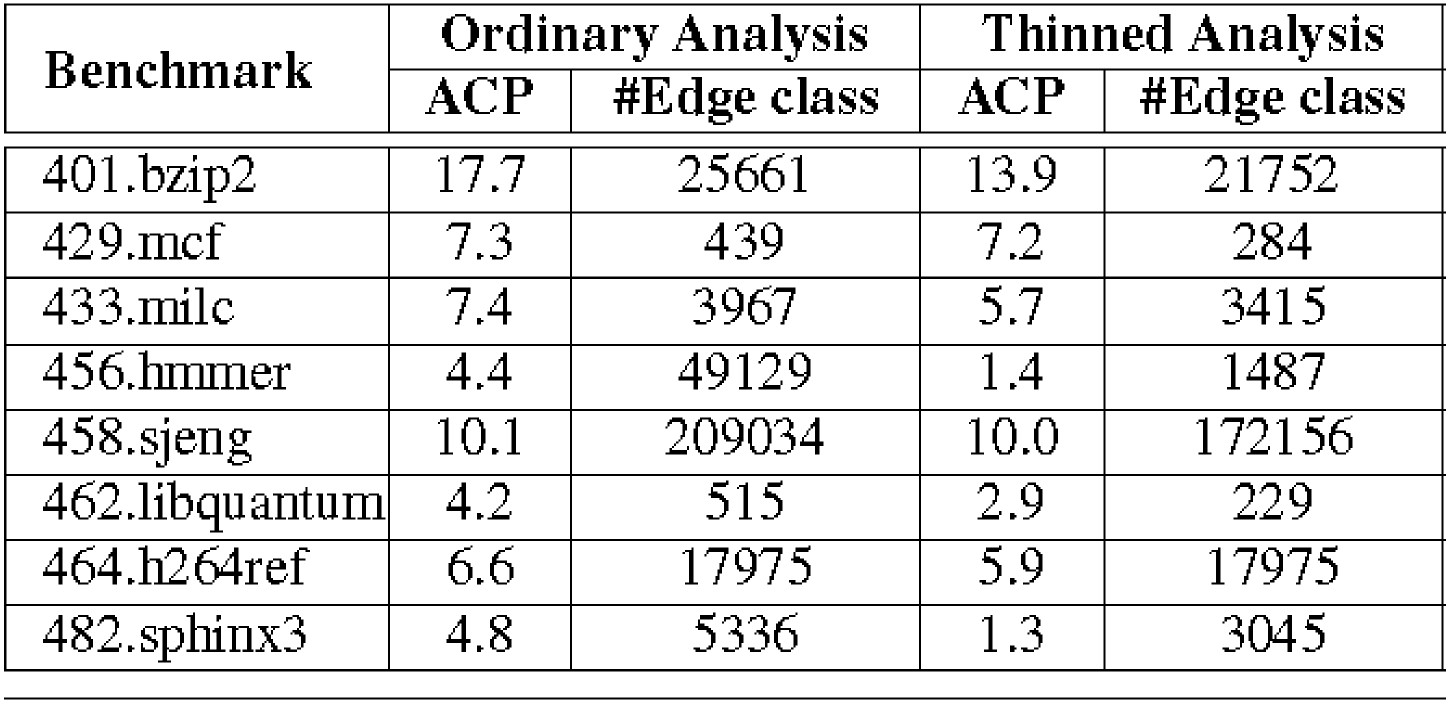
# Visualizing DD graph by DDGrapher

```
struct dpmatrix_s *
CreatePlan7Matrix(int N, int M, int padN, int padM)
{
  struct dpmatrix_s *mx;
  int i;

  mx          = (struct dpmatrix_s *) MallocOrDie (sizeof(struct dpmatrix_s));
  mx->xmx     = (int **) MallocOrDie (sizeof(int *) * (N+1));
  mx->mmx     = (int **) MallocOrDie (sizeof(int *) * (N+1));
```

```
in Function: main_loop_se
Node(w) in File: core_algor
```

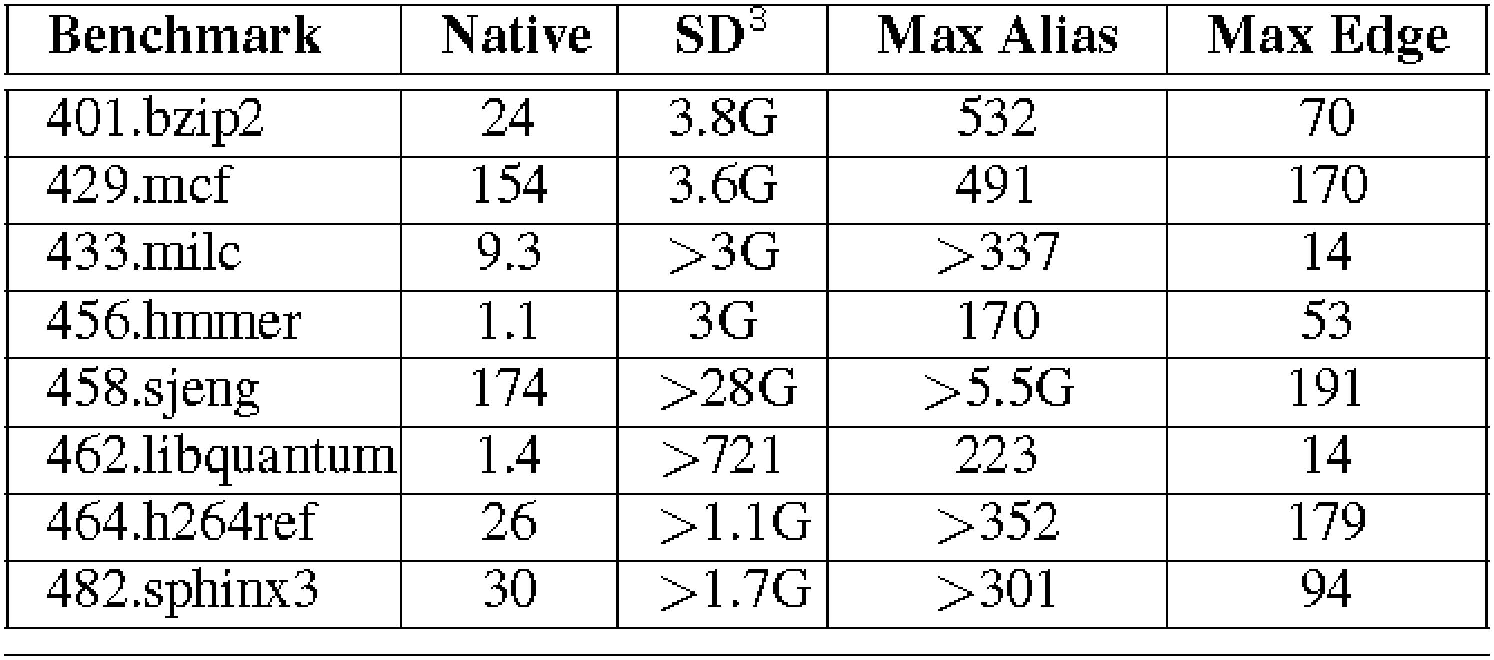
# Effect of thinned analysis



| Benchmark | Ordinary Analysis | | Thinned Analysis | |
|---|---|---|---|---|
| | ACP | #Edge class | ACP | #Edge class |
| 401.bzip2 | 17.7 | 25661 | 13.9 | 21752 |
| 429.mcf | 7.3 | 439 | 7.2 | 284 |
| 433.milc | 7.4 | 3967 | 5.7 | 3415 |
| 456.hmmer | 4.4 | 49129 | 1.4 | 1487 |
| 458.sjeng | 10.1 | 209034 | 10.0 | 172156 |
| 462.libquantum | 4.2 | 515 | 2.9 | 229 |
| 464.h264ref | 6.6 | 17975 | 5.9 | 17975 |
| 482.sphinx3 | 4.8 | 5336 | 1.3 | 3045 |

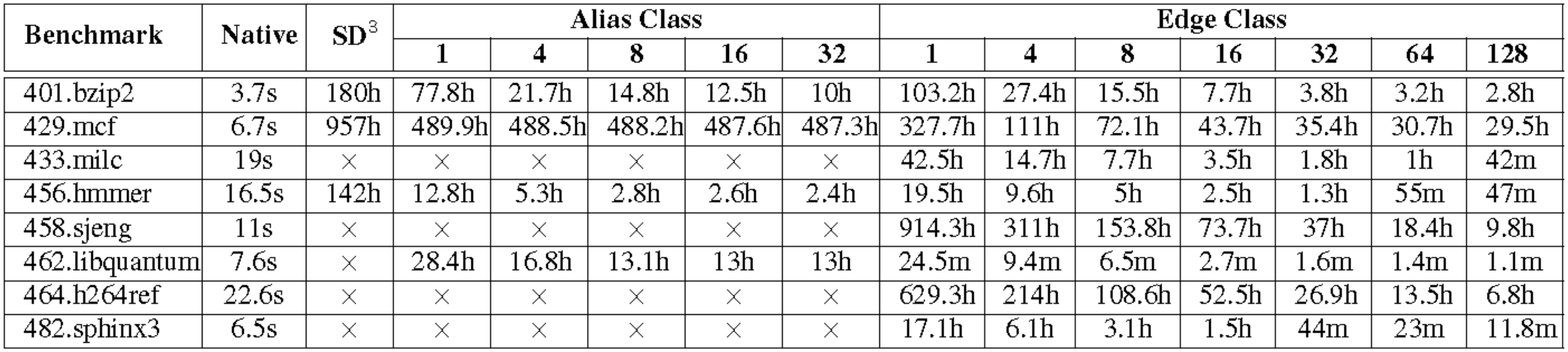
ACP (Alias Class Population): the average size of an alias class, counted by the number of load/stores.

# Memory usage during profiling (MB)



| Benchmark | Native | SD$^3$ | Max Alias | Max Edge |
|---|---|---|---|---|
| 401.bzip2 | 24 | 3.8G | 532 | 70 |
| 429.mcf | 154 | 3.6G | 491 | 170 |
| 433.milc | 9.3 | >3G | >337 | 14 |
| 456.hmmer | 1.1 | 3G | 170 | 53 |
| 458.sjeng | 174 | >28G | >5.5G | 191 |
| 462.libquantum | 1.4 | >721 | 223 | 14 |
| 464.h264ref | 26 | >1.1G | >352 | 179 |
| 482.sphinx3 | 30 | >1.7G | >301 | 94 |

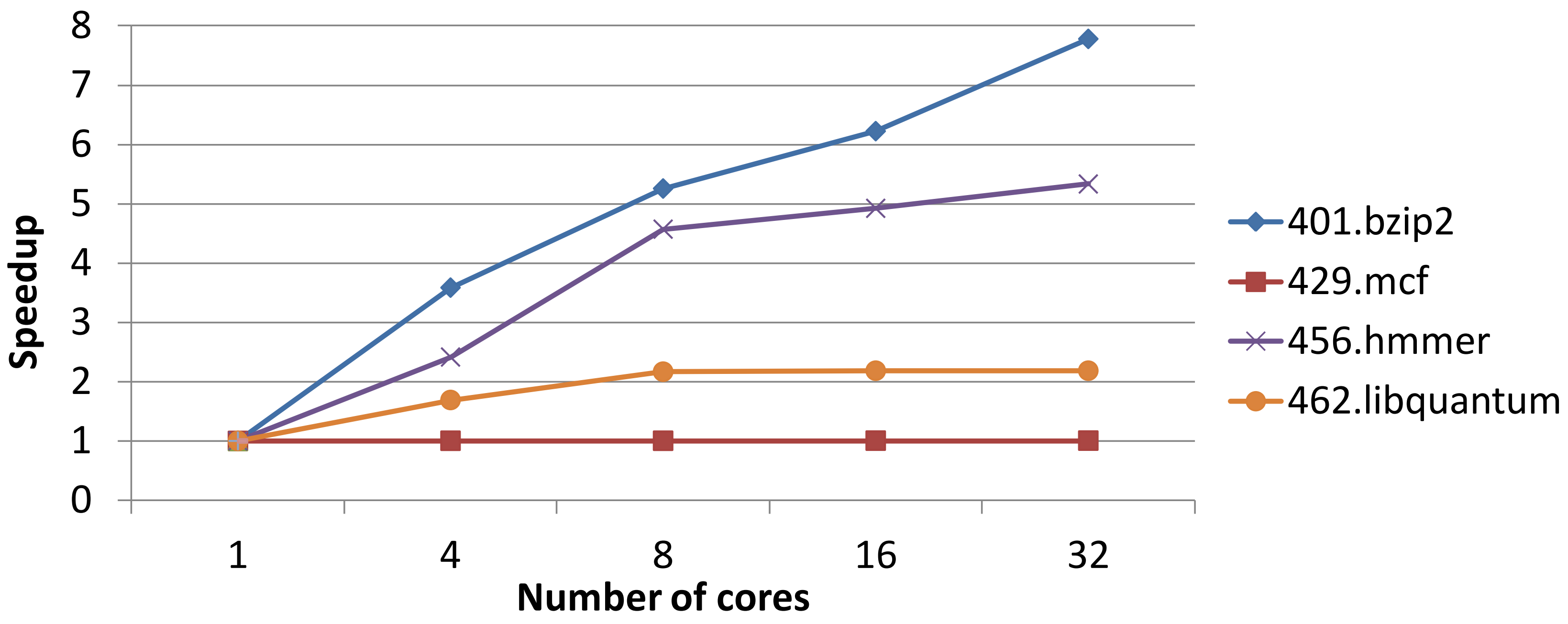
# Profiling time with the multi-slicing approach



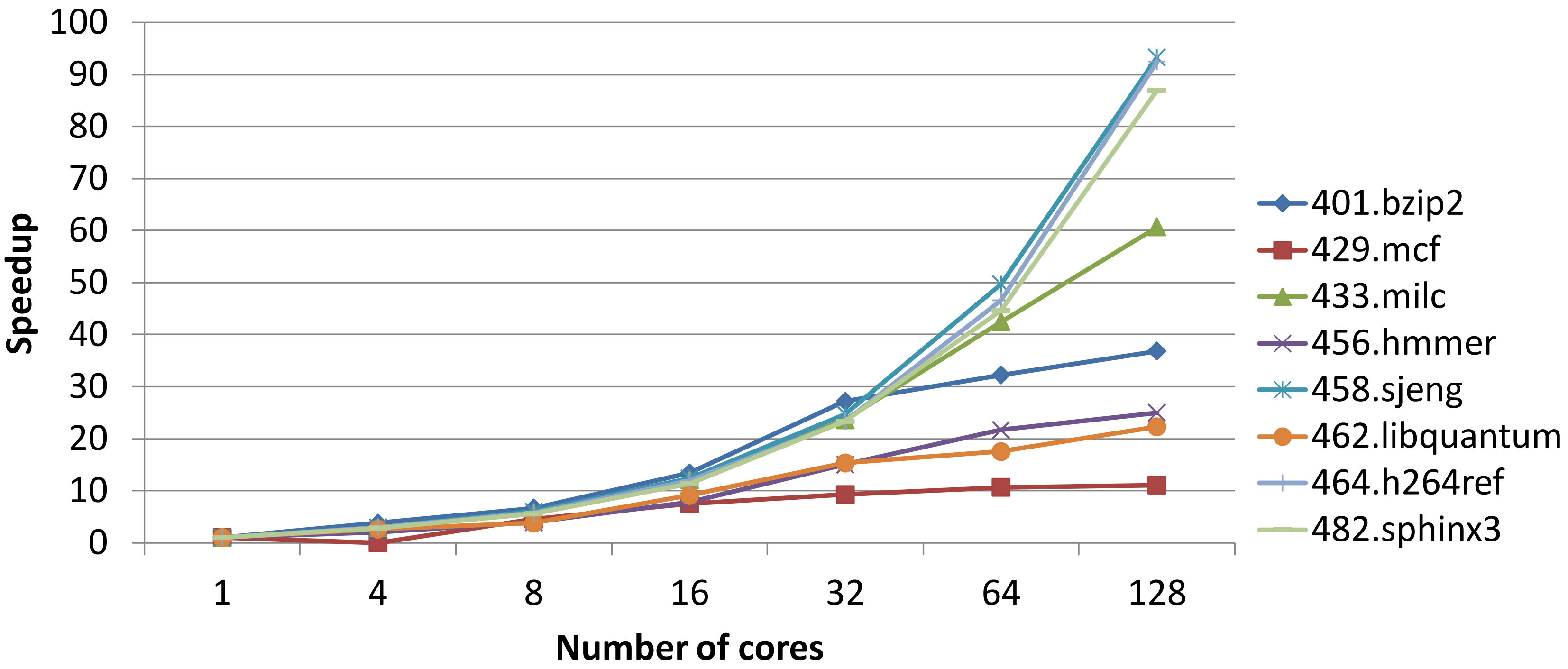
| Benchmark | Native | SD$^3$ | Alias Class |  |  |  |  | Edge Class |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  | **1** | **4** | **8** | **16** | **32** | **1** | **4** | **8** | **16** | **32** | **64** | **128** |
| 401.bzip2 | 3.7s | 180h | 77.8h | 21.7h | 14.8h | 12.5h | 10h | 103.2h | 27.4h | 15.5h | 7.7h | 3.8h | 3.2h | 2.8h |
| 429.mcf | 6.7s | 957h | 489.9h | 488.5h | 488.2h | 487.6h | 487.3h | 327.7h | 111h | 72.1h | 43.7h | 35.4h | 30.7h | 29.5h |
| 433.milc | 19s | × | × | × | × | × | × | 42.5h | 14.7h | 7.7h | 3.5h | 1.8h | 1h | 42m |
| 456.hmmer | 16.5s | 142h | 12.8h | 5.3h | 2.8h | 2.6h | 2.4h | 19.5h | 9.6h | 5h | 2.5h | 1.3h | 55m | 47m |
| 458.sjeng | 11s | × | × | × | × | × | × | 914.3h | 311h | 153.8h | 73.7h | 37h | 18.4h | 9.8h |
| 462.libquantum | 7.6s | × | 28.4h | 16.8h | 13.1h | 13h | 13h | 24.5m | 9.4m | 6.5m | 2.7m | 1.6m | 1.4m | 1.1m |
| 464.h264ref | 22.6s | × | × | × | × | × | × | 629.3h | 214h | 108.6h | 52.5h | 26.9h | 13.5h | 6.8h |
| 482.sphinx3 | 6.5s | × | × | × | × | × | × | 17.1h | 6.1h | 3.1h | 1.5h | 44m | 23m | 11.8m |

All versions compiled with “O0” option.
For profiling, “O0” allows precise drawing of dependence graph at source level

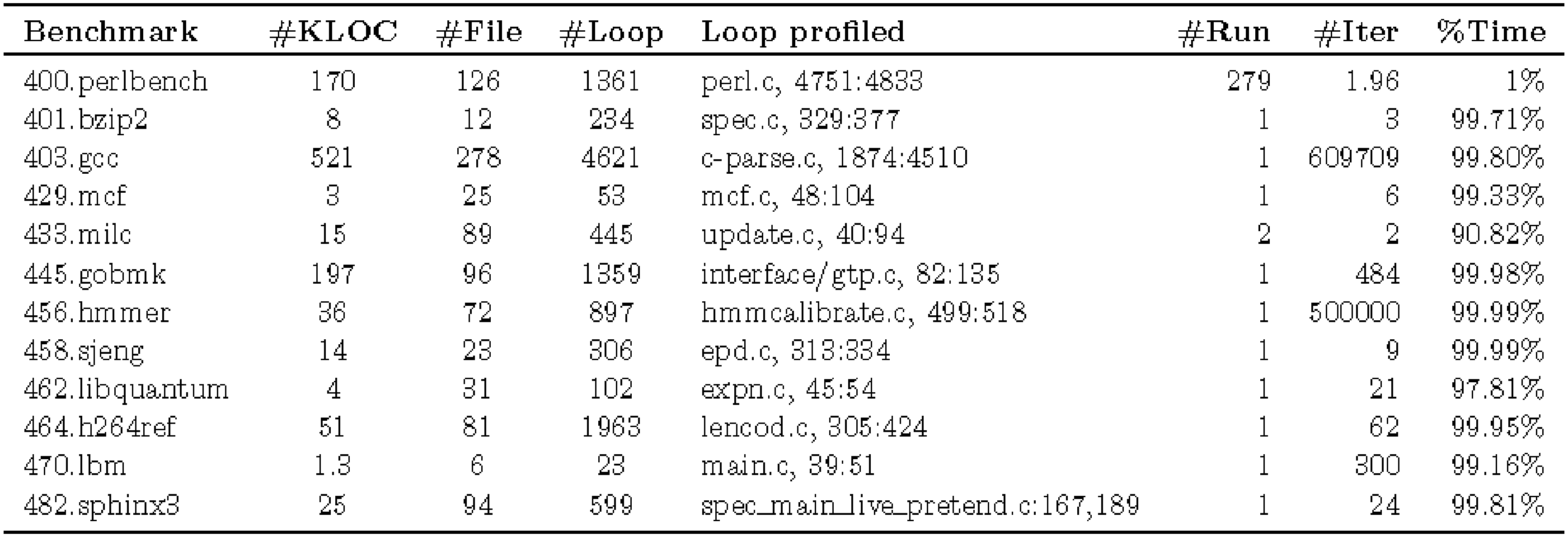
# Multi-slicing parallel efficiency (alias-class partitioning)



# Multi-slicing parallel efficiency (edge partitioning)



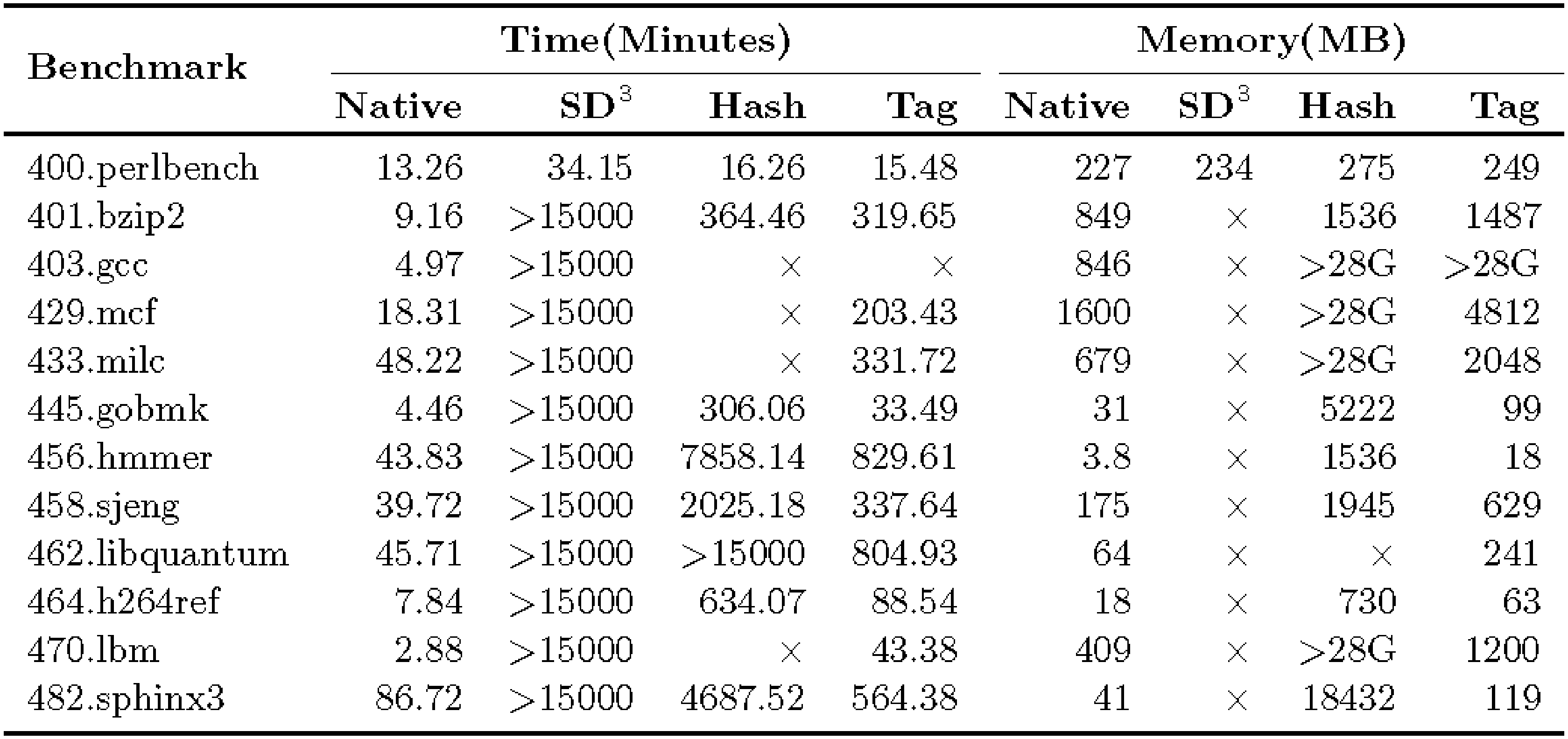
# Characteristics of the Benchmarks (“reference” input size)



| Benchmark | #KLOC | #File | #Loop | Loop profiled | #Run | #Iter | %Time |
|---|---|---|---|---|---|---|---|
| 400.perlbench | 170 | 126 | 1361 | perl.c, 4751:4833 | 279 | 1.96 | 1% |
| 401.bzip2 | 8 | 12 | 234 | spec.c, 329:377 | 1 | 3 | 99.71% |
| 403.gcc | 521 | 278 | 4621 | c-parse.c, 1874:4510 | 1 | 609709 | 99.80% |
| 429.mcf | 3 | 25 | 53 | mcf.c, 48:104 | 1 | 6 | 99.33% |
| 433.milc | 15 | 89 | 445 | update.c, 40:94 | 2 | 2 | 90.82% |
| 445.gobmk | 197 | 96 | 1359 | interface/gtp.c, 82:135 | 1 | 484 | 99.98% |
| 456.hmmer | 36 | 72 | 897 | hmmcalibrate.c, 499:518 | 1 | 500000 | 99.99% |
| 458.sjeng | 14 | 23 | 306 | epd.c, 313:334 | 1 | 9 | 99.99% |
| 462.libquantum | 4 | 31 | 102 | expn.c, 45:54 | 1 | 21 | 97.81% |
| 464.h264ref | 51 | 81 | 1963 | lencod.c, 305:424 | 1 | 62 | 99.95% |
| 470.lbm | 1.3 | 6 | 23 | main.c, 39:51 | 1 | 300 | 99.16% |
| 482.sphinx3 | 25 | 94 | 599 | spec_main_live_pretend.c:167,189 | 1 | 24 | 99.81% |

All versions compiled with “O0” option.
For profiling, “O0” allows precise drawing of dependence graph at source level

# Timing and memory usage result of different profiling techniques (“reference” input size)

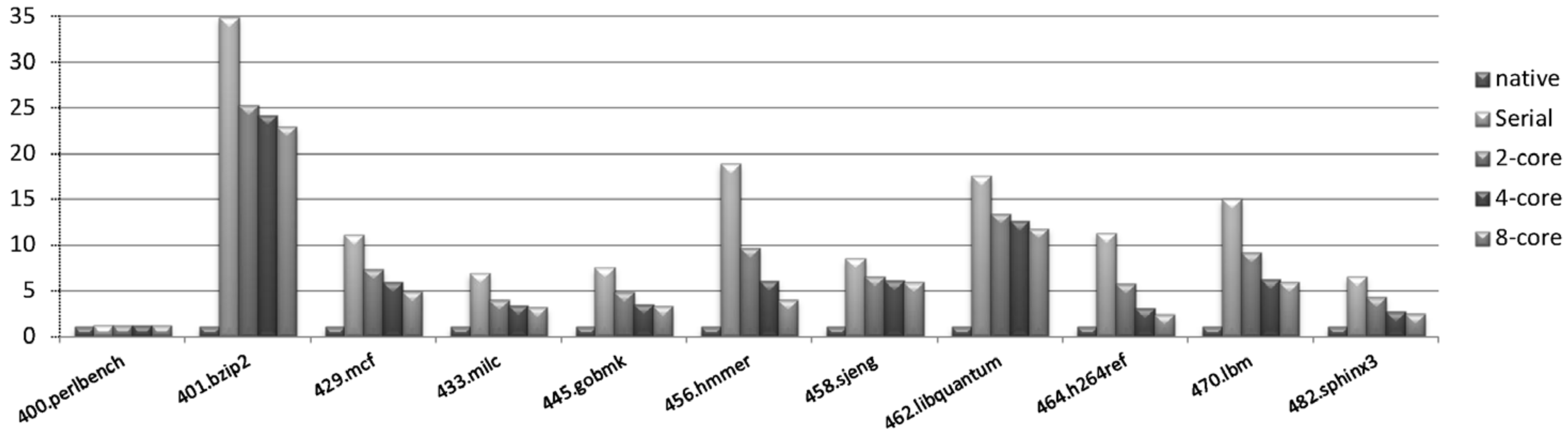


| Benchmark | Time(Minutes) | | | | Memory(MB) | | | |
|---|---|---|---|---|---|---|---|---|
| | Native | $SD^3$ | Hash | Tag | Native | $SD^3$ | Hash | Tag |
| 400.perlbench | 13.26 | 34.15 | 16.26 | 15.48 | 227 | 234 | 275 | 249 |
| 401.bzip2 | 9.16 | >15000 | 364.46 | 319.65 | 849 | × | 1536 | 1487 |
| 403.gcc | 4.97 | >15000 | × | × | 846 | × | >28G | >28G |
| 429.mcf | 18.31 | >15000 | × | 203.43 | 1600 | × | >28G | 4812 |
| 433.milc | 48.22 | >15000 | × | 331.72 | 679 | × | >28G | 2048 |
| 445.gobmk | 4.46 | >15000 | 306.06 | 33.49 | 31 | × | 5222 | 99 |
| 456.hmmer | 43.83 | >15000 | 7858.14 | 829.61 | 3.8 | × | 1536 | 18 |
| 458.sjeng | 39.72 | >15000 | 2025.18 | 337.64 | 175 | × | 1945 | 629 |
| 462.libquantum | 45.71 | >15000 | >15000 | 804.93 | 64 | × | × | 241 |
| 464.h264ref | 7.84 | >15000 | 634.07 | 88.54 | 18 | × | 730 | 63 |
| 470.lbm | 2.88 | >15000 | × | 43.38 | 409 | × | >28G | 1200 |
| 482.sphinx3 | 86.72 | >15000 | 4687.52 | 564.38 | 41 | × | 18432 | 119 |

“Thinned analysis” and “partial profile” are both implemented in the “Hash” and “Tag” versions

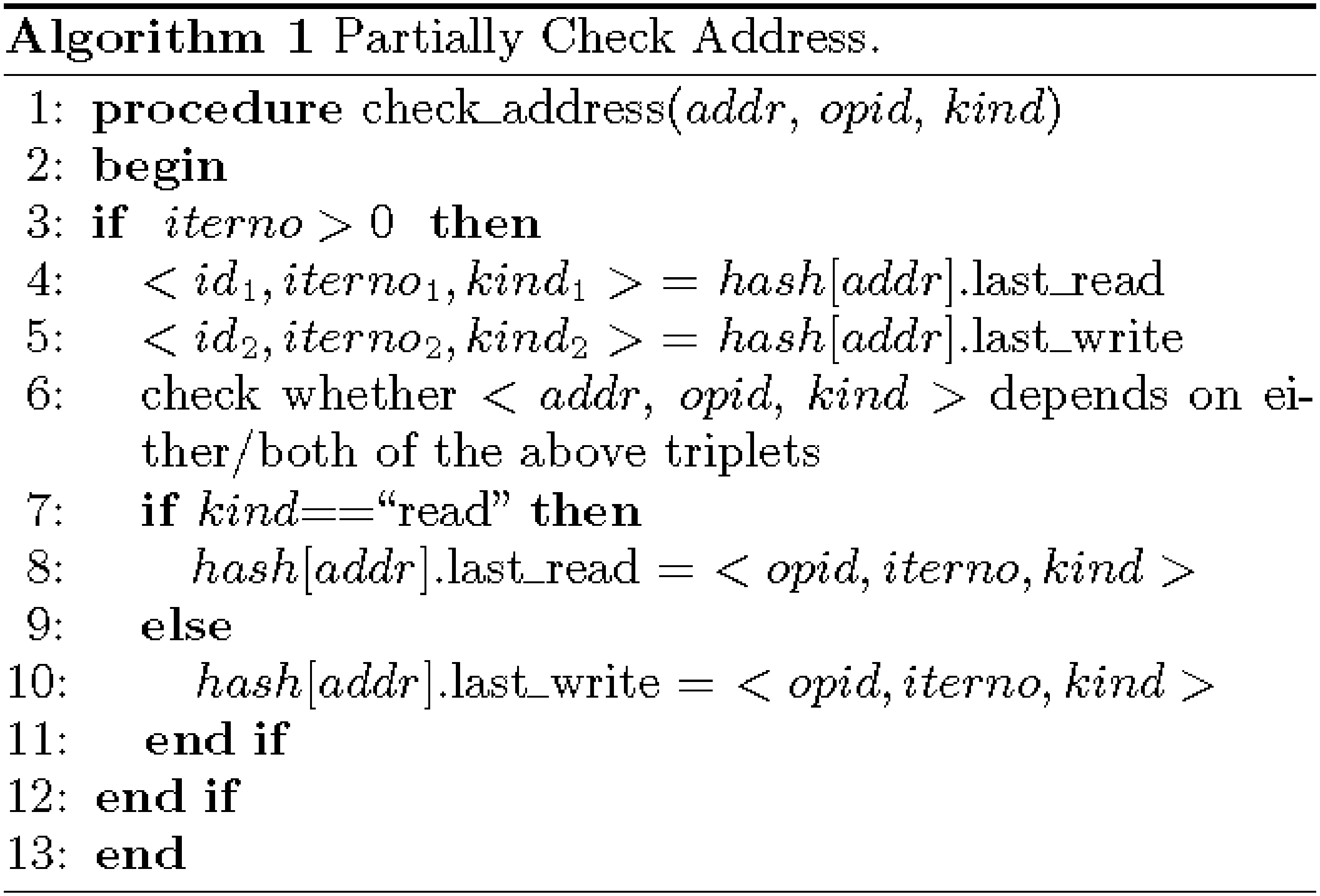
# Parallel profiling time (minutes) with the memory-tag approach

| Benchmark | Serial | 2-core | 4-core | 8-core |
|---|---:|---:|---:|---:|
| 400.perlbench | 15.48 | 15.38 | 15.22 | 15.26 |
| 401.bzip2 | 319.65 | 231.6 | 221.07 | 210.22 |
| 429.mcf | 203.43 | 133.40 | 108.26 | 89.9 |
| 433.milc | 331.72 | 191.31 | 163.61 | 151.68 |
| 445.gobmk | 33.49 | 21.55 | 15.64 | 14.46 |
| 456.hmmer | 829.61 | 424.67 | 264.88 | 175.41 |
| 458.sjeng | 337.64 | 260.18 | 240.89 | 233.61 |
| 462.libquantum | 804.93 | 612.90 | 576.39 | 550.26 |
| 464.h264ref | 88.54 | 45.07 | 24.08 | 18.64 |
| 470.lbm | 43.38 | 26.41 | 17.99 | 16.91 |
| 482.sphinx3 | 564.38 | 368.87 | 231.89 | 212.40 |


35
30
25
20
15
10
5
0
400.perlbench
401.bzip2
429.mcf
433.milc
445.gobmk
456.hmmer
458.sjeng
462.libquantum
464.h264ref
470.lbm
482.sphinx3
native
Serial
2-core
4-core
8-core

# Idea of partial but sufficient profiling

- A partial dependence graph
  - contains only the latest data dependence concerned with each statement
  - The access record for each monitored memory address keeps only the most recent load and store.
- Under such simplification, we are able to capture the most recent instance of loop-independent and loop-carried dependence.
- We also keep the shortest dependence distance



**Algorithm 1** Partially Check Address.

```
procedure check_address($addr$, $opid$, $kind$)
begin
if $iterno > 0$ then
    $< id_1, iterno_1, kind_1 > = hash[addr]$.last_read
    $< id_2, iterno_2, kind_2 > = hash[addr]$.last_write
    check whether $< addr, opid, kind >$ depends on either/both of the above triplets
    if $kind$==“read” then
        $hash[addr]$.last_read $= < opid, iterno, kind >$
    else
        $hash[addr]$.last_write $= < opid, iterno, kind >$
    end if
end if
end
```

# Idea of partial but sufficient profiling

- Theorem 1 (Fundamental Theorem of Dependence) :

  *"Any reordering transformation that preserves every dependence in a program preserves the meaning of that program."*

- Theorem 2 :

  *A transformation or a loop execution schedule violates a dependence in the partial dependence graph if and only if it violates a dependence in the complete dependence graph*

# Memory tagging

- Avoid hash table operations
  - Especially to avoid byte-level hashing
- Embed memory tags in the original data structure
  - Direct comparison between memory tags
  - Fixed offset from the base address
- Requires careful type analysis
  - Especially in the presence of type cast, union, aliasing
- Look for "promotable" objects
  - Set up promotion rules

# Recap

- Basic mechanisms
  - Hash tables for memory addresses
  - Regular strides (and limitation)
- Obvious compiler-based techniques to reduce cost
  - Some works but some don't
- "Multi-slicing" (or parallel profiling based on partitioning)
  - Partitioning by alias classes
  - Partitioning by potential dependence edges
  - Input-dependent specialization ("thinned analysis")
- To make profiling much more practical
  - Partial profiling that is sufficient for loop parallelization
- A more efficient mechanism (memory-tags)
  - Need compiler support
- Future work:
  - We are yet to combine partial profiling and memory tagging with edge partitioning
  - Use profiling result to produce parallel code automatically

- To be continued …