Understanding Memory Access Bottlenecks On Multicore

Mini Symposium “Scalability and Usability of HPC Programming Tools”
ParCo 2007, FZ Jülich, September 5, 2007

Josef Weidendorfer

Lehrstuhl für Rechnertechnik und Rechnerorganisation
Institut für Informatik, Technische Universität München

MUNICH-CORE INITIATIVE

Outline

- Introduction
  Relevance for Mini Symposium Topic
- Memory Access Bottlenecks and Metrics
  From Single- to Multicore
- Our Tool Suite
  Callgrind / KCachegrind
  Issues with Existing Visualizations & Experiences
- Conclusion and Future Work

Relevance for Mini Symposium Topic

- Usability
- Scalability

Cope with large amount of events (memory accesses !)

- In the measurement tool
- In the visualization tool

In measurement

- Use statistical methods / simulation
- Online processing to reduce amount of data
  - Filtering / Selection
  - Aggregation ($\rightarrow$ Profile)

Memory Access Bottlenecks

Main problem: “Memory wall”
Increasing performance gap main memory vs. processor

True for Multicore, too

Solution: Caches

True for Multicore, too

- Exploit locality of memory accesses (temporal / spatial)
- Lowers access latency by putting data copies into fast memory
  - Keep recently used copies (accounts for temporal locality)
  - Block oriented (accounts for spatial locality)
- „Bad memory access behavior“: Poor exploitation of caches

Optimization strategies

True for Multicore, too

- Improve temporal locality by reordering accesses
- Improve spatial locality by changing data layout
- Prefetch data needed in the future

Memory Access Bottlenecks

Multicore adds

- All the issues of parallel code
  - Load balancing, synchronization overhead, difficult to program …
- Cores share available resources for
  - Connection to main memory
  - Caches
- Software has to cope with new configurations
  - Caches shared vs. separate per core, increased memory hierarchy

Optimization strategies

- Best process placing for good exploitation of available caches?

Analysis of Memory Access Behavior

Good Tools™ give following answers

- Yes, we have a problem because of memory accesses
- And it happens there
  (code position + call path + data structure + thread/process)
- Yes, it makes sense to optimize (potential benefit)
- Just try to do this and that to avoid cache pollution / bad layout / …
  (good hints with expertise)

What are good metrics to see the problem and path to solution?

Metrics for Bad Memory Access Behavior

- Cache miss counts
  - Good: Pinpoints where time is lost (But: how much? Data structure?)
  - Difficult to derive optimization (e.g. what/how to block?)

- Temporal / spatial cache line usage:
  How much / often used before eviction
  - Easy to see bad memory layout (e.g. hash lookups)

- Model of idealized (unlimited) cache with LRU list of data accesses

- Simulation of bandwidth requirement

LRU List of Data Accesses

- Relates to behavior of fully associate cache
- Example: Address sequence 1 2 3 4 2 3 2 2 5

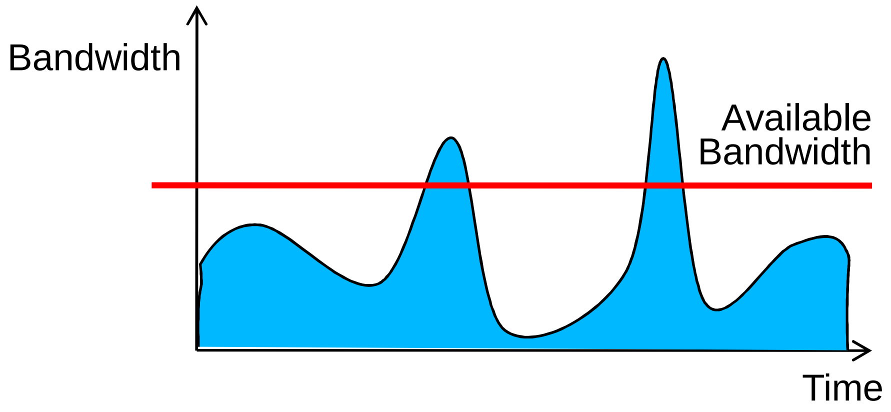
LRU offset	0	1	2	3	4
Address	5	2	3	4	1
Last access (time steps)	0	1	3	5	8

Distance	1	2	3
Accesses	1	1	2

- Provides size of working set (used in given time span)
- Stack reuse distance: histogram over move distances on access (See papers of K. Beyls)
  - Percentage of accesses covered by given cache size
  - Which data structures have bad locality with high influence

Simulation of Bandwidth Requirement

- What is the required amount of data with ideal memory? (Suppose only computation-boundness)
  - Has to be simulated
  - Simple CPU model (given latency for each opcode)



  - Which structures are better served from cache?
  - Gives hints for prefetching (can prefetching pollute cache?)

Adaption for Multicore

- Bandwidth requirement diagram (easy)
  - Additional curves for shared vs. separate caches
- Multiple LRU lists
  - for all cores
    - reuse distances with shared cache
    - provides information for workset overlapping
  - for each core pair: shared with other“, „invalidated by other“
    - Shows amount communication between cores (size + number)
- Should provide hints for
  - placement of processes (exploit same workset via shared cache)
  - prefetch helper thread for multiple cores

Our Tool Suite: Callgrind / KCachegrind

Measurement

- Based on Valgrind (runtime instrumentation, known for "memcheck")
- Instrument memory accesses feed cache simulator
- Profiling tool relating cache events to call-graph (path relation possible)

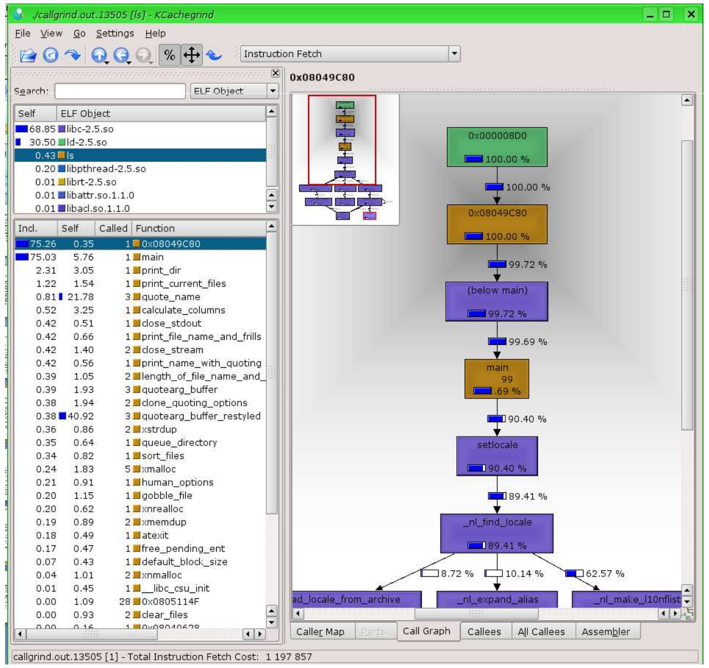
Pro and Contra

- Memory accesses only from user-level code, Slowdown (40-60x)
- Synchronous 2-level inclusive cache (optional hardware prefetcher)

✓ Does not need root access / can not crash machine
✓ Allows for sophisticated metrics (line usage, stack reuse distance)
✓ Easy to understand / reconstruct for user
✓ Reproducible results independent on real machine load
✓ Derived optimizations applicable for many architectures

Our Tool Suite: Callgrind / KCachegrind

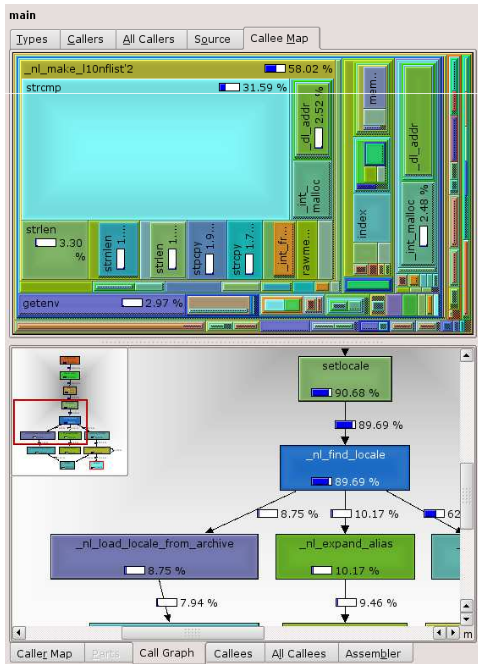
- Visualization
  - Call relationship of functions (callers, callees, call graph, tree map)
  - Source/Assembly annotation: event counts + control flow

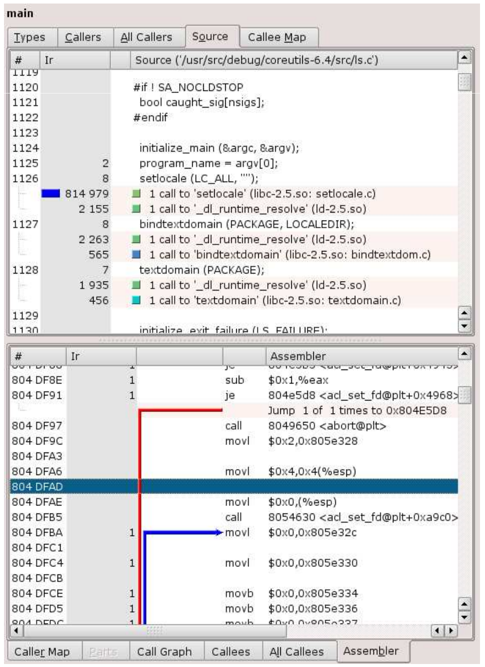


Self	ELF Object
68.85	libc-2.5.so
30.50	ld-2.5.so
0.43	ls
0.20	libpthread-2.5.so
0.01	librt-2.5.so
0.01	libattr.so.1.1.0
0.01	libacl.so.1.1.0

Incl.	Self	Called	Function
75.26	0.35	1	0x08049C80
75.03	5.76	1	main
2.31	3.05	1	print_dir
1.22	1.54	1	print_current_files
0.81	21.78	3	quote_name
0.52	3.25	1	calculate_columns
0.42	0.51	1	close_stdout
0.42	0.66	1	print_file_name_and_frills
0.42	1.40	2	close_stream
0.42	0.56	1	print_name_with_quoting
0.39	1.05	2	length_of_file_name_and_
0.39	1.93	3	quotearg_buffer
0.38	1.94	2	clone_quoting_options
0.38	40.92	3	quotearg_buffer_restyled
0.36	0.86	2	xstrdup
0.35	0.64	1	queue_directory
0.34	0.82	1	sort_files
0.24	1.83	5	xmalloc
0.21	0.91	1	human_options
0.20	1.15	1	gobble_file
0.20	0.62	1	xnrealloc
0.19	0.89	2	xmemdup
0.18	0.49	1	atexit
0.17	0.47	1	free_pending_ent
0.07	0.43	1	default_block_size
0.04	1.01	2	xnmalloc
0.01	0.45	1	__libc_csu_init
0.00	1.09	28	0x0805114F
0.00	0.93	2	clear_files

callgrind.out.13505 [1] - Total Instruction Fetch Cost: 1 197 857





```
#     Ir        Source ('/usr/src/debug/coreutils-6.4/src/ls.c')
1120            #if ! SA_NOCLDSTOP
1121              bool caught_sig[nsigs];
1122            #endif
1123
1124              initialize_main (&argc, &argv);
1125         2    program_name = argv[0];
1126         8    setlocale (LC_ALL, "");
       814 979    1 call to 'setlocale' (libc-2.5.so: setlocale.c)
         2 155    1 call to '_dl_runtime_resolve' (ld-2.5.so)
1127         8    bindtextdomain (PACKAGE, LOCALEDIR);
         2 263    1 call to '_dl_runtime_resolve' (ld-2.5.so)
           565    1 call to 'bindtextdomain' (libc-2.5.so: bindtextdom.c)
1128         7    textdomain (PACKAGE);
         1 935    1 call to '_dl_runtime_resolve' (ld-2.5.so)
           456    1 call to 'textdomain' (libc-2.5.so: textdomain.c)
1129
```

```
#          Ir        Assembler
804 DF8E    1   sub    $0x1,%eax
804 DF91    1   je     804e5d8 <acl_set_fd@plt+0x4968>
                       Jump  1 of  1 times to 0x804E5D8
804 DF97        call   8049650 <abort@plt>
804 DF9C        movl   $0x2,0x805e328
804 DFA3
804 DFA6        movl   $0x4,0x4(%esp)
804 DFAD
804 DFAE        movl   $0x0,(%esp)
804 DFB5        call   8054630 <acl_set_fd@plt+0xa9c0>
804 DFBA    1   movl   $0x0,0x805e32c
804 DFC1
804 DFC4    1   movl   $0x0,0x805e330
804 DFCB
804 DFCE    1   movb   $0x0,0x805e334
804 DFD5    1   movb   $0x0,0x805e336
```

Experiences with existing Visualizations

Provide not only graphical results, but

- export to ASCII lists (HTML / XML), or better
- equivalent command line tool for scriptability: merging, querying, …

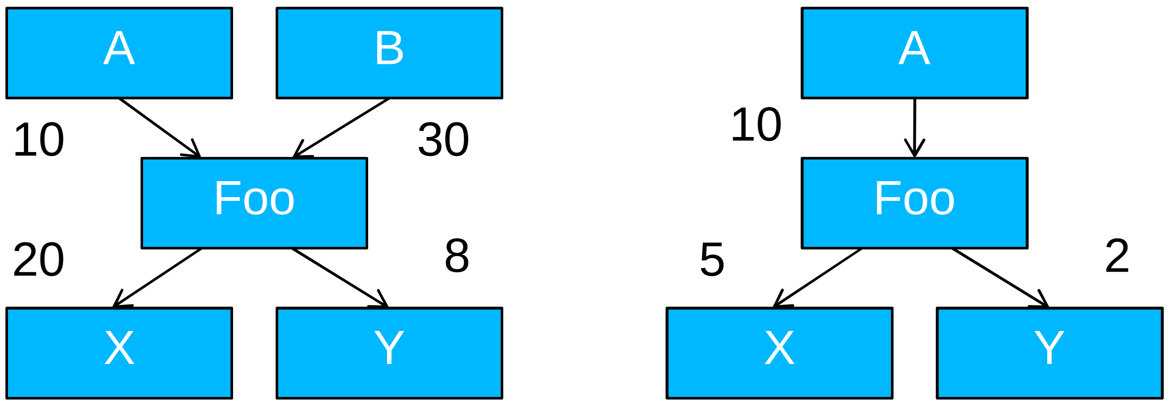
Graphical interface

- Very clean user interface, small number of views / areas
- Only a few different view types into data, familiar to users
  - ✓ lists, call graphs, bar charts, heat maps, diagrams
  - – tree maps, fully crowded 3-D views
- Highly interactive views
  - responsive
  - zoomable (one visualization with multiple detail levels)
  - intuitive narrowing / filtering of data
  - non-disruptive browsing (incremental or animation)
- Careful, consistent use of color coding

Improvements for existing Visualization: Call Graph

Issue: Tools seems to claim to always have full path profiling data

Example: Foo selected A selected

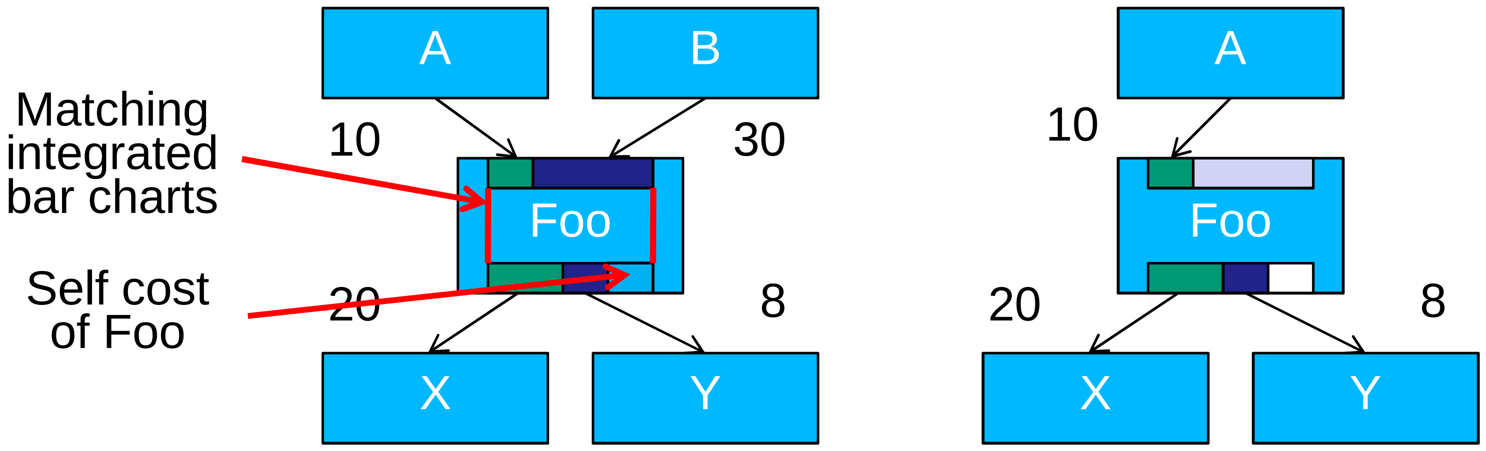


Reality could have been: Foo → X only when called from B !

Improvements for existing Visualization: Call Graph

Issue: Tools seems to claim to always have full path profiling data

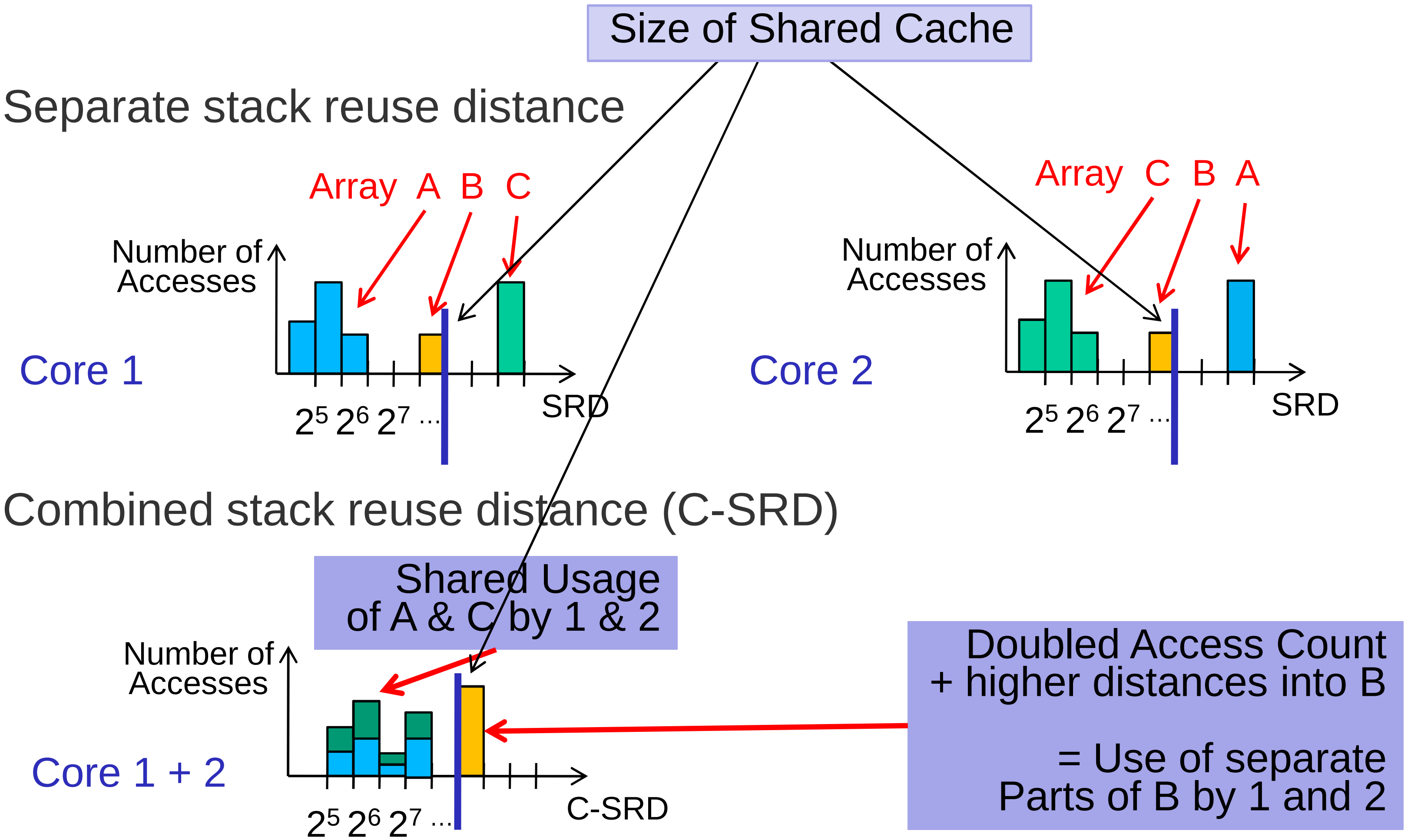
Example: Foo selected A selected (improved)



- Tells reality
- With real path profiles: Allows for interactive filtering of data
- Also for filtering in other dimensions (partitioning against threads)

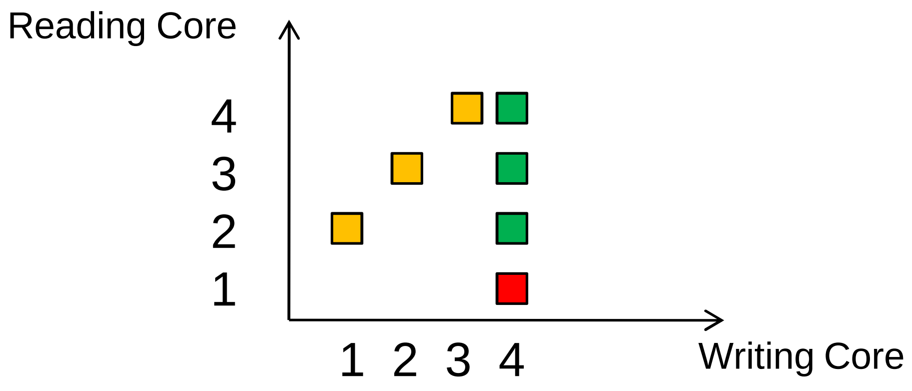
Visualization for Multicore

- Separate stack reuse distance
- Combined stack reuse distance (C-SRD)



Visualization for Multicore (2)

- Similar to get communication behavior among cores
  - Only count invalidations (with separate caches)
    - Heat map (color: number of invalidations)



- Histograms similar to last slide…

Conclusion

- Memory Access analysis on Single/Multicore benefits from better metrics than cache misses
- Simulation with runtime instrumentation proved useful for this
- We propose metrics and visualization for workset overlapping/communication on Multicore

Future work

Work in progress

- Measurement
  - Should stay usable (resource consumption of simulation)
  - Enhancement for multicore simulation in industry cooperation
  - Keep amount of data small
    - E.g. online aggregation also for bandwidth requirement
    - Statistical reconstruction of approximative time scale visualization ?
- Visualization
  - Prototype some visualizations for multicore memory access
  - Conduct usability studies