Featherlight Reuse-distance Measurement

APPENDIX

A. TIME REUSE TO STACK REUSE

RDX allocates several bins to include every sampled reuse pair with time distance falling into different ranges. Common to prior work [1, 2], the histogram uses the *logarithmic* scale for the bin size as the horizontal axis and the percentage of total sampled reuse pairs as the vertical axis.

With the time distance histogram, RDX leverages the existing technique [1] to approximate the reuse distance histogram. This section only shows the intuitive idea of this approximation. Considering a memory access sequence `aXXXXa`, the time distance of `a` is 5 while we have no idea how many distinct elements between a. If three time reuses are of distance 1 and one reuses is of distance 5, we definitely know there should be only two distinct elements between `a` by exhaustive searching.

However, this greedy method usually does not work for calculating the reuse distance of a common application, especially directly calculating the reuse distance of the specific reuse instance of an element. Thus we seek to a statistical model to calculate the probability of a specific reuse distance instead of obtaining the reuse distance of a specific reuse instance. Assuming that the probability of a data element is independent from others, whether a data element is accessed in a given time interval Δ is actually a Bernoulli process. Then the probability of having k distinct data elements in this Δ interval is

$$P(k,\Delta) = \binom{N}{k} P_{interval}(\Delta)^k (1 - P_{interval}(\Delta))^{(N-k)} \tag{4}$$

where $P_{interval}(\Delta)$ is the probability for a data element to appear in the interval Δ and N is total number of distinct data elements. To calculate $P_R(k)$ (the probability of having reuse distance k for the entire program), we need to consider all the possibilities of having reuse distance of k from the interval length ranging between 1 and T, where T is the total number of memory accesses. The probability of having time interval Δ can be obtained from the time reuse histogram, denoted as $P_T(\Delta)$. Thus we have

$$P_R(k) = \sum_{\Delta=1}^{T} P(k,\Delta) P_T(\Delta) \tag{5}$$

Since $P_{interval}(\Delta)$ can be derived from $P_T(\Delta)$ and the details can be found in [3], Equation (5) transforms time reuse histogram to stack reuse histogram.

Their later work [1] further developed the algorithm by extending each bar width of both time and stack reuse histograms from 1 to any arbitrary number, which is utilized in our work. Their evaluation shows that this model gives more than 99% accuracy as to cache block granularity.

B. INPUT OF SPEC CPU BENCHMARKS

We use *ref* input to run all SPEC CPU benchmarks but some have multiple inputs, which are distinguished with numerical suffixes such as `gcc-1`, `gcc-2`, etc. The actual mapping of these names are shown in Figure 13 and Figure 14.

C. STACK REUSE HISTOGRAMS OF SPEC CPU 2006 BENCHMARKS WITH OFF-BY-ONE PROBLEM

In SPEC CPU2006, `bwaves` (Figure 15), `leslie3d` (Figure 16), `sphinx3` (Figure 17), `GemsFDTD` (Figure 18), `lbm` (Figure 19), and `cactusADM` (Figure 20) have the off-by-one problem, which can be inferred from their stack reuse histograms.

D. STACK REUSE HISTOGRAMS OF SPEC CPU2017

We have plotted the stack reuse histograms of `perlbench_r` (Figure 21, Figure 22 and Figure 23), `gcc_r` (Figure 24, Figure 25, Figure 26, Figure 27 and Figure 28), `bwaves_r` (Figure 29, Figure 30, Figure 31 and Figure 32), `mcf_r` (Figure 33), `cactuBSSN_r` (Figure 34), `namd_r` (Figure 35), `povray_r` (Figure 36), `lbm_r` (Figure 37), `omnetpp_r` (Figure 38), `wrf_r` (Figure 39), `xalancbmk_r` (Figure 40), `x264_r` (Figure 41, Figure 42 and Figure 43), `blender_r` (Figure 44), `cam4_r` (Figure 45), `deepsjeng_r` (Figure 46), `imagick_r` (Figure 47), `leela_r` (Figure 48), `nab_r` (Figure 49), `exchange2_r` (Figure 50), `fotonik3d_r` (Figure 51), `roms_r` (Figure 52), `xz_r` (Figure 53, Figure 54 and Figure 55), `perlbench_s` (Figure 56, Figure 57 and Figure 58), `gcc_s` (Figure 59, Figure 60 and Figure 61), `bwaves_s` (Figure 62 and Figure 63), `mcf_s` (Figure 64), `cactuB-`

Benchmark	Input argument
perlbench-1	`-I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1`
perlbench-2	`-I./lib diffmail.pl 4 800 10 17 19 300`
perlbench-3	`-I./lib splitmail.pl 1600 12 26 16 4500`
bzip2-1	`input.source 280`
bzip2-2	`chicken.jpg 30`
bzip2-3	`liberty.jpg 30`
bzip2-4	`input.program 280`
bzip2-5	`text.html 280`
bzip2-6	`input.combined 200`
gcc-1	`166.i -o 166.s`
gcc-2	`200.i -o 200.s`
gcc-3	`c-typeck.i -o c-typeck.s`
gcc-4	`cp-decl.i -o cp-decl.s`
gcc-5	`expr.i -o expr.s`
gcc-6	`expr2.i -o expr2.s`
gcc-7	`g23.i -o g23.s`
gcc-8	`s04.i -o s04.s`
gcc-9	`scilab.i -o scilab.s`
gamess-1	`< cytosine.2.config`
gamess-2	`< h2ocu2+.gradient.config`
gamess-3	`< triazolium.config`
gobmk-1	`--quiet --mode gtp" < 13x13.tst`
gobmk-2	`--quiet --mode gtp" < nngs.tst`
gobmk-3	`--quiet --mode gtp" < score2.tst`
gobmk-4	`--quiet --mode gtp" < trevorc.tst`
gobmk-5	`--quiet --mode gtp" < trevord.tst`
soplex-1	`-s1 -e -m45000 pds-50.mps`
soplex-2	`-m3500 ref.mps`
hmmer-1	`nph3.hmm swiss41`
hmmer-2	`--fixed 0 --mean 500 --num 500000 --sd 350 --seed 0 retro.hmm`
h264ref-1	`-d foreman_ref_encoder_baseline.cfg`
h264ref-2	`-d foreman_ref_encoder_main.cfg`
h264ref-3	`-d sss_encoder_main.cfg`
astar-1	`BigLakes2048.cfg`
astar-2	`rivers.cfg`

Figure 13: The input arguments of benchmarks which have several *ref* inputs for SPEC CPU 2006.

Benchmark	Input argument
perlbench_r-1	`-I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1`
perlbench_r-2	`-I./lib diffmail.pl 4 800 10 17 19 300`
perlbench_r-3	`-I./lib splitmail.pl 6400 12 26 16 100 0`
gcc_r-1	`gcc-pp.c -O3 -finline-limit=0 -fif-conversion -fif-conversion2 -o gcc-pp.opts-O3_-finline-limit_0_-fif-conversion_-fif-conversion2.s`
gcc_r-2	`gcc-pp.c -O2 -finline-limit=36000 -fpic -o gcc-pp.opts-O2_-finline-limit_36000_-fpic.s`
gcc_r-3	`gcc-smaller.c -O3 -fipa-pta -o gcc-smaller.opts-O3_-fipa-pta.s`
gcc_r-4	`ref32.c -O5 -o ref32.opts-O5.s`
gcc_r-5	`ref32.c -O3 -fselective-scheduling -fselective-scheduling2 -o ref32.opts-O3_-fselective-scheduling_-fselective-scheduling2.s`
bwaves_r-1	`bwaves_1 < bwaves_1.in`
bwaves_r-2	`bwaves_2 < bwaves_2.in`
bwaves_r-3	`bwaves_3 < bwaves_3.in`
bwaves_r-4	`bwaves_4 < bwaves_4.in`
x264_r-1	`--pass 1 --stats x264_stats.log --bitrate 1000 --frames 1000 -o BuckBunny_New.264 BuckBunny.yuv 1280x720`
x264_r-2	`--pass 2 --stats x264_stats.log --bitrate 1000 --dumpyuv 200 --frames 1000 -o BuckBunny_New.264 BuckBunny.yuv 1280x720`
x264_r-3	`--seek 500 --dumpyuv 200 --frames 1250 -o BuckBunny_New.264 BuckBunny.yuv 1280x720`
xz_r-1	`cld.tar.xz 160 19cf30ae51eddcbefda78dd06014b4b96281456e078ca7c13e1c0c9e6aaea8dff3efb4ad6b0456697718cede6bd5454852652806a657bb56e07d61128434b474 59796407 61004416 6`
xz_r-2	`cpu2006docs.tar.xz 250 055ce243071129412e9dd0b3b69a21654033a9b723d874b2015c774fac1553d9713be561ca86f74e4f16f22e664fc17a79f30caa5ad2c04fbc447549c2810fae 23047774 23513385 6e`
xz_r-3	`input.combined.xz 250 a841f68f38572a49d86226b7ff5baeb31bd19dc637a922a972b2e6d1257a890f6a544ecab967c313e370478c74f760eb229d4eef8a8d2836d233d3e9dd1430bf 40401484 41217675 7`
perlbench_s-1	`-I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1`
perlbench_s-2	`-I./lib diffmail.pl 4 800 10 17 19 300`
perlbench_s-3	`-I./lib splitmail.pl 6400 12 26 16 100 0`
gcc_s-1	`gcc-pp.c -O5 -fipa-pta -o gcc-pp.opts-O5_-fipa-pta.s`
gcc_s-2	`gcc-pp.c -O5 -finline-limit=1000 -fselective-scheduling -fselective-scheduling2 -o gcc-pp.opts-O5_-finline-limit_1000_-fselective-scheduling_-fselective-scheduling2.s`
gcc_s-3	`gcc-pp.c -O5 -finline-limit=24000 -fgcse -fgcse-las -fgcse-lm -fgcse-sm -o gcc-pp.opts-O5_-finline-limit_24000_-fgcse_-fgcse-las_-fgcse-lm_-fgcse-sm.s`
bwaves_s-1	`bwaves_1 < bwaves_1.in`
bwaves_s-2	`bwaves_2 < bwaves_2.in`
x264_s-1	`--pass 1 --stats x264_stats.log --bitrate 1000 --frames 1000 -o BuckBunny_New.264 BuckBunny.yuv 1280x720`
x264_s-2	`--pass 2 --stats x264_stats.log --bitrate 1000 --dumpyuv 200 --frames 1000 -o BuckBunny_New.264 BuckBunny.yuv 1280x720`
x264_s-3	`--seek 500 --dumpyuv 200 --frames 1250 -o BuckBunny_New.264 BuckBunny.yuv 1280x720`
xz_s-1	`cpu2006docs.tar.xz 6643 055ce243071129412e9dd0b3b69a21654033a9b723d874b2015c774fac1553d9713be561ca86f74e4f16f22e664fc17a79f30caa5ad2c04fbc447549c2810fae 1036078272 1111795472 4`
xz_s-2	`cld.tar.xz 1400 19cf30ae51eddcbefda78dd06014b4b96281456e078ca7c13e1c0c9e6aaea8dff3efb4ad6b0456697718cede6bd5454852652806a657bb56e07d61128434b474 536995164 539938872 8`

Figure 14: The input arguments of benchmarks which have several *ref* inputs for SPEC CPU 2017.

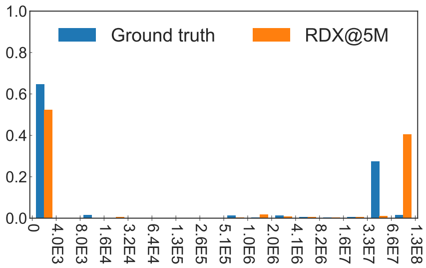

Figure 15: Stack reuse histogram of `bwaves`.

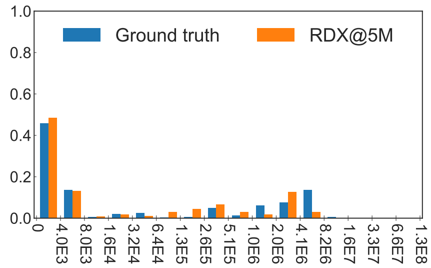

Figure 16: Stack reuse histogram of `leslie3d`.

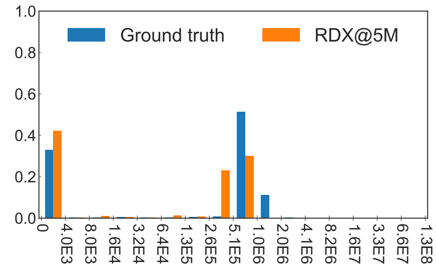

Figure 17: Stack reuse histogram of `sphinx3`.

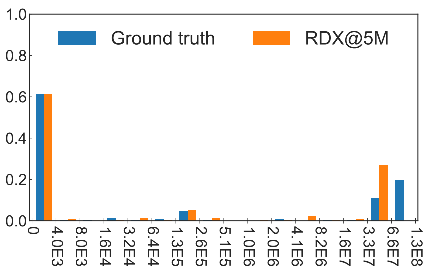

Figure 18: Stack reuse histogram of `GemsFDTDs`.

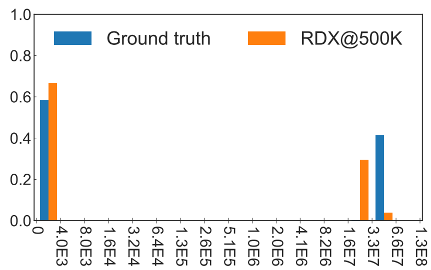

Figure 19: Stack reuse histogram of `lbm`.

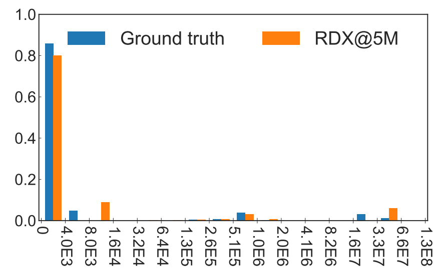

Figure 20: Stack reuse histogram of `cactusADM`.

`SSN_s` (Figure 65), `lbm_s` (Figure 66), `omnetpp_s` (Figure 67), `wrf_s` (Figure 68), `xalancbmk_s` (Figure 69), `x264_s` (Figure 70, Figure 71 and Figure 72), `cam4_s` (Figure 73), `pop2_s` (Figure 74), `deepsjeng_s` (Figure 75), `imagick_s` (Figure 76), `leela_s` (Figure 77), `nab_s` (Figure 78), `exchange2_s` (Figure 79), `fotonik3d_s` (Figure 80), `roms_s` (Figure 81), `xz_s` (Figure 82 and Figure 83) from SPEC CPU2017.

E. CASE STUDY: OPTIMIZING LULESH

LULESH [4], a UHPC application benchmark developed by Lawrence Livermore National Laboratory (LLNL), solves a simple Sedov blast problem with analytic answers. We launch it with 32 threads monitored by RDX and Figure 84 shows our graphic user interface highlighting the locations with high latency. The interface consists of three panels. The top one displays the source code; the bottom left one shows the application contexts with functions, loops and statements; the bottom right panel displays metrics related to the contexts.

We start to look for the locations with high latency. A loop region (line:1284) accounts for 7.9% of the total latency as shown in Figure 84. With the knowledge of the problematic location, we correlate all the reuse pairs with this region. We find that the application stores the arrays `x8n` (line:1525), `y8n` (line:1526) and `z8n` (line:1527) in the loop (line:1509) and later uses them many times in the loop (line:1284) inside the function `CalcFBHourglassForceForElems`. Figure 85 shows a reuse pair of the variable `z8n`. The time reuse distance is $\sim 4 \times 10^7$ for `x8n`, `y8n` and `z8n` and RDX reports high access latencies for these arrays.

To enhance the temporal locality, we fused the two loops (line:1509 and 1284). The fusion is straightforward and safe: they share the same loop bounds and stride and do not change the dependence direction after fusing. The fusion optimization achieves a $1.54\times$ speedup.

F. COMPARISON WITH THREADSPOTTER

ThreadSpotter introduces a large overhead (GeoMean $\sim 17\times$) shown in Figure 86. Furthermore, the time reuse accuracy of ThreadSpotter is always lower than RDX as shown in Figure 87. In certain cases, ThreadSpotter has significantly low accuracy, as low as 37% for `astar`. Figure 88 shows the histogram of `astar`, where ThreadSpotter incorrectly attributes more on the longer reuses and misses many short-distance reuses.

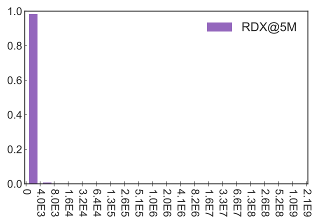

Figure 21: Stack reuse histogram of `perlbench_r-1`.

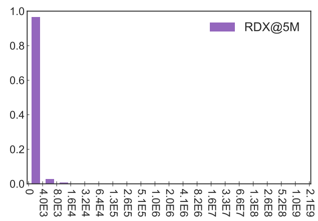

Figure 22: Stack reuse histogram of `perlbench_r-2`.

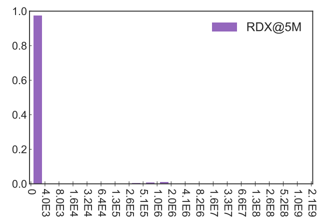

Figure 23: Stack reuse histogram of `perlbench_r-3`.

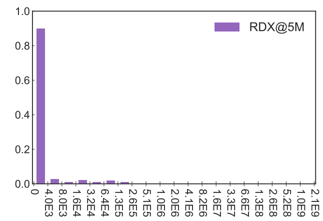

Figure 24: Stack reuse histogram of `gcc_r-1`.

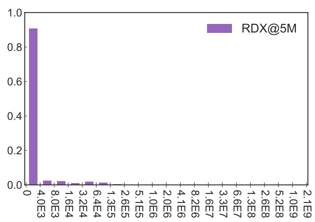

Figure 25: Stack reuse histogram of `gcc_r-2`.

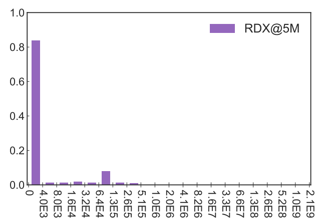

Figure 26: Stack reuse histogram of `gcc_r-3`.

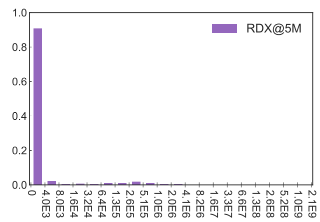

Figure 27: Stack reuse histogram of `gcc_r-4`.

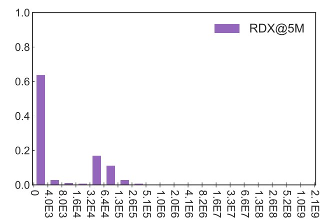

Figure 28: Stack reuse histogram of `gcc_r-5`.

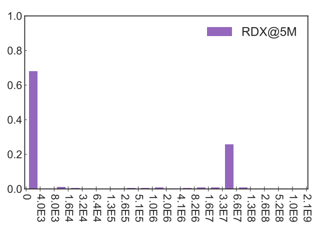

Figure 29: Stack reuse histogram of `bwaves_r-1`.

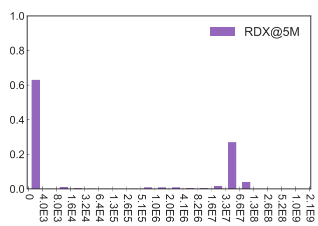

Figure 30: Stack reuse histogram of `bwaves_r-2`.

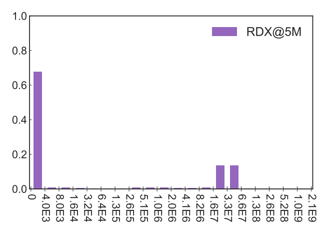

Figure 31: Stack reuse histogram of `bwaves_r-3`.

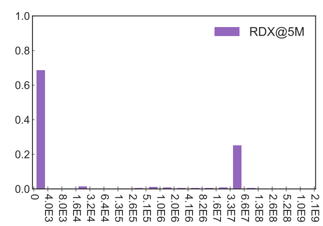

Figure 32: Stack reuse histogram of `bwaves_r-4`.

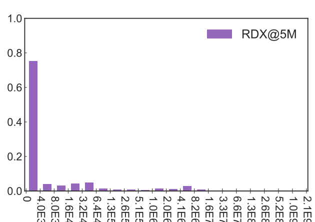

Figure 33: Stack reuse histogram of `mcf_r`.

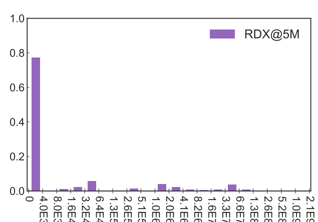

Figure 34: Stack reuse histogram of `cactuBSSN_r`.

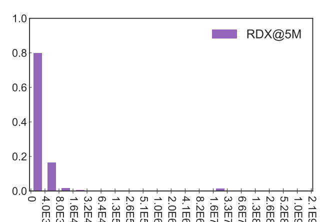

Figure 35: Stack reuse histogram of `namd_r`.

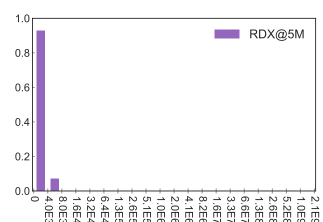

Figure 36: Stack reuse histogram of `povray_r`.

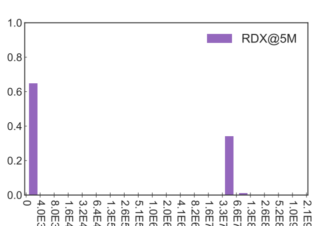

Figure 37: Stack reuse histogram of `lbm_r`.

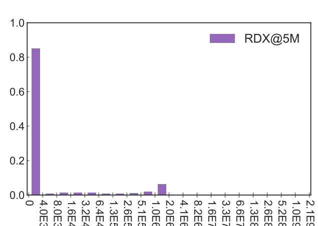

Figure 38: Stack reuse histogram of `omnetpp_r`.

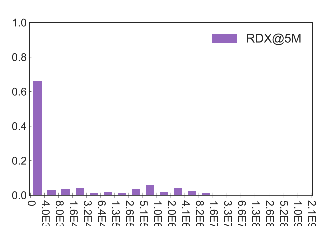

Figure 39: Stack reuse histogram of `wrf_r`.

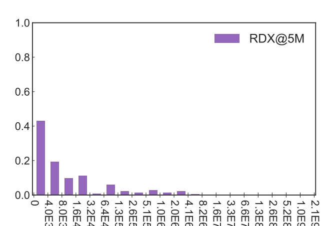

Figure 40: Stack reuse histogram of `xalancbmk_r`.

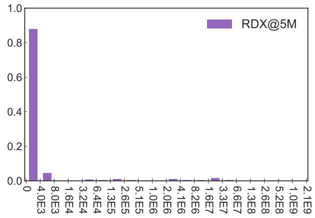

Figure 41: Stack reuse histogram of `x264_r-1`**.**

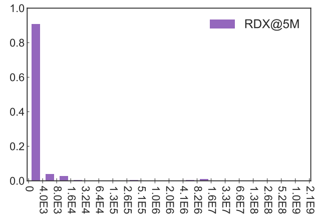

Figure 42: Stack reuse histogram of `x264_r-2`**.**

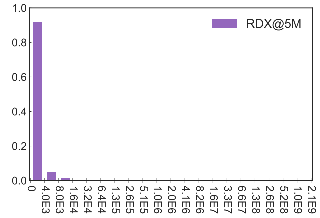

Figure 43: Stack reuse histogram of `x264_r-3`**.**

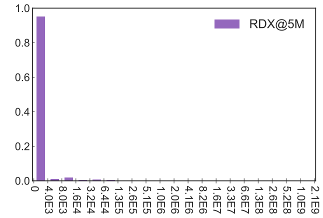

Figure 44: Stack reuse histogram of `blender_r`**.**

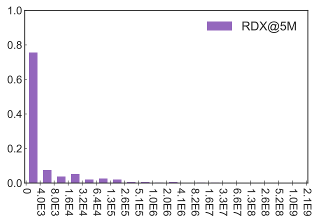

Figure 45: Stack reuse histogram of `cam4_r`**.**

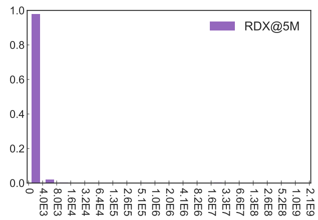

Figure 46: Stack reuse histogram of `deep-sjeng_r`**.**

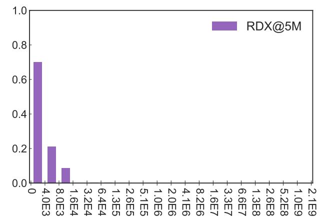

Figure 47: Stack reuse histogram of `imagick_r`**.**

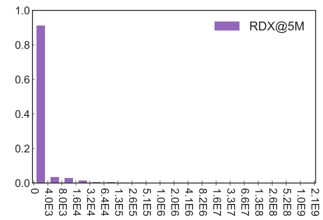

Figure 48: Stack reuse histogram of `leela_r`**.**

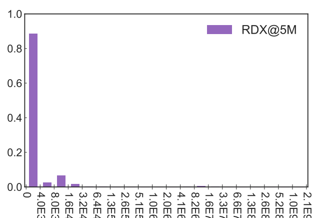

Figure 49: Stack reuse histogram of `nab_r`**.**

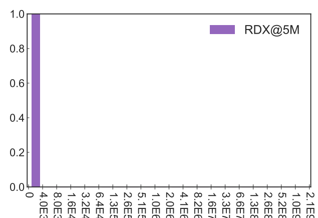

Figure 50: Stack reuse histogram of `exchange2_r`**.**

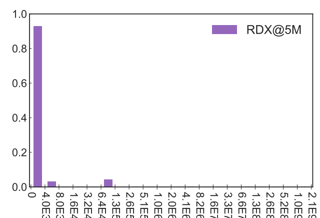

Figure 51: Stack reuse histogram of `fotonik3d_r`**.**

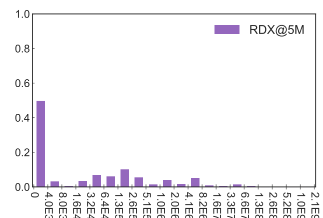

Figure 52: Stack reuse histogram of `roms_r`**.**

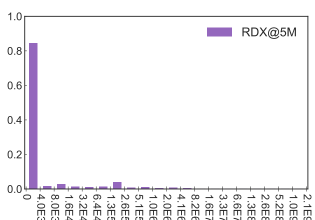

Figure 53: Stack reuse histogram of `xz_r-1`**.**

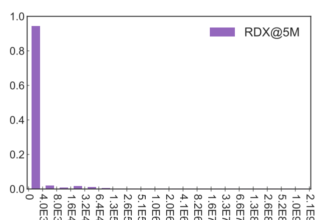

Figure 54: Stack reuse histogram of `xz_r-2`**.**

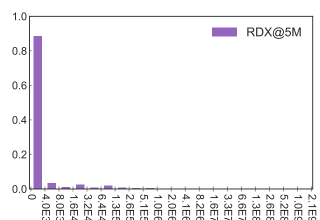

Figure 55: Stack reuse histogram of `xz_r-3`**.**

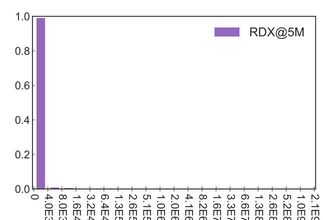

Figure 56: Stack reuse histogram of `perlbench_s-1`**.**

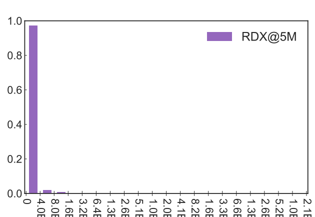

Figure 57: Stack reuse histogram of `perlbench_s-2`**.**

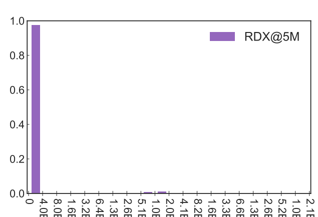

Figure 58: Stack reuse histogram of `perlbench_s-3`**.**

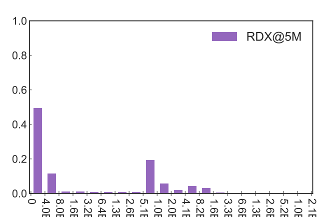

Figure 59: Stack reuse histogram of `gcc_s-1`**.**

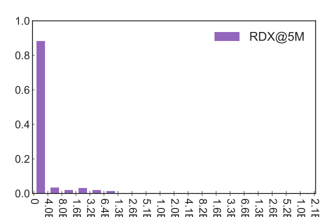

Figure 60: Stack reuse histogram of `gcc_s-2`**.**

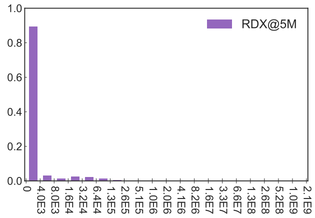

Figure 61: Stack reuse histogram of `gcc_s-3`.

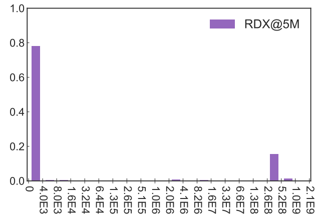

Figure 62: Stack reuse histogram of `bwaves_s-1`.

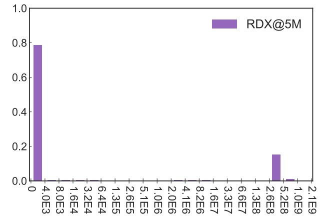

Figure 63: Stack reuse histogram of `bwaves_s-2`.

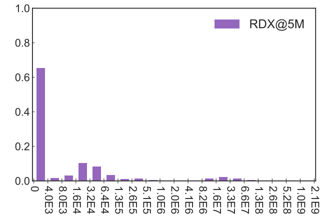

Figure 64: Stack reuse histogram of `mcf_s`.

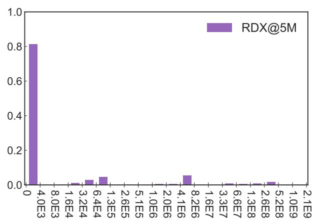

Figure 65: Stack reuse histogram of `cactuBSSN_s`.

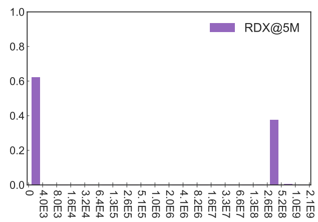

Figure 66: Stack reuse histogram of `lbm_s`.

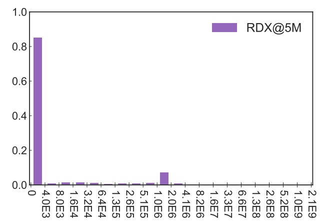

Figure 67: Stack reuse histogram of `omnetpp_s`.

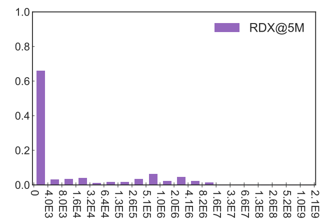

Figure 68: Stack reuse histogram of `wrf_s`.

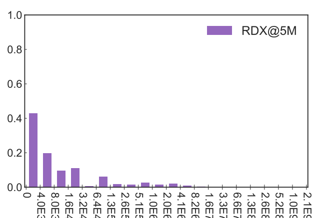

Figure 69: Stack reuse histogram of `xalancbmk_s`.

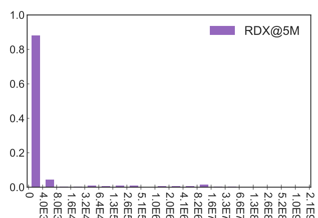

Figure 70: Stack reuse histogram of `x264_s-1`.

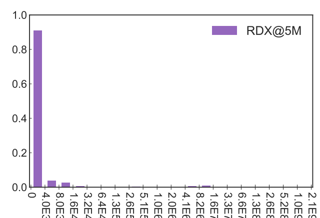

Figure 71: Stack reuse histogram of `x264_s-2`.

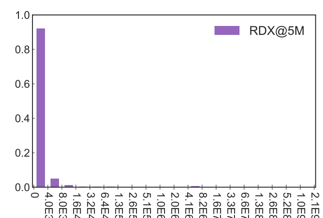

Figure 72: Stack reuse histogram of `x264_s-3`.

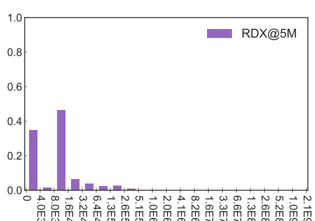

Figure 73: Stack reuse histogram of `cam4_s`.

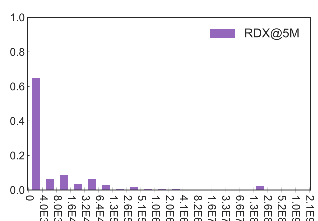

Figure 74: Stack reuse histogram of `pop2_s`.

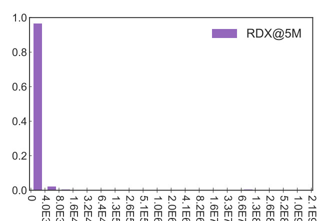

Figure 75: Stack reuse histogram of `deepsjeng_s`.

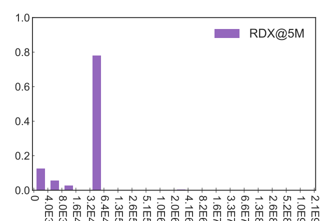

Figure 76: Stack reuse histogram of `imagick_s`.

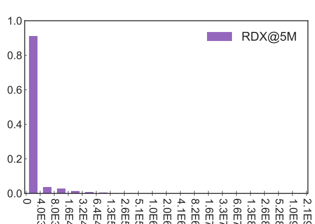

Figure 77: Stack reuse histogram of `leela_s`.

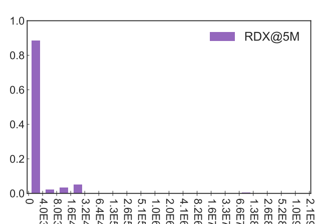

Figure 78: Stack reuse histogram of `nab_s`.

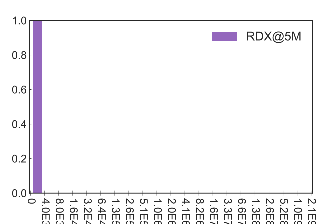

Figure 79: Stack reuse histogram of `exchange2_s`.

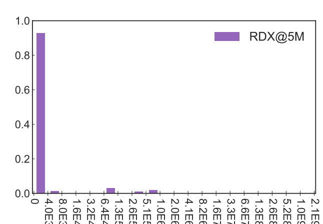

Figure 80: Stack reuse histogram of `fotonik3d_s`.

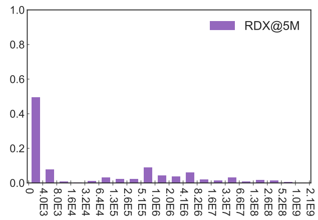

Figure 81: Stack reuse histogram of `roms_s`.

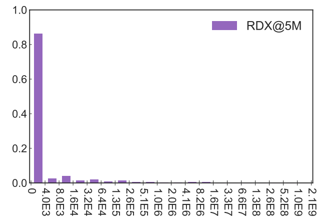

Figure 82: Stack reuse histogram of `xz_s-1`.

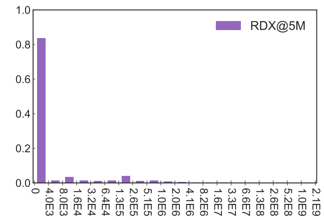

Figure 83: Stack reuse histogram of `xz_s-2`.

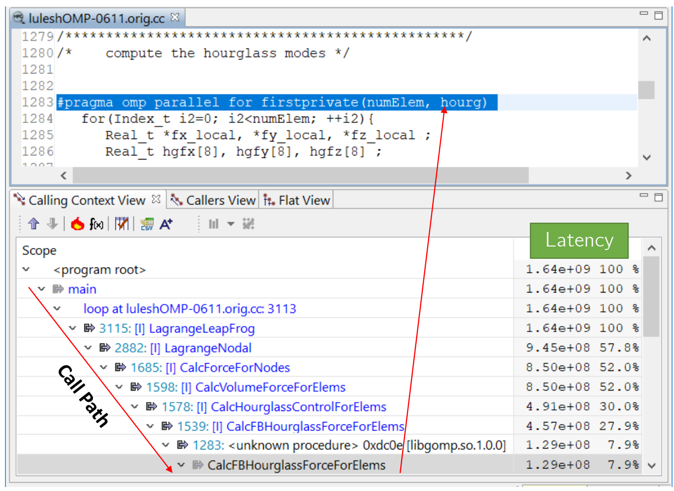

Figure 84: Locate a loop region with high latency in LULESH.

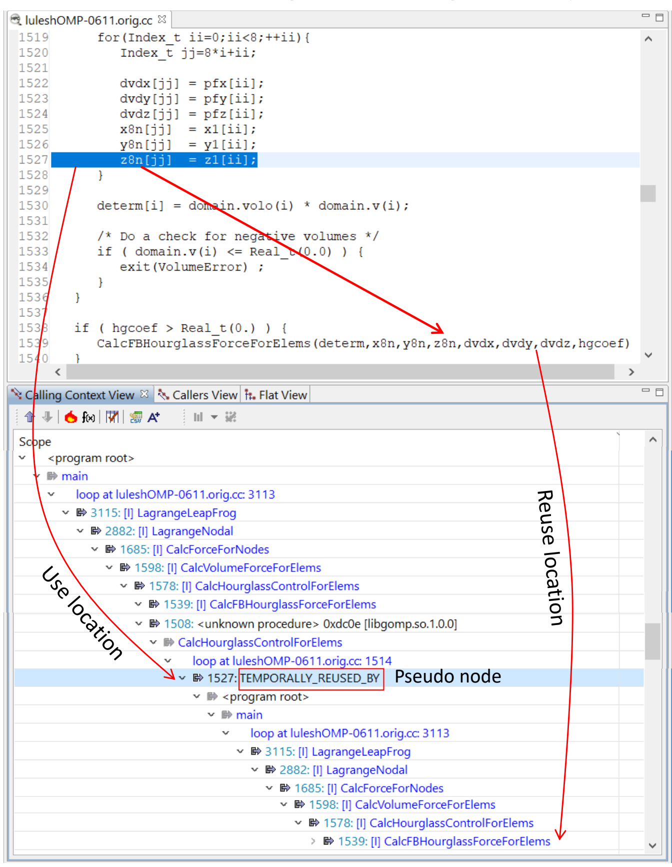

Figure 85: One reuse pair of the variable `z8n` is related to the problematic region shown in Figure 84.

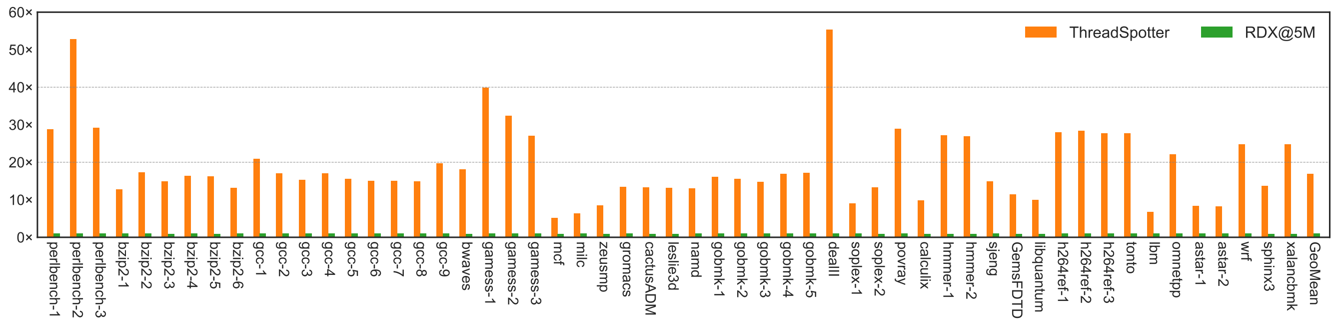

Figure 86: Time overhead comparison between ThreadSpotter (running in default sampling rate) and RDX for SPEC CPU2006 benchmarks. One benchmark may have multiple lines due to runs with different inputs.

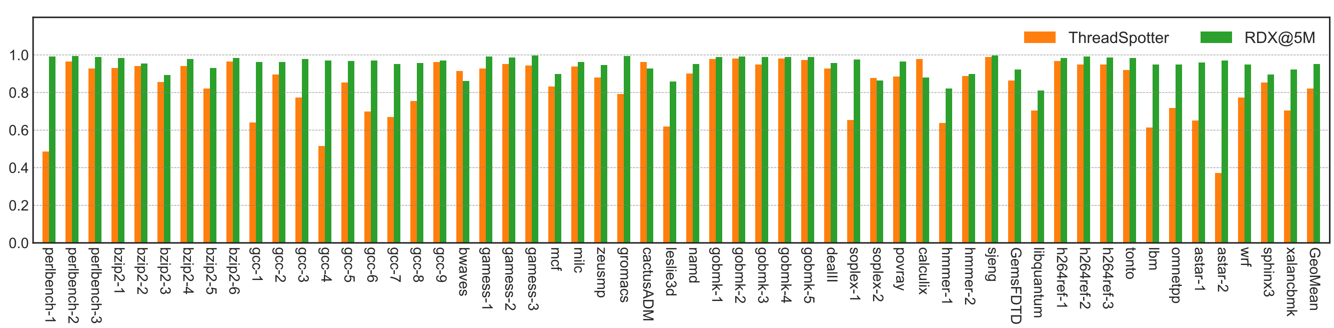

Figure 87: Time reuse accuracy comparison between ThreadSpotter (running in default sampling rate) and RDX for SPEC CPU2006 benchmarks. One benchmark may have multiple lines due to runs with different inputs.

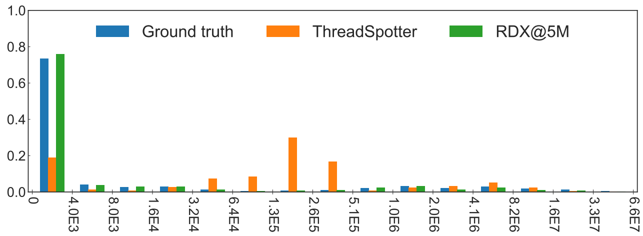

Figure 88: Time reuse histogram of astar (input 2) from SPEC CPU2006 benchmark.

7. REFERENCES

[1] X. Shen, J. Shaw, and B. Meeker, "Accurate approximation of locality from time distance histograms," tech. rep., Technical Report TR902, Computer Science Department, University of Rochester, 2006.

[2] E. Berg and E. Hagersten, "Statcache: a probabilistic approach to efficient and accurate data locality analysis," in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, pp. 20–27, IEEE, 2004.

[3] X. Shen, J. Shaw, B. Meeker, and C. Ding, "Locality approximation using time," in *Proc. of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, POPL '07, (New York, NY, USA), pp. 55–61, ACM, 2007.

[4] Lawrence Livermore National Laboratory, "Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)." `https://computation.llnl.gov/projects/co-design/lulesh`.