BioSPLASH: A sample workload for bioinformatics and computational biology for optimizing next-generation high-performance computer systems

David A. Bader* Vipin Sachdeva
Department of Electrical and Computer Engineering
University of New Mexico, Albuquerque, NM 87131

Virat Agarwal Gaurav Goel Abhishek N. Singh
Indian Institute of Technology, New Delhi

May 1, 2005

Abstract

BioSPLASH is a suite of representative applications that we have assembled from the computational biology community, where the codes are carefully selected to span a breadth of algorithms and performance characteristics. The main results of this paper are **the assembly of a scalable bioinformatics workload with impact to the DARPA High Productivity Computing Systems Program to develop revolutionarily-new economically-viable high-performance computing systems**, and **analyses of the performance of these codes for computationally demanding instances using the cycle-accurate IBM MAMBO simulator and real performance monitoring on an Apple G5 system**. Hence, our work is novel in that it is one of the first efforts to incorporate life science application performance for optimizing high-end computer system architectures.

1 Algorithms in Computational Biology

In the 50 years since the discovery of the structure of DNA, and with new techniques for sequencing the entire genome of organisms, biology is rapidly moving towards a data-intensive, computational science. Biologists are in search of biomolecular sequence data, for its comparison with other genomes, and because its structure determines function and leads to the understanding of biochemical pathways, disease prevention and cure, and the mechanisms of life itself. Computational biology has been aided by recent advances in both technology and algorithms; for instance, the ability to sequence short contiguous strings of DNA and from these reconstruct the whole genome (e.g., see [34, 2, 33]) and the proliferation of high-speed micro array, gene, and protein chips (e.g., see [27]) for the study of gene expression and function determination. These high-throughput

*Email: dbader@ece.unm.edu. This work was supported in part by DARPA Contract NBCH30390004; and NSF Grants CAREER ACI-00-93039, NSF DBI-0420513, ITR ACI-00-81404, DEB-99-10123, ITR EIA-01-21377, Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654.

techniques have led to an exponential growth of available genomic data. Algorithms for solving problems from computational biology often require parallel processing techniques due to the data- and compute-intensive nature of the computations. Many problems use polynomial time algorithms (e.g., all-to-all comparisons) but have long running times due to the large data volume to process; for example, the assembly of an entire genome or the all-to-all comparison of gene sequence data. Other problems are compute-intensive due to their inherent algorithmic complexity, such as protein folding and reconstructing evolutionary histories from molecular data, that are known to be NP-hard (or harder) and often require approximations that are also complex. Algorithms and applications in this new computational field of biology, called computational biology and bioinformatics, is now one of the biggest consumers of computational power. This is not clear; and hence, a part of our research interest. Since current architectures have not been designed with these applications in mind, we are interested in finding architectural trade-offs that can benefit computational biology without degrading performance of other workloads.

To realize this, it is increasingly important that we analyze the algorithms and applications used by the biological community.

We propose **BioSplash** with a wide variety of applications selected from computational biology and bioinformatics as part of this endeavor. The codes have been selected from a consideration of heterogeneity of algorithms, biological problems, popularity among the biological community, and memory traits, seeking the suite to be of importance to both biologists and computer scientists. To ease the portability of this suite to new architectures and systems, only freely available open-source codes are included, with no commercial software included in this suite. For each of these codes, we have assembled input datasets with varying sizes which can be used in conjunction with the applications included in this suite. We have performed an exhaustive analysis of these codes on the cycle-accurate IBM MAMBO simulator [9] and the Apple G5 (PowerPC G5) workstation. As our results show, computational biology is an extremely data-intensive science with a very high ratio of cache and memory accesses to computations. We are not only reporting the aggregate performance characteristics at the end of the run, but also "live-graph"performance data which show the variation of the performance exhibited during the execution of the application. Using such data, we correlate the performance data with the algorithm in phases of the application, and suggest optimizations targeted at separate regions of the application. We also compared these codes on dual-core systems to quantify to what extent computational biology applications can utilized and benefit from these commodity enhancements in computer architecture. We are actively using this data to propose architectural features in the design of next generation petaflop computing systems under the DARPA High Productivity Computing Systems (HPCS) Project [12]. We are actively vetting this **BioSplash** suite with the computational biology community, for instance, with a technical presentation at the Intelligent Systems of Molecular Biology (ISMB) 2005 [7].

2 Previous Work

One of the most successful attempts to create standardized benchmark suites is SPEC (Standard Performance Evaluation Corporation), which started initially as an effort to deliver better benchmarks for workstations. Over the years, SPEC evolved to cover different application classes, such as the SPECSFS for the NFS performance and the SPECWeb for performance of Web servers. Other examples of domain-specific benchmarks include transaction-processing benchmarks TPC, benchmarks for embedded processors such as the EEMBC benchmarks and many others One of the important benchmark suites in the scientific research community has been the SPLASH (Stanford Parallel Applications for Shared Memory) suite [29], later updated to SPLASH-2 [35]. SPLASH-2

included mostly codes from linear algebra and computational physics, and was designed to measure the performance of these applications on centralized and distributed memory-machines. Few comprehensive suite of computationally-intensive computational biology applications are available to the computer science community, and this works fills this needed gap.

3 BioSplash suite

Bioinformatics is a field for which problems itself have not been categorized thoroughly. Algorithms and the applications are still being studied. Many of the problems themselves are NP-hard with increasing input sets. This has led to heuristics being developed to solve the problems, which give sub-optimal results to a reasonable degree of accuracy quickly. Thus, the field is still in it's infancy with problems, algorithms, applications, and even system architecture requirements, changing everyday. The present suite of tools should therefore be treated as a starting point. As the field evolves, we expect the list of codes to evolve to encompass important emerging trends. We therefore plan to iterate over the list. Our endeavor is to finally find a representative set of bioinformatics tools which encompasses the field of bioinformatics in terms of the problems it represents and the solutions which are devised for those problems. We use this set of bioinformatics applications to drive changes in computer architecture for high-performance computing systems specifically targeted towards the computational biology applications.

Algorithms for solving problems from computational biology often require parallel processing techniques due to the data- and compute-intensive nature of the computations. However many problems in biology are integer-based using abstract data and irregular in structure making them significantly more challenging to parallelize. Also, since the sequential algorithms have been not been developed fully, their parallel solutions or implementations are themselves not available. Thus, while our imperative is to identify parallel codes, we have included uniprocessor implementations also in this suite as of now; for lack of parallel implementations in some cases. Out of the 8 codes included in this suite, 4 of these codes are multi-threaded. As the field matures, we should find increasing parallel implementations of the codes for bioinformatics sub-problems.

Most of the codes are handpicked from the following broad problems identified by the biological community of interest to computer scientist's.

- Sequence alignment – pairwise and multiple.
- Phylogeny Reconstruction.
- Protein structure prediction.
- Sequence Homology and Gene-finding.

The codes in **BioSplash** v1.0 consist of the following 8 codes along with their brief description.

- **BLAST** (blastp) – (Basic Local Alignment Search Tool) [1] suite of programs are the most widely used computational tool by the biology community. BLAST includes several programs in it's suite – *blastp* is a multithreaded implementation which is used for searching amino acid sequences against databases of amino acid sequences for identifying sequences which are homologous to the query sequence. Other programs in the suite include looking up amino acid sequences against nucleotide databases and vice-versa.
- **ClustalW_SMP** (clustalw_smp) – is a symmetric multiprocessors implementation of ClustalW – ClustalW is a widely used heuristic tool for the multiple sequence alignment problem.

ClustalW uses progressive alignments [32] which are perhaps the most practical and widely used methods. Progressive alignments compare all sequences pairwise, perform cluster analysis on the pairwise data to generate a hierarchy for alignment (guide tree), and then build the alignment step by step according to the guide tree.

- **HMMER** (hmmpfam) – HMMER, suite of programs uses profile HMM's [16], which are statistical descriptions of a sequence family's consensus to do sensitive database searching. *hmmpfam* compares one or more sequences to a database of profile hidden Markov models, such as the Pfam library, in order to identify known domains within a sequence, using either the Viterbi or the forward algorithm.

- **Glimmer** (glimmer2) – Glimmer is a widely used gene-finding tool for microbial DNA, especially for bacteria, archea, and viruses [13]. Glimmer uses interpolated Markov models (IMM's) [26] to identify the coding regions and distinguish them from non coding DNA. The program consists of essentially two steps – the first step trains the IMM from an input set of sequences, the second step uses this trained IMM for finding putative genes in the input genome. them. Glimmer's predictions are used as input to the BLAST and FASTA programs is therefore used for gene annotation.

- **GRAPPA** (grappa) – (Genome Rearrangements Analysis under Parsimony and other Phylogenetic Algorithms), a software for phylogeny reconstruction [23]. To date, almost every model of speciation and genomic evolution used in phylogenetic reconstruction has given rise to NP-hard optimization problems. GRAPPA is a reimplementation of the breakpoint analysis [8] developed by Blanchette and Sankoff, and also provides the first linear-time implementation of inversion distances improving upon Hannenhalli and Pevzner's polynomial time approach [5].Currently, GRAPPA also handles inversion phylogeny and unequal gene content.

- **Phylip** (proml) – PHYLIP is a collection of programs for inferring phylogenies [18]. Methods that are available in the package include parsimony, distance matrix, and likelihood methods [17], including bootstrapping and consensus trees. Data types that can be handled include molecular sequences, gene frequencies, restriction sites, distance matrices, and 0/1 discrete characters.

- **Predator** (predator) – [20, 21] is a tool for finding protein structures. Predator is based on the calculated propensities of every 400 amino-acid pairs [19] to interact inside an α-helix or one upon three types of β-bridges. It then incorporates non-local interaction statistics. PREDATOR also uses propensities for α-helix, β-strand and coil derived form a nearest-neighbor approach. To use information coming from homologous proteins, PREDATOR relies on local pairwise alignments for input.

4 Experimental Environment

Our experimental environment consisted of the Mambo [28] simulator from IBM and the Apple PowerMac G5 having PowerPC G5 processor. Mambo is an IBM proprietary full-system simulation tool for the PowerPC architecture and is used within IBM to model future systems, support early software development, and design new system software [28, 9]. Importantly Mambo is being used for the DARPA High Productivity Computing Systems program aimed at bringing sustained

multi-petaflop and autonomic capabilities to commercial supercomputers. Such sustained performance improvement is being made possible by research into architectures, high-end programming environments, software tools, and hardware components. Mambo shares some of its roots with the PowerPC extensions added to the SimOS [24] simulator. The system is designed for multiple configuration options. Mambo can be used to design uniprocessor or shared memory multiprocessors with various user-specific PowerPC extensions and attributes such as VMX support, hypervisor, cache geometries, segment lookaside buffers (SLBs) and translation lookaside buffers (TLBs), and thereafter collecting and dumping cache statistics at selected execution points of the program. PowerPC G5 processor is 64-bit, dual-core processor designed for symmetric multiprocessing. It has a superscalar execution core based on the IBM Power processor with up to 215 inflight instructions and a velocity engine for accelerated SIMD processing. Other features include advanced 3-bit component branch prediction logic to increase processing efficiency, and dual frontside buses for enhanced bandwidth. We used the MONster tool, part of the Apple's native graphical performance analysis tool CHUD (Computer Hardware Understanding Development) for the performance analysis [3]. MONster provides direct access to performance counters; it can collect samples at a system-wide or process-specific level and display the metrics for the collected data. Besides the extreme accuracy in their collected performance data, another advantage of both of these tools is the "live graph"capabilities of both MONster and Mambo, reporting accumulated data not only at the end of each run, but also at chosen regular sampling intervals. Such temporal data helps to visualize the performance as it varies during the execution of the code, and can be used to inspect specific intervals of the run. We compiled the codes using gcc-3.3 on the PowerMac G5, and using powerpc64-linux-gcc on the Mambo simulator, typically with default switches given by the makefiles associated with each program. No additional optimization switches were added for these architectures. Along with the selection of the codes and the experimental environment, another important consideration was the input sets for each program. For meaningful impact to the biologists, the input sets must be motivated by real world biological data, actual instances of the data that biologists collect experimentally and use with these codes. Since our performance platforms are a contemporary architecture and a simulator, this created different requirements on the size of the input datasets. Mambo entailed a sizable slowdown simulating a complex architecture (requiring 2000 seconds of wallclock time to simulate 1 second of the operation of the system under investigation), the size of the input sets had to be small. However since small runtimes create a possibility of skewed results, we decided to use extremely long-running instances of the datasets for each code. The sequences we chose in PowerMac runs are among the longest in Genbank and Swissprot, database searches were performed against complete databases such as the Genbank, Swissprot, and PFAM, databases. We therefore created a size varying set of input sets for each code, with the inputs categorized as **small**, **medium** and **large**.

PowerMac G5 was booted into a single processor for uniprocessor runs (*nvram boot-args=1*) for the uniprocessor runs, for dual processor runs the system was again booted with both processors. The multithreaded codes are run with two threads for dual core processor runs.

The following is a brief description for the codes used, their availability, their running options along with the input datasets used for the runs.

5 Synopsis of Included Software in BioSplash v0.1

Name	Domain	URL	Parallelisation
BLAST (*blastp*) [1] [11]	Heuristic-based local sequence alignment	ftp://ftp.ncbi.nlm.nih.gov/toolbox/ncbi_tools/ncbi.tar.gz	POSIX threads MPI [11]
ClustalW_SMP (clustalw)	Progressive multiple sequence alignment	http://abs.cit.nih.gov/clustalw/	POSIX threads version.[14]. MPI versions of ClustalW ([22] and [15]). POSIX threads [14]. Cache-friendly distributed version [10].
T-Coffee (tcoffee)	Progressive multiple sequence alignment	http://igs-server.cnrs-mrs.fr/~cnotred/Projects_home_page/t_coffee_home_page.html	No
HMMER (hmmpfam)	Sequence Homology	http://hmmer.wustl.edu/	Multithreaded.
Glimmer (glimmer2)	Gene finding	www.tigr.org/software/glimmer/	No
GRAPPA (grappa)	Phylogeny Tree Construction	www.cs.unm.edu/~moret/GRAPPA	Parallel version tested under linux.
Phylip (proml)	Phylogeny Tree Construction	http://evolution.gs.washington.edu/phylip.html	No
Predator (predator)	Protein Secondary Structure Prediction	ftp://ftp.ebi.ac.uk/pub/software/unix/predator/	No.

6 BioSplash programs and their Input Datasets

Code	Input Datasets
BLAST (blastp)	Search of 16 sequences each more than 5000 residues against database Swissprot
ClustalW_SMP (clustalw_smp)	File 6000.seq included with the executable was used as input. 318 sequences with average length of about 1450 residues
HMMER (hmmpfam)	Aminoacid Sequence Q89F91 of almost 8800 residues searched against the PFAM database.
GRAPPA (grappa)	12 sequences of the bluebell flower species Campanlacae.
T-Coffee (tcoffee)	1yge_1byt included extracted from the Prefab database. Consists of 50 sequences of average length 850.
Glimmer (glimmer2)	Bacteria Genome NC_004463.fna consisting of more than 9200 kilobase pairs.
Predator (predator)	5 sequences extracted from Swissprot each of almost 7500 residues.
Phylip (proml)	Input is aligned dataset of 92 cyclophilins and cyclophilin-related proteins from eukaryotes each of length 220.

Another table with the codes and their run-commands for generating the performance graphs is included in [6].

7 Livegraph and Cumulative performance analysis for BioSplash

As we said earlier, the interesting feature of Mambo and MONster tool was their ability to generate livegraphs during the run, which enables us to visualize the performance of the code as it varied during the run. Data is collected at end of every user-defined sampling interval, thus generating

tables of data for every hardware counter recorded. Such livegraphs can be used for suggesting optimizations targeted at these separate phases. e.g. *blast* and *tcoffee* livegraph of instructions per cycle with the L1 data miss rate and we can see the direct correlation, instructions per cycle increases in the same cycle as the L1 data miss rate decreases. *Hmmpfam* shows the instructions per cycle oscillating with the application showing only one phase in it's entire run. This graph agrees with the algorithm, since *hmmpfam* is a sensetive database searching tool comparing all the sequences in a database with the input sequence, searching for homologues of the input sequence.

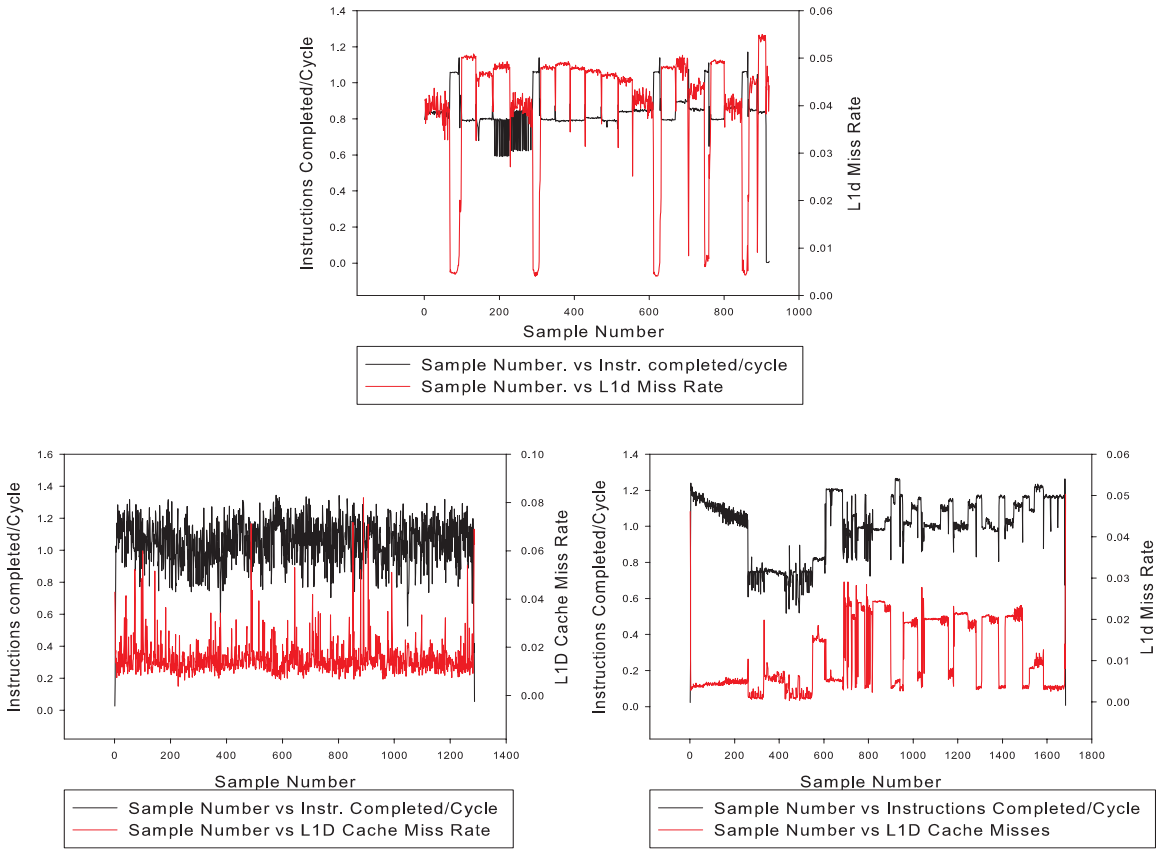

Figure 1: Blast (top), hmmpfam (bottom left), tcoffee (bottom right) performance Graphs: Instructions per cycle show a direct correlation with L1 data miss rate

In the *clustalw* livegraphs, we find an interesting observation, the instructions per cycle increases in the last phase of the application, even though L1 data miss rate increases in the same phase suggesting that performance of *clustalw* might not be as directly correlated with L1 data miss rate as we might have expected. However if we see the branch mispredicts, we find the branch mispredicts decreasing in the same phase, implying that the performance of the last phase is more closely related to branch mispredicts than L1 data miss rate.

Another useful livegraph is the data on instruction profiles, e.g. *grappa* data showing that the branch and compare instructions are the main instructions profiles in the latter part of the program, with the arithmetic instructions a negligible fraction, *blast* showing a uniform instruction profile, with the number of arithmetic, load and branch instructions in the same order, though not by a

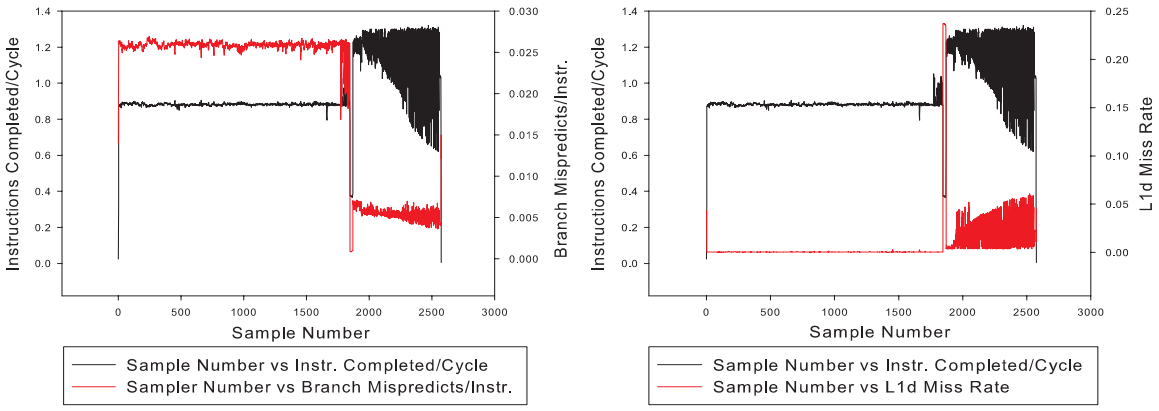

Figure 2: Clustalw Performance Graphs: Instructions per cycle show a direct correlation with branch mispredicts (left) as compared to the miss rate (right)

marked difference. *Blast* algorithm is a heuristic search method that seeks small words between the query sequence and the database sequences (load), scores the small words (arithmetic instruction) and subsequently extends the small words if they are greater than a predefined score(branch and arithmetic instruction), which explains the instruction profile.. A significant part of our research

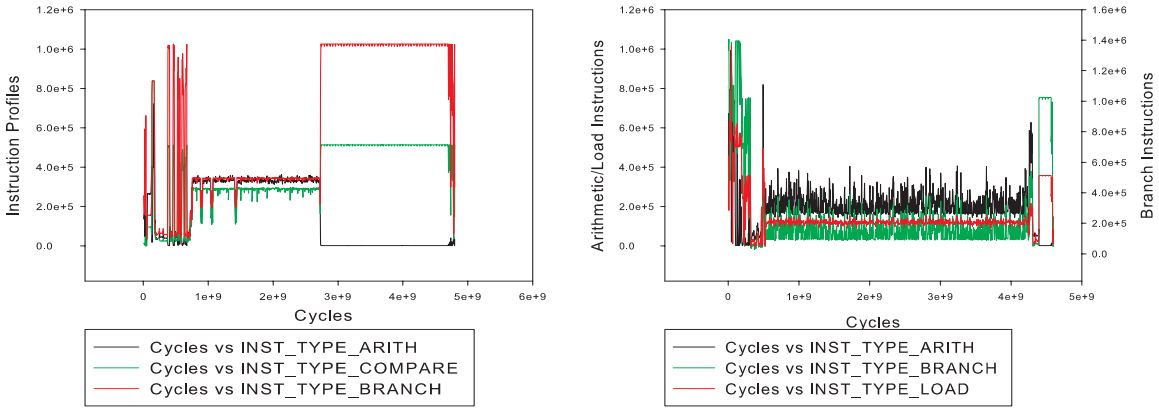

Figure 3: Grappa (left) and blast (right) graphs of instruction profiles

is correlating the performance metrics at the machine level with the higher-level algorithm the phase is working on. Data can be separated for phases of the application, and could be looked into individually with no relation to the other phases. This step is done with inserting Mambo API's into the source code of the application, collecting data separately for every phase and then resetting the performance metrics after every phase, before recording data for the next phase. We illustrate this technique with *clustalw*. Clustalw's performance can basically be categorized into 3 regions: the first phase in which every sequence is compared against every other sequence by Smith Waterman [30], a quadratic time complexity dynamic programming algorithm. The next step is the neighbor joining method [25] in which comparison score of sequences is used to make a guide tree with the sequences at the leaves of the tree. In the last step, the sequences are combined into

a multiple sequence alignment according to the guide tree [31]. *Clustalw* livegraph plotted for the first and the third phase shows an order of magnitude higher arithmetic instructions compared to the second phase, denoting the higher time complexity of the algorithm in phase one and three. L1 data cache misses are higher in the second phase, due to unpredictable access pattern; the sequences with the lowest distance are joined together in the guide tree, with the other sequences recomputing distance to the joined sequence. The heuristic is repeated, until all the sequences are joined together into a guide tree formation.

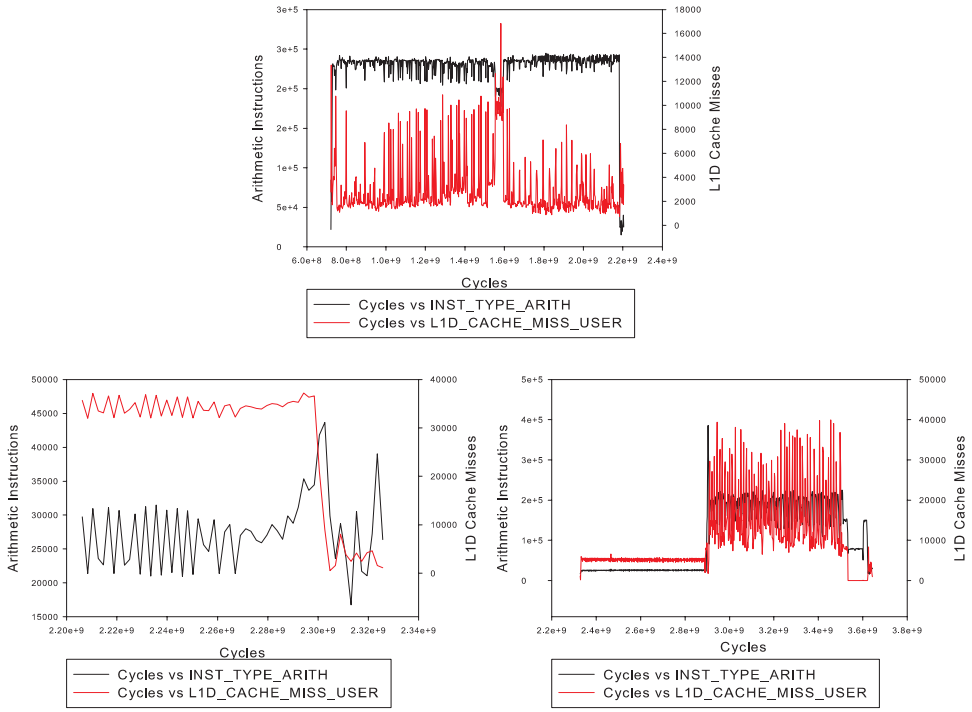

Figure 4: ClustalW region I (top), II (bottom left) and III (bottom right) showing differences in algorithmic complexity and memory access pattern

We are not able to include all the livegraphs in this document due to space constraints, fully detailed analysis and all the livegraphs are included in [6]. For cumulative performance analysis, we decided to use the data generated by the MONster tool, since the PowerPC G5 codes were run for significantly larger input data sets. We have included the list of hardware counters available in PowerPC G5 in the instruction and memory level analysis that we have collected for each code. We performed several runs for each code, due to conflict of performance counters for collecting data.

- Instruction-level analysis: Instructions dispatched, instructions completed(including and excluding IO and load/store), branch mispredicts due to condition register value and target address predict were recorded as part of this analysis.

- Memory-level analysis: L1 cand L2 cache loads and stores, L1 and L2 cache load and store misses, TLB and SLB misses, read/write request bytes, number of memory transactions and Load Miss Queue(LMQ) full events were included in this analysis.

Application	IPC	Loads/Instr.	Stores/Instr.	Branch mispredicts/1000 instr.	TLB misses/100k cycles	L1d Hit Rate	L2d Hit Rate	% of io,ld,st instr.
blastp	0.835	0.264	0.088	21.95	5.83	95.93%	92.32%	7.345%
clustalw	0.952	0.574	0.041	19.74	1.47	99.24%	97.50%	3.71%
glimmer	0.688	0.242	0.114	8.53	6.63	95.48%	88.47%	7.299%
grappa	1.012	0.349	0.234	28.84	212.96	99.88%	96.78%	15.80%
hmmpfam	1.052	0.623	0.104	9.69	4.65	98.46%	97.96%	7.14%
predator	0.902	0.664	0.062	10.90	16.92	97.87%	90.63%	3.06%
proml	0.924	0.641	0.079	1.59	2.34	97.68%	99.89%	0.086%
tcoffee	1.007	0.570	0.089	4.05	5.76	98.90%	94.26%	1.592%

Table 4: Summary of critical statistics for each code based on their Apple PowerPC G5 runs.

Table 7 gives a summary of performance metrics for each code running on an Apple G5 system. The instructions per cycle of BioSplash ranges from 0.688 (*glimmer*) to 1.012 (*grappa*, with a mean of about 0.90 for all the codes. The common characterstics of these workloads is the high ratio of data accesses to computation. In general, the number of loads and stores on a per-instruction basis ranges from a low of 0.35 to a high of 0.72 with a mean of 0.59. The high ratio of loads/stores doesn't affect the performance of most codes; many applications with high instructions per cycle also have a high percentage of loads/stores for instance *hmmpfam*, *tcoffee*, *clustalw*, *proml* all have instructions per cycle greater than 0.90, yet also have ratio of loads to instructions to be more than 50%, on the opposite side, *blast* and *glimmer* have lower instructions per cycle (0.835 and 1.012) despite the lower percentage of loads per instruction (0.264 and 0.349). The PowerPC G5 has 2 load/store units [4], which is fairly typical of modern-day processors, hence it is expected that the high percentage of loads/stores will not lead to reduced performance on other class of families also. The high instructions per cycle despite higher loads and stores can be partially explained by the high L1 data hit rate of most codes; it also explains the lower performance of *blast* and *glimmer* with their hit rates among the lowest of all codes (almost 96%). Infact *glimmer* has the lowest L1 and L2 hit rates which explains to a large extent, it's lowest count of instructions per cycle. *Grappa* has the second highest instructions per cycle count, despite having TLB misses of almost 2 orders of magnitude compared to other codes, and the highest instruction percentage for load/store/io. Based on this data, and other performance counters collected during the runs, we find that it is difficult to pinpoint the performance of these codes on a single system parameter, but are rather dependant on an interesting interplay of system parameters both at the memory and the instruction level. This is because the bioinformatics applications are very heterogenous in nature, with wide ranging differences in problems, algorithms and their implementations.

8 Dual-core runs of BioSplash parallel codes

Besides sequential runs, we also performed dual core runs of the parallel BioSplash codes; *blast*, *clustalw* and *hmmpfam* showed almost linear speedup using both processors of the dual-core Apple PowerPC G5. *Grappa* showed a small speedup using both processors as the only a part of grappa's algorithm was parallelized. ClustalW_SMP was also not completely parallelized, but since the initial part of comparing every sequence against every other sequence was the major time-consuming

speedup,*clustalw* showed an impressive speedup. This shows that parallel multithreaded bioinformatics codes stand to benefit from dual-core processors. Below is the livegraphs of *blast* and *grappa* with data for both processors.

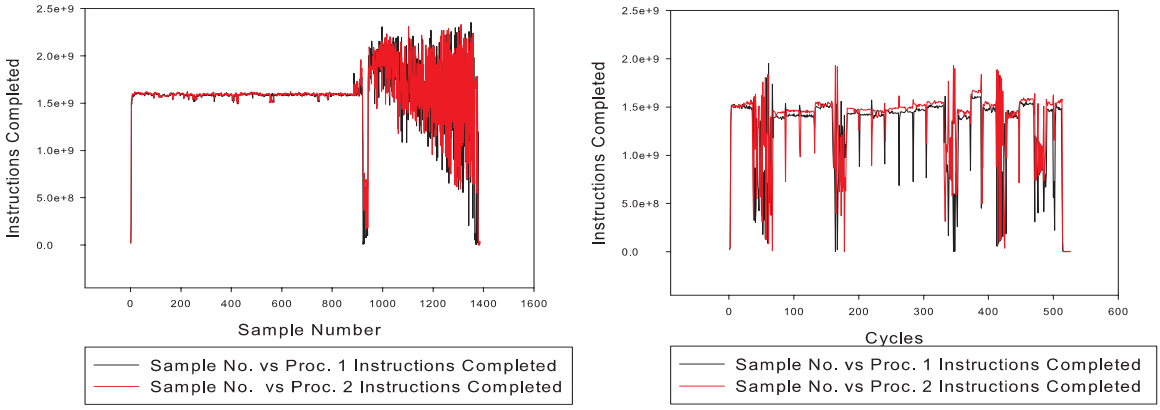

Figure 5: Clustalw (left) and blast (right) graphs of parallel runs

9 Acknowledgments

We would like to thank Ram Rajamony of IBM Austin Research Laboratories for his help with the Mambo simulator and the CHUD performance tools and performance metrics.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Molecular Biology*, 215:403–410, 1990.

[2] E. Anson and E.W. Myers. Algorithms for whole genome shotgun sequencing. In *Proc. 3rd Ann. Int'l Conf. on Computational Molecular Biology (RECOMB99)*, Lyon, France, April 1999. ACM.

[3] Apple Computer, Inc. C.H.U.D. performance tools. `http://developer.apple.com/performance/`.

[4] Apple Computer, Inc. *PowerPC G5: White Paper*, June 2004. `images.apple.com/powermac/pdf/PowerPCG5_WP_06092004.pdf`.

[5] D.A. Bader, B.M.E. Moret, and M. Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. In *Proc. 7th Int'l Workshop on Algorithms and Data Structures (WADS 2001)*, volume 2125 of *Lecture Notes in Computer Science*, pages 365–376, Providence, RI, 2001. Springer-Verlag.

[6] D.A. Bader, V. Sachdeva, V. Agarwal, G. Goel, and A.N. Singh. BioSPLASH: A sample workload for bioinformatics and computational biology for optimizing next-generation high-performance computer systems. Technical report, University of New Mexico, Albuquerque, NM, April 2005.

[7] D.A. Bader, V. Sachdeva, A. Trehan, V. Agarwal, G. Gupta, and A.N. Singh. BioSPLASH: A sample workload from bioinformatics and computational biology for optimizing next-generation high-performance computer systems. In *Proc. 13th Int'l Conf. on Intel. Sys. for Mol. Bio. (ISMB 2005)*, Detroit, MI, June 2005. Poster Session.

[8] M. Blanchette, G. Bourque, and D. Sankoff. Breakpoint phylogenies. In S. Miyano and T. Takagi, editors, *Genome Informatics*, pages 25–34. University Academy Press, Tokyo, Japan, 1997.

[9] P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang. Mambo – a full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, 2004.

[10] U. Catalyurek, E. Stahlberg, R. Ferreira, T. Kurc, and J. Saltz. Improving performance of multiple sequence alignment analysis in multi-client environments. In *Proc. 1st Workshop on High Performance Computational Biology (HiCOMB 2002)*, Fort Lauderdale, FL, April 2002.

[11] A.E. Darling, L. Carey, and W. Feng. The design, implementation and performance of mpiBLAST. In *Proc. ClusterWorld Conf. and Expo. 2003*, San Jose, CA, June 2003.

[12] Defense Advanced Research Projects Agency (DARPA). High productivity computing systems program. `http://www.darpa.mil/ipto/programs/hpcs/`.

[13] A.L. Delcher, D. Harmon, S. Kasif, O. White, and S.L. Salzberg. Improved microbial gene identification with GLIMMER. *Nucleic Acids Res.*, 27(23):4636–4641, 1998.

[14] O. Duzlevski. SMP version of ClustalW 1.82. `http://bioinfo.pbi.nrc.ca/clustalw-smp/`, 2002.

[15] J. Ebedes and A. Datta. Multiple sequence alignment in parallel on a workstation cluster. *Bioinformatics*, 20(7):1193–1195, 2004.

[16] S. R. Eddy. Profile hidden Markov models. *Bioinformatics*, 25:755–763, 1998.

[17] J. Felsenstein. Evolutionary trees from DNA sequences: a maximum likelihood approach. *J. Mol. Evol.*, 17:368–376, 1981.

[18] J. Felsenstein. PHYLIP – phylogeny inference package (version 3.2). *Cladistics*, 5:164–166, 1989.

[19] D. Frishman and P. Argos. Knowledge-based secondary structure assignment. *Proteins: structure, function and genetics*, 23:566–579, 1995.

[20] D. Frishman and P. Argos. Incorporation of long-distance interactions into a secondary structure prediction algorithm. *Protein Engineering*, 9:133–142, 1996.

[21] D. Frishman and P. Argos. 75% accuracy in protein secondary structure prediction. *Proteins*, 27:329–335, 1997.

[22] K.B. Li. ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics*, 19(12):1585–1586, 2003.

[23] B. M.E. Moret, D.A. Bader, T. Warnow, S.K. Wyman, and M. Yan. GRAPPA: a high-performance computational tool for phylogeny reconstruction from gene-order data. In *Proc. Botany*, Albuquerque, NM, August 2001.

[24] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology: Systems & Technology*, 3(4):34–43, 1995.

[25] N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstruction of phylogenetic trees. *Molecular Biological and Evolution*, 4:406–425, 1987.

[26] S. Salzberg, A. Delcher, S. Kasif, and O. White. Microbial gene identification using interpolated Markov models. *Nucleic Acids Res.*, 26(2):544–548, 1998.

[27] M. Schena, D. Shalon, R.W. Davis, and P.O. Brown. Quantitative monitoring of gene expression patterns with a complementary DNA microarray. *Science*, 270(5235):467–470, 1995.

[28] H. Shafi, P. Bohrer, J. Phelan, C. Rusu, and J. Peterson. Design and validation of a performance and power simulator for PowerPC systems. *IBM J. Research and Development*, 47(5/6), 2003.

[29] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, 1992.

[30] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Molecular Biology*, 147:195–197, 1981.

[31] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTALW: improving the senstivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res.*, 22:4673–4680, 1994.

[32] J. D. Thompson, F. Plewniak, and O. Poch. A comprehensive comparison of multiple sequence alignment programs. *Nucleic Acids Res.*, 27:2682–2690, 1999.

[33] J.C. Venter and *et al.* The sequence of the human genome. *Science*, 291(5507):1304–1351, 2001.

[34] J.L. Weber and E.W. Myers. Human whole-genome shotgun sequencing. *Genome Research*, 7(5):401–409, 1997.

[35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, pages 24–36, June 1995.

A BioSplash program options used running on Mambo

Code	Summary	Running Options
BLAST *(blastp)*	blastp searches for the homologues of an input amino acid sequence against a database of amino acid	*./blastall -p blastp -i Drosoph.txt -d Drosoph/drosoph.aa -o outdd -p* *-i* input query file *-d* Database *-o* Output file
FASTA (*ssearch*)	ssearch does an exact Smith-Watermann of an input sequence with every sequence of an input library library printing the results	*./ssearch34_t -a -b 20 -q -O <Output Alignment File> < Input Sequence > < Input Library File >* *-a* Show entire length in alignment *-b* Number of high scores to display *-q* Quiet
ClustalW_SMP	Clustalw makes a multiple sequence alignment of the unaligned sequences given	*./clustalw < Unaligned sequences file >*
T-Coffee	T-Coffee makes a multiple sequence alignment of unaligned sequences	*./tcoffee < Input sequences file > -dp_mode = myers_miller_pair_wise -in = lalign_id_pair, clustalw_pair -tree_mode = slow* *-dp_mode* = Dynamic programming mode is Myers and Miller, linear space and quadratic time complexity *-in* = methods used for library making, lalign_id_pair is the local alignment using FASTA function, clustalw_pair is the global alignment using the Smith-Watermann. *tree_mode* = slow, similarity matrix construction done using dynamic programming mode.
HMMER (*hmmbuild*)	hmmbuild makes a profile hidden markov model from aligned sequences	*./hmmbuild <Output HMM file > < Input aligned sequences file >*
(*hmmpfam*)	hmmpfam searches for a sequence in a database of profile HMM's	*./hmmpfam < HMM database > < Input sequence >*
Glimmer	Finds genes in microbial DNA especially bacteria and archae	*./run-glimmer2 < genome file >*
Grappa	Tool for phylogeny reconstruction	*./grappa -f < Input file > -o < Output file > -m* *-m Tighten circular lower bound*
Phylip (*ProML*)	Protein Maximum Likelihood Program	*./proml < scriptproml > < output >*
(*ProMLK*)	Protein Maximum Likelihood Program with molecular clock	*./proml < scriptproml > < output >*
Predator	Predicts the 3D structure of a protein taking an amino acid sequence as input	*./predator < Input Sequence > -u -h -a -f< Ouput file >*

		-u Do not copy assignment directly from the PDB database if query sequence is found in PDB *-h Indicate progress by dots* *-a Make prediction for all sequences in input file*

B Mambo LiveGraphs

B.1 BLAST

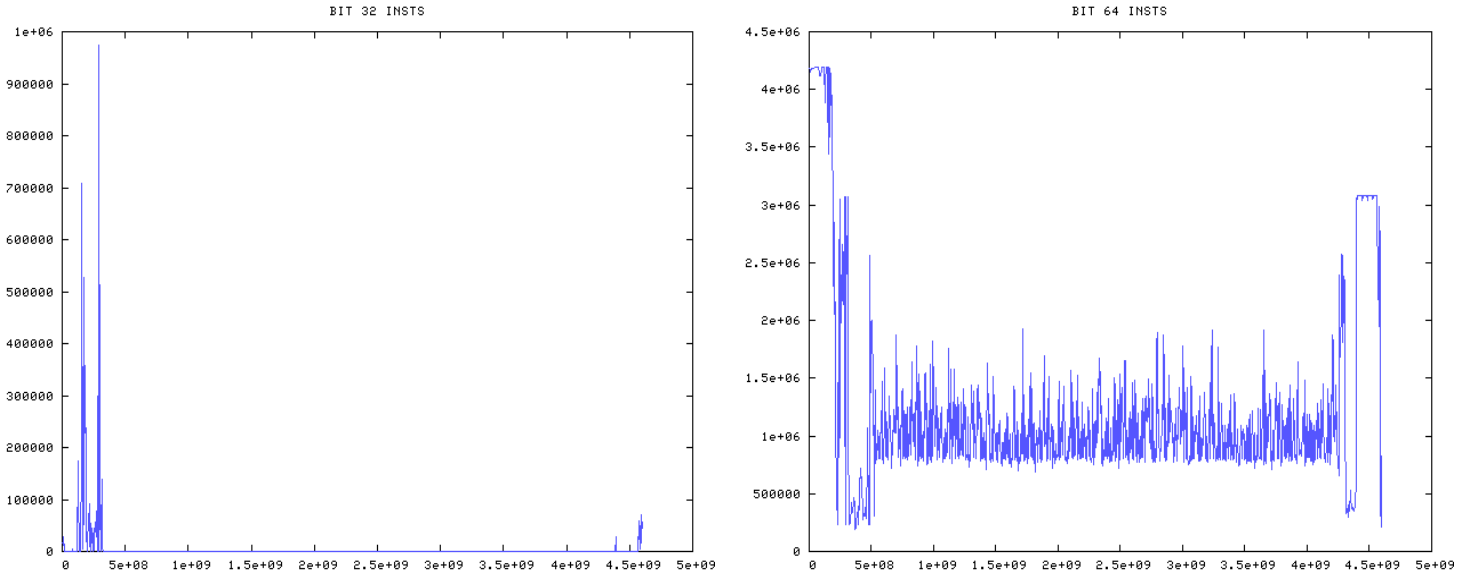

Figure 6: 32-bit (left) and 64-bit (right) instruction livegraphs

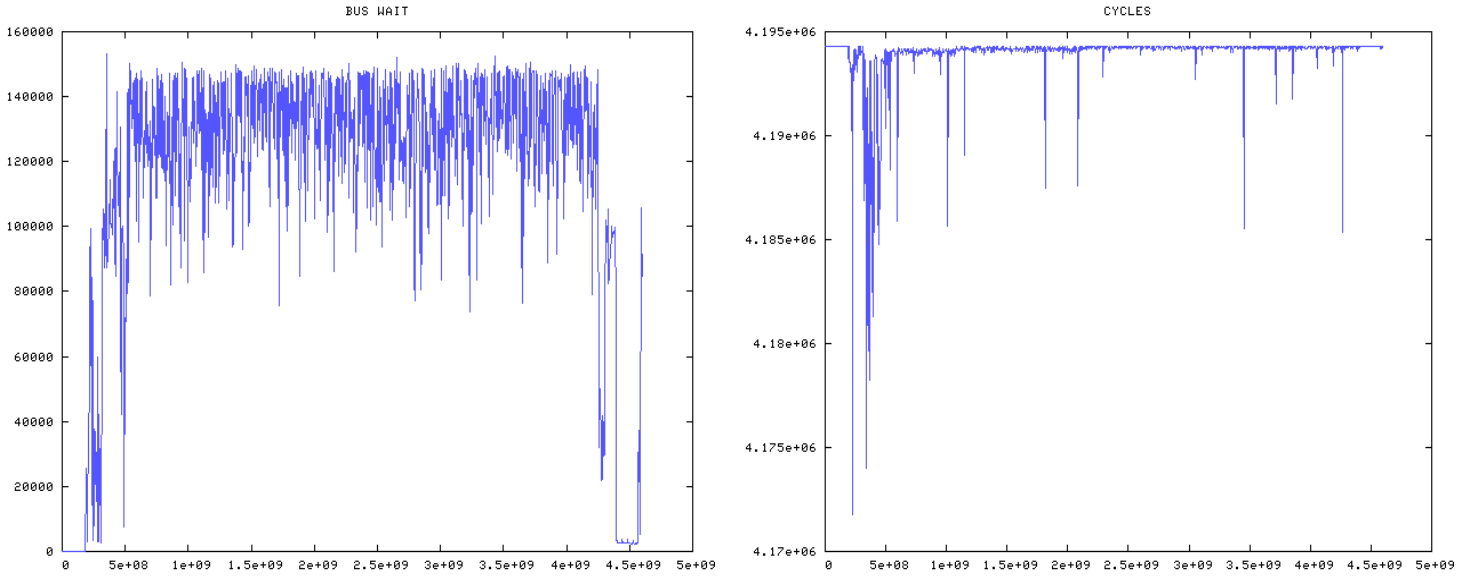

Figure 7: Bus-wait (left) and Cycles (right) livegraphs

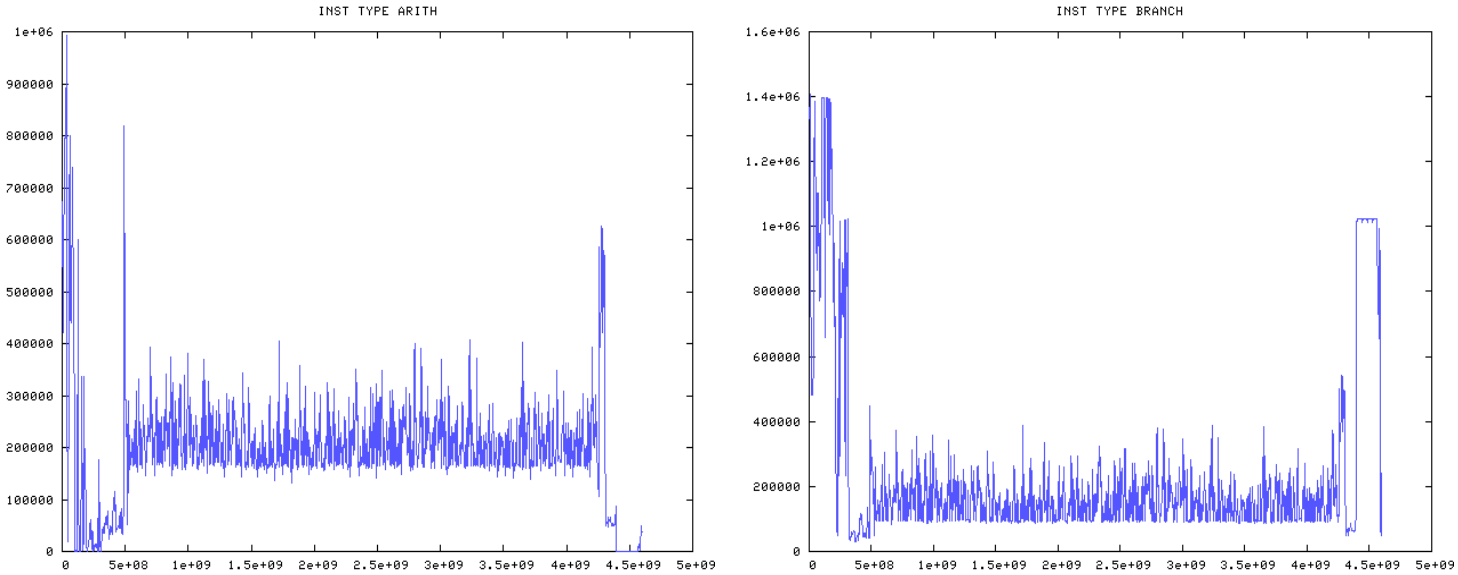

Figure 8: Arithmetic (left) and Branch (right) instructions livegraphs

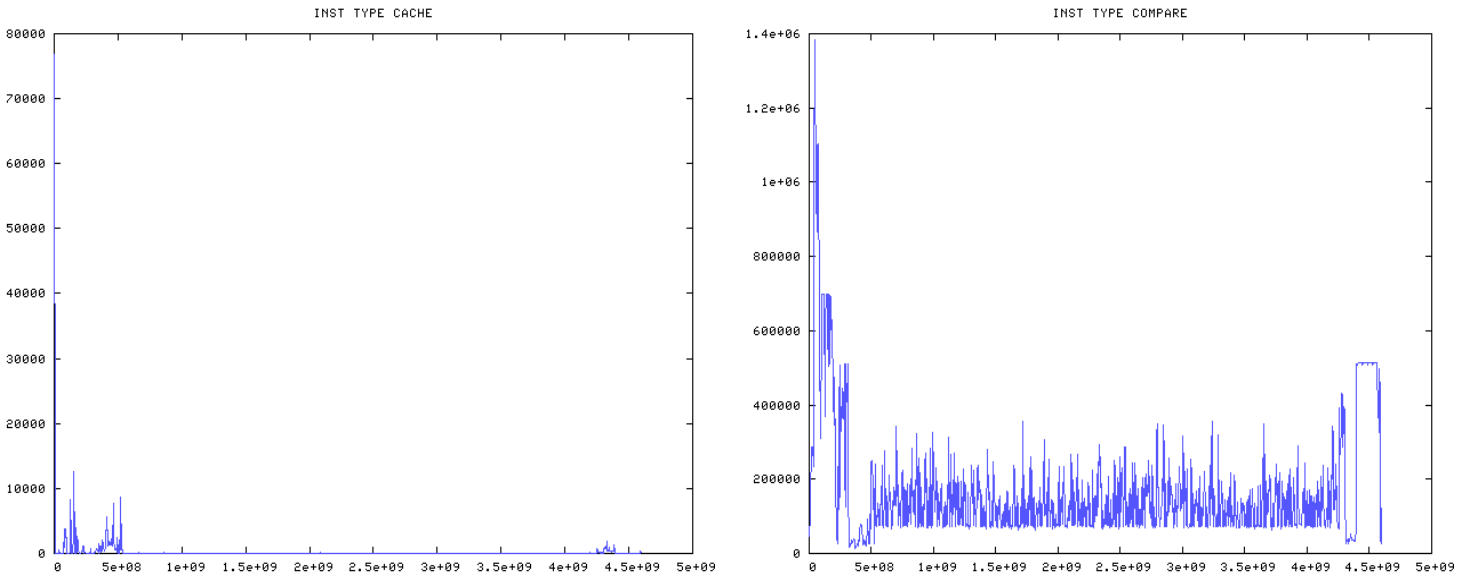

Figure 9: Cache (left) and Compare (right) instructions livegraphs

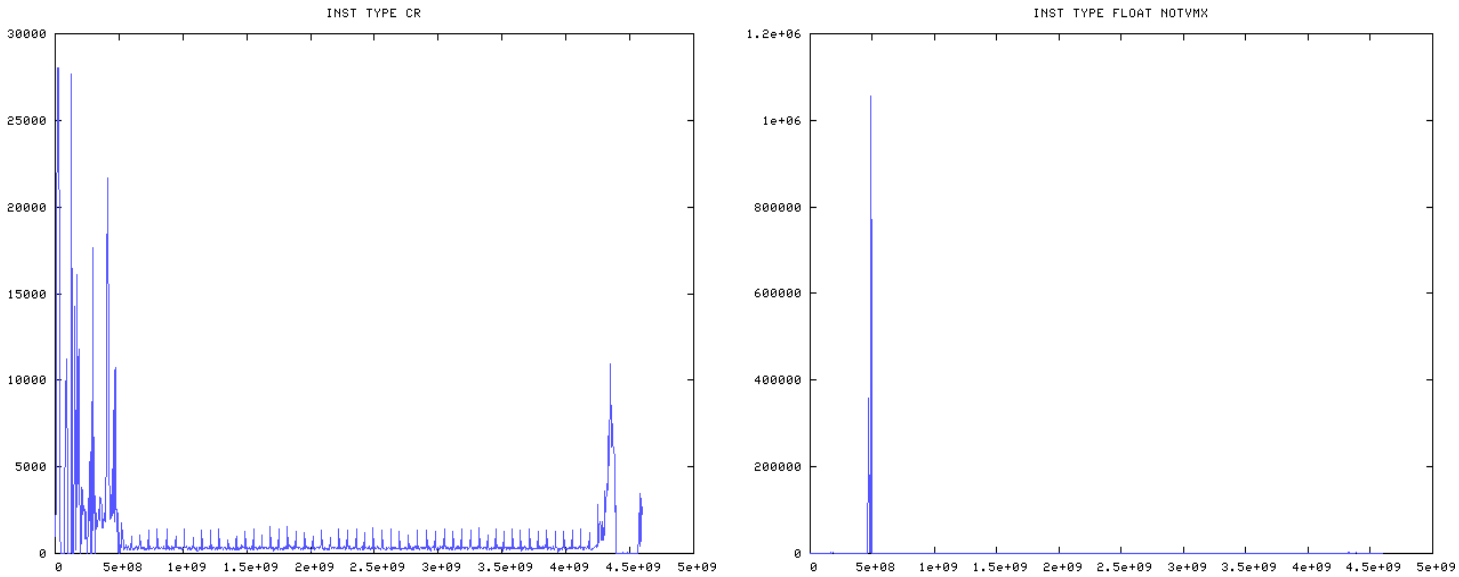

Figure 10: CR (left) and Not VMX-Float (right) instructions livegraphs

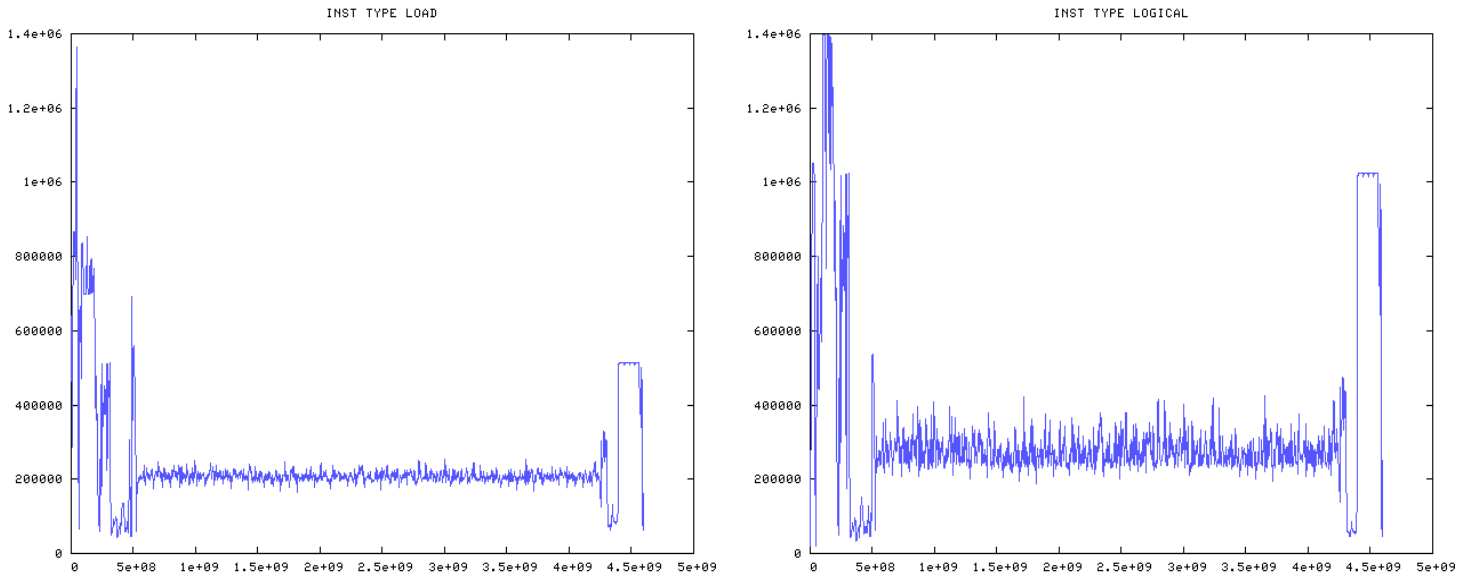

Figure 11: Load (left) and Logical (right) instructions livegraphs

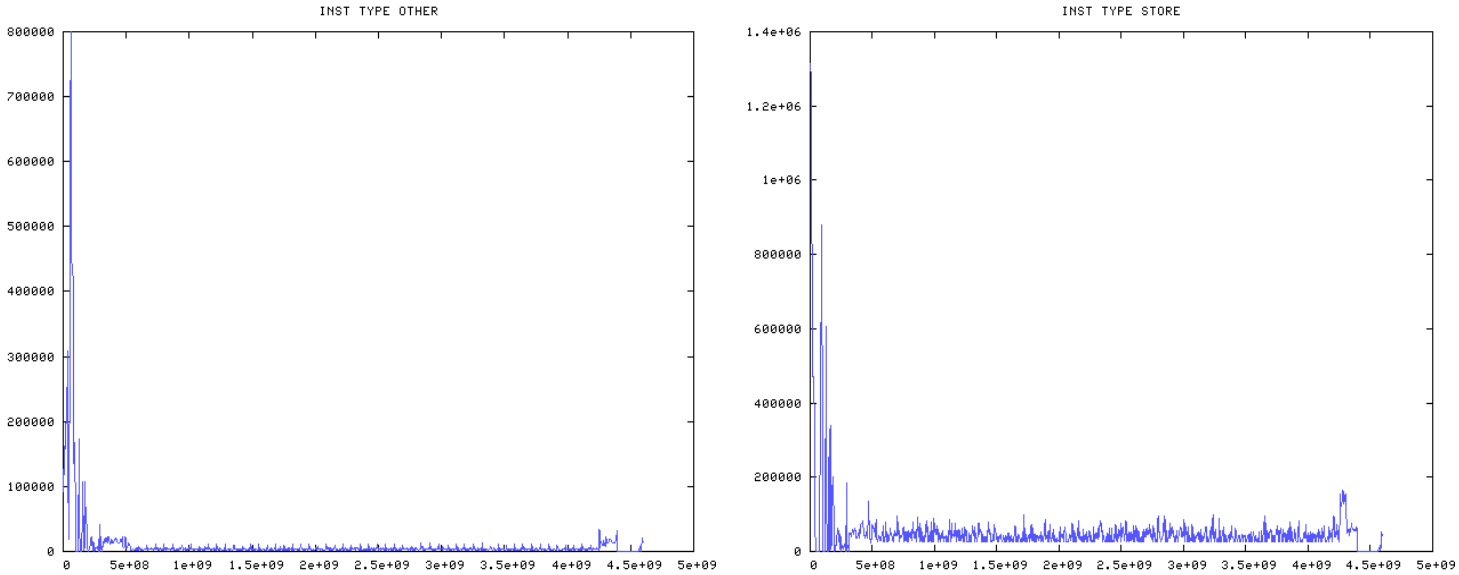

Figure 12: Other (left) and Store (right) instructions livegraphs

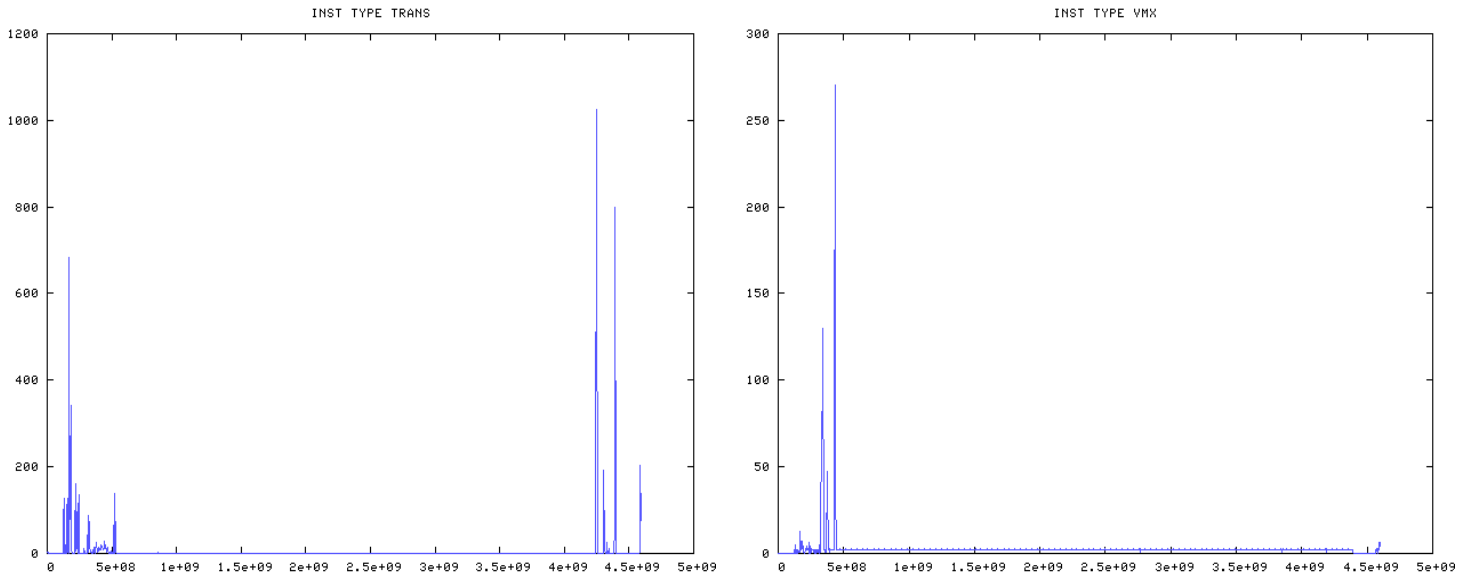

Figure 13: Trans (left) and VMX (right) instructions livegraphs

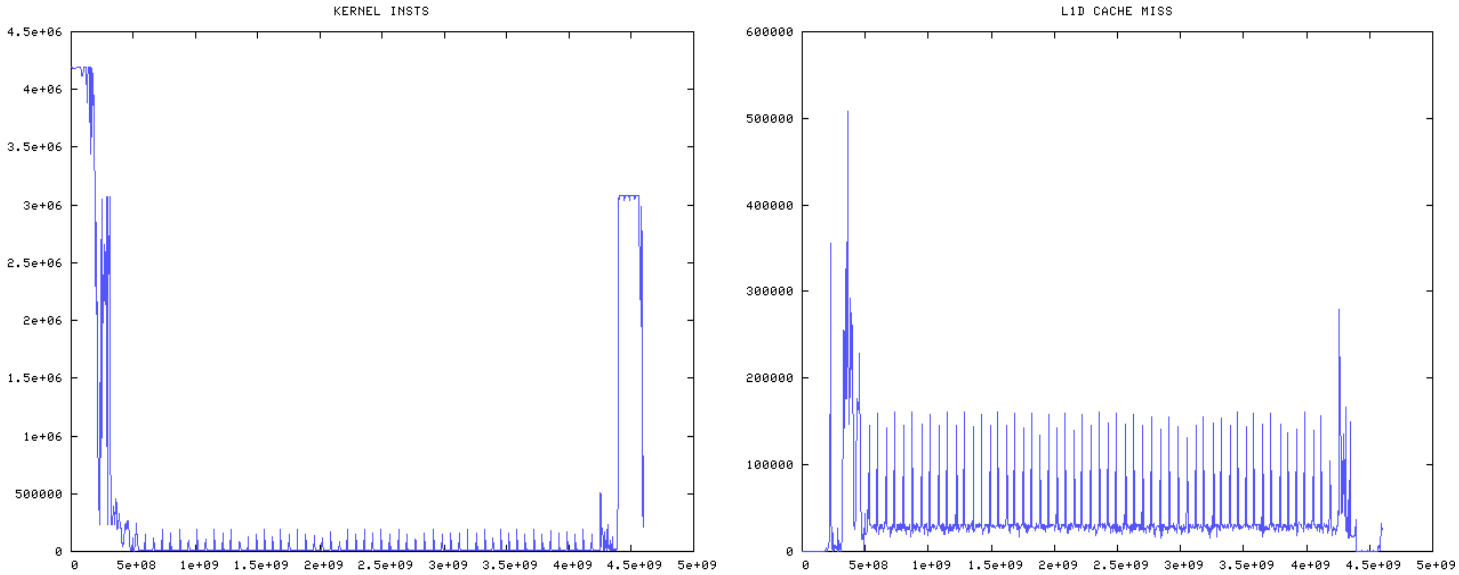

Figure 14: Kernel instructions (left) and L1D Cache Miss (right) livegraphs

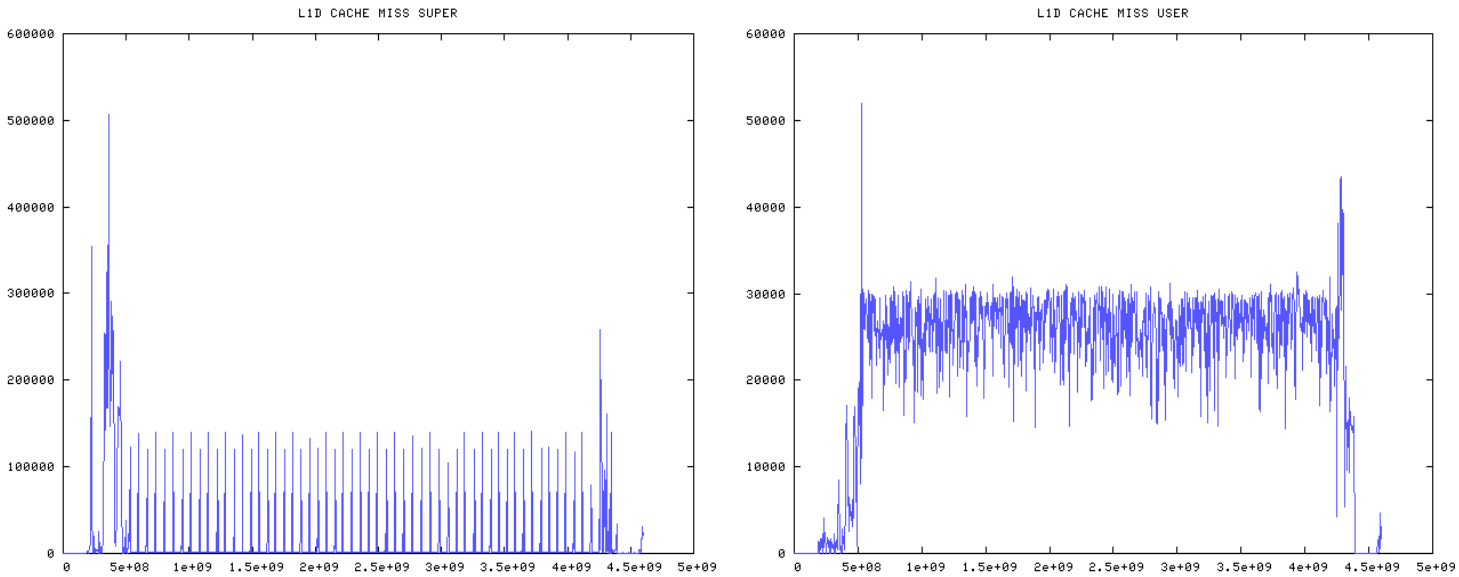

Figure 15: L1d Cache Miss Super (left) and L1d Cache Miss User (right) livegraphs

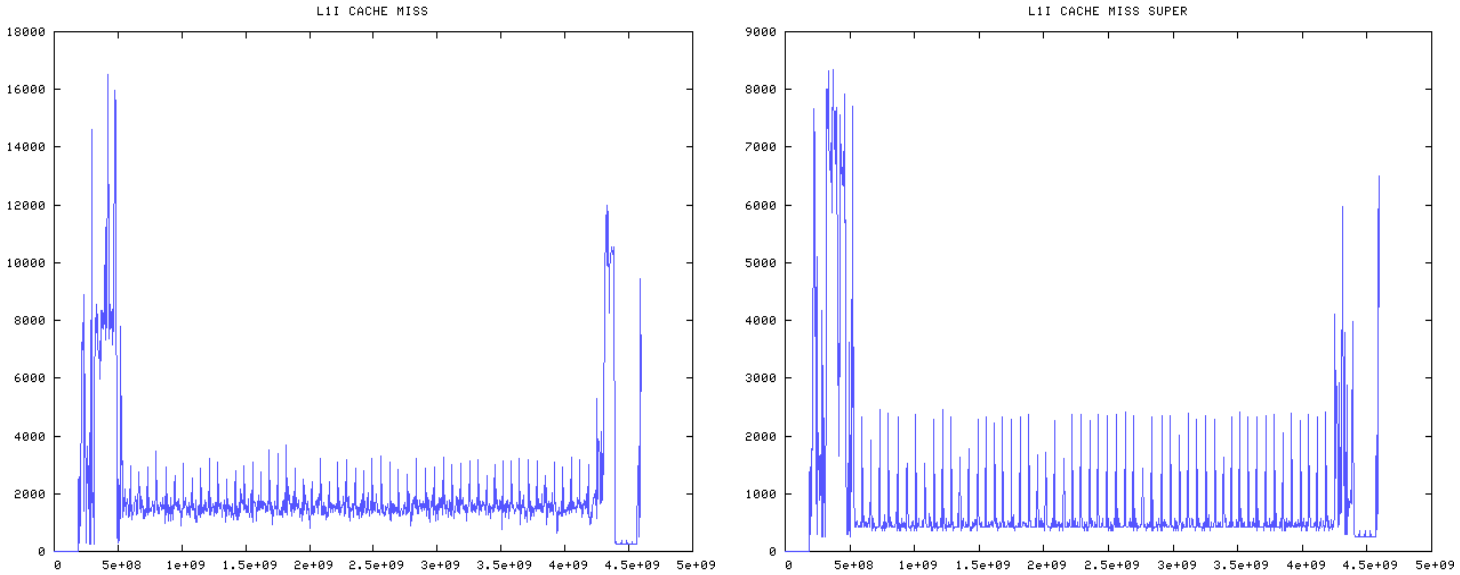

Figure 16: L1I Cache Miss (left) and L1I Cache Miss Super (right) livegraphs

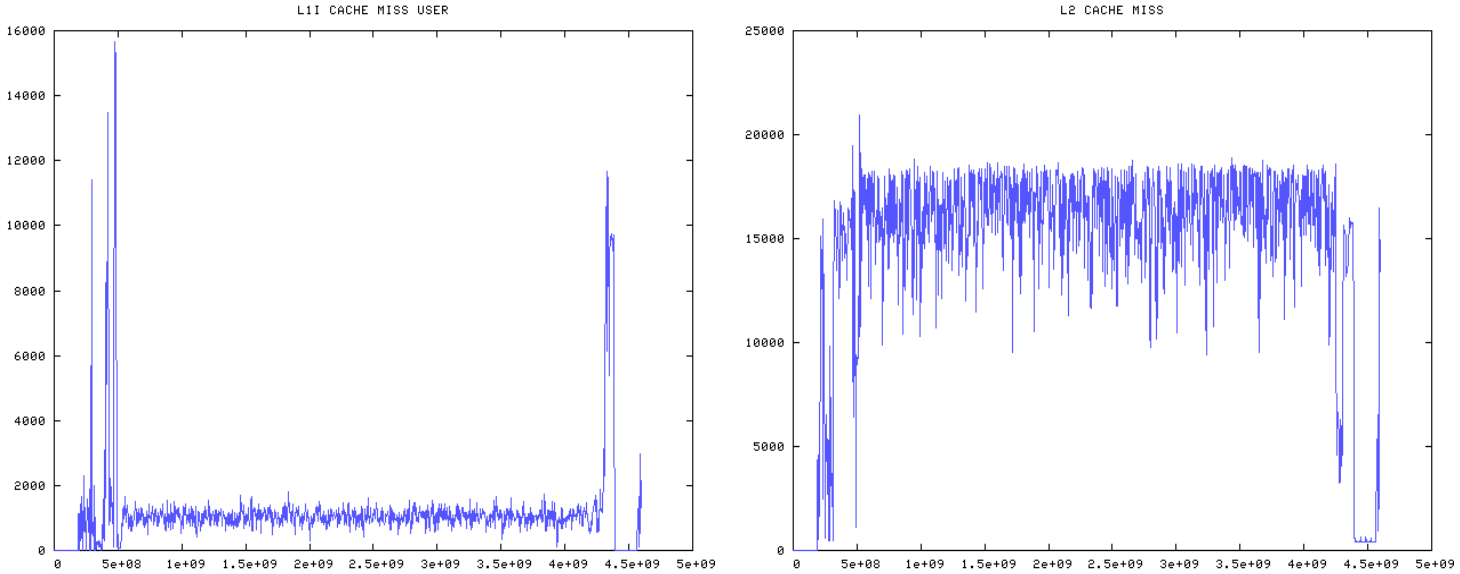

Figure 17: L1I Cache Miss User (left) and L2 Cache Miss (right) livegraphs

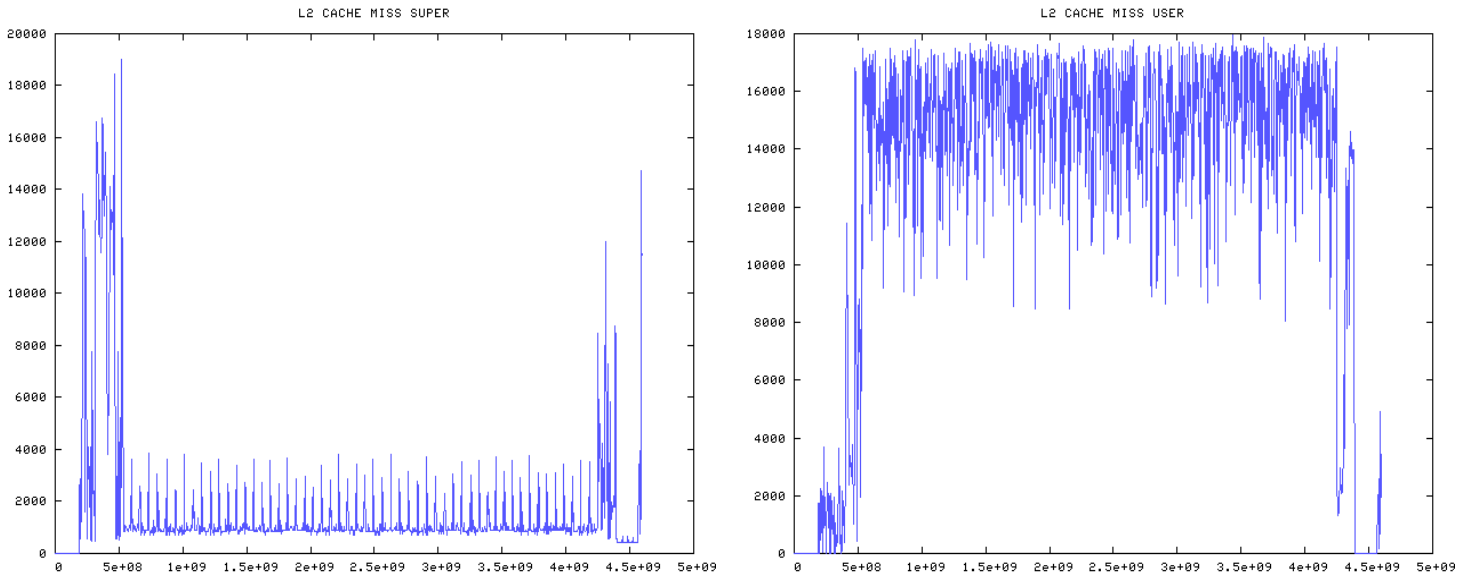

Figure 18: L2 Cache Miss Super (left) and L2 Cache Miss User (right) livegraphs

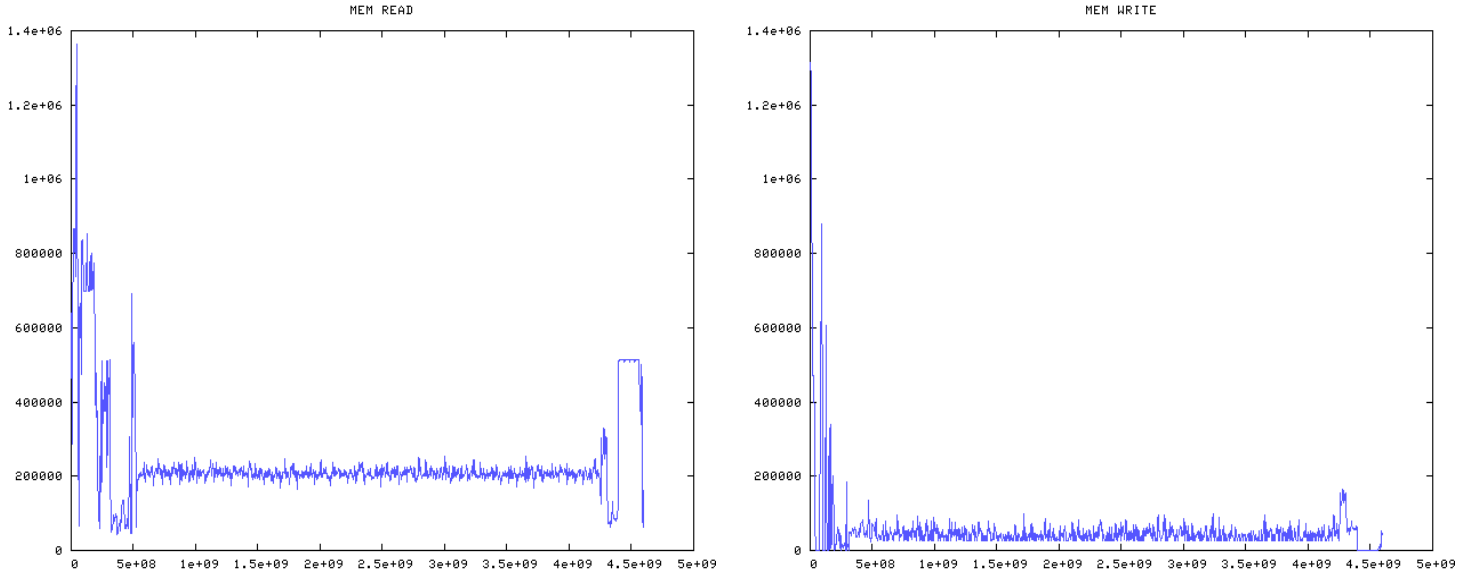

Figure 19: Memory-read (left) and Memory-Write (right) livegraphs

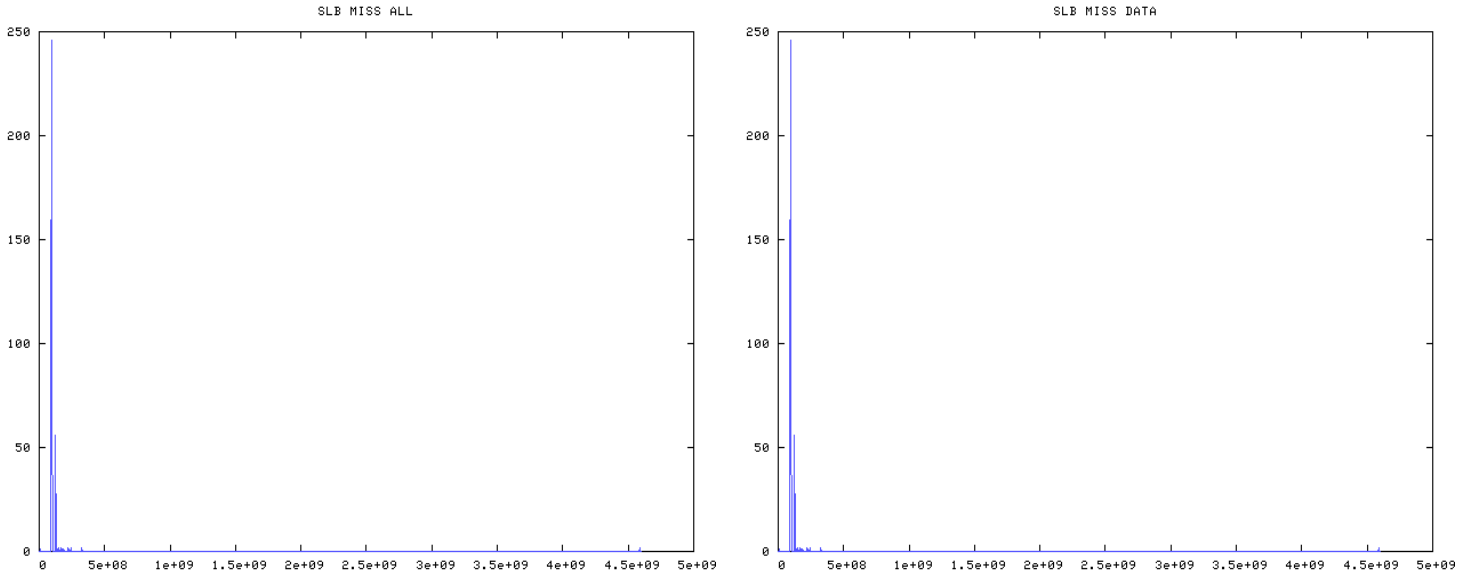

Figure 20: All SLB misses (left) and Data SLB Misses (right) livegraphs

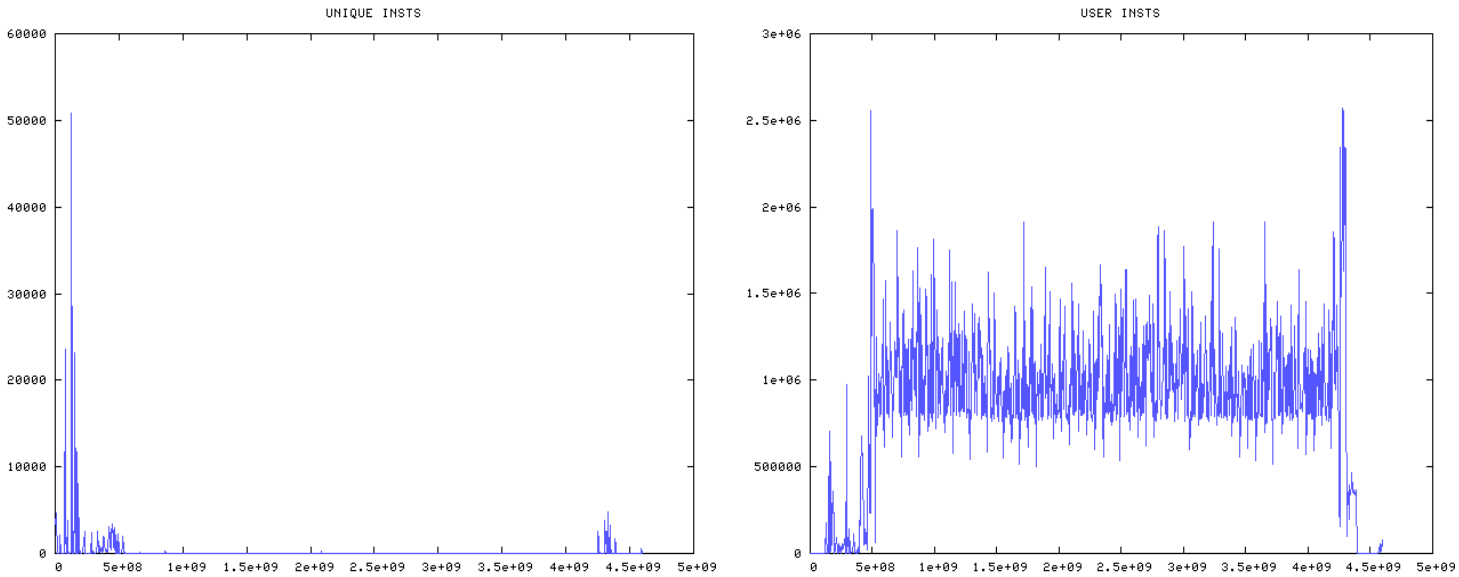

Figure 21: Unique Instructions (left) and User Instructions (right) livegraphs

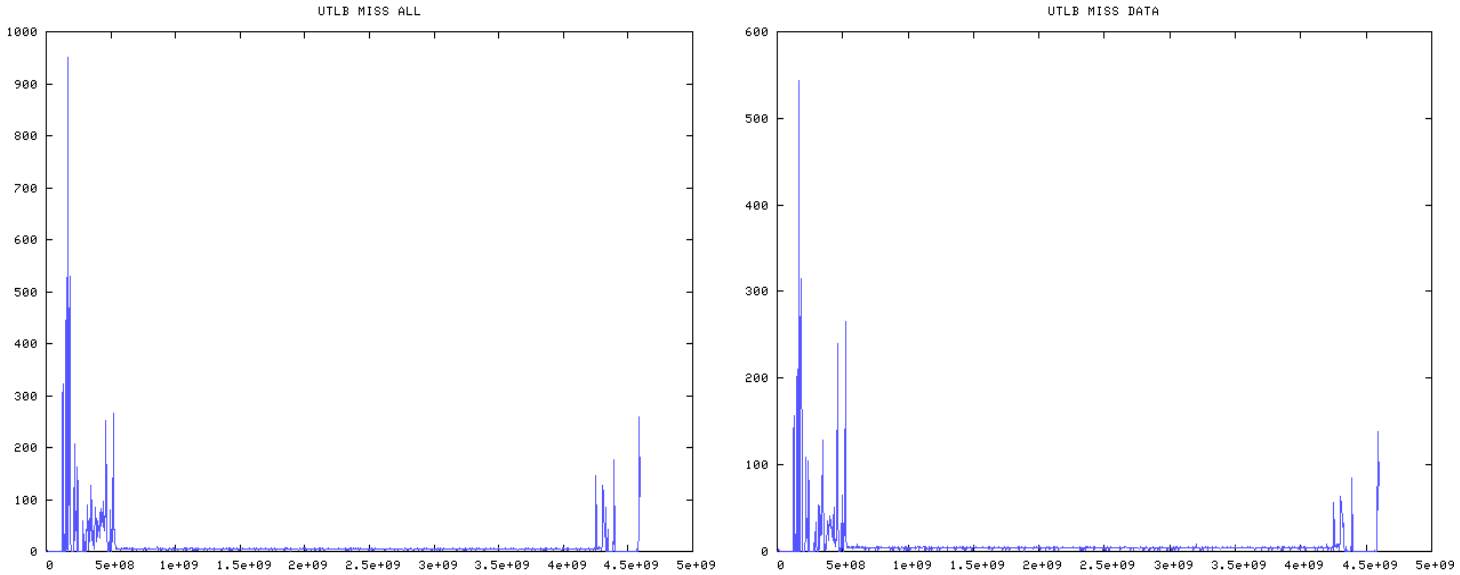

Figure 22: All User TLB Misses (left) and Data TLB Misses (right) livegraphs

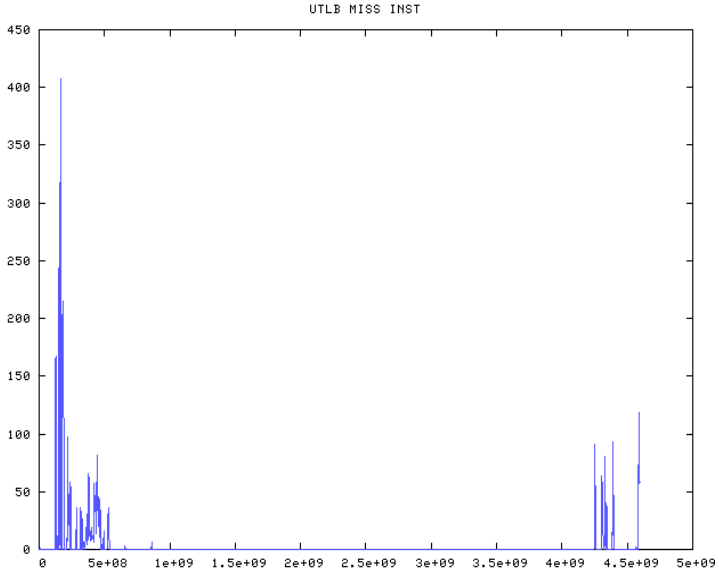

Figure 23: User TLB Instruction Miss livegraphs

B.2 CLUSTALW

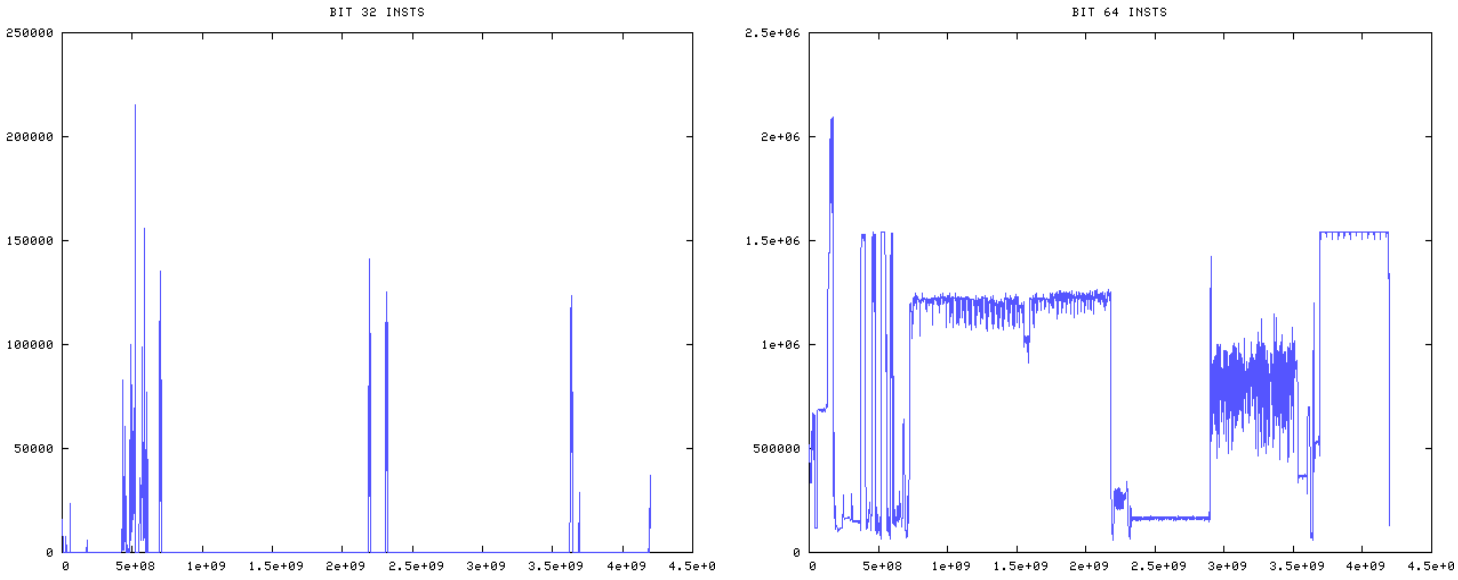

Figure 24: 32-bit (left) and 64-bit (right) instruction livegraphs

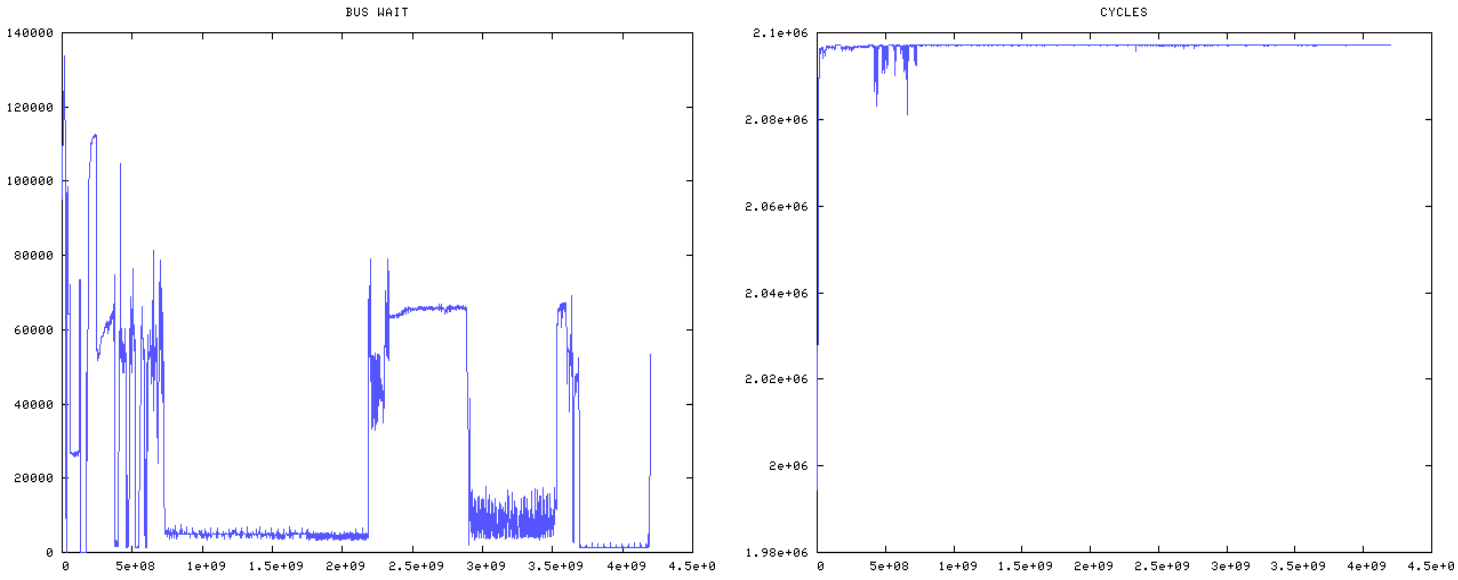

Figure 25: Bus-wait (left) and Cycles (right) livegraphs

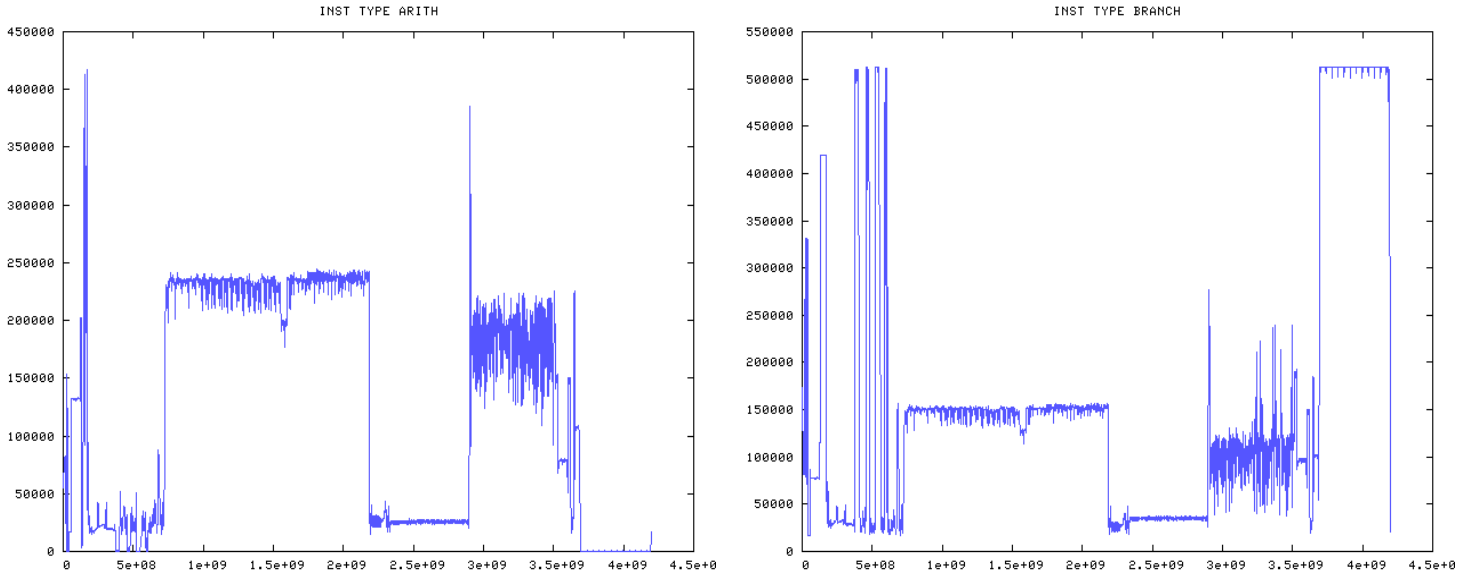

Figure 26: Arithmetic (left) and Branch (right) instructions livegraphs

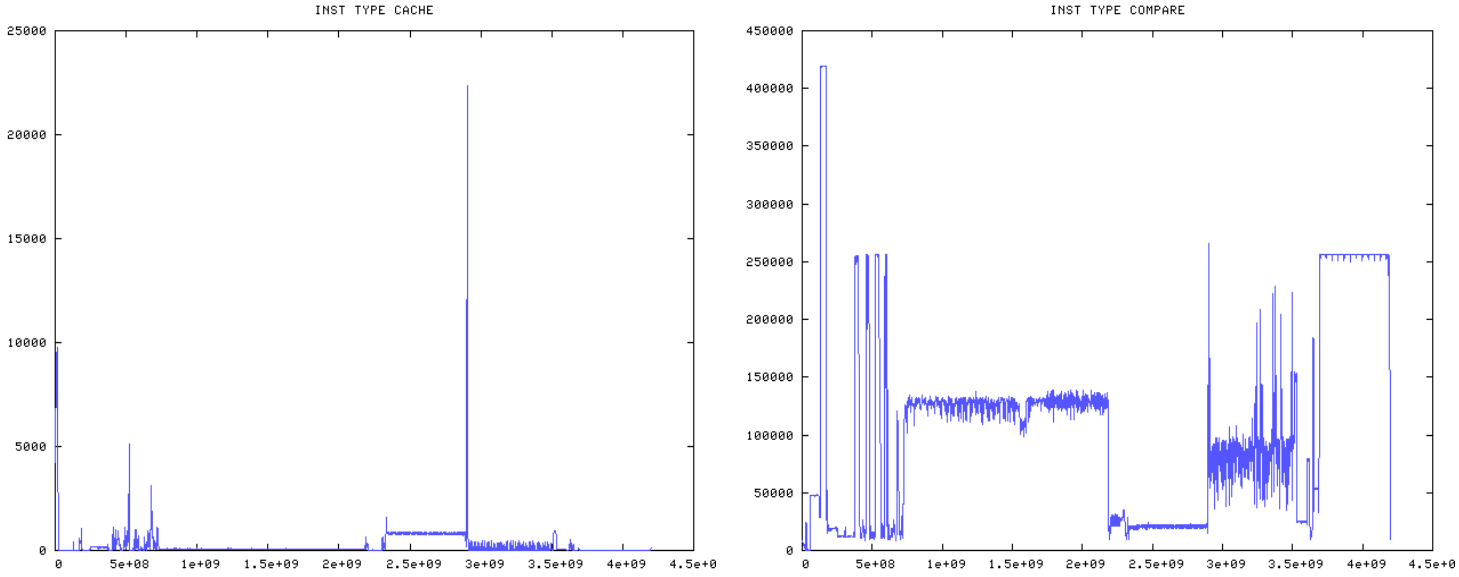

Figure 27: Cache (left) and Compare (right) instructions livegraphs

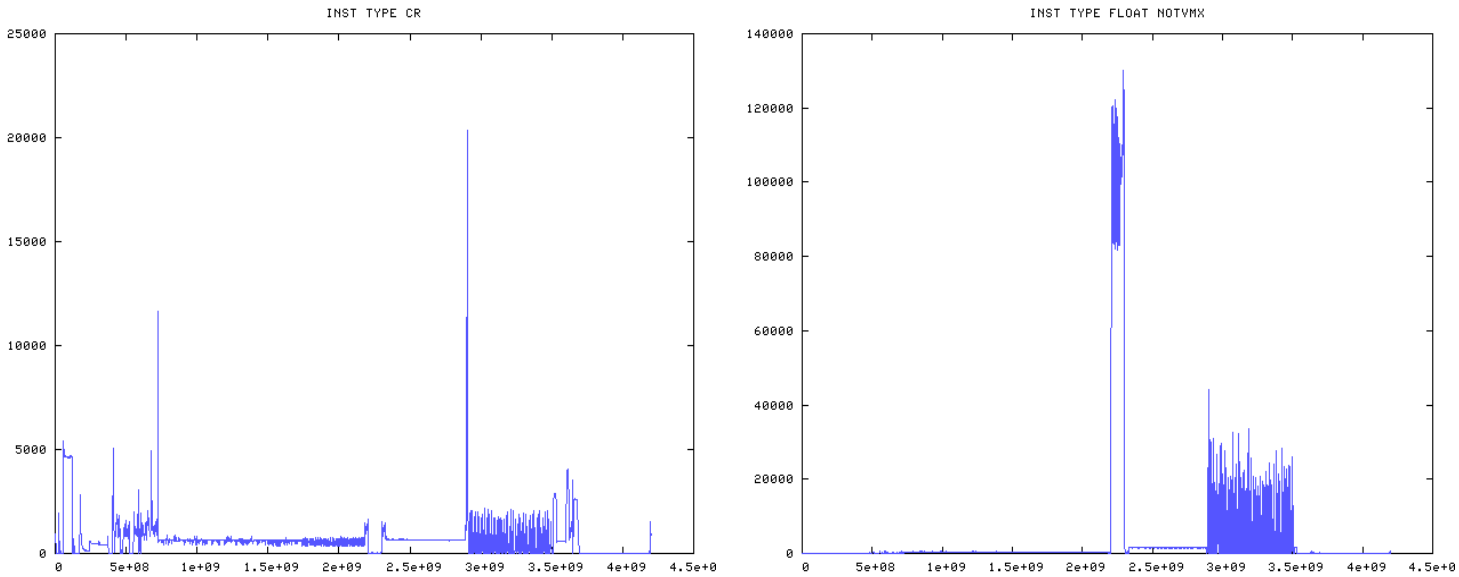

Figure 28: CR (left) and Not VMX-Float (right) instructions livegraphs

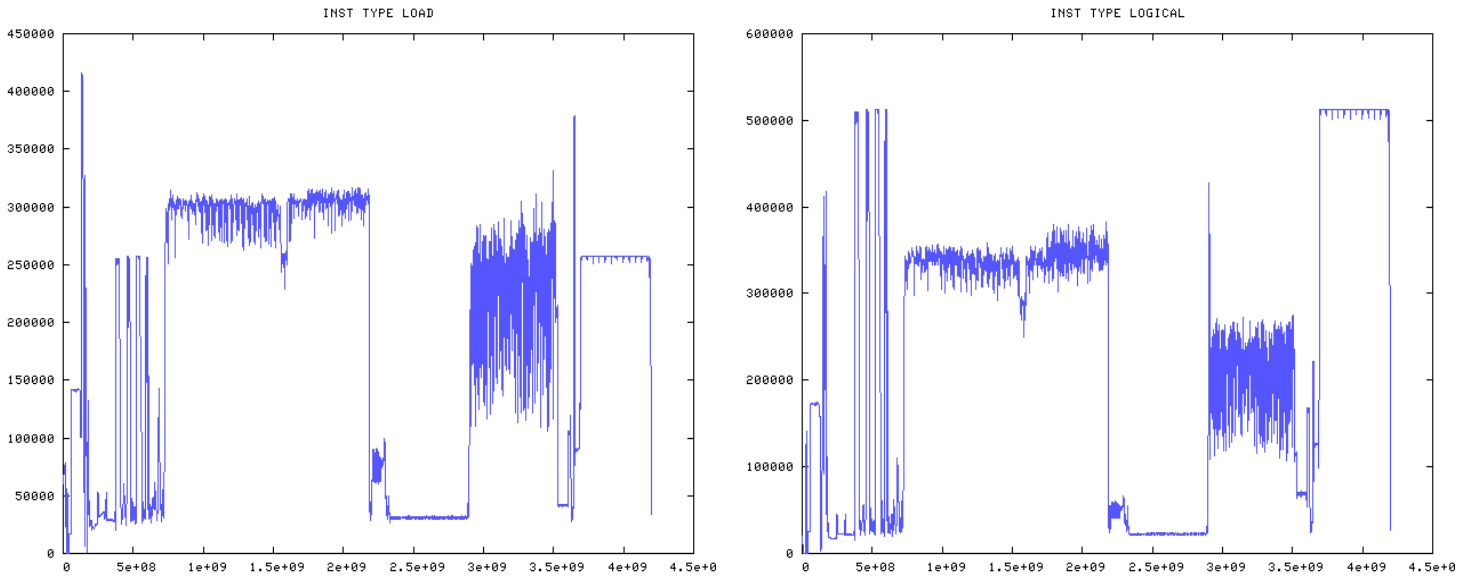

Figure 29: Load (left) and Logical (right) instructions livegraphs

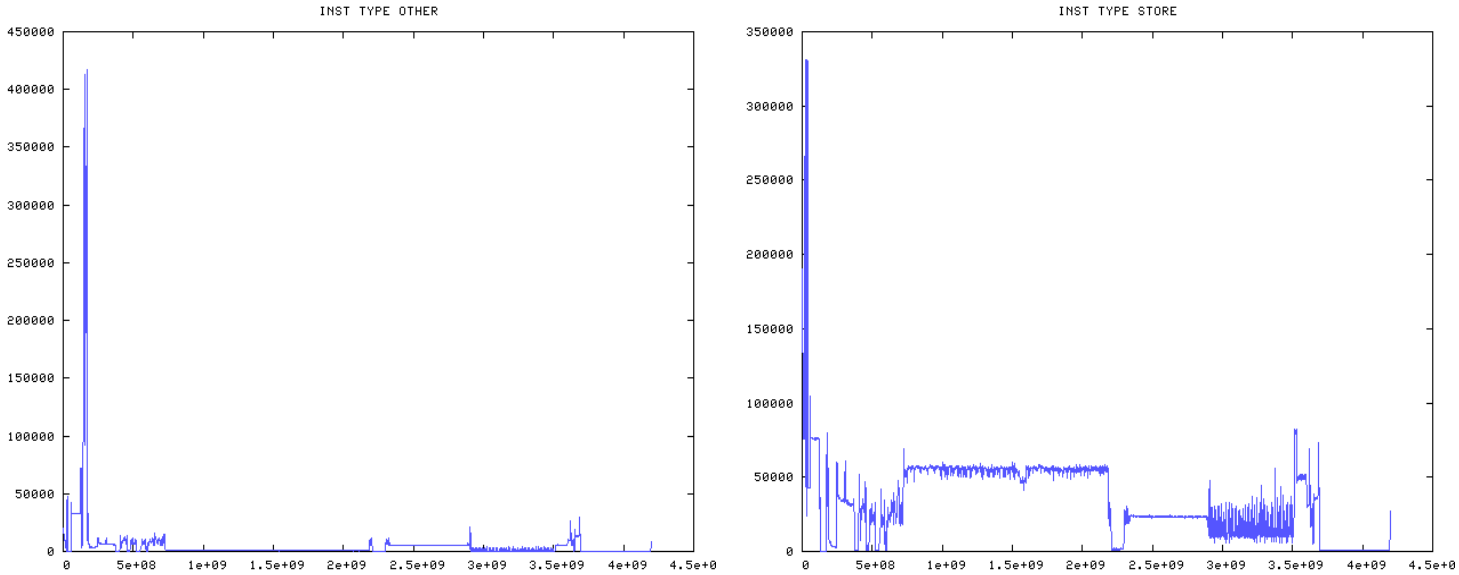

Figure 30: Other (left) and Store (right) instructions livegraphs

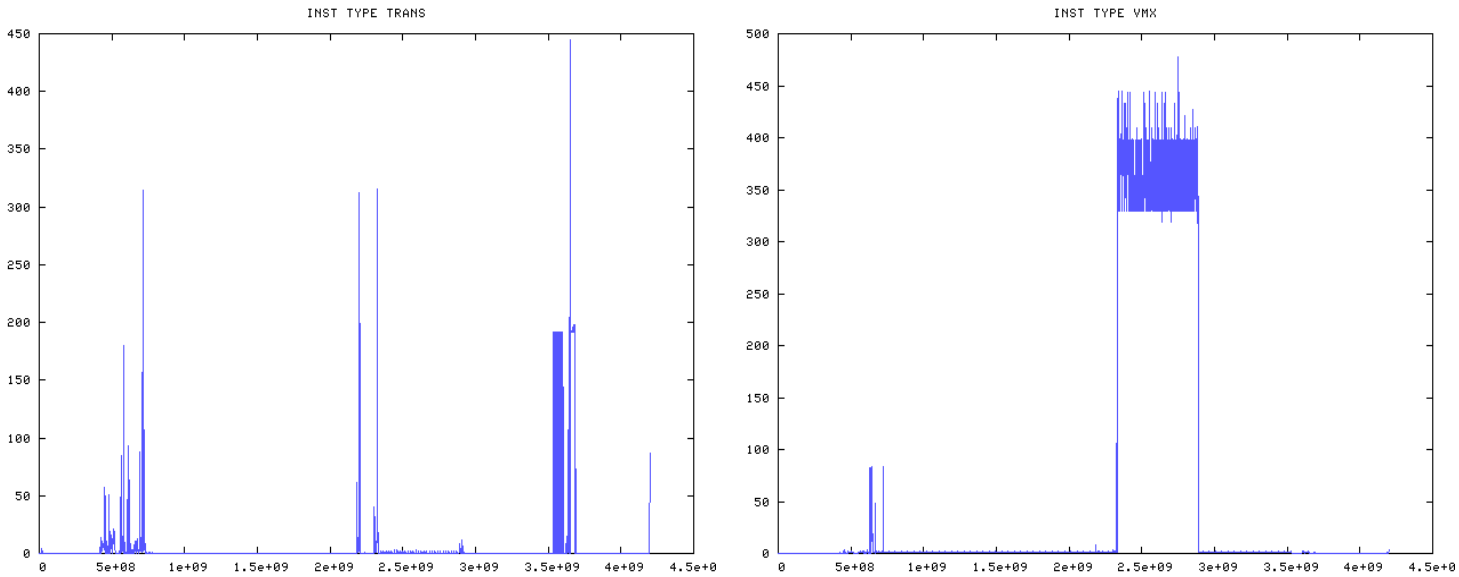

Figure 31: Trans (left) and VMX (right) instructions livegraphs

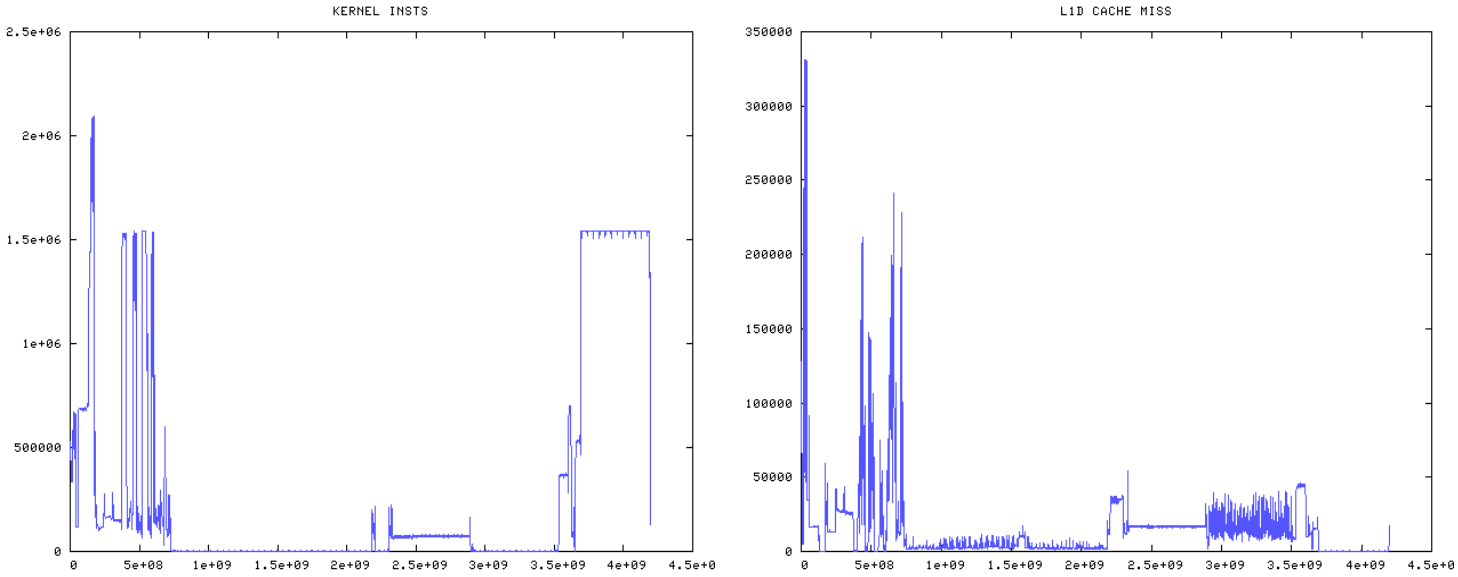

Figure 32: Kernel instructions (left) and L1D Cache Miss (right) livegraphs

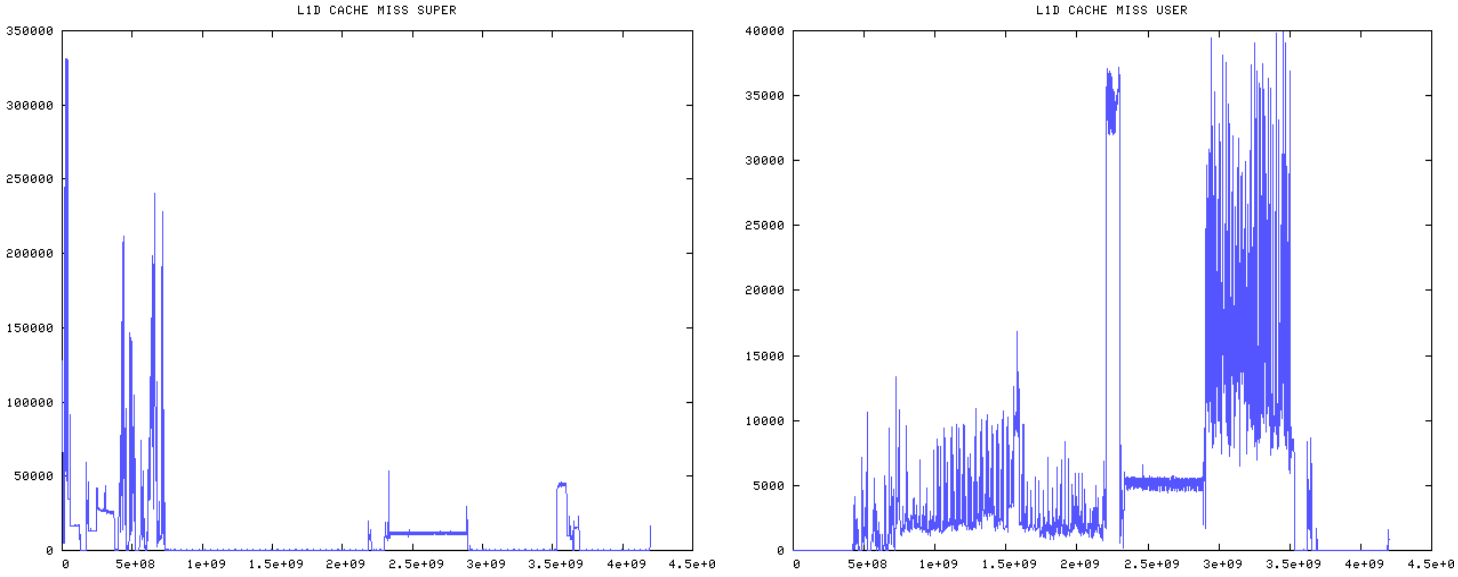

Figure 33: L1d Cache Miss Super (left) and L1d Cache Miss User (right) livegraphs

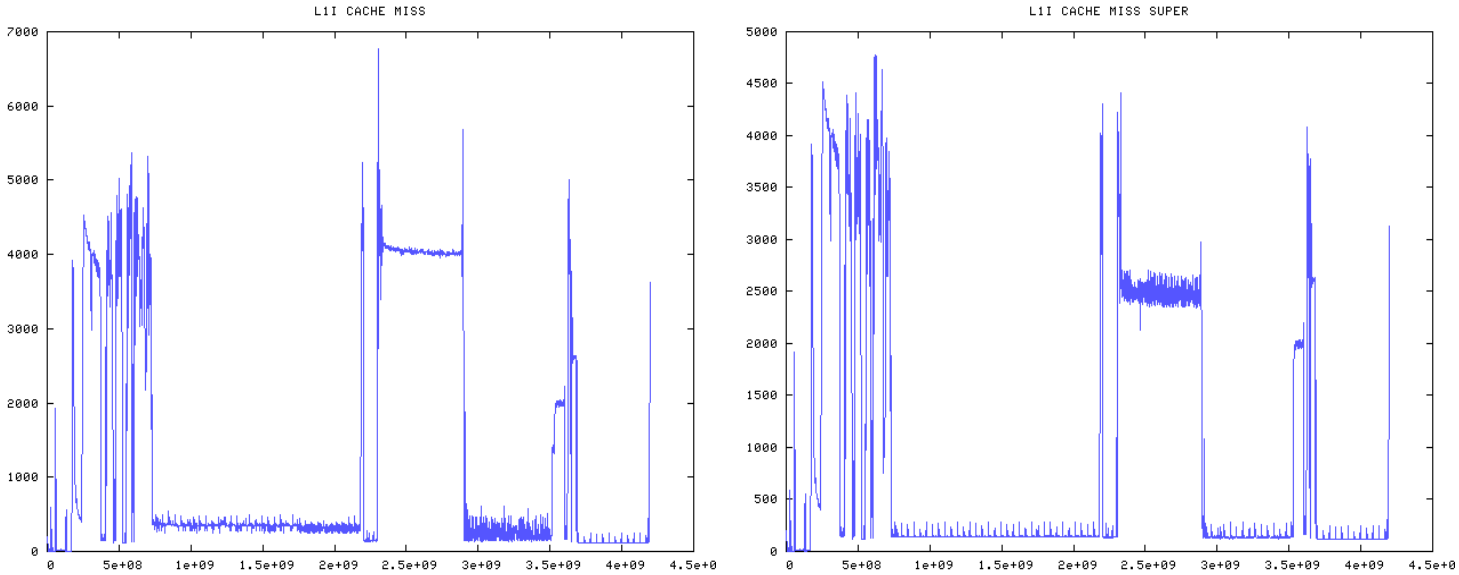

Figure 34: L1I Cache Miss (left) and L1I Cache Miss Super (right) livegraphs

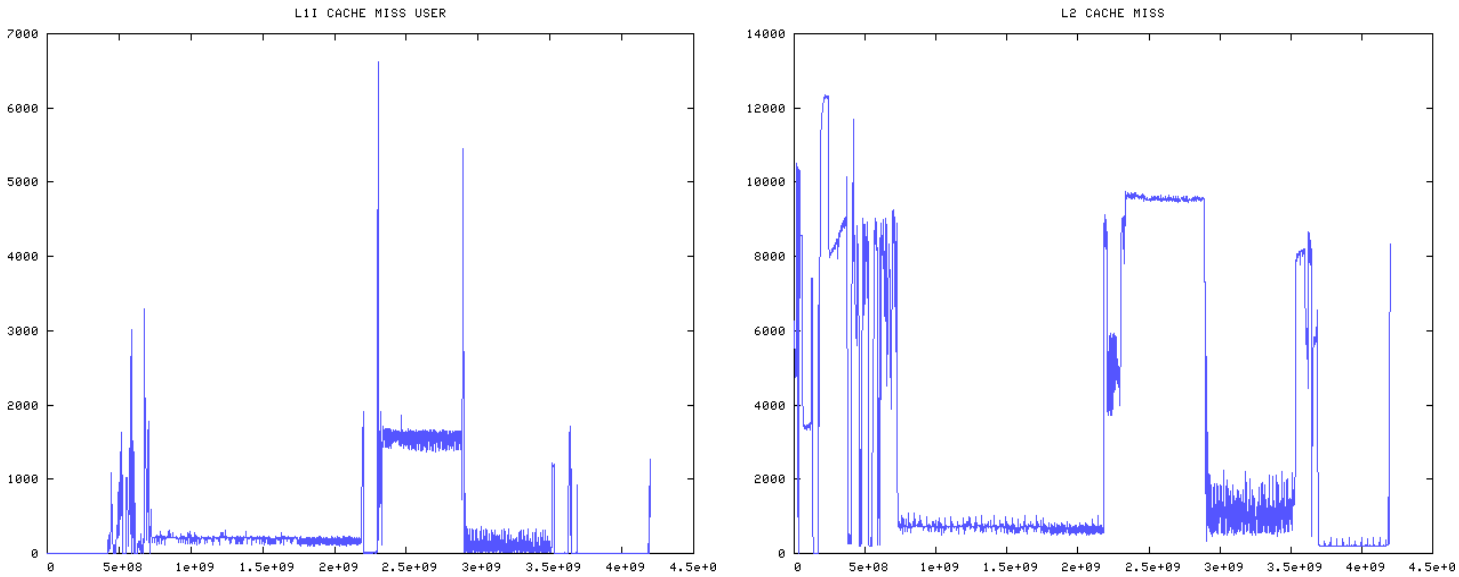

Figure 35: L1I Cache Miss User (left) and L2 Cache Miss (right) livegraphs

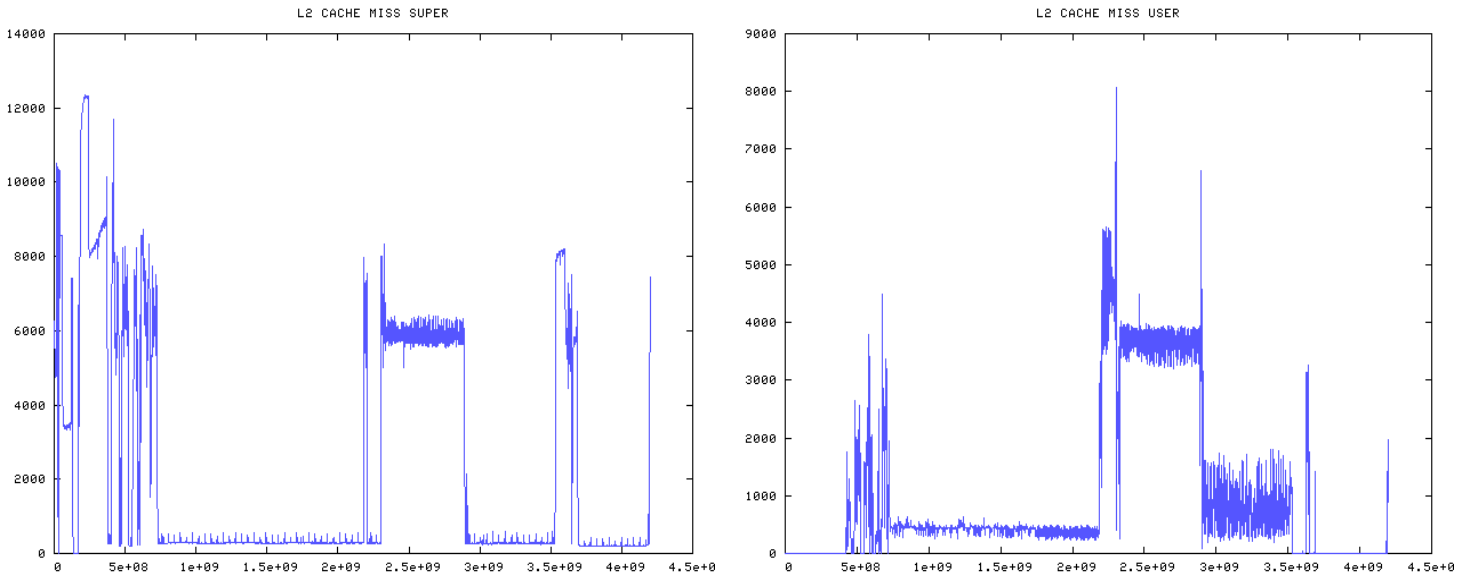

Figure 36: L2 Cache Miss Super (left) and L2 Cache Miss User (right) livegraphs

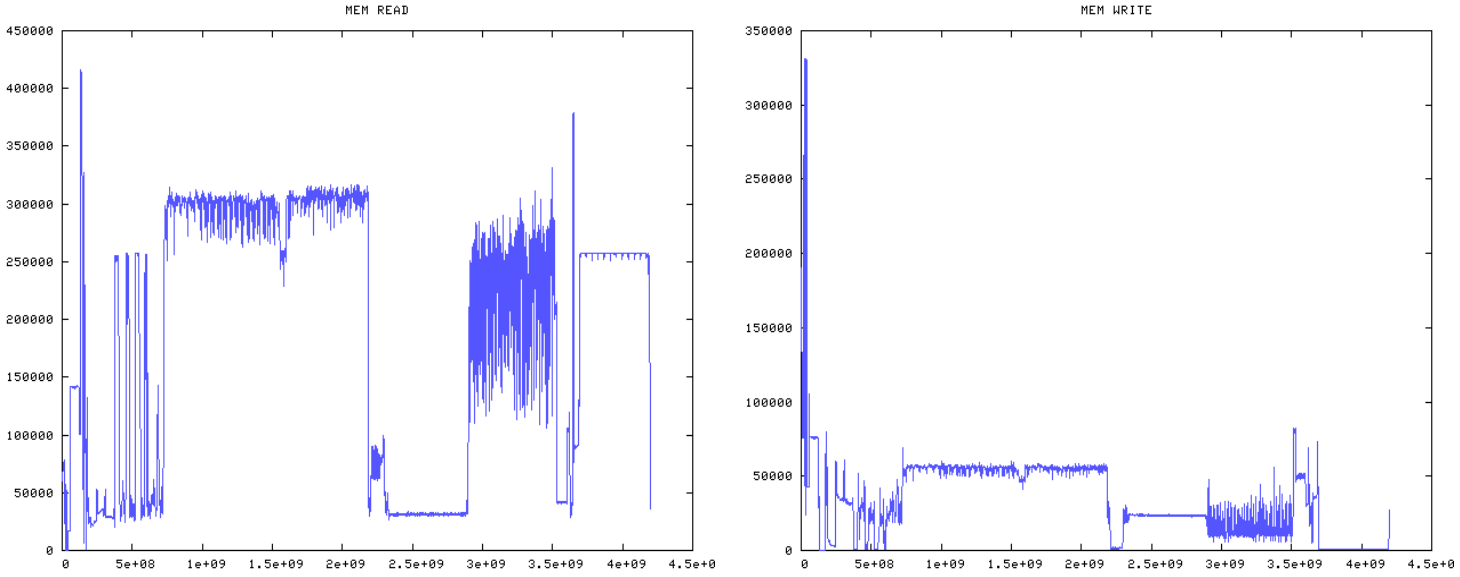

Figure 37: Memory-read (left) and Memory-Write (right) livegraphs

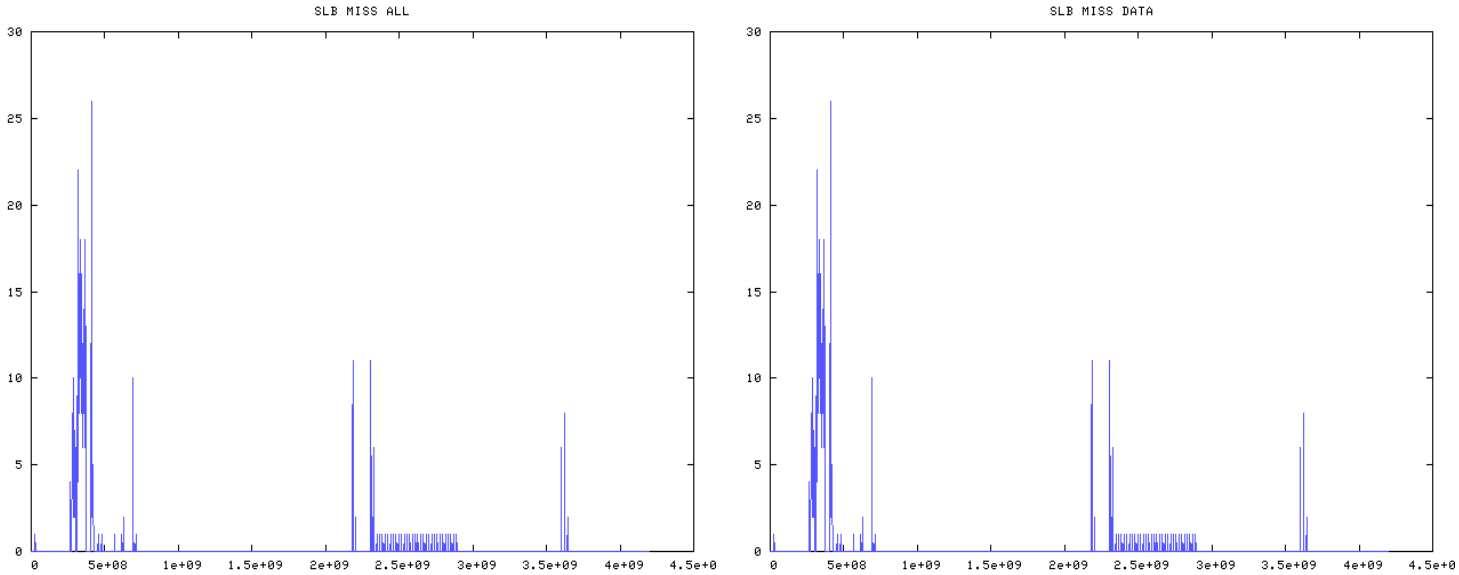

Figure 38: All SLB misses (left) and Data SLB Misses (right) livegraphs

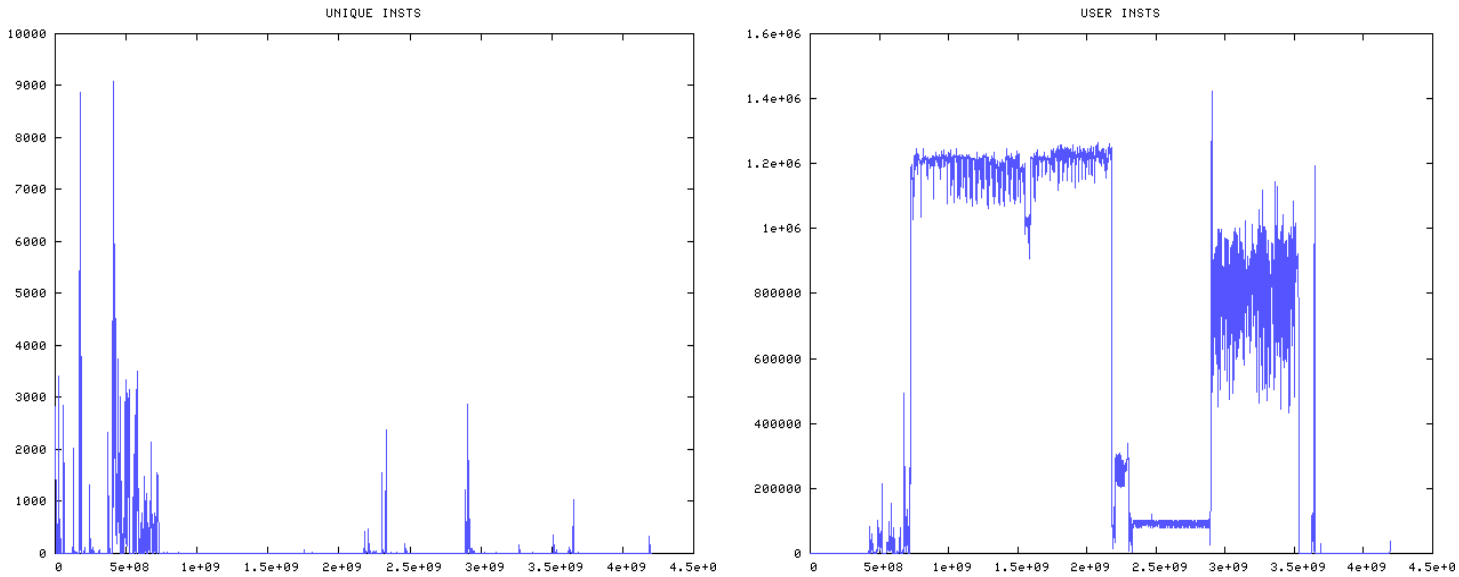

Figure 39: Unique Instructions (left) and User Instructions (right) livegraphs

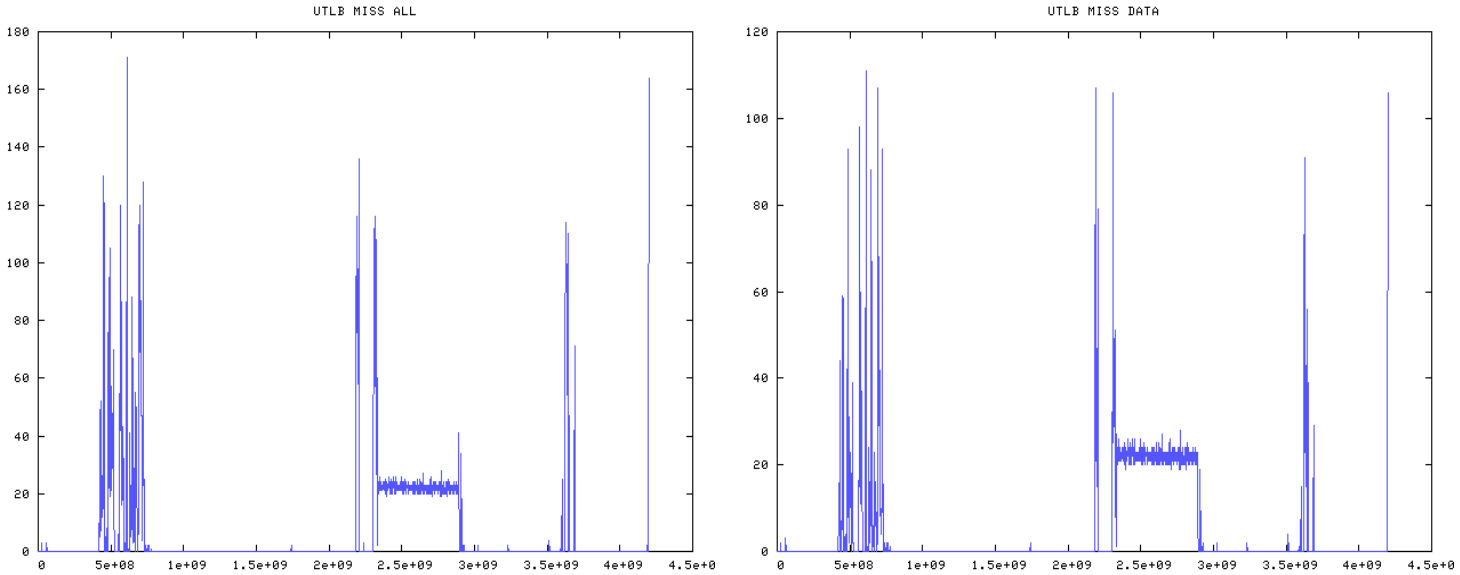

Figure 40: All User TLB Misses (left) and Data TLB Misses (right) livegraphs

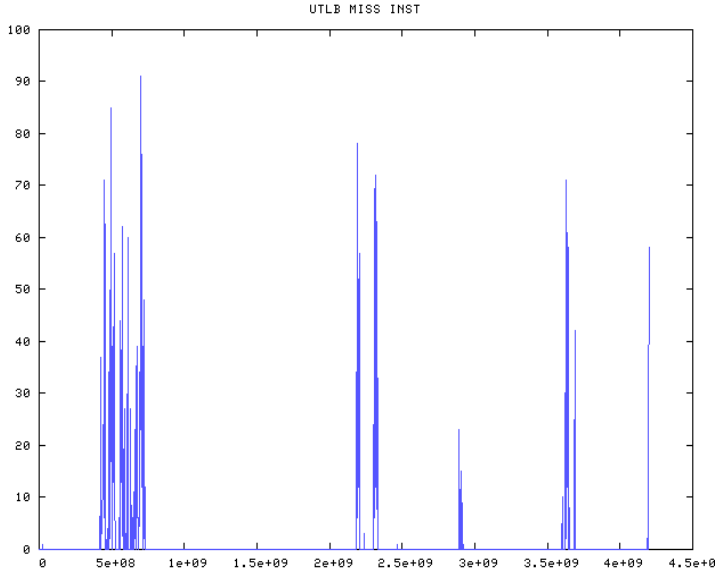

Figure 41: User TLB Instruction Miss livegraphs

B.3 FASTA

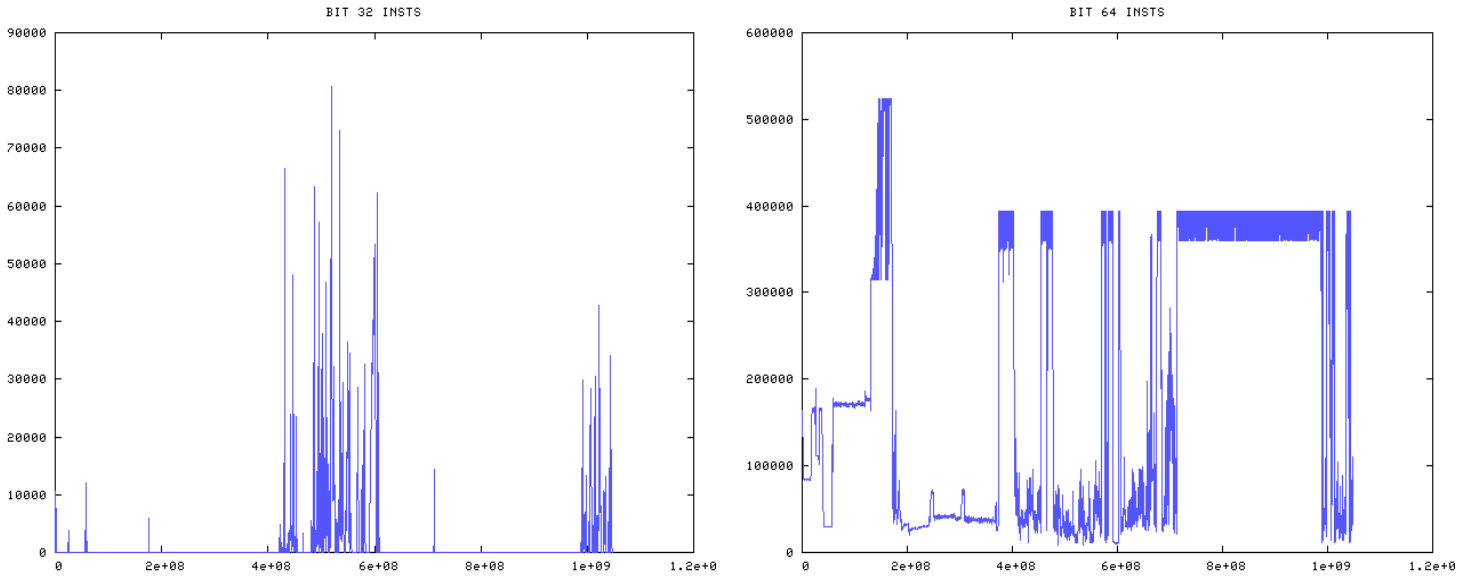

Figure 42: 32-bit (left) and 64-bit (right) instruction livegraphs

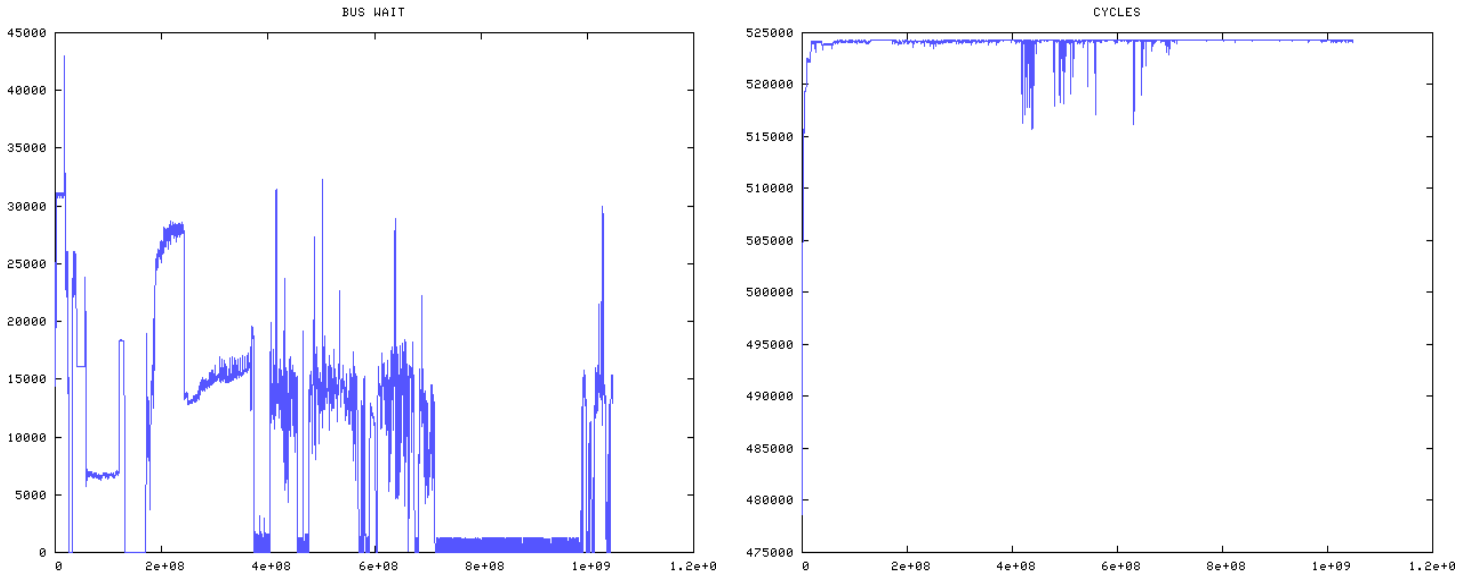

Figure 43: Bus-wait (left) and Cycles (right) livegraphs

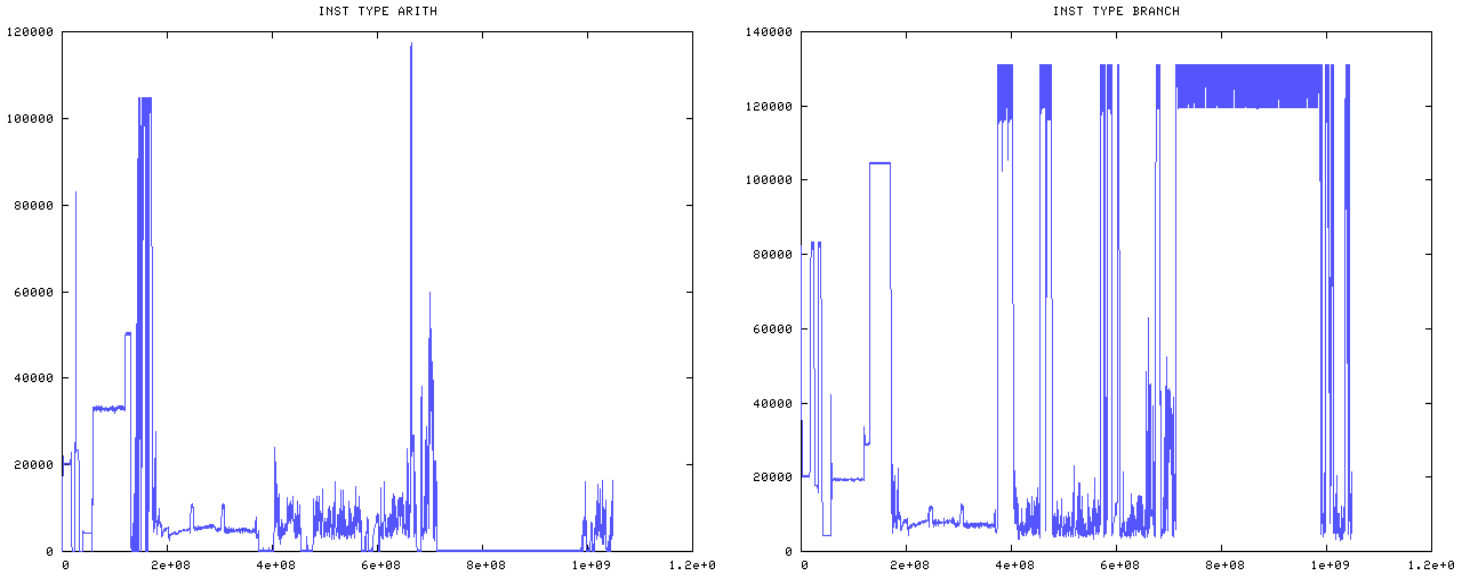

Figure 44: Arithmetic (left) and Branch (right) instructions livegraphs

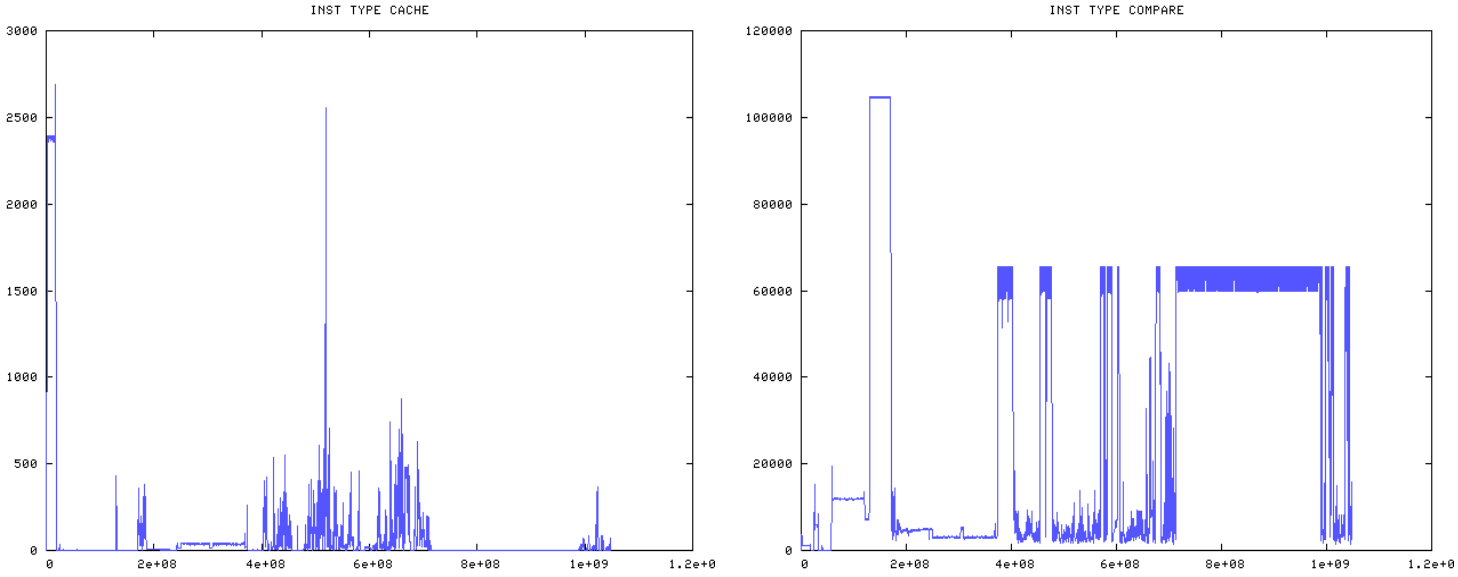

Figure 45: Cache (left) and Compare (right) instructions livegraphs

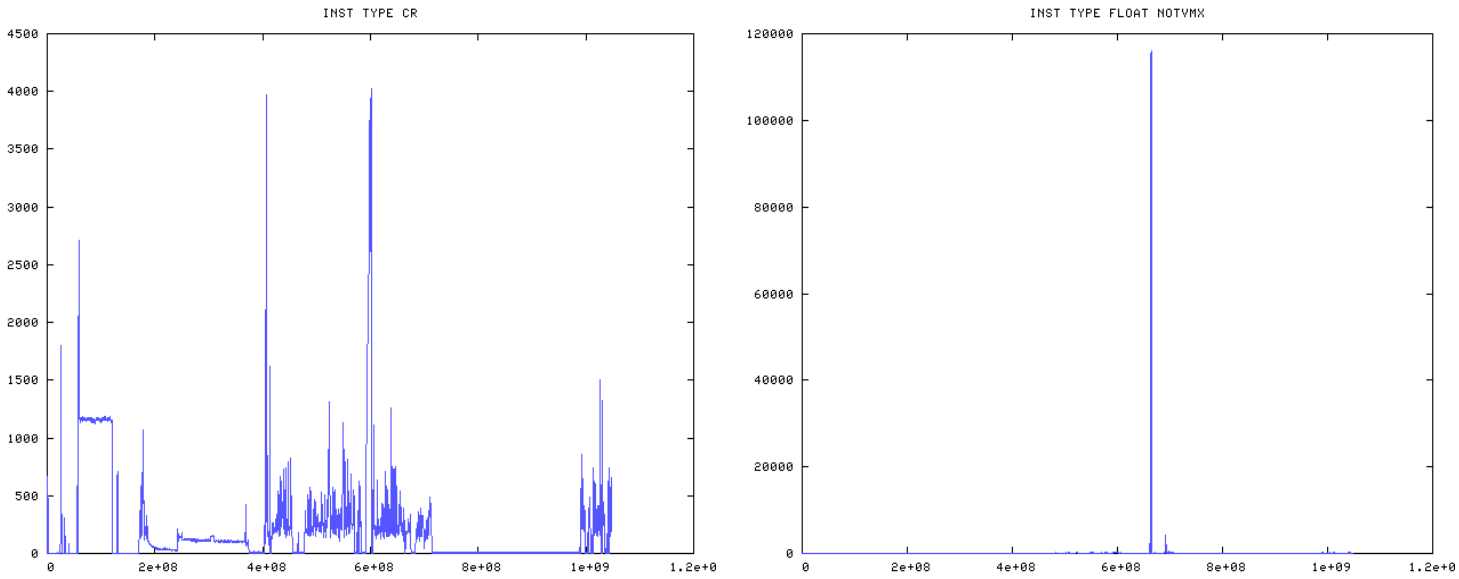

Figure 46: CR (left) and Not VMX-Float (right) instructions livegraphs

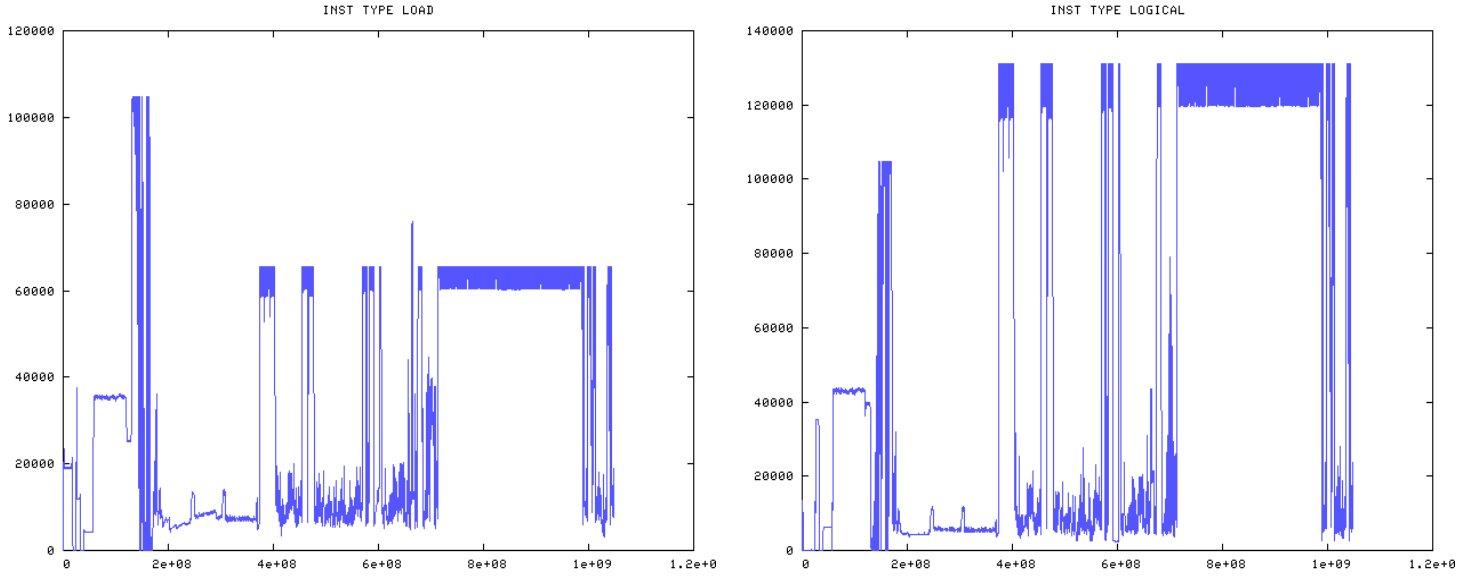

Figure 47: Load (left) and Logical (right) instructions livegraphs

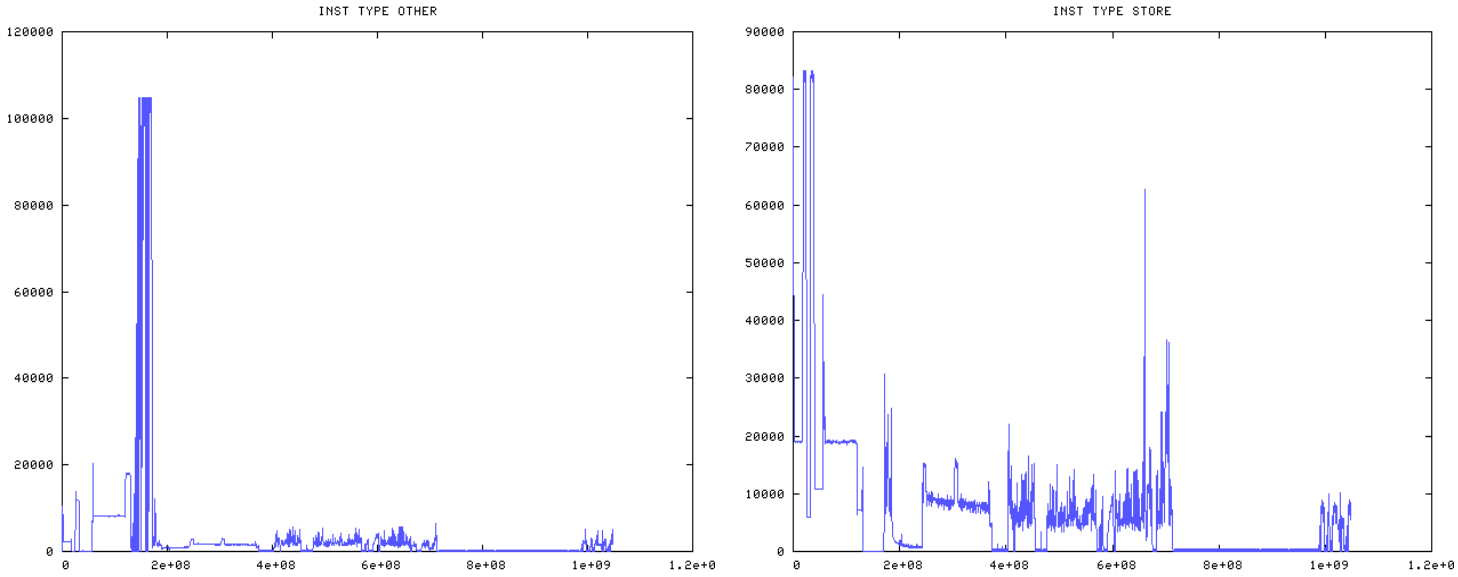

Figure 48: Other (left) and Store (right) instructions livegraphs

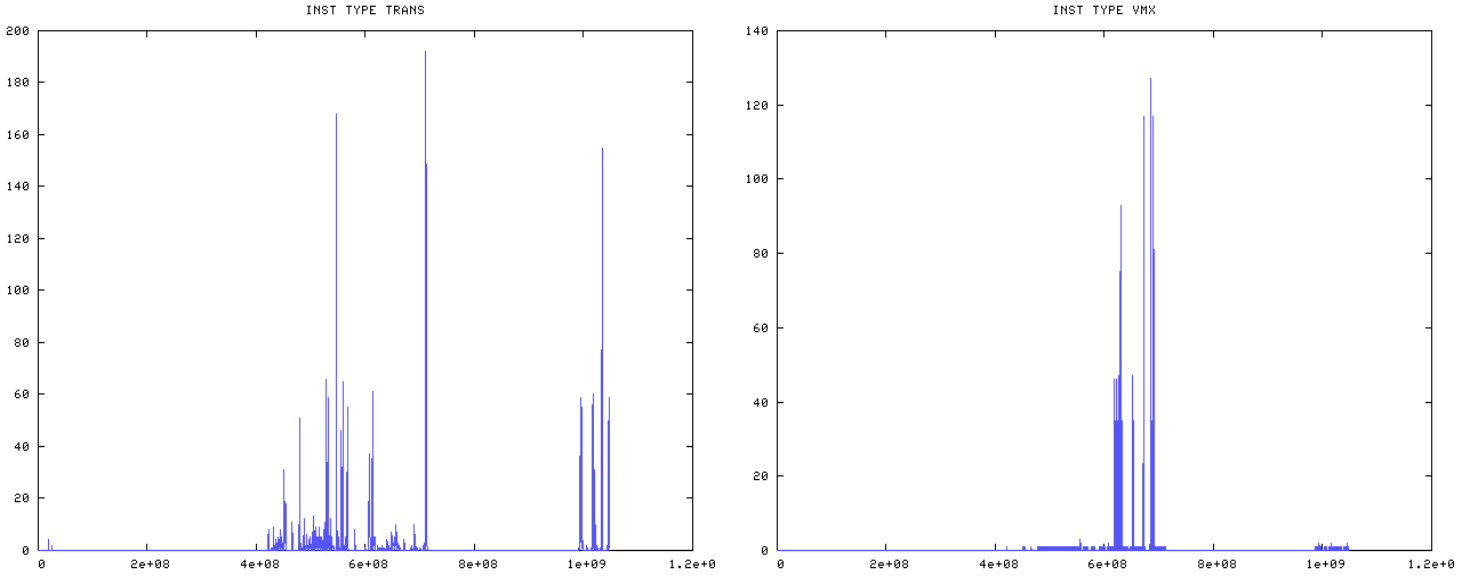

Figure 49: Trans (left) and VMX (right) instructions livegraphs

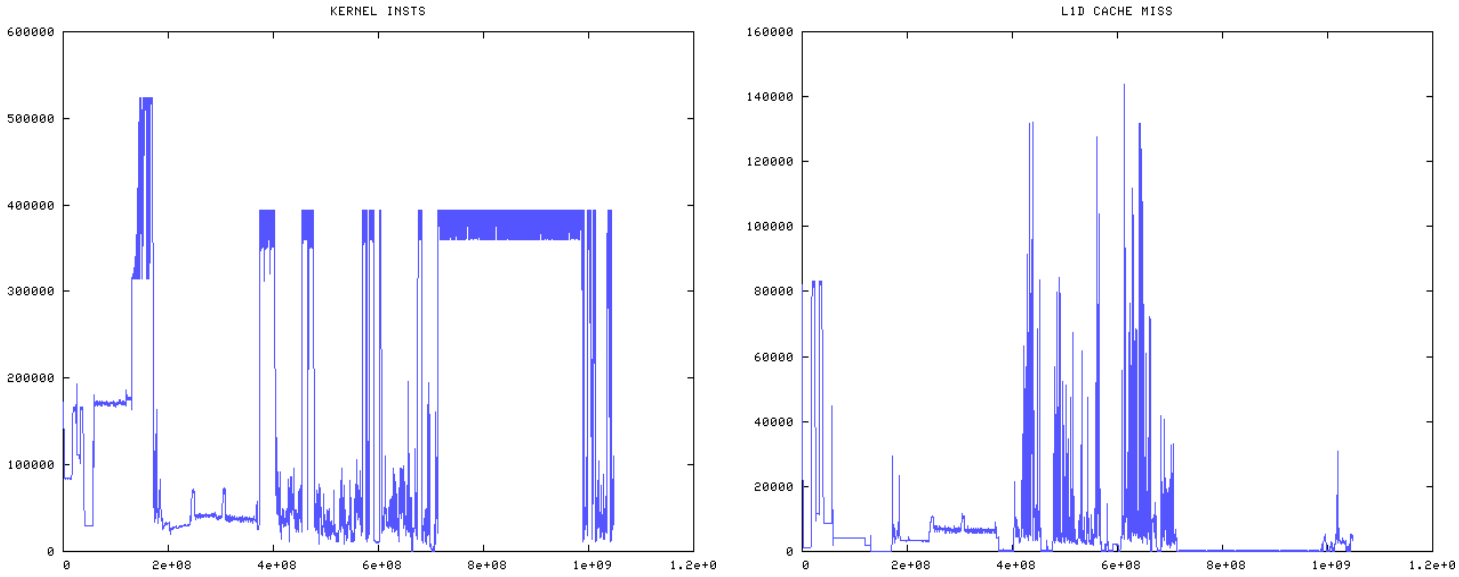

Figure 50: Kernel instructions (left) and L1D Cache Miss (right) livegraphs

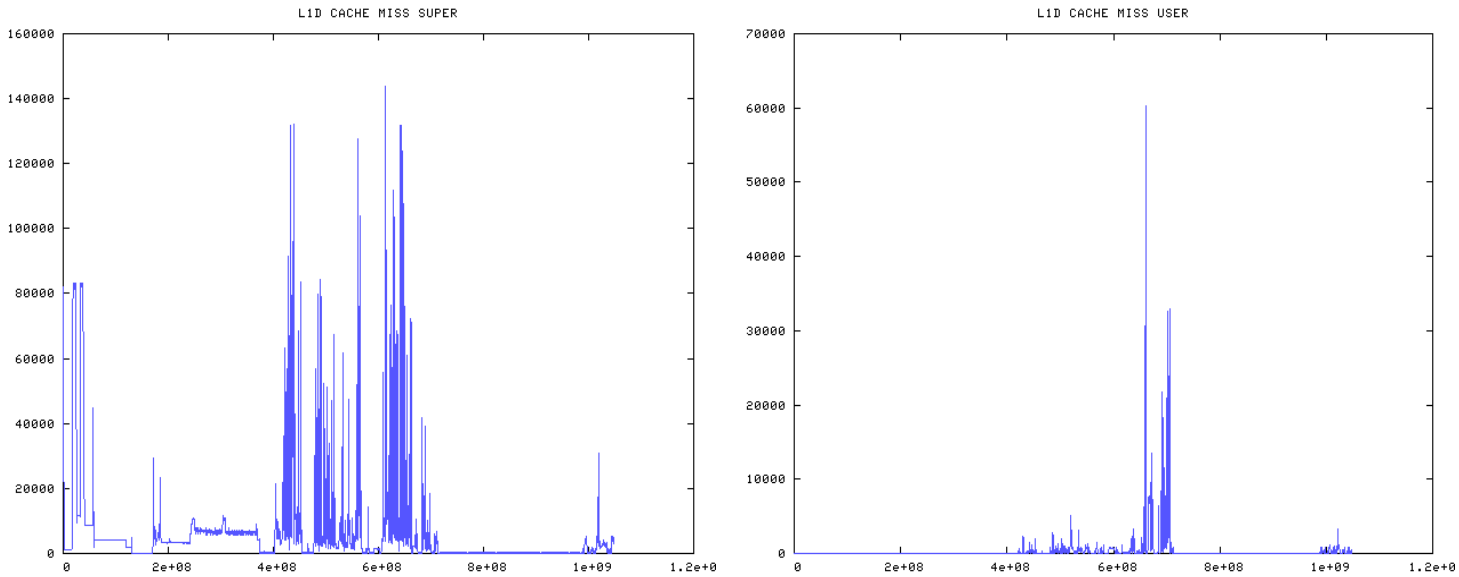

Figure 51: L1d Cache Miss Super (left) and L1d Cache Miss User (right) livegraphs

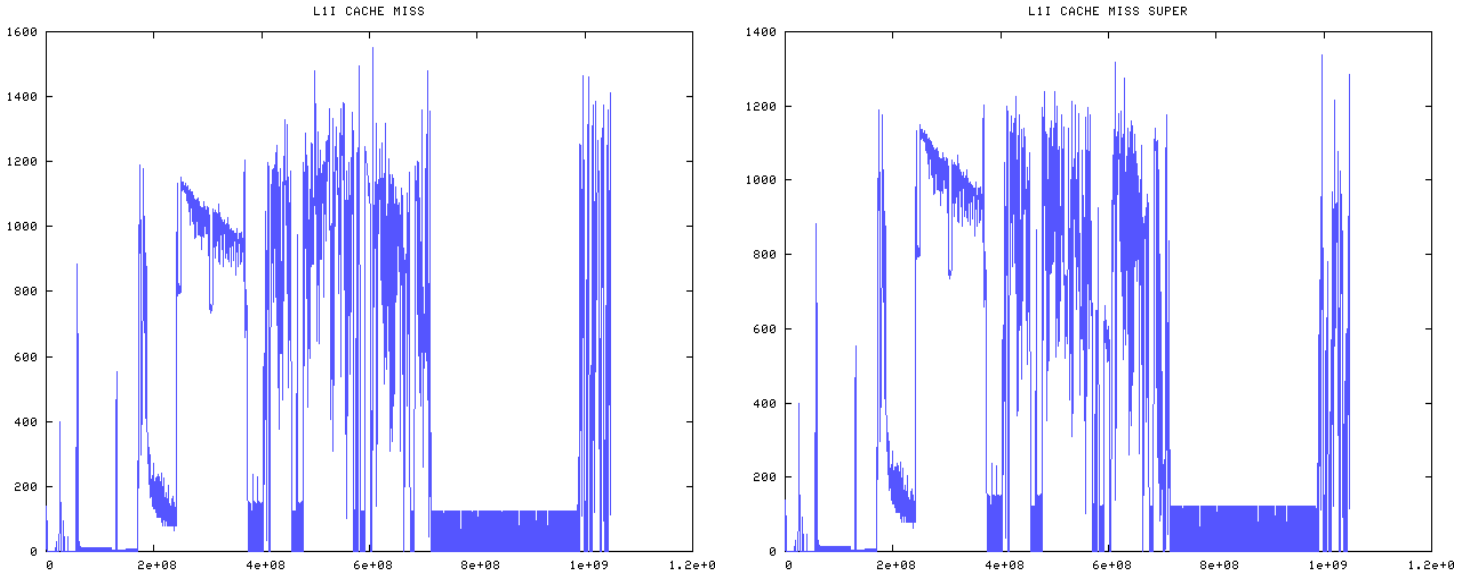

Figure 52: L1I Cache Miss (left) and L1I Cache Miss Super (right) livegraphs

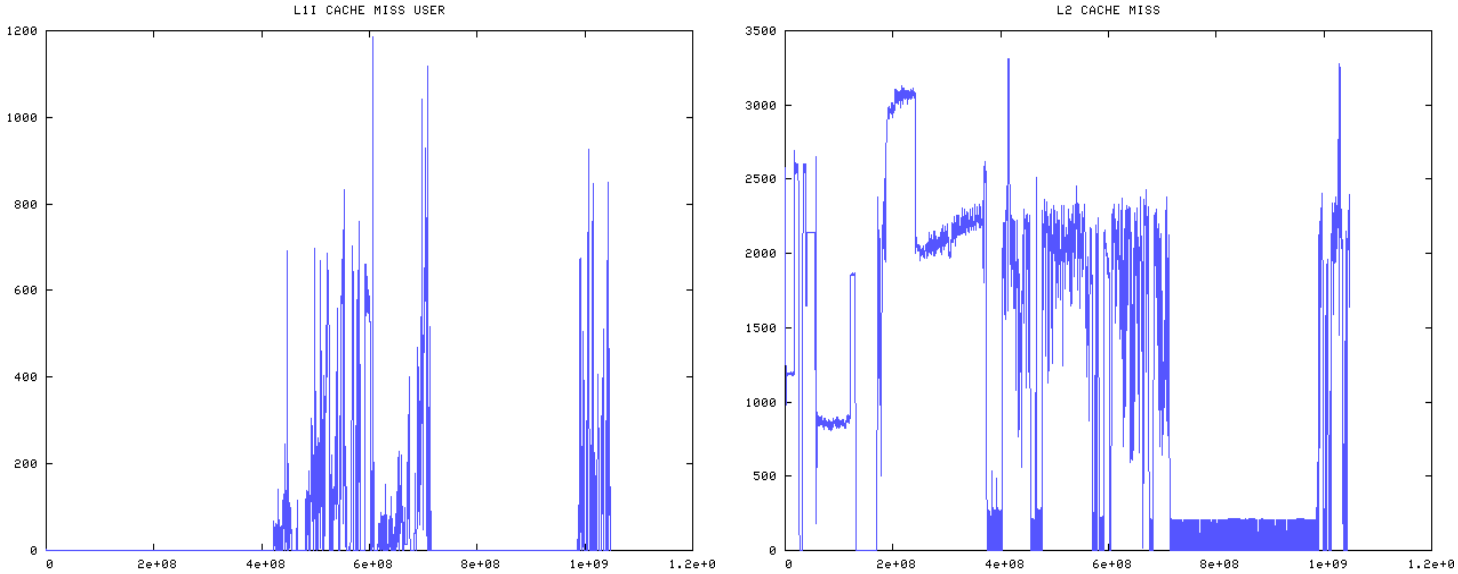

Figure 53: L1I Cache Miss User (left) and L2 Cache Miss (right) livegraphs

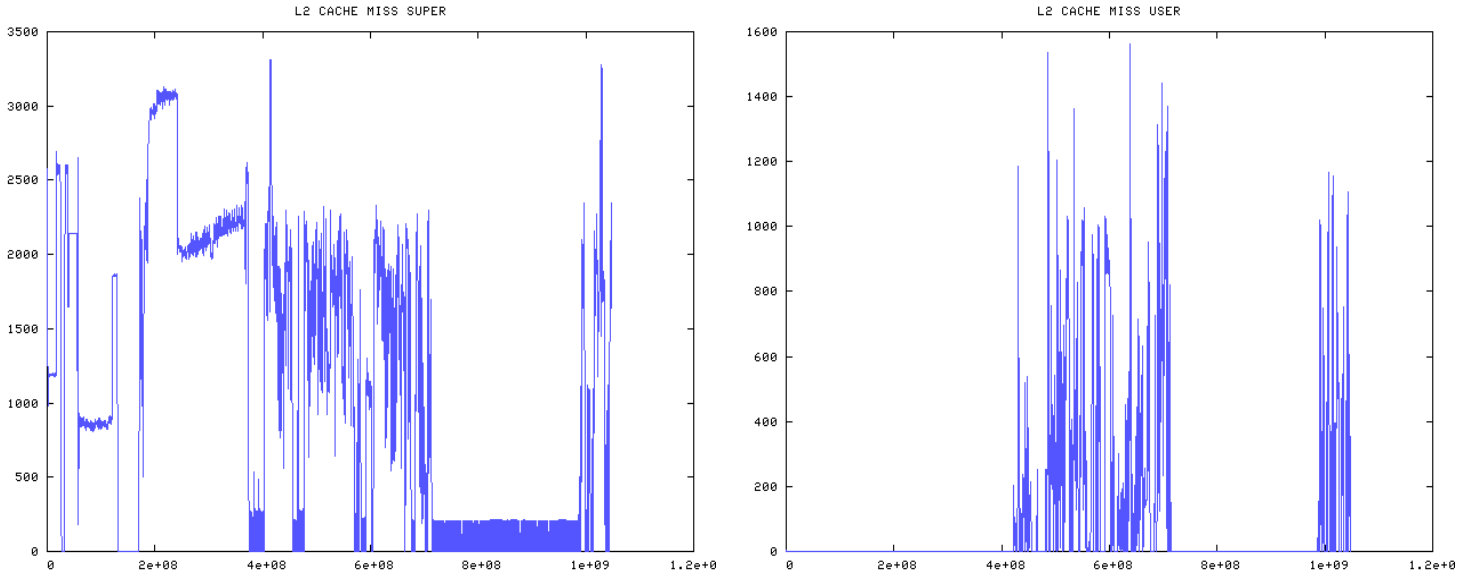

Figure 54: L2 Cache Miss Super (left) and L2 Cache Miss User (right) livegraphs

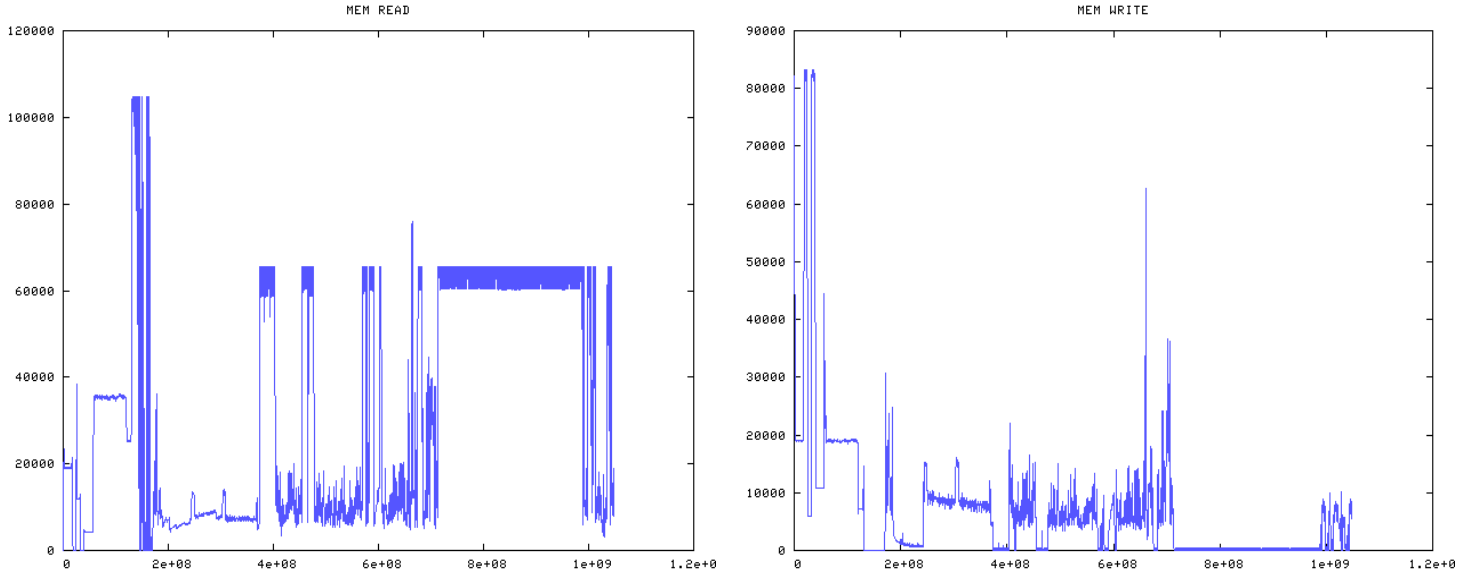

Figure 55: Memory-read (left) and Memory-Write (right) livegraphs

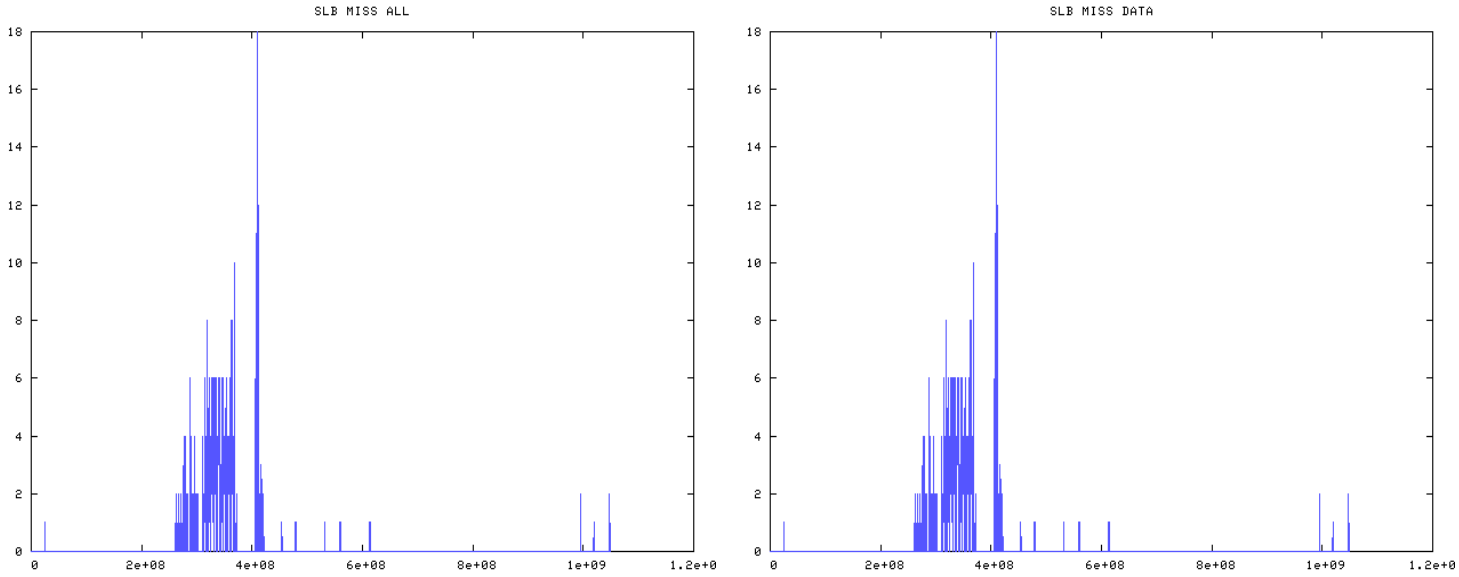

Figure 56: All SLB misses (left) and Data SLB Misses (right) livegraphs

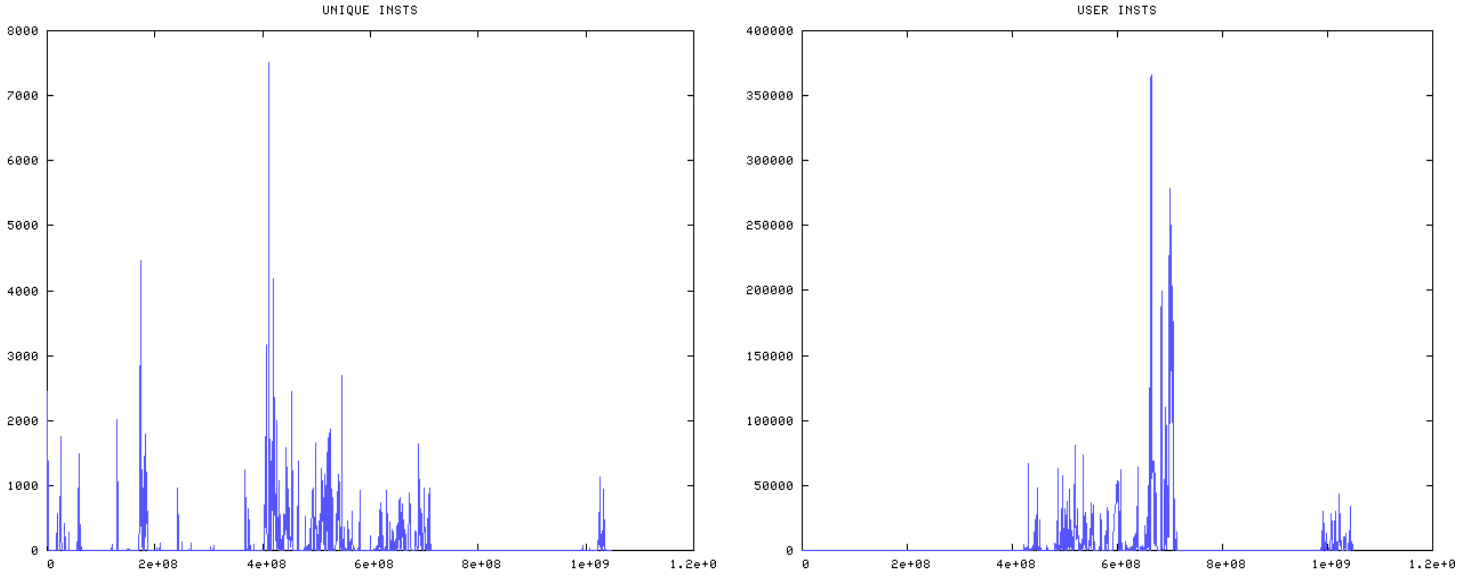

Figure 57: Unique Instructions (left) and User Instructions (right) livegraphs

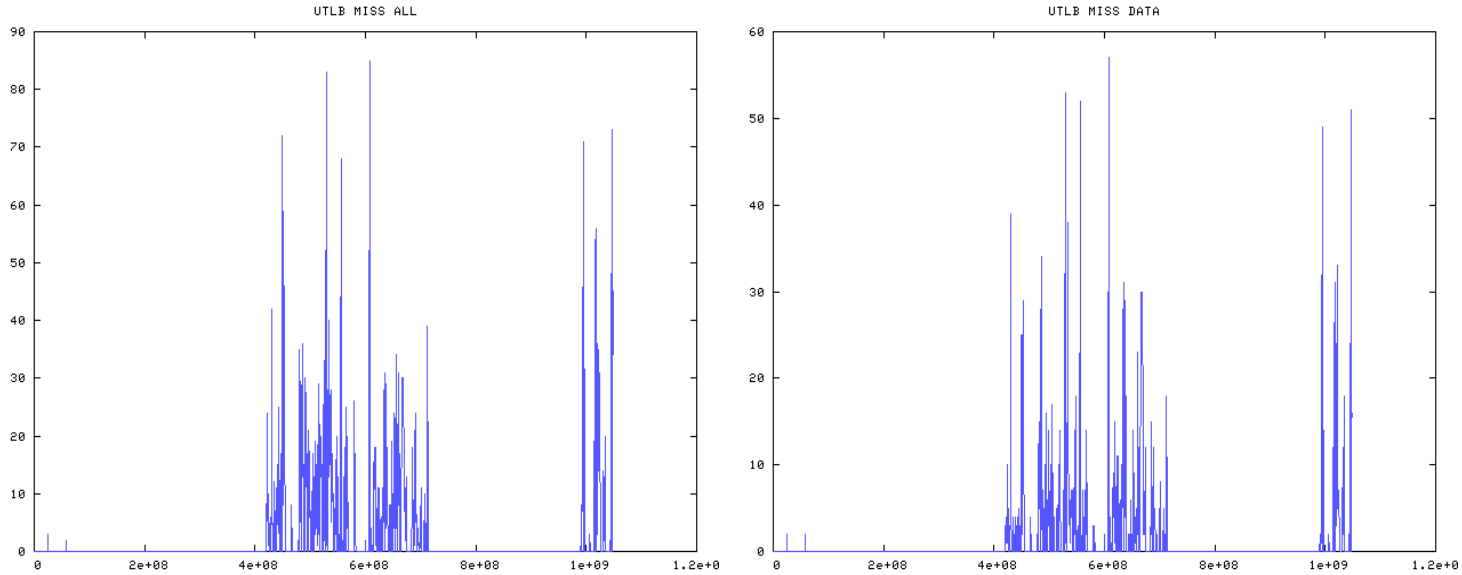

Figure 58: All User TLB Misses (left) and Data TLB Misses (right) livegraphs

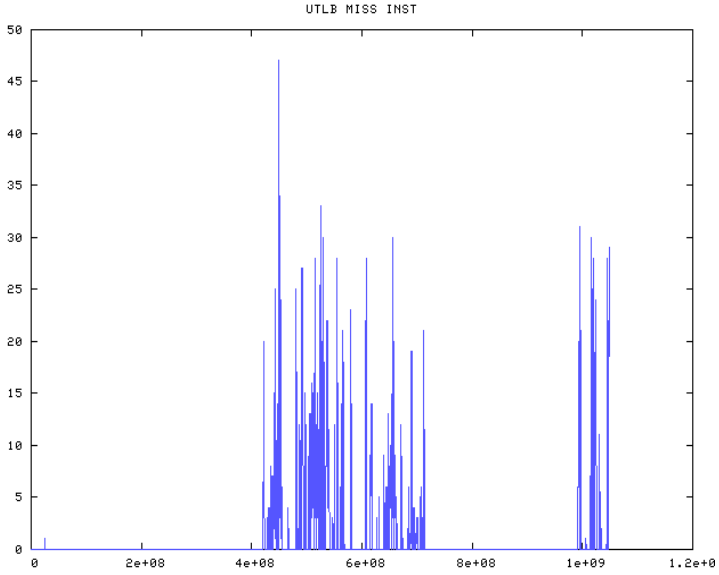

Figure 59: User TLB Instruction Miss livegraphs

B.4 GLIMMER

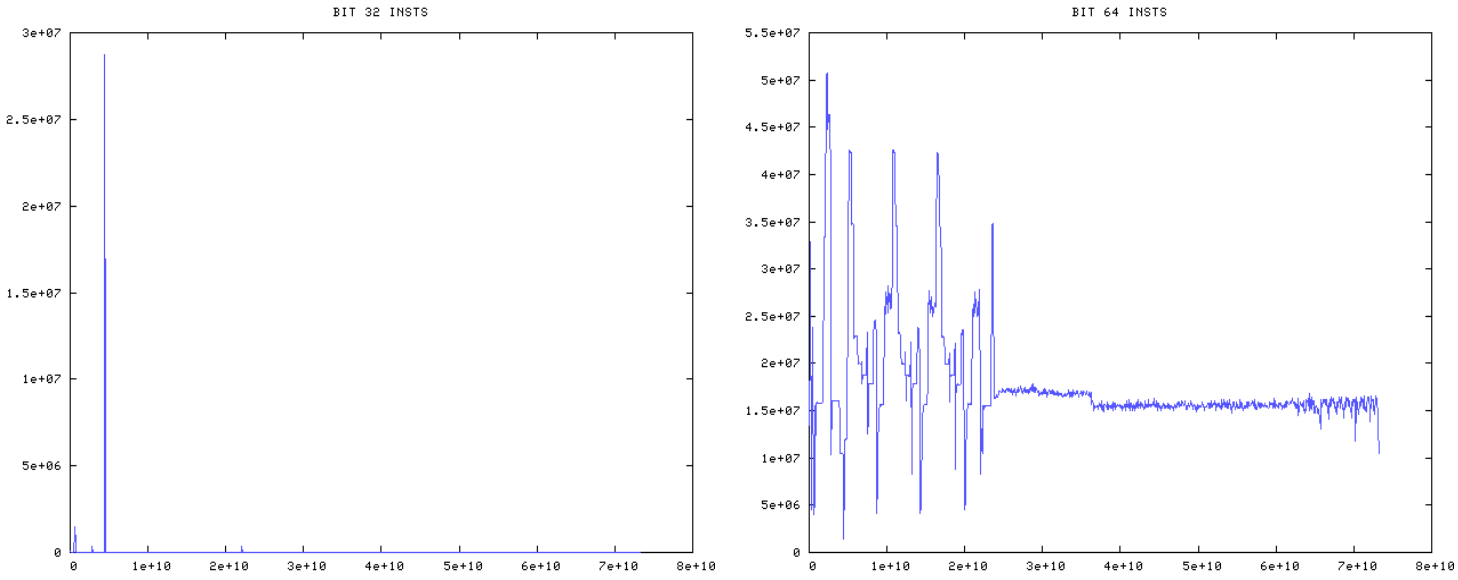

Figure 60: 32-bit (left) and 64-bit (right) instruction livegraphs

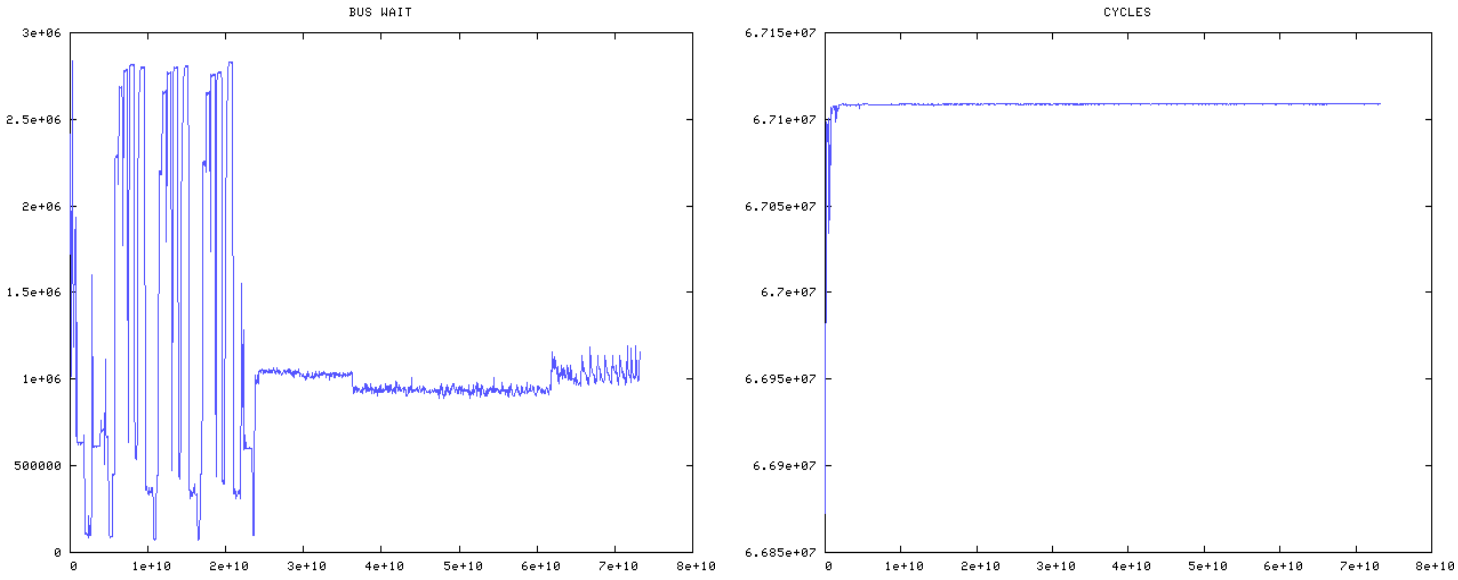

Figure 61: Bus-wait (left) and Cycles (right) livegraphs

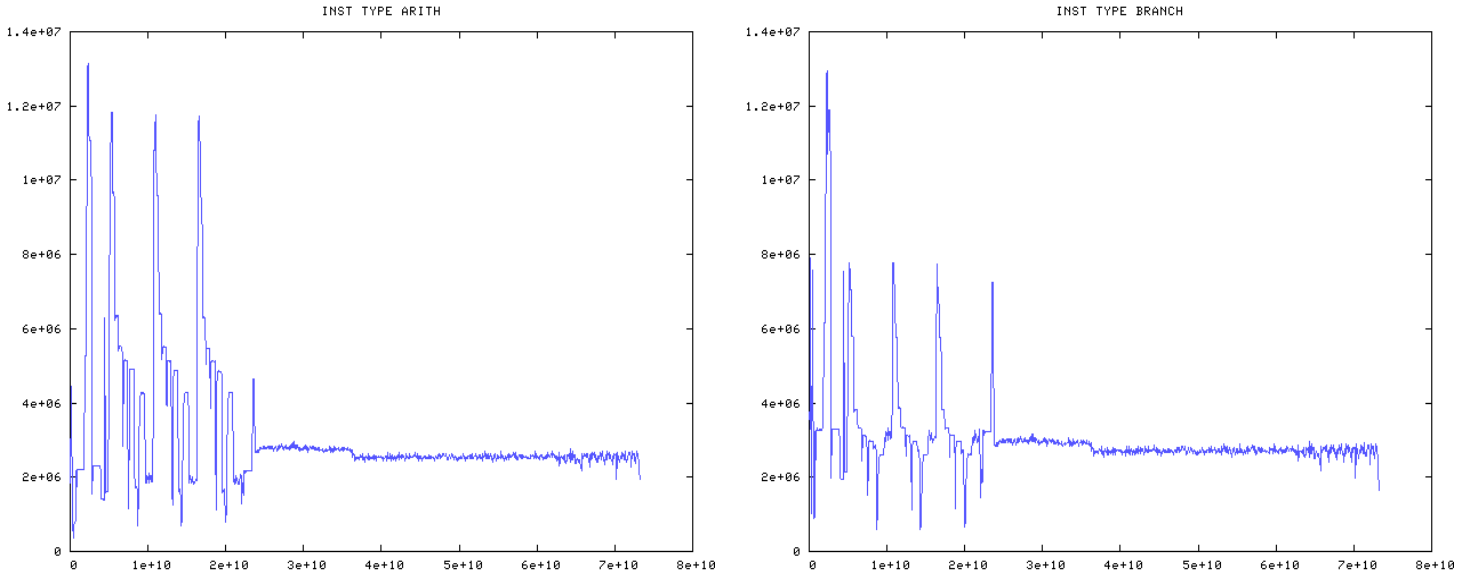

Figure 62: Arithmetic (left) and Branch (right) instructions livegraphs

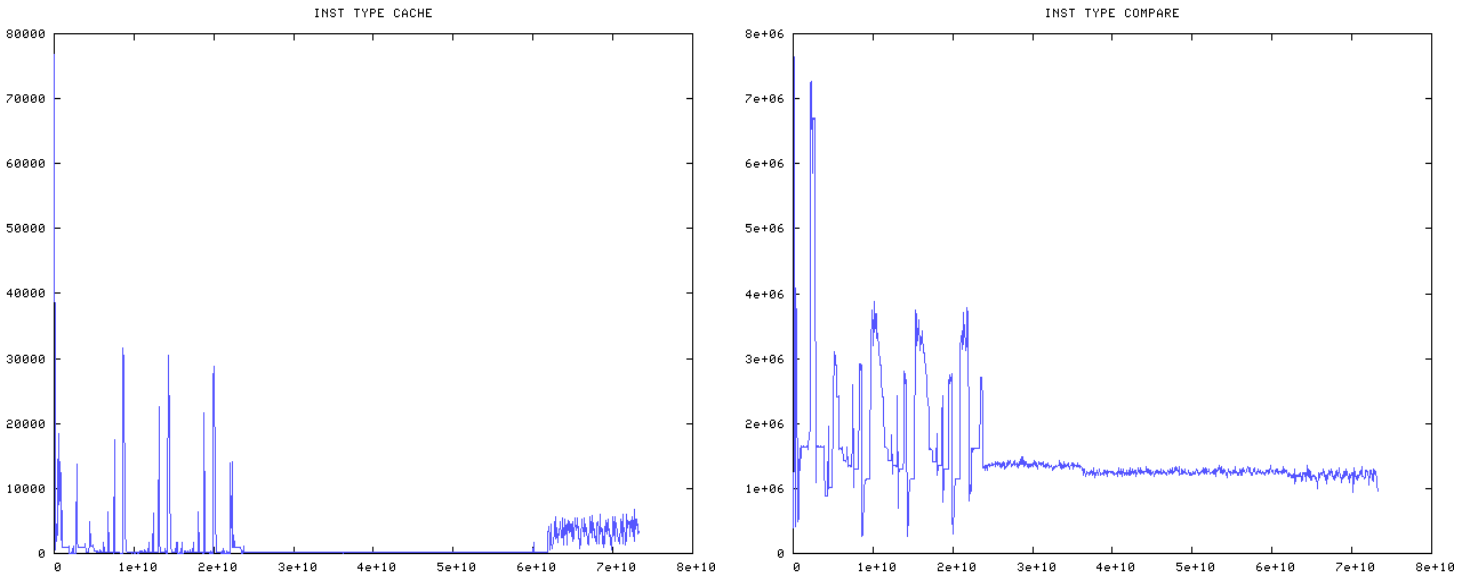

Figure 63: Cache (left) and Compare (right) instructions livegraphs

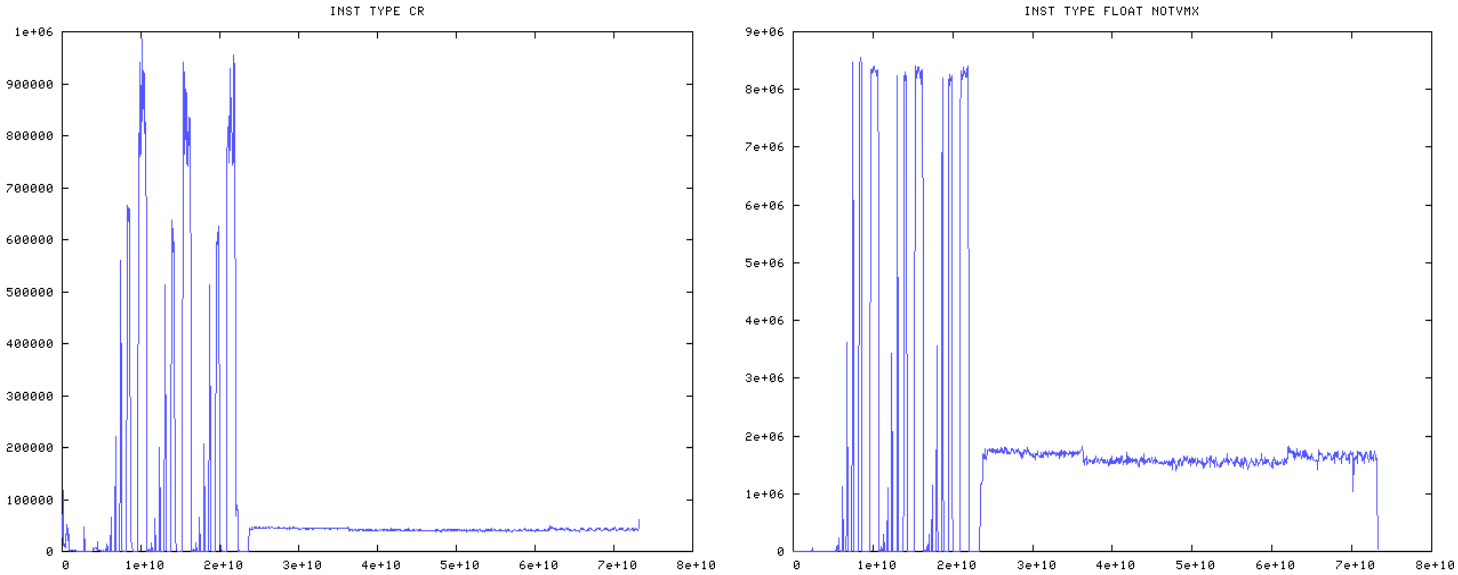

Figure 64: CR (left) and Not VMX-Float (right) instructions livegraphs

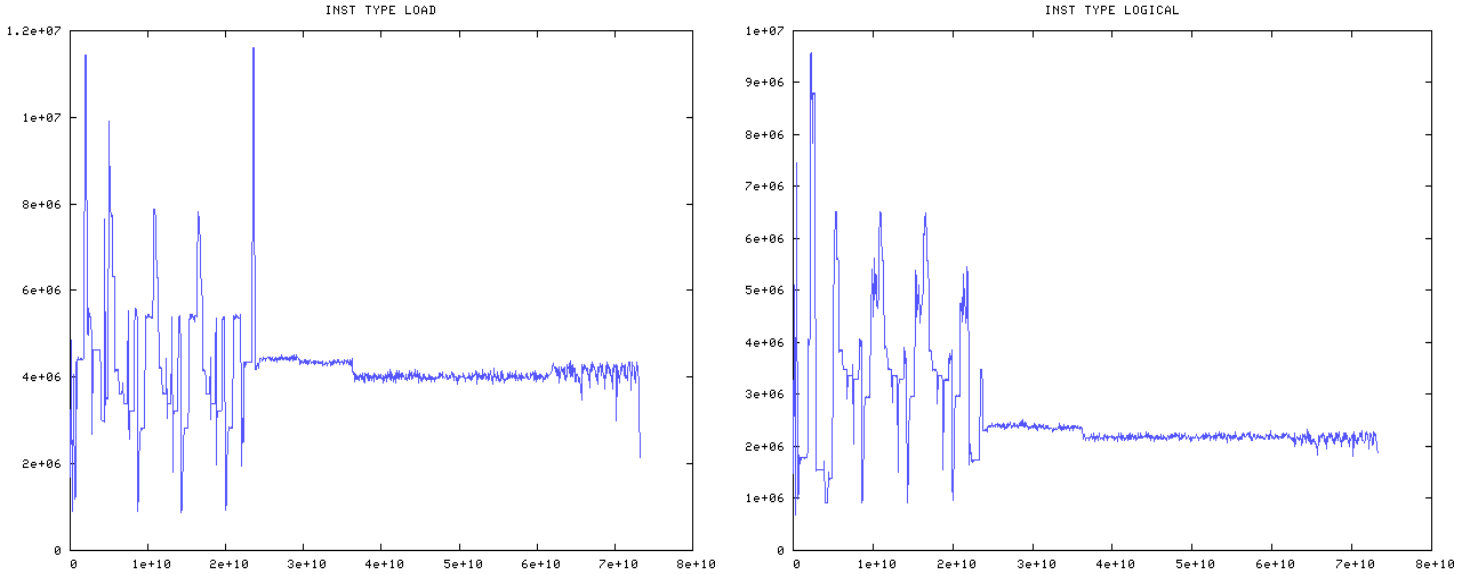

Figure 65: Load (left) and Logical (right) instructions livegraphs

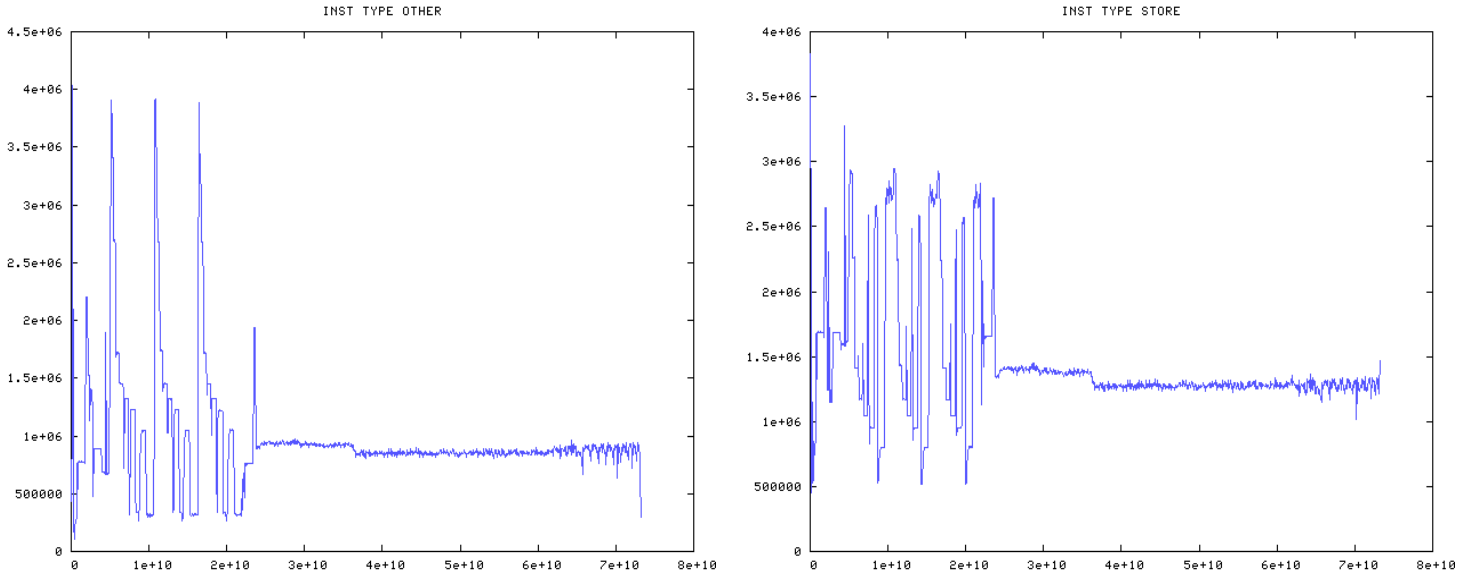

Figure 66: Other (left) and Store (right) instructions livegraphs

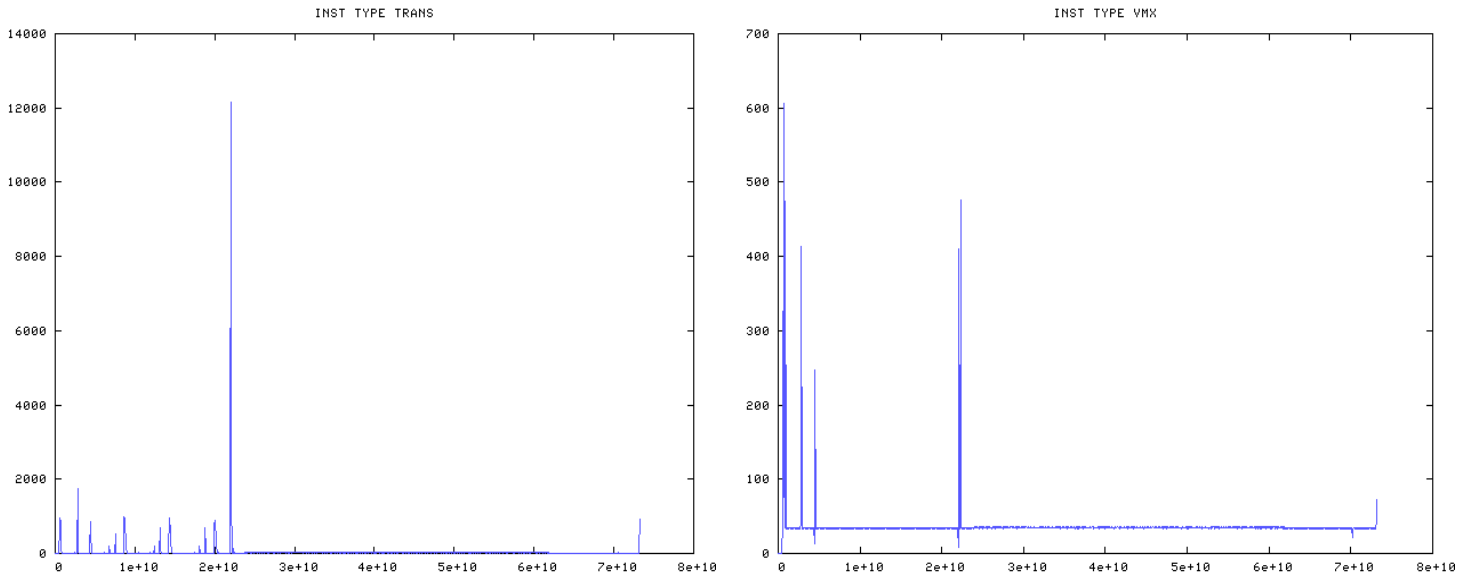

Figure 67: Trans (left) and VMX (right) instructions livegraphs

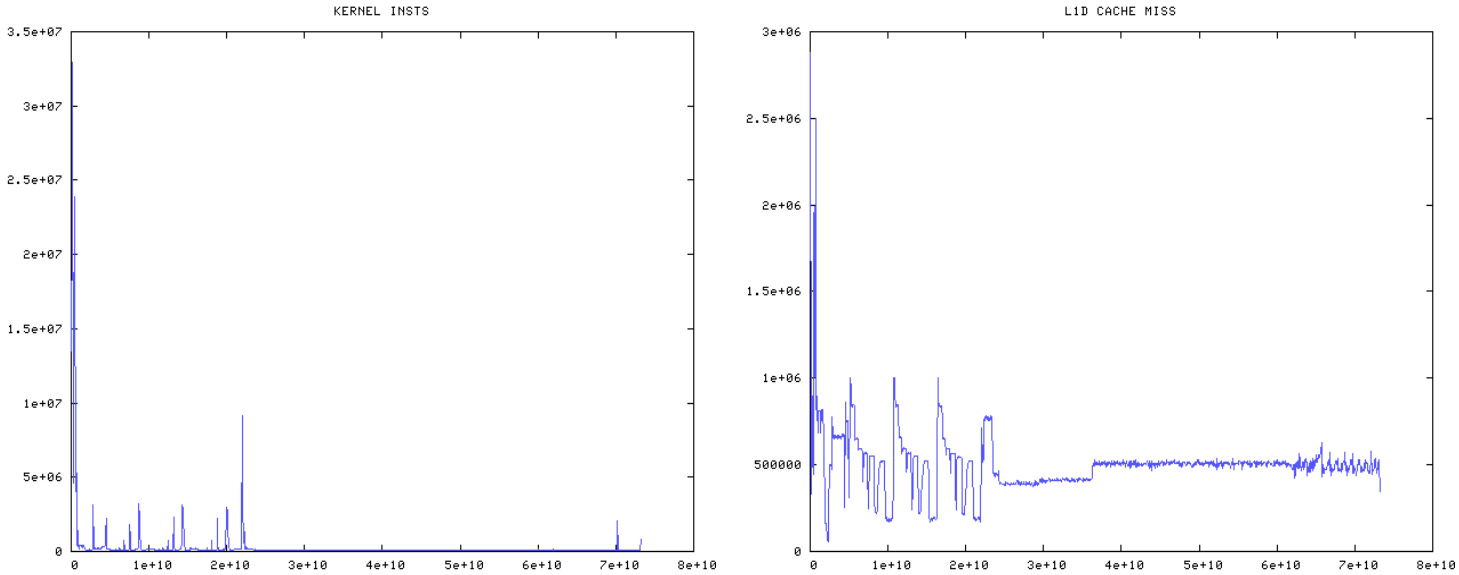

Figure 68: Kernel instructions (left) and L1D Cache Miss (right) livegraphs

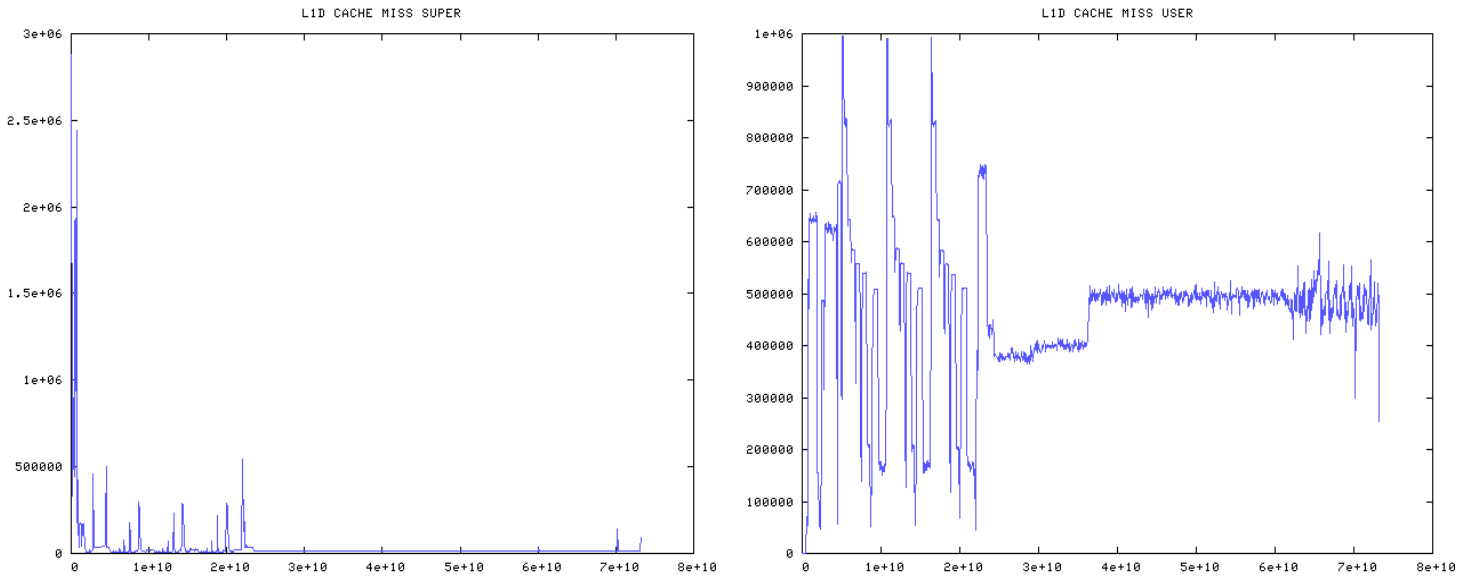

Figure 69: L1d Cache Miss Super (left) and L1d Cache Miss User (right) livegraphs

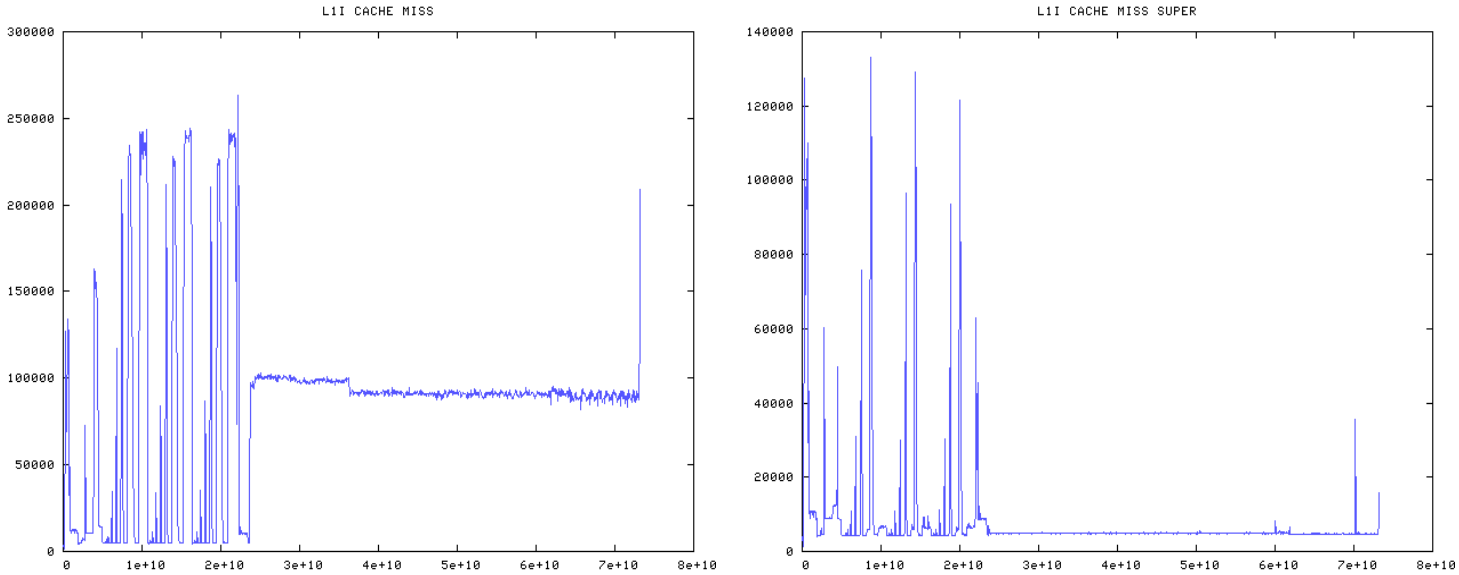

Figure 70: L1I Cache Miss (left) and L1I Cache Miss Super (right) livegraphs

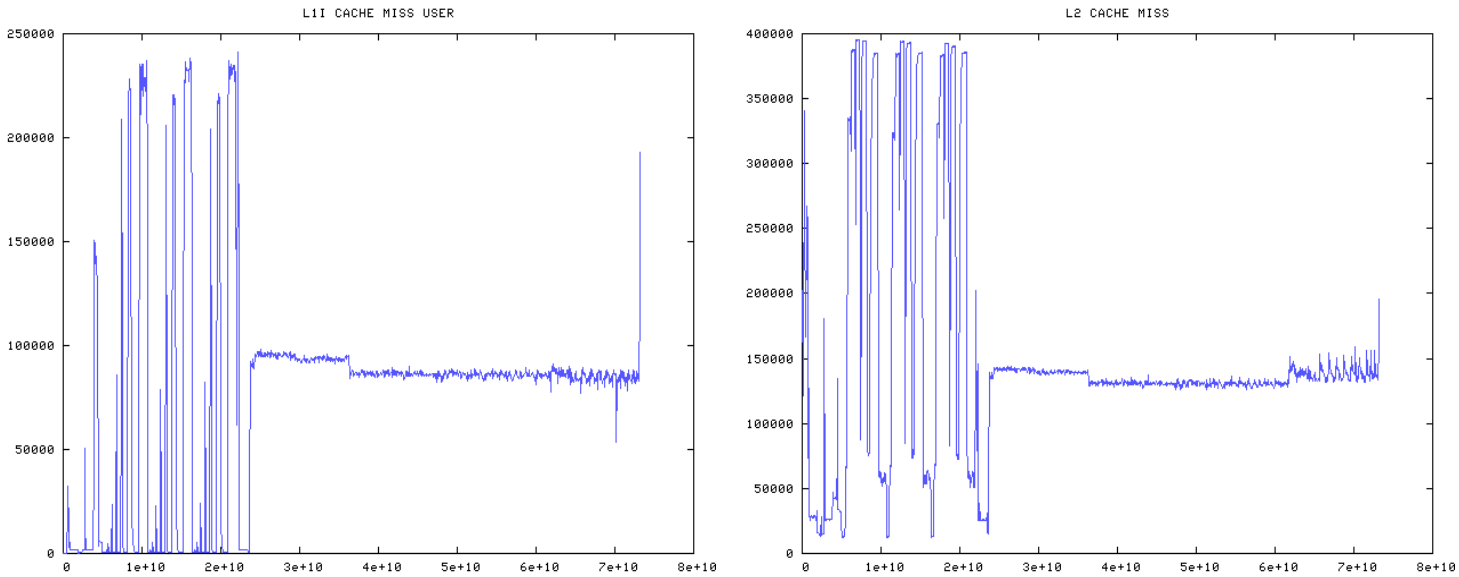

Figure 71: L1I Cache Miss User (left) and L2 Cache Miss (right) livegraphs

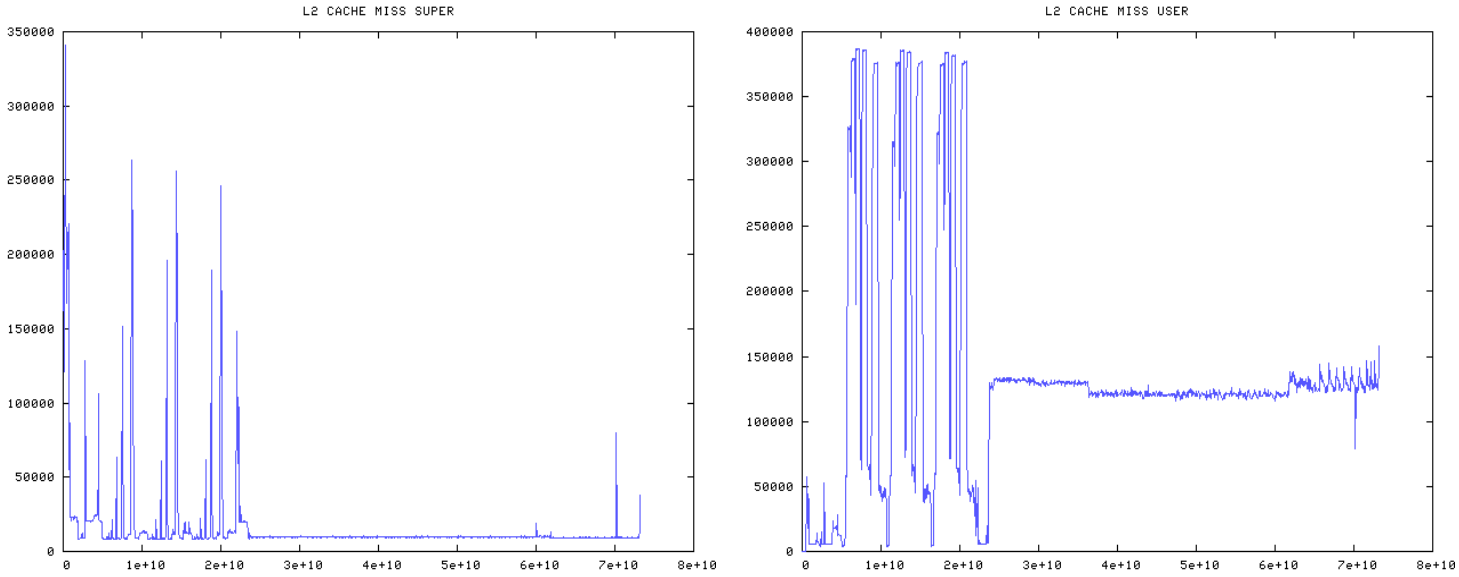

Figure 72: L2 Cache Miss Super (left) and L2 Cache Miss User (right) livegraphs

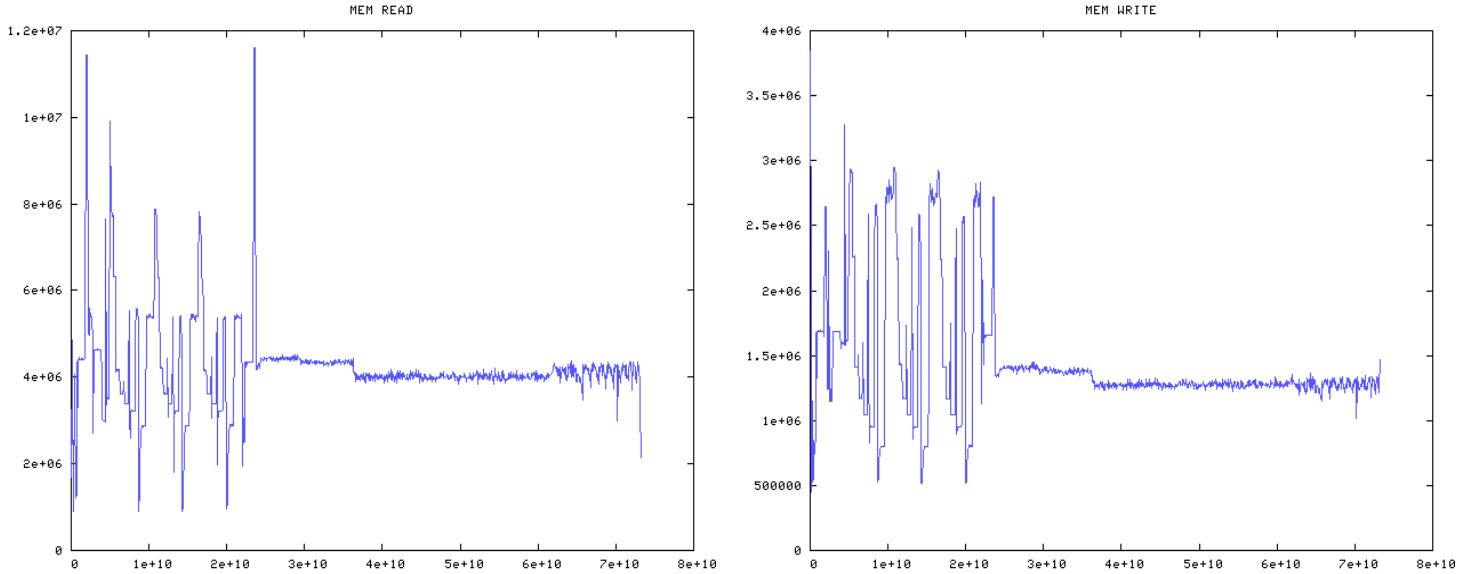

Figure 73: Memory-read (left) and Memory-Write (right) livegraphs

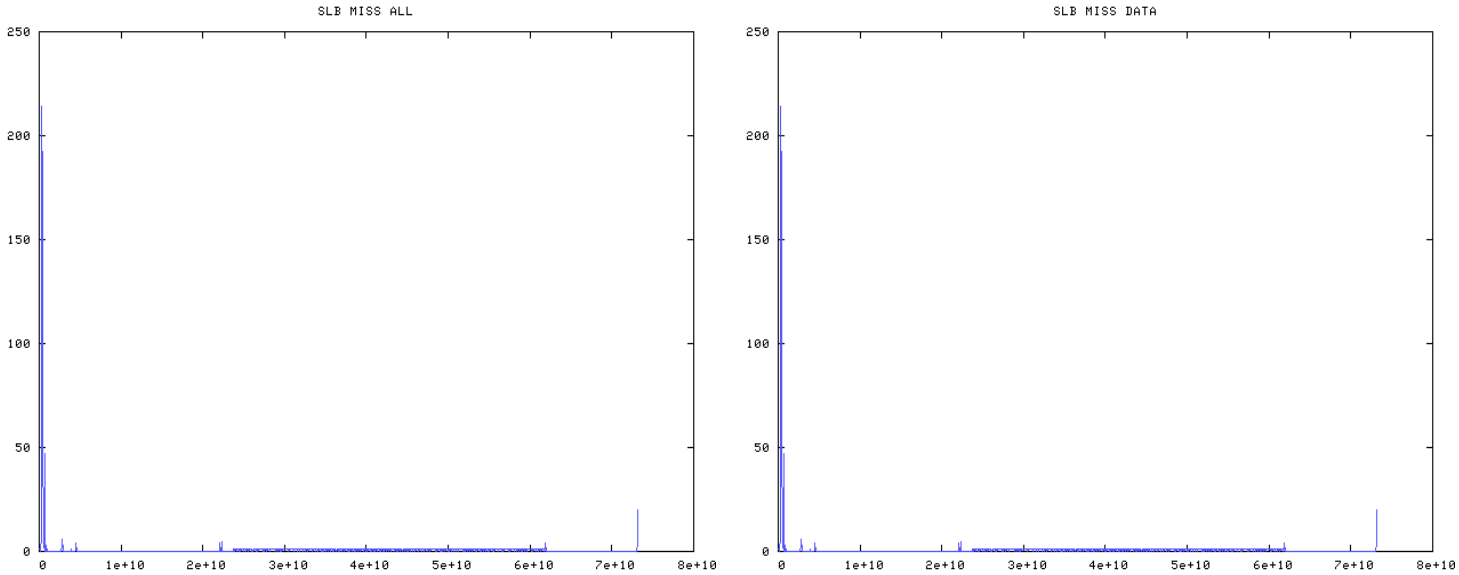

Figure 74: All SLB misses (left) and Data SLB Misses (right) livegraphs

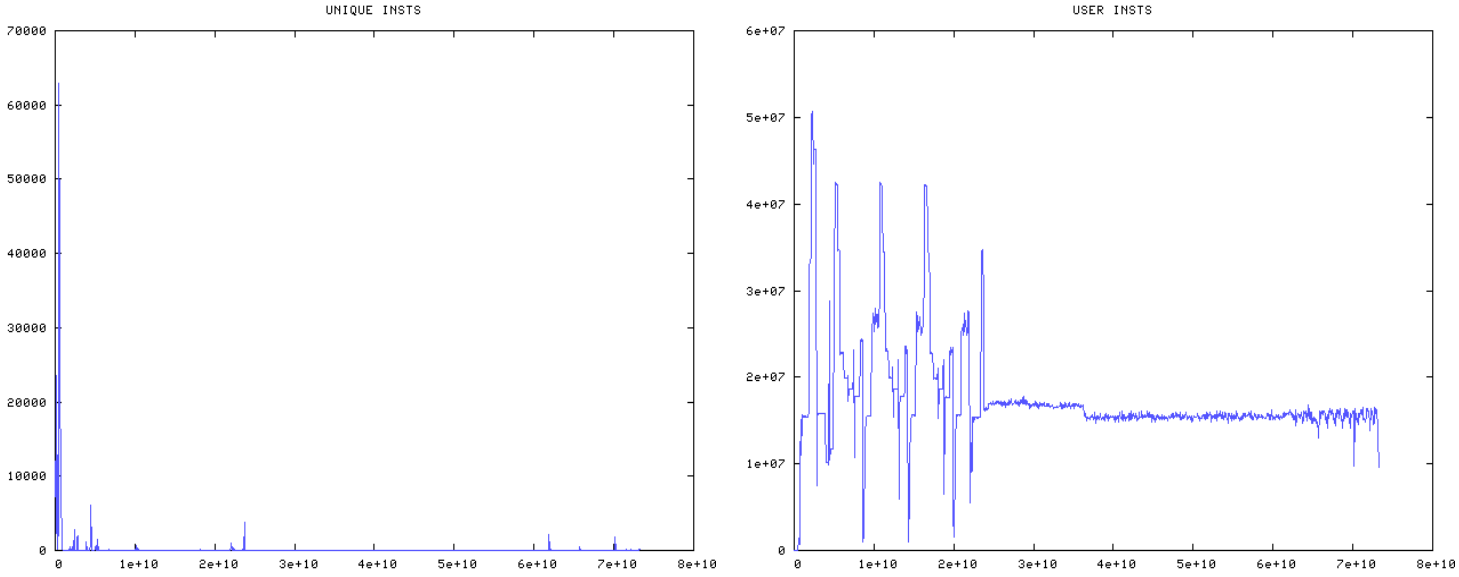

Figure 75: Unique Instructions (left) and User Instructions (right) livegraphs

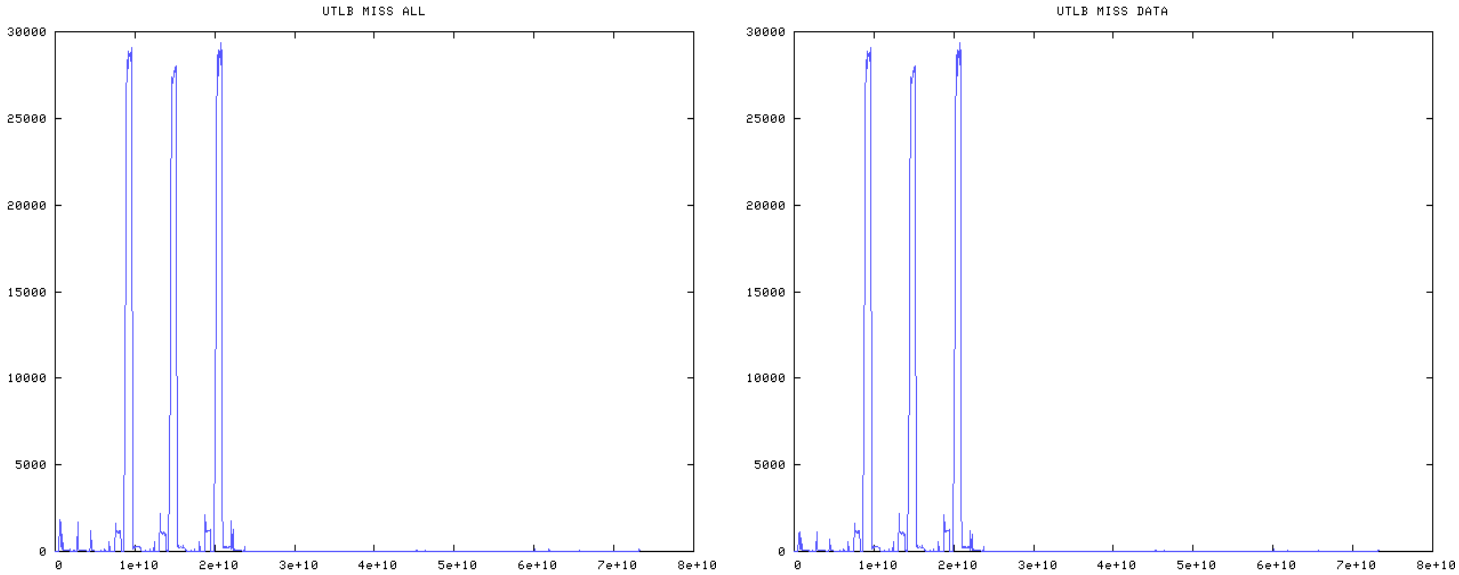

Figure 76: All User TLB Misses (left) and Data TLB Misses (right) livegraphs

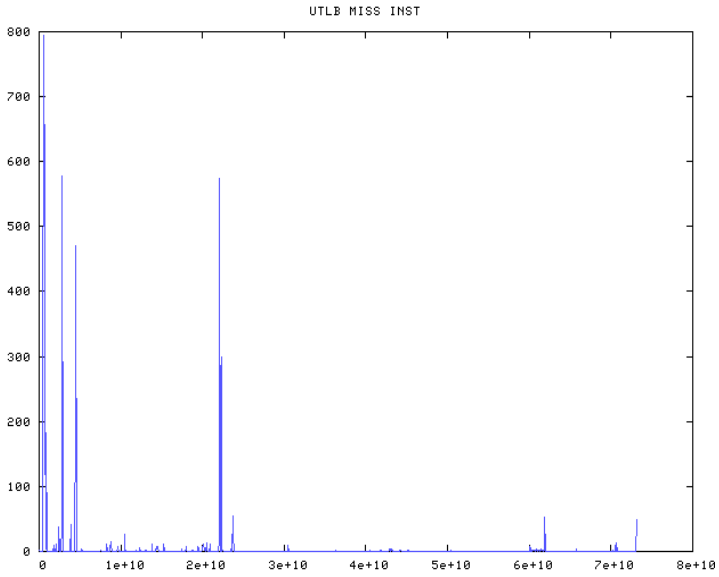

Figure 77: User TLB Instruction Miss livegraphs

B.5 GRAPPA

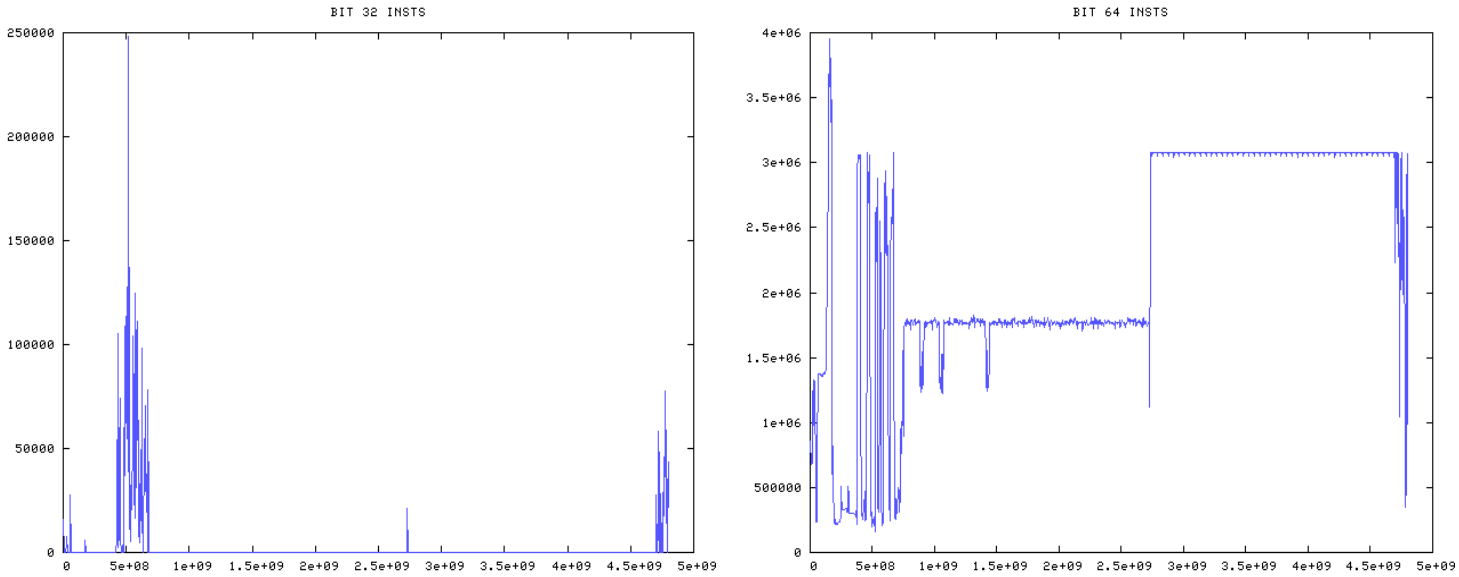

Figure 78: 32-bit (left) and 64-bit (right) instruction livegraphs

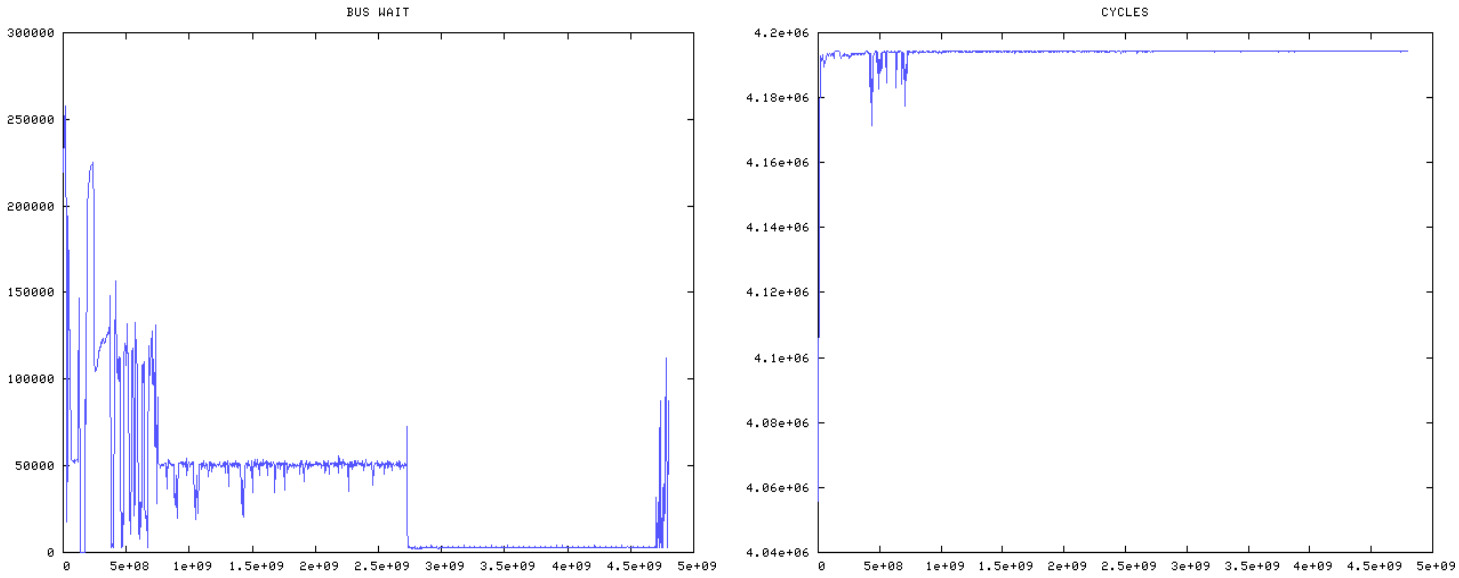

Figure 79: Bus-wait (left) and Cycles (right) livegraphs

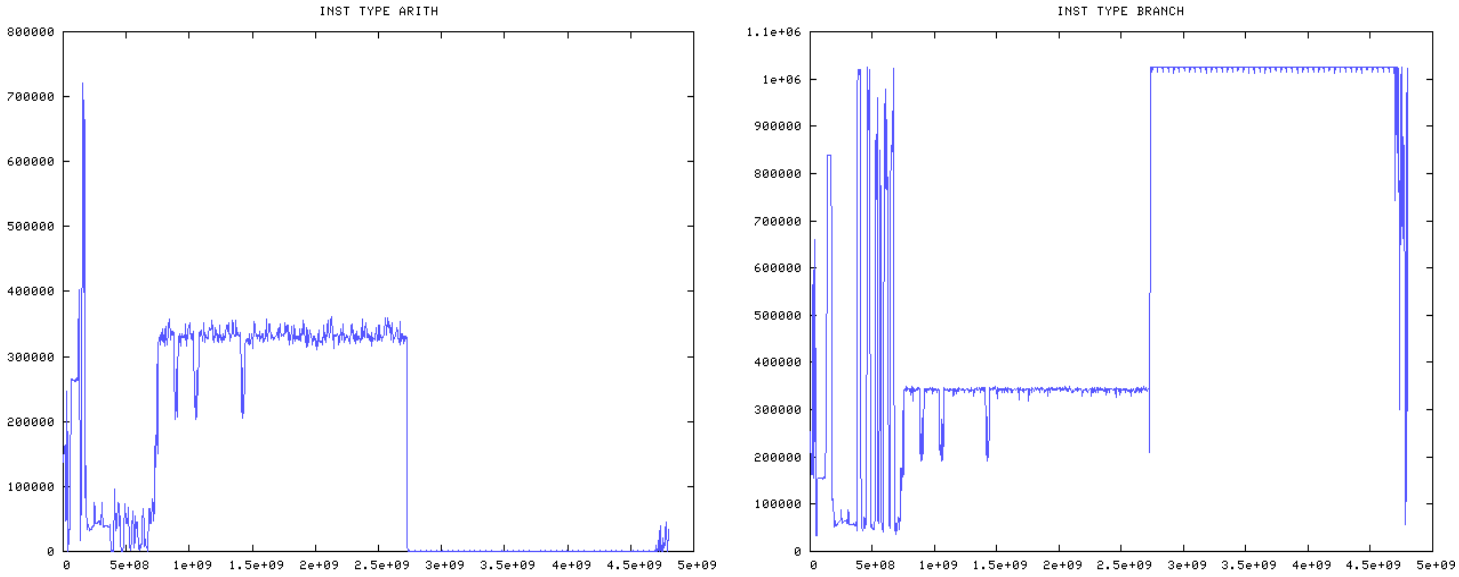

Figure 80: Arithmetic (left) and Branch (right) instructions livegraphs

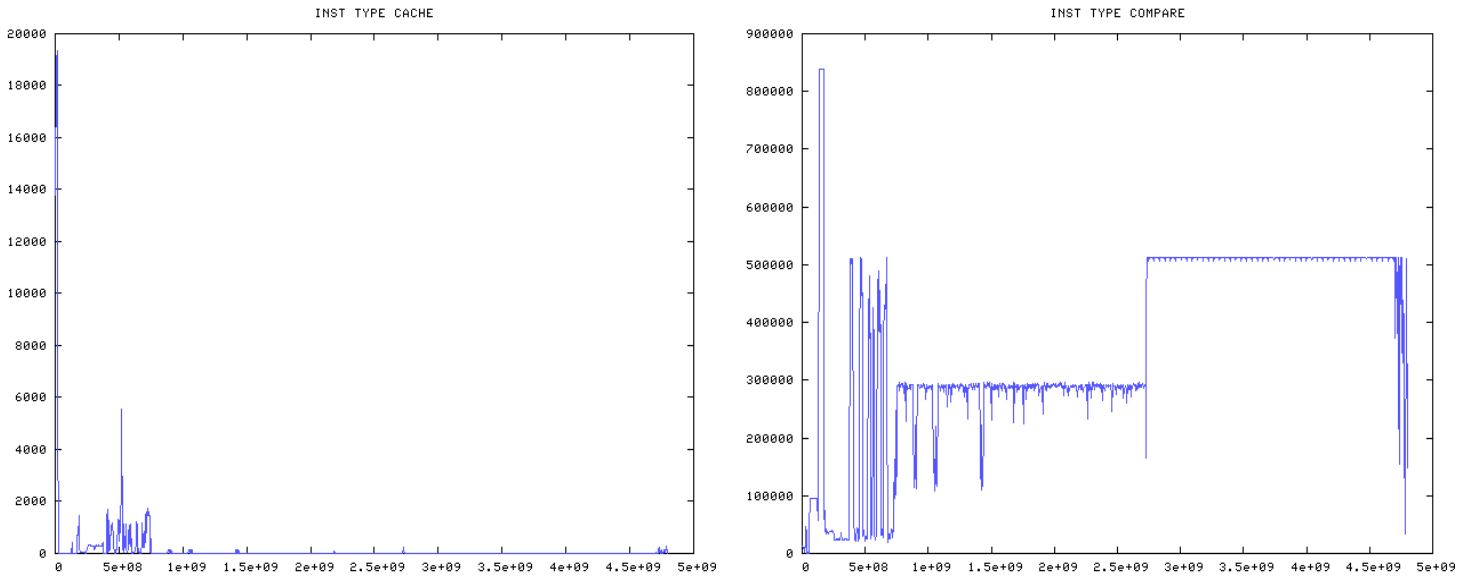

Figure 81: Cache (left) and Compare (right) instructions livegraphs

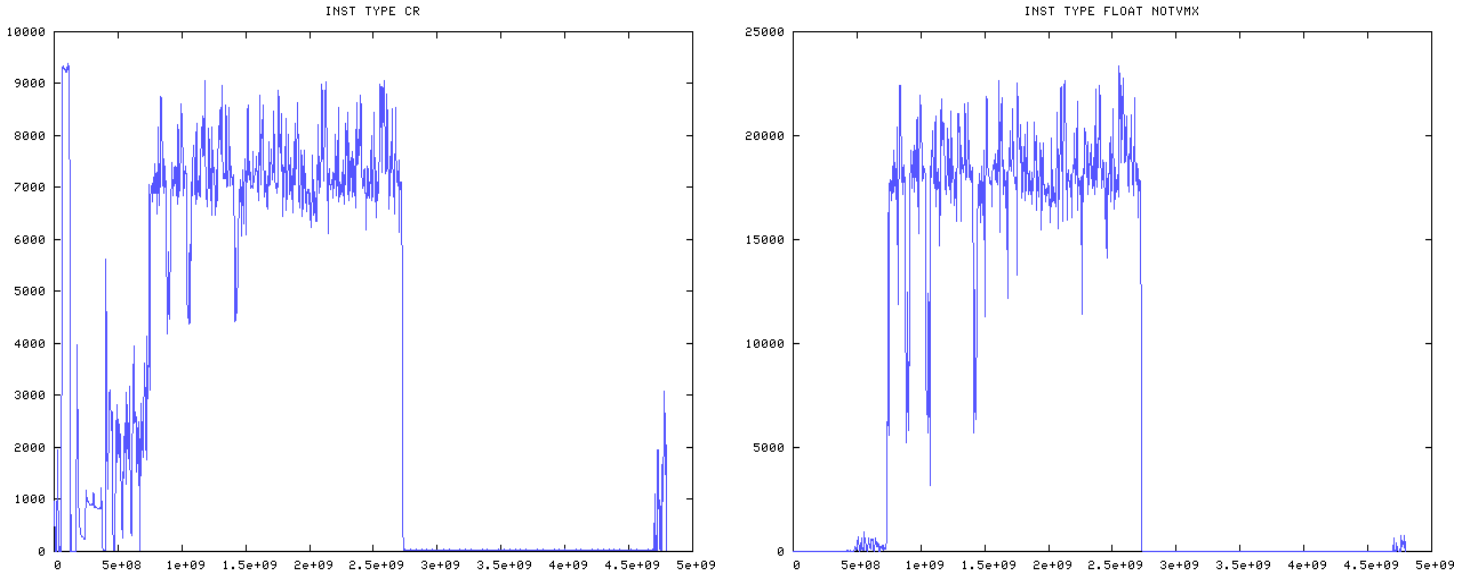

Figure 82: CR (left) and Not VMX-Float (right) instructions livegraphs

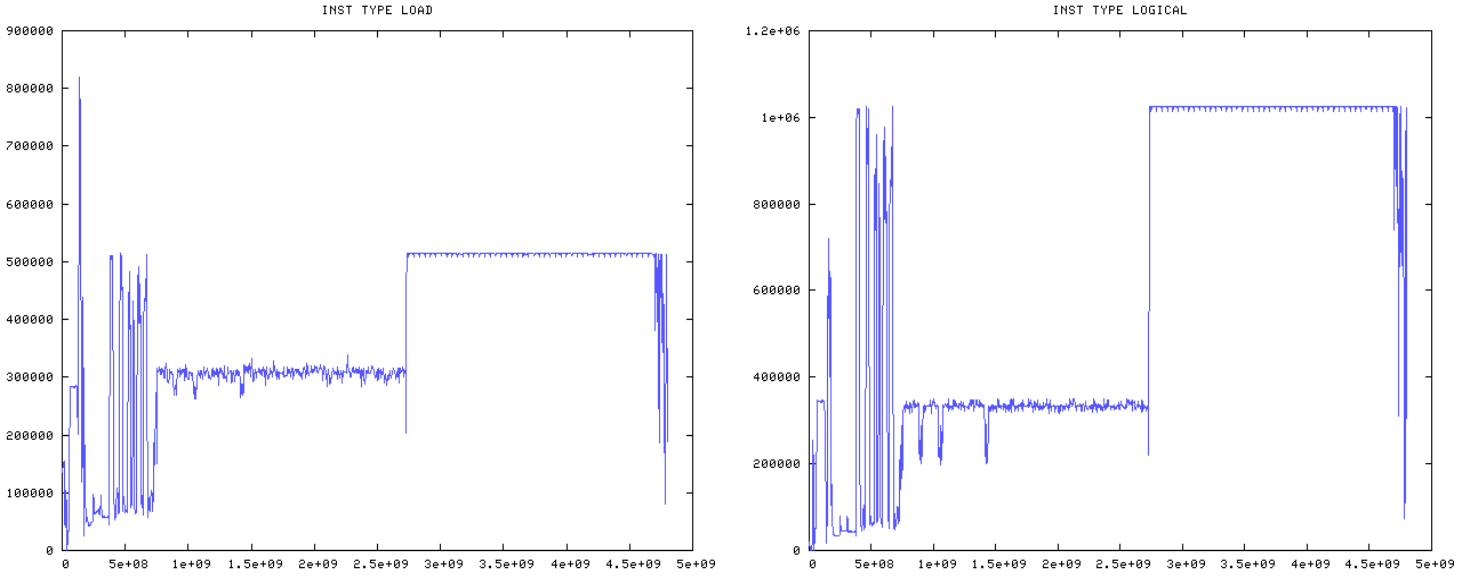

Figure 83: Load (left) and Logical (right) instructions livegraphs

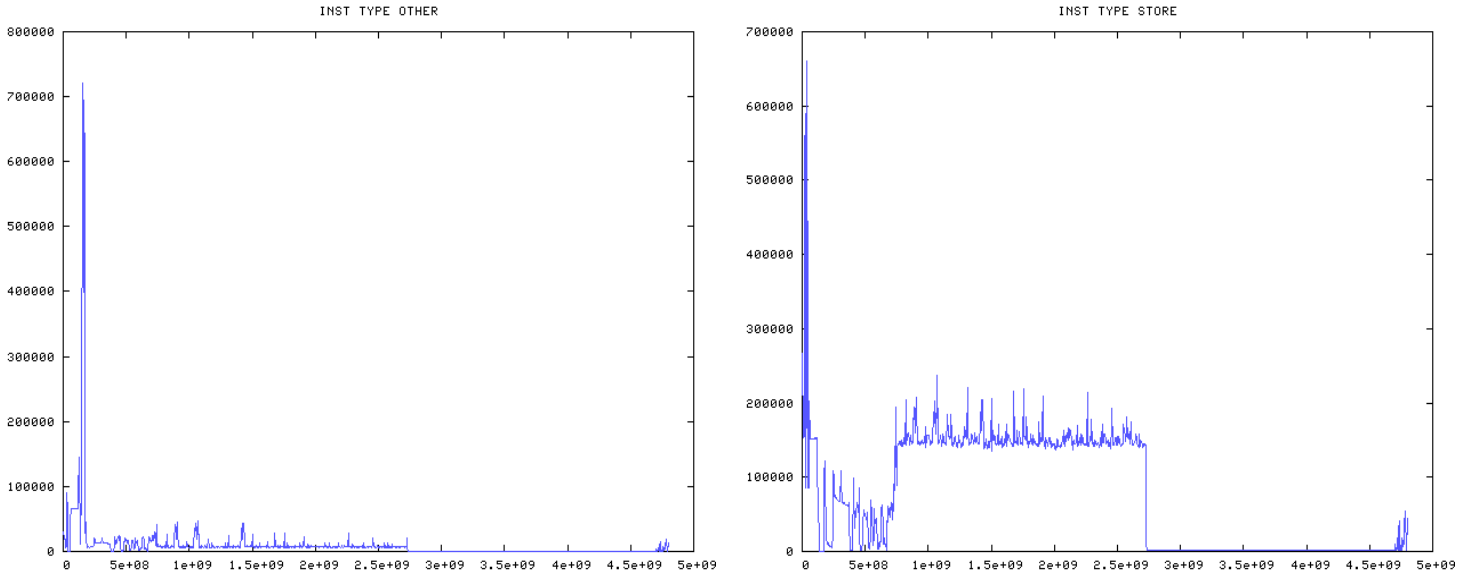

Figure 84: Other (left) and Store (right) instructions livegraphs

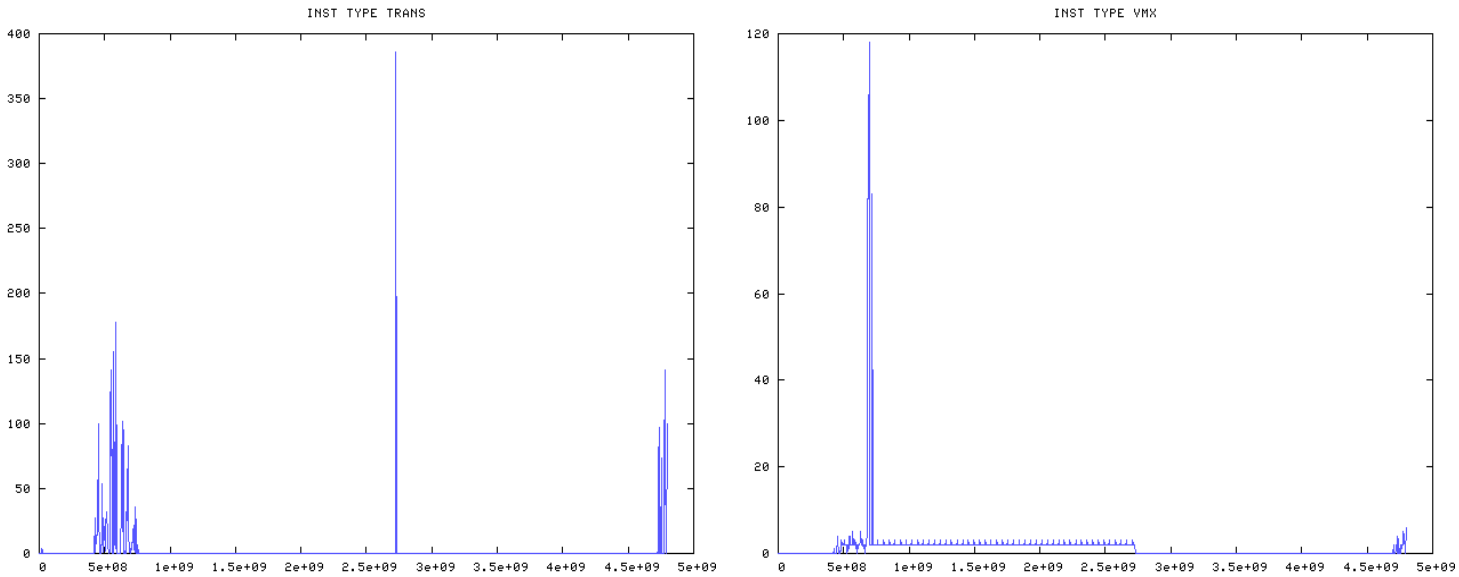

Figure 85: Trans (left) and VMX (right) instructions livegraphs

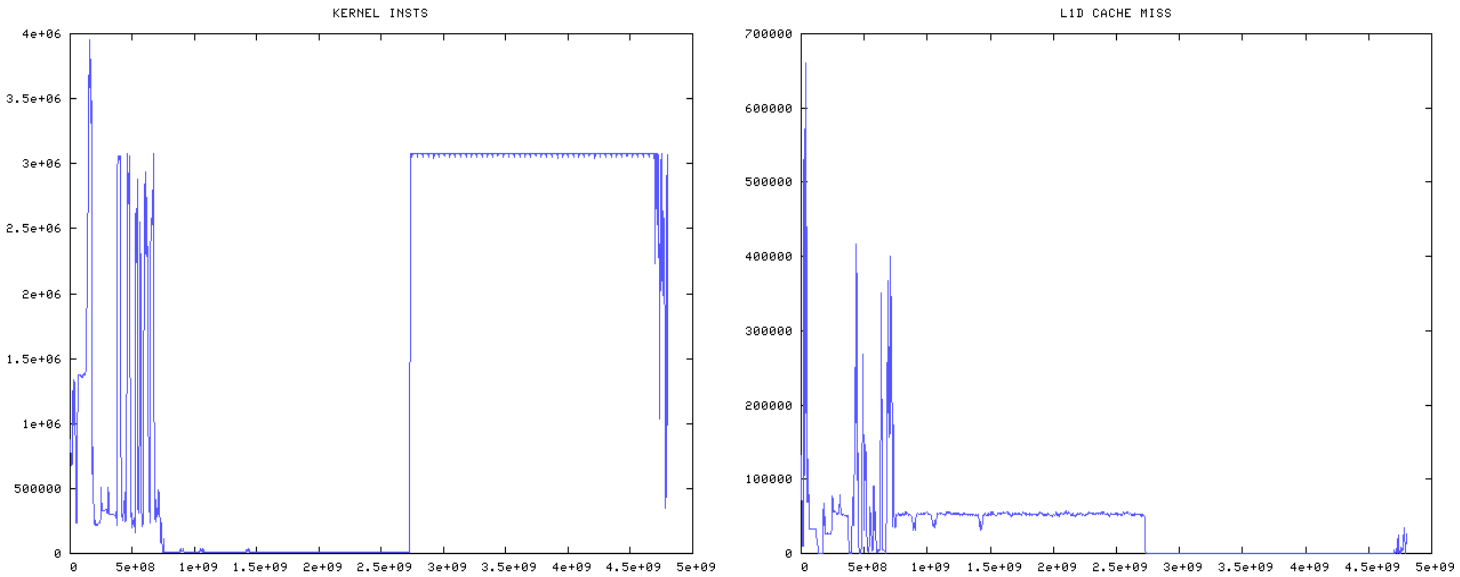

Figure 86: Kernel instructions (left) and L1D Cache Miss (right) livegraphs

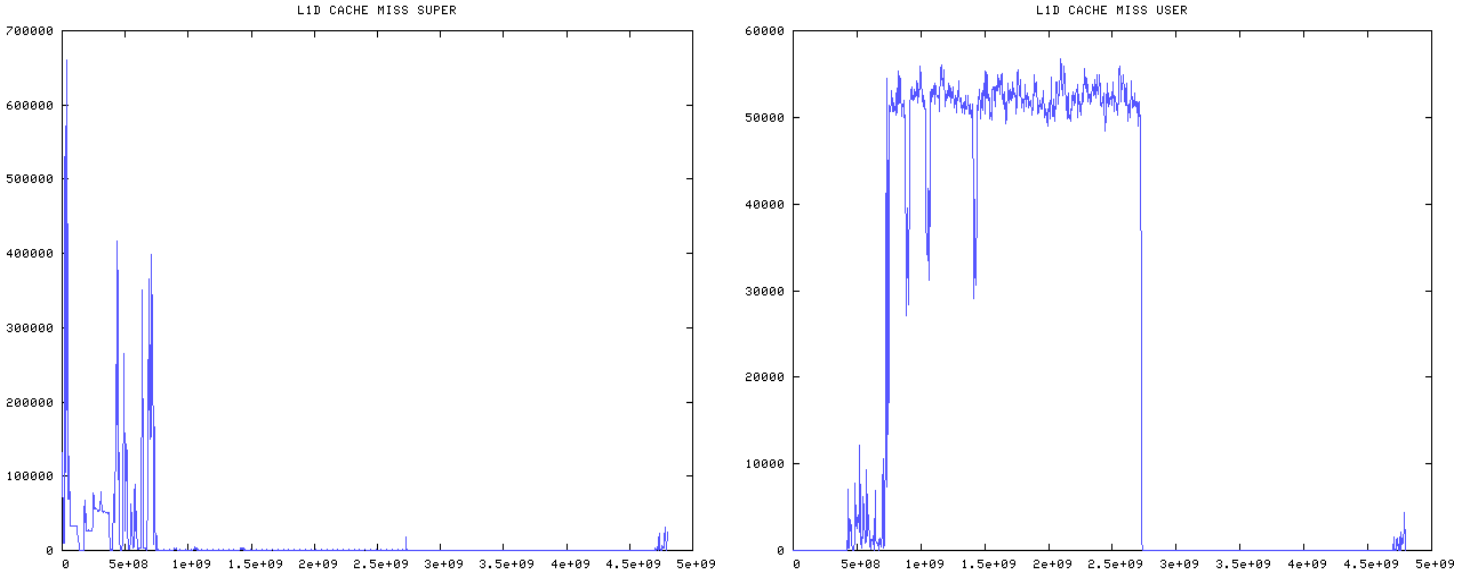

Figure 87: L1d Cache Miss Super (left) and L1d Cache Miss User (right) livegraphs

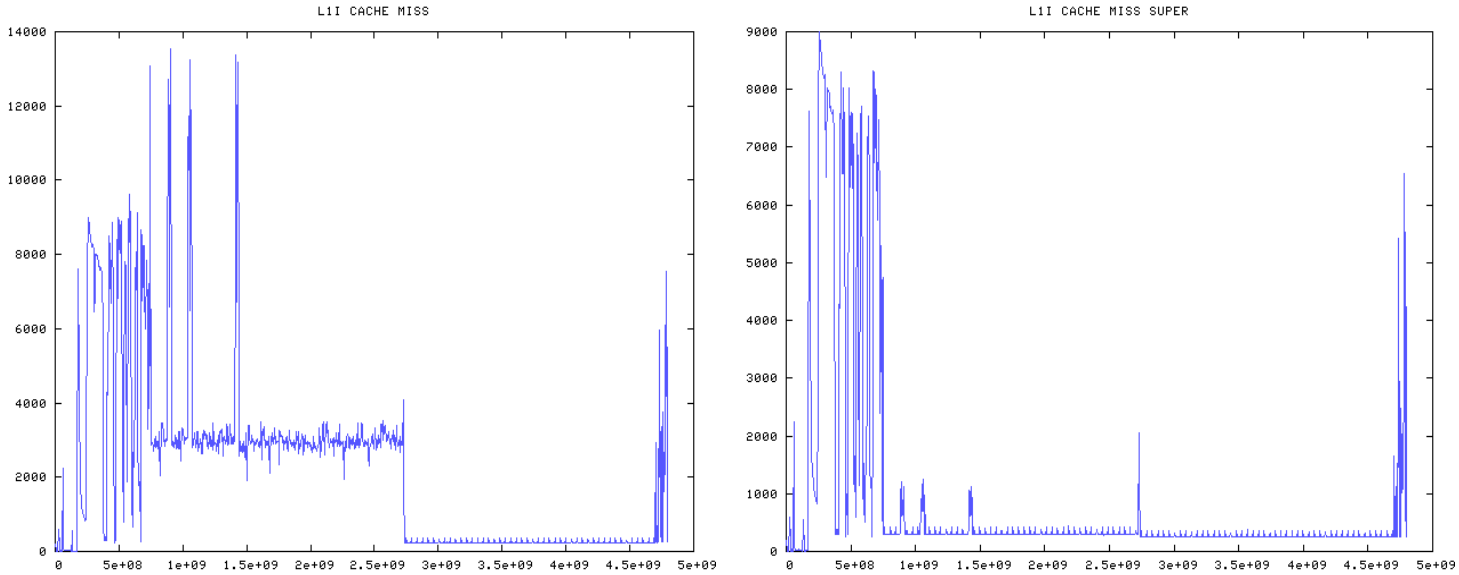

Figure 88: L1I Cache Miss (left) and L1I Cache Miss Super (right) livegraphs

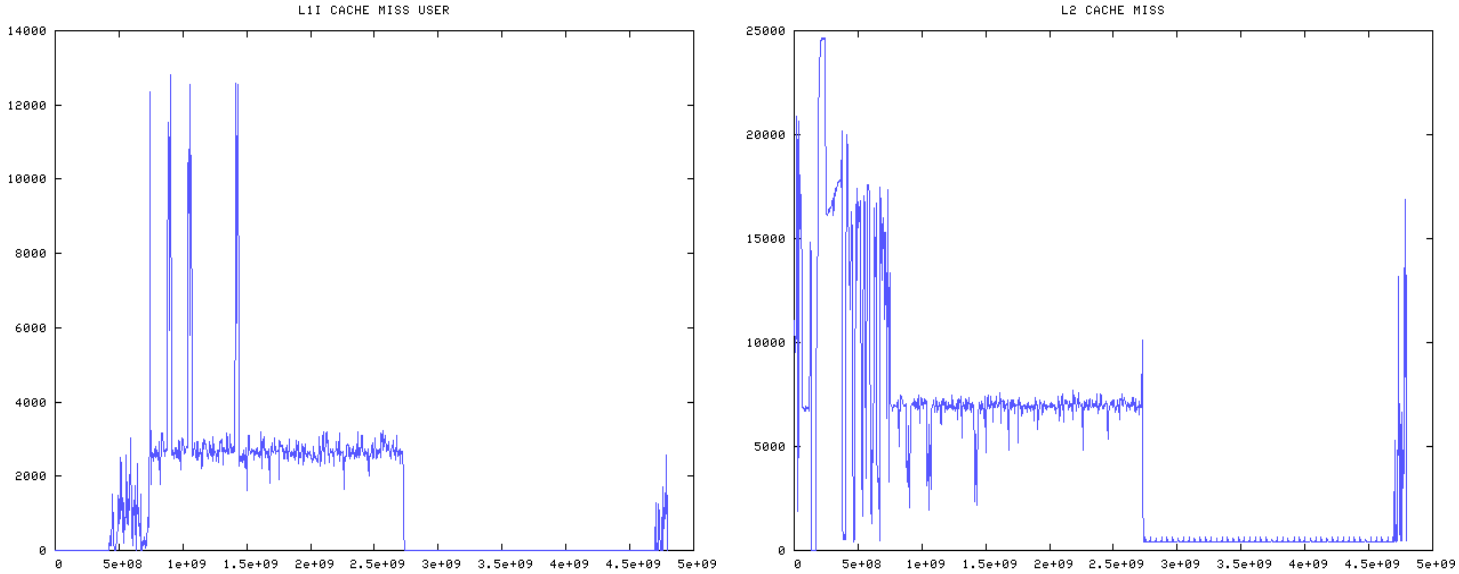

Figure 89: L1I Cache Miss User (left) and L2 Cache Miss (right) livegraphs

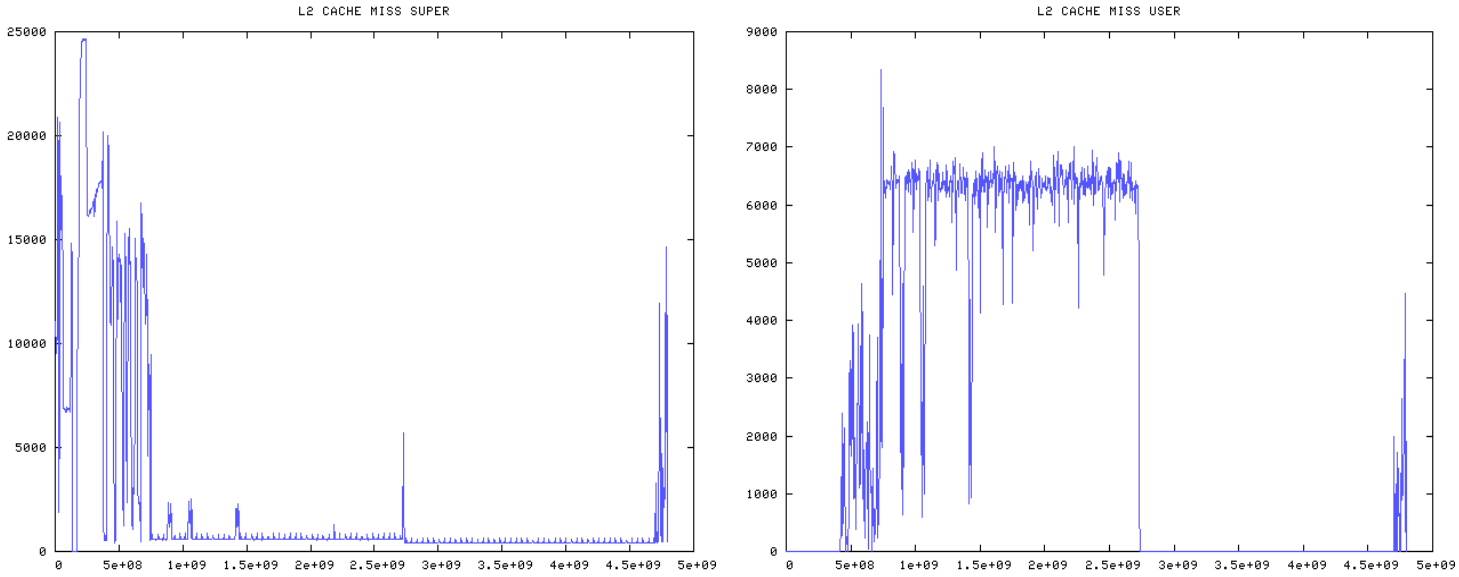

Figure 90: L2 Cache Miss Super (left) and L2 Cache Miss User (right) livegraphs

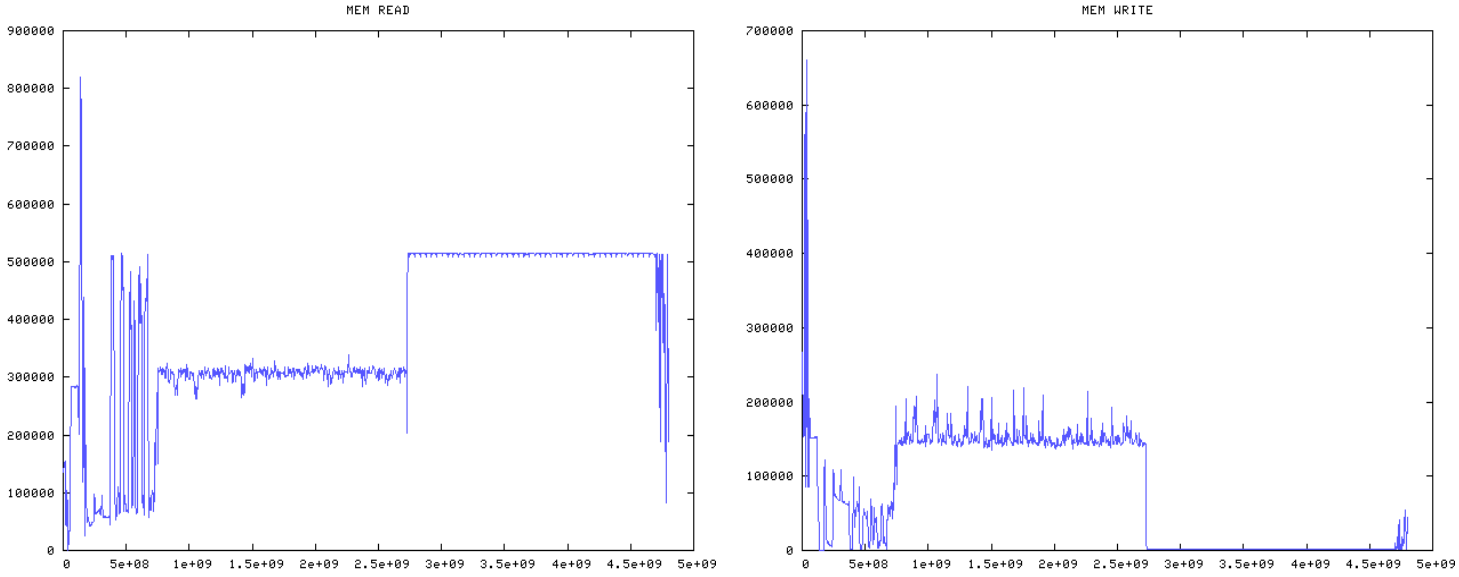

Figure 91: Memory-read (left) and Memory-Write (right) livegraphs

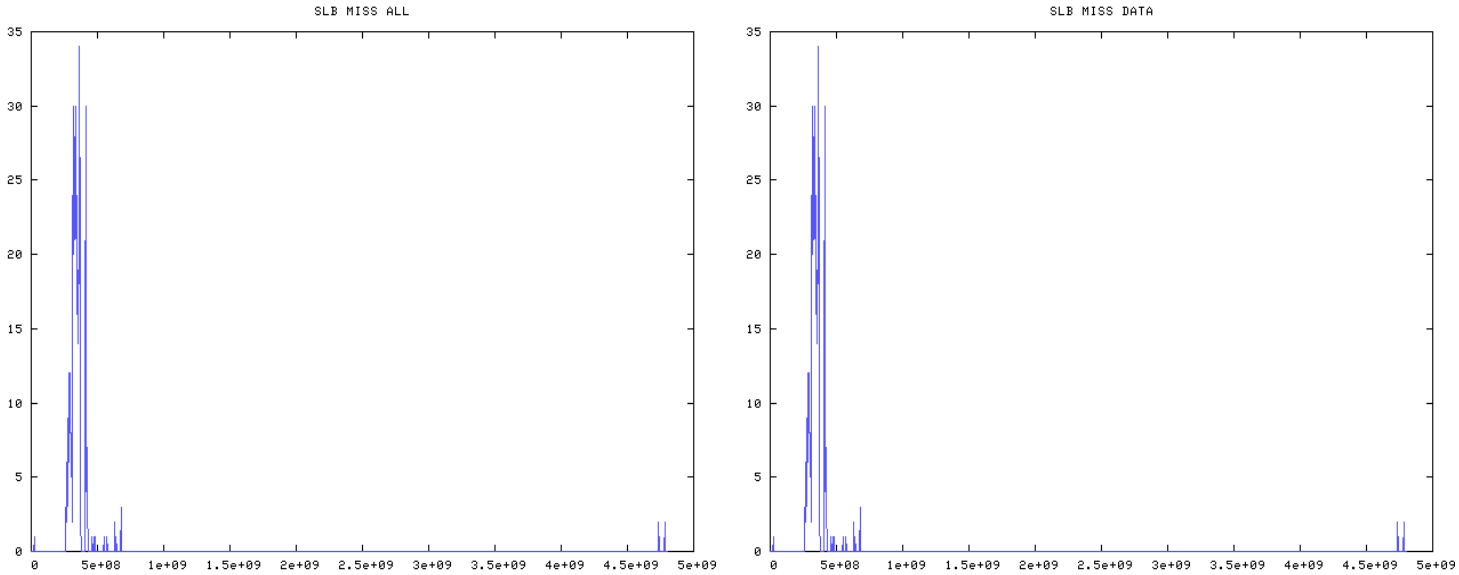

Figure 92: All SLB misses (left) and Data SLB Misses (right) livegraphs

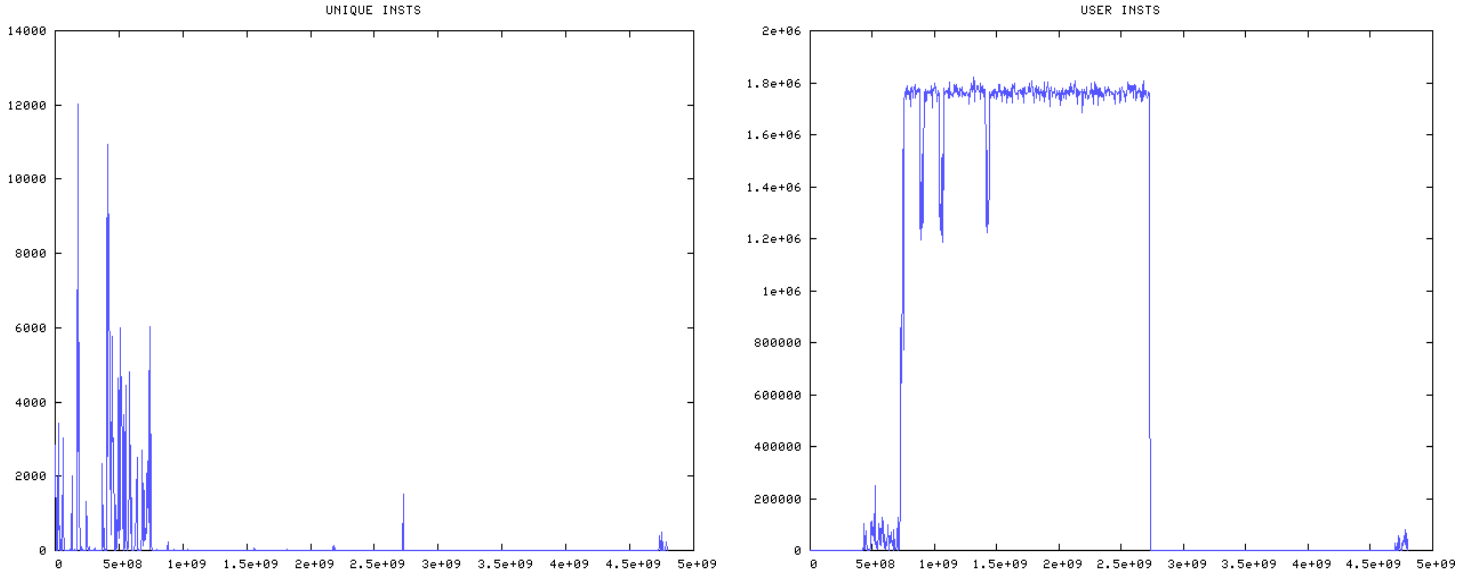

Figure 93: Unique Instructions (left) and User Instructions (right) livegraphs

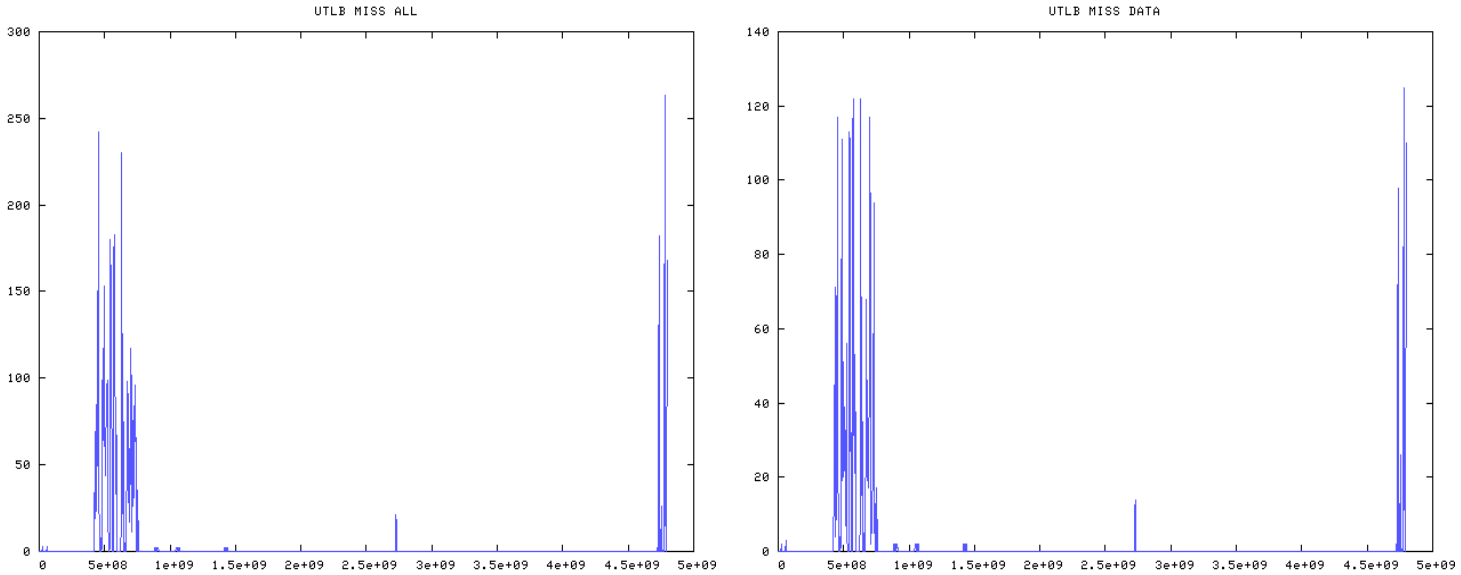

Figure 94: All User TLB Misses (left) and Data TLB Misses (right) livegraphs

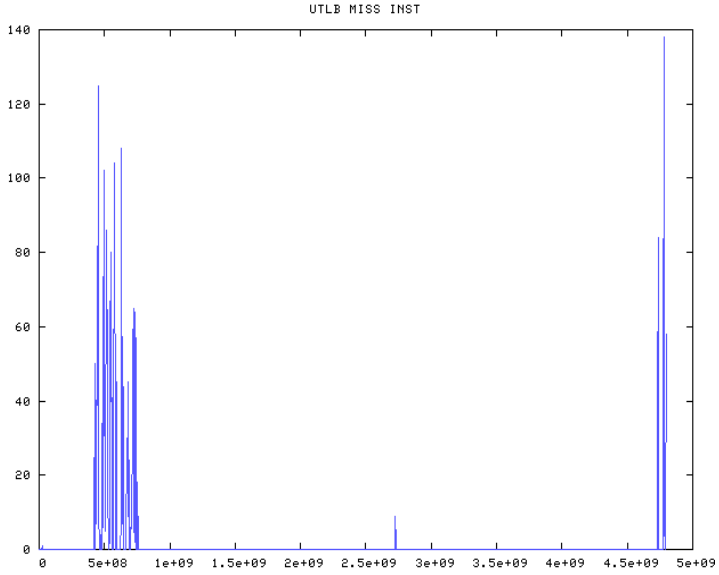

Figure 95: User TLB Instruction Miss livegraphs

B.6 HMMER

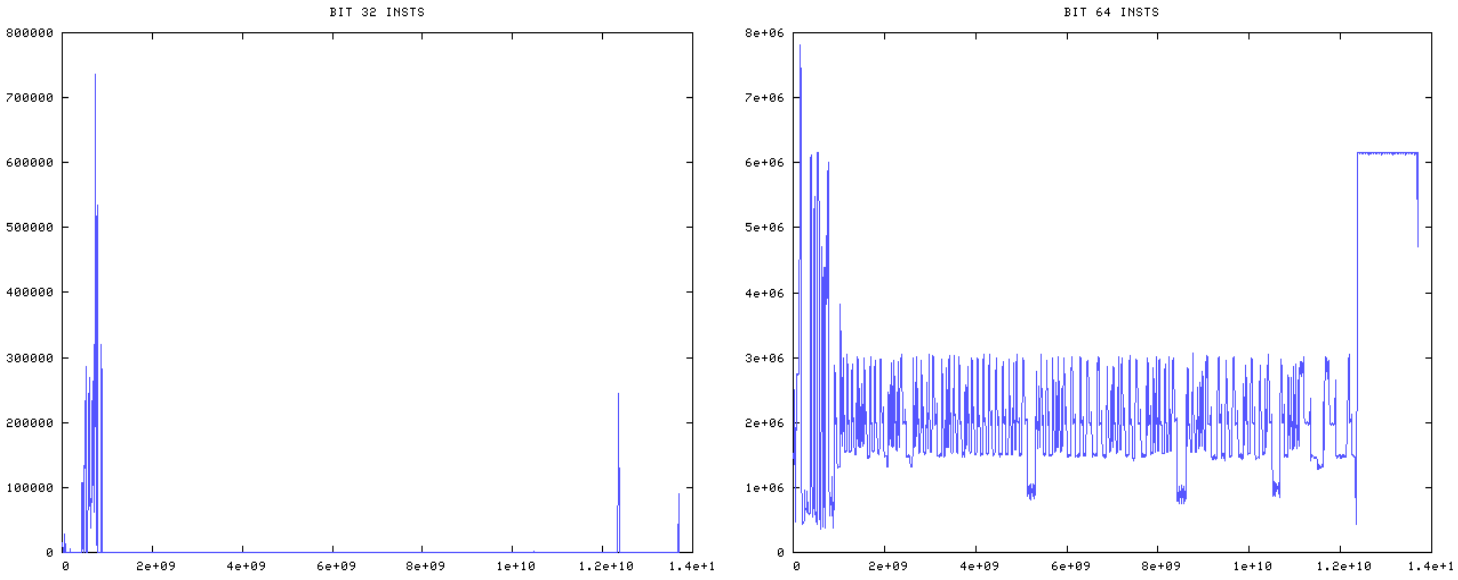

Figure 96: 32-bit (left) and 64-bit (right) instruction livegraphs

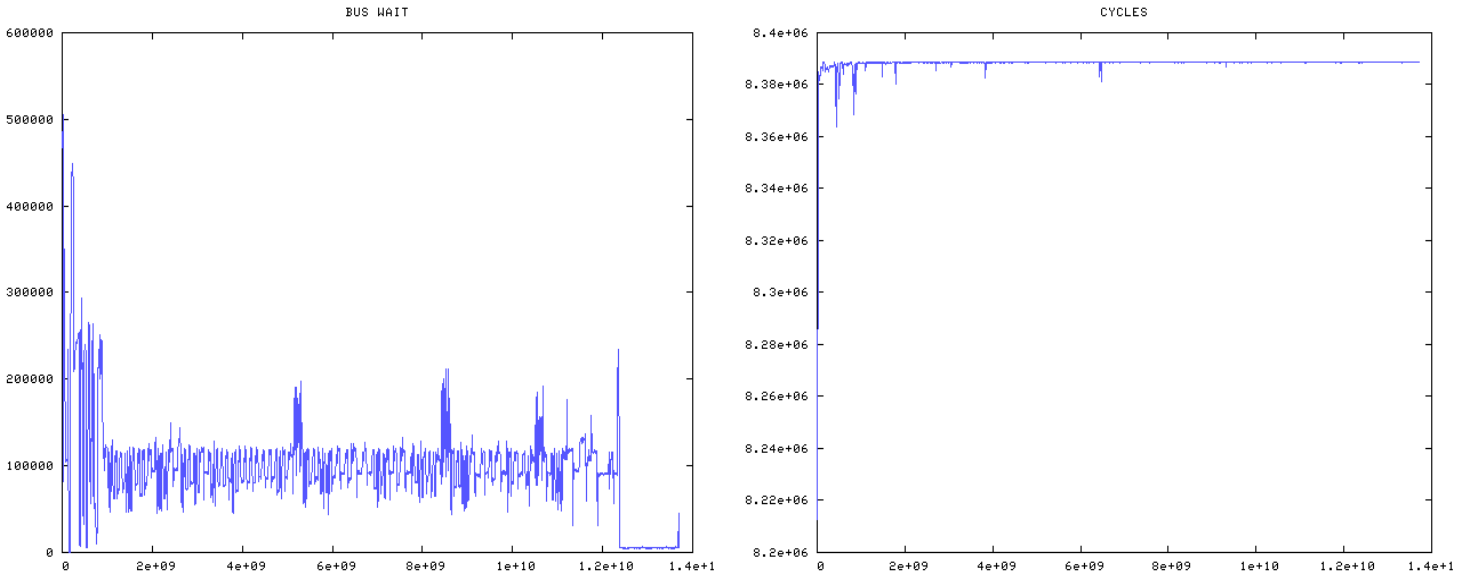

Figure 97: Bus-wait (left) and Cycles (right) livegraphs

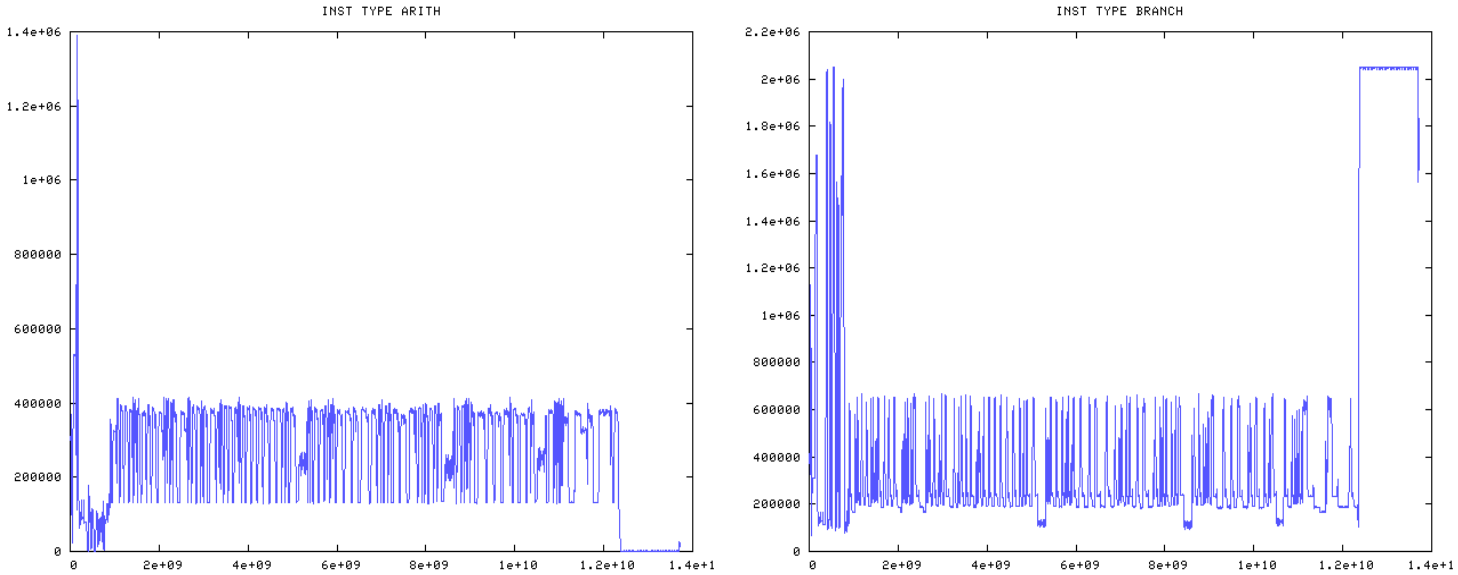

Figure 98: Arithmetic (left) and Branch (right) instructions livegraphs

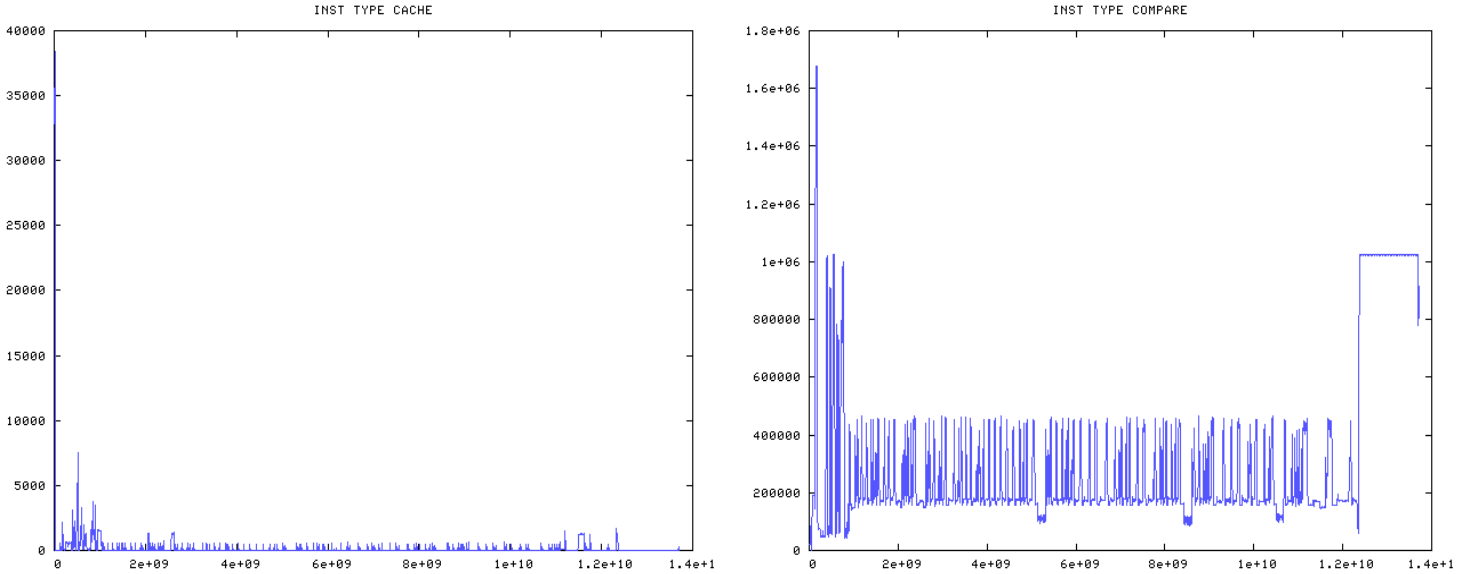

Figure 99: Cache (left) and Compare (right) instructions livegraphs

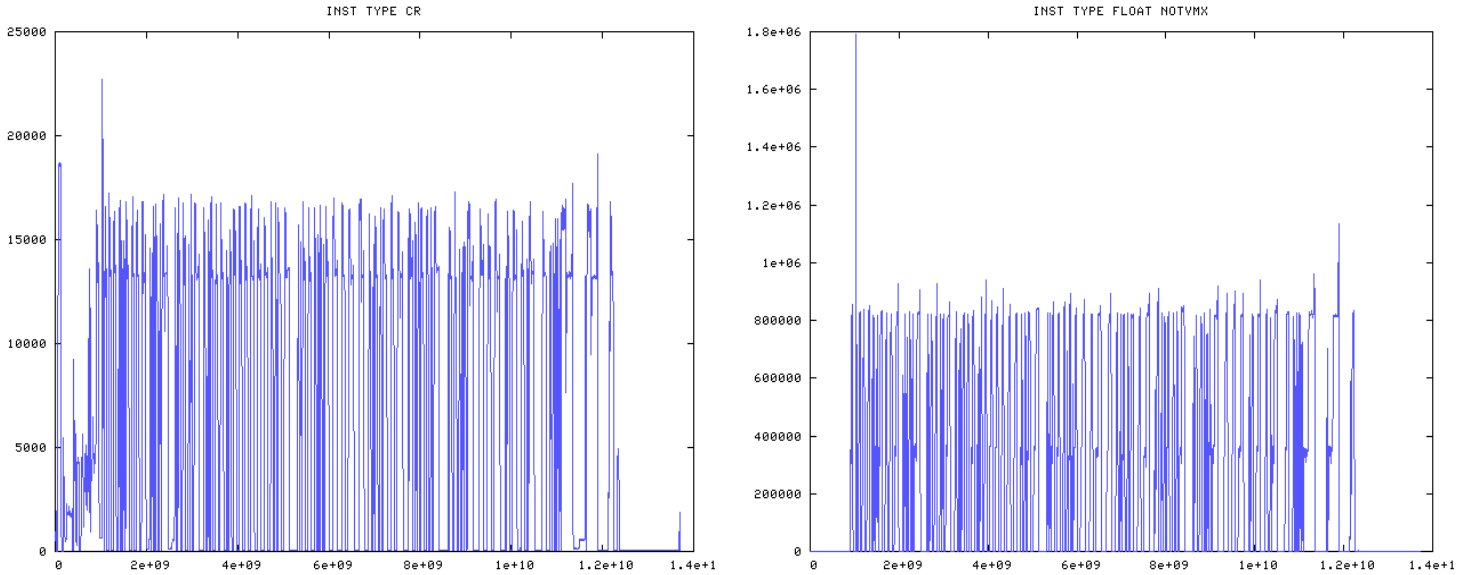

Figure 100: CR (left) and Not VMX-Float (right) instructions livegraphs

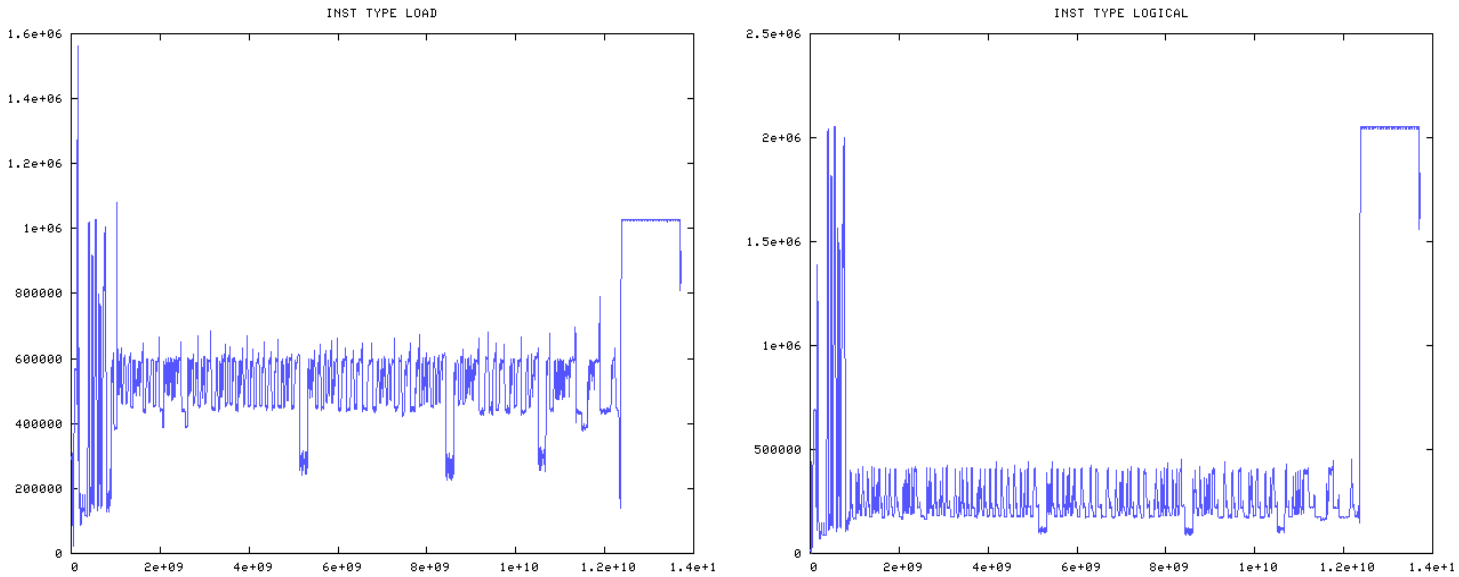

Figure 101: Load (left) and Logical (right) instructions livegraphs

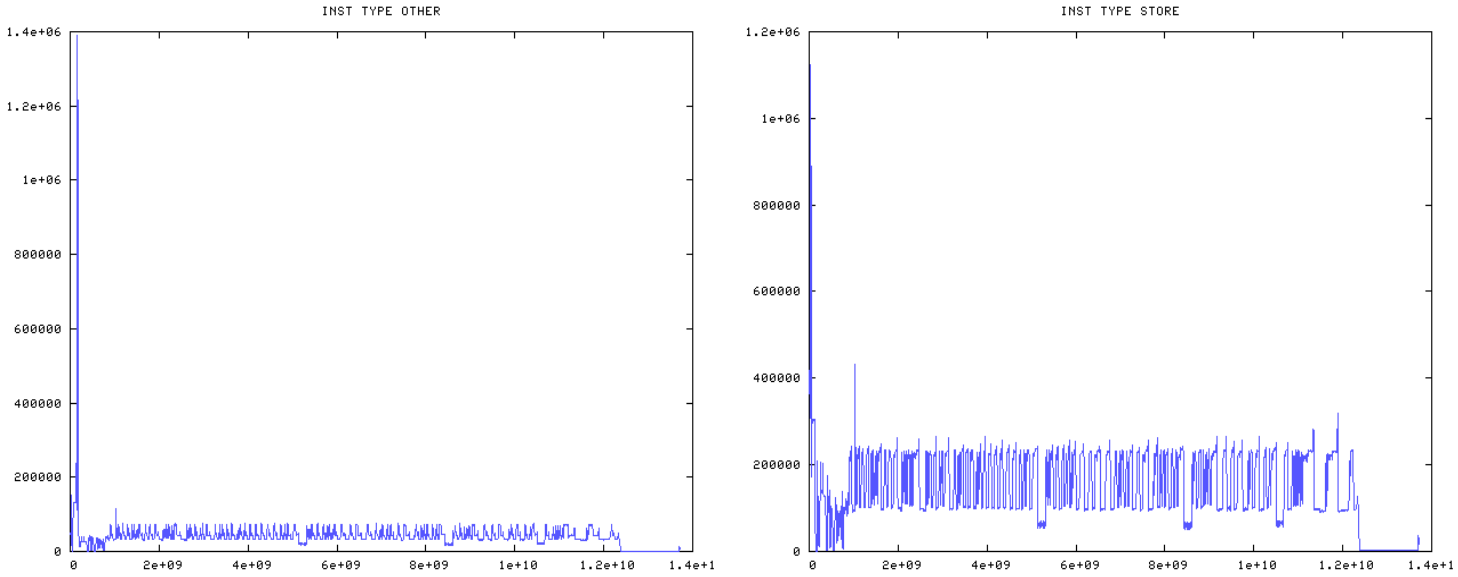

Figure 102: Other (left) and Store (right) instructions livegraphs

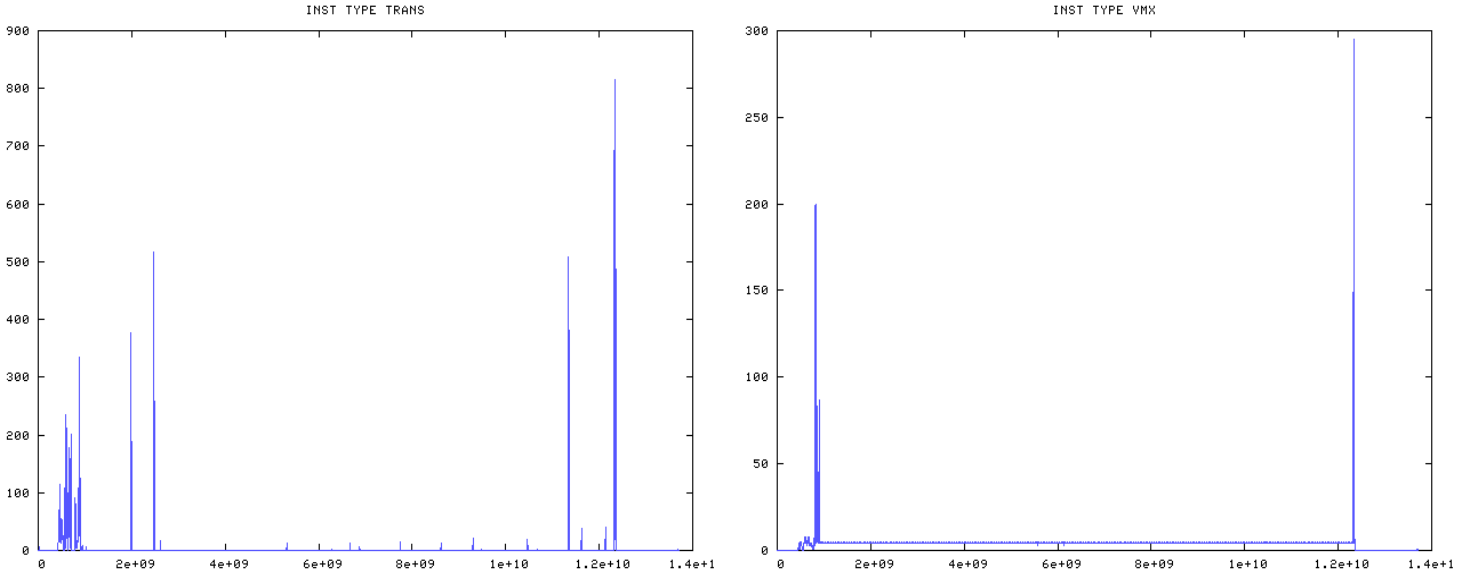

Figure 103: Trans (left) and VMX (right) instructions livegraphs

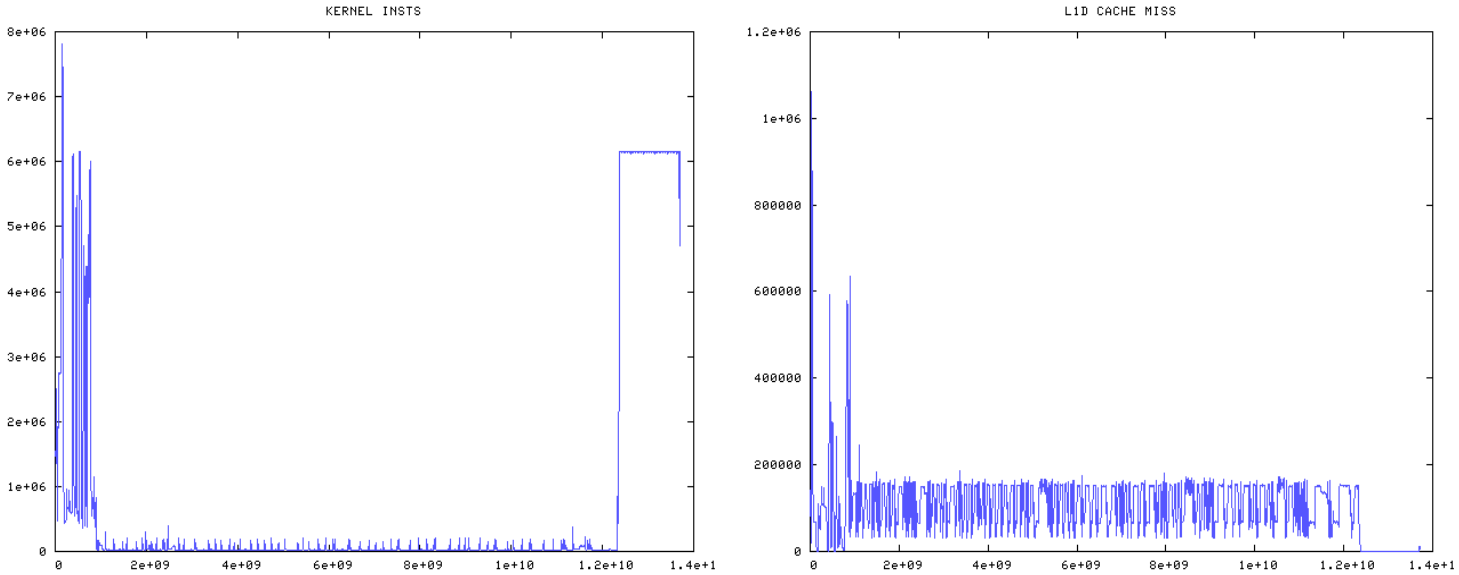

Figure 104: Kernel instructions (left) and L1D Cache Miss (right) livegraphs

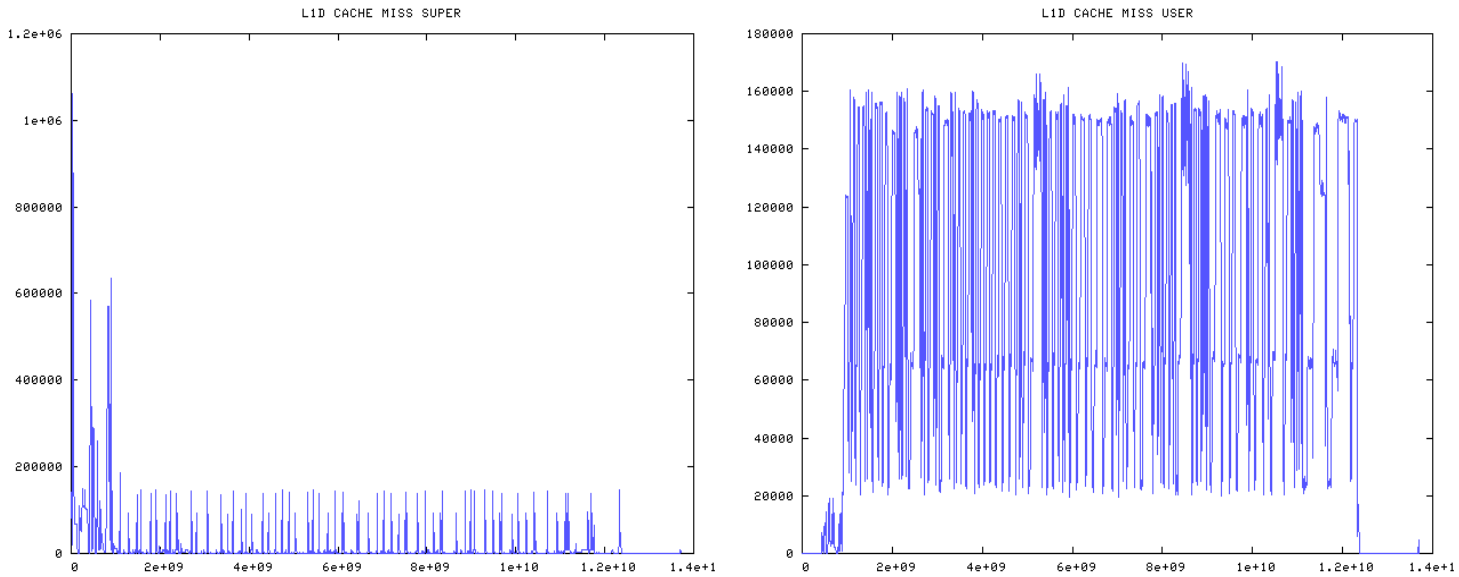

Figure 105: L1d Cache Miss Super (left) and L1d Cache Miss User (right) livegraphs

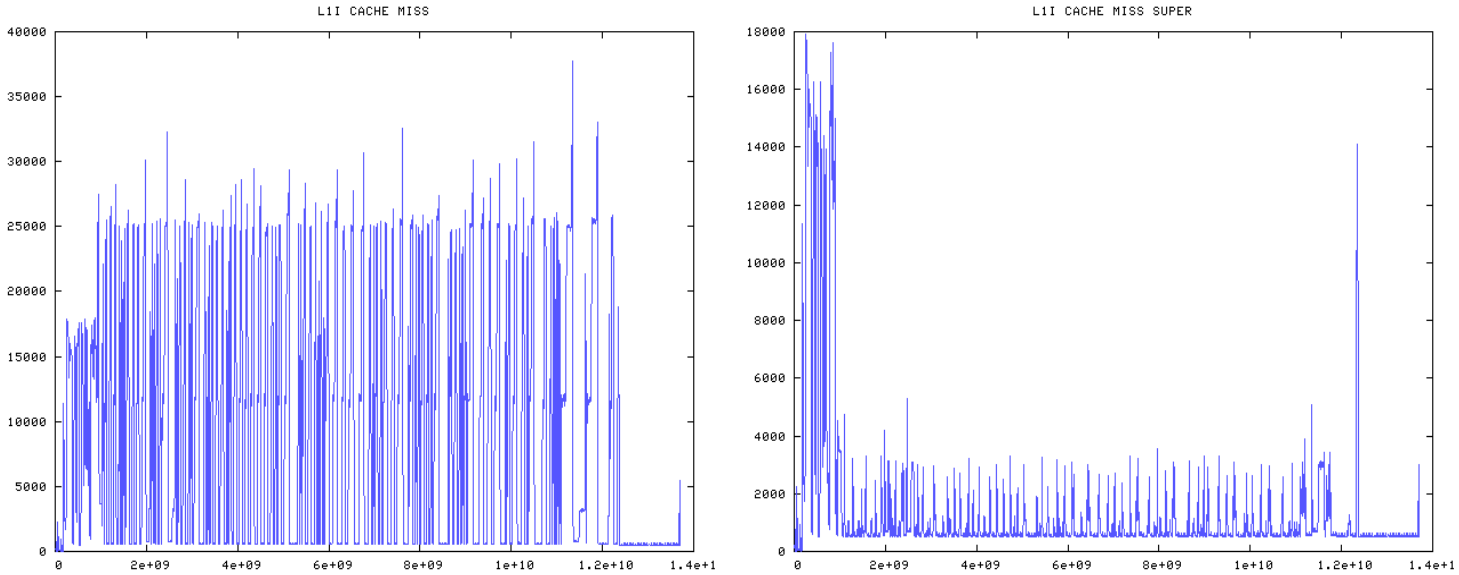

Figure 106: L1I Cache Miss (left) and L1I Cache Miss Super (right) livegraphs

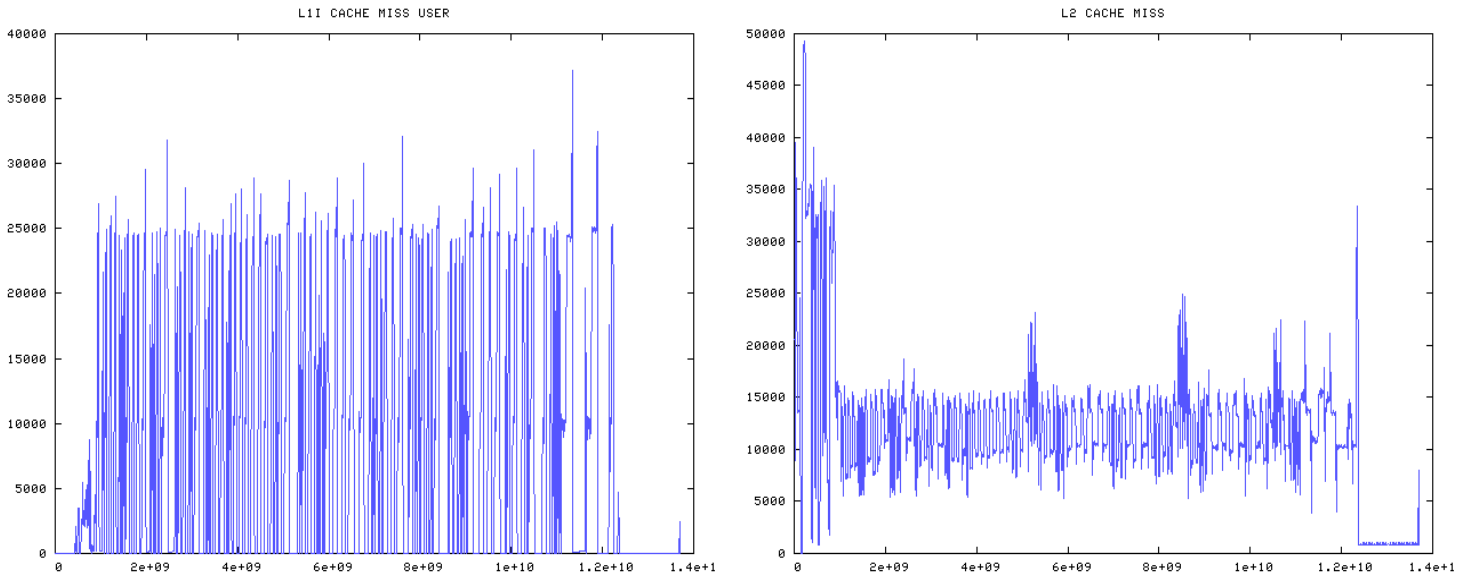

Figure 107: L1I Cache Miss User (left) and L2 Cache Miss (right) livegraphs

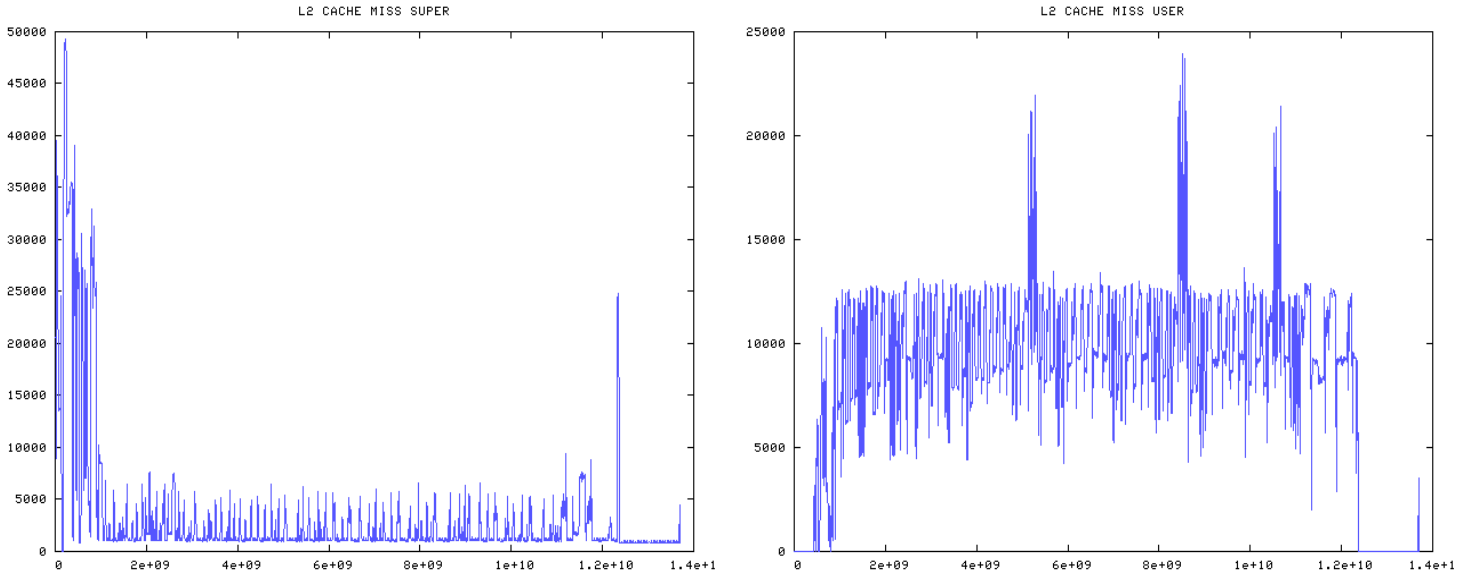

Figure 108: L2 Cache Miss Super (left) and L2 Cache Miss User (right) livegraphs

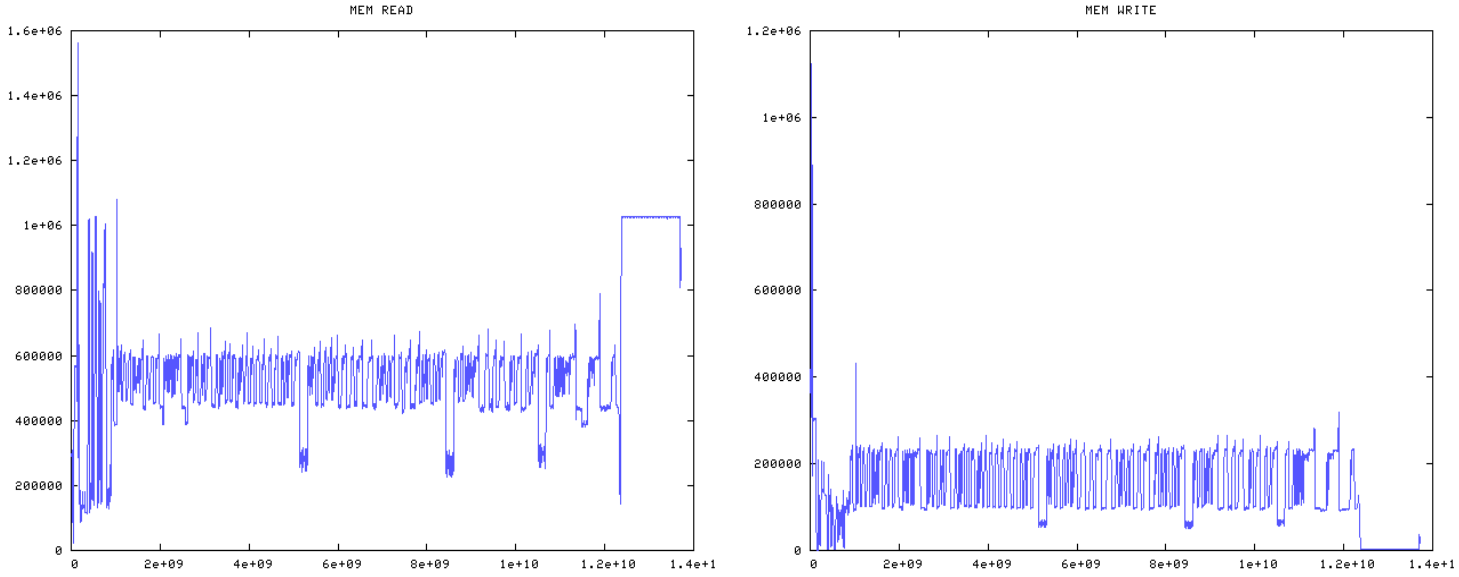

Figure 109: Memory-read (left) and Memory-Write (right) livegraphs

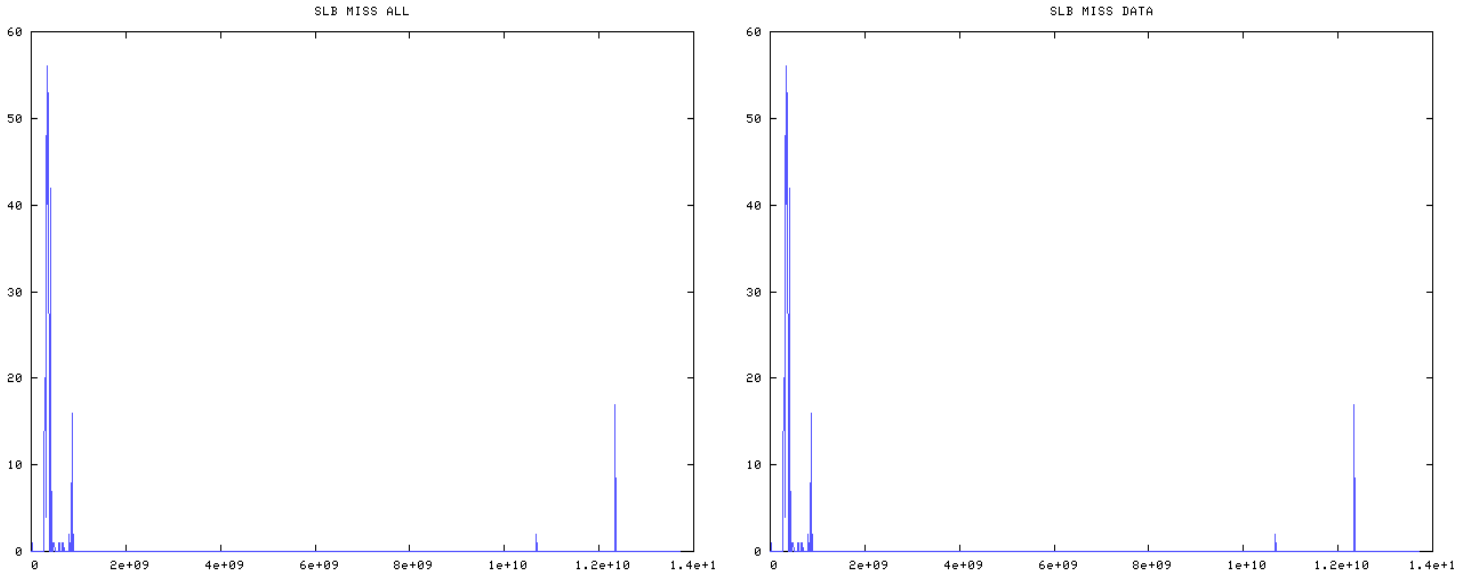

Figure 110: All SLB misses (left) and Data SLB Misses (right) livegraphs

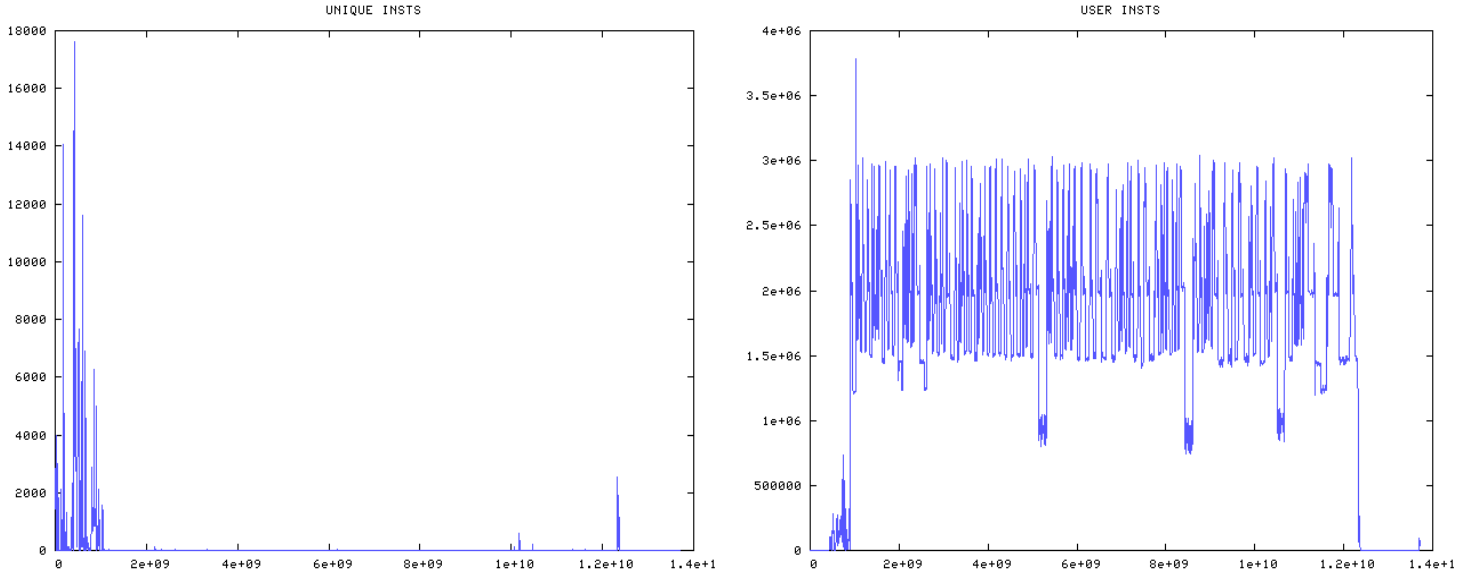

Figure 111: Unique Instructions (left) and User Instructions (right) livegraphs

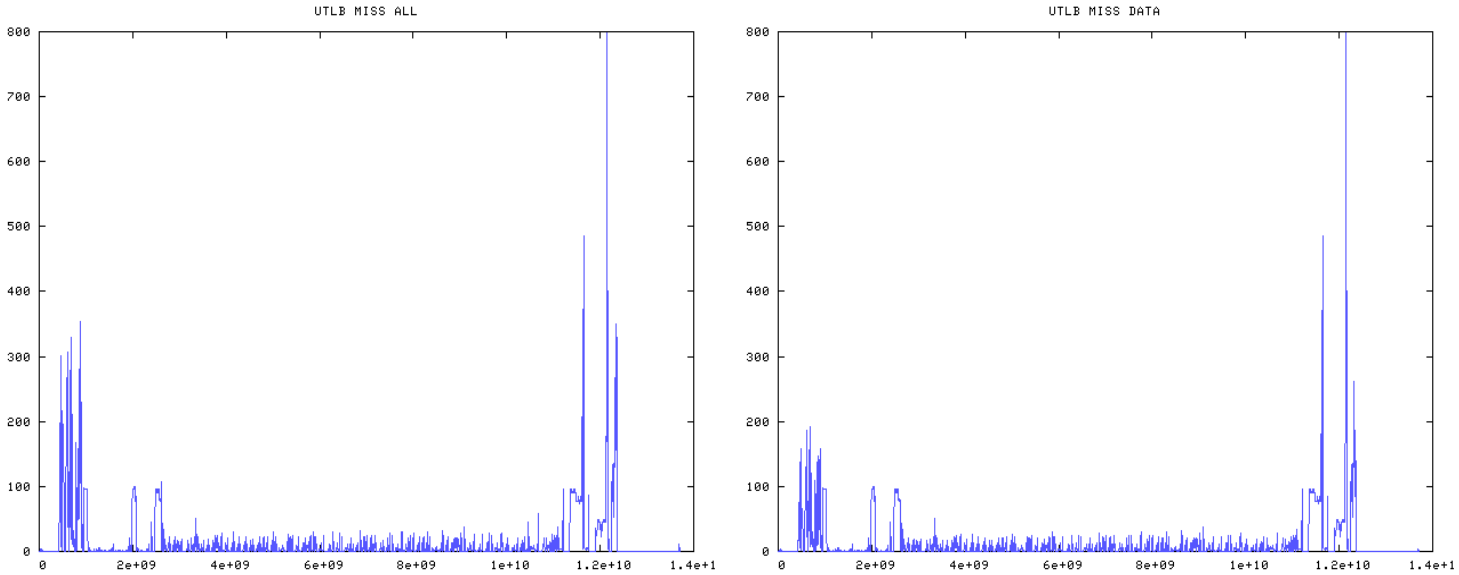

Figure 112: All User TLB Misses (left) and Data TLB Misses (right) livegraphs

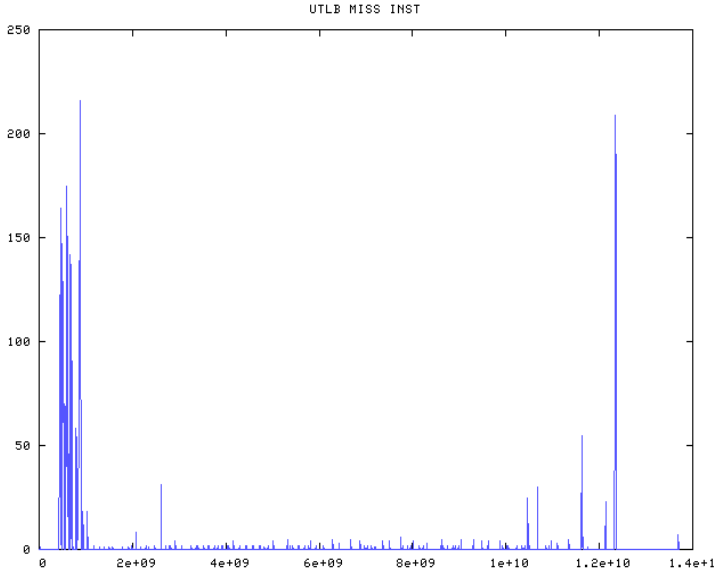

Figure 113: User TLB Instruction Miss livegraphs

B.7 PHYLIP

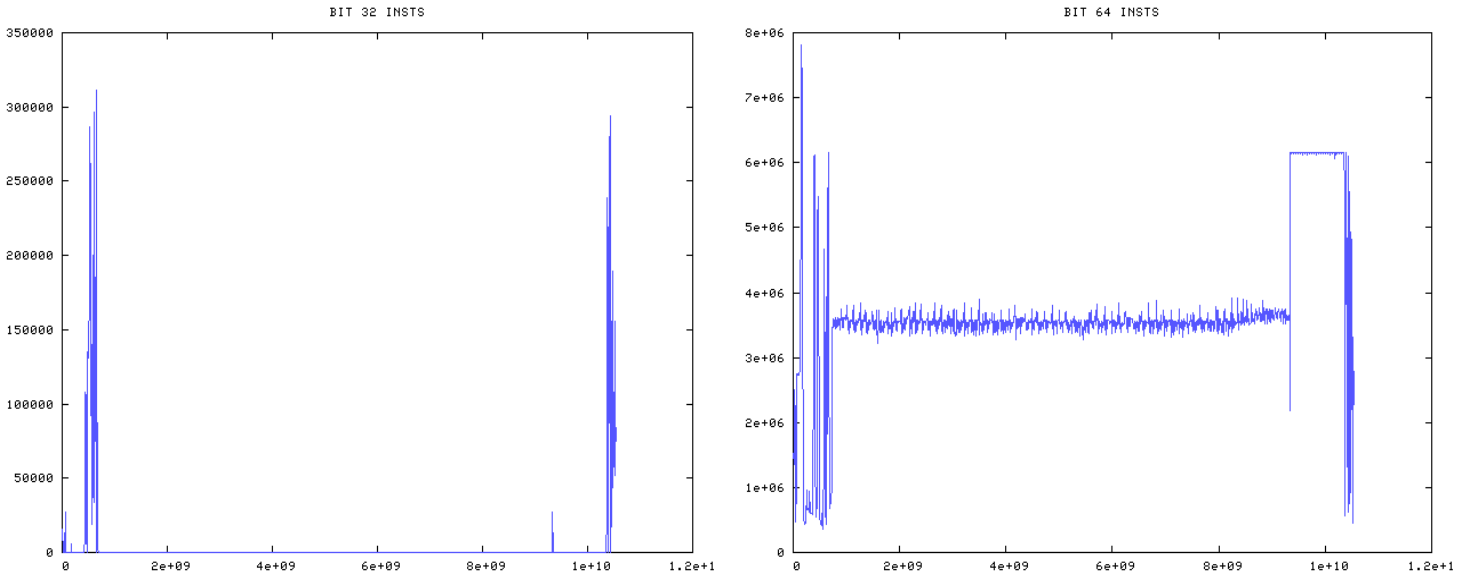

Figure 114: 32-bit (left) and 64-bit (right) instruction livegraphs

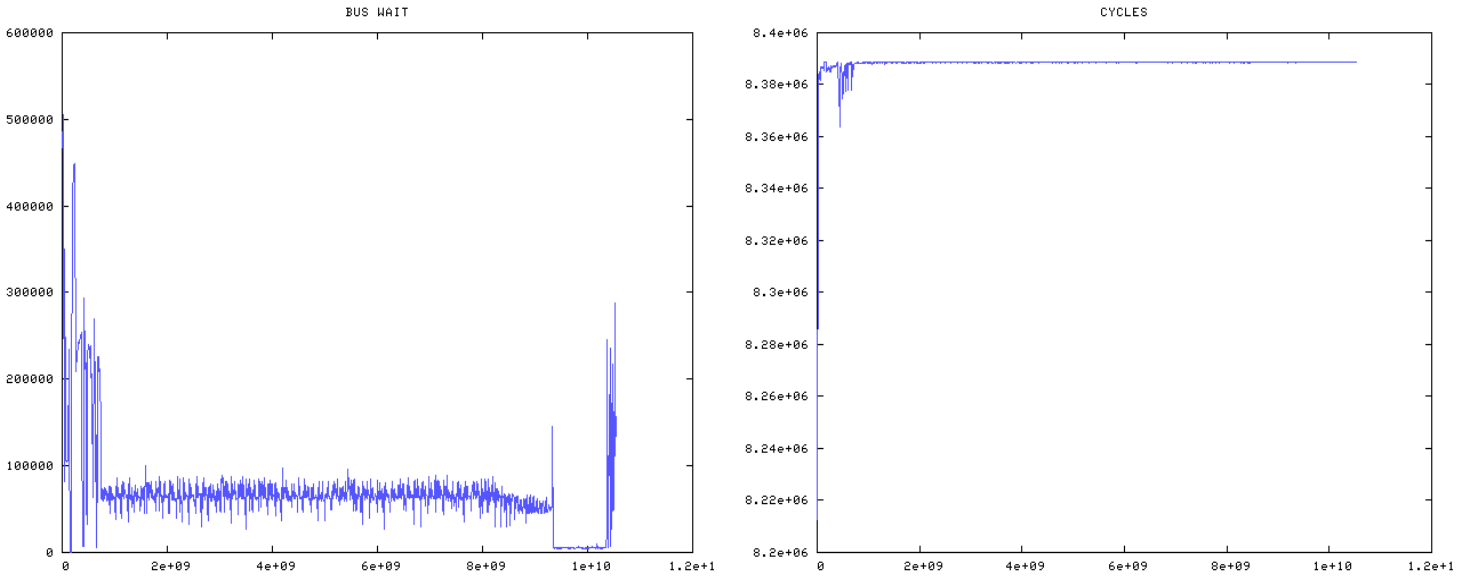

Figure 115: Bus-wait (left) and Cycles (right) livegraphs

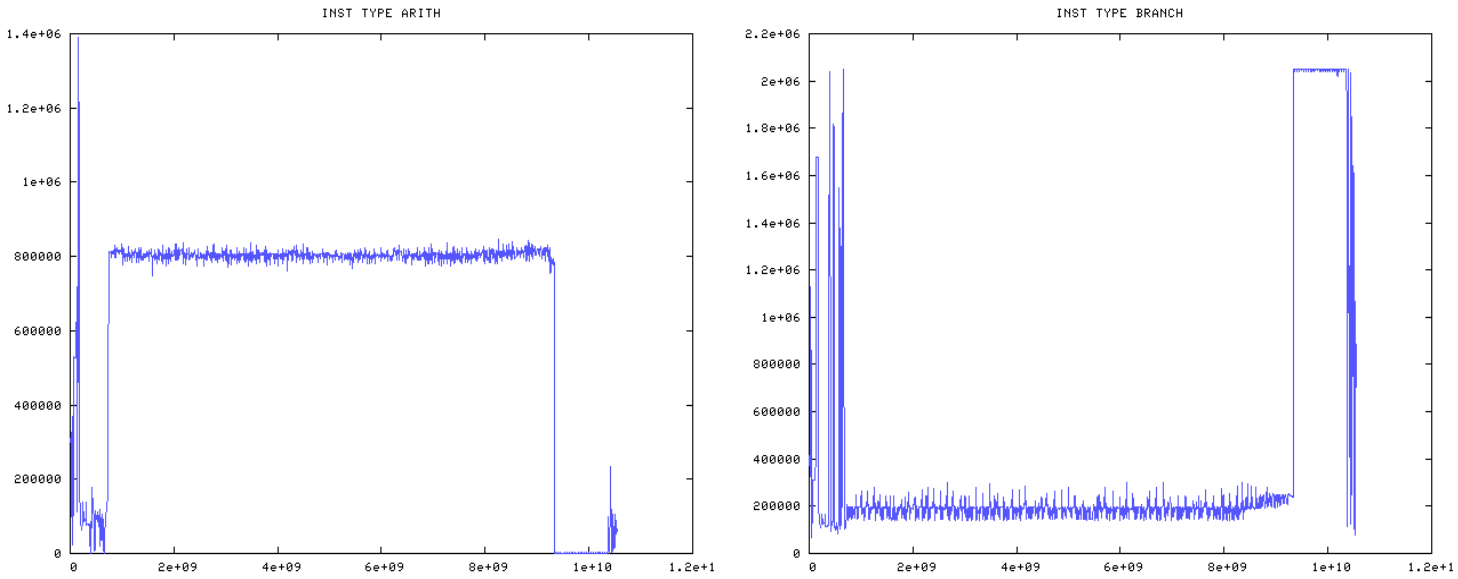

Figure 116: Arithmetic (left) and Branch (right) instructions livegraphs

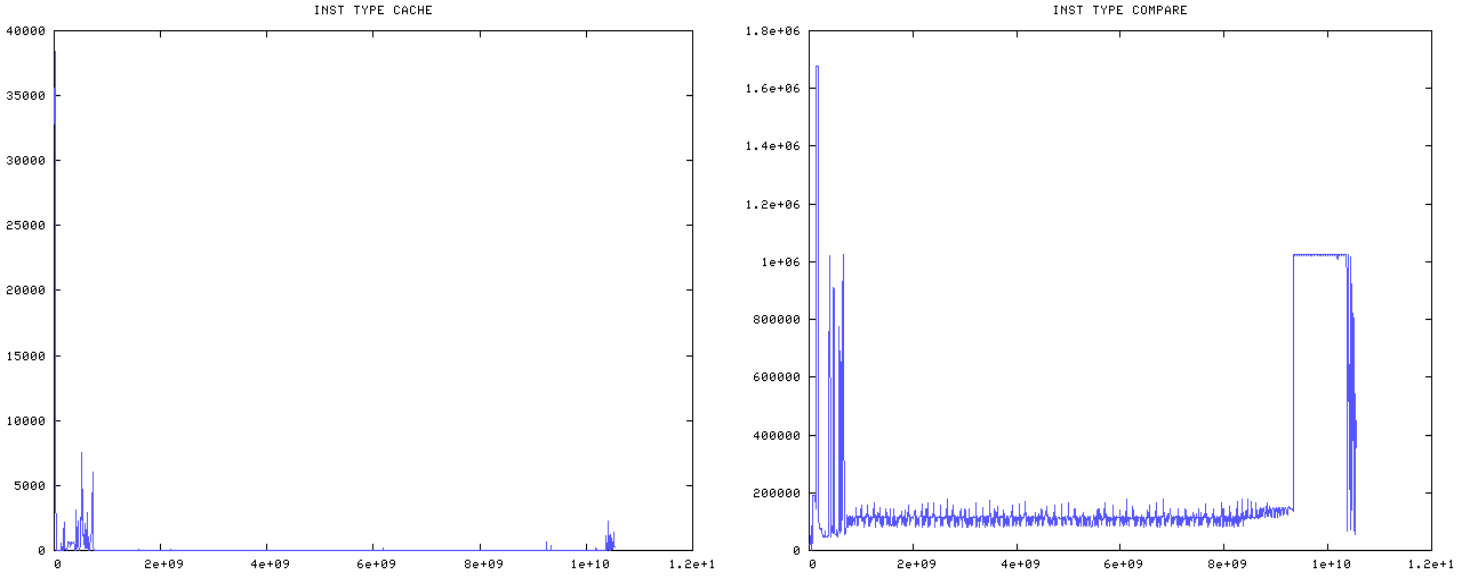

Figure 117: Cache (left) and Compare (right) instructions livegraphs

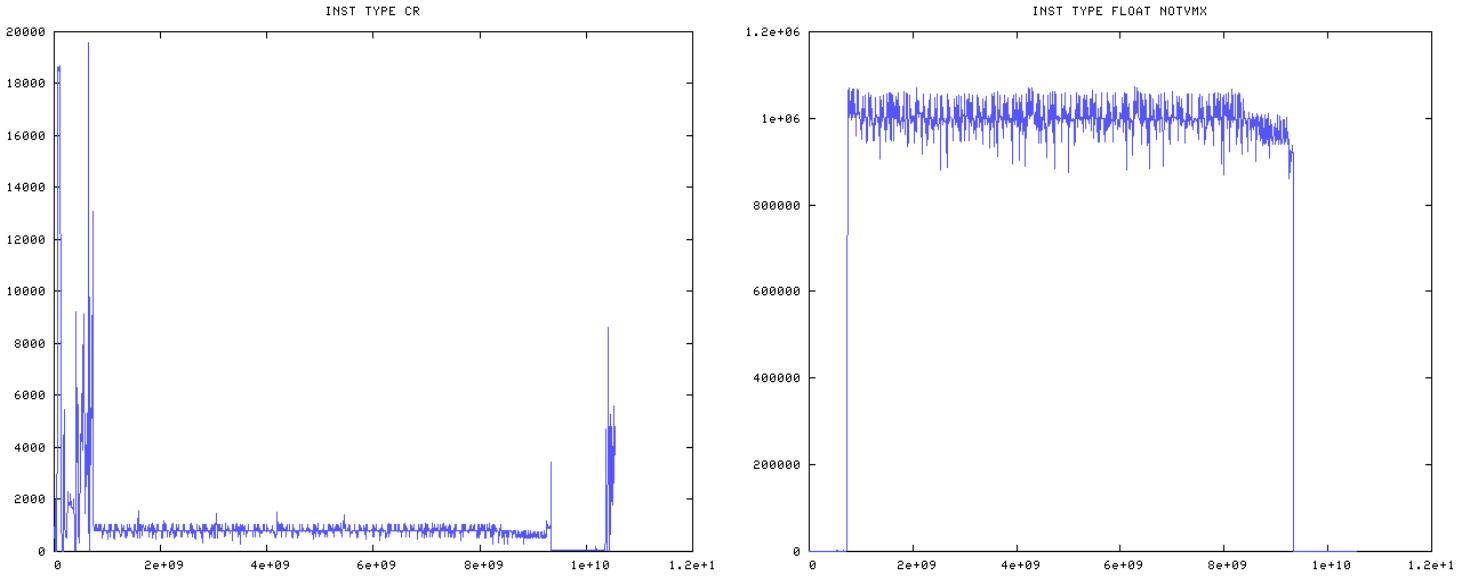

Figure 118: CR (left) and Not VMX-Float (right) instructions livegraphs

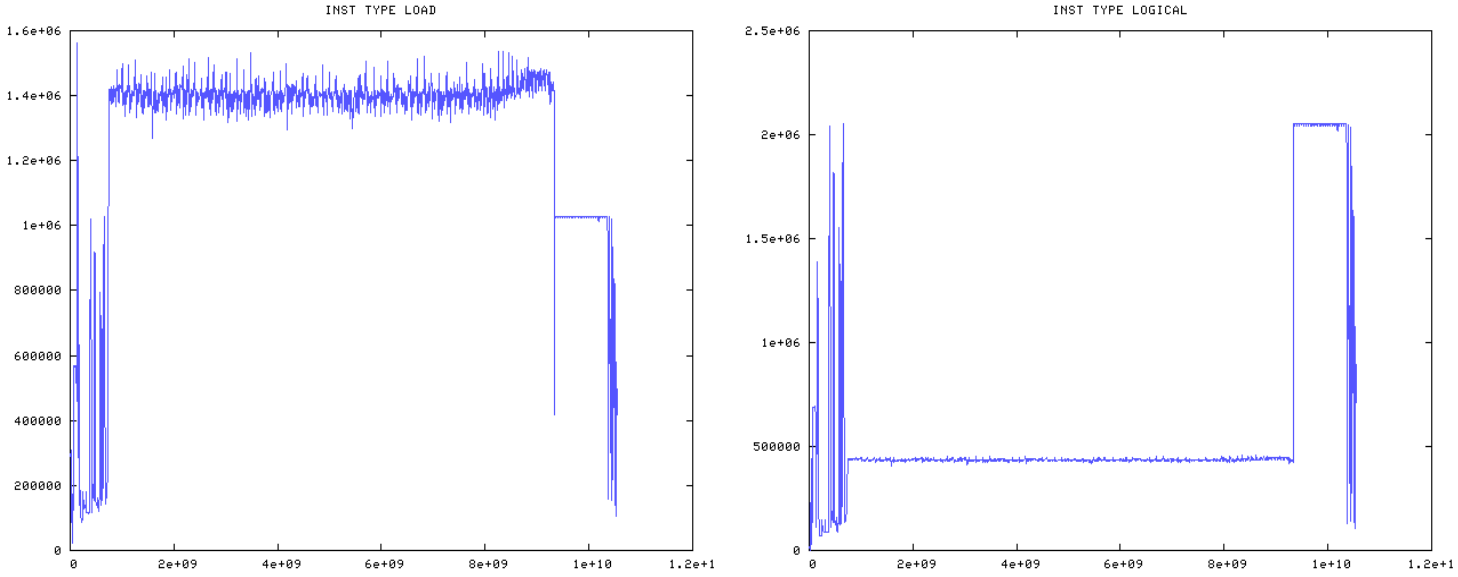

Figure 119: Load (left) and Logical (right) instructions livegraphs

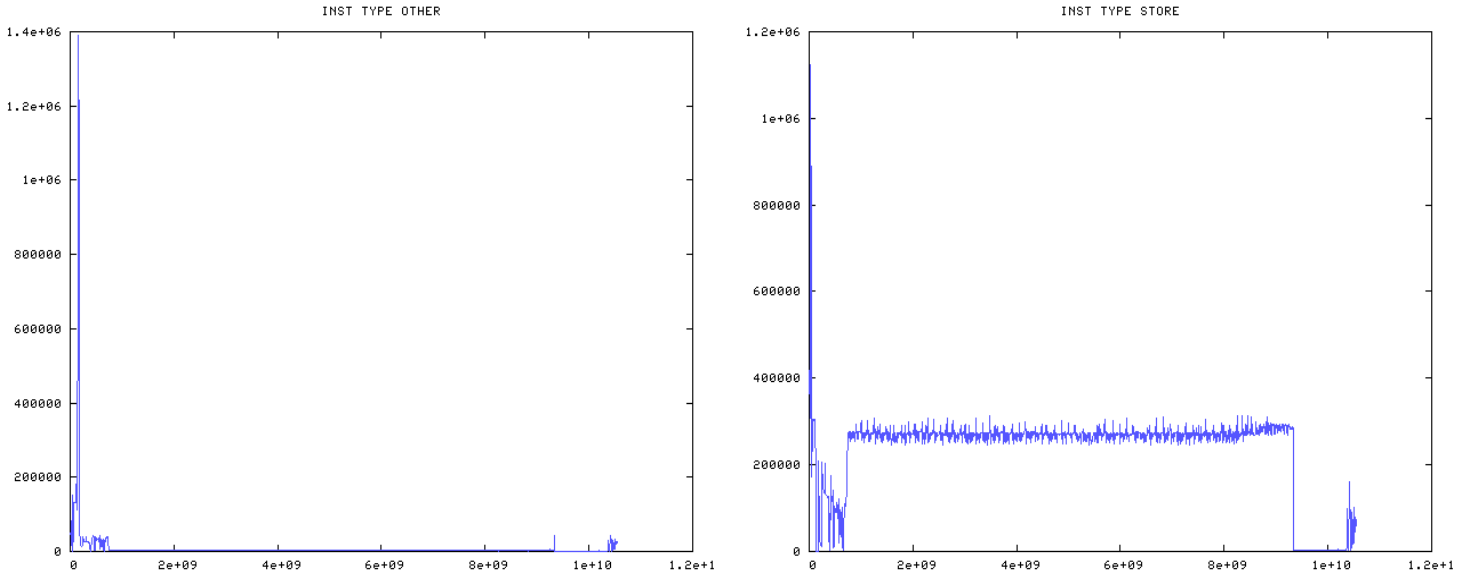

Figure 120: Other (left) and Store (right) instructions livegraphs

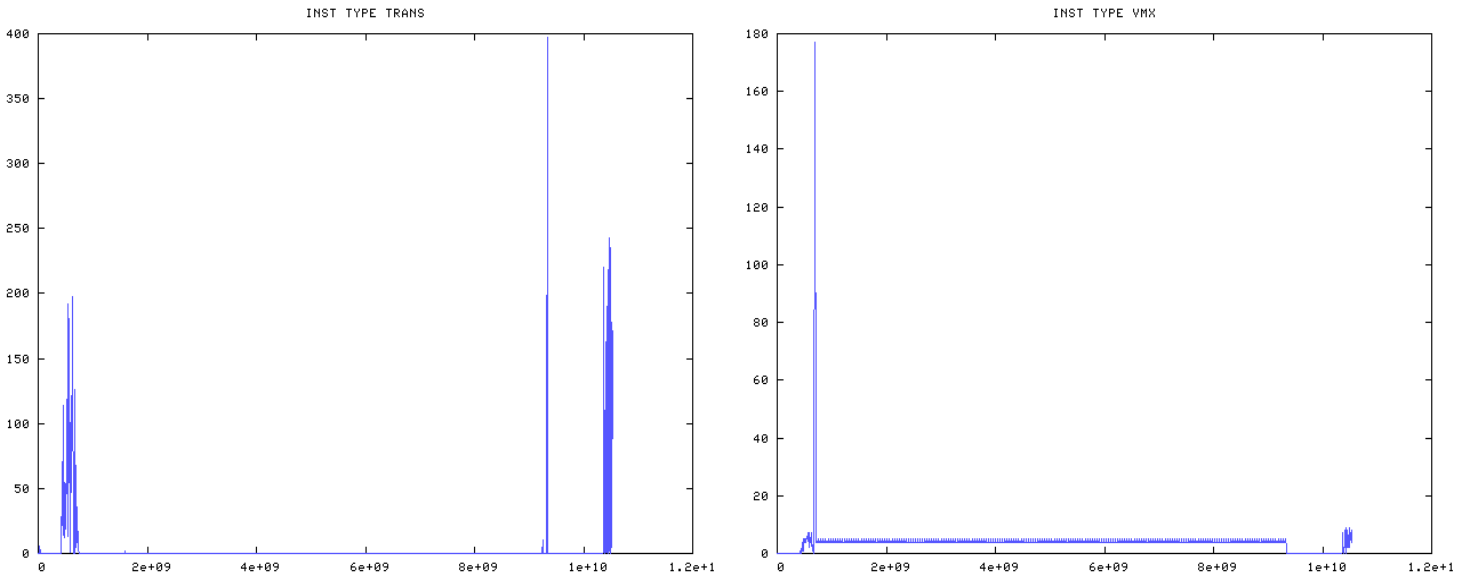

Figure 121: Trans (left) and VMX (right) instructions livegraphs

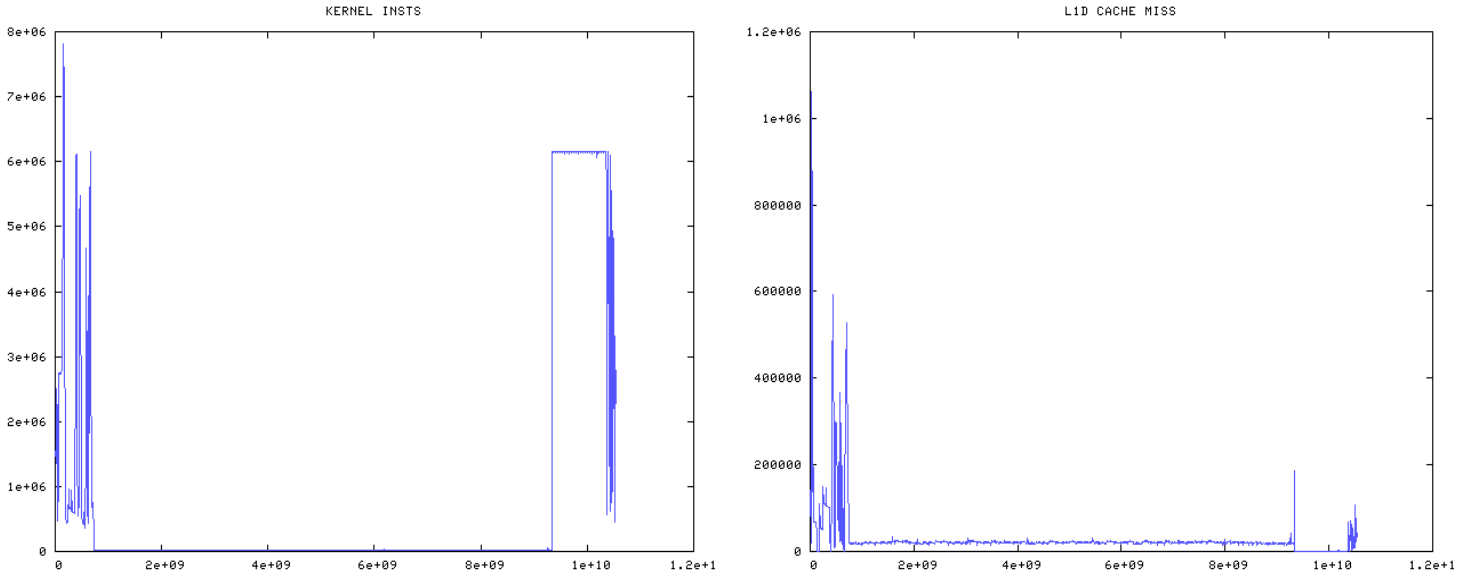

Figure 122: Kernel instructions (left) and L1D Cache Miss (right) livegraphs

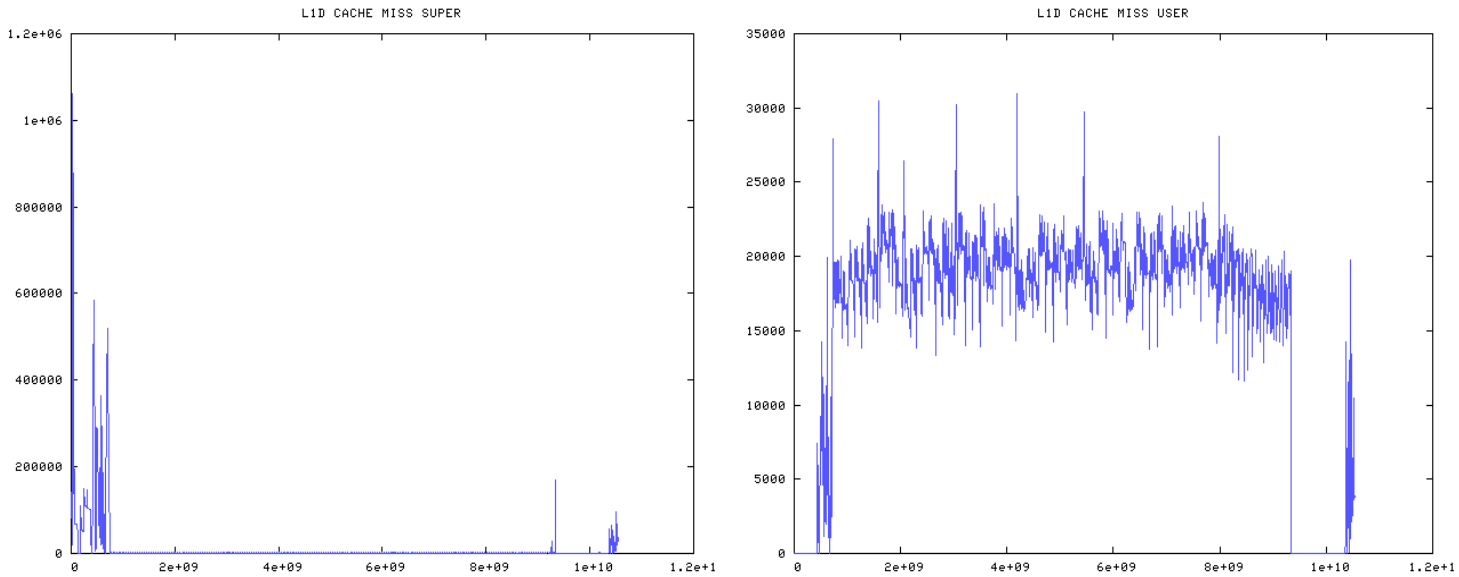

Figure 123: L1d Cache Miss Super (left) and L1d Cache Miss User (right) livegraphs

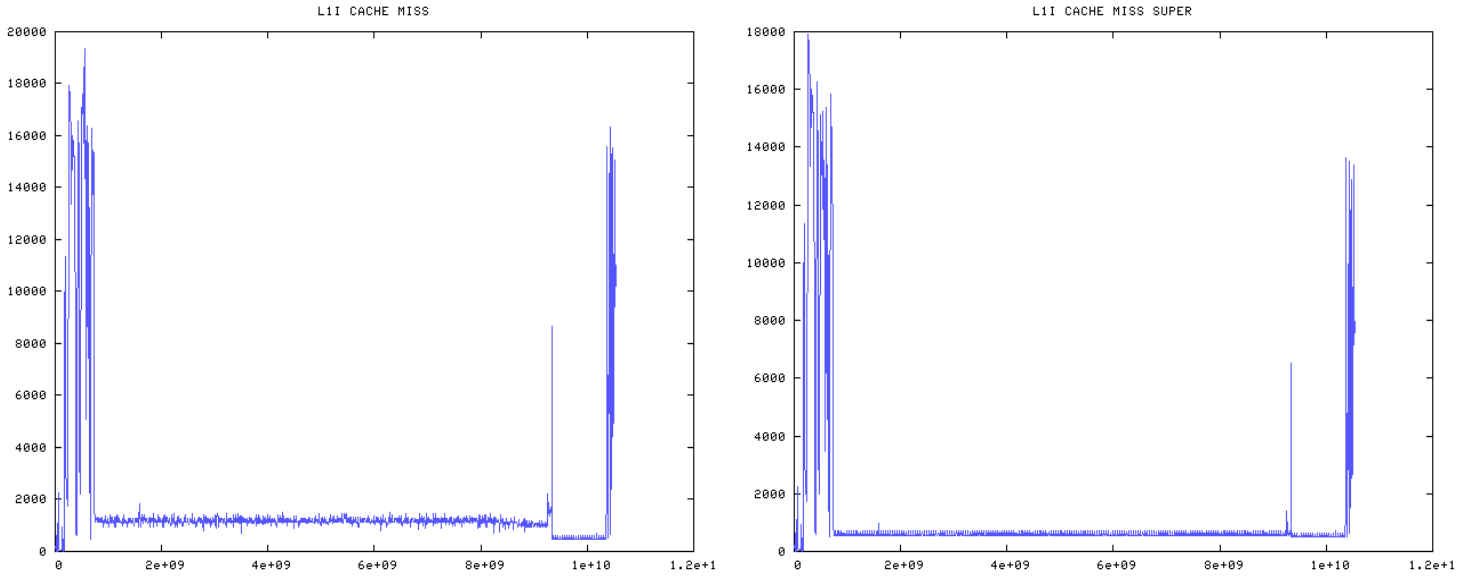

Figure 124: L1I Cache Miss (left) and L1I Cache Miss Super (right) livegraphs

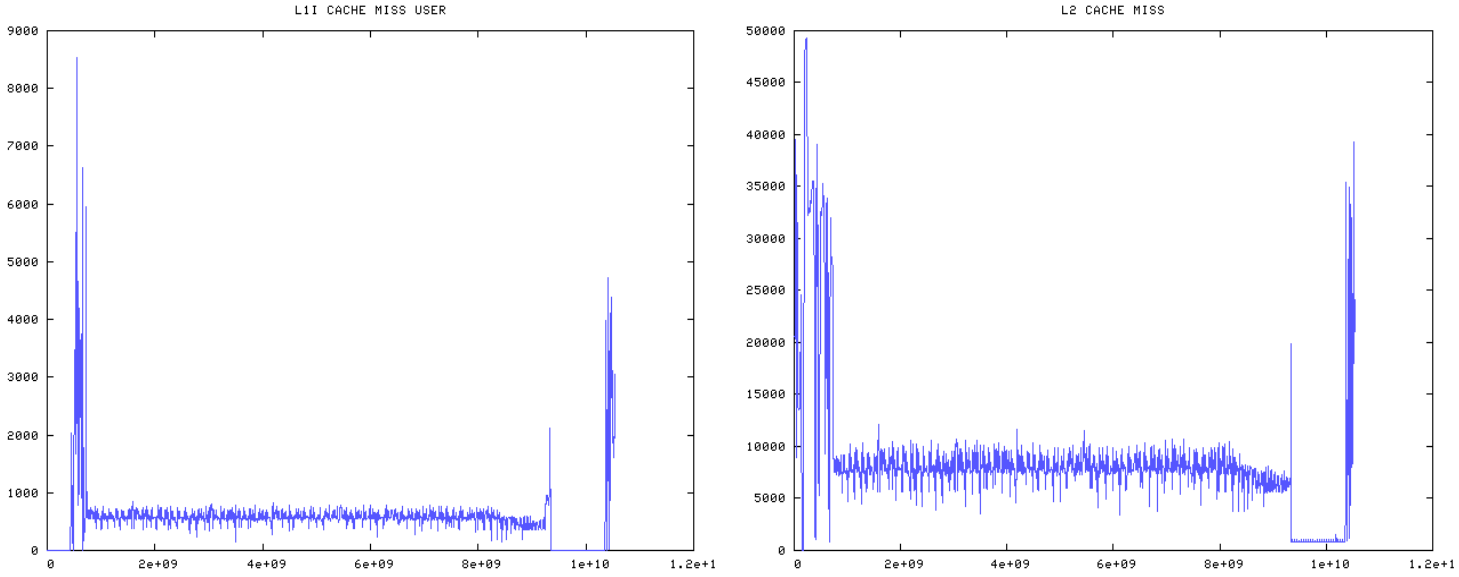

Figure 125: L1I Cache Miss User (left) and L2 Cache Miss (right) livegraphs

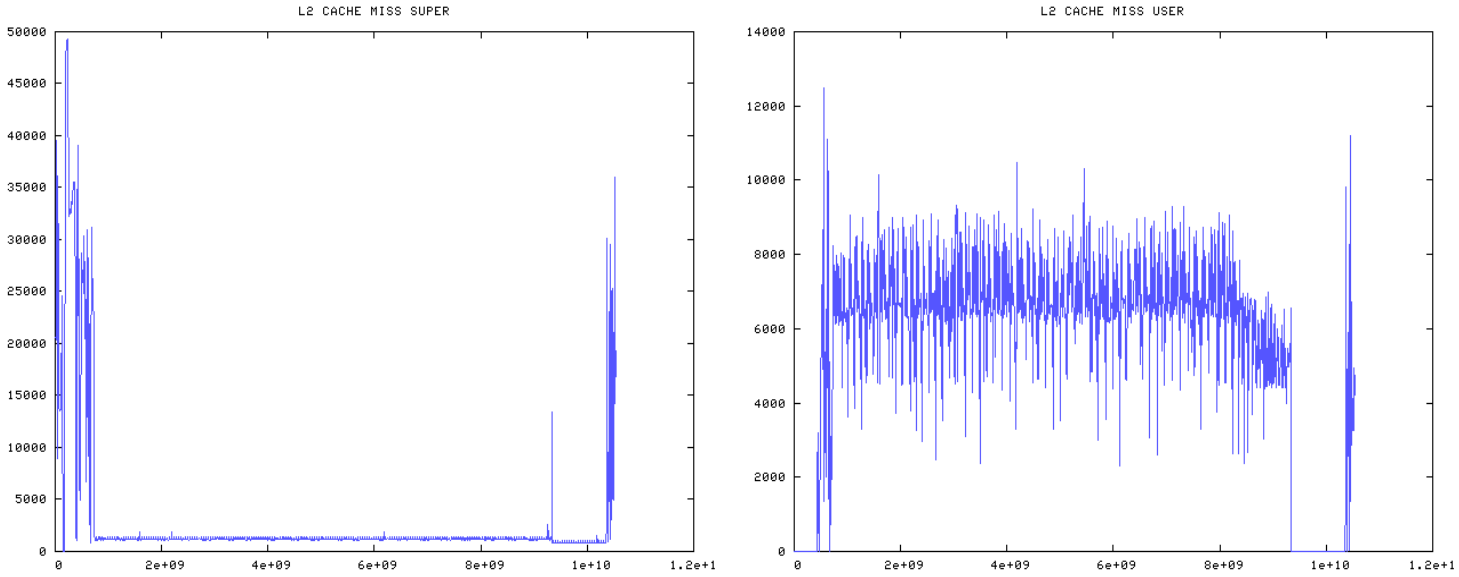

Figure 126: L2 Cache Miss Super (left) and L2 Cache Miss User (right) livegraphs

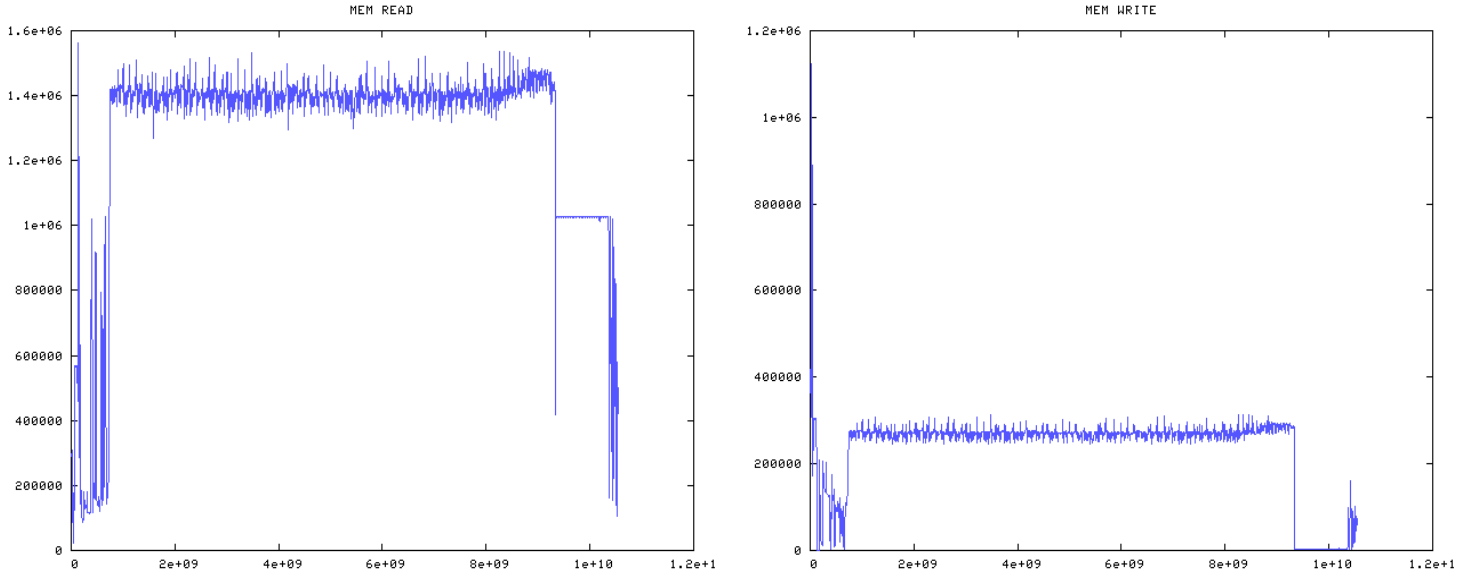

Figure 127: Memory-read (left) and Memory-Write (right) livegraphs

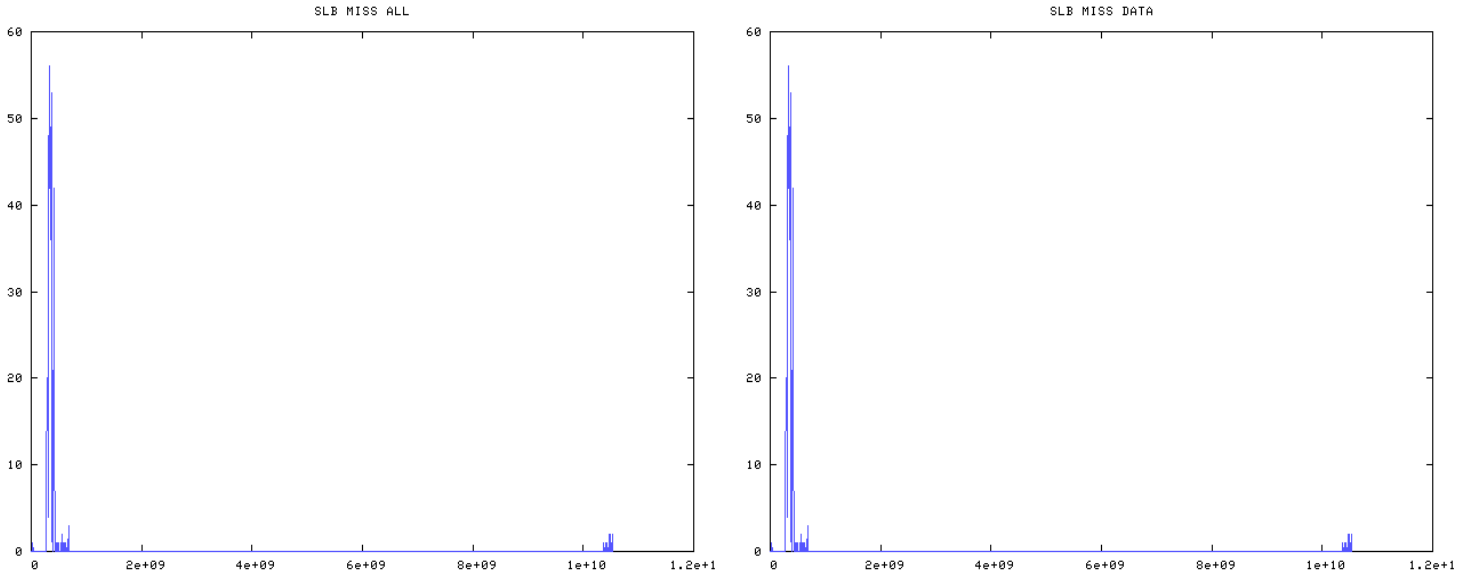

Figure 128: All SLB misses (left) and Data SLB Misses (right) livegraphs

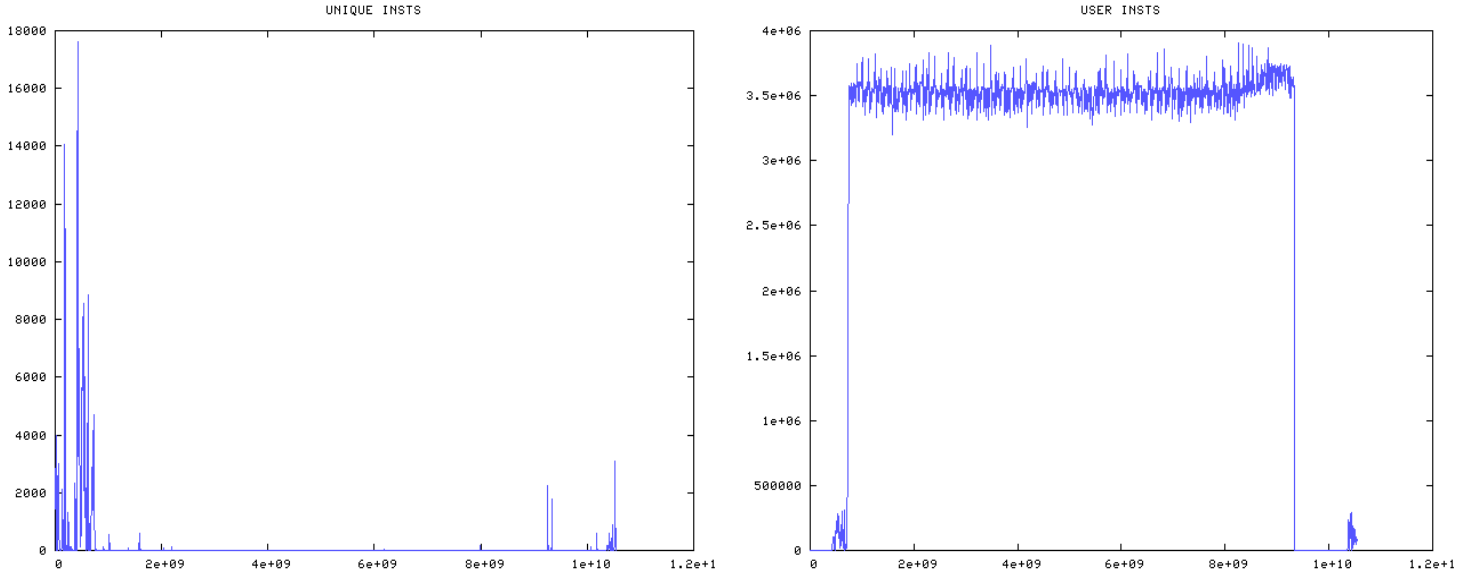

Figure 129: Unique Instructions (left) and User Instructions (right) livegraphs

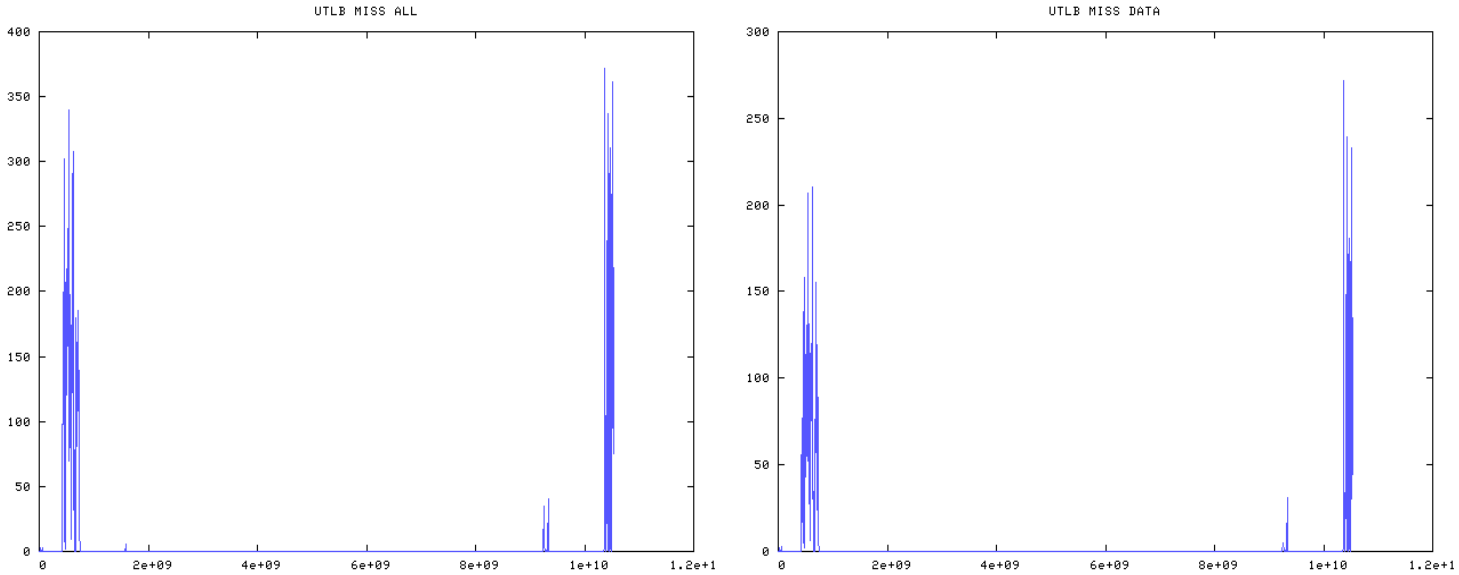

Figure 130: All User TLB Misses (left) and Data TLB Misses (right) livegraphs

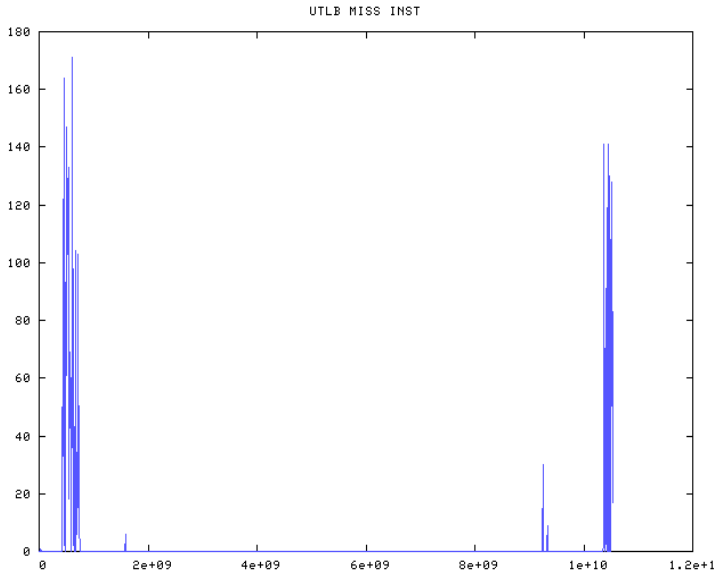

Figure 131: User TLB Instruction Miss livegraphs

B.8 PREDATOR

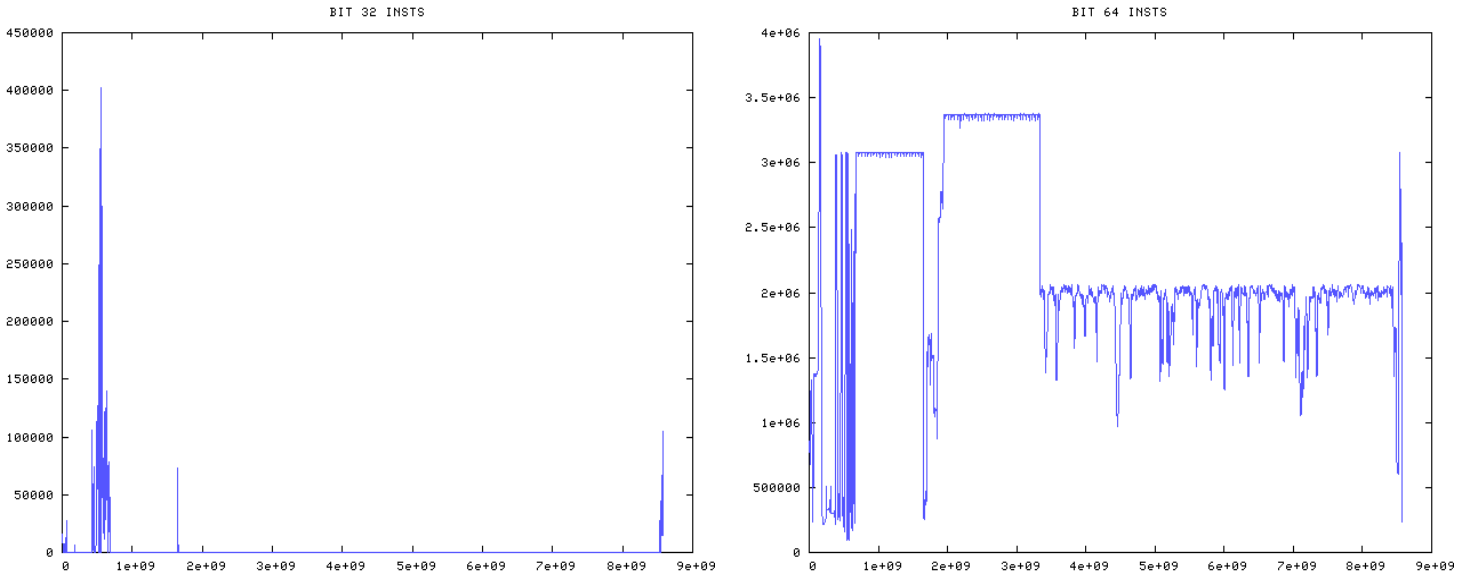

Figure 132: 32-bit (left) and 64-bit (right) instruction livegraphs

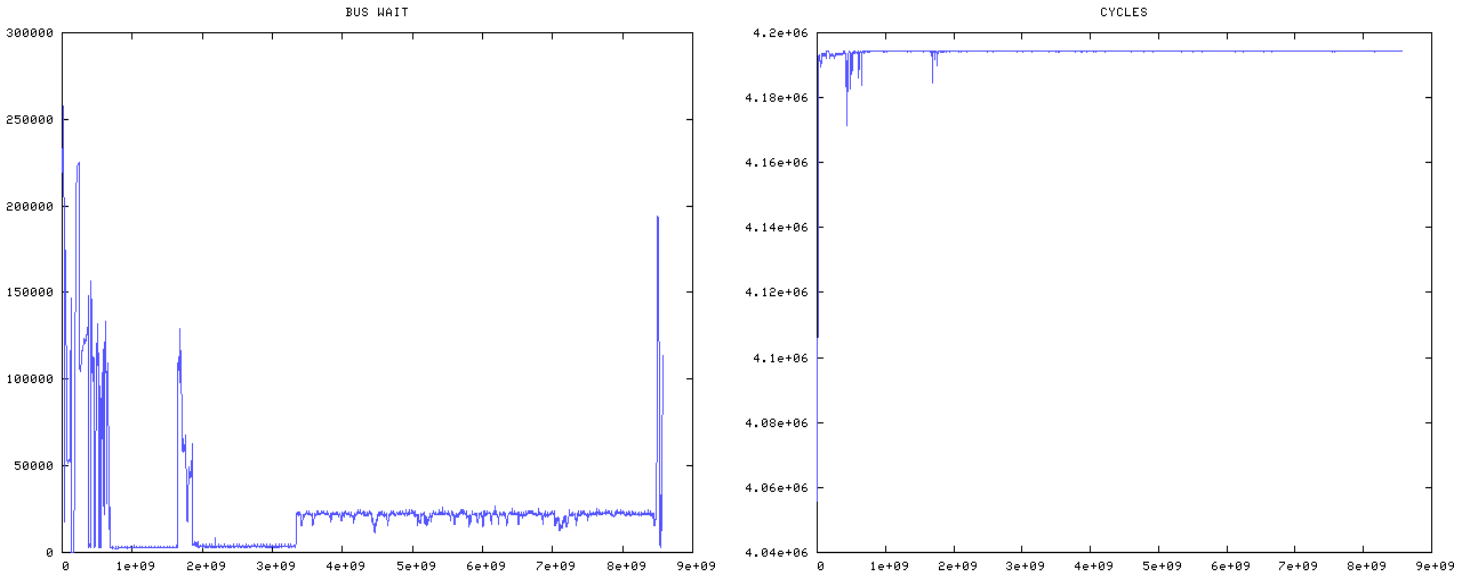

Figure 133: Bus-wait (left) and Cycles (right) livegraphs

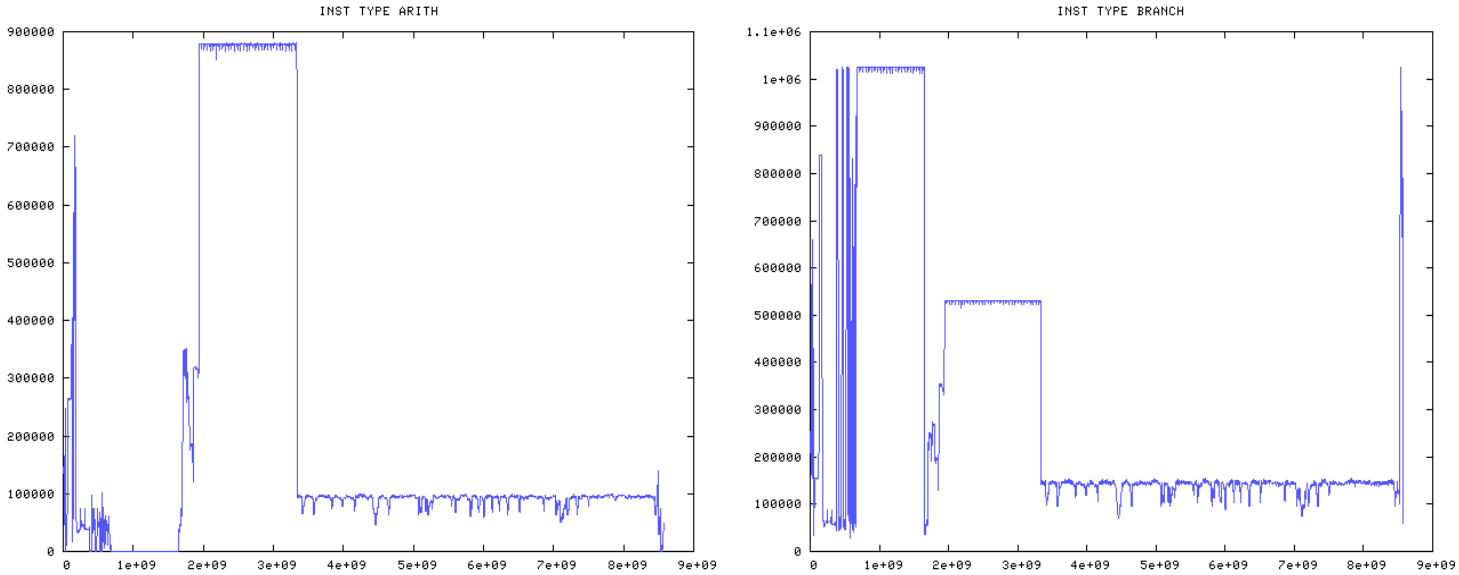

Figure 134: Arithmetic (left) and Branch (right) instructions livegraphs

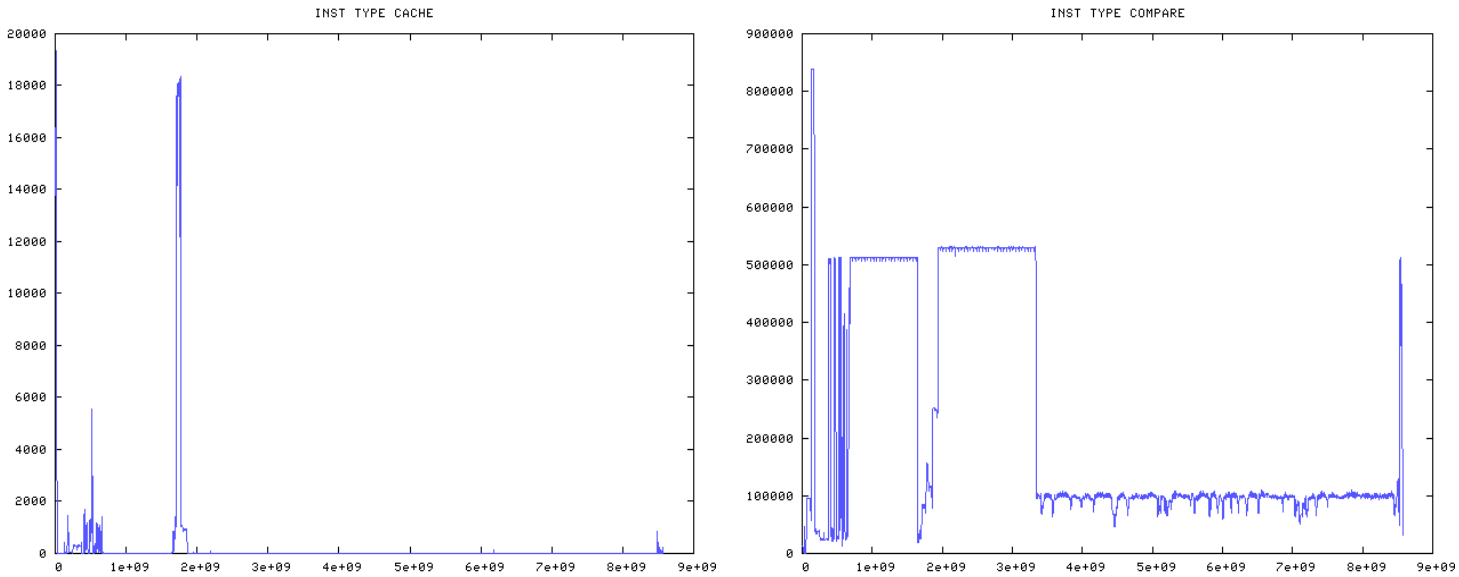

Figure 135: Cache (left) and Compare (right) instructions livegraphs

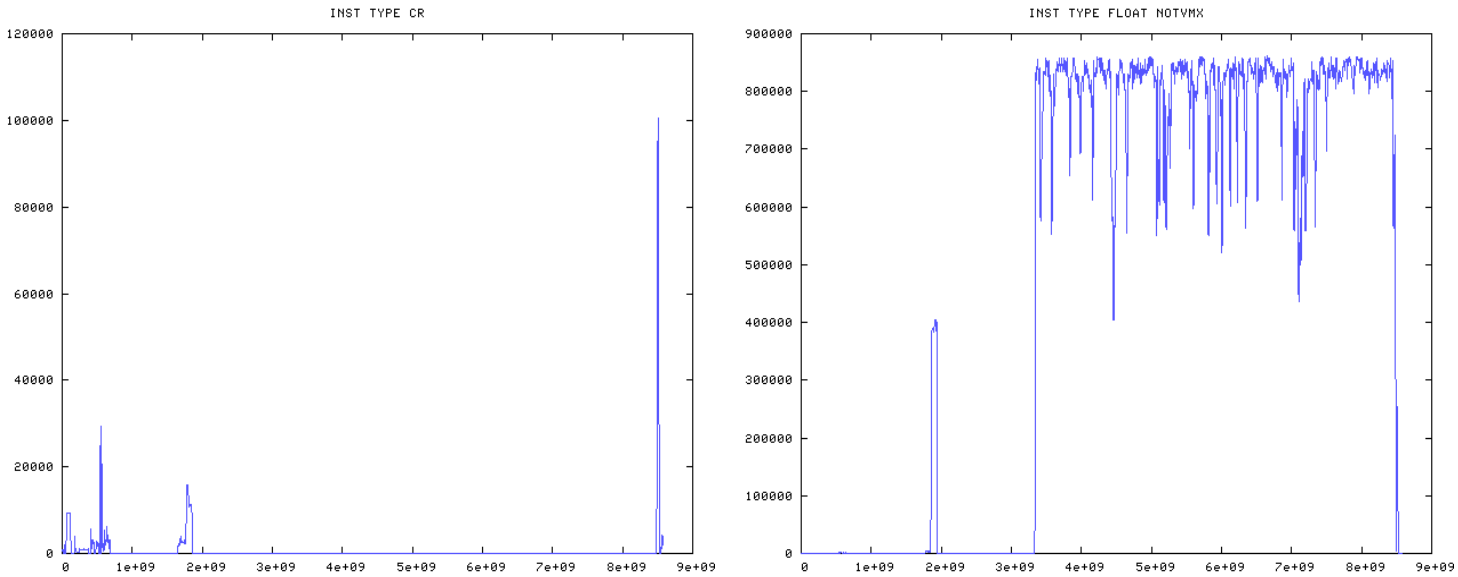

Figure 136: CR (left) and Not VMX-Float (right) instructions livegraphs

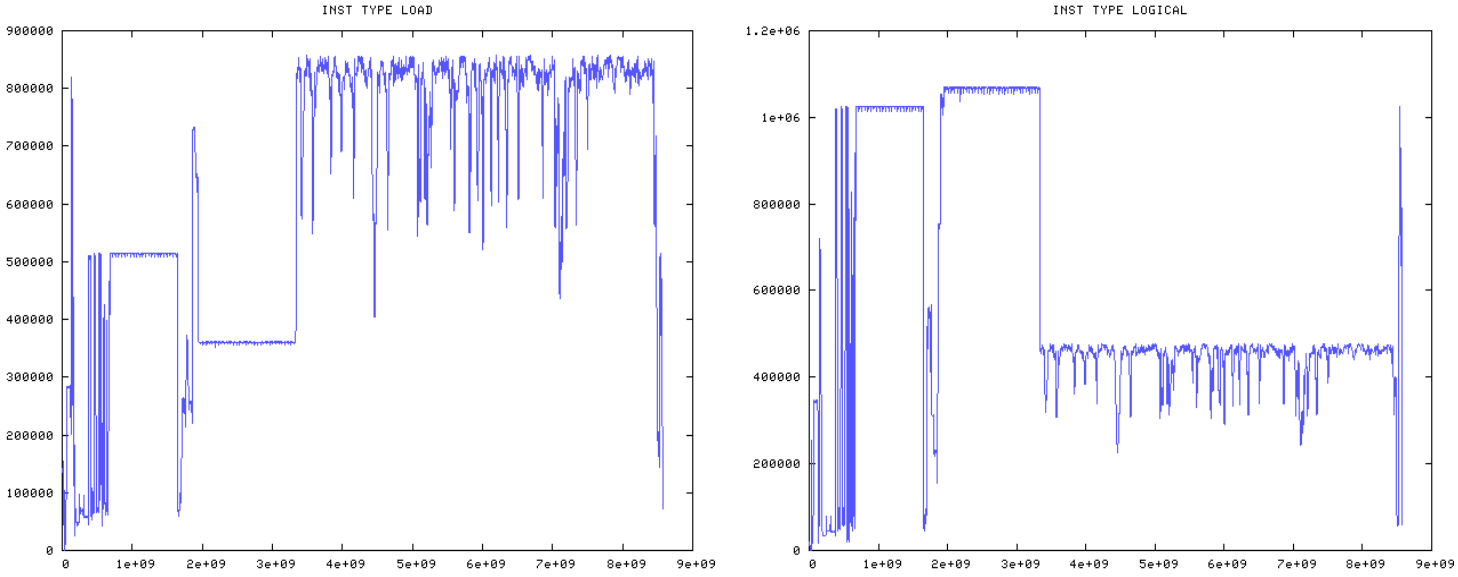

Figure 137: Load (left) and Logical (right) instructions livegraphs

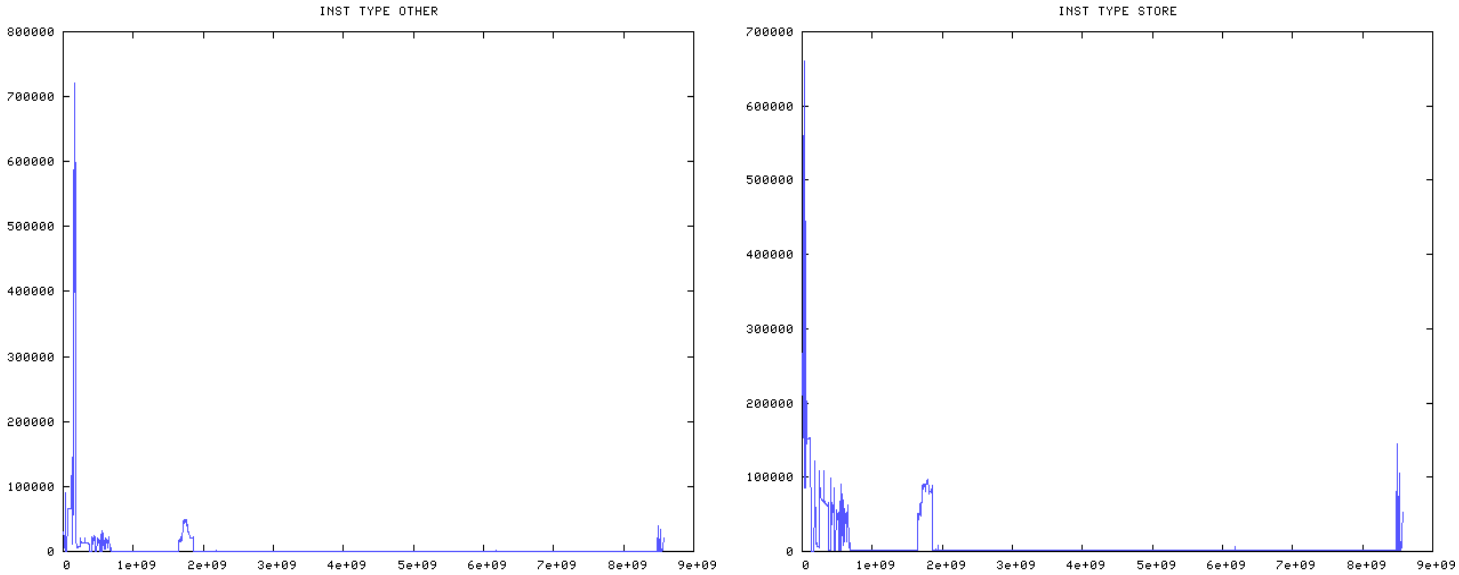

Figure 138: Other (left) and Store (right) instructions livegraphs

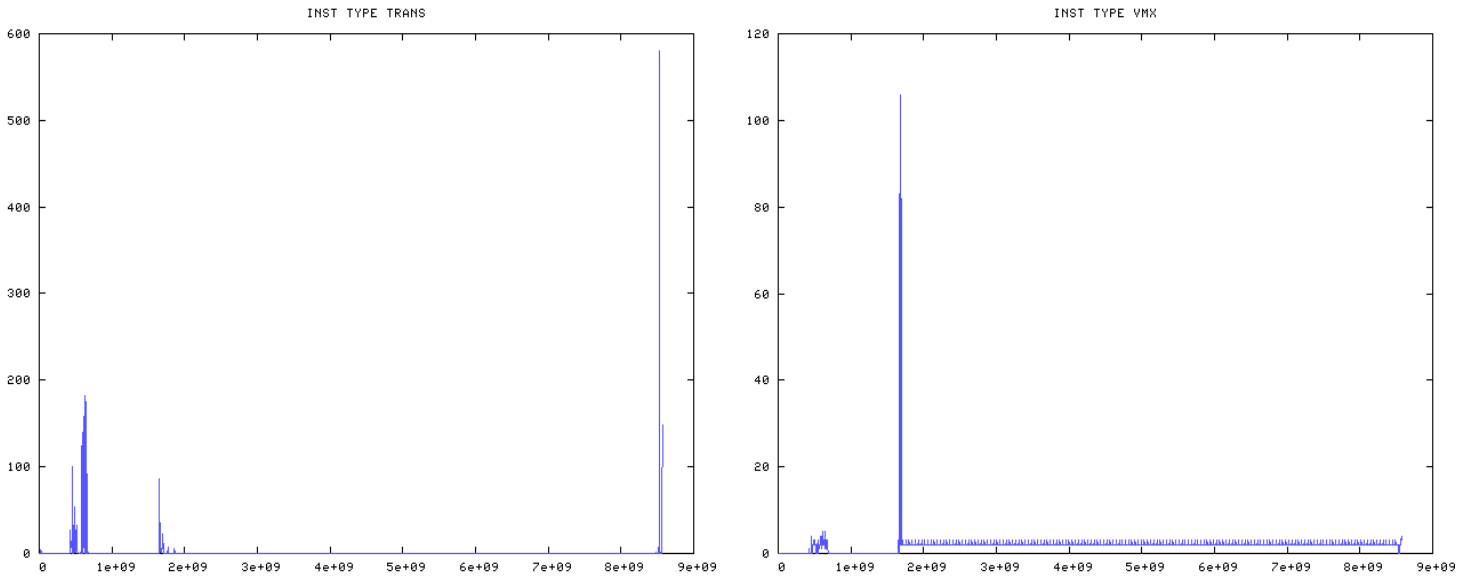

Figure 139: Trans (left) and VMX (right) instructions livegraphs

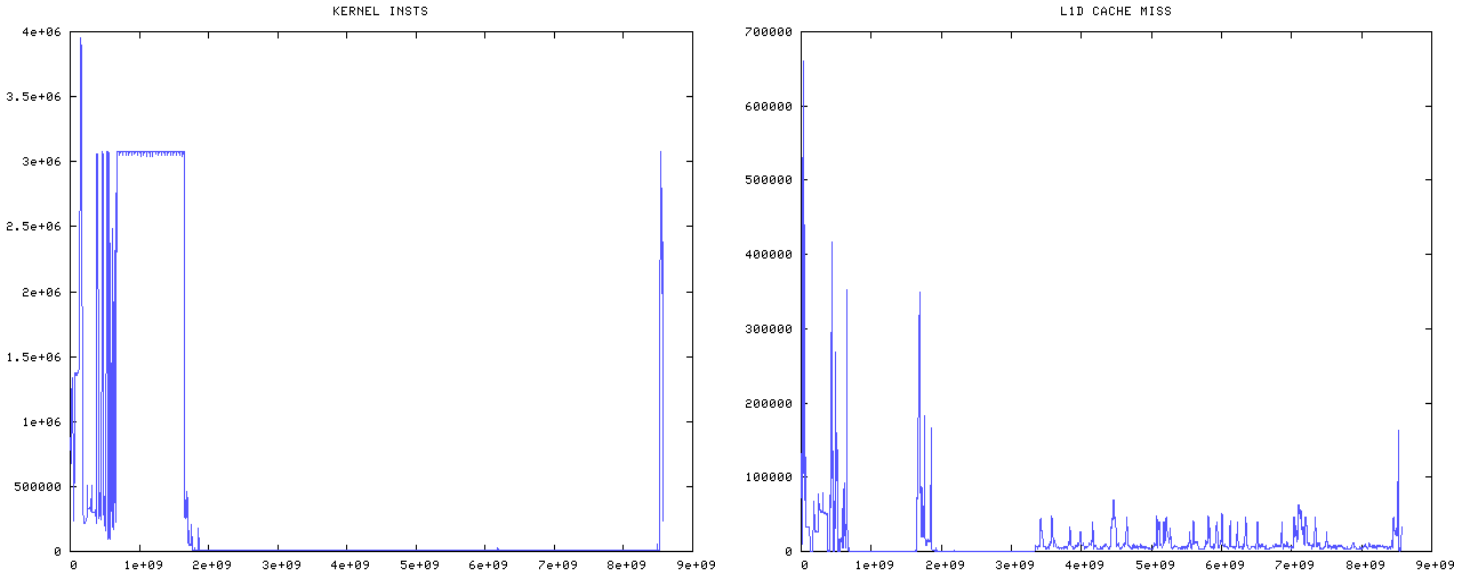

Figure 140: Kernel instructions (left) and L1D Cache Miss (right) livegraphs

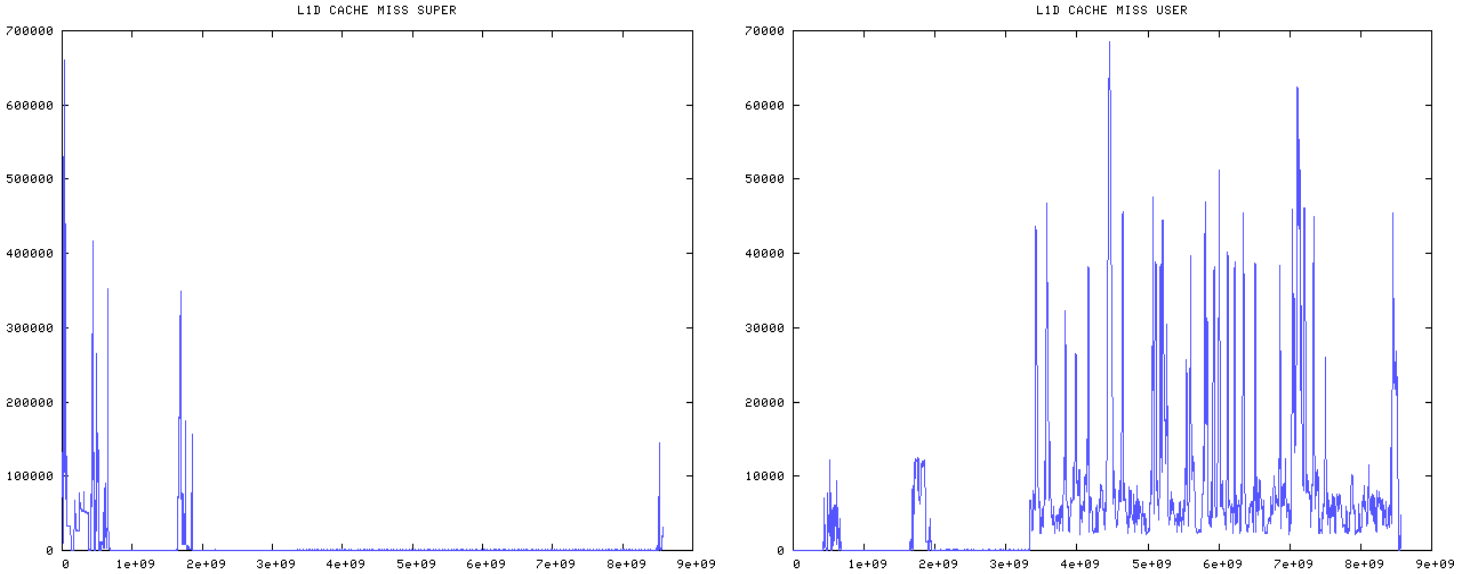

Figure 141: L1d Cache Miss Super (left) and L1d Cache Miss User (right) livegraphs

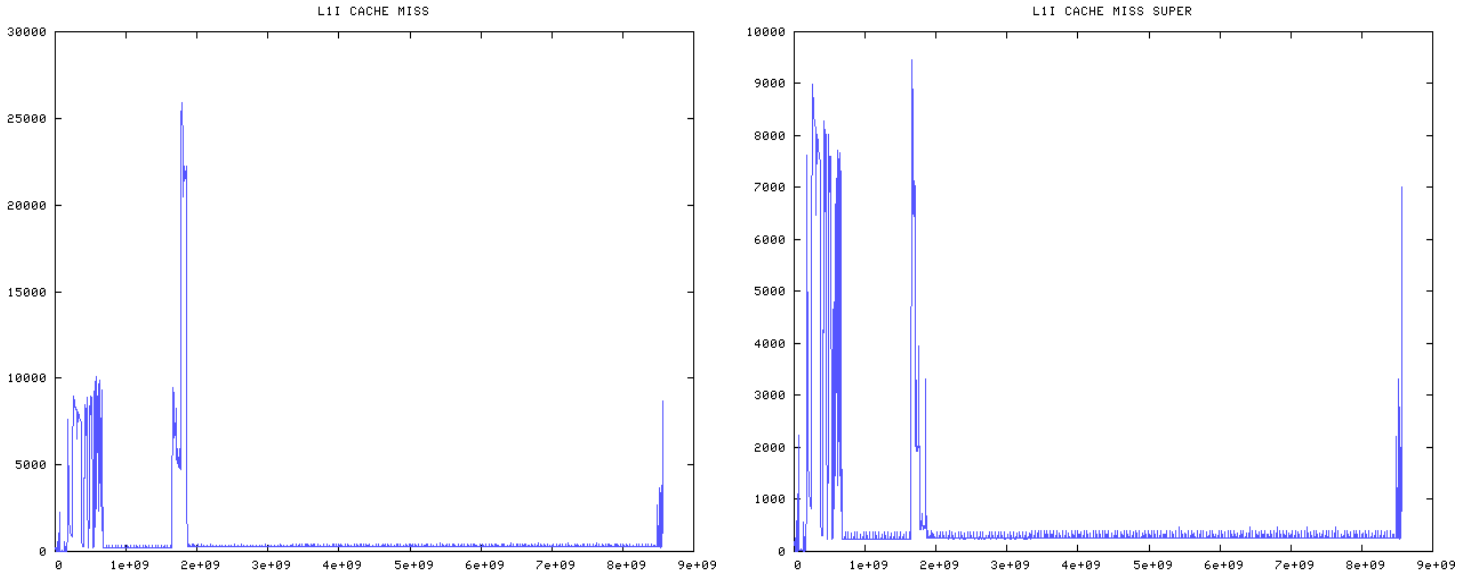

Figure 142: L1I Cache Miss (left) and L1I Cache Miss Super (right) livegraphs

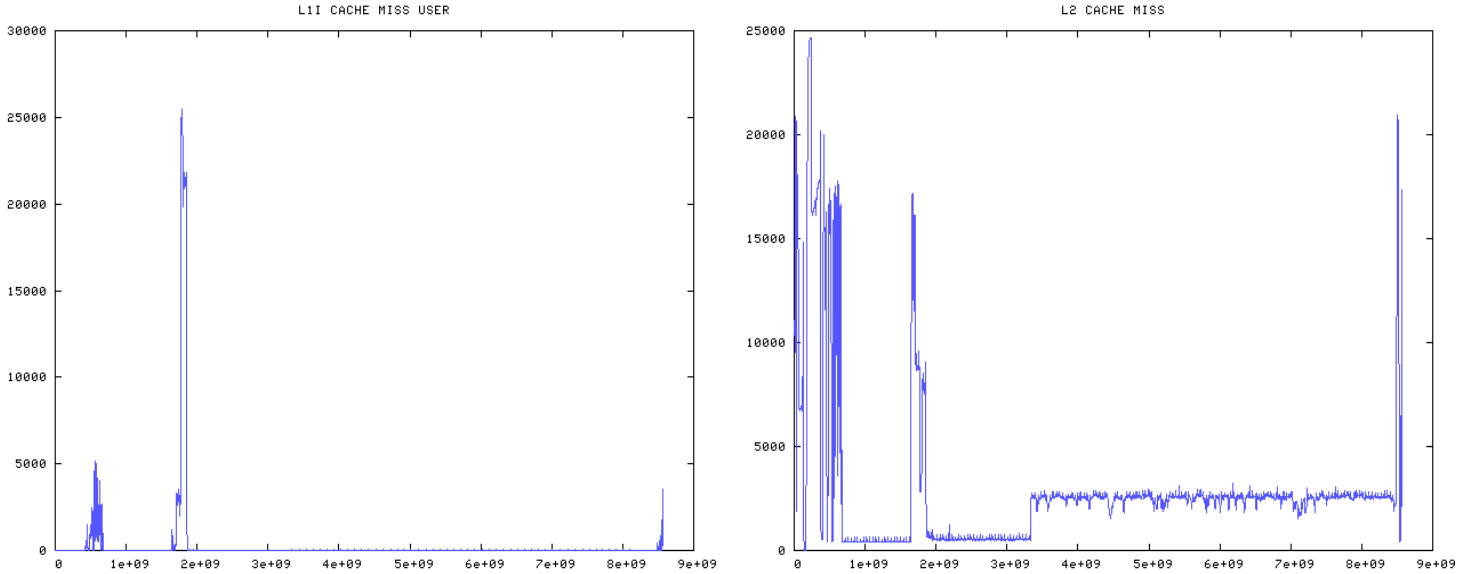

Figure 143: L1I Cache Miss User (left) and L2 Cache Miss (right) livegraphs

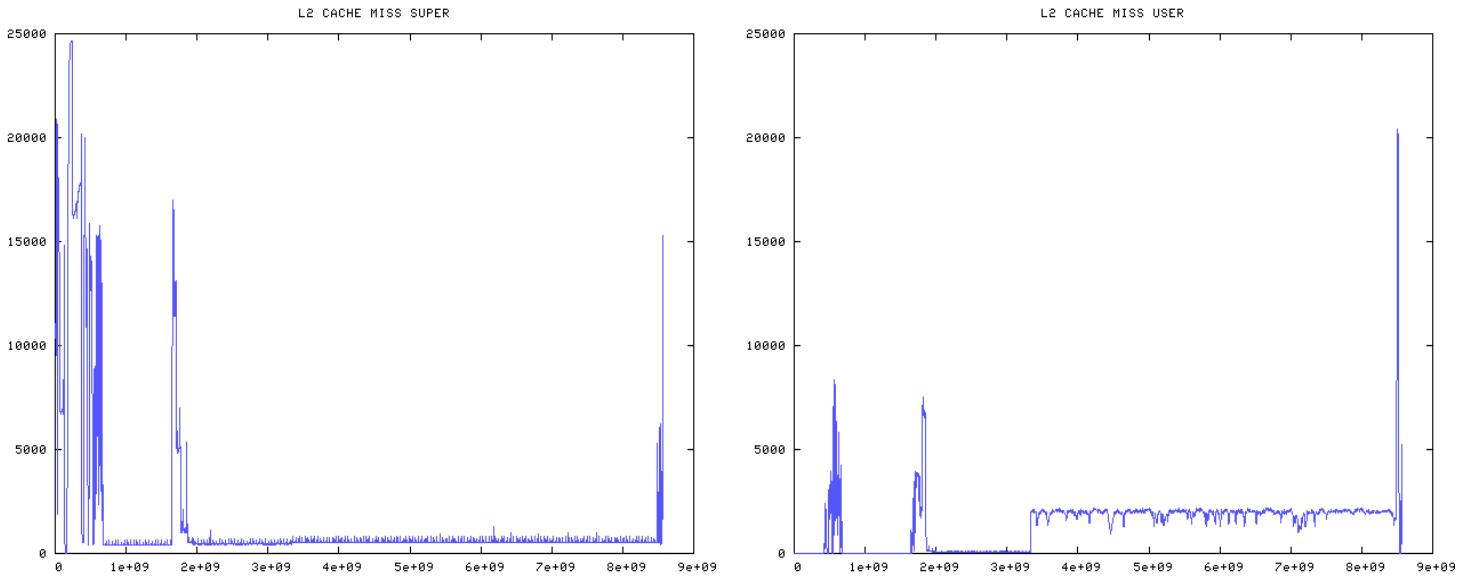

Figure 144: L2 Cache Miss Super (left) and L2 Cache Miss User (right) livegraphs

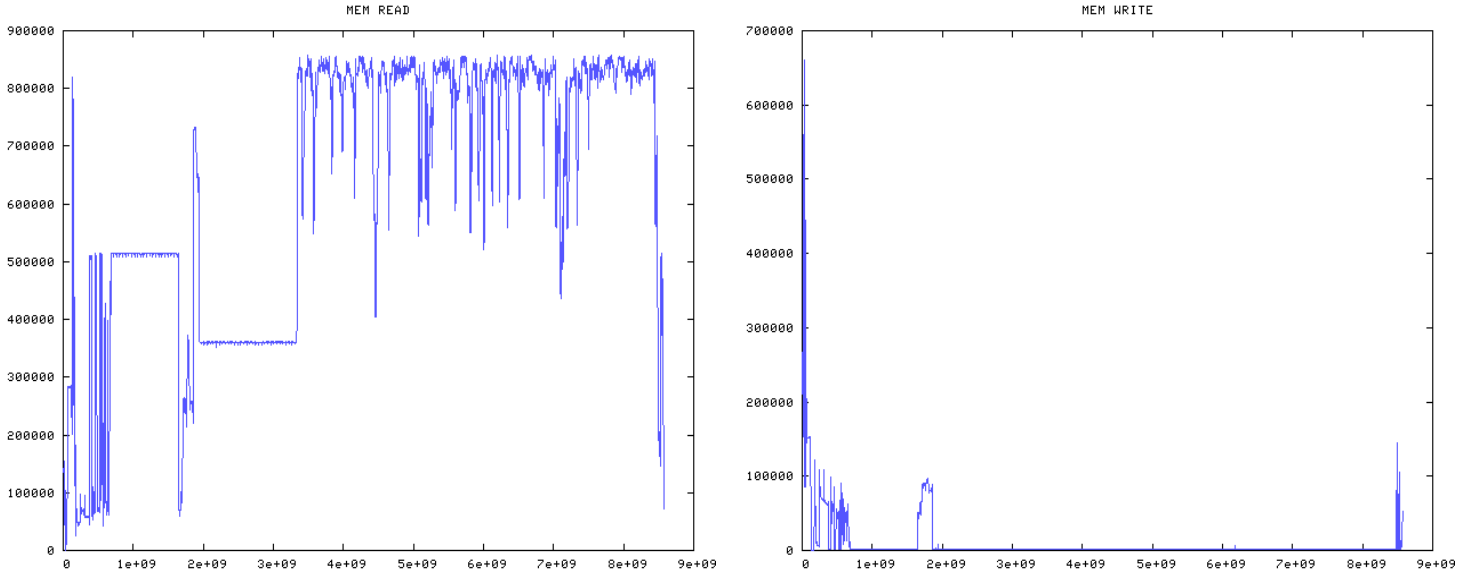

Figure 145: Memory-read (left) and Memory-Write (right) livegraphs

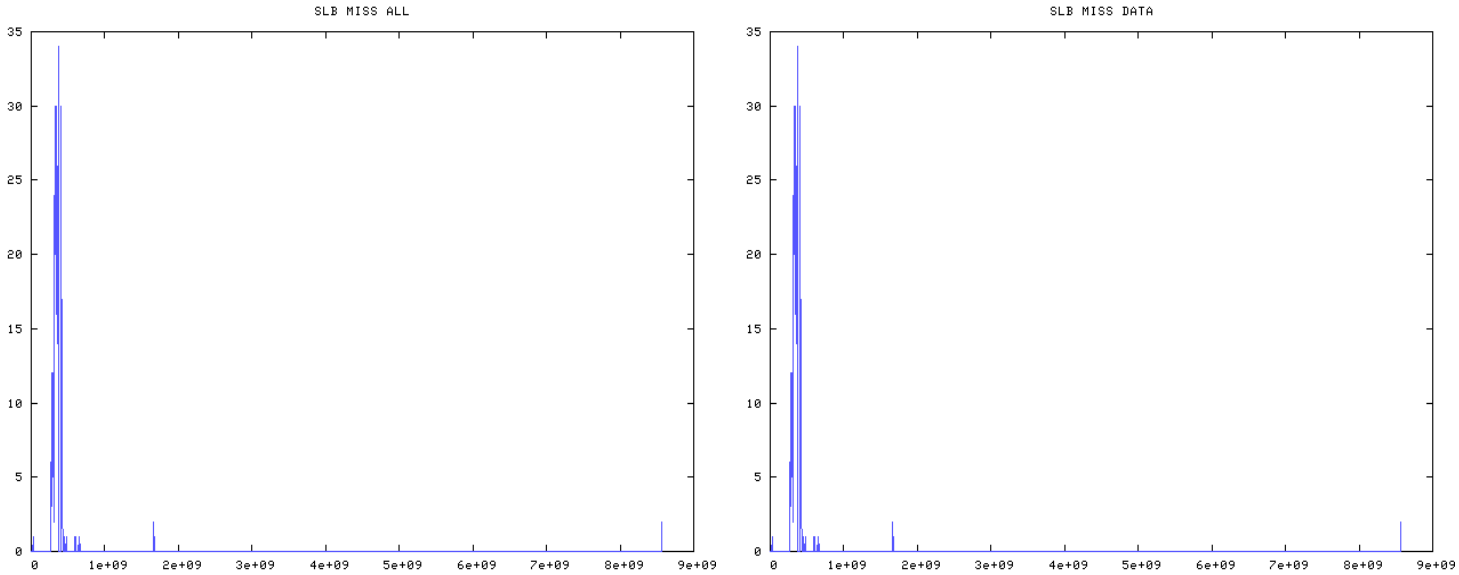

Figure 146: All SLB misses (left) and Data SLB Misses (right) livegraphs

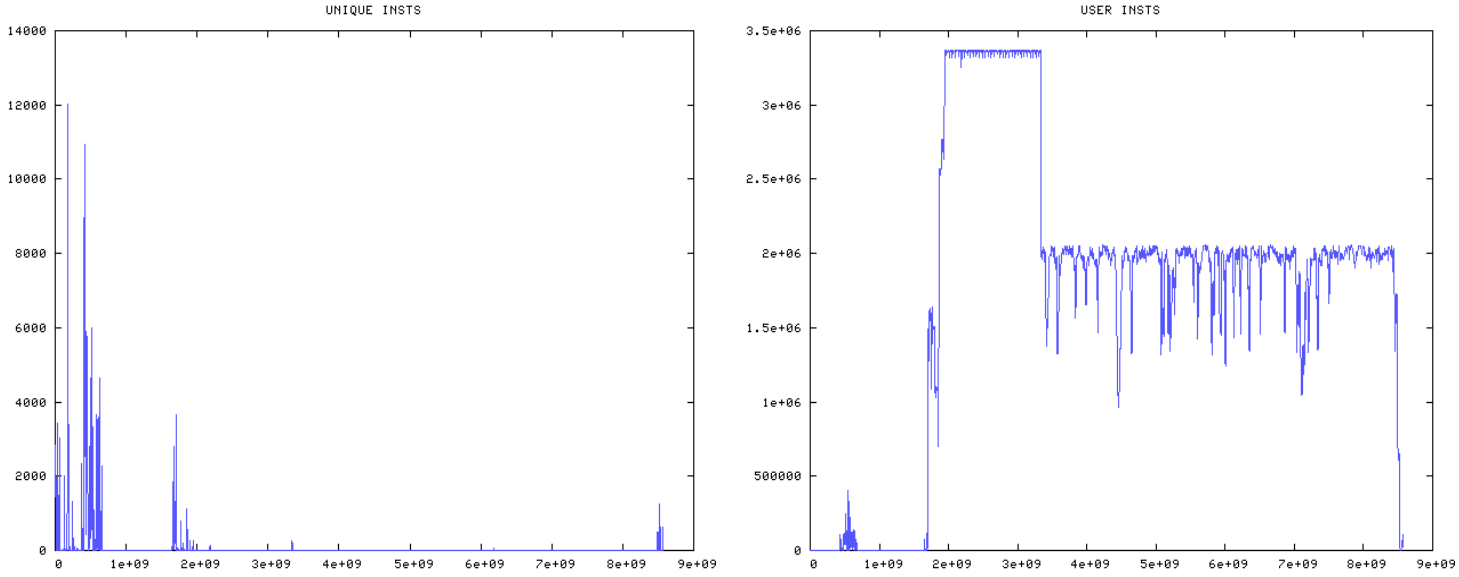

Figure 147: Unique Instructions (left) and User Instructions (right) livegraphs

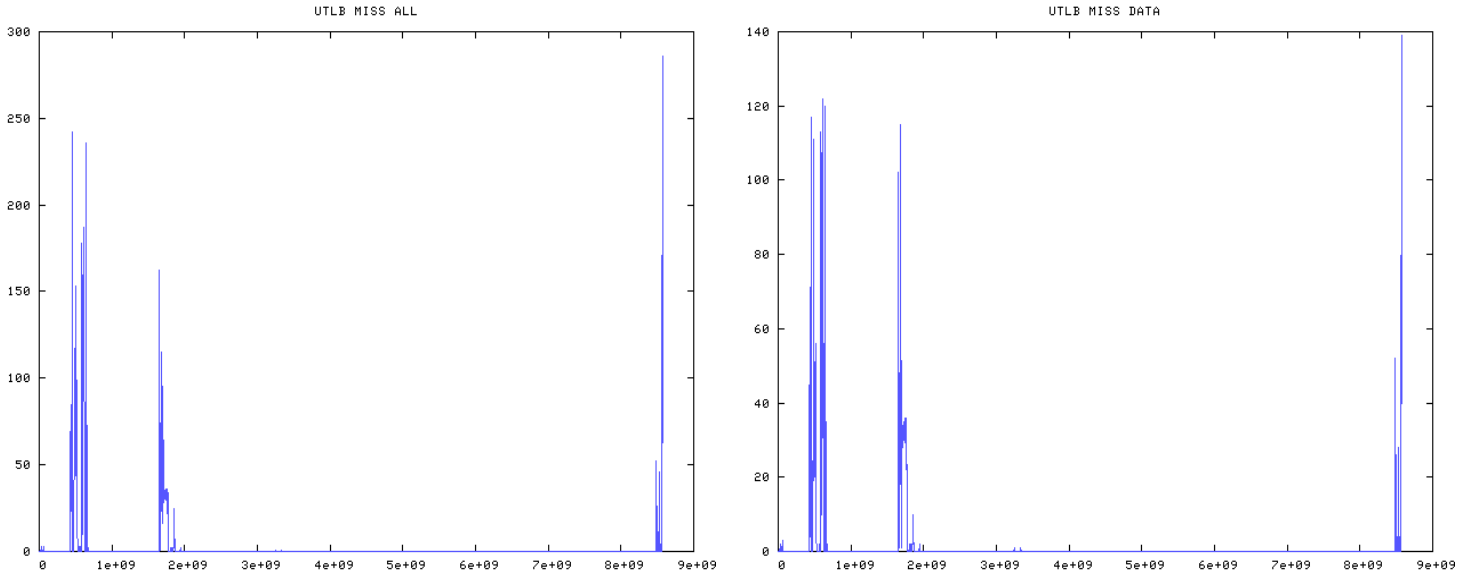

Figure 148: All User TLB Misses (left) and Data TLB Misses (right) livegraphs

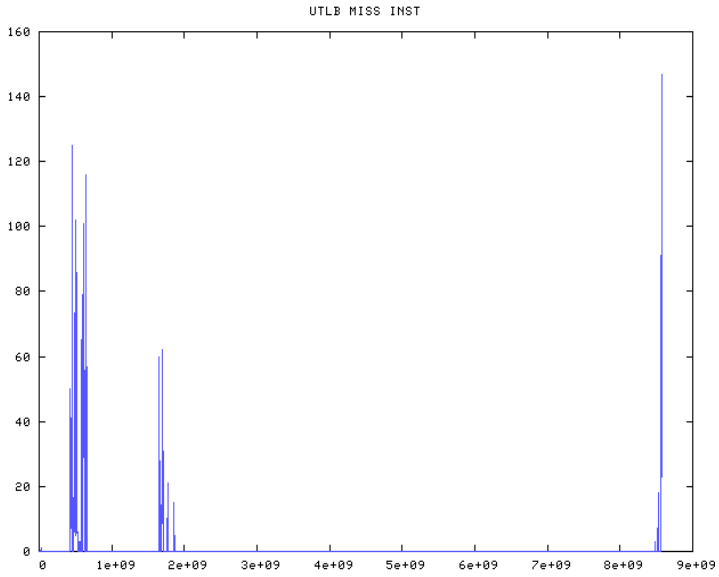

Figure 149: User TLB Instruction Miss livegraphs

C Apple G5 CHUD

C.1 BLAST

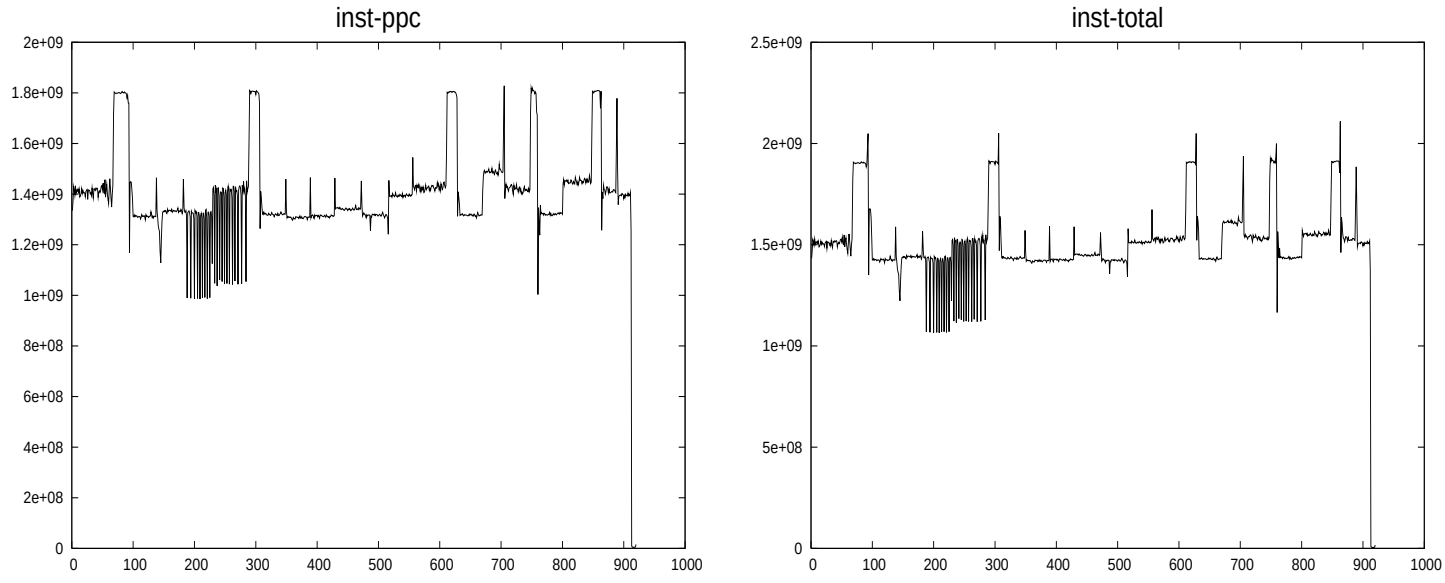

Figure 150: Instructions(ppc) (left) and Instructions (total) (right) livegraphs

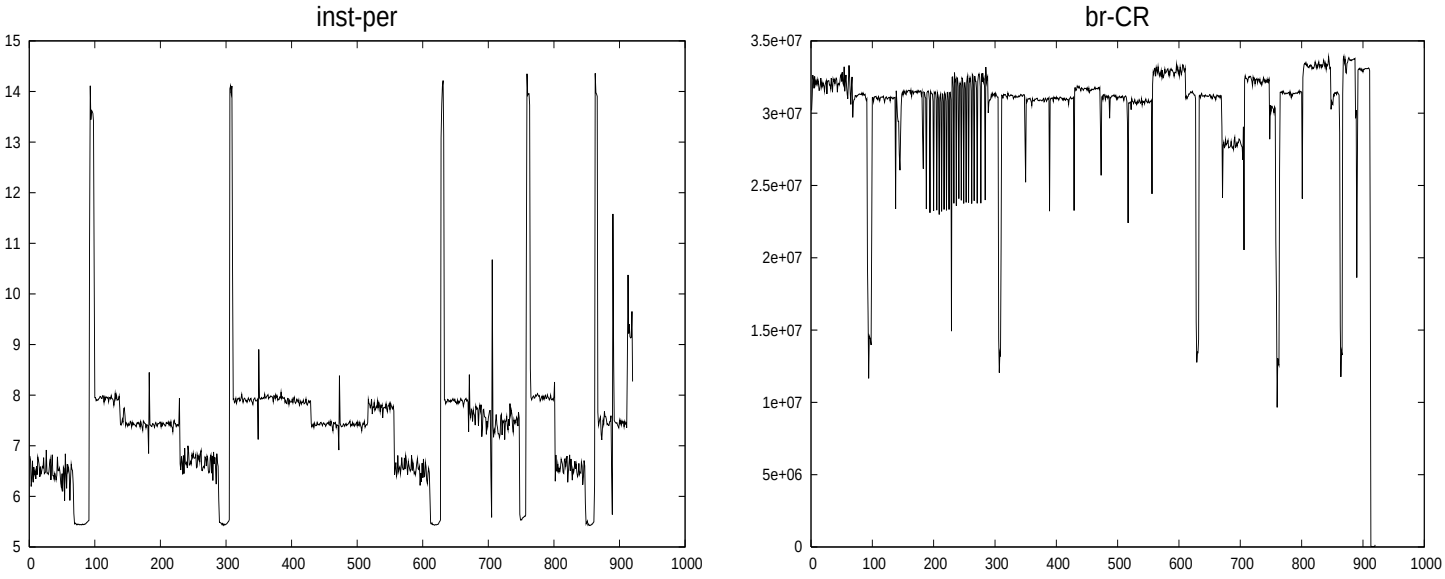

Figure 151: Percentage of io/ld/st instruction (left) and Branch Mispredicts (Condition Register) (right) livegraphs

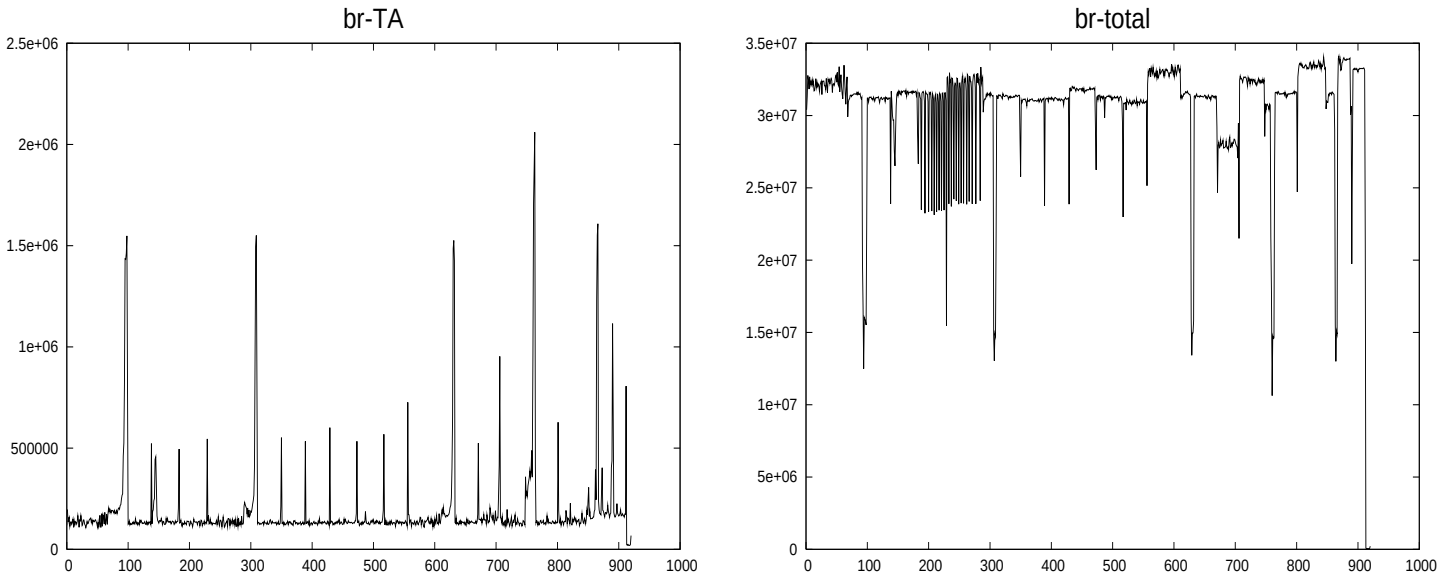

Figure 152: Branch Mispredicts (Target Address) (left) and Branch Mispredicts (total) (right) livegraphs

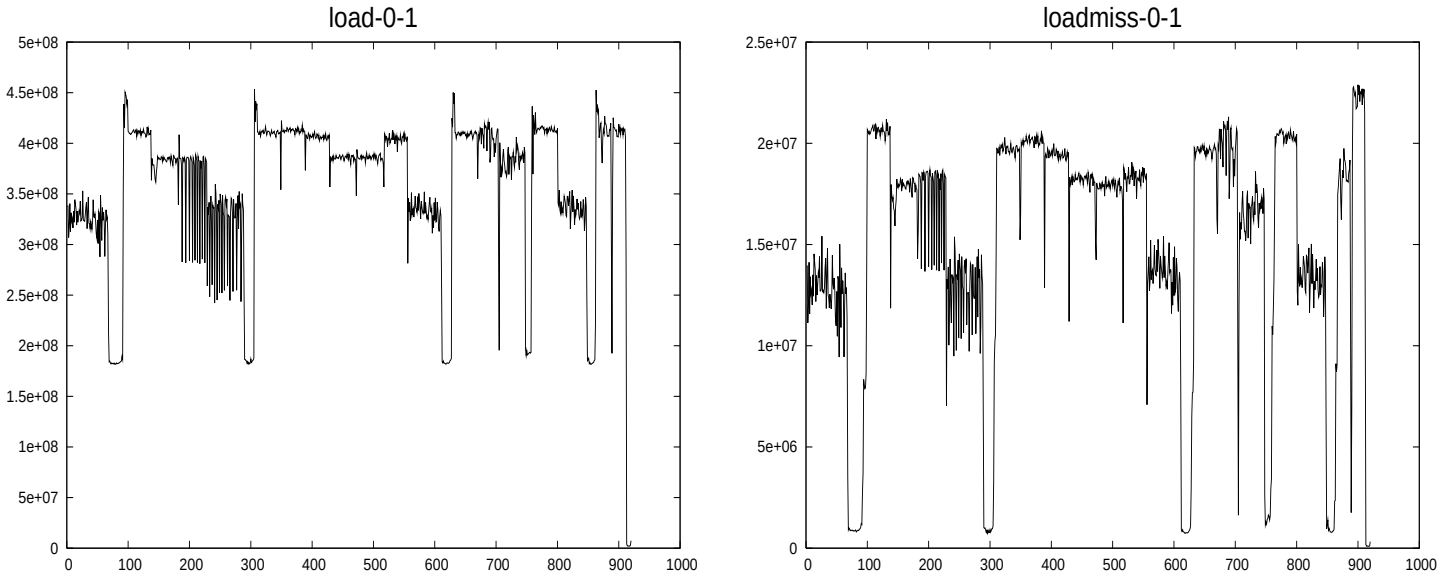

Figure 153: Loads (left) and Load Misses (right) livegraphs

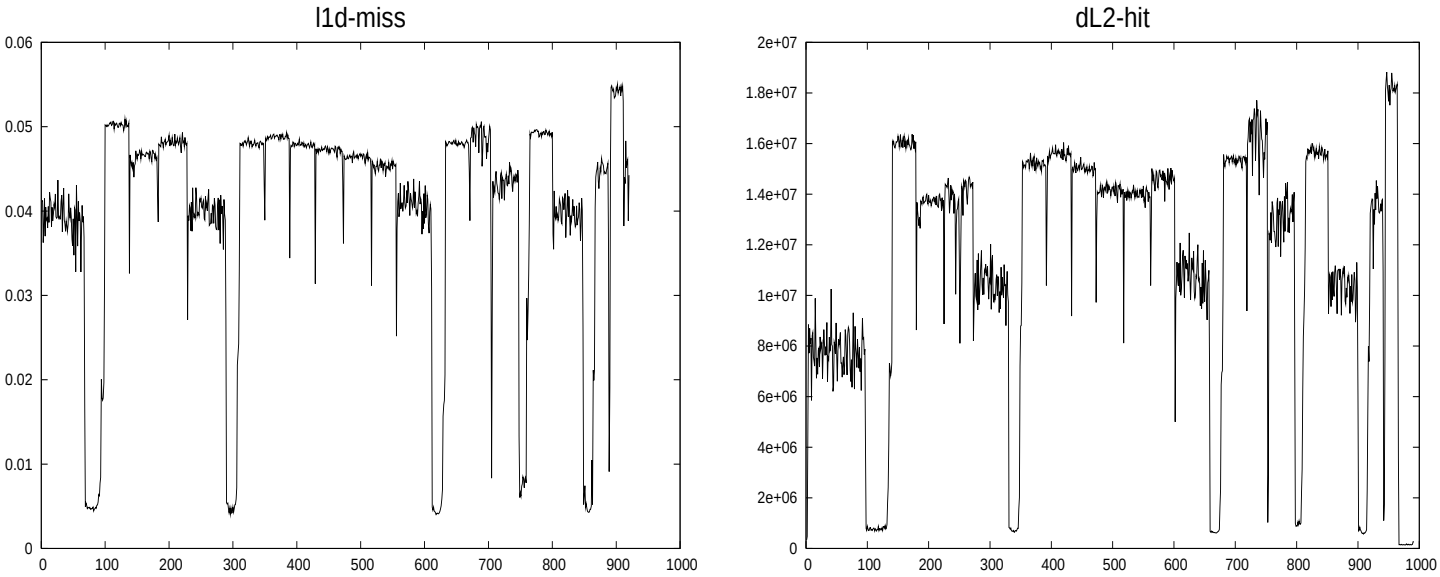

Figure 154: L1 data misses (left) and L2 data hits (right) livegraphs

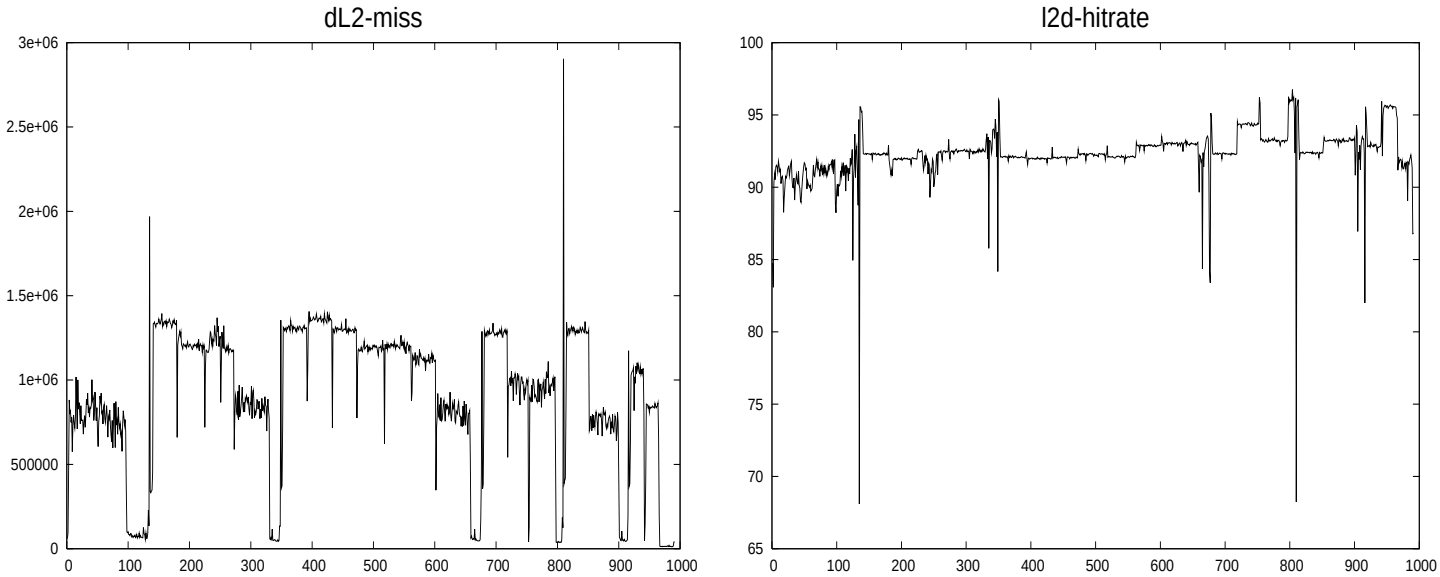

Figure 155: L2 data misses (left) and L2 data hitrate (right) livegraphs

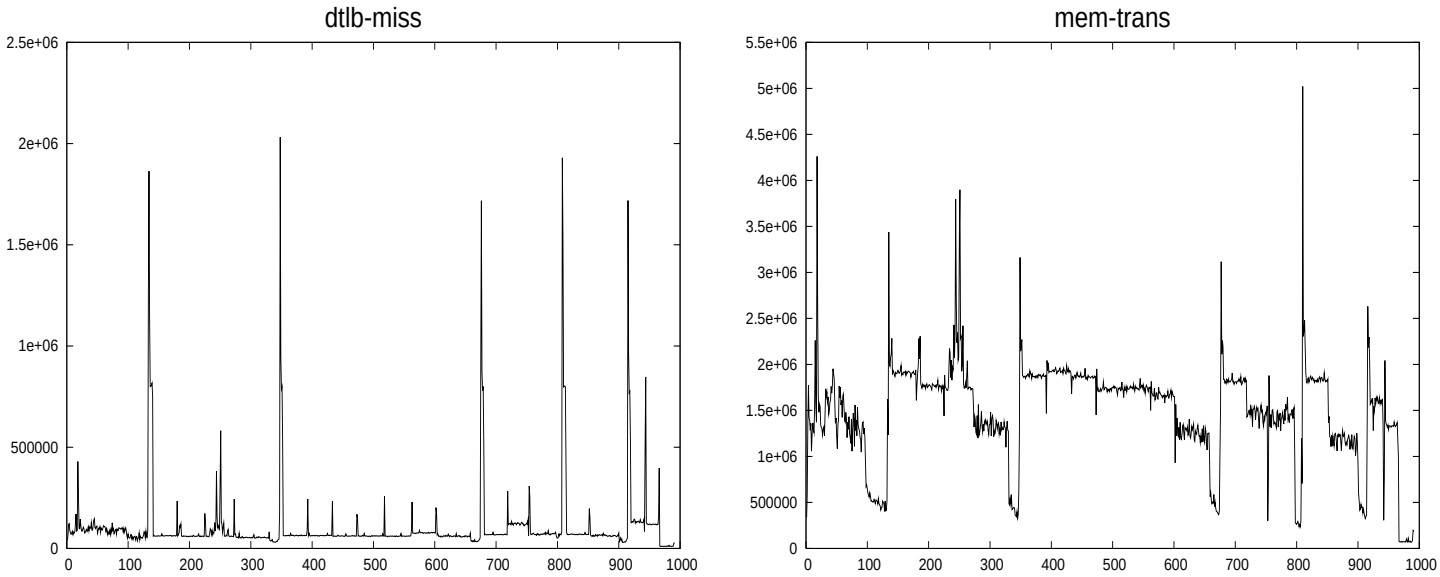

Figure 156: Data TLB misses (left) and Memory Transactions (right) livegraphs

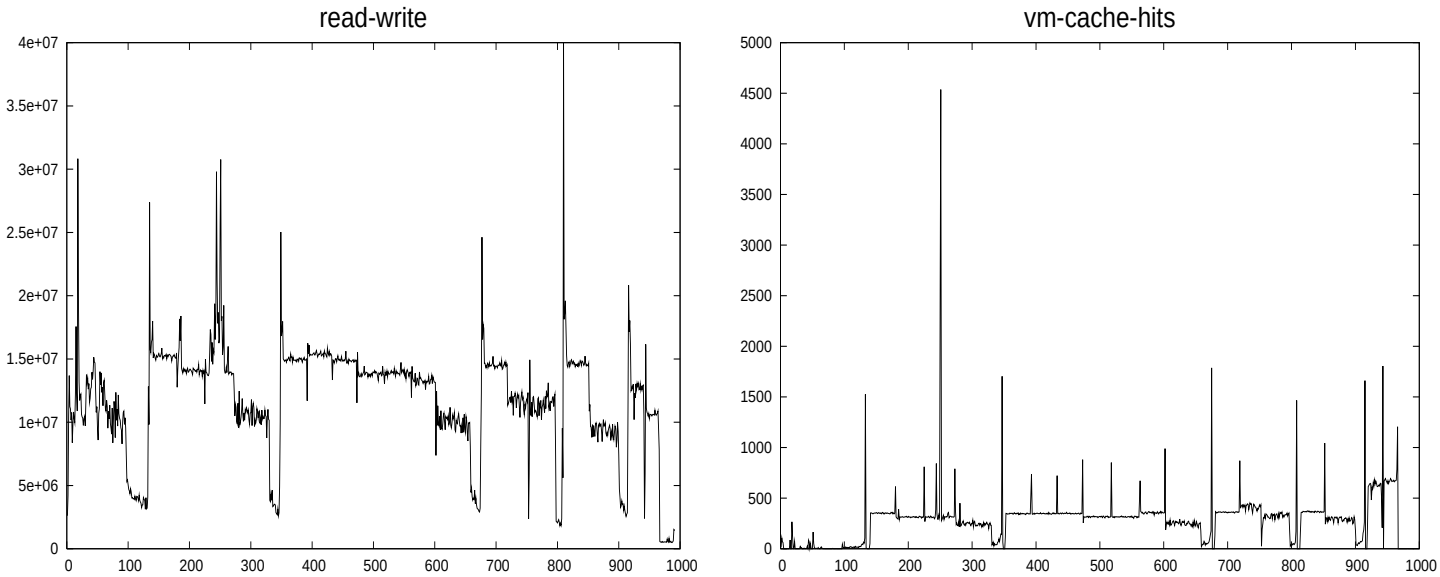

Figure 157: Read/Writes (left) and VM Page Cache Hits (right) livegraphs

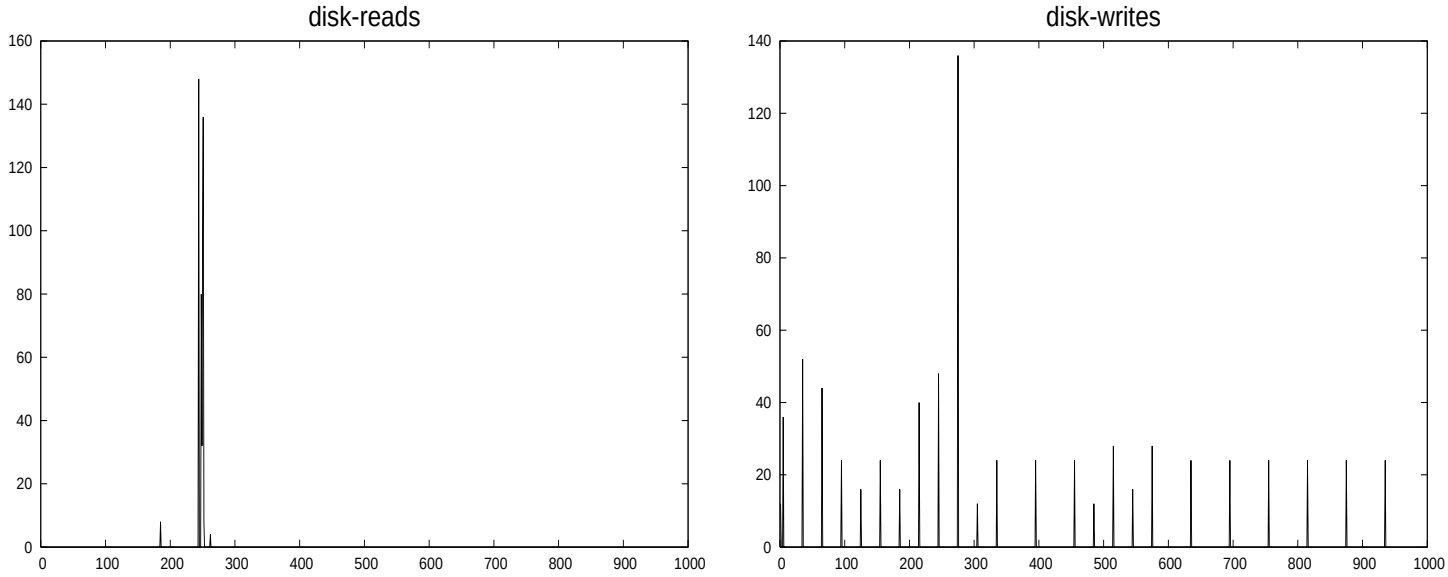

Figure 158: Disk Reads (left) and Disk Writes (right) livegraphs

C.2 CLUSTALW

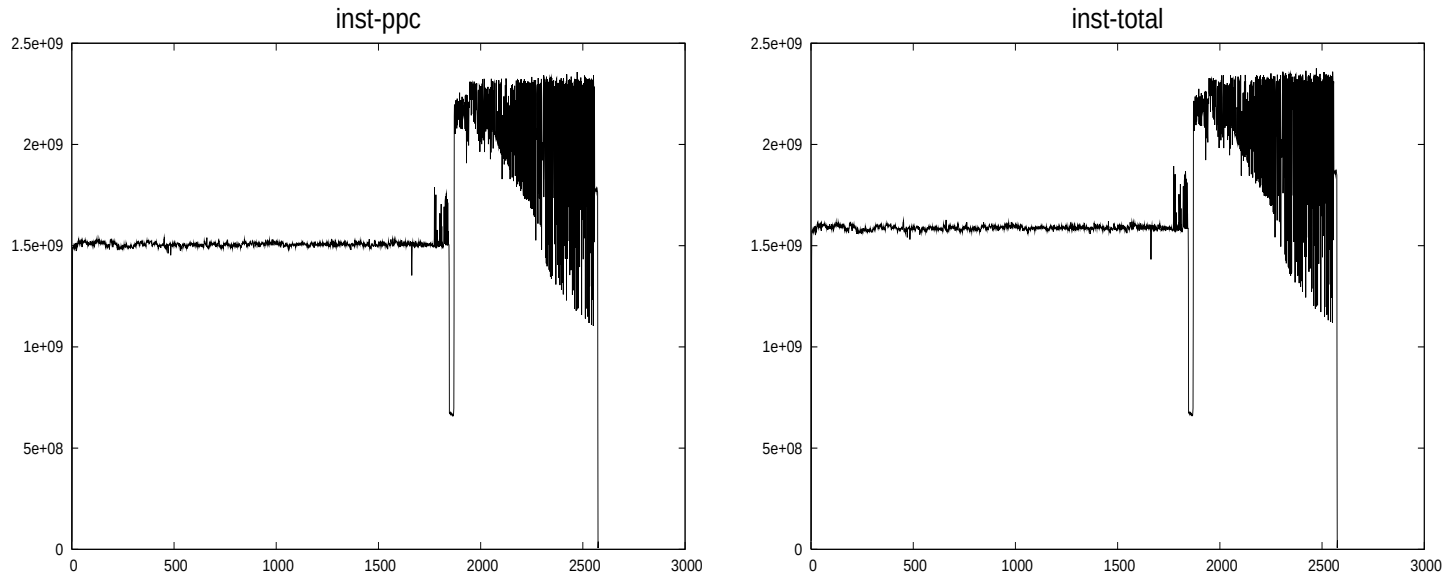

Figure 159: Instructions(ppc) (left) and Instructions (total) (right) livegraphs

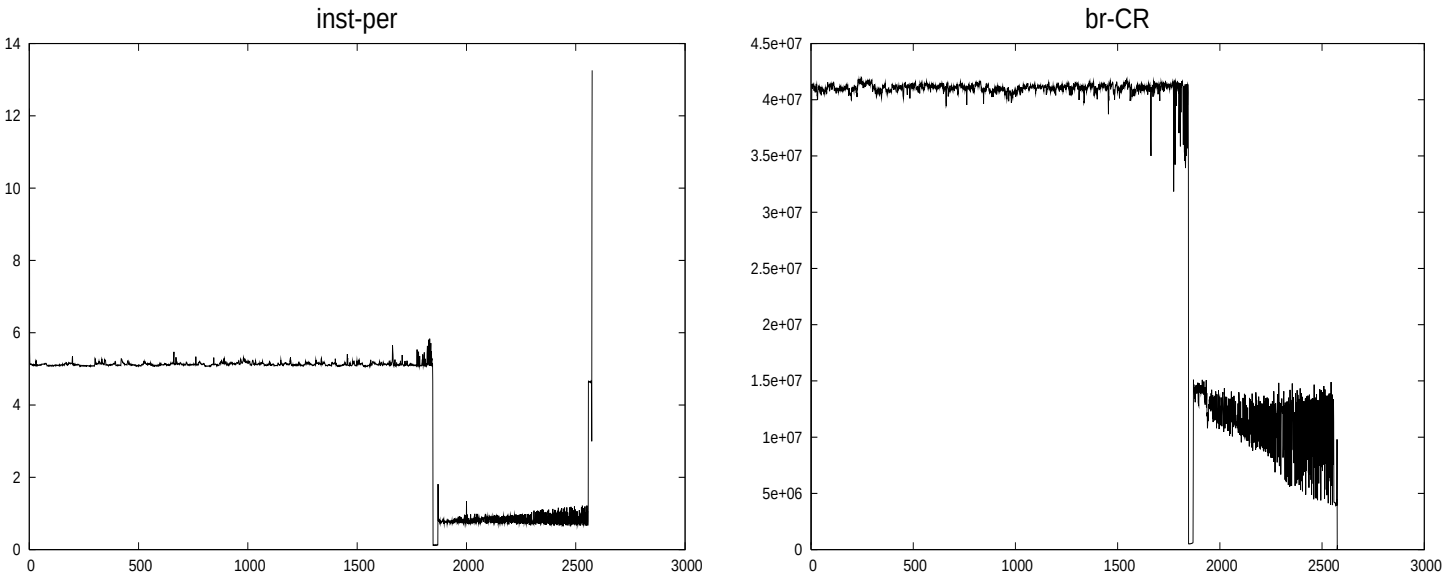

Figure 160: Percentage of io/ld/st instruction (left) and Branch Mispredicts (Condition Register) (right) livegraphs

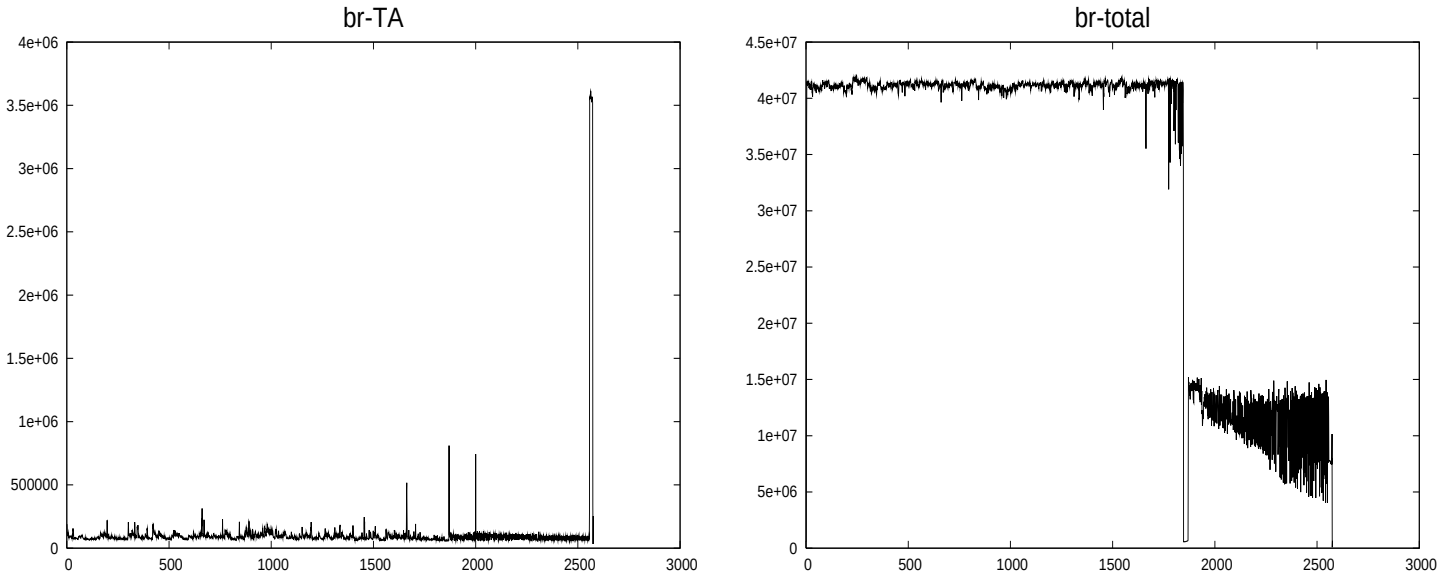

Figure 161: Branch Mispredicts (Target Address) (left) and Branch Mispredicts (total) (right) livegraphs

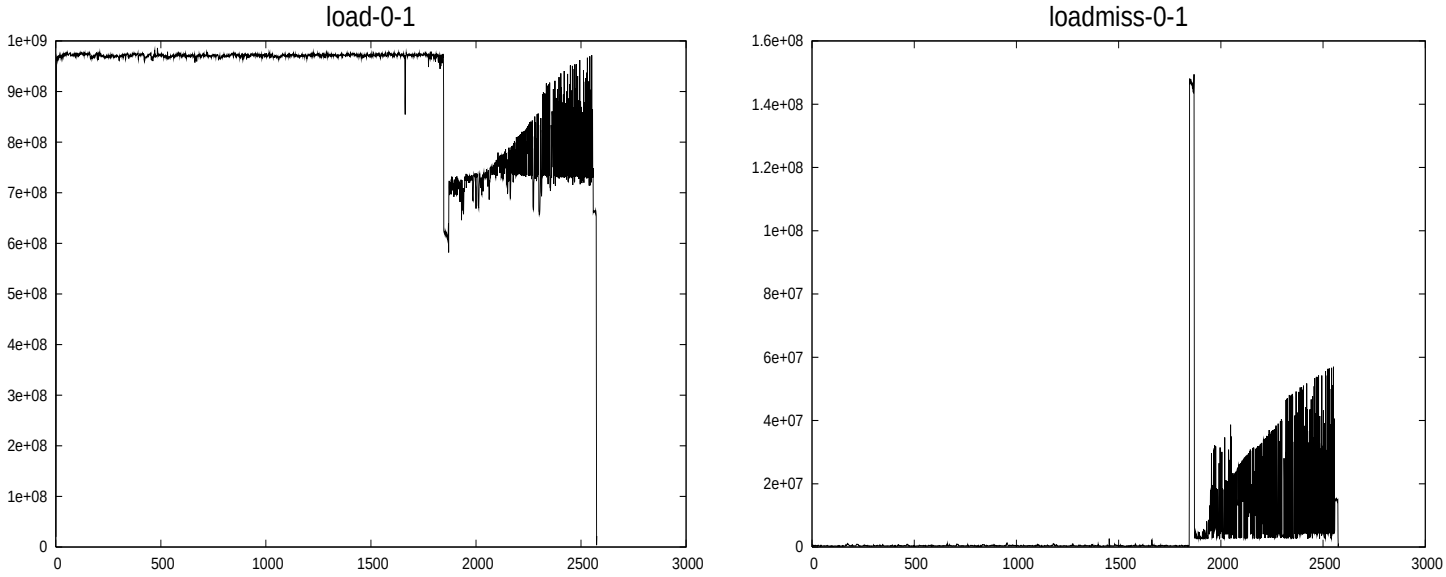

Figure 162: Loads (left) and Load Misses (right) livegraphs

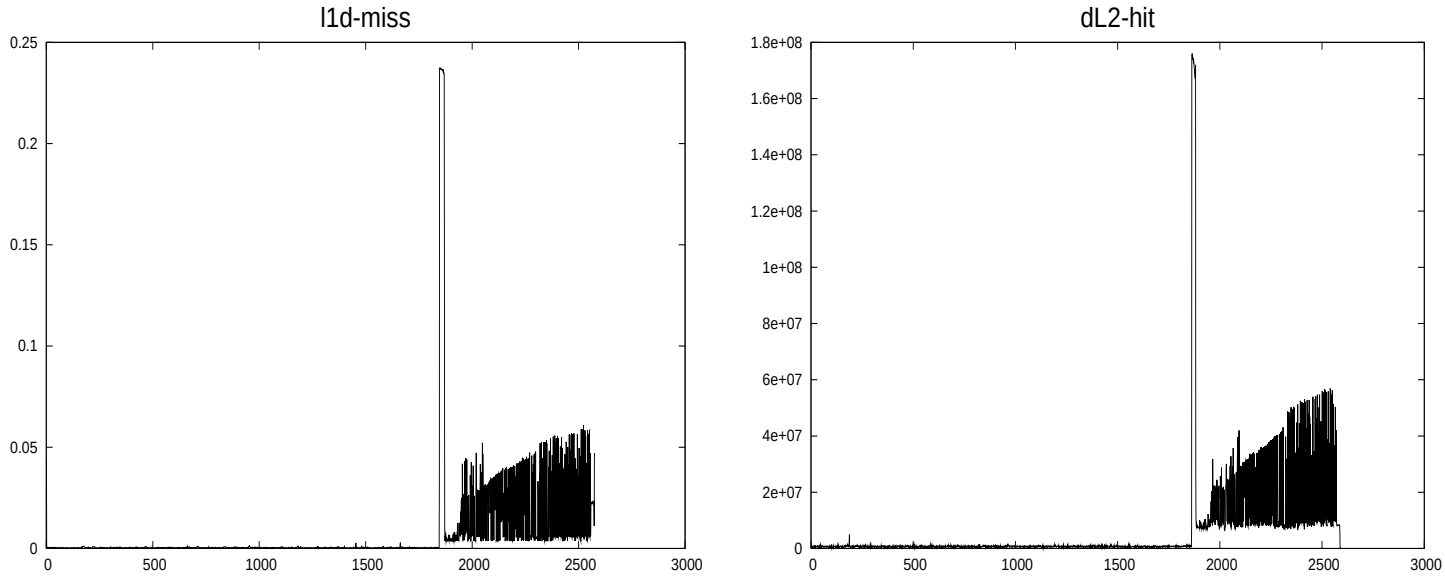

Figure 163: L1 data misses (left) and L2 data hits (right) livegraphs

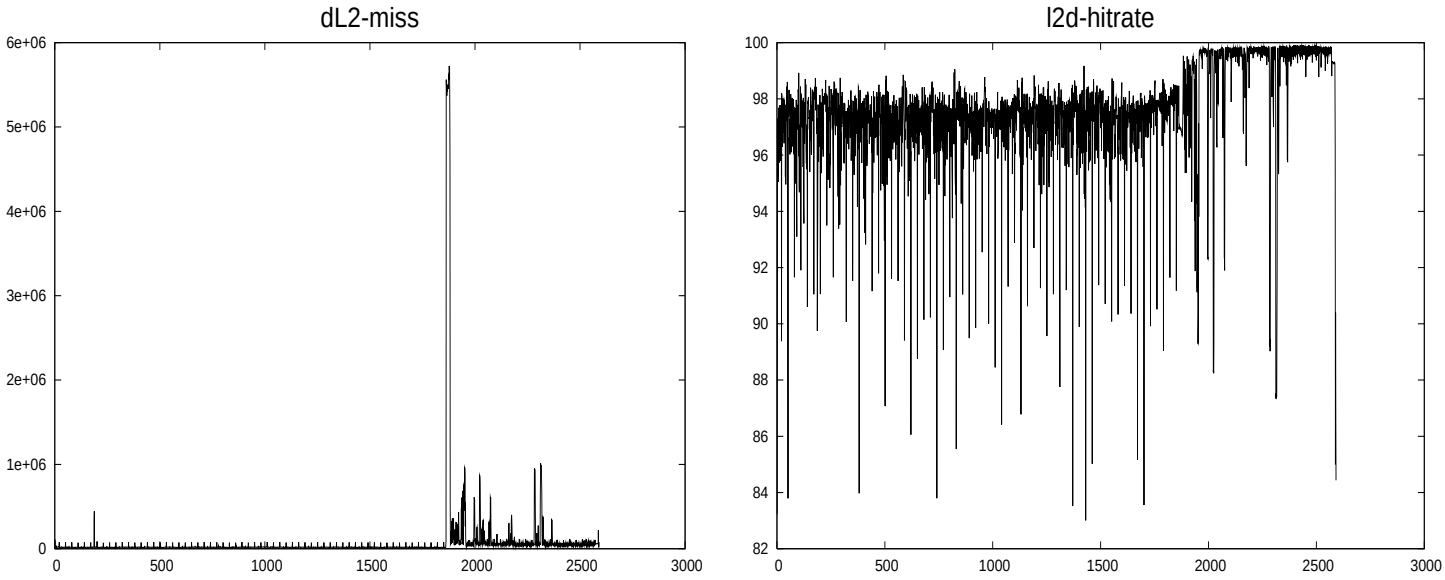

Figure 164: L2 data misses (left) and L2 data hitrate (right) livegraphs

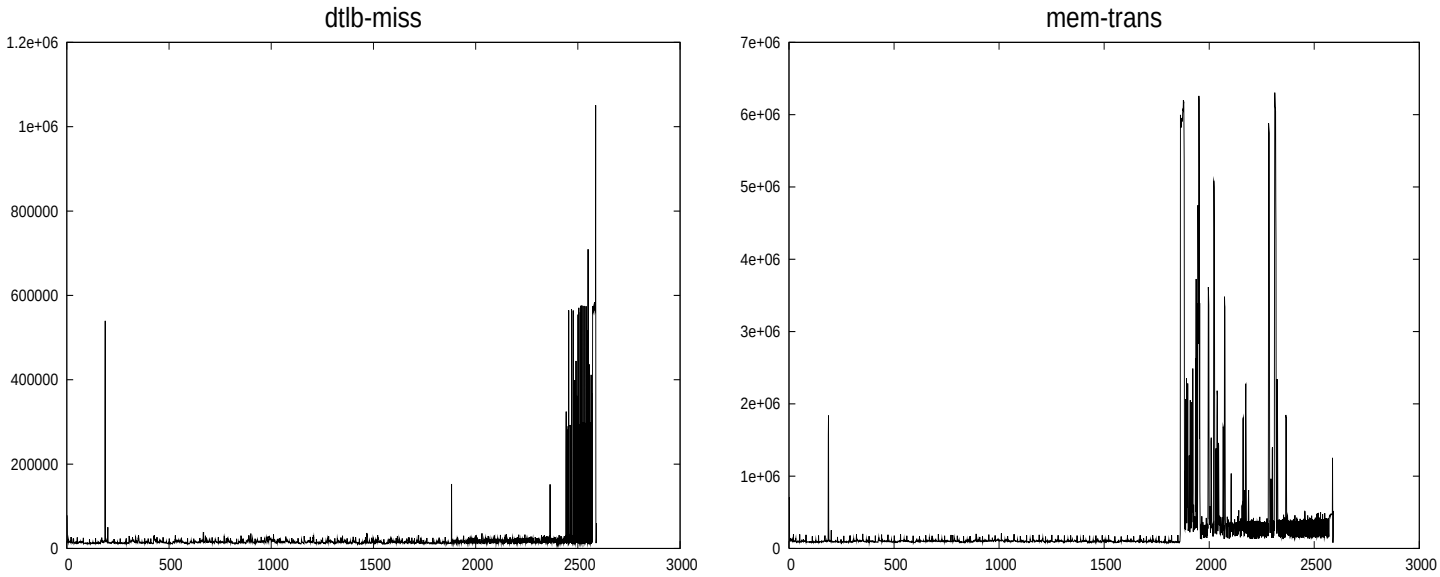

Figure 165: Data TLB misses (left) and Memory Transactions (right) livegraphs

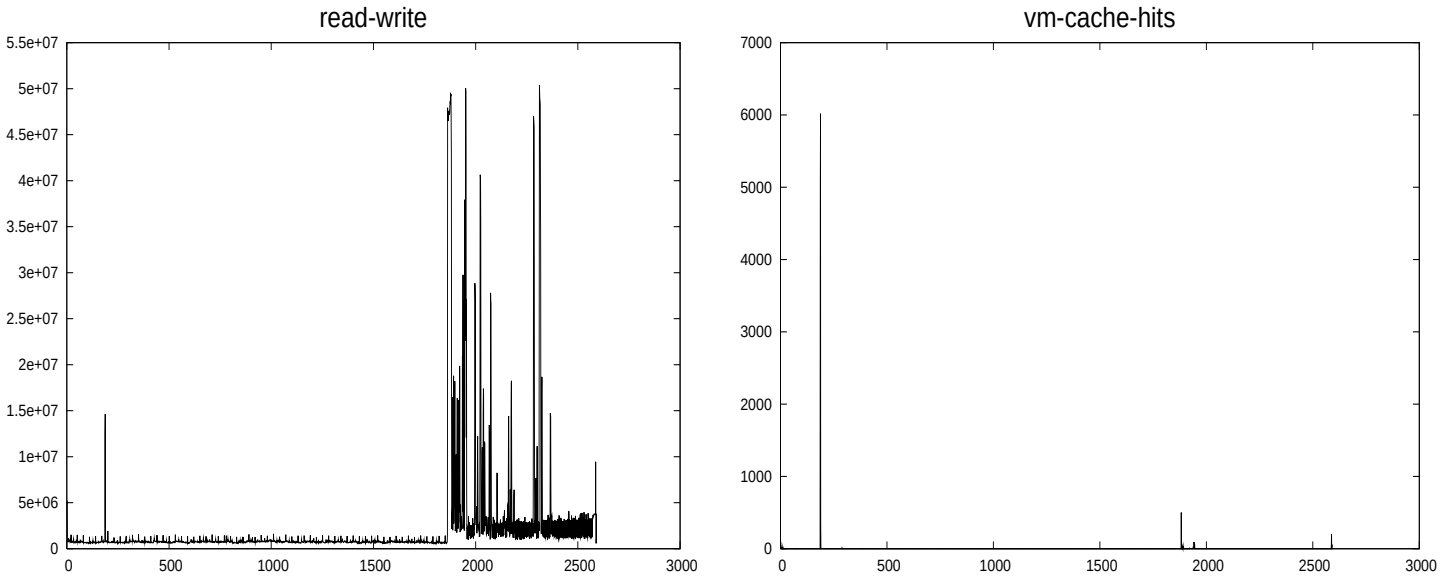

Figure 166: Read/Writes (left) and VM Page Cache Hits (right) livegraphs

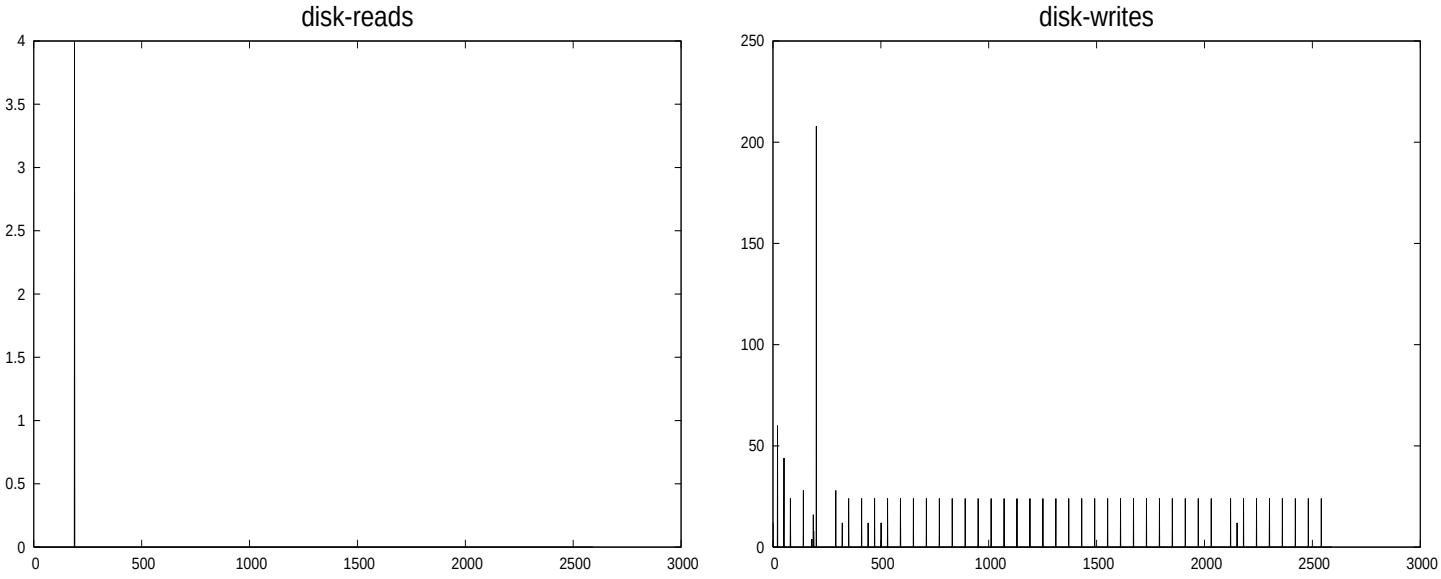

Figure 167: Disk Reads (left) and Disk Writes (right) livegraphs

C.3 GLIMMER

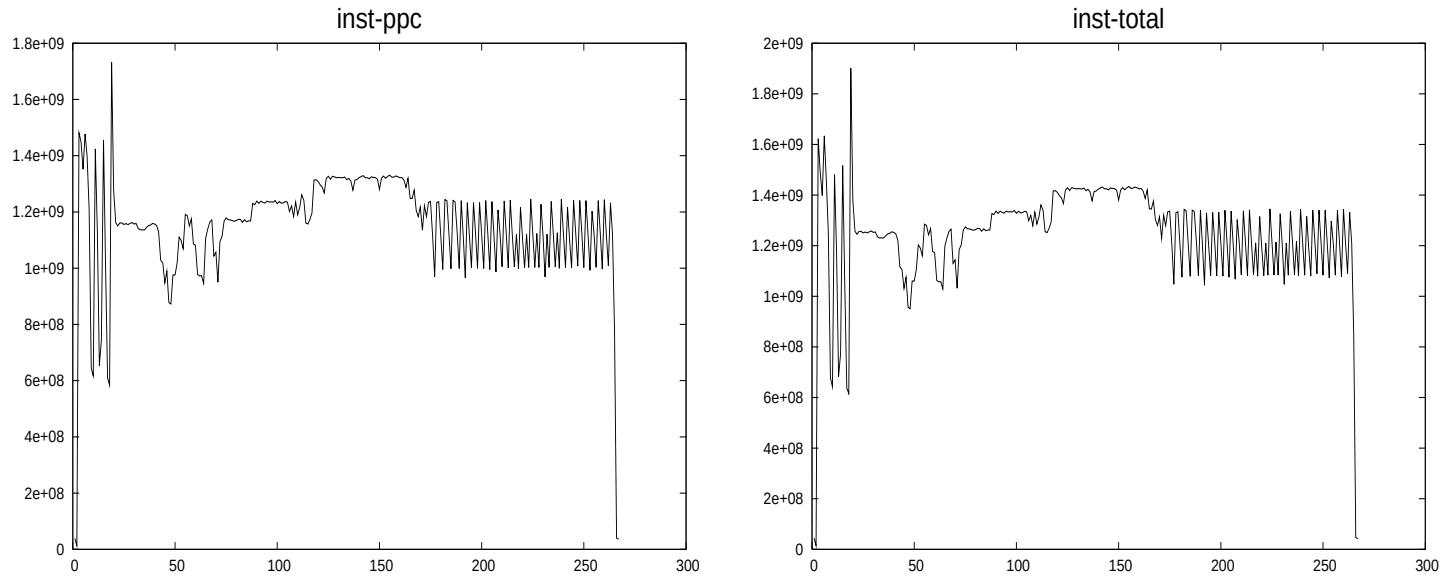

Figure 168: Instructions(ppc) (left) and Instructions (total) (right) livegraphs

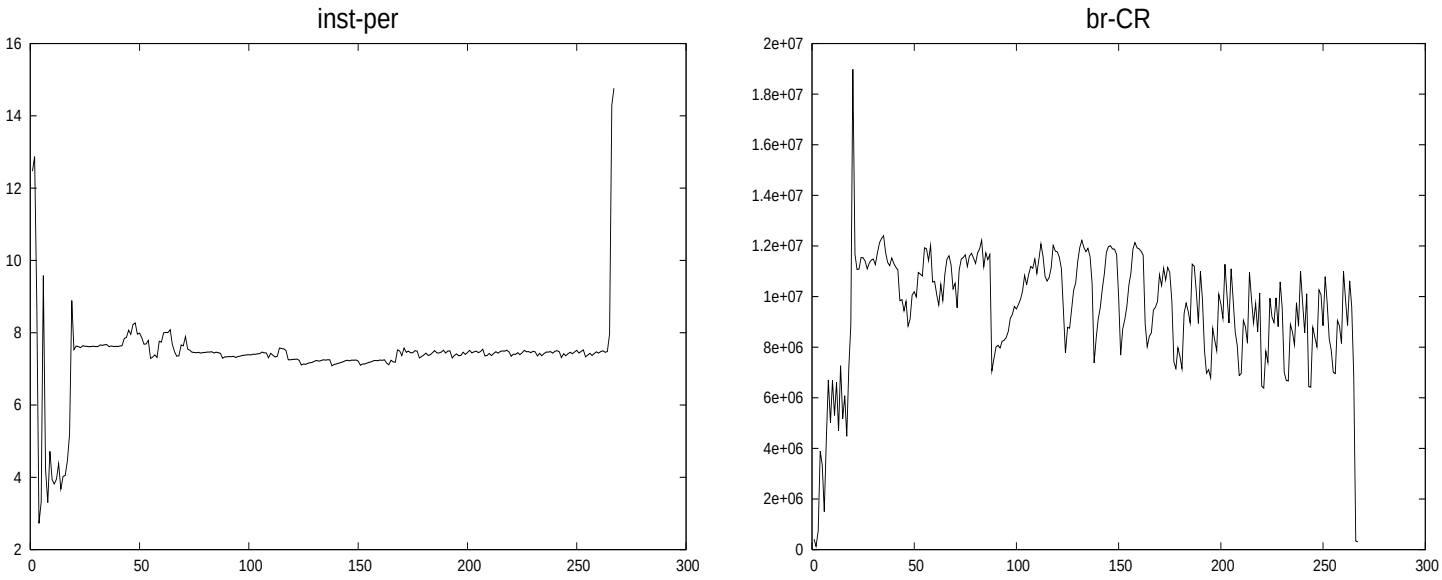

Figure 169: Percentage of io/ld/st instruction (left) and Branch Mispredicts (Condition Register) (right) livegraphs

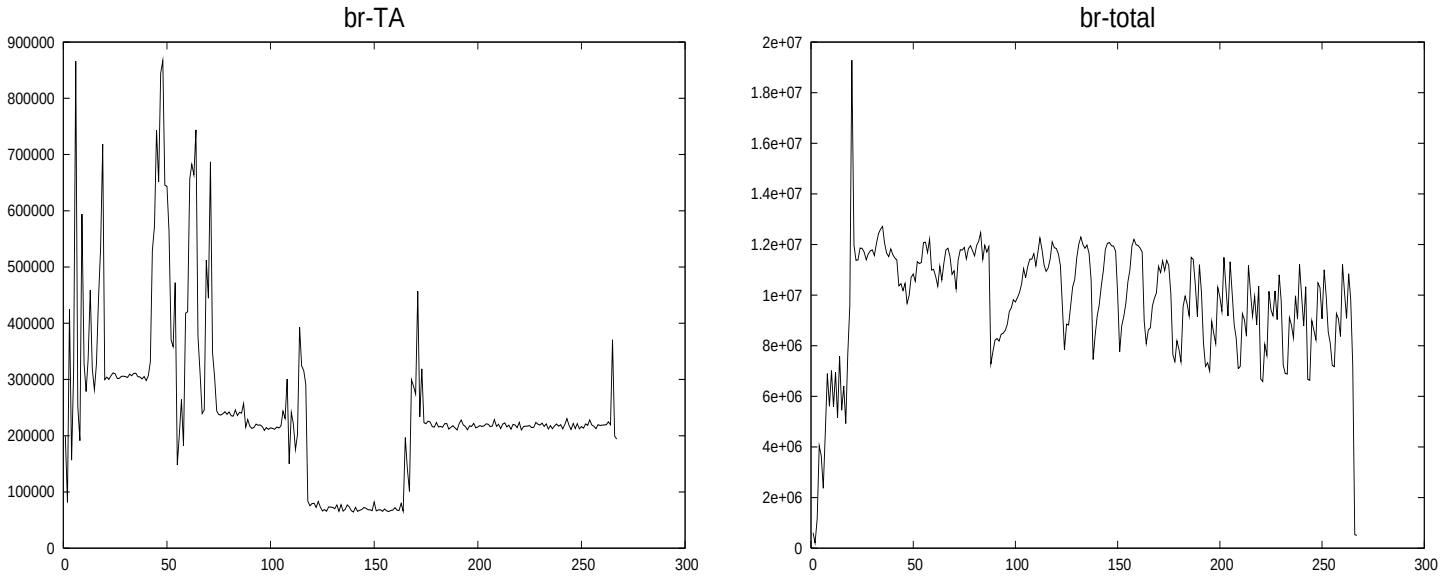

Figure 170: Branch Mispredicts (Target Address) (left) and Branch Mispredicts (total) (right) livegraphs

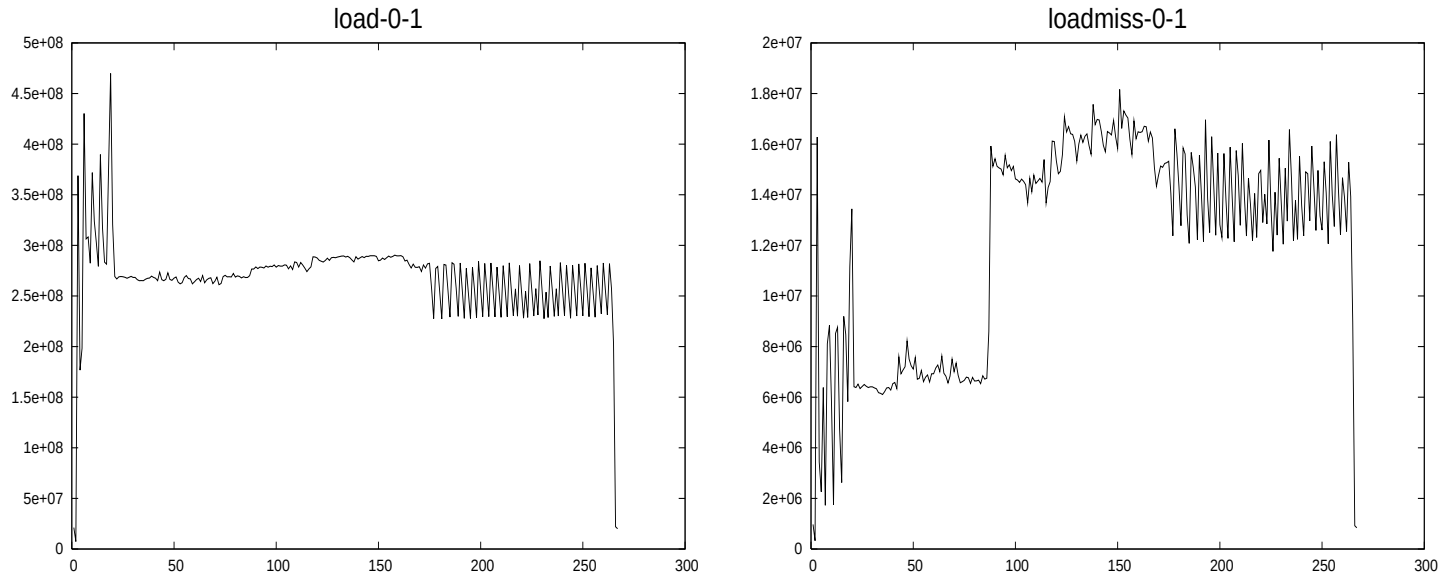

Figure 171: Loads (left) and Load Misses (right) livegraphs

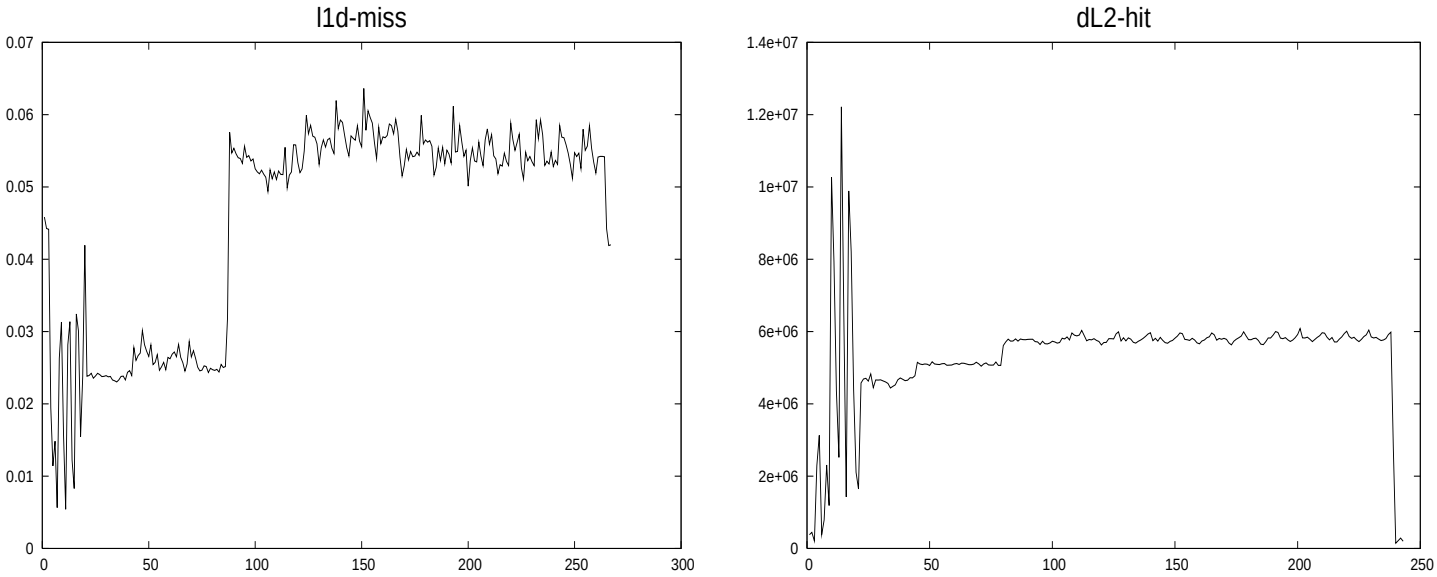

Figure 172: L1 data misses (left) and L2 data hits (right) livegraphs

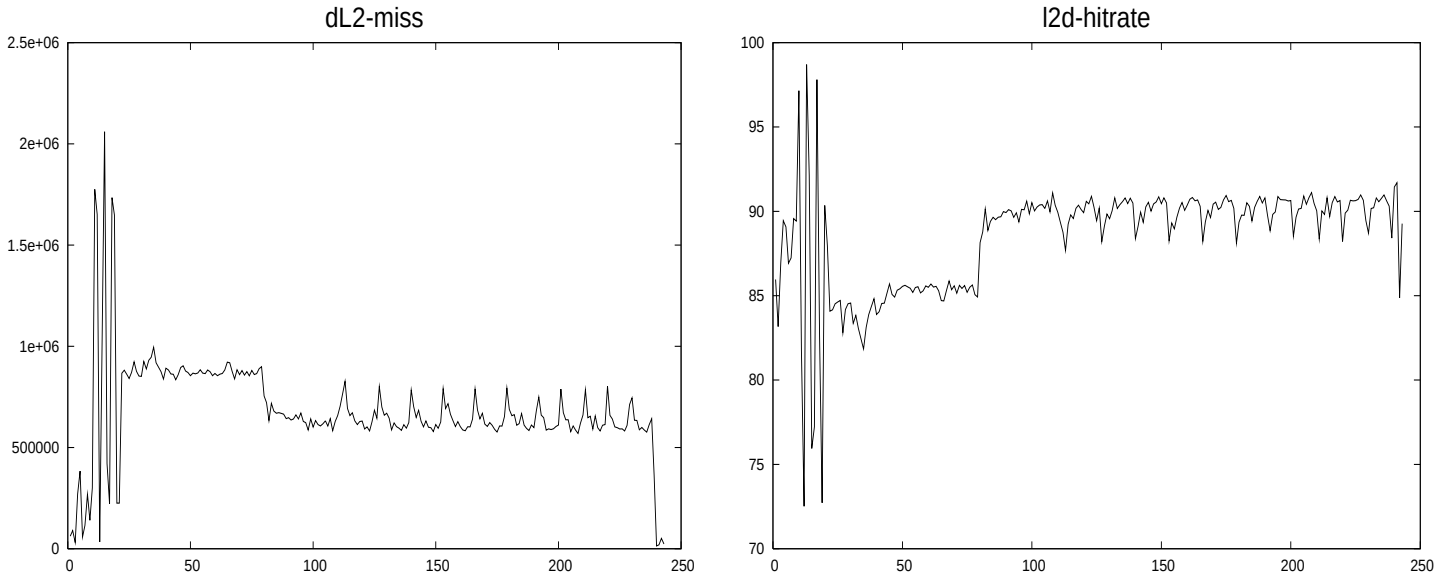

Figure 173: L2 data misses (left) and L2 data hitrate (right) livegraphs

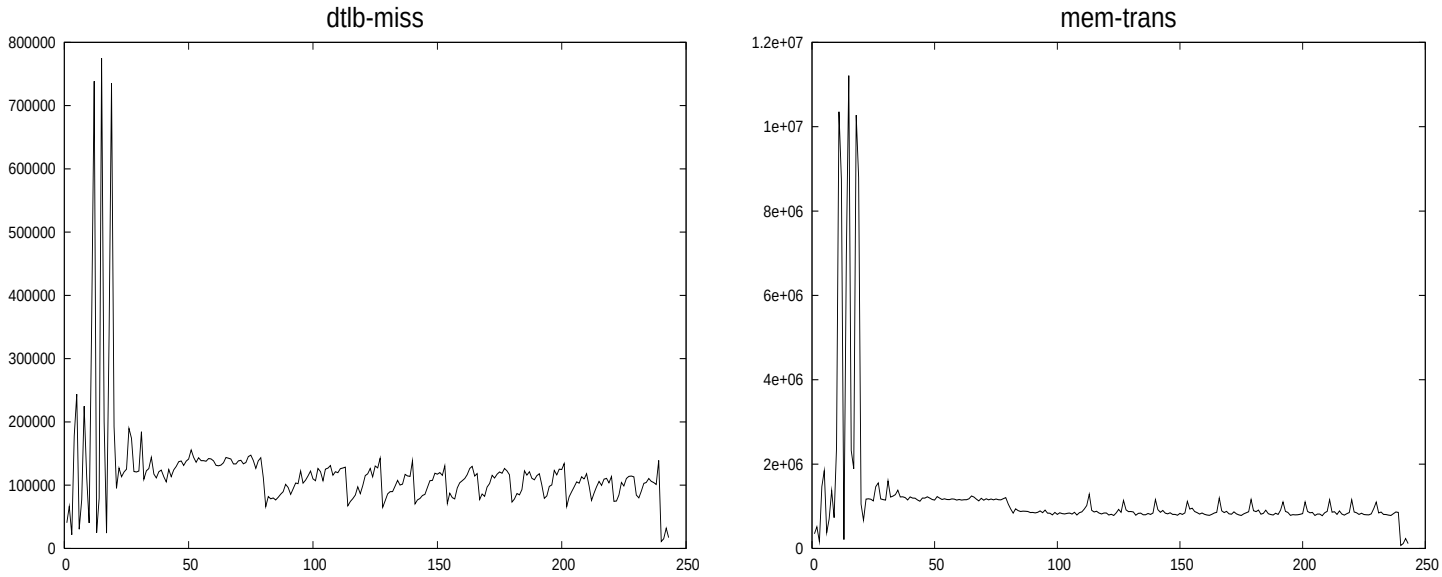

Figure 174: Data TLB misses (left) and Memory Transactions (right) livegraphs

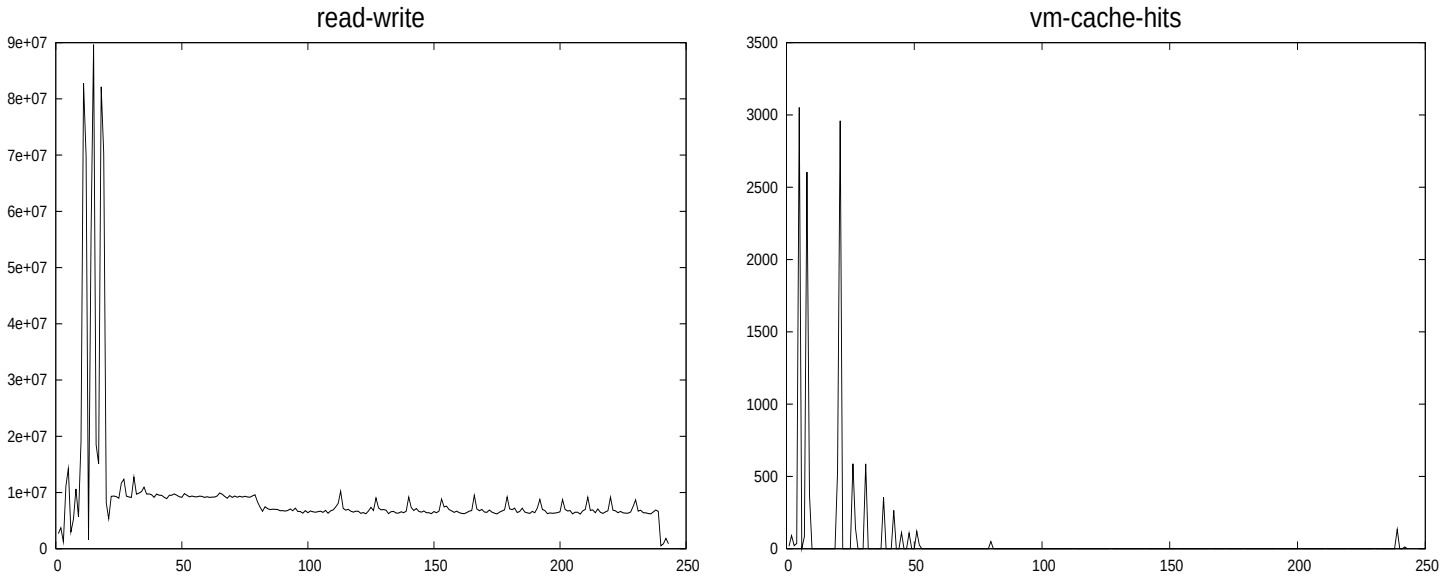

Figure 175: Read/Writes (left) and VM Page Cache Hits (right) livegraphs

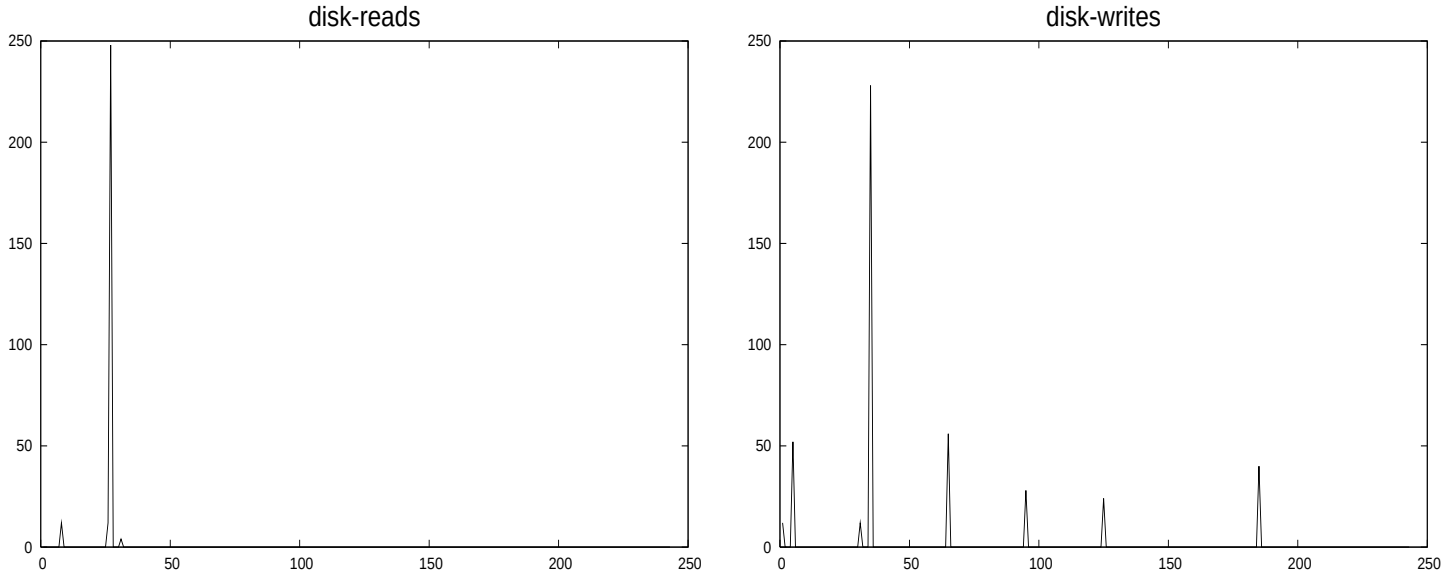

Figure 176: Disk Reads (left) and Disk Writes (right) livegraphs

C.4 GRAPPA

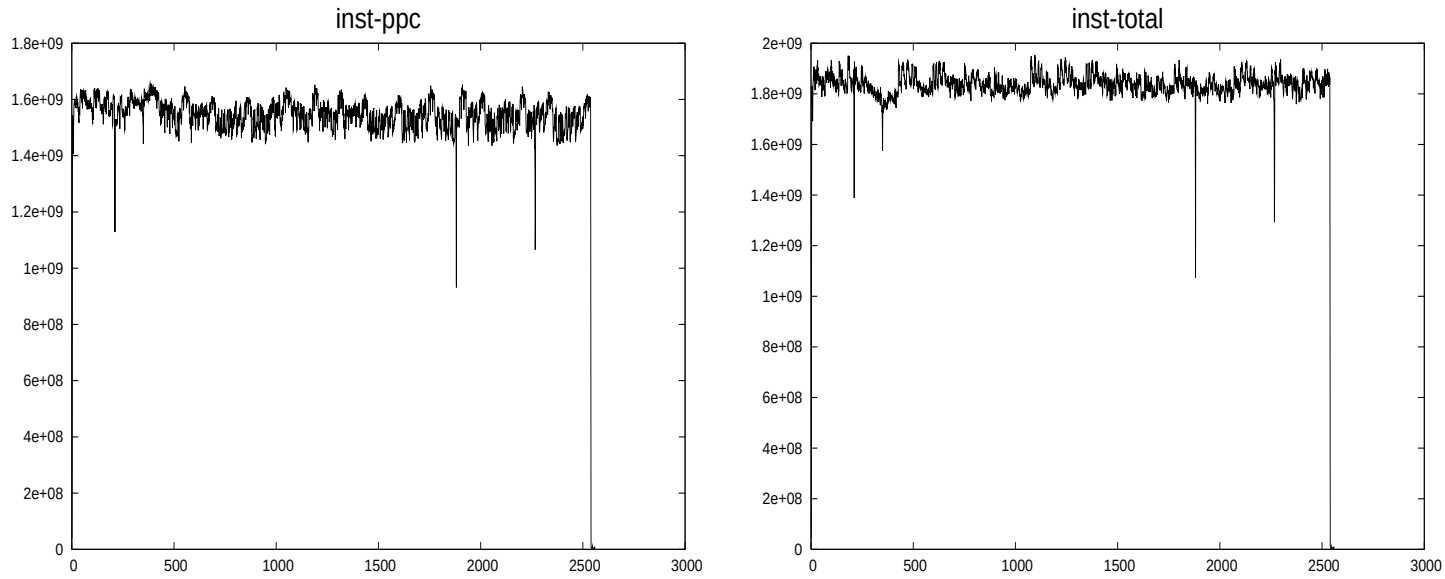

Figure 177: Instructions(ppc) (left) and Instructions (total) (right) livegraphs

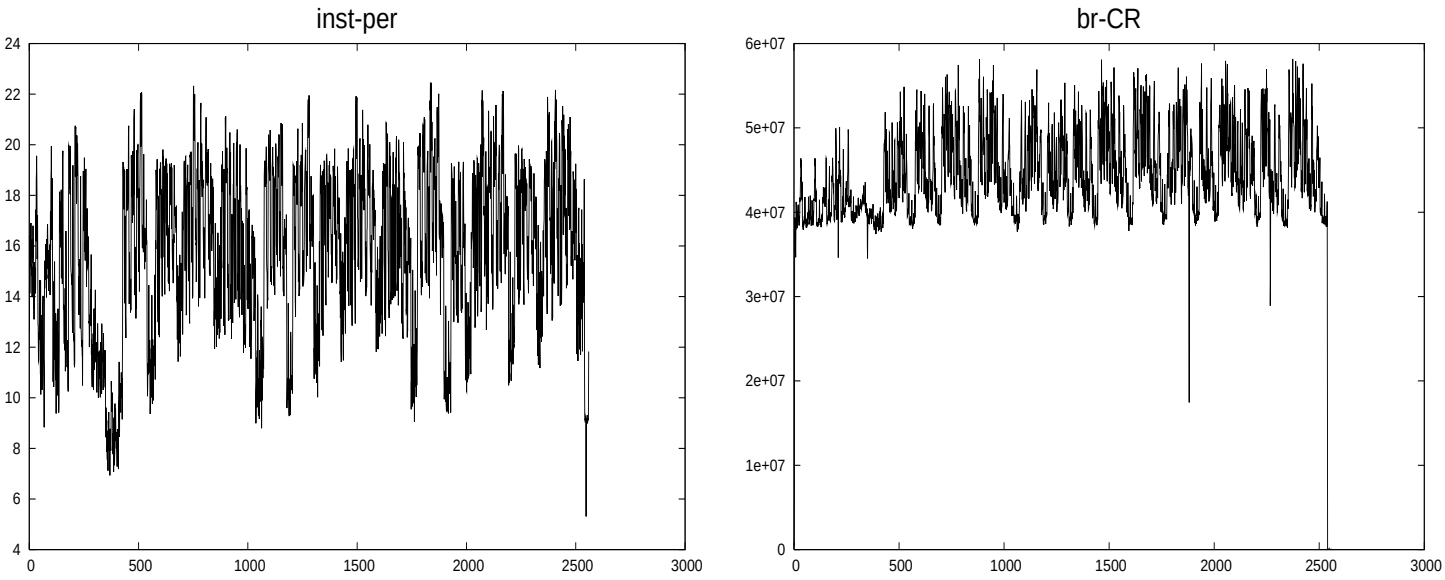

Figure 178: Percentage of io/ld/st instruction (left) and Branch Mispredicts (Condition Register) (right) livegraphs

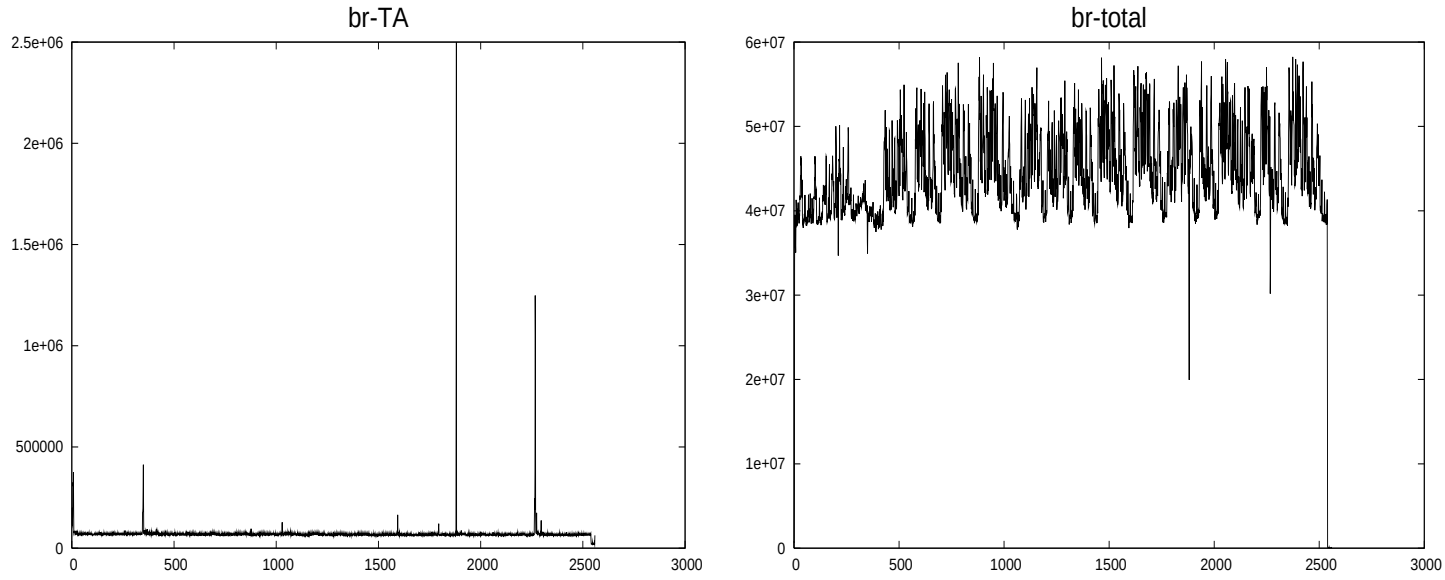

Figure 179: Branch Mispredicts (Target Address) (left) and Branch Mispredicts (total) (right) livegraphs

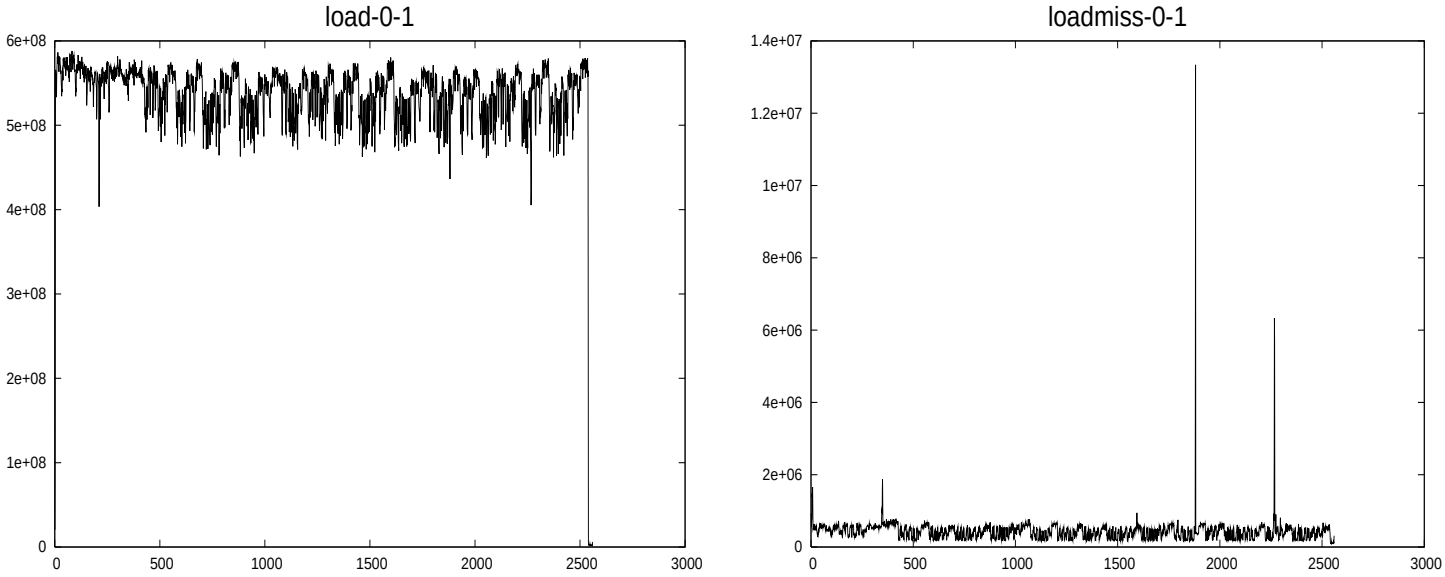

Figure 180: Loads (left) and Load Misses (right) livegraphs

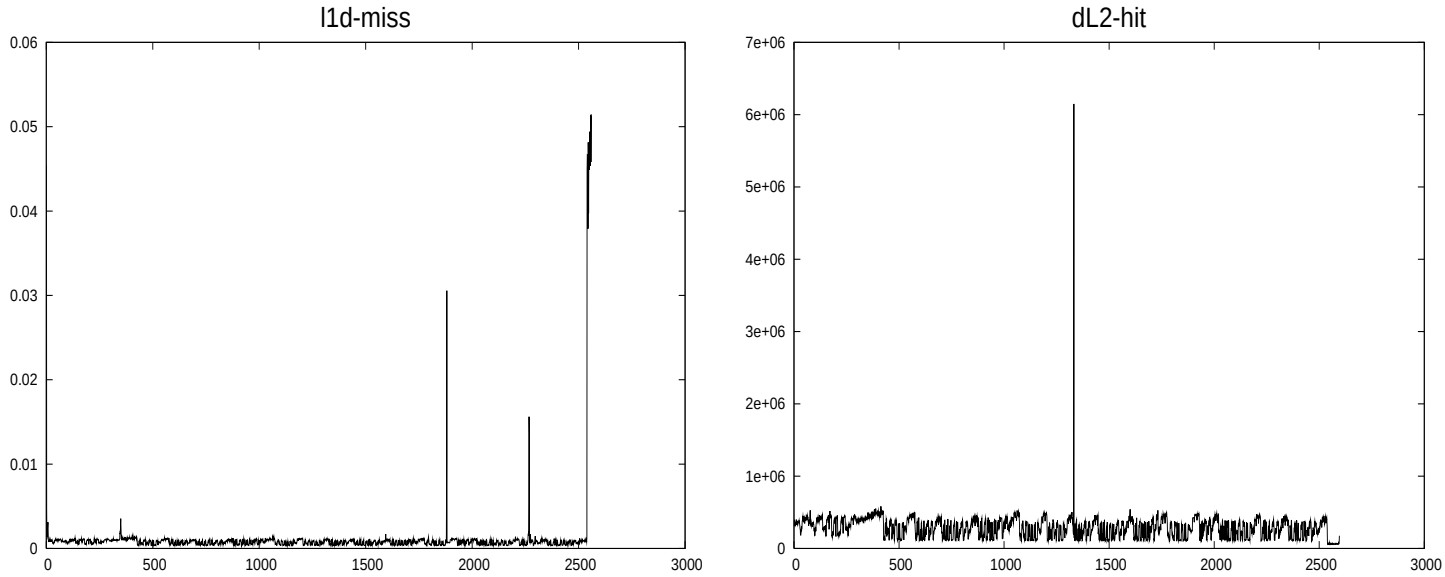

Figure 181: L1 data misses (left) and L2 data hits (right) livegraphs

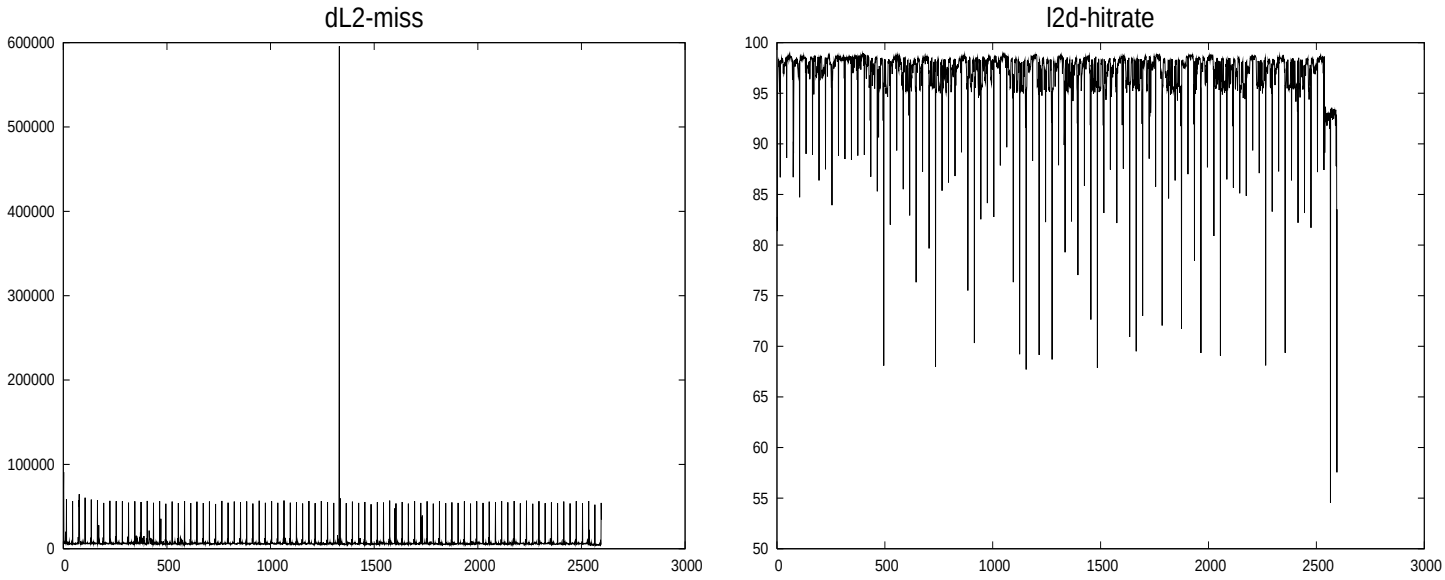

Figure 182: L2 data misses (left) and L2 data hitrate (right) livegraphs

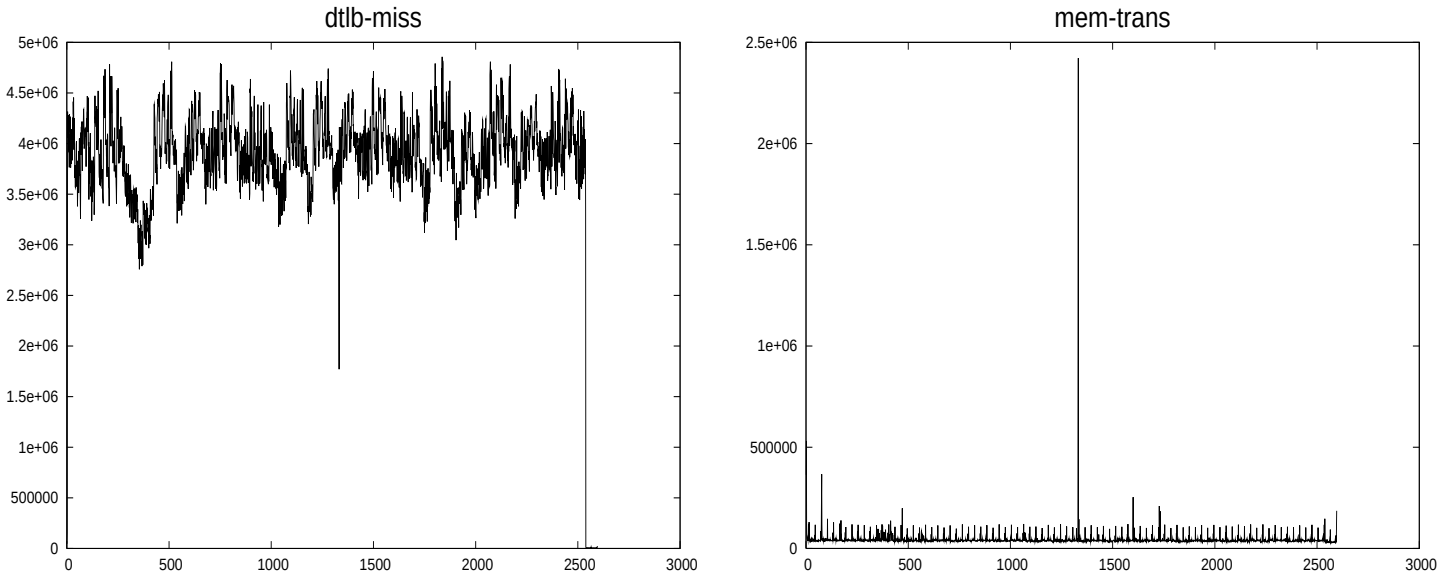

Figure 183: Data TLB misses (left) and Memory Transactions (right) livegraphs

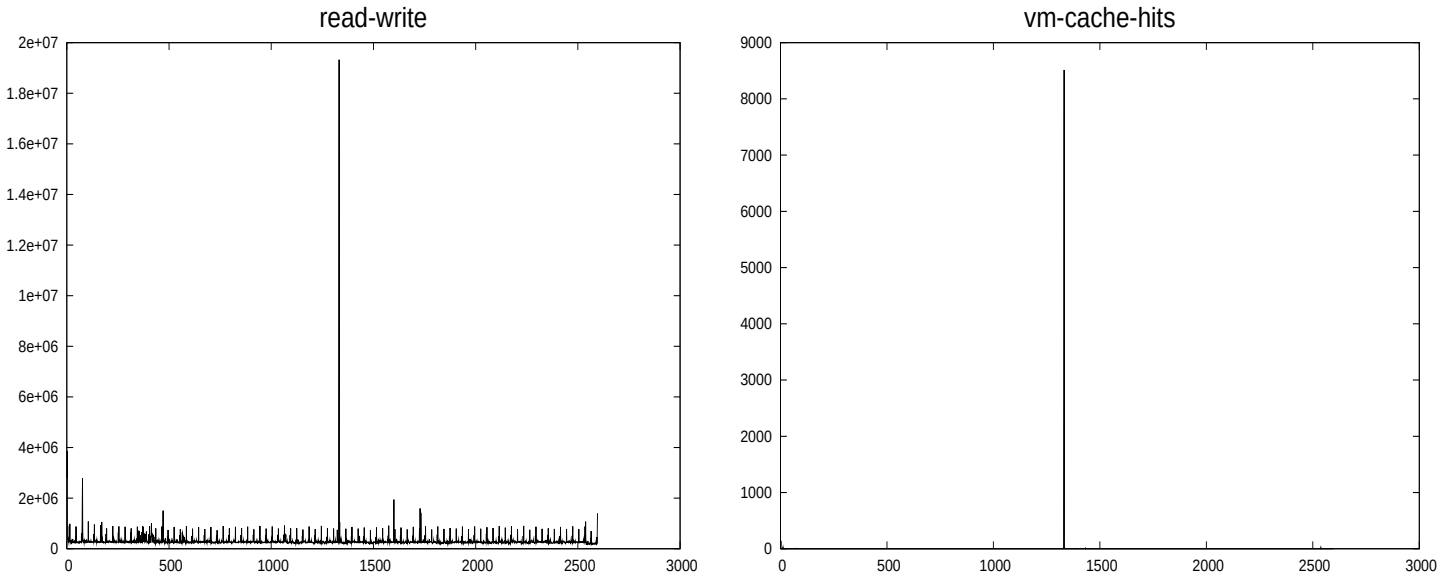

Figure 184: Read/Writes (left) and VM Page Cache Hits (right) livegraphs

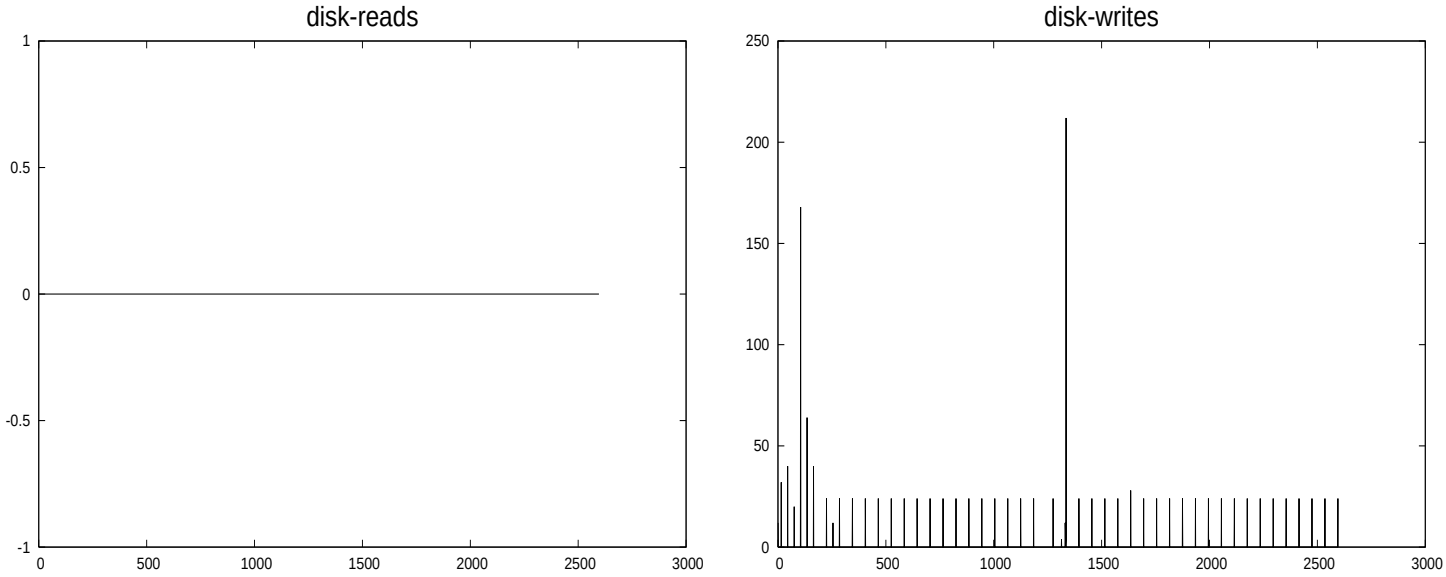

Figure 185: Disk Reads (left) and Disk Writes (right) livegraphs

C.5 HMMER

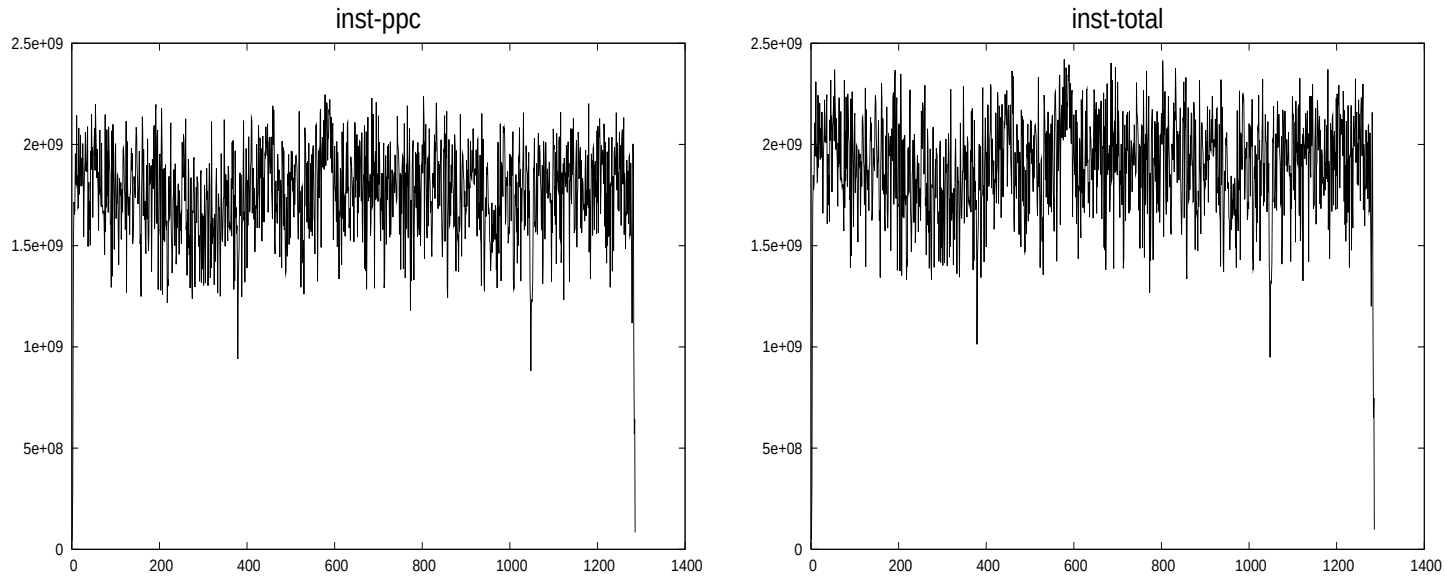

Figure 186: Instructions(ppc) (left) and Instructions (total) (right) livegraphs

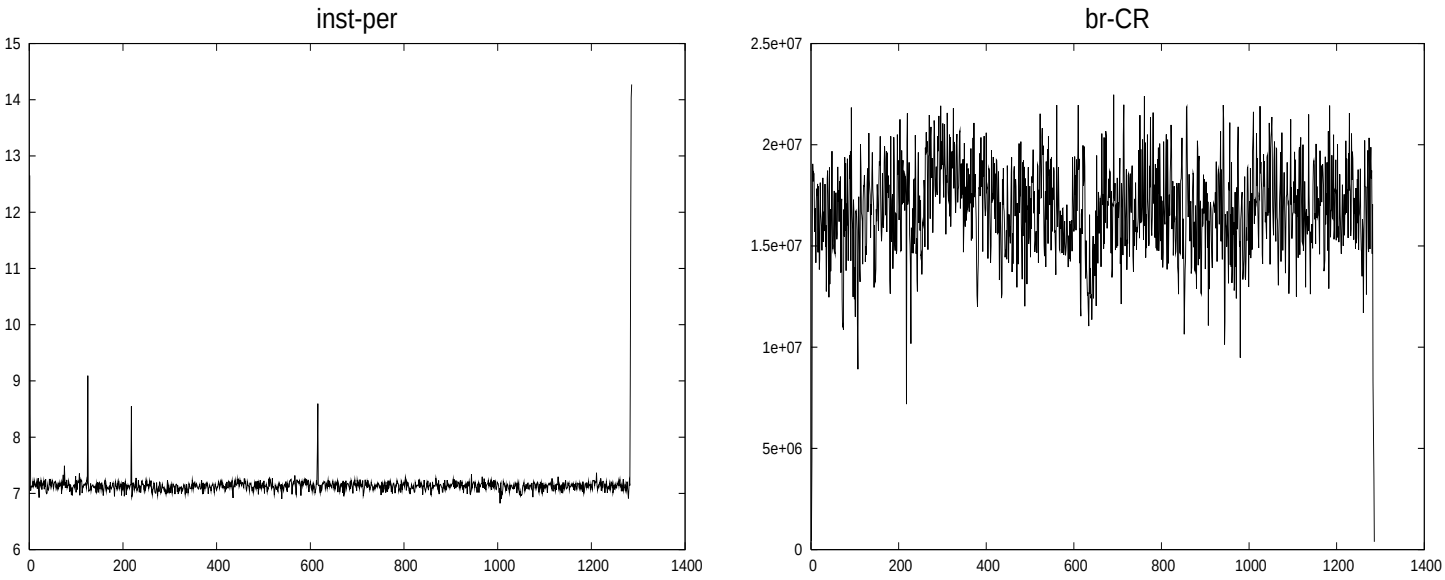

Figure 187: Percentage of io/ld/st instruction (left) and Branch Mispredicts (Condition Register) (right) livegraphs

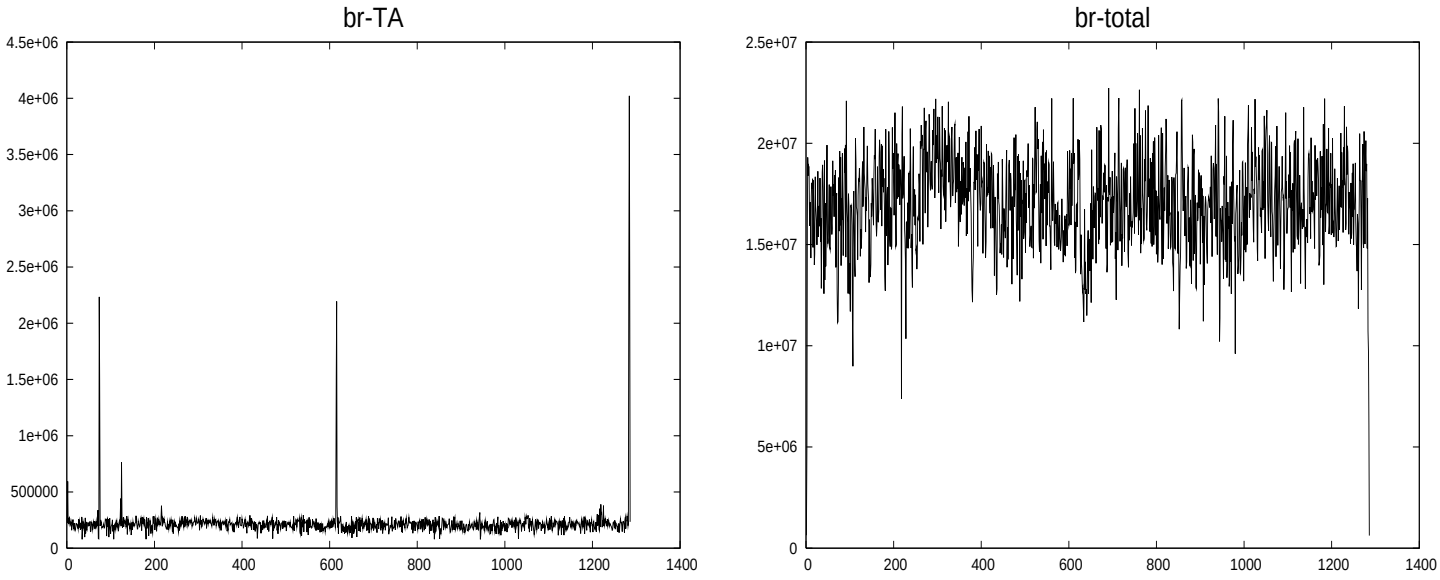

Figure 188: Branch Mispredicts (Target Address) (left) and Branch Mispredicts (total) (right) livegraphs

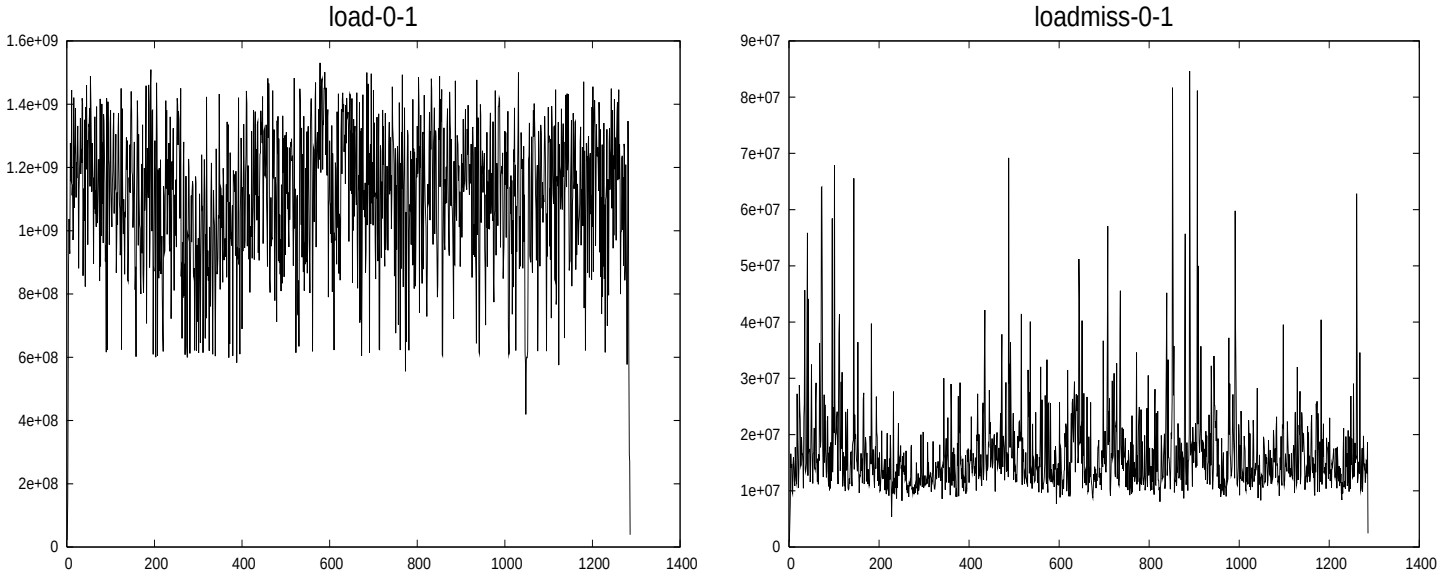

Figure 189: Loads (left) and Load Misses (right) livegraphs

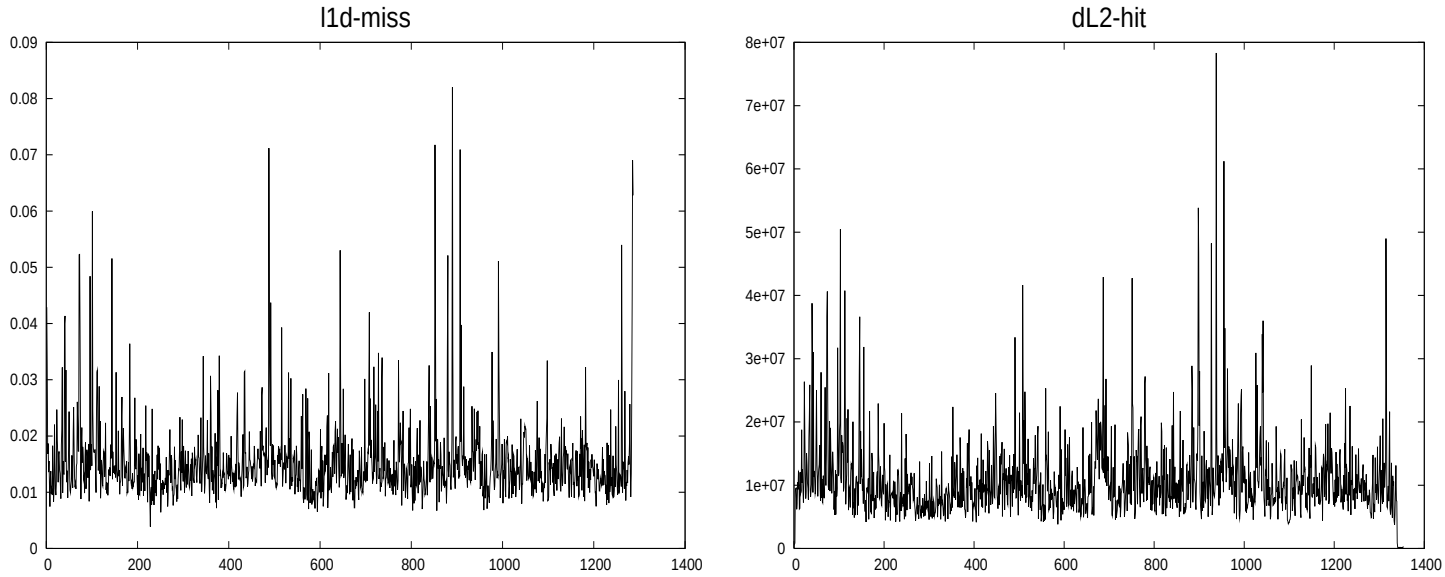

Figure 190: L1 data misses (left) and L2 data hits (right) livegraphs

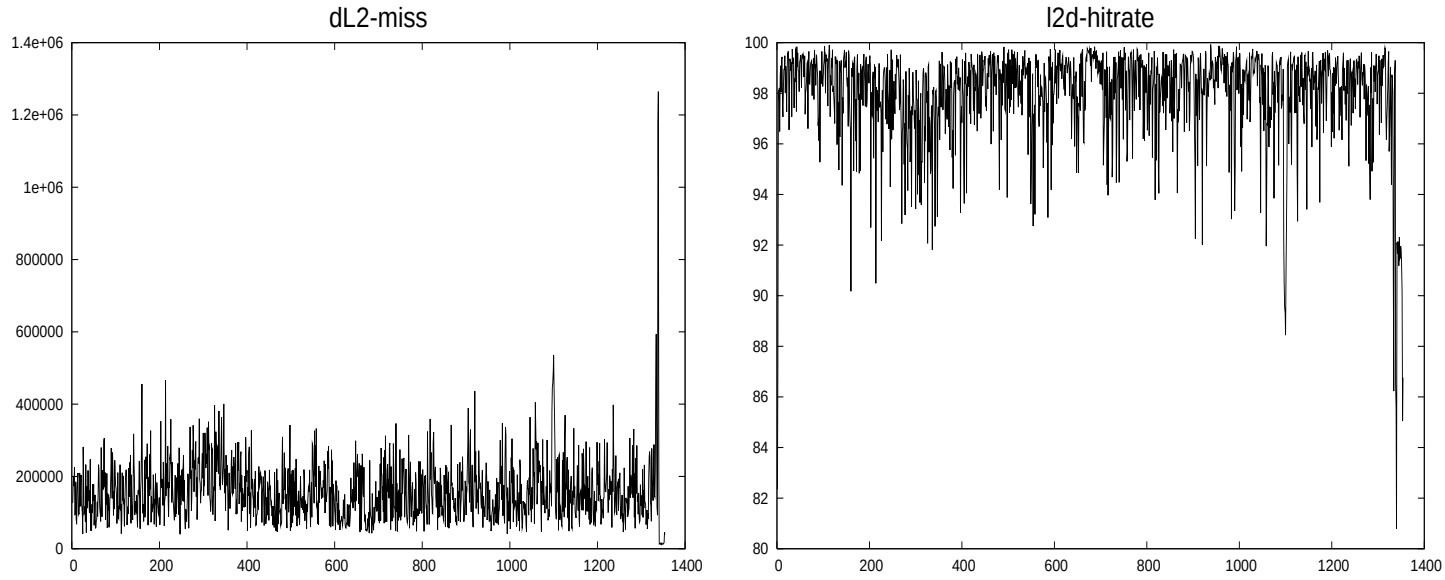

Figure 191: L2 data misses (left) and L2 data hitrate (right) livegraphs

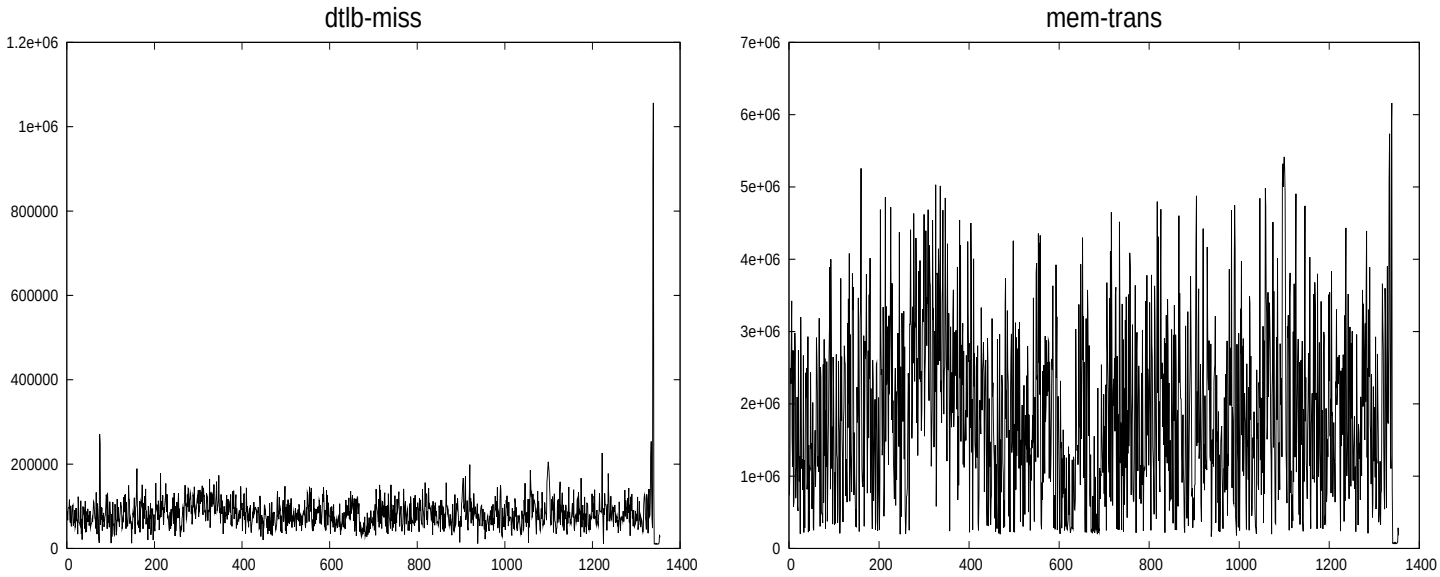

Figure 192: Data TLB misses (left) and Memory Transactions (right) livegraphs

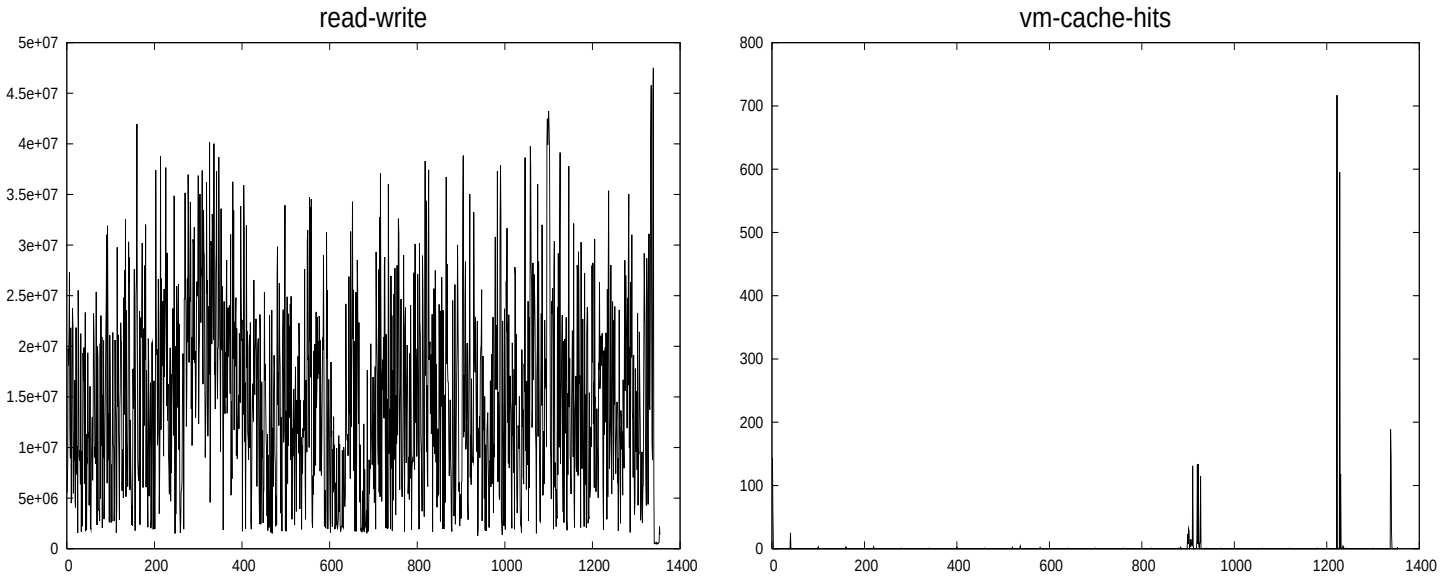

Figure 193: Read/Writes (left) and VM Page Cache Hits (right) livegraphs

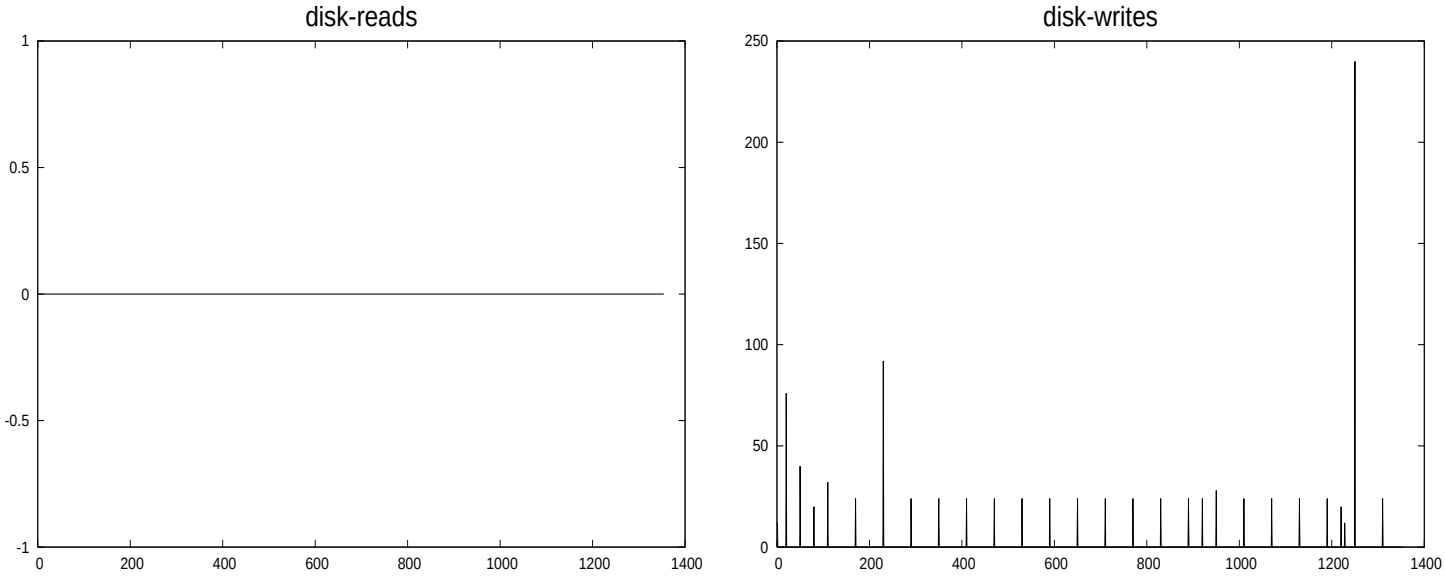

Figure 194: Disk Reads (left) and Disk Writes (right) livegraphs

C.6 PHYLIP

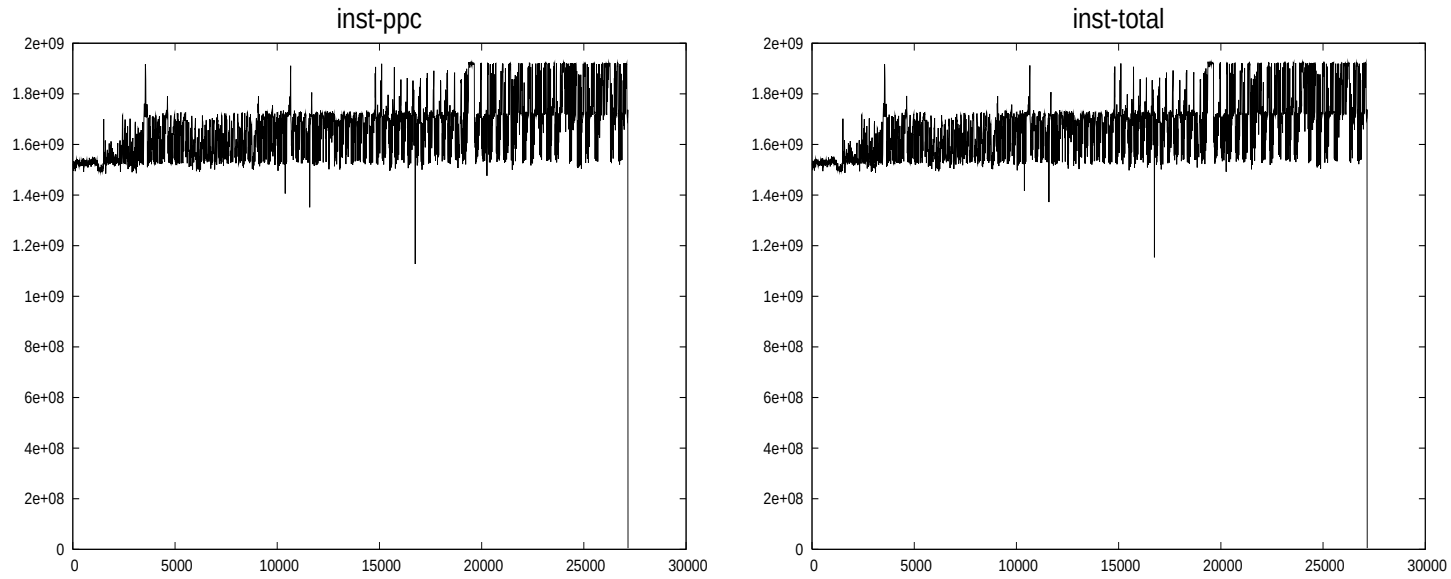

Figure 195: Instructions(ppc) (left) and Instructions (total) (right) livegraphs

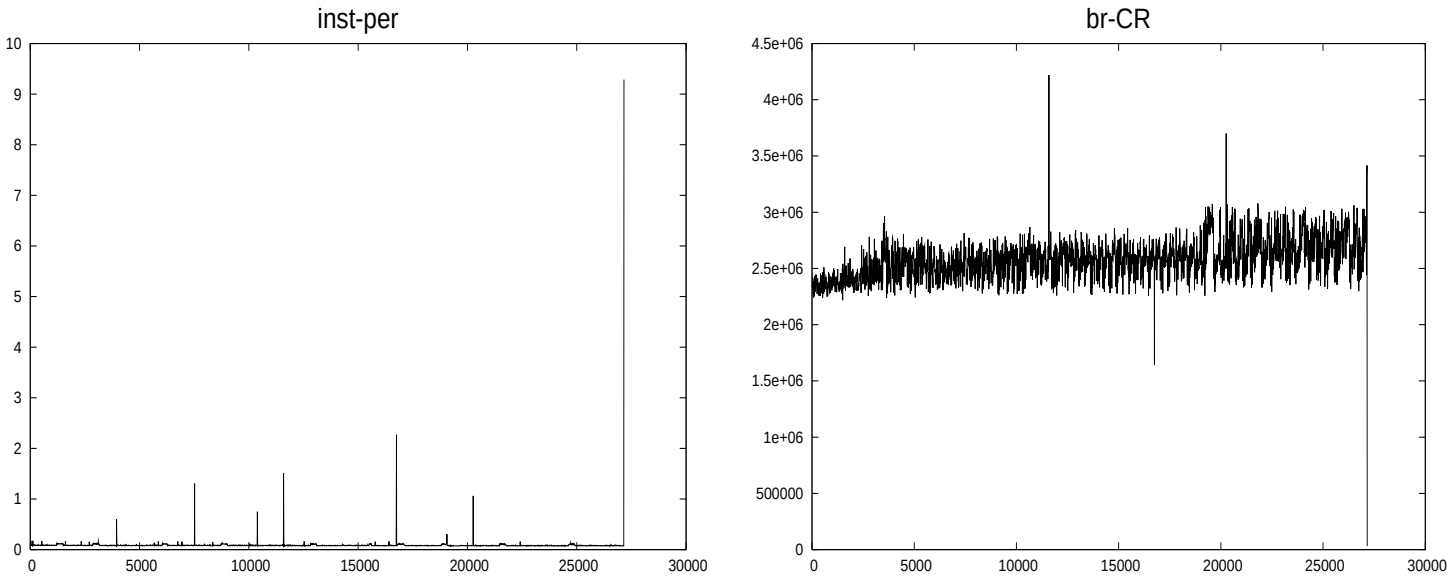

Figure 196: Percentage of io/ld/st instruction (left) and Branch Mispredicts (Condition Register) (right) livegraphs

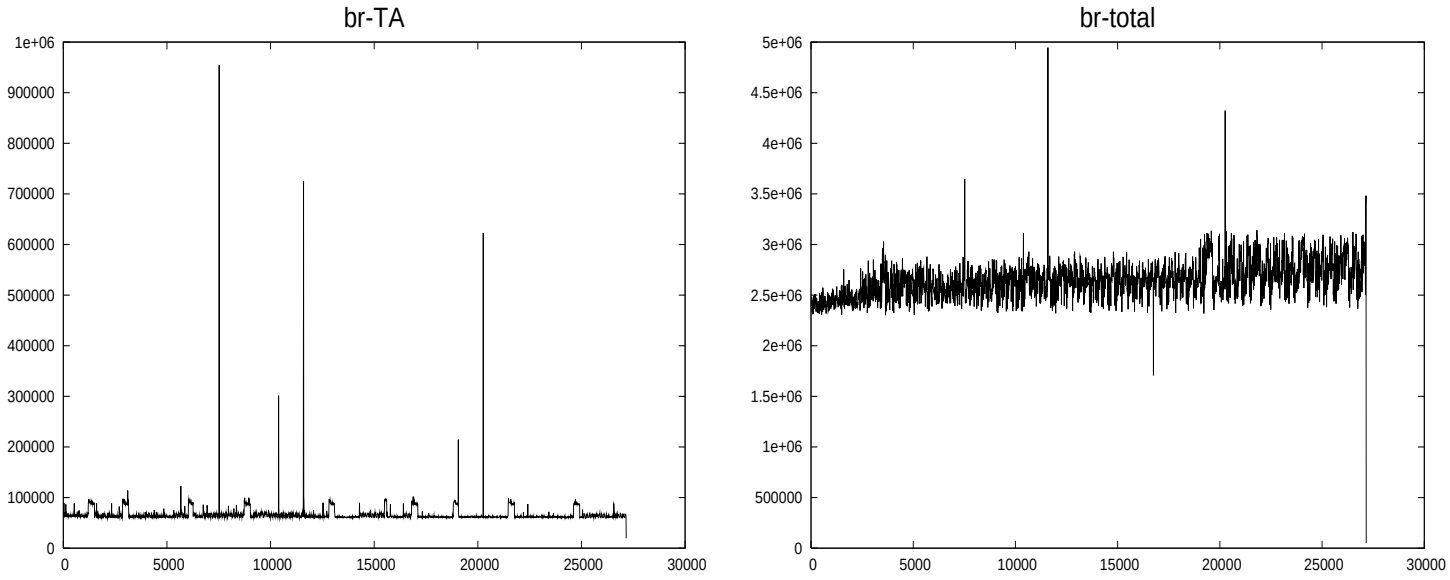

Figure 197: Branch Mispredicts (Target Address) (left) and Branch Mispredicts (total) (right) livegraphs

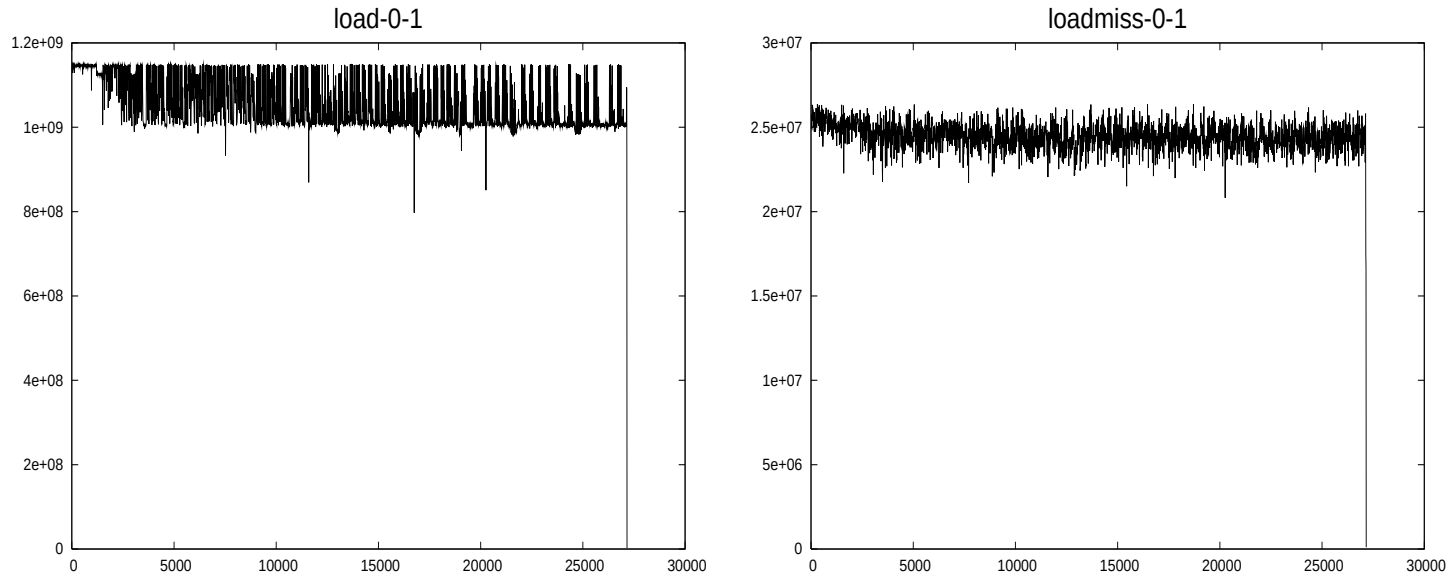

Figure 198: Loads (left) and Load Misses (right) livegraphs

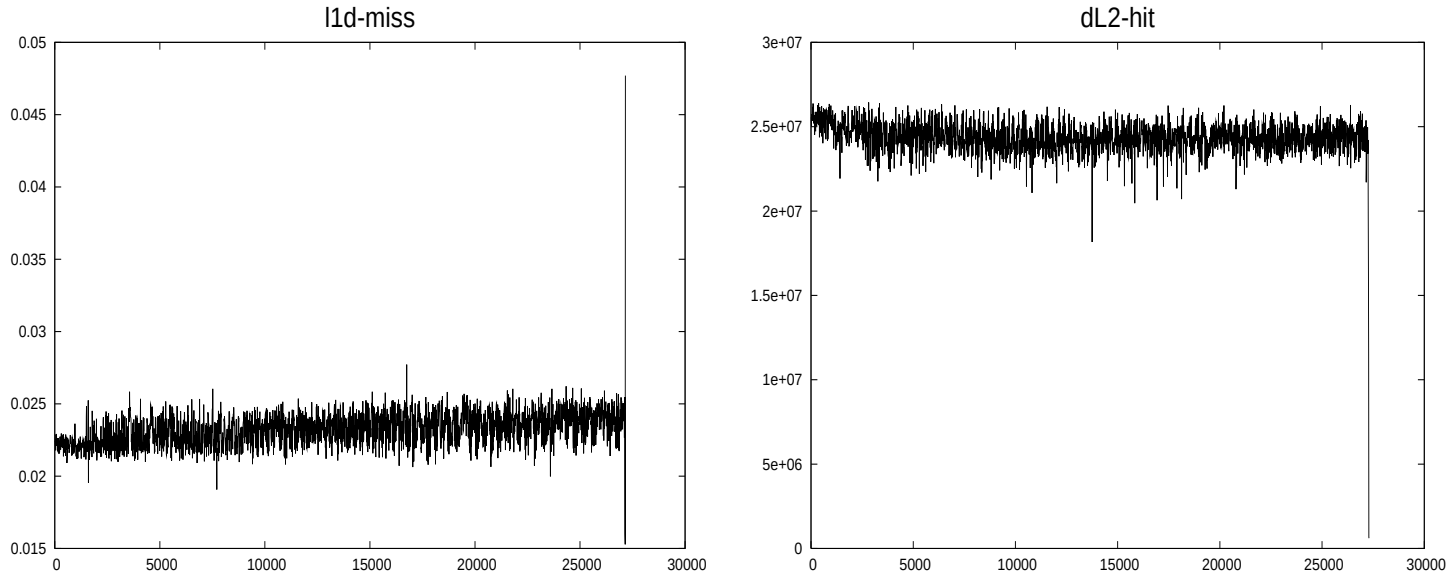

Figure 199: L1 data misses (left) and L2 data hits (right) livegraphs

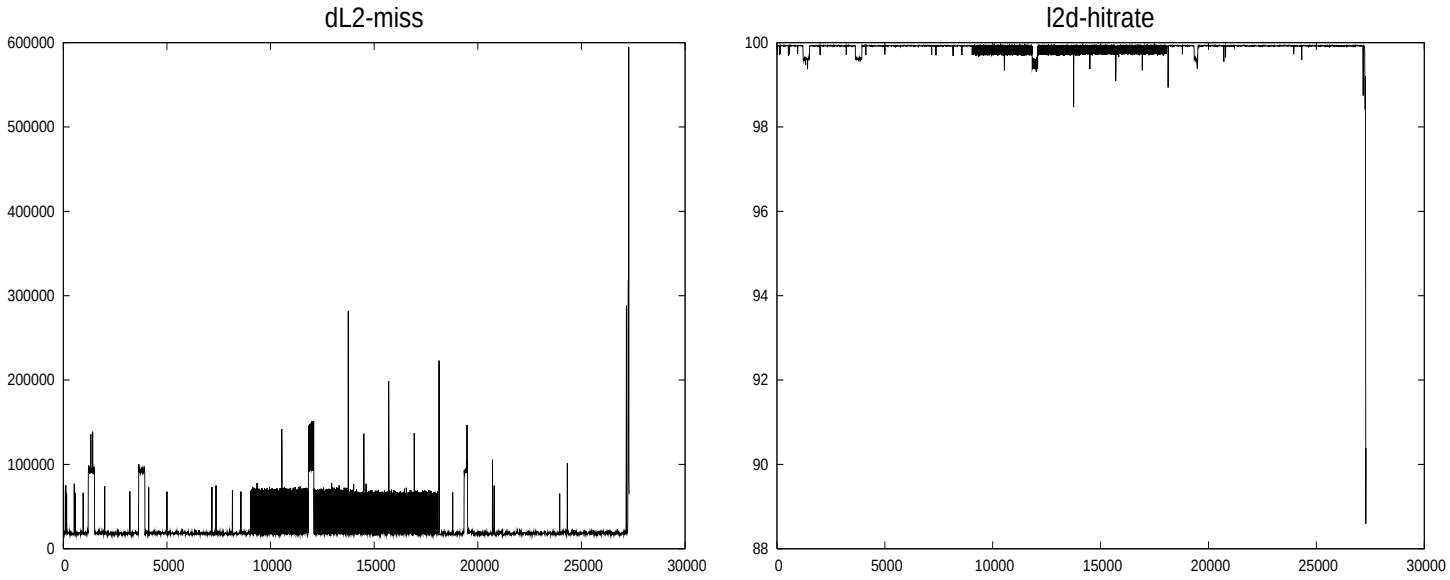

Figure 200: L2 data misses (left) and L2 data hitrate (right) livegraphs

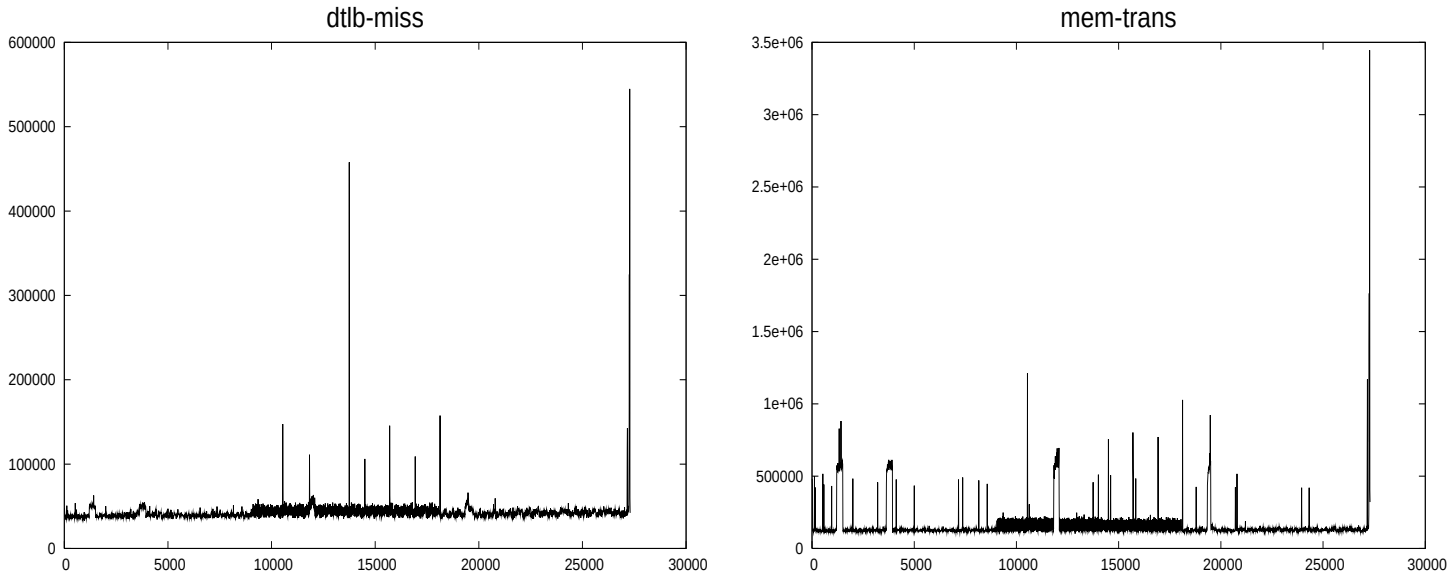

Figure 201: Data TLB misses (left) and Memory Transactions (right) livegraphs

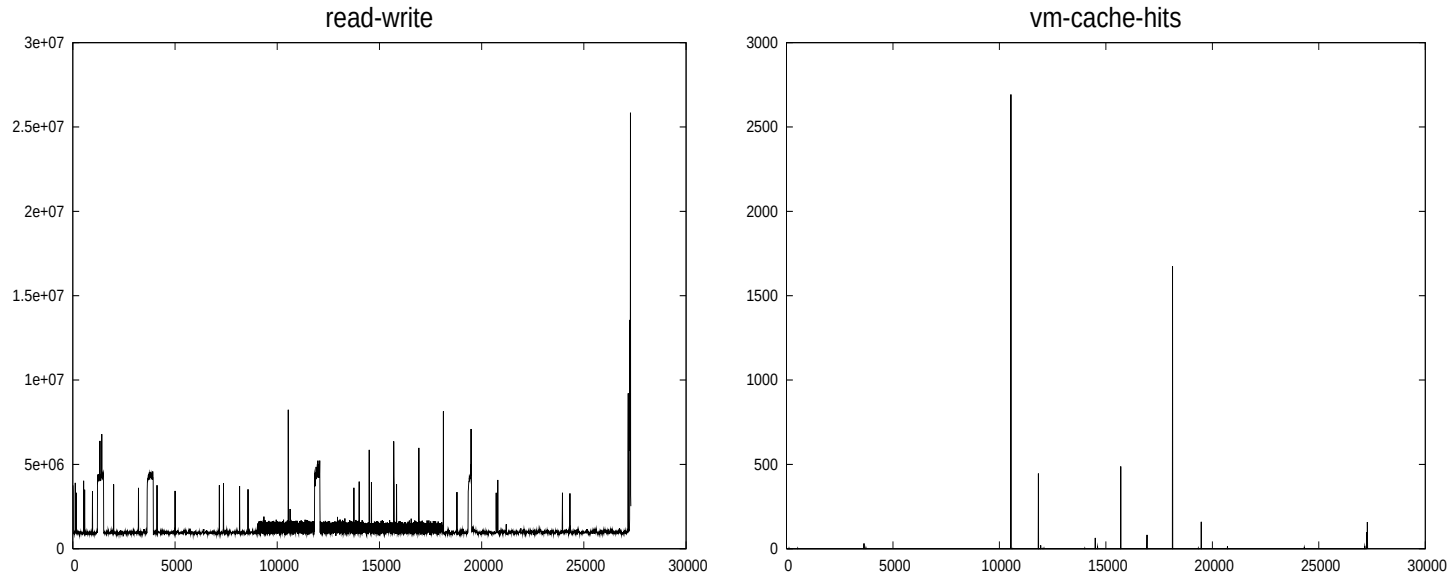

Figure 202: Read/Writes (left) and VM Page Cache Hits (right) livegraphs

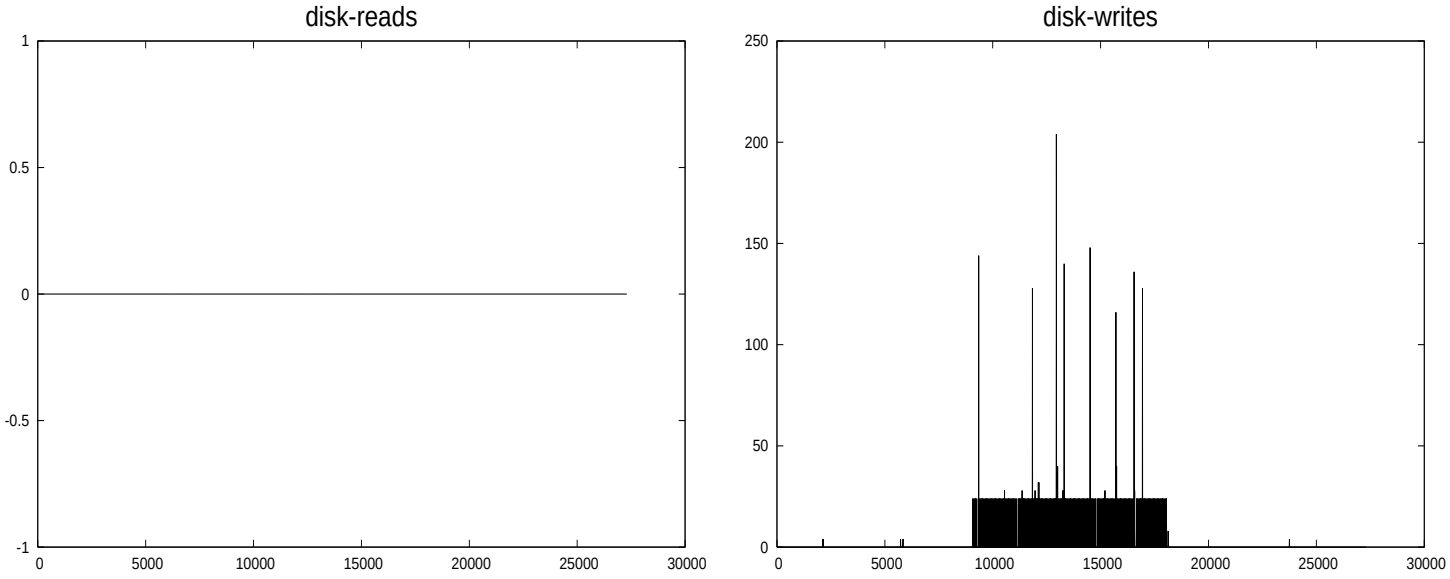

Figure 203: Disk Reads (left) and Disk Writes (right) livegraphs

C.7 PREDATOR

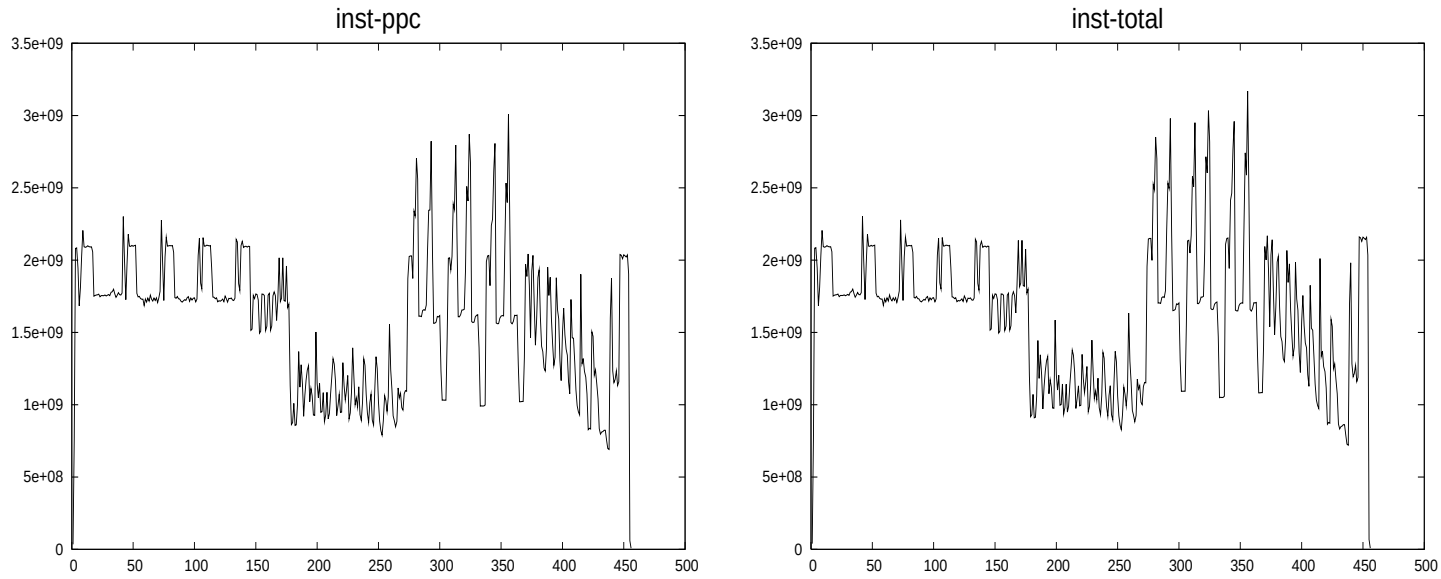

Figure 204: Instructions(ppc) (left) and Instructions (total) (right) livegraphs

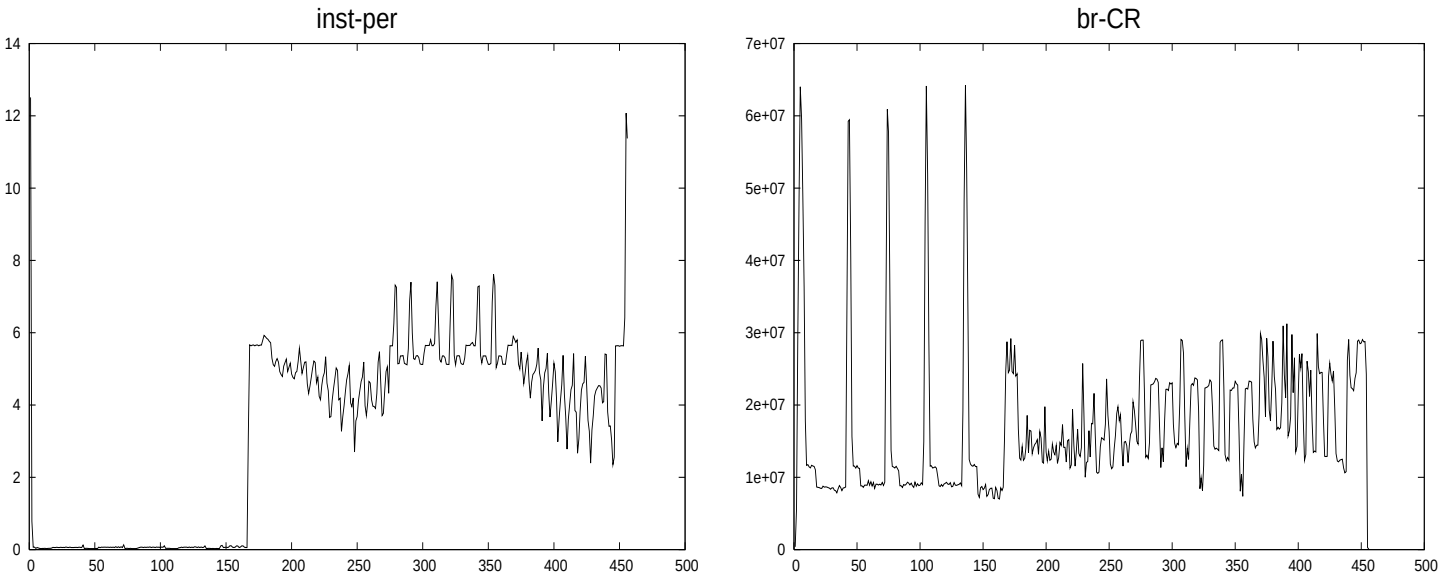

Figure 205: Percentage of io/ld/st instruction (left) and Branch Mispredicts (Condition Register) (right) livegraphs

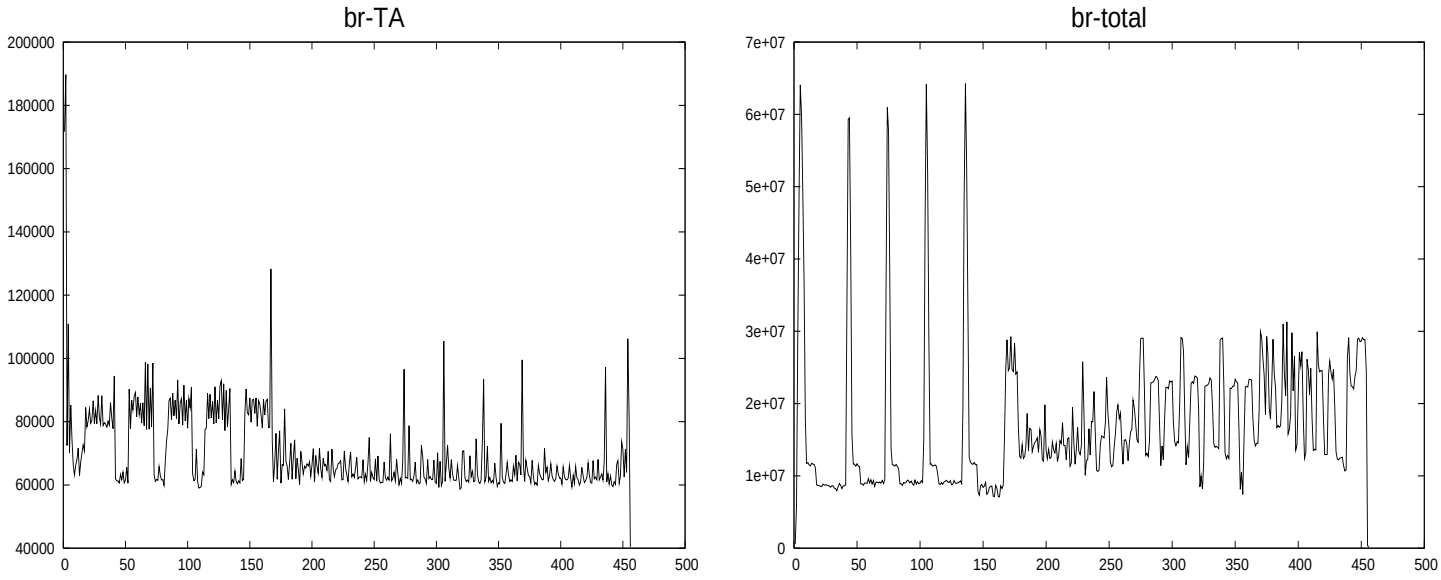

Figure 206: Branch Mispredicts (Target Address) (left) and Branch Mispredicts (total) (right) livegraphs

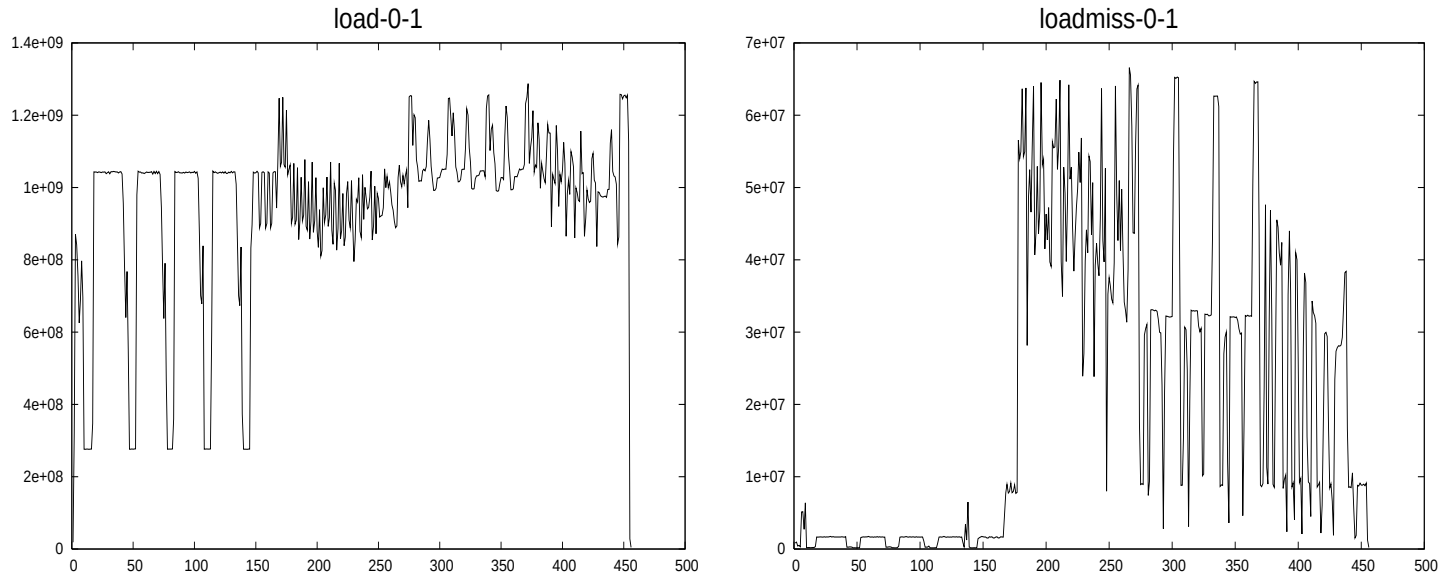

Figure 207: Loads (left) and Load Misses (right) livegraphs

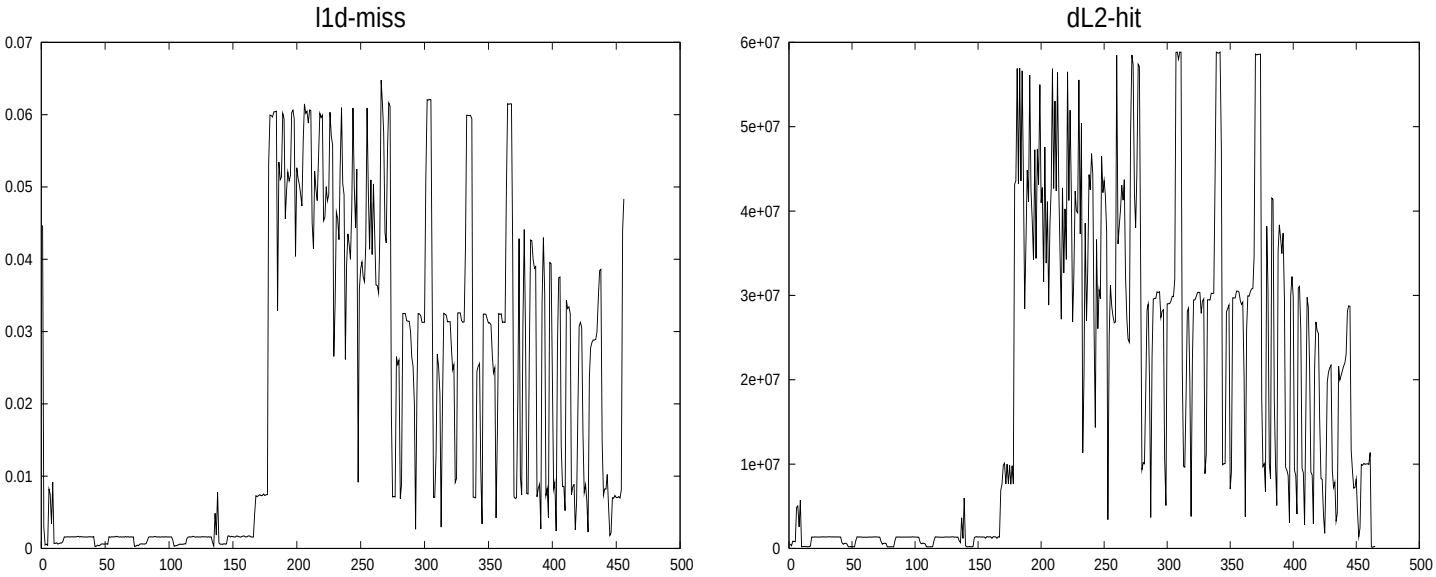

Figure 208: L1 data misses (left) and L2 data hits (right) livegraphs

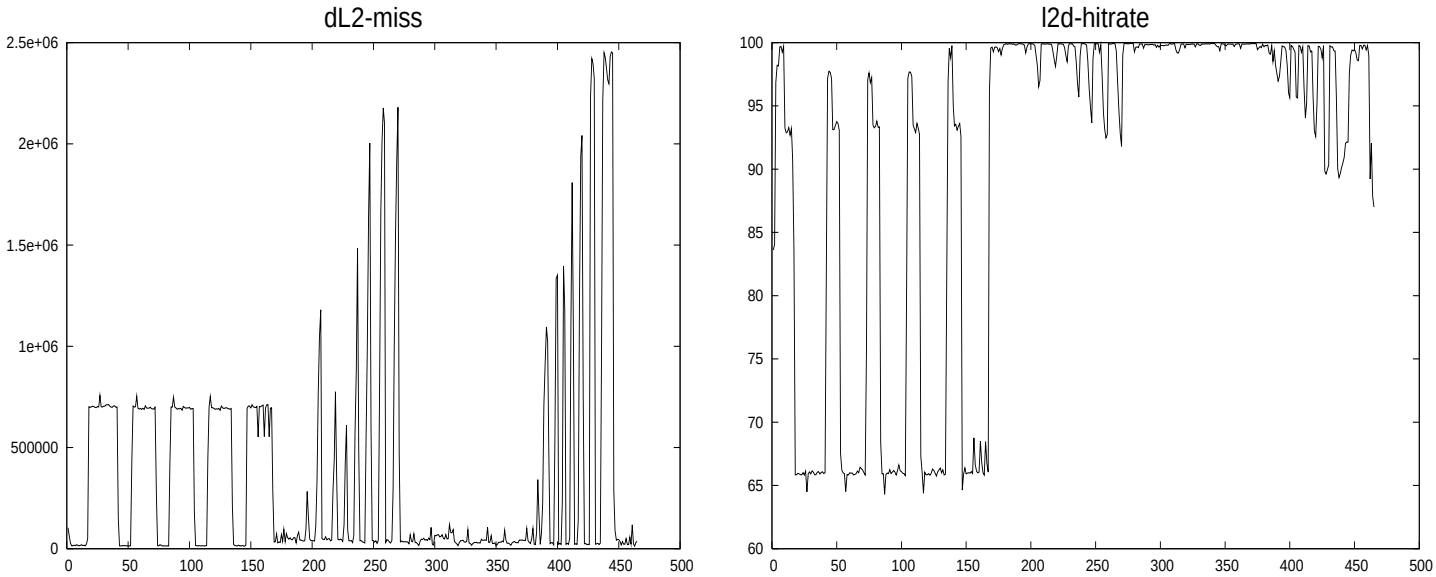

Figure 209: L2 data misses (left) and L2 data hitrate (right) livegraphs

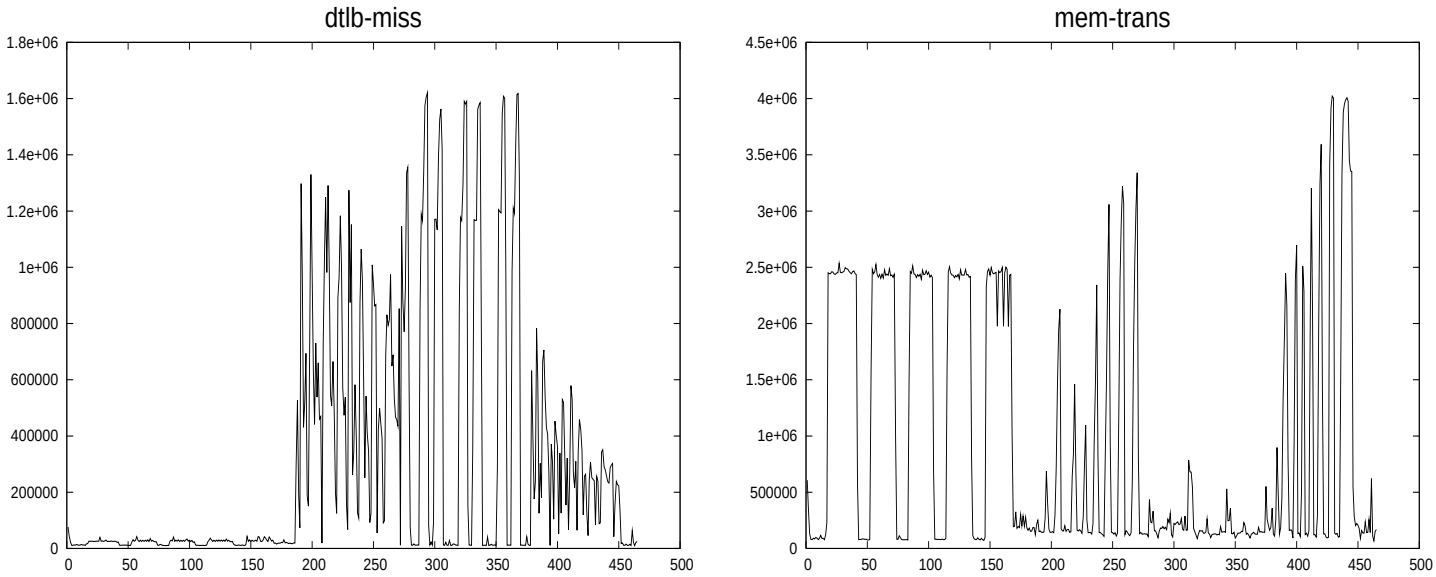

Figure 210: Data TLB misses (left) and Memory Transactions (right) livegraphs

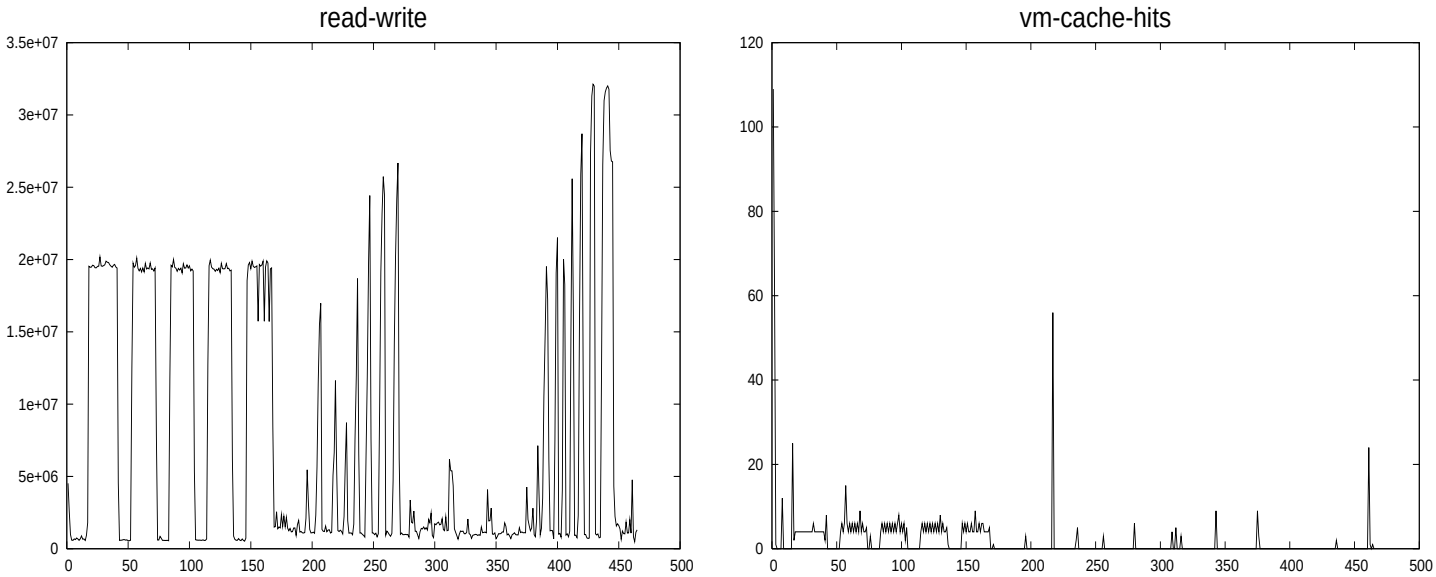

Figure 211: Read/Writes (left) and VM Page Cache Hits (right) livegraphs

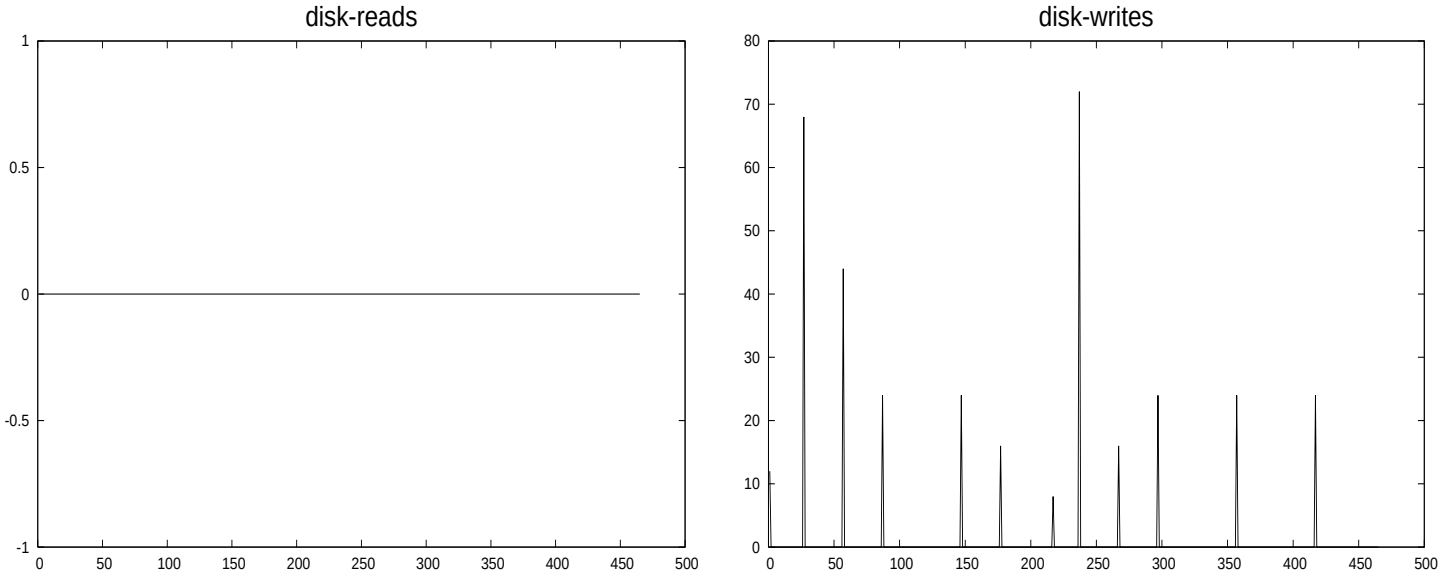

Figure 212: Disk Reads (left) and Disk Writes (right) livegraphs

C.8 TCOFFEE

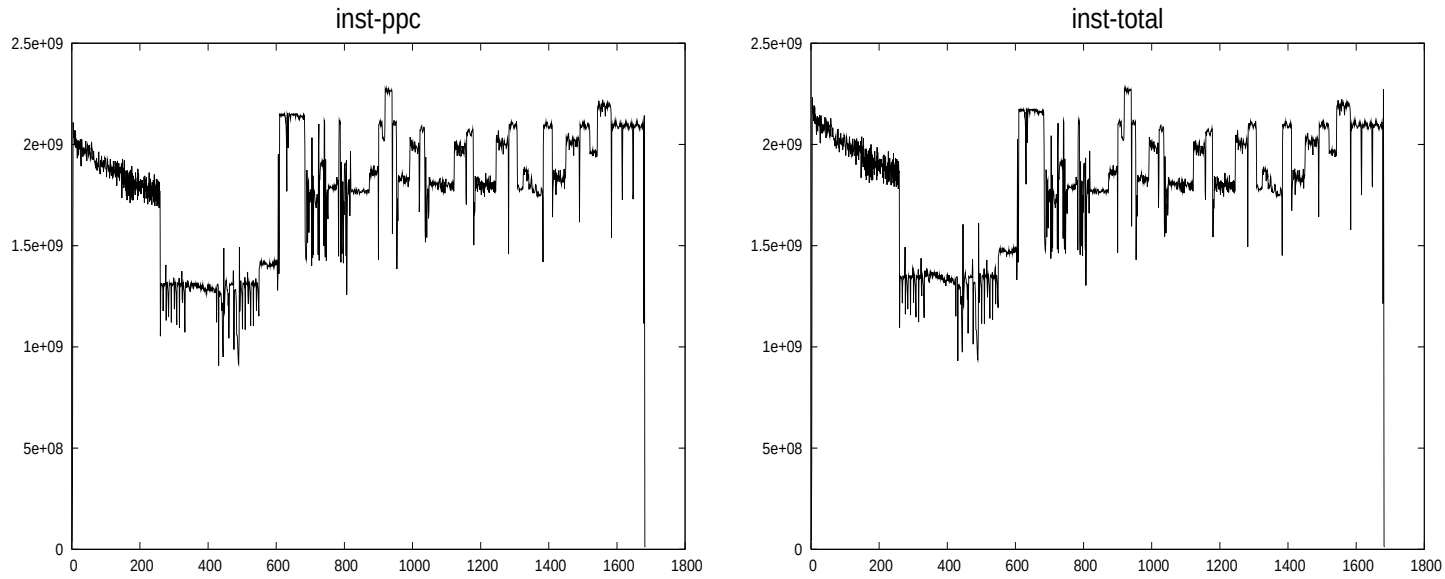

Figure 213: Instructions(ppc) (left) and Instructions (total) (right) livegraphs

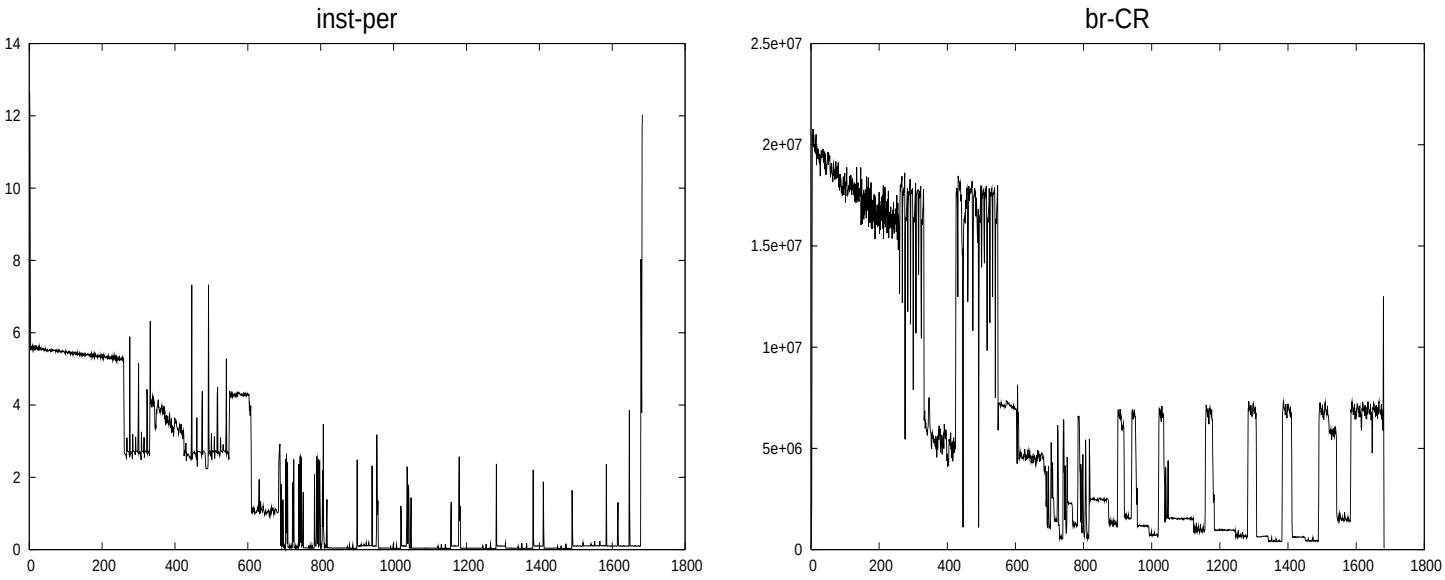

Figure 214: Percentage of io/ld/st instruction (left) and Branch Mispredicts (Condition Register) (right) livegraphs

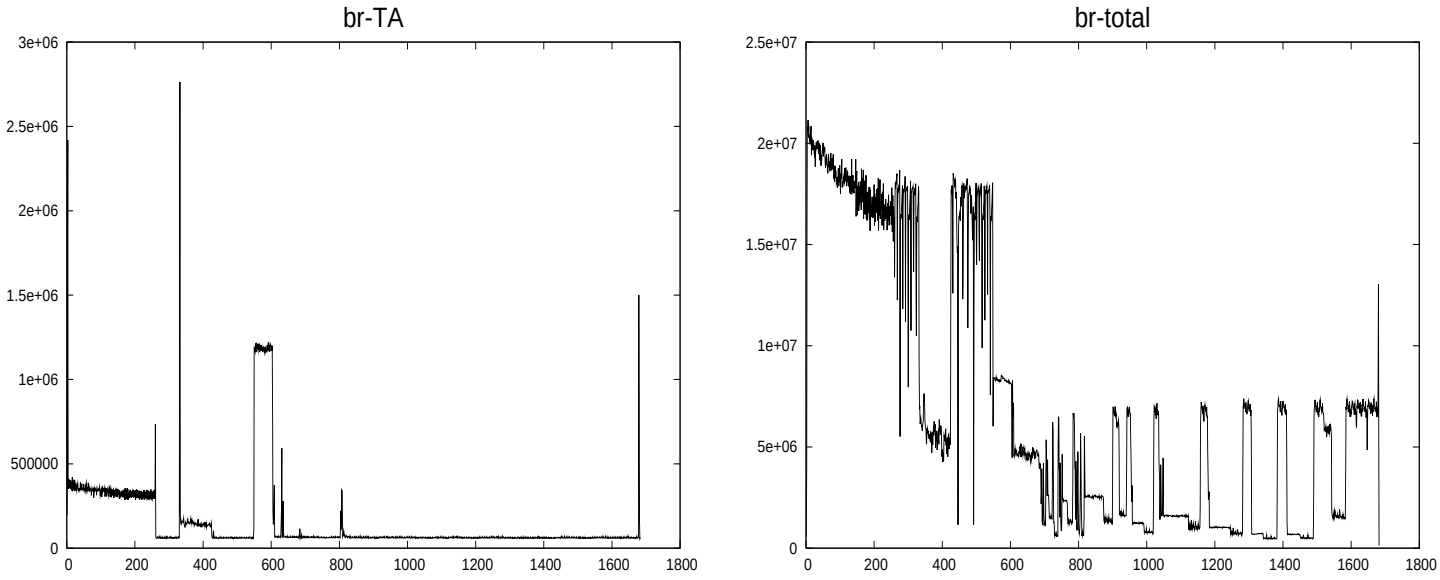

Figure 215: Branch Mispredicts (Target Address) (left) and Branch Mispredicts (total) (right) livegraphs

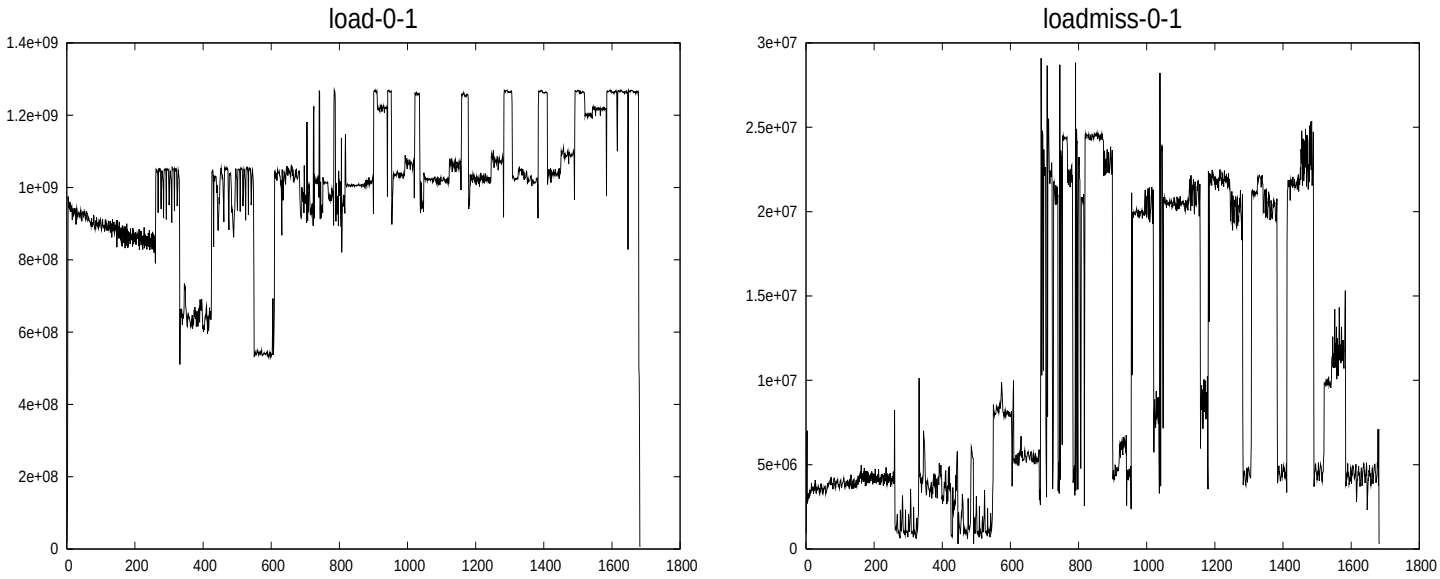

Figure 216: Loads (left) and Load Misses (right) livegraphs

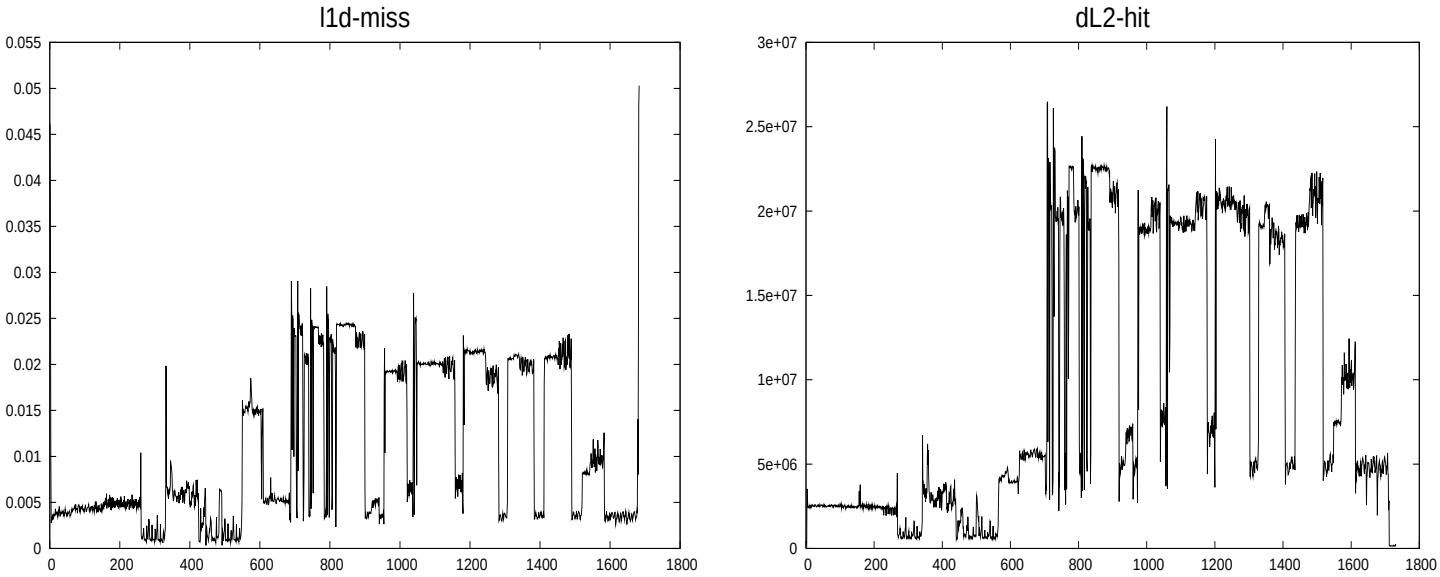

Figure 217: L1 data misses (left) and L2 data hits (right) livegraphs

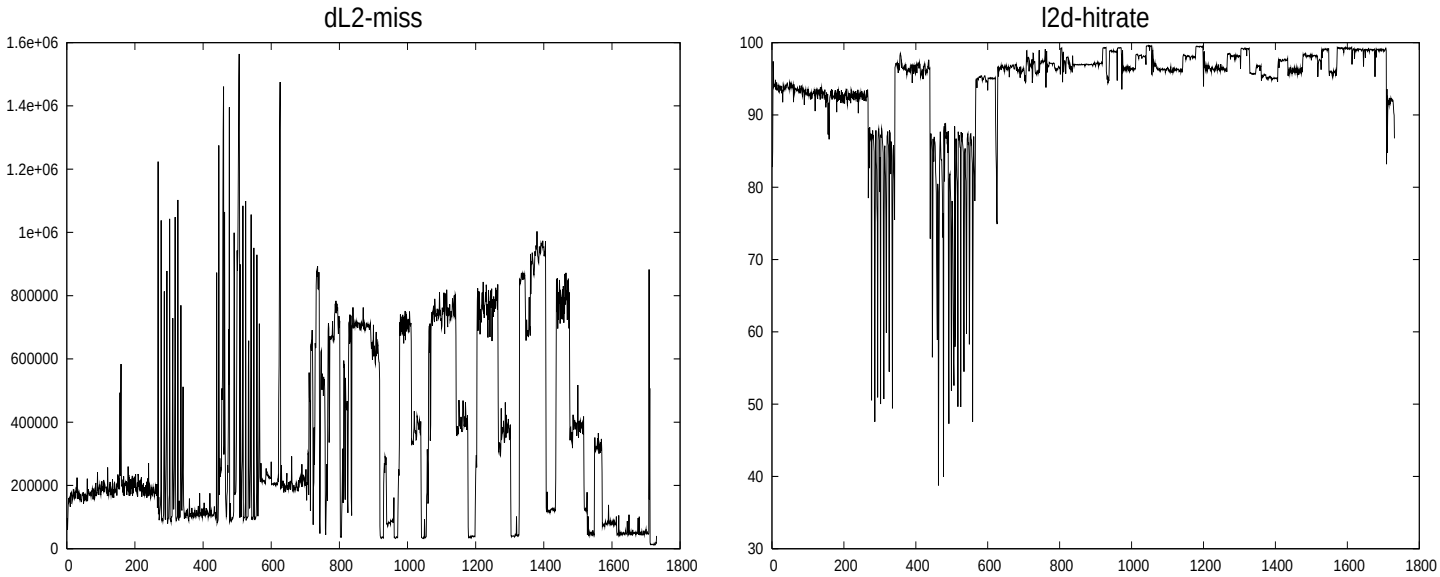

Figure 218: L2 data misses (left) and L2 data hitrate (right) livegraphs

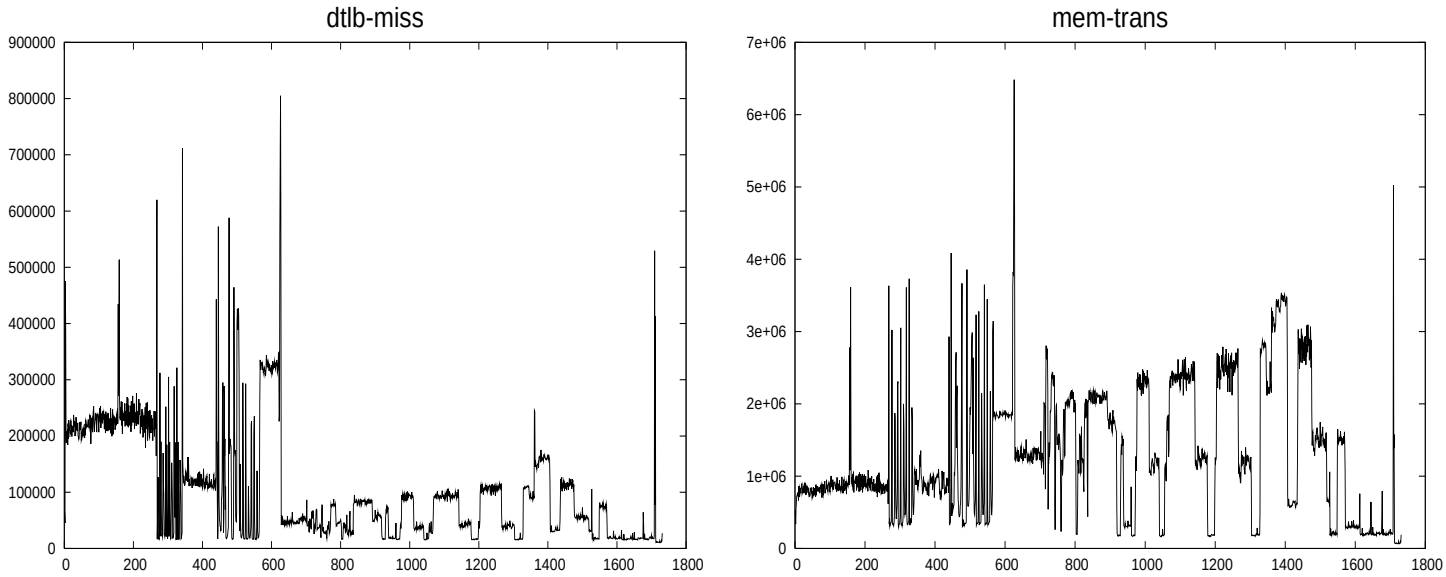

Figure 219: Data TLB misses (left) and Memory Transactions (right) livegraphs

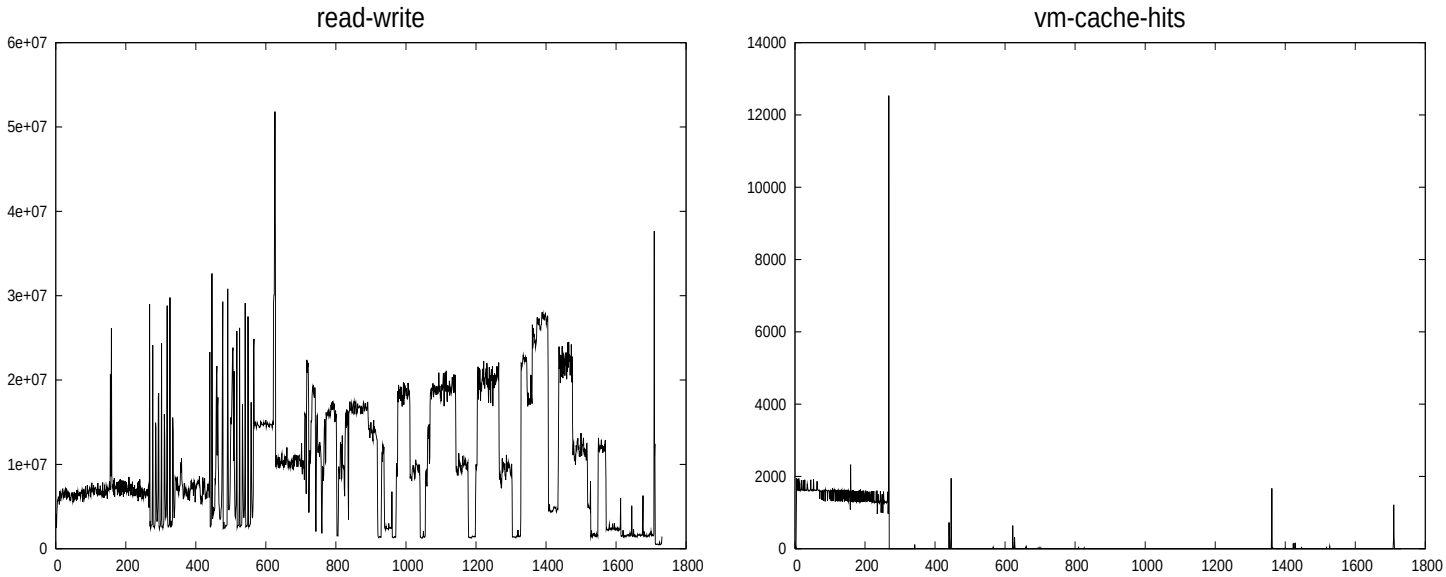

Figure 220: Read/Writes (left) and VM Page Cache Hits (right) livegraphs

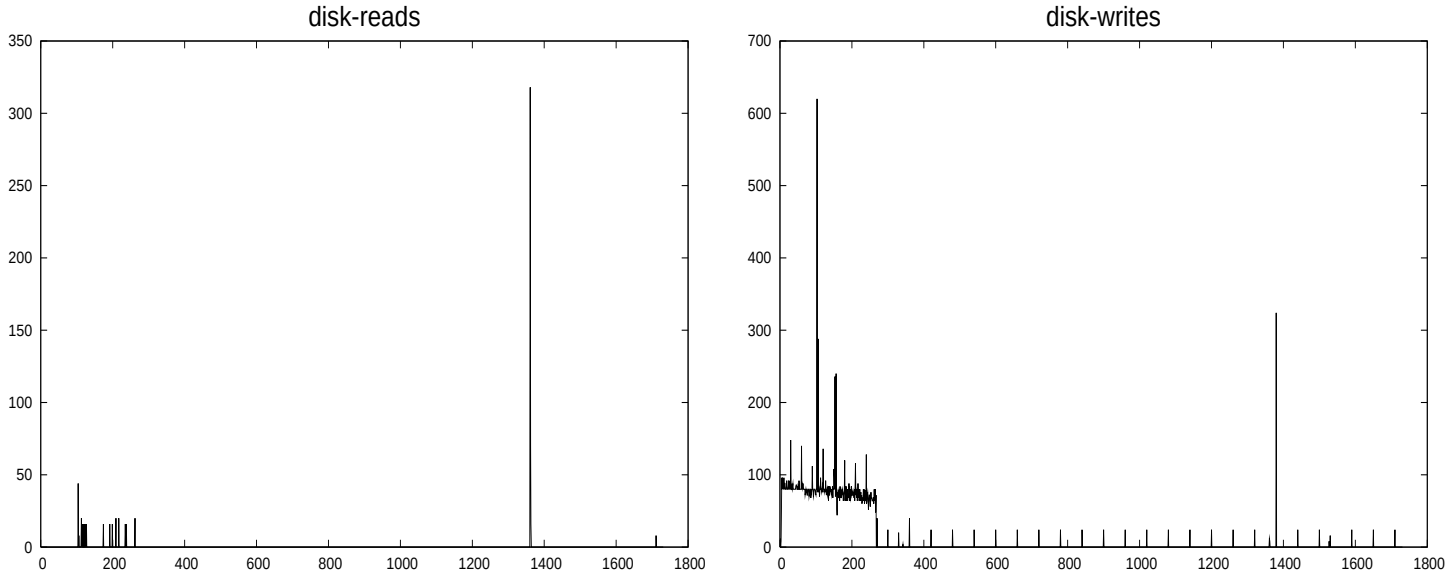

Figure 221: Disk Reads (left) and Disk Writes (right) livegraphs