Hotspot Detection of SPEC CPU 2006 Benchmarks with Performance Event Counters*

Qiang Wu, Oskar Mencer, Carlos Tavares, and Kubilay Atasu

Department of Computing, Imperial College London,
South Kensington, London SW7 2AZ, UK
{qiangwu,oskar,ctavares,atasu}@doc.ic.ac.uk
http://comparch.doc.ic.ac.uk

Abstract. Hotspot is the part of a program where most execution time is spent. Detecting the hotspot enables the optimization of the program. The performance event counters embedded in modern processors provide the hardware support for the hotspot detection. By sampling the instruction addresses of the running program with performance event counters, hotspot of the program can be statistically detected. This technical report describes our tool to find the sections of the code that are detected as the hotspot of the program with performance event counters. SPEC CPU 2006 benchmarks are tested with our tool and the results show the hotspot sections and overhead of the hotspot detection tool.

Key words: hotspot detection, performance event counters, performance engineering

1 Introduction

Performance engineering of software programs has been an active research topic for years. The goal of performance engineering is to adapt the program to the architecture on which the program is executed in order to satisfy the performance requirement at an acceptable cost[1]. To achieve the performance engineering goal, an insightful understanding of the dynamic characteristics of the original program is needed. One of the dynamic characteristics is the hotspot of the program in execution, which is always the first thing one wants to know about a program, since the hotspot is the part of the program where most of the execution time is spent.

With the progress of the integrated circuits technology, most modern processors are equipped with performance event counters which provide a low overhead facility to investigate running programs[2][3][4][5][6]. The basic functionality provided by the performance event counters is to count the events occured in the hardware microarchitecture, and if configured appropriately, they can issue an interrupt when the count reaches some preset value or in another word, overflow. In the interrupt handler of the performance event counter overflow, one can

* This work is supported by EPSRC grant - Liquid Circuits: Automated Dynamic Hardware Acceleration of Compute-Intensive Applications

record the microarchitecture information available at the moment, such as the Instruction Pointer (IP) address, and other register or performance counter values interested. With the information recorded in the performance event counter overflow interrupts, we can get a statistical profile of the program's execution behavior. For instance, if we let the performance event counters count the unhalted CPU clock cycles and issue an interrupt every ten million cycles, by analyzing the collected IP addresses in the interrupts we can figure out which part in the program is most time-consuming, in a statistical manner. Nowadays most tools utilizing the performance event couters support the IP recording or sampling functionality and are able to aggregate the counts of different IP addresses or to attribute the number of recorded IP addresses to the corresponding functions and modules, resulting in a histogram of the IP addresses distributed among the program constructs[7][8][9][10][11].

The IP address histogram gives the performance engineers a hint to the hotspot of the program. However, relating the recorded IP addresses to the original program is a tedious work. Aggregating the counts of recorded IP addresses to corresponding functions or modules does help, but requires more processing and sometimes is not easy to carry out if the executalbe was built with little source information, which is the common case of operational programs. Since the hotspot of the program includes sequences of instructions instead of solitary instructions, it is not necessary to record each IP address in the statistical sampling. In this technical report, we propose to record the sections of the address space while smapling the IP addresses of the program. That is to say, we attribute the number of sampled IP addresses to the sections of the code in the program rather than split the number among different IP addresses or functions and modules. By counting the number of IP addresses hitting in each section, we need not save all the IP addresses encountered in the sampling process as in the IP address histogram case, nor we need demangle the function or module address regions as in the function or module histogram case, which we believe would be helpful to ease the workload of hotspot detection of the program. Main contributions of our work are outlined as follows:

1. A method to record IP address sections when sampling the IP addresses of the running program.
2. A tool implementing the proposed method for Linux on X86-64 architecture.
3. Preliminary experiments of the tool on SPEC CPU 2006 benchmarks.

The rest of the paper is organized to four sections. The next section describes the method. Section 3 introduces the tool and its design, as well as a brief of the related framework for dynamic software acceleration. Section 4 shows the results of the experiments on SPEC CPU 2006 benchmarks. Section 5 concludes the paper with the brief of future plan.

2 Method

Our idea is to start from relative large sections of instruction addresses. If the hit count of IP addresses in one section reaches some preset threshold in sampling

process, we split the section into smaller ones and continue the process recursively until the size of the section is small enough for post-processing.

2.1 Section Tree

The splitting operation mentioned in above generates a tree of sections. Each section is a node in the tree. If a section is to be splitted, the resultant smaller sections are regarded as the children of the original section, hence forming a tree.

Since the entire range of the possible address space is too big to begin with, especially for the AMD 64-bit platforms where the virtual address space can span to 256 tera bytes, we start the splitting from a certain size smaller than the entire address space. From the orignal section, a tree will be built during the sampling process. These original sections are collected in a list as the roots of the section trees. Each time an IP address is sampled, the section trees in the list are searched to locate the section in which the IP address hits. Then the corresponding section tree will be updated to increase the hit count of the section or split the section further if the hit count is over preset limit. After that, the updated section tree will be swapped to the beginning of the section tree list, hoping that the next sampled IP address will hit in this section tree again, hence reducing the search time.

2.2 Splitting Mechnism

Splitting is controlled by a preset threshold of the count that the sampled IP addresses hit in the given section. If the hit count of a section exceeds the threshold, the section will be splitted. Each child section will have the same threshold of the splitted section. But the hit count of the original section is divided evenly among the child sections, meaning that the hits are treated distributing uniformly in the whole section.

Since the threshold is the same for the section and its child sections, the hit density of the splitting section increases with the growth of the section tree. Suppose the threshold is T, the size of the section is S, and the section will be spltted to m child sections if the hit count exceeds T. So S/m is the size of each child section. Obviously, the maximal density of the section before splitting is $d_{section} = T/S$. Similarly the maximal density of one child section before the hit count saturates is $d_{childsection} = T/(S/m) = m * (T/S) = m * d_{section}$.

This gives a zoom-in effect when we follow the hits to the hotspot sections of the program. The denser the hits are, the smaller the size of the section is. Generally, the section that recieves more hits is more likely the hotspot. At the same time, this section may have a small size resulting from a series of splitting, hence is easier for us to locate where the hotspot is. In addition, to locate the hotspot more accurately, the minimal and maximal IP addresses hitting in the given section are also reocrded, giving a possibility to narrow down the focus even below the smallest section size.

As mentioned before, the section will be splitted until small enough for analysis. The minimal size is determined at a balance of accuracy and performance.

Similarly, the starting size of the original section is determined at a balance of performance and memory space efficiency. These parameters are chosen with some tests.

3 Tool

We developed a tool to detect the hotspot of the program based on the method described in section 2.

3.1 Design

The tool utilizes the infratructure of pfmon[11] from HP for the manipulation of performance event counters. A kernel module, a user module and a wrapper script are developed and integrated with other tools in our dynamic software acceleration framework. The kernel module is in charge of handling the interrupts generated by the performance event counters during the sampling process. It stores the IP addresses and if instructed, the return addresses in the stack, in a kernel buffer. When the buffer is about to be full, the kernel module notifies the user module to read and process the IP addresses saved in the buffer.

The user module is in charge of building the section tree with the IP addresses passed from the kernel buffer. It checks each address, finds which section the address falls in, then increases the hit count of that section. If the section's hit count exceeds the threshold, the section will be splitted and child sections are inserted into the section tree.

If instructed by the user, the user module can print out the current hotspot section list at a specified period during the sampling. The hotspot section list is generated from the leaf sections in the section tree which have the minimal section size and a hit count above the given threshold. The period is specified in terms of second. The minimum is one second, allowing enough time for the module to process the IP addresses.

At the end of the execution, the user module will print out the resultant hotspot section list by order of hit count. Each section is printed with its starting address and size. The minimal and maximal addresses ever encountered in each section are also printed, in order to give a starting point in post-processing.

The kernel and user modules are programmed in C and built within the pfmon infrastructure. They can be used as the other smapling modules provided in pfmon. However, to ease the user's experience, a wrapper script by Python is also developed. It encapsulates the usage of pfmon with our custom sampling module. The user can specify the options in command line supported by the kernel and user modules we developed. Other options and commands required to launch the pfmon are coded in the script without the need of user intervention. With the wrapper script, the user can specify the program he wants to monitor, and the options such as the period of section tree print-out. The script parses the input information, find the program and its arguments, then lauches the pfmon to monitor and sample the program's execution, printing out the hotspot section list detected during the execution.

3.2 Framework

The hotspot detection tool is part of our dynamic software acceleration framework. After detecting the hotspot of the program, the post-processing tools will analyze the hotspot section, figure out the control and data flow of the code, extract acceleratable sequences of instructions, and convert them to appropriate form for acceleration. The details of the post-processing tools are beyond the scope of this paper. We will report the design and implementation of these post-processing tools in future publications.

4 Experiment

We perform some experiments on SPEC CPU 2006[12] benchmarks with the tool we developed. All the benchmarks in SPEC CPU 2006 has been tested. Runtimes are recorded and compared with those of SPEC CPU 2006 benchmarks without sampling. Memory size occupied by the section trees are caculated also to figure out the memory consumption of the tool.

4.1 Test Platform

The test platform is a desktop computer with 2 AMD opteron dual core processors running at 2210 MHz. The CPU family number is 15, model number is 33, stepping is 2. The processor cache size is 1024 KB. The size of the system RAM is 2 GB.

Operating system is Ubuntu Linux 7.10 with kernel 2.6.24.3, configured to run in 64-bit mode. The kernel is patched with perfmon[11] interface to the performance event counters. The version of perfmon kernel patch is 2.8 which is required for pfmlib[11] and pfmon[11] tool version 3.3.

SPEC CPU 2006 benchmarks are installed on the system, and are built with GCC and GFortran 4.1.2. Optimization switch for the GCC and GFortran is -O2, and the debug information switch is -gdwarf-2. It should be noted the debug information is embedded for the post-processing tools, not for the hotspot detection tool. The hotspot detection tool does not use the debug information in the experiments.

4.2 Test Parameters

The starting and stopping section sizes are set to be 1M bytes and 1K bytes respectively. The preliminary tests reveal that most programs occupy a memory space ranging from dozens of kilobytes to several megabytes. So we choose the starting and stopping section sizes as one megabytes and one kilobytes.

The hit count threshold is selected to be 128. The reason is as follows. We expect the hotspot section to be hit once at each word in average during the execution. Since the stopping section size is chosen as 1 kilobytes, or 128 words for a 64-bit platform, the number 128 means each word in the section will get one

hit in average. Higher thresholds can assure more hits per word in the section, but may lead to slower splitting, hence are not selected.

The number of child sections split from the original section is set to be 4. This means the section tree is a four-ary tree. Before choosing 4 as the splitting number, we initially used the binary tree to hold the sections. However, the binary tree needs up to 10 splittings from the original section size of one megabytes to the stopping section size of one kilobytes. So we changed to four-ary tree, which has less hierarchies when split from 1M bytes section to 1K bytes section than the binary tree does. Higher child section numbers are not selected because we find in practice, splitting by 4 is efficient both in execution speed and memory space. A higher child section number requires a larger node size of the section tree. Therefore it is preferrable to use a fair child section number rather than a large one.

4.3 Test Results

We set the period of printing out the intermediate hotspot sections detected during the execution to be 10 seconds, then we launch the SPEC CPU 2006 benchmarks with our hotspot detection tool. The graphs depicting the hotspot sections of the SPEC CPU 2006 benchmarks in the test can be found in the Appendix.

Fig. 1 shows the maximal memory ever occupied by the section tree of each benchmark during the execution. The X axis lines the name of the benchmarks. Since each SPEC CPU 2006 may have several different input data sets, a input number is attached to each benchmark name. The Y axis gives the size of the maximal memory space occupied in bytes. It can be seen in the Fig. 1, that the maximal memory overhead is around 35 KB. For most benchmarks, the hotspot section tree consumes about 5 to 15 KB memory, some of them occupy only less than 5 KB of the memory.

To find out the runtime overhead introduced by the hotspot detection, we add intructions to read time stamp counter values at the start and end of the hotspot detection procedure. The additional run time for hotspot detection is calculated with $T_{detection} = (ts_{end} - ts_{start})/cpu\_clock\_rate$. The original run times of the SPEC CPU 2006 benchmarks are obtained by the time command. We run each benchmark without hotspot detection 5 times repeatedly. For each execution we use time command to get the User mode run time, System mode run time and Wall clock real time. Since the wall clock real time may be influenced much by the dynamic environment of the system, we choose the sum of the User mode and System mode run time as final run time of each execution.

Table 1 lists the minimal, maximal and average runtimes of SPEC CPU 2006 benchmarks without hotspot detection, as well as the hotspot detection time of each benchmark. Fig. 2 shows the runtime overhead percentage of the hotspot detecion on the SPEC CPU 2006 benchmarks. The X axis lines the name and input number of each benchmark. The Y axis represents the runtime overhead in percent of the benchmark. The percentage is calculated by $P_{overhead} = T_{detection}/MinTime_{benchmark} * 100$. We choose the minimal run

time of each execution in consideration that the minimal run time should have least influence from the dynamic environment of the system. From the figure, we can find that the runtime overheads are less than 0.04%, with some of them are close to 0.01%.

5 Conclusion

In this technical report, we introduce a hotspot detection tool to record hotspot sections instead of each IP addresses. One advantage of recording hotspot sections rather than IP addresses is to save memory space required for storing the different IP addresses encountered during the execution. Another advantage is that by collecting the IP addresses in sections, we obtain some information about the relations among the encountered IP addresses, such as concurrencies of different IP addresses, which is believed to be helpful in post-process of the performance monitoring data. We have seen from the periodical hotspot section graphs in Appendix that different sections of the program can be active at different times during the execution. Thus the hotspot sections with periodical sampling reveal the program's dynamic characteristics statistically in both the IP space and the time frame.

Meanwhile, the memory cost of the hotspot section storage is relatively low. In our experiments with SPEC CPU 2006 benchmarks, the maximal memory ever consumed by the hotspot section is about 35 KB, while for most benchmarks this consumption is around 5 to 15 KB. So we can have the hotspot sections resident in the main memory and reference the hotspot section information during the execution. In our future work, we plan to utilize the hotspot section information for dynamic acceleration of the program.

From the experiments, we also find that the run time overheads of the hotspot section detection are not high. Run time overheads of the SPEC CPU 2006 benchmarks with hotspot detection are within 0.04% of the minimal run time ever recorded for the original programs. For some benchmarks, the run time overheads are close to 0.01% of the original run time.

In a word, the preliminary experiments so far show that the hotspot detection based on IP sections is a promising approach to study the dynamic characteristics of the program. Our next step is to extend it with post-processing and analysis techniques, such as binary code extraction and transformation for the dynamic accleration of the programs.

References

1. Reiner R. Dumke, Claus Rautenstrauch, Andreas Schmietendorf, Andre Scholz (Eds.): Performance Engineering, State of the Art and Current Trends. Lectur Notes in Computer Science vol. 2047, Springer (2001)
2. Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, et al. Continuous profiling: where have all the cycles gone? ACM Transactions on Computer Systems (TOCS). Vol. 15, Iss. 4, pp. 357-390. 1997.

3. Glenn Ammons, Thomas Ball, James R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 85-96. 1997.
4. Reza Azimi, Michael Stumm, Robert W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In: Proceedings of the 19th annual international conference on Supercomputing (ICS), pp. 101-110. 2005.
5. John L. Henning. Performance counters and development of SPEC CPU2006. ACM SIGARCH Computer Architecture News. Vol. 35, Iss. 1, pp. 118-121. 2007.
6. Stephane Eranian. What can performance counters do for memory subsystem analysis? In: Proceedings of the ACM SIGPLAN workshop on Memory systems performance and correctness held in conjunction with the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 26-30. 2008.
7. Innovative Computing Laboratory (ICL), College of Engineering at the University of Tennessee. Performance Application Programming Interface (PAPI). `http://icl.cs.utk.edu/papi/index.html`
8. Intel Corporation. Intel VTune Performance Analyzer. `http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm`
9. AMD Inc. AMD CodeAnalyst Performance Analyzer. `http://developer.amd.com/CPU/Pages/default.aspx`
10. OProfile - A System Profiler for Linux. `http://oprofile.sourceforge.net`
11. perfmon project, `http://perfmon2.sourceforge.net`
12. Standard Performance Evaluation Corporation, `http://www.spec.org`

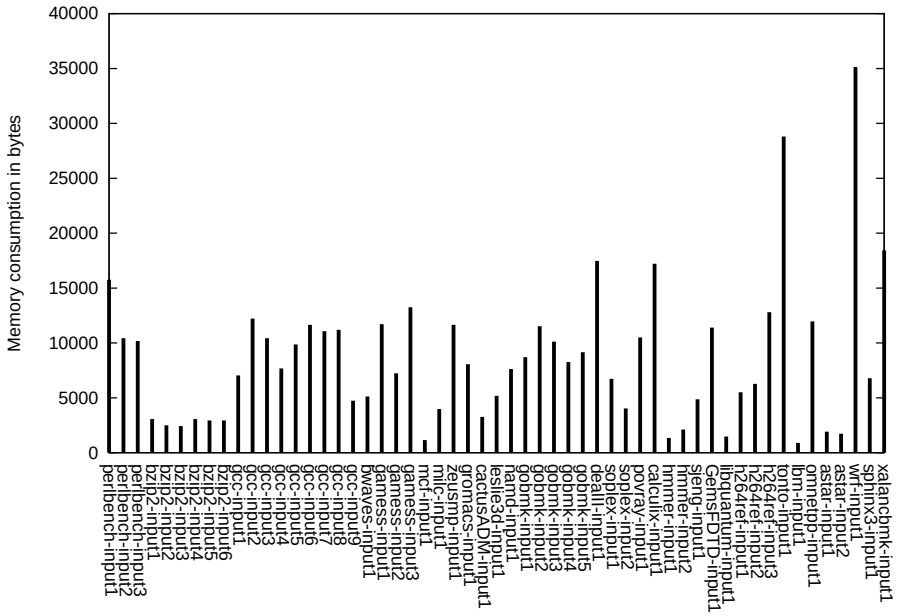

Fig. 1. Memory consumption of the hotspot detection tool for the benchmarks

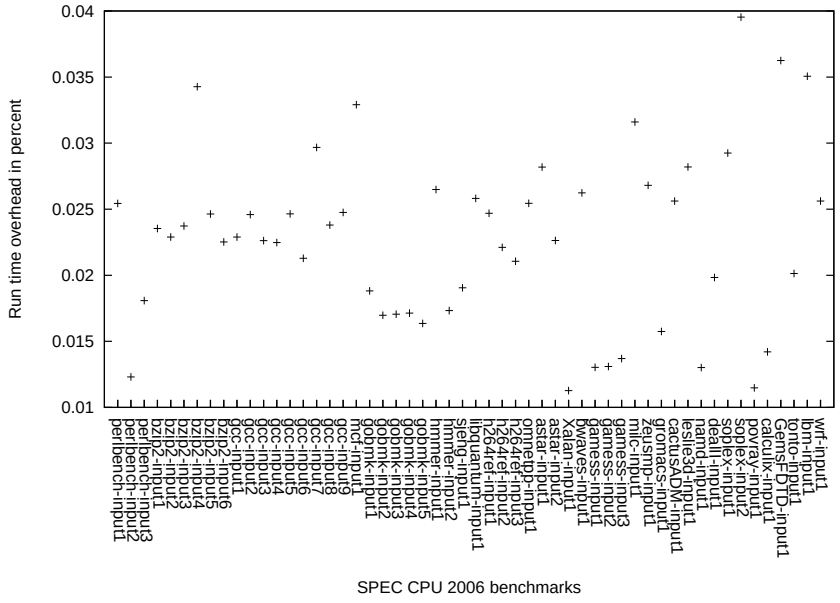

Fig. 2. Run times difference between SPEC CPU 2006 benchmarks with and without hotspot detection

Table 1. The run times of SPEC CPU 2006 benchmarks and the hotspot detection overhead

Benchmark	Min time	Max time	Avg time	Overhead
perlbench-input1	456.36	460.71	458.094	0.116070893
perlbench-input2	146.08	147.41	146.63	0.017976389
perlbench-input3	222.46	226.74	224.97	0.04022204
bzip2-input1	229.79	232.64	231.926	0.05408798
bzip2-input2	92.72	95.24	93.986	0.021223212
bzip2-input3	190.49	197.33	195.174	0.045197416
bzip2-input4	229.77	247.61	234.092	0.078738794
bzip2-input5	273.72	307.53	300.206	0.067435971
bzip2-input6	187.88	189.07	188.634	0.042314554
gcc-input1	77.04	77.73	77.374	0.017633632
gcc-input2	103.73	104.28	103.984	0.025511858
gcc-input3	128.6	138.38	134.34	0.029095956
gcc-input4	88.66	95.92	92.972	0.019924354
gcc-input5	103.28	122.7	114.81	0.025456447
gcc-input6	176.63	178.43	177.858	0.037593003
gcc-input7	193.94	219.72	214.156	0.057549938
gcc-input8	199.47	201.24	200.308	0.047485616
gcc-input9	34.88	35.07	34.992	0.008635035
mcf-input1	1463.15	1586.59	1536.656	0.481510989
gobmk-input1	110.33	111.07	110.732	0.020758631
gobmk-input2	285.7	286.5	286.156	0.048490138
gobmk-input3	156.15	156.54	156.264	0.02663531
gobmk-input4	109.16	109.59	109.34	0.018703295
gobmk-input5	149.23	151.87	149.816	0.02440289
hmmer-input1	374.19	381.77	377.024	0.099132176
hmmer-input2	763.15	769.78	766.818	0.132229487
sjeng-input1	1114.34	1118.02	1116.944	0.212317577
libquantum-input1	1700.45	1710.69	1706.968	0.439062569
h264ref-input1	200.58	202.27	201.178	0.049523845
h264ref-input2	142.03	143.59	142.664	0.031402884
h264ref-input3	1292.58	1296.0	1294.398	0.272224459
omnetpp-input1	894.17	906.47	901.702	0.227561194
astar-input1	444.53	459.04	453.264	0.125284414
astar-input2	610.28	629.12	619.49	0.138127209
Xalan-input1	825.98	831.5	829.996	0.093072355
bwaves-input1	2748.42	2883.87	2855.558	0.721040153
gamess-input1	361.18	373.97	369.702	0.047064231
gamess-input2	233.55	235.43	234.402	0.030591559
gamess-input3	1135.73	1148.15	1140.946	0.155524513
milc-input1	973.3	1056.62	1007.206	0.307592858
zeusmp-input1	1215.1	1216.6	1216.138	0.325735739

Table 1. The run times of SPEC CPU 2006 benchmarks and the hotspot detection overhead (continue)

Benchmark	Min time	Max time	Avg time	Overhead
gromacs-input1	1005.35	1006.01	1005.518	0.158280145
cactusADM-input1	2054.36	2272.86	2152.436	0.526251359
leslie3d-input1	1236.48	1244.11	1240.926	0.348669861
namd-input1	817.66	819.18	818.288	0.106409637
dealII-input1	968.15	1014.94	984.902	0.192005753
soplex-input1	507.06	513.27	509.828	0.148307147
soplex-input2	469.7	598.06	521.178	0.185700659
povray-input1	430.51	436.39	434.136	0.04940989
calculix-input1	3063.97	3085.22	3080.148	0.435295182
GemsFDTD-input1	1424.7	1696.13	1585.128	0.516489503
tonto-input1	1209.26	1221.78	1215.696	0.243500487
lbm-input1	1522.18	1766.4	1668.792	0.533738552
wrf-input1	1600.49	1690.94	1636.66	0.409940721
sphinx-input1	1641.51	1872.27	1734.18	0.588052983

Appendix: Graphs of Hotspot Sections Detected in SPEC CPU 2006 Benchmarks

The graphs in the following show the hotspot sections detected at each interval of time during the execution of SPEC CPU 2006 benchmarks. The X axis of each graph represents the time in seconds. Each tic along the X axis separates the sampling period of the hotspot sections. Currently the period is set to be 10 seconds long. Since the benchmarks do not have an execution time as the times of 10 seconds, the last period on the X axis of each graph is not necessarily 10 seconds. The Y axis represents the sections of IP addresses. Each pair of tics along the Y axis indicates one section and one separating space alternatively. The colored boxes in the graphs indicates the hit density of the section at the given time interval. If a section has more IP addresses falling in during the specific time interval, the color of the corresponding box is darker. The hit density is calculated by dividing the number of IP addresses falling in the section at the given time interval with the size of the section and the length of the time interval.

Form the graphs, we can see that most SPEC CPU 2006 benchmarks have several hotspot sections, but not active all the way of the execution, probably due to the dynamic characteristics of the benchmarks, or due to the incomplete coverage of the statistical sampling. Exceptional cases are the graphs of 470.lbm and 481.wrf benchmarks. 470.lbm seems to have steadly active hotspot sections during the execution, while 481.wrf seems to have no easily visuable hotspot sections in its graph. We may make more specific experiments for the 470.lbm and 481.wrf in the future to find out the detailed reasons.

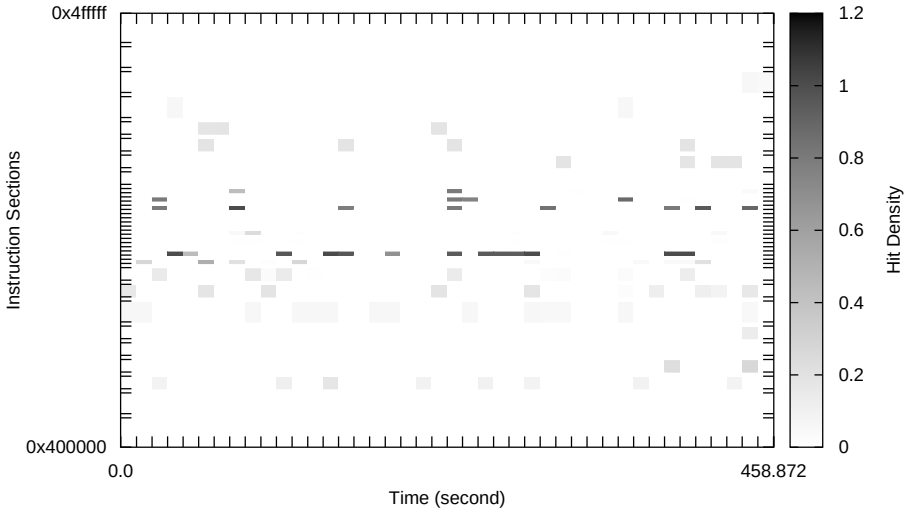

Fig. 3. Hotspot sections of 400.perlbench with input set 1

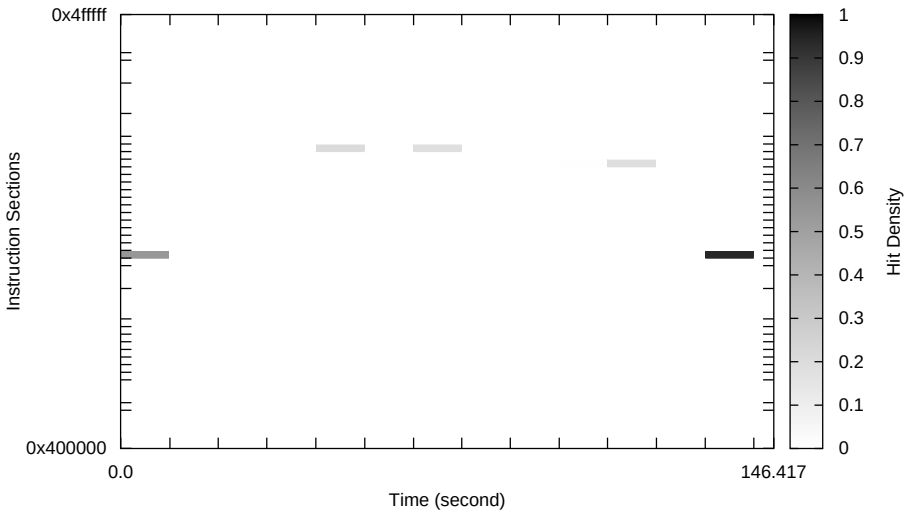

Fig. 4. Hotspot sections of 400.perlbench with input set 2

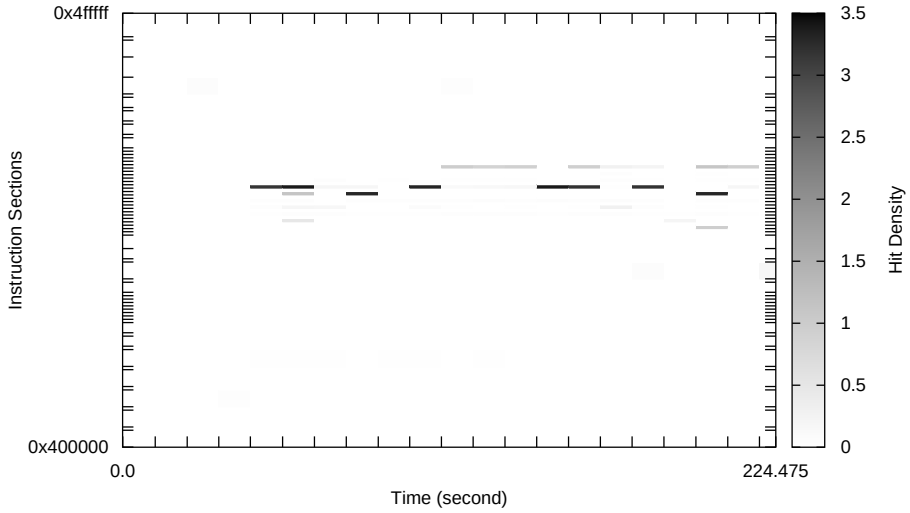

Fig. 5. Hotspot sections of 400.perlbench with input set 3

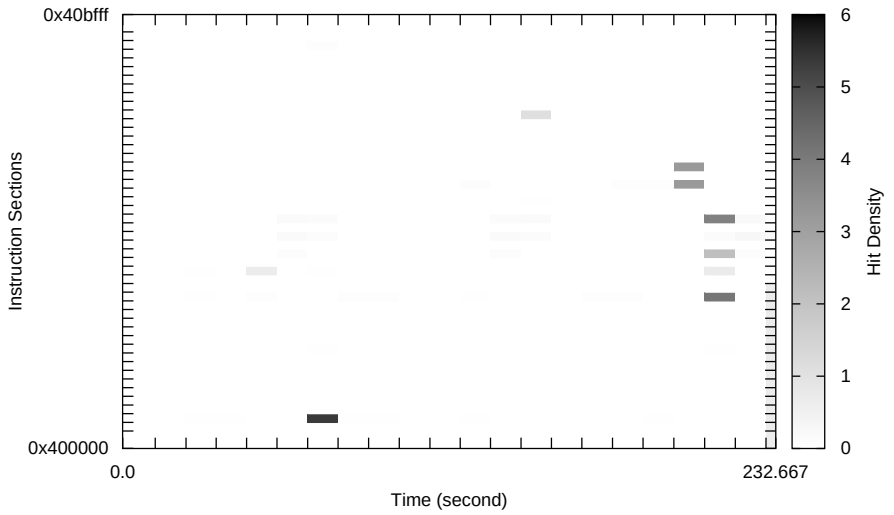

Fig. 6. Hotspot sections of 401.bzip2 with input set 1

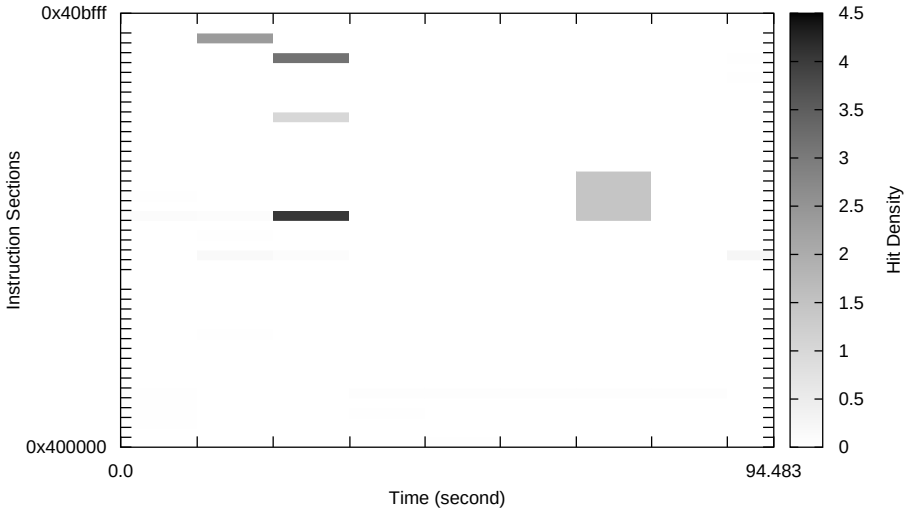

Fig. 7. Hotspot sections of 401.bzip2 with input set 2

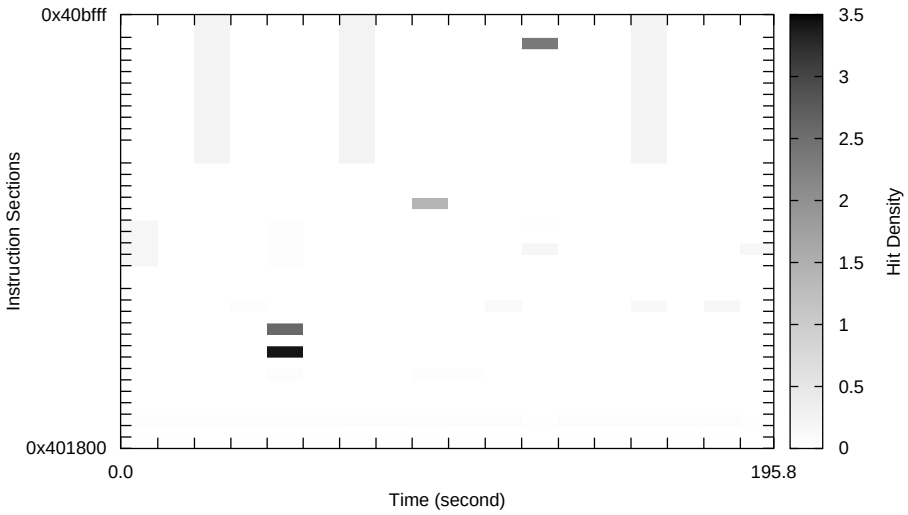

Fig. 8. Hotspot sections of 401.bzip2 with input set 3

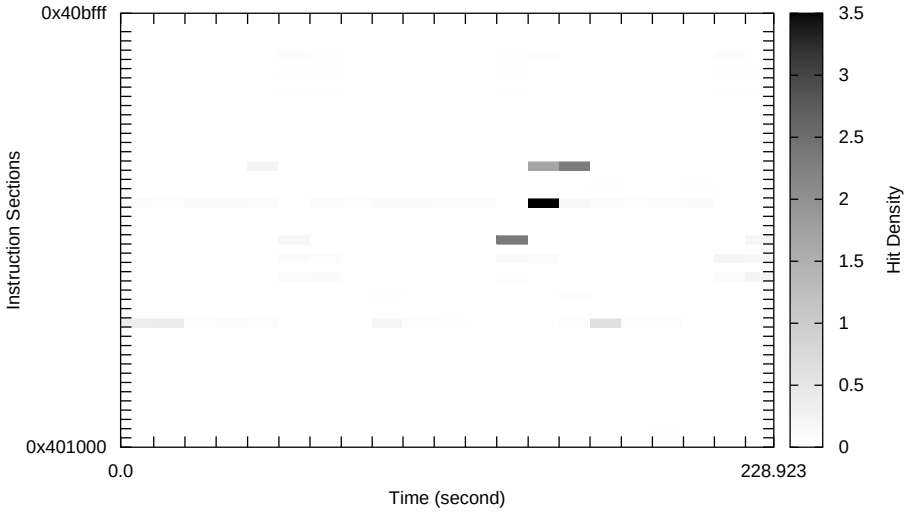

Fig. 9. Hotspot sections of 401.bzip2 with input set 4

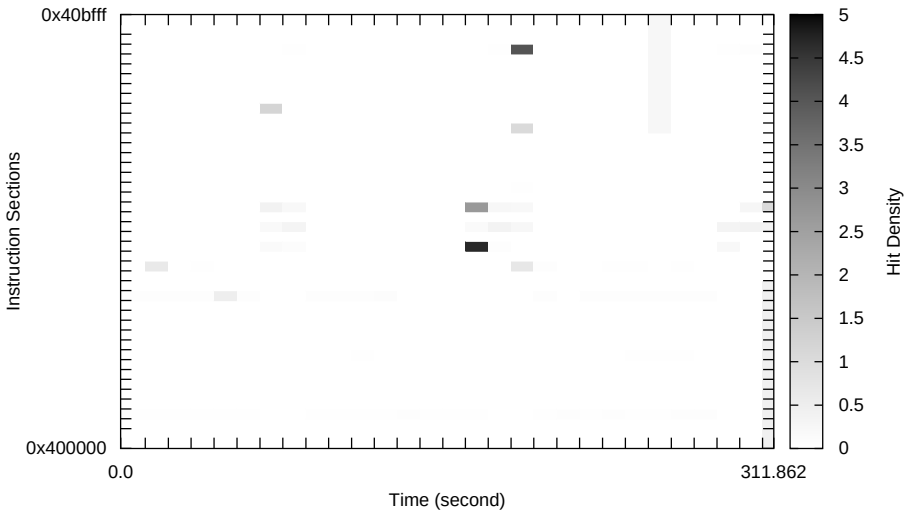

Fig. 10. Hotspot sections of 401.bzip2 with input set 5

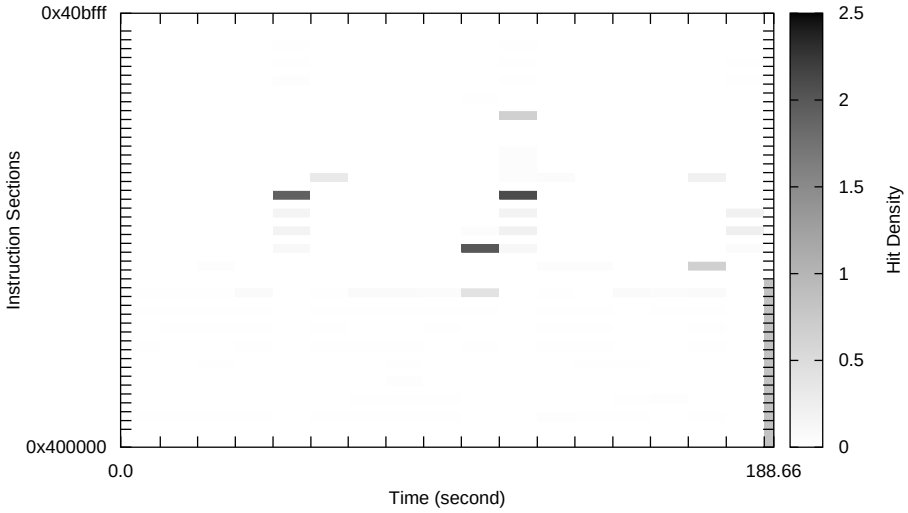

Fig. 11. Hotspot sections of 401.bzip2 with input set 6

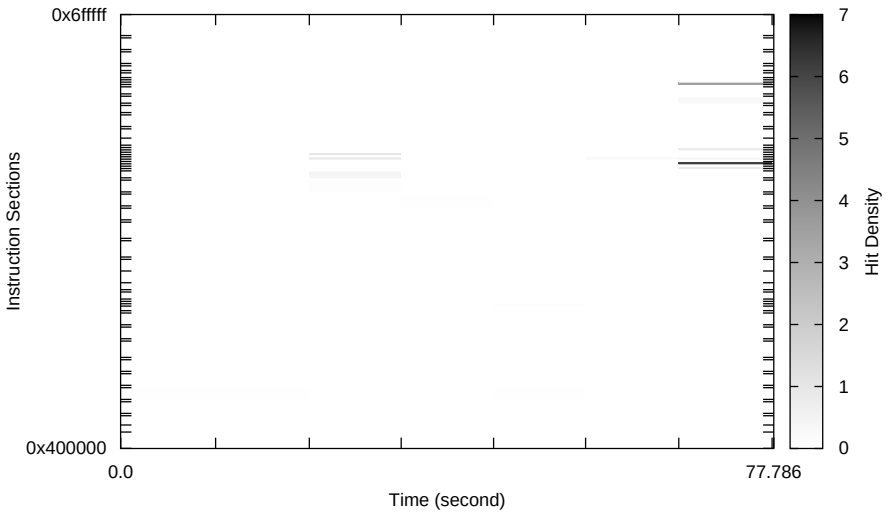

Fig. 12. Hotspot sections of 403.gcc with input set 1

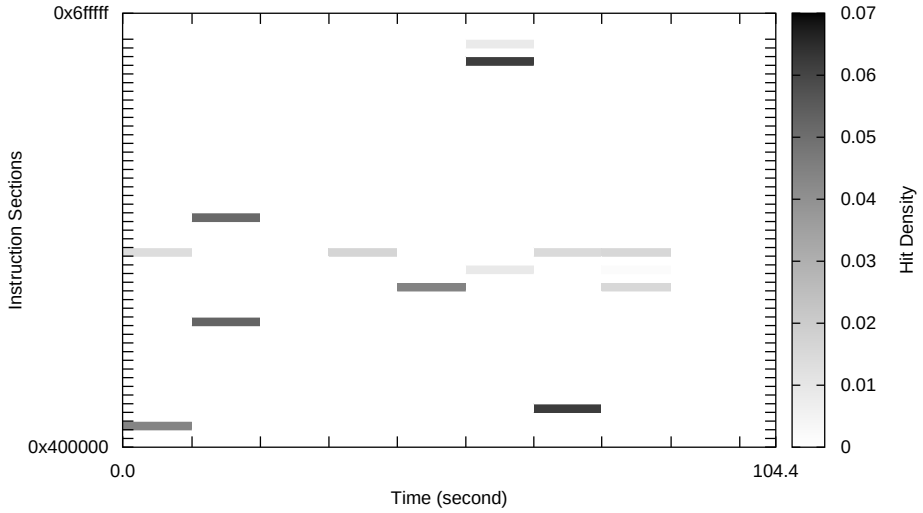

Fig. 13. Hotspot sections of 403.gcc with input set 2

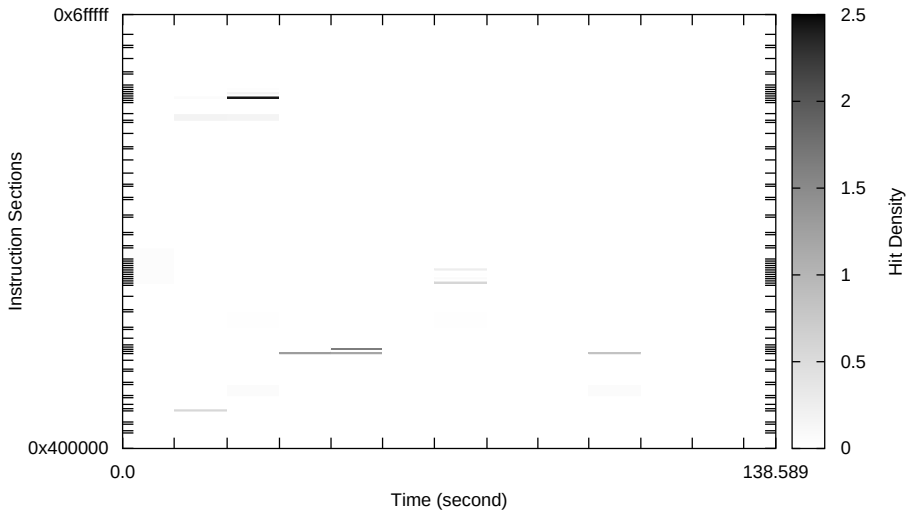

Fig. 14. Hotspot sections of 403.gcc with input set 3

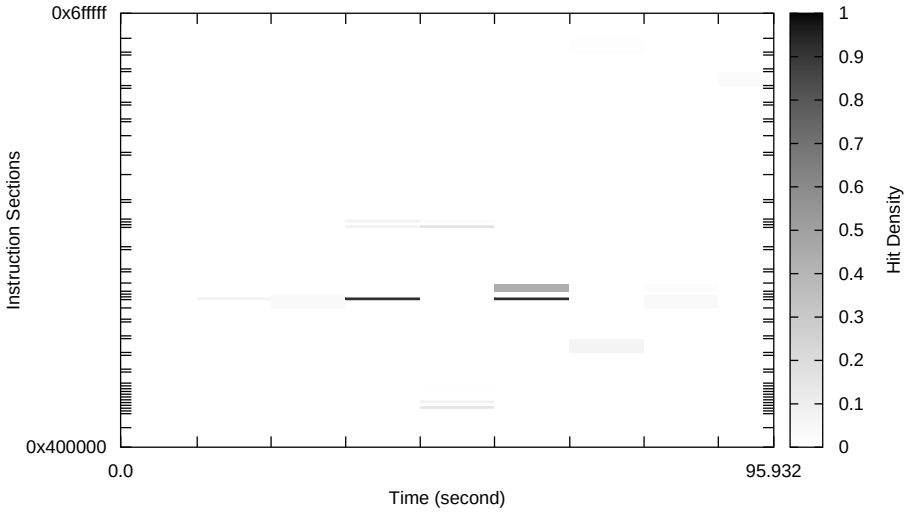

Fig. 15. Hotspot sections of 403.gcc with input set 4

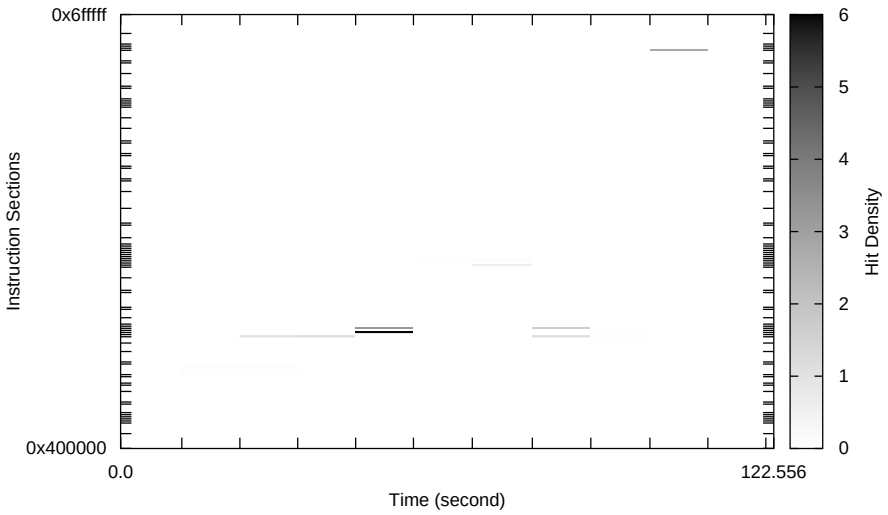

Fig. 16. Hotspot sections of 403.gcc with input set 5

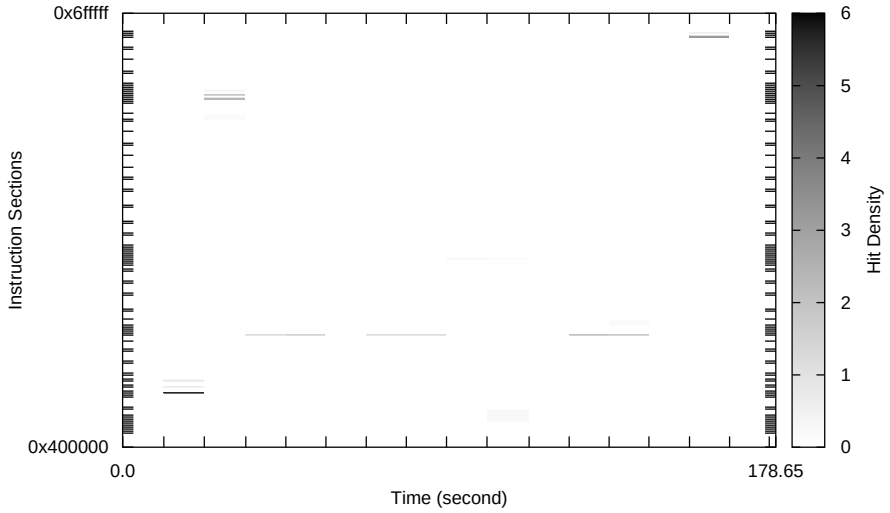

Fig. 17. Hotspot sections of 403.gcc with input set 6

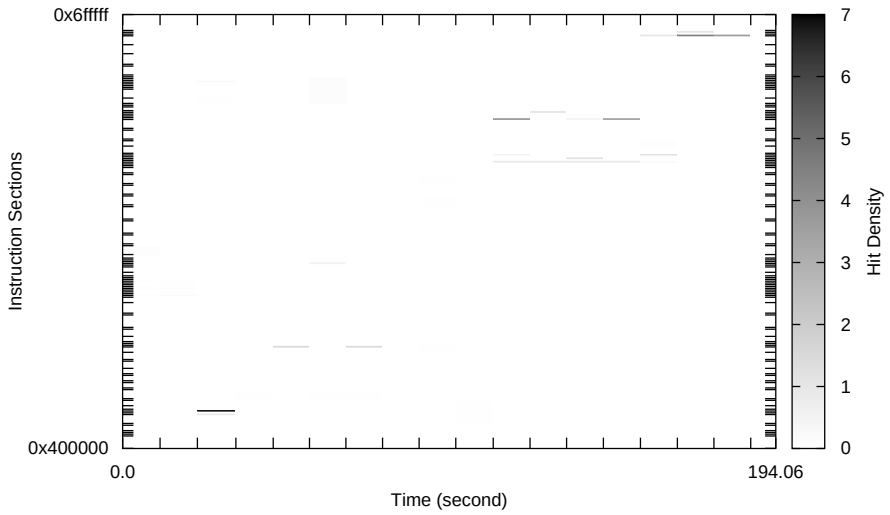

Fig. 18. Hotspot sections of 403.gcc with input set 7

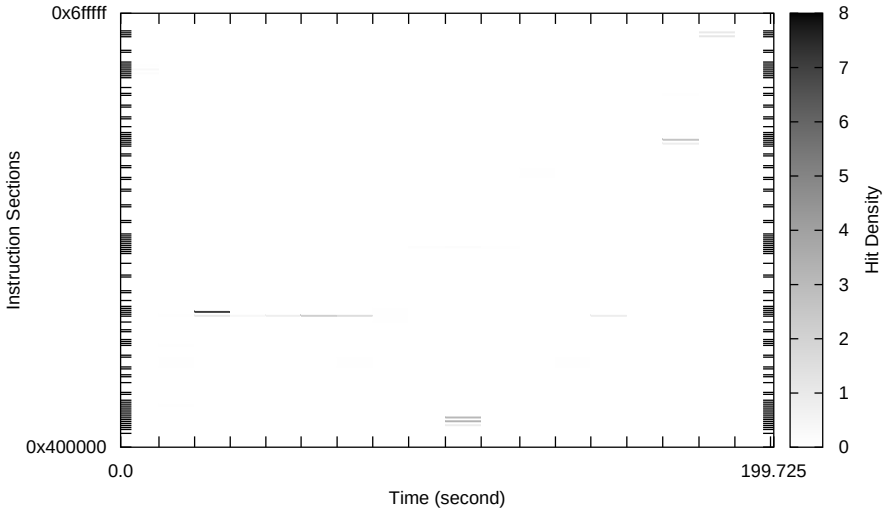

Fig. 19. Hotspot sections of 403.gcc with input set 8

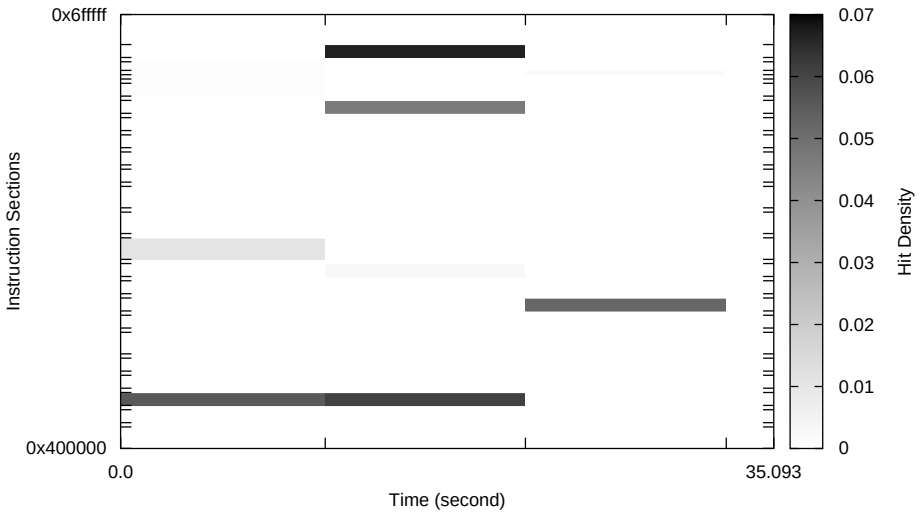

Fig. 20. Hotspot sections of 403.gcc with input set 9

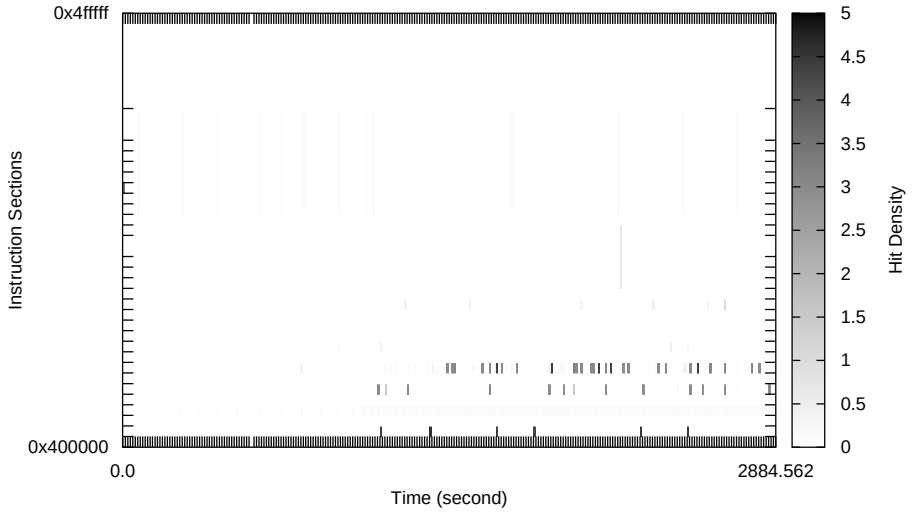

Fig. 21. Hotspot sections of 410.bwaves with input set 1

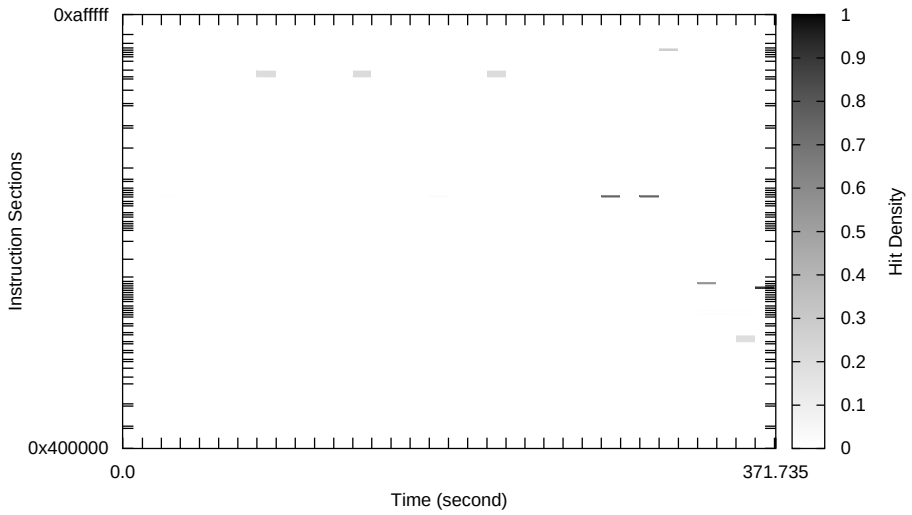

Fig. 22. Hotspot sections of 416.gamess with input set 1

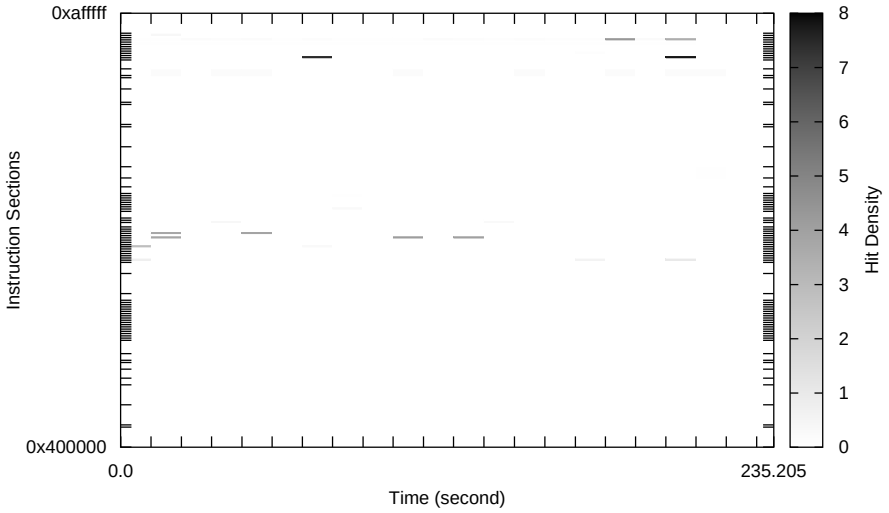

Fig. 23. Hotspot sections of 416.gamess with input set 2

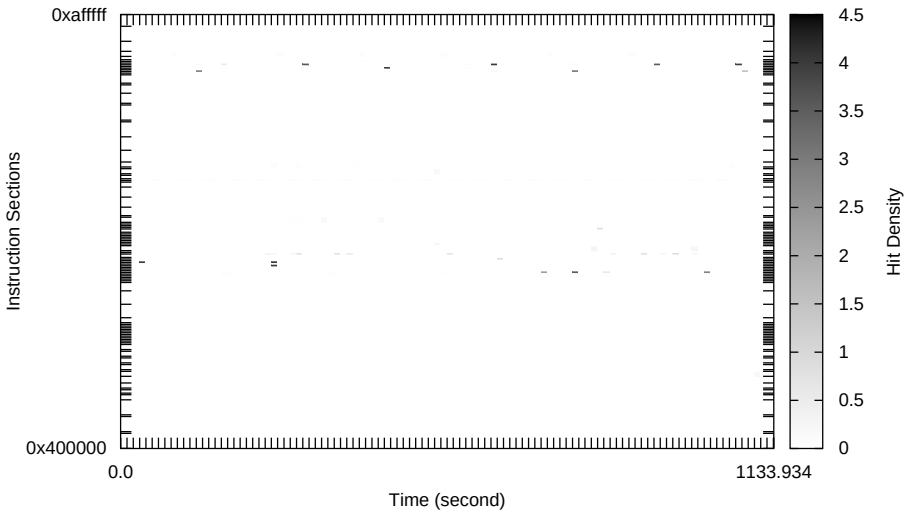

Fig. 24. Hotspot sections of 416.gamess with input set 3

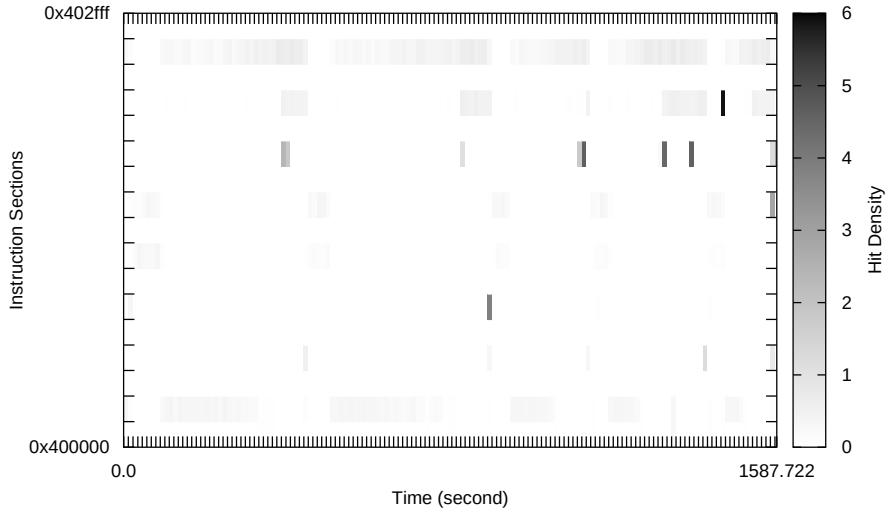

Fig. 25. Hotspot sections of 429.mcf with input set 1

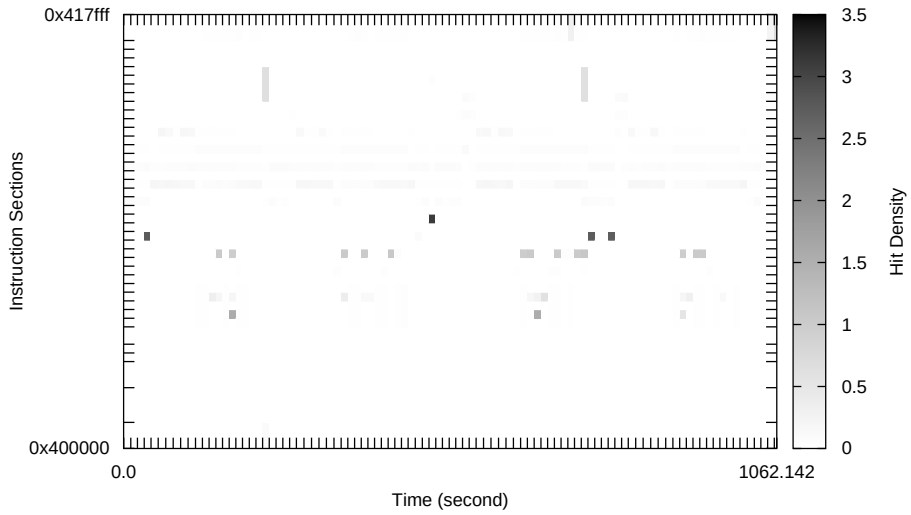

Fig. 26. Hotspot sections of 433.milc with input set 1

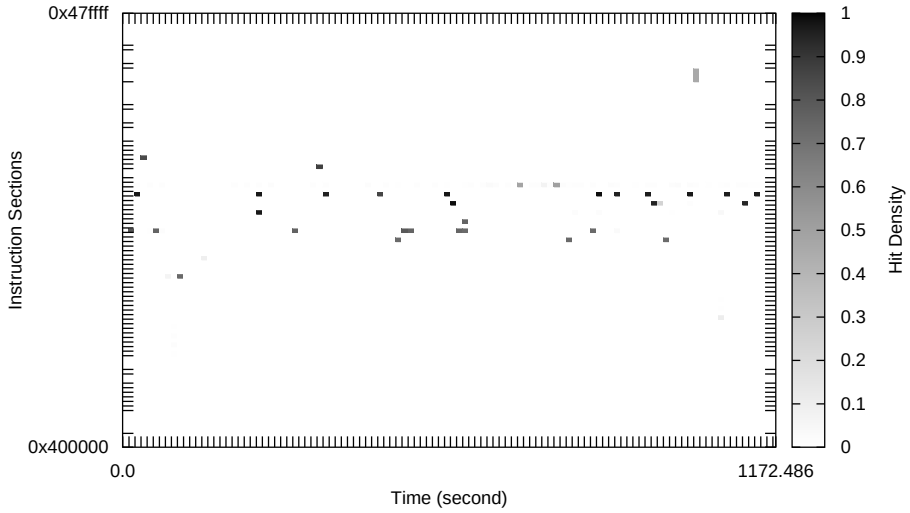

Fig. 27. Hotspot sections of 434.zeusmp with input set 1

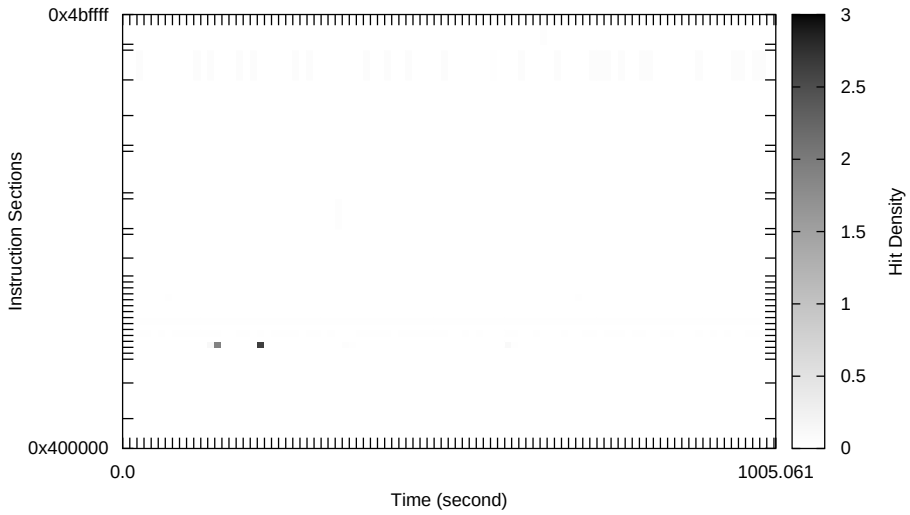

Fig. 28. Hotspot sections of 435.gromacs with input set 1

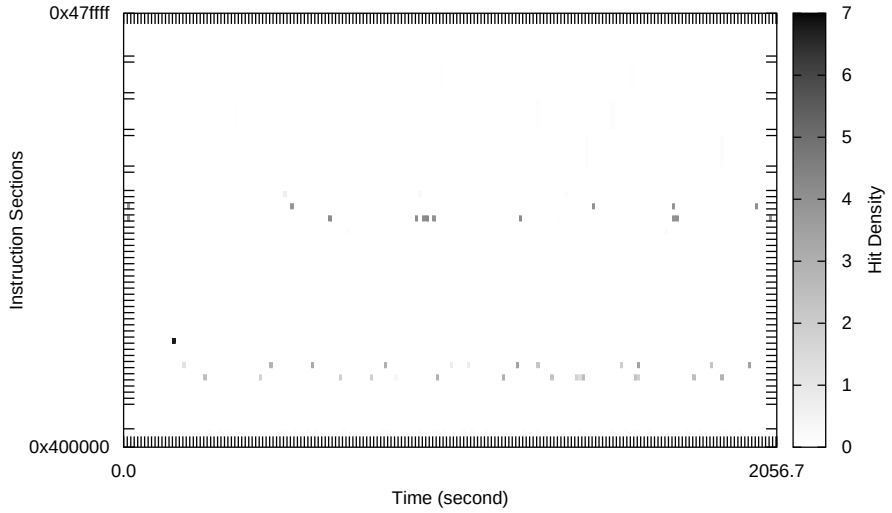

Fig. 29. Hotspot sections of 436.cactusADM with input set 1

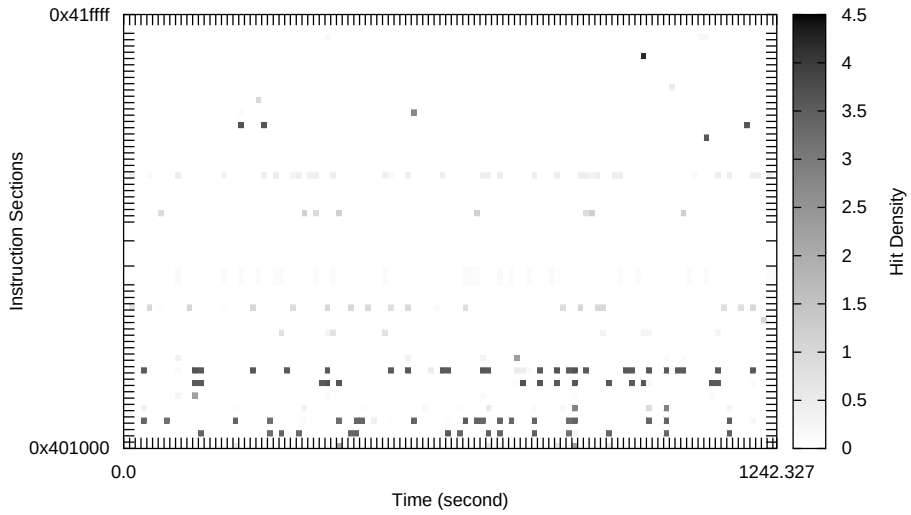

Fig. 30. Hotspot sections of 437.leslie3d with input set 1

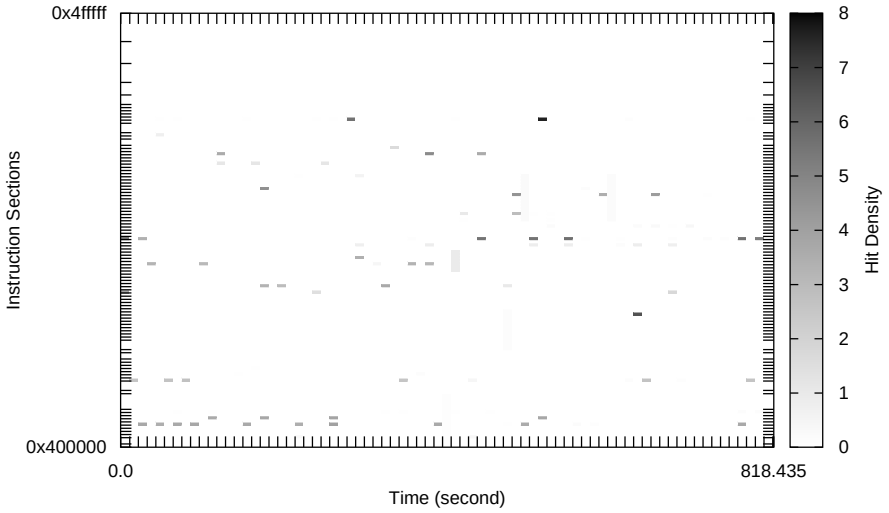

Fig. 31. Hotspot sections of 444.namd with input set 1

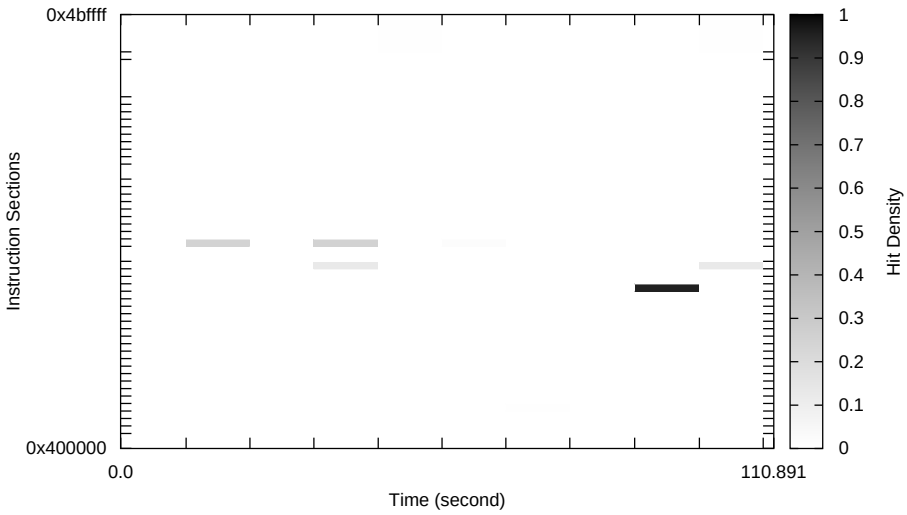

Fig. 32. Hotspot sections of 445.gobmk with input set 1

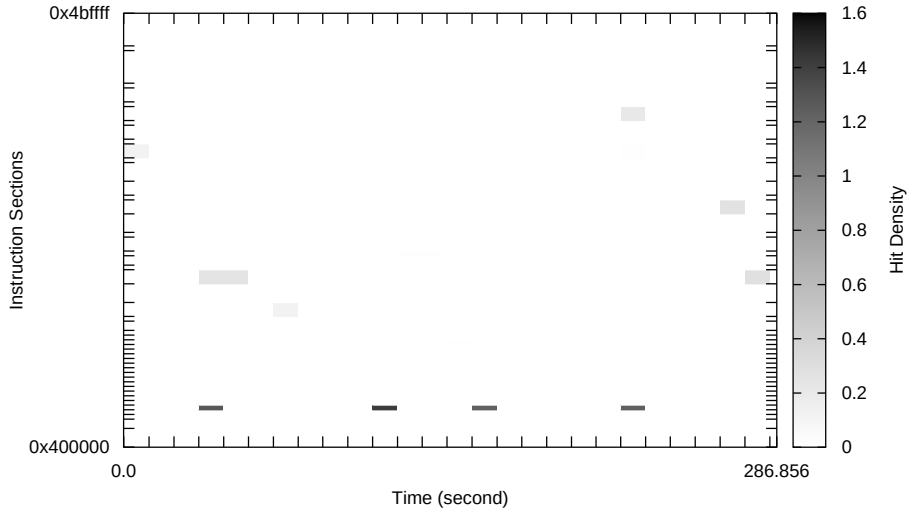

Fig. 33. Hotspot sections of 445.gobmk with input set 2

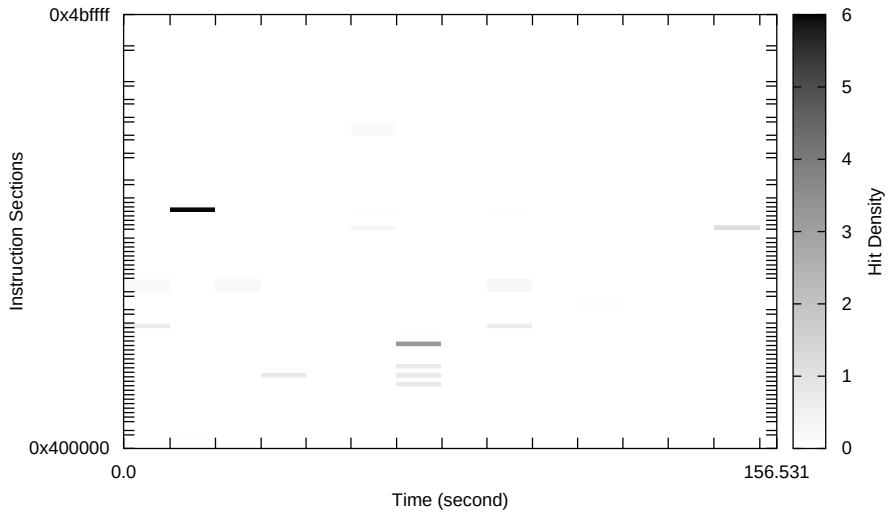

Fig. 34. Hotspot sections of 445.gobmk with input set 3

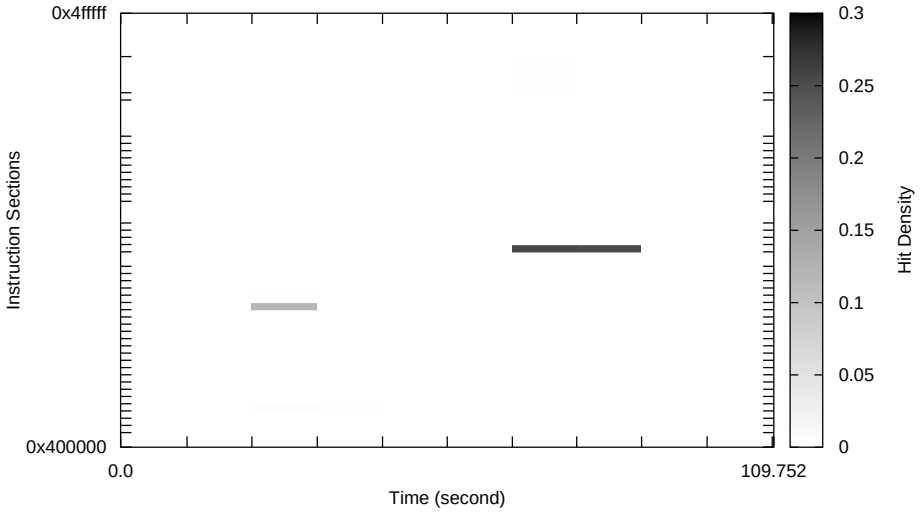

Fig. 35. Hotspot sections of 445.gobmk with input set 4

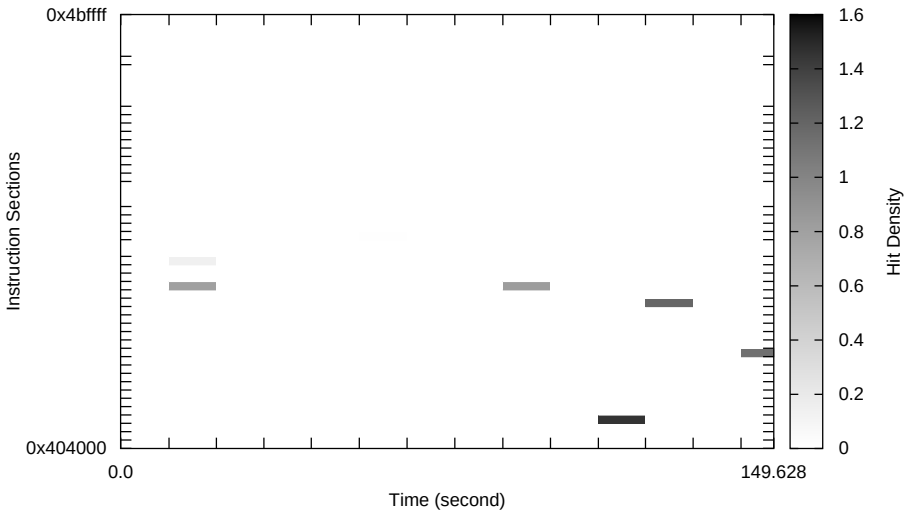

Fig. 36. Hotspot sections of 445.gobmk with input set 5

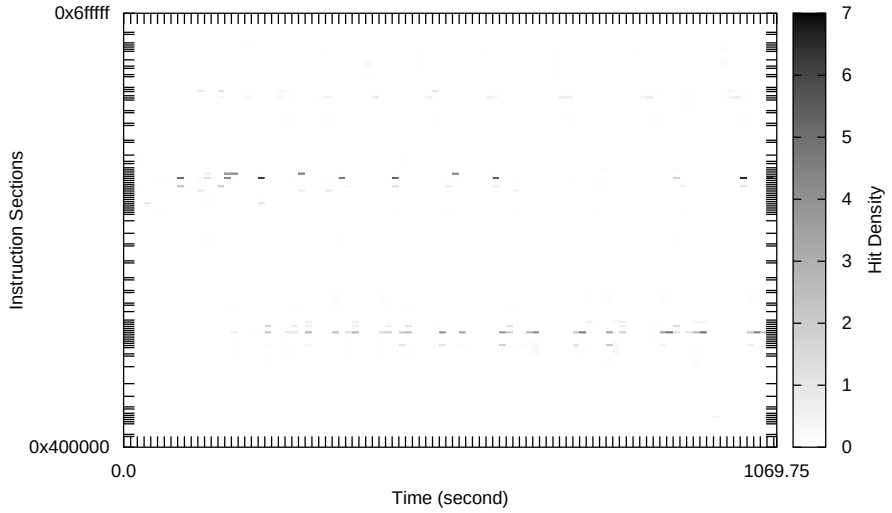

Fig. 37. Hotspot sections of 447.dealII with input set 1

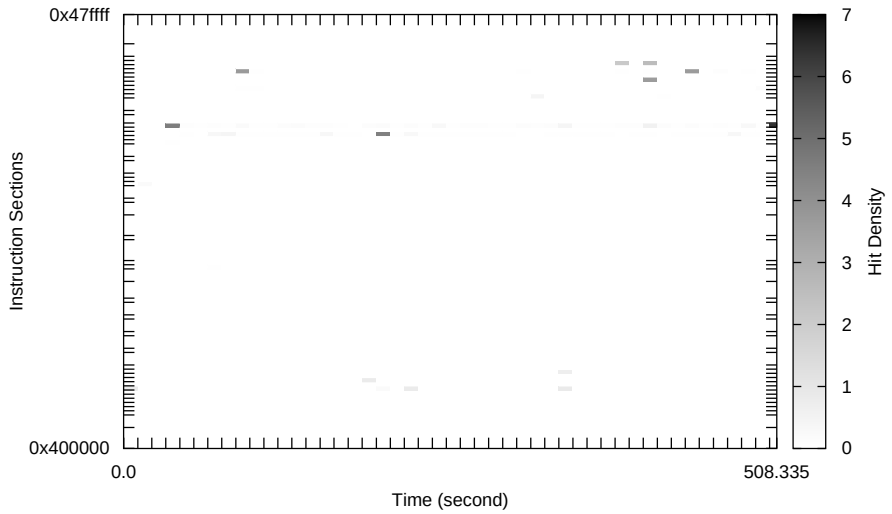

Fig. 38. Hotspot sections of 450.soplex with input set 1

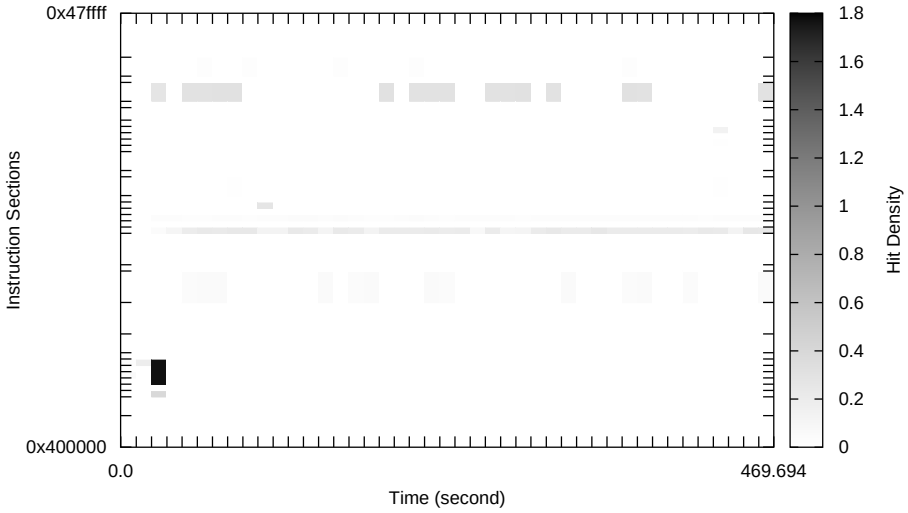

Fig. 39. Hotspot sections of 450.soplex with input set 2

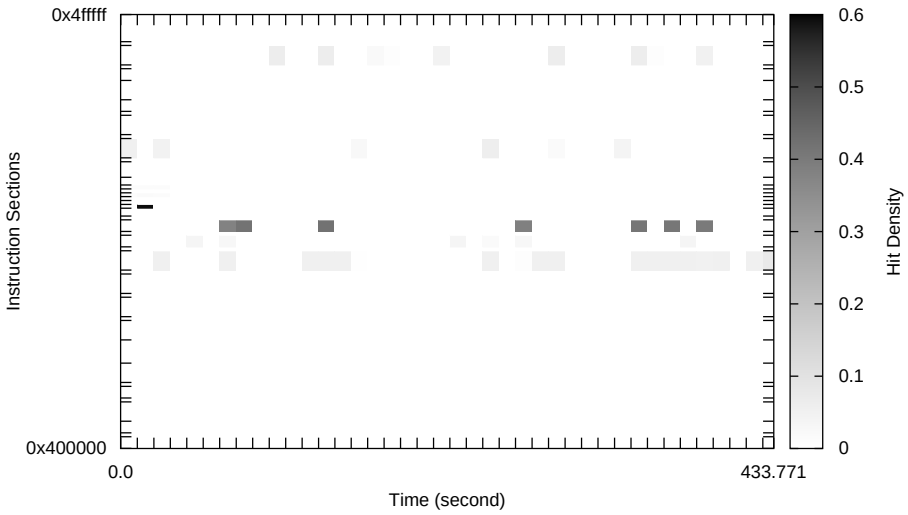

Fig. 40. Hotspot sections of 453.povray with input set 1

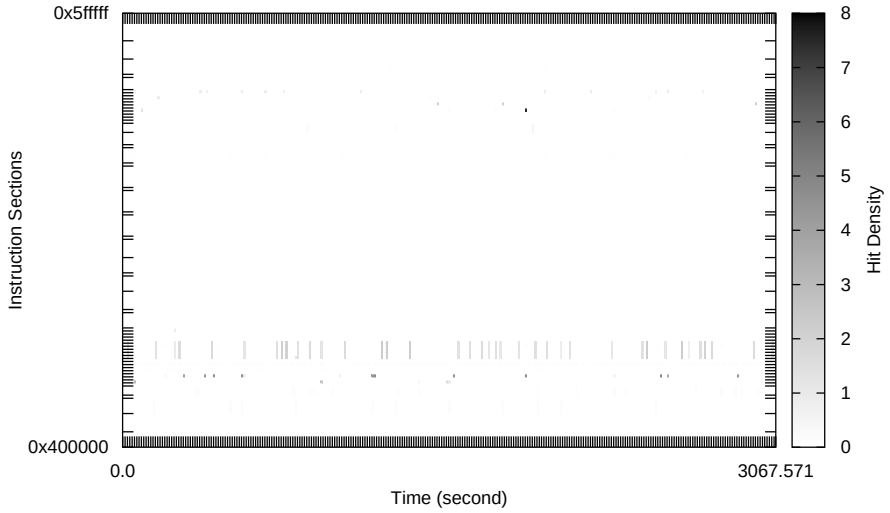

Fig. 41. Hotspot sections of 454.calculix with input set 1

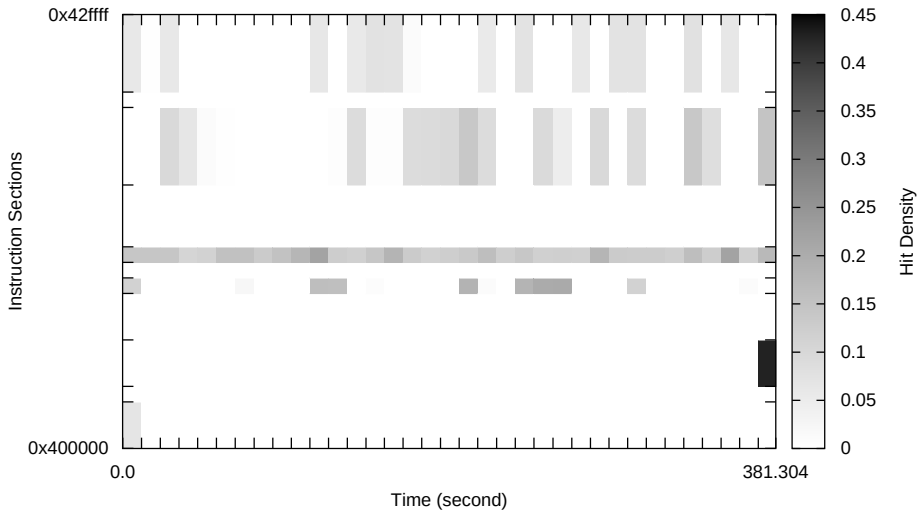

Fig. 42. Hotspot sections of 456.hmmer with input set 1

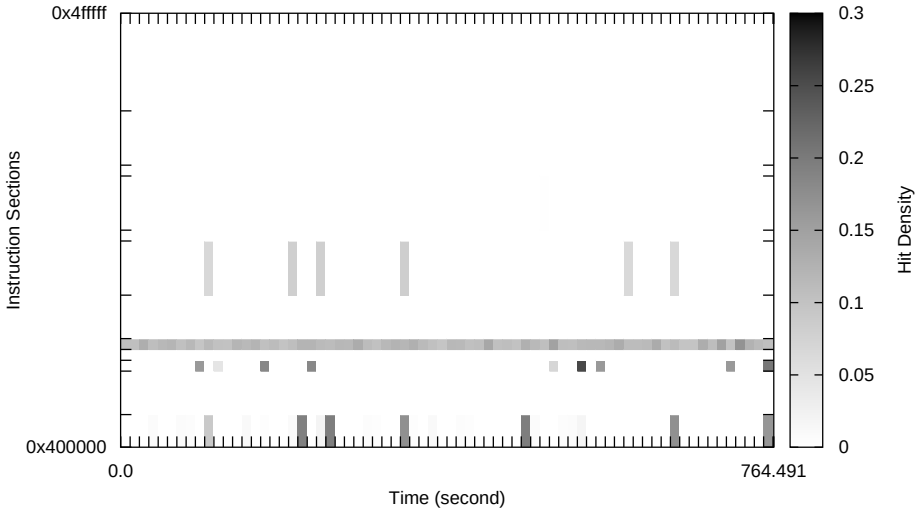

Fig. 43. Hotspot sections of 456.hmmer with input set 2

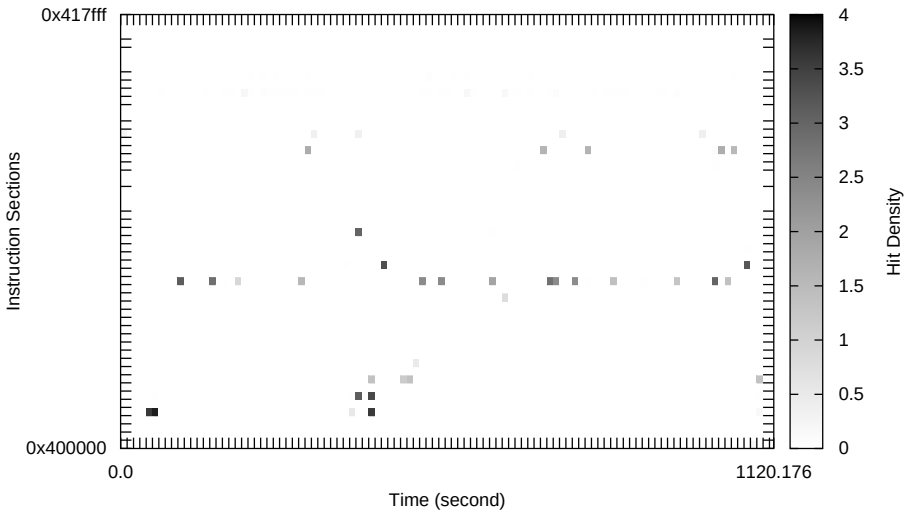

Fig. 44. Hotspot sections of 458.sjeng with input set 1

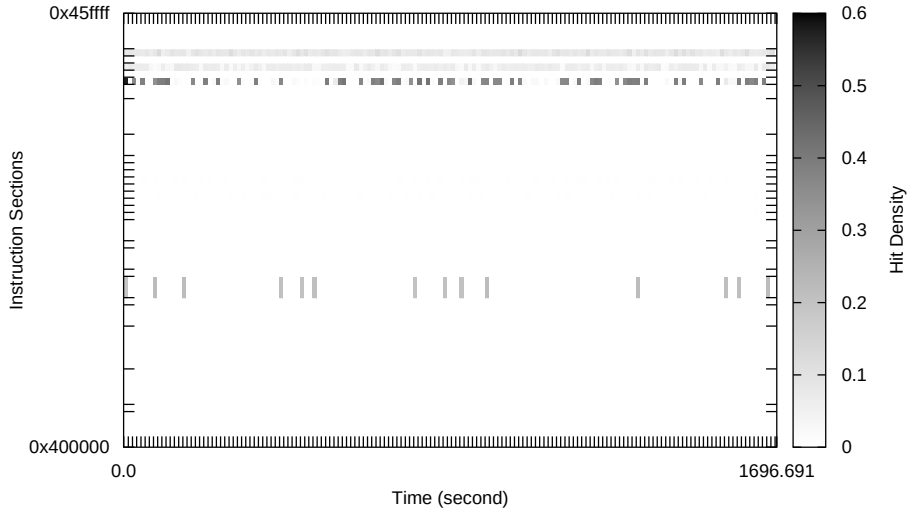

Fig. 45. Hotspot sections of 459.GemsFDTD with input set 1

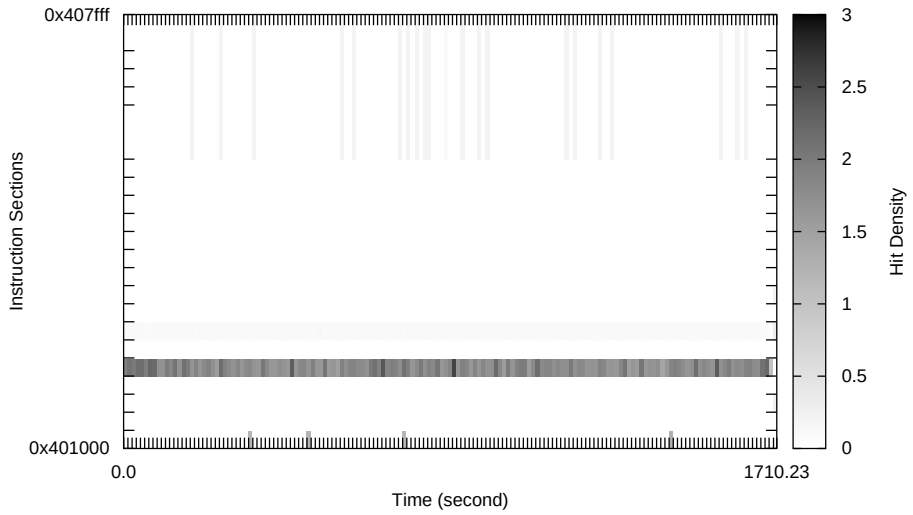

Fig. 46. Hotspot sections of 462.libquantum with input set 1

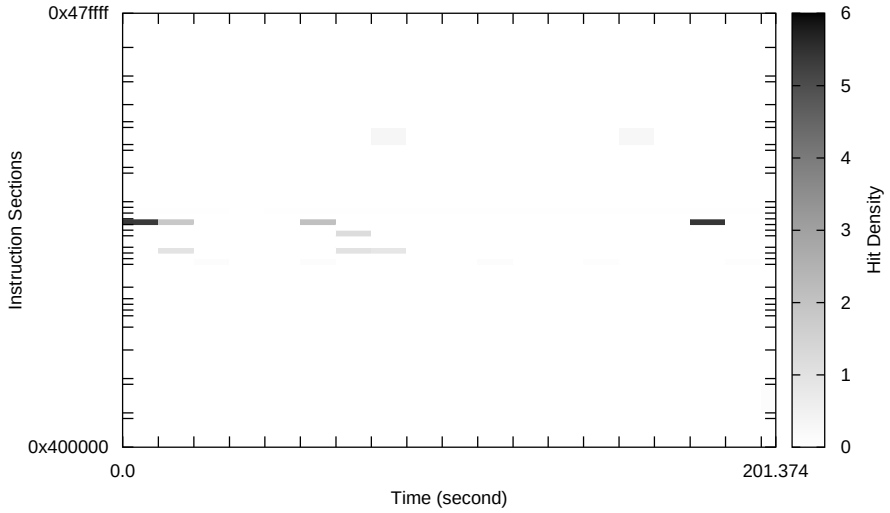

Fig. 47. Hotspot sections of 464.h264ref with input set 1

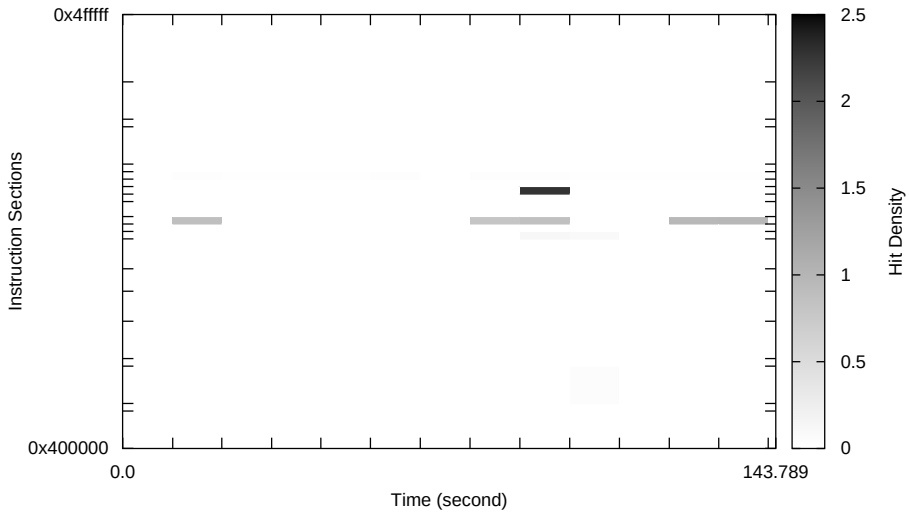

Fig. 48. Hotspot sections of 464.h264ref with input set 2

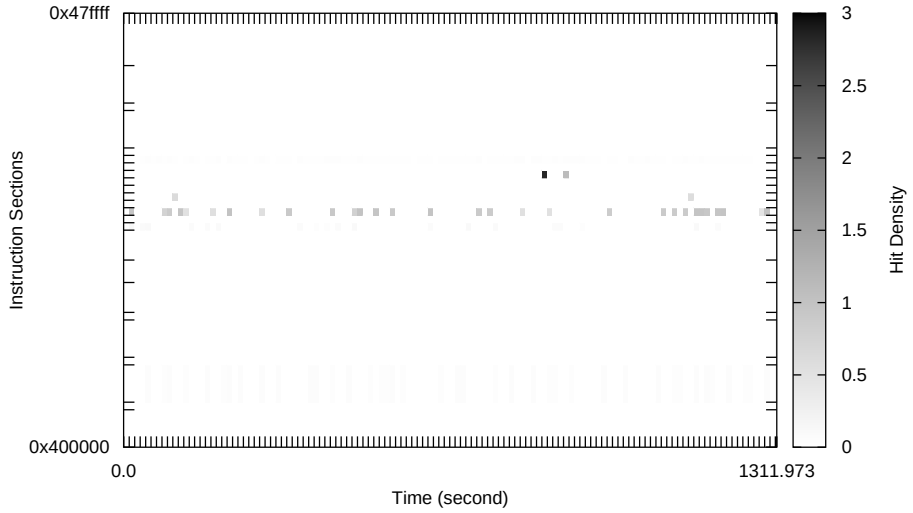

Fig. 49. Hotspot sections of 464.h264ref with input set 3

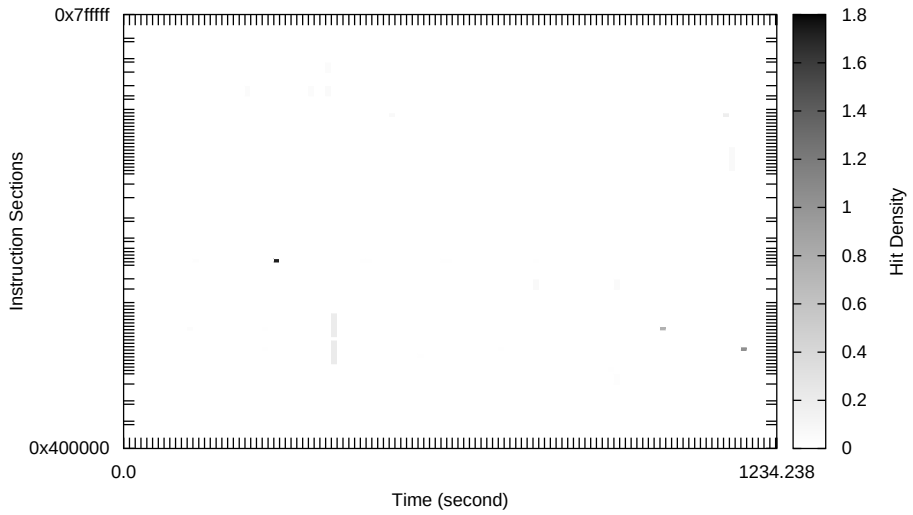

Fig. 50. Hotspot sections of 465.tonto with input set 1

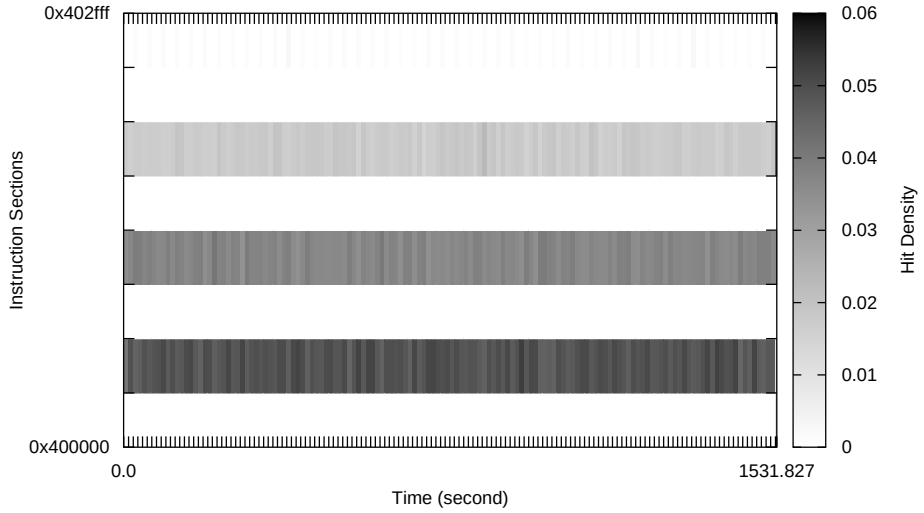

Fig. 51. Hotspot sections of 470.lbm with input set 1

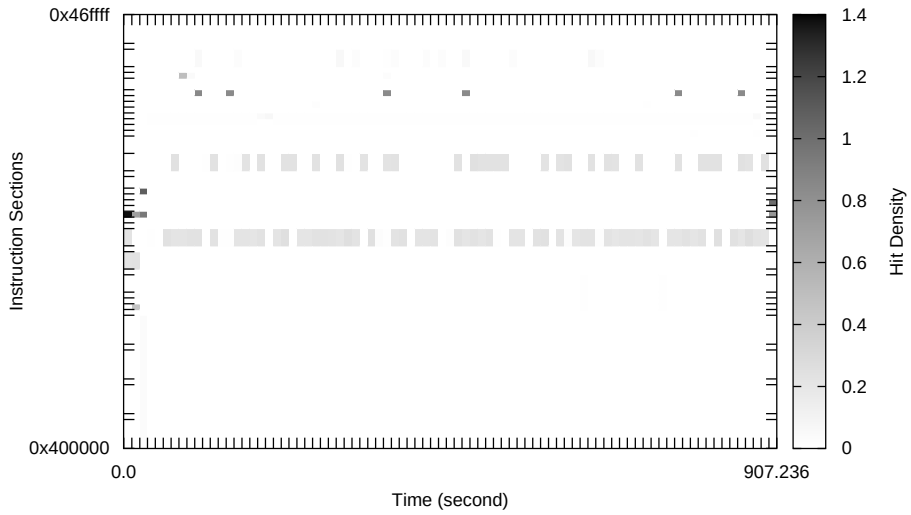

Fig. 52. Hotspot sections of 471.omnetpp with input set 1

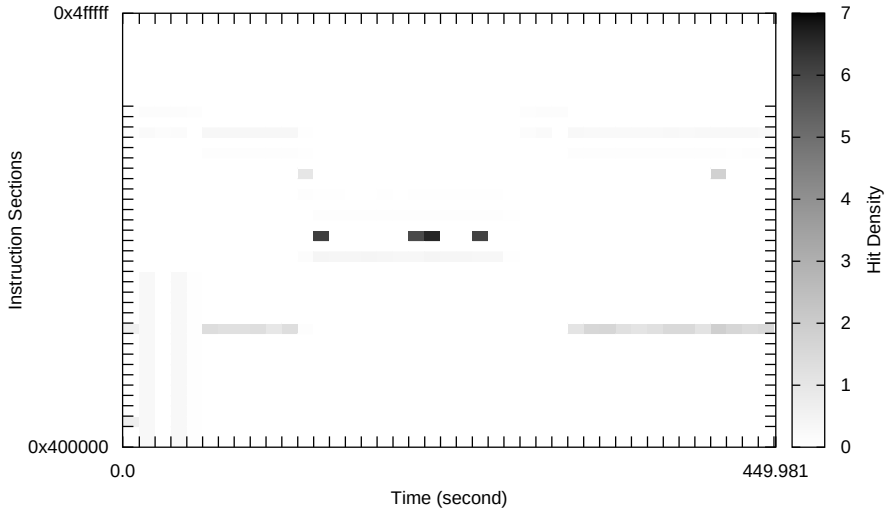

Fig. 53. Hotspot sections of 473.astar with input set 1

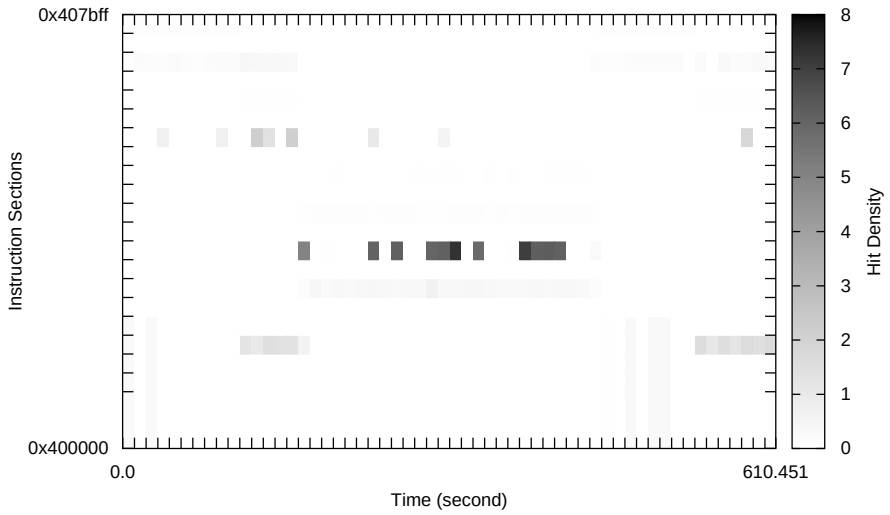

Fig. 54. Hotspot sections of 473.astar with input set 2

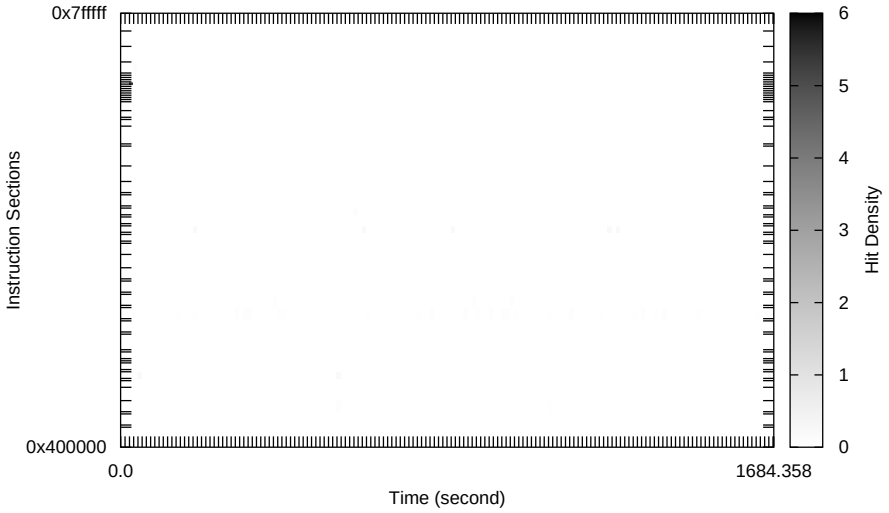

Fig. 55. Hotspot sections of 481.wrf with input set 1

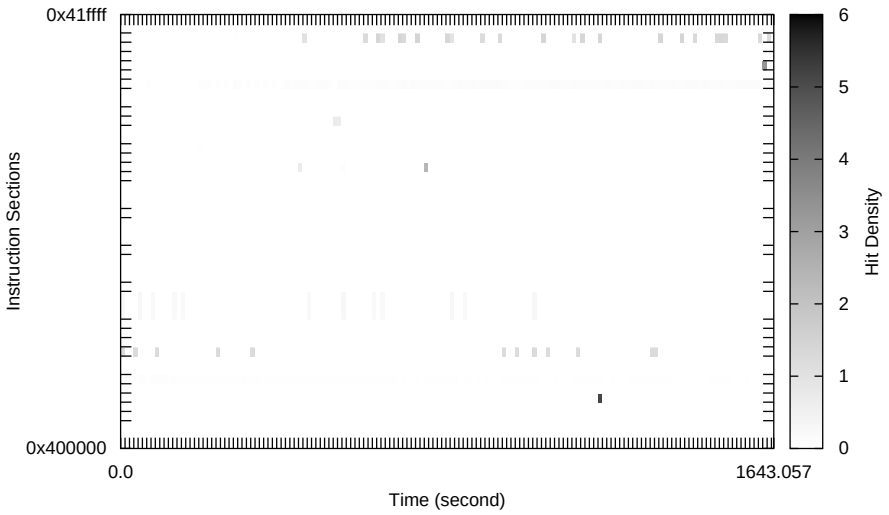

Fig. 56. Hotspot sections of 482.sphinx3 with input set 1

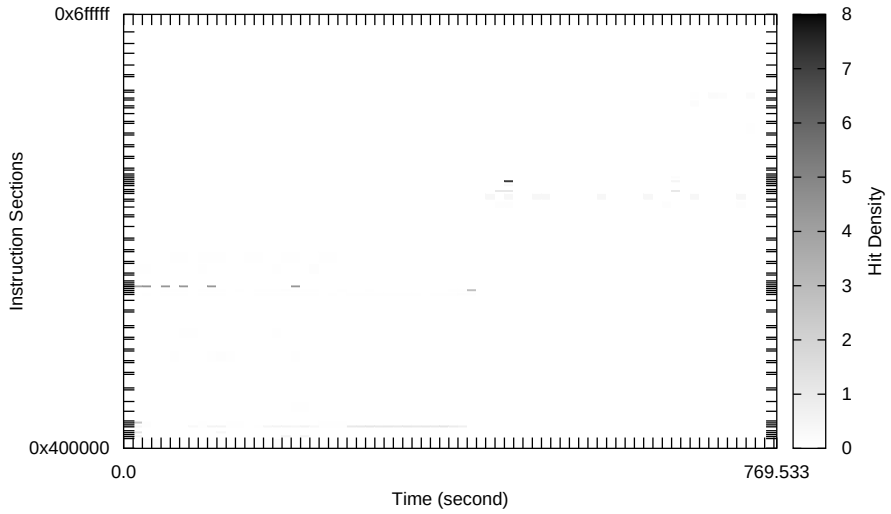

Fig. 57. Hotspot sections of 483.xalancbmk with input set 1