

# Visualizing Indirect Branch Hot Spots in Object-Oriented Programs

Matthew Holly and Karel Driesen  
School of Computer Science  
McGill University  
Montreal, Quebec, Canada  
{beerman,karel}@cs.mcgill.ca  
<http://www.cs.mcgill.ca/acl>

## Abstract

We demonstrate four visualizations of indirect branch instructions corresponding to switch statements and virtual function calls in object-oriented programs. *Spatial and Temporal hot spot* visualizations highlight code locality. *Footprints* show dynamic program size while *prediction profiles* visualize the regularity of a program phase.

## Keywords

Visualization, Hot spot, Indirect branch, Virtual function call, Polymorphism, Temporal locality, Spatial locality.

## 1. Introduction

Program behavior and performance on modern processors is hard to understand without taking into account on-chip data structures such as caches and branch predictors. These adaptive structures exploit spatial and temporal locality in the instruction and data stream. When the program executes a limited set of instructions for a long duration (a *hot spot*), temporal and spatial locality is typically very high.

We study indirect branch hot spots because object-oriented programs contain many more indirect branches than programs written in a procedural language; every virtual function call has at its core an indirect branch. In previous work ([3] and [5]), we studied indirect branch prediction micro-architectures aimed at reducing the cost of indirect branches.

Here we visualize indirect branch activity in order to gain a better understanding of the dynamic behavior and run time size of the indirect branch working set. These visualizations also provide a view on the whole program, in particular with regard to the occurrence and re-occurrence of different phases in program execution.

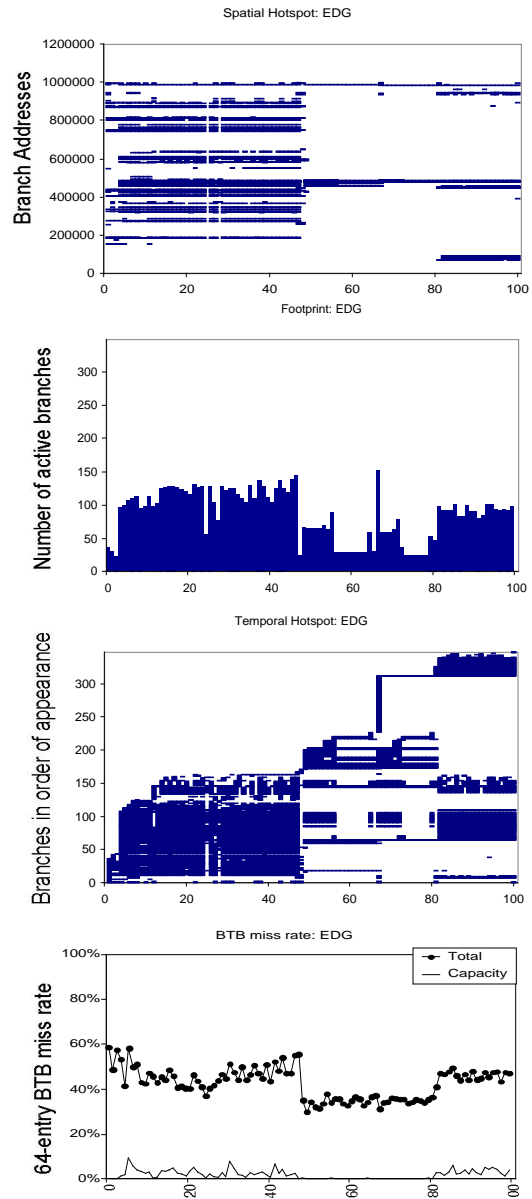


Figure 1. EDG: 350 static 584893 dynamic

## 2. Visualization Methods

We demonstrate our visualization techniques on indirect branch execution traces. We show four visualizations of indirect branches in SPARC executables from 13 large object-oriented C++ and C programs, each of which execute at least one indirect branch every 200 instructions (excluding procedure returns). These traces were used in [1] as an indirect branch prediction benchmark. In related work ([4],[6],[7],[10]), we applied hot spot visualization to a variety of other domains and obtained similar results, which leads us to believe that these visualization techniques are equally applicable to instruction and data cache working sets, among others.

On the previous page we show EDG, a C++ front end. The X-axis of each graph shows the entire trace in 100 segments. The EDG trace contains 350 different indirect branch instructions (static) which combine for 548,893 executions (dynamic). Each segment therefore collects information from 5489 executions.

### 2.1 Spatial Hot Spot Visualization

The first visualization, to the best of our knowledge first used by Merten et al. [11], highlights the spatial locality of indirect branch instructions. Two instructions that are close together in memory will appear close together on the graph when viewed using this method. The mapping to achieve this is both simple and intuitive; the Y-axis value of a point on the graph corresponds to its indirect branch address. A *hot spot* in this context is a working set that remains stable for a significant amount of time. For EDG, we observe three different phases in the program run. The first phase executes indirect branches from a large number of different locations in memory, the second phase executes a hot spot of two locations, while the third and final phase exercises branches from three distinct locations.

Spatial hot spot visualization gives a rough, often inaccurate impression of the size of the branch working set. The wide range of branch addresses causes nearby branch instructions to overlap, so that they appear as one line in the graph.

### 2.2 Footprint Visualization

Footprint visualization<sup>1</sup> shows the size of indirect branch working sets, which is equal to the number of *different* branches exercised within a sample. The precise rendering of working set size comes at a high price because all recurrence and locality information is lost.

---

<sup>1</sup> This terminology derives from memory footprints.

### 2.3 Temporal Hot Spot Visualization

Temporal hot spot visualization re-numbers branches *in order of appearance*. Thus the first branch address seen in the trace is given the number 0, the second number 1, and so on. This temporal stamp then becomes the new Y-axis value of the indirect branch site. This scheme has several benefits for visualization:

- the vertical size of a hot spot is proportional to the working set size and therefore shows similar information as the footprint.
- space on the Y-axis is better used than in spatial hot spots, since every value appears in at least one sample. This is in contrast to spatial hot spot visualization where large unused horizontal gaps in the graph are common.
- temporal visualization shows highly characteristic patterns for different program phases by the interleaving of recurring branches (horizontal lines) and non-recurring branches (vertical lines). For example, the second phase of EDG clearly consists of two very similar but separate phases, which are hard to distinguish in spatial hot spot visualization.
- two runs of the same program on different machines will look different when real addresses are used. Re-numbering ensures that identical runs have identical profiles, and therefore renders a platform-independent visualization of program behavior.

### 2.4 Branch Target Buffer Misprediction Rates

The final graph shows the total misprediction rate (*Total*) of a 64-entry, fully associative Branch Target Buffer (BTB), and the portion of the misprediction rate which is due to capacity misses (*Capacity*). We obtained the latter by subtracting the misprediction rate of an ideal BTB (unlimited, fully associative), which encounters no capacity misses. For EDG, capacity misses constitute a very small portion of the total misses, even though the footprint shows that the branch working set is up to twice the size of the BTB. Together with the footprint graph, this graph suggests that program phases with smaller footprints exhibit lower misprediction rates. In other words, indirect branch targets do not vary as much as they do in programs with large footprints

## 3. Benchmarks

Our benchmark suite consists of large object-oriented C++ applications ranging from 8,000 to over 75,000 non-blank lines of C++ code each (see Table 1), and *beta*, a compiler for the Beta programming language [9], written in Beta. We also measured the SPECint95 benchmarks, excluding those benchmarks which execute indirect branches less frequently

than once every 200 instructions (*compress*, *m88ksim*, *vortex*, *jpeg*, and *go*). Together, the benchmarks represent over 500,000 non-comment source lines. All C and C++ programs were compiled with GNU gcc and run under the *shade* instruction-level simulator [1] to obtain traces of all indirect branches. Procedure returns were excluded because they are usually handled by a return address stack [8]. All programs were run to completion or until six million indirect branches were executed. In *jhm* we excluded the initialization phase by skipping the first 5 million indirect branches.

For each benchmark, Table 1 gives a description, the number of indirect branches executed (dynamic), as well as the number of *different* indirect branch instructions executed (static). For instance, the XLISP trace contains six million indirect branch executions, in which only thirteen different branch instructions participate.

The last three columns show the percentage of branches executed (dynamic) which are virtual function calls, switch statements, and other indirect branches. For example, XLISP executes hardly any switches and no virtual functions, while LCOM executes only virtual functions and switches.

Benchmark	Description	indirect branches (static)	indirect branches (dynamic)	%virtual	%switch	%indirect
idl	IDL compiler <sup>a</sup>	543	1,883,641	93.2	3.2	3.6
jhm	JHM <sup>b</sup> 6-12M	155	6,000,000	93.6	1.2	5.2
self	Self-93 VM	2658	6,000,000	76.0	4.4	19.6
xlisp	SPEC95	13	6,000,000	0.0	0.1	99.9
troff	GNU groff 1.09	161	1,110,592	73.7	12.5	13.8
lcom	HDL <sup>c</sup> compiler	328	1,737,751	63.2	36.8	0.0
perl	SPEC95	24	300,000	0.0	31.7	68.3
porky	scalar optimizer <sup>d</sup>	285	5,392,890	70.6	23.8	5.6
ixx	IDL parser <sup>e</sup>	203	212,035	46.5	52.2	1.3
edg	C++ front end	350	548,893	0.0	62.4	37.6
eqn	equation typesetter	114	296,425	33.8	66.2	0.0
gcc	SPEC95	166	864,838	0.0	31.5	68.5
beta	BETA compiler	376	1,005,995	0.0	2.3	97.7

**Table 1.** Benchmarks

- <sup>a</sup> SunSoft version 1.3
- <sup>b</sup> Java High-level Class Modifier
- <sup>c</sup> Hardware Description Language compiler
- <sup>d</sup> SUIF 1.0
- <sup>e</sup> Fresco X11R6 library

## 4. Position Statement

The aims of this research are described below.

### 4.1 Satisfy our Curiosity

In the short run, we want to be able answer the following detail questions on any running program:

- where does the program spend its time in memory?
- when does a program phase occur for the first time?
- when does a program phase re-occur?
- how much hardware is occupied by the program at any given time during its execution?
- how regular or predictable is the program?

### 4.2 Anticipate Software/Hardware Miss-matches

Visualizations allow us to anticipate problems caused by insufficient or miss-matched hardware resources. For example, a small Branch Target Buffer can reduce performance because the working set of branches overloads the BTB. Similarly, insufficient I-Cache or DRAM space can reduce performance.

### 4.3 Design Reconfigurable and Adaptive Hardware

We expect that different programs will have different hardware requirements. For example, Table 2 (extract from [3]) shows that SELF is better served by a BTB than by a two-level predictor, both employing the same 128-entry, fully

Benchmark	BTB	2-Level P2
self	25%	37%
xlisp	14%	7%

**Table 2.** 128-entry misprediction rates

associative table to store branch targets. For XLISP, the situation is reversed: two-level prediction has half the misprediction rate of a BTB. Figure 4 and Figure 5 in the Appendix provide ample information to explain this effect. Within a 60,000-branch interval of execution SELF touches between 500 and 1000 different branches, while XLISP touches no more than 12 different branches. SELF already suffers from 50% capacity misses on a 128-entry BTB, which uses only one entry per branch, where a two-level predictor of path length 2 uses multiple entries, one for each path leading up to a branch. This is advantageous for XLISP, which has table space to spare, but not for SELF, with its large indirect branch footprint.

We are planning experiments with reconfigurable two-level predictors that monitor the branch stream and adapt to program phases by changing the predictor configuration.

## 4.4 Study the Life of Programs

We like the sense of discovery that comes with looking at a program execution visualization for the first time. We are convinced that the understanding gained from watching programs in action on different types of hardware will contribute to our understanding of computation in general, and we hope this will lead to better hardware designs and high-performance applications.

## 5. Related Work

Merten et al. [11] first used spatial hot spot visualization of branch targets (both conditional and indirect). Jinsight ([2]) visualizes many high-level aspects of object-oriented program behavior, where we focus more on the low-level hardware perspective. COMB ([12]) implements a wide variety of techniques applicable to software visualization which should also improve hot spot visualization.

## 6. Conclusions and future work

We have shown four different visualizations of the indirect branch working set of thirteen large object-oriented programs. These visualizations characterize the temporal locality, the spatial locality, the run-time size and the predictability of indirect branches. Among the four techniques, temporal hot spot visualization is most information-rich, since it renders accurately both the size and recurrence of branch working sets, combining aspects of spatial hot spots and footprints. Temporal hot spots have the added advantage of being invariant to the hardware platform being used.

We plan to apply these visualization techniques to other instruction and data domains, such as load addresses and object allocation types, and to integrate temporal hot spot visualization in an environment that allows switching between source code and visualization.

We also plan to refine the hot spot visualizations by emphasizing the hottest spots using grayscale or color. The resulting tool will be used to investigate reconfigurable hardware that takes advantage of run time hot spot profiling.

**Acknowledgments.** This work was supported in part by NSERC (Canada), and FGSR McGill. We would like to thank Olivier Zendra for his comments on earlier versions of this paper.

## 7. References

- [1] Robert F. Cmelik and David Keppel. *Shade: A Fast Instruction-Set Simulator for Execution Profiling*. Sun Microsystems Laboratories, Technical Report SMLI TR-93-12, 1993.
- [2] De Pauw, W., Mitchell, N., Robillard, M., Sevitsky, G., and Srinivasan, H. Drive-by Analysis of Running Programs, *Proceedings for Workshop on Software Visualization*, International Conference on Software Engineering, May 12-13, 2001, Toronto.
- [3] Karel Driesen. *Efficient Polymorphic Calls*, Kluwer Academic Publishers, 2001.
- [4] Karel Driesen, Nagi Basha, David Eng, Matt Holly, John Jorgensen, Georges Kanaan, Babak Mahdavi, Qin Wang. *Visualizing Hot Spots in Various Domains*, Technical Report SOCS-01.3, School of Computer Science, McGill University, Montreal, Canada, March 2001. Also appeared in the proceedings of the Workshop on Software Visualization, ICSE2001.
- [5] Karel Driesen and Urs Hölzle. Accurate Indirect Branch Prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, 1998.
- [6] Matthew Holly. *Temporal and Spatial Program Hot Spot Visualization*. Technical Report SOCS-01.6, School of Computer Science, McGill University, Montreal, Canada, May 2001.
- [7] John Jorgensen. *Soot Leaves Hot Spots*. Technical Report SOCS-01.7, School of Computer Science, McGill University, Montreal, Canada, May 2001.
- [8] David Kaeli and P. G. Emma. Branch History Table Prediction of Moving Target Branches due to Subroutine Returns. *ISCA '91 Proceedings*, May 1991.
- [9] Ole Lehrmann Madsen, Birger Moller-Pedersen, Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley 1993.
- [10] Babak Mahdavi and Karel Driesen. *Heap Hot Spots Visualization in Java*. Technical Report SOCS-01.8, School of Computer Science, McGill University, Montreal, Canada, May 2001.
- [11] Matthew Merten, Andrew Trick, Christopher George, John Gyllenhaal, Wen-mei Hwu. A Hardware- Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA)*, 1999.
- [12] Steven P. Reiss. Bee/Hive: A Software Visualization Back End, *Proceedings for Workshop on Software Visualization*, International Conference on Software Engineering, May 12-13, 2001, Toronto.

## Appendix :Benchmark Visualizations

The following pages show the four visualizations for all twelve benchmarks in Table 1 (Edg is not repeated)

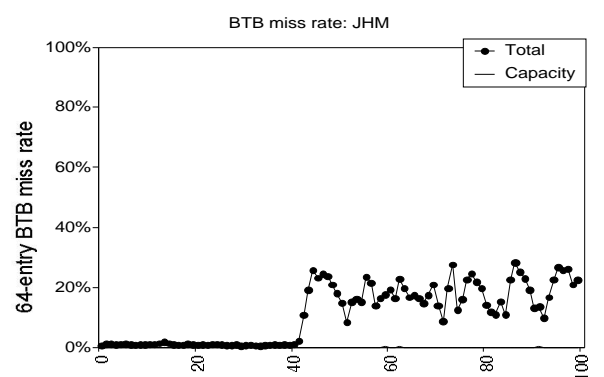
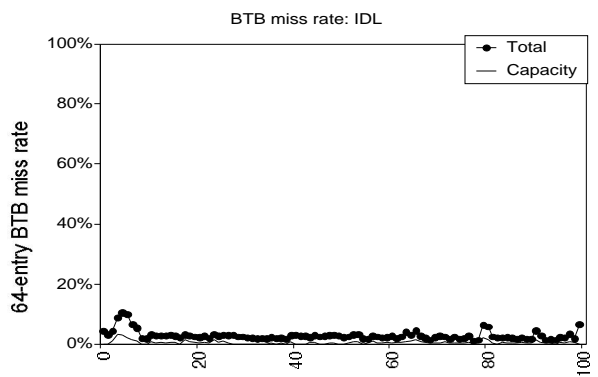
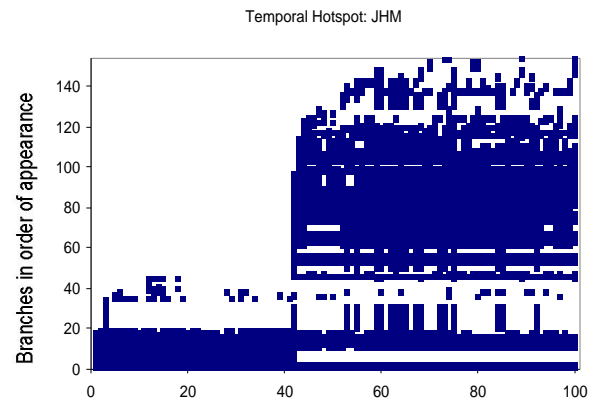
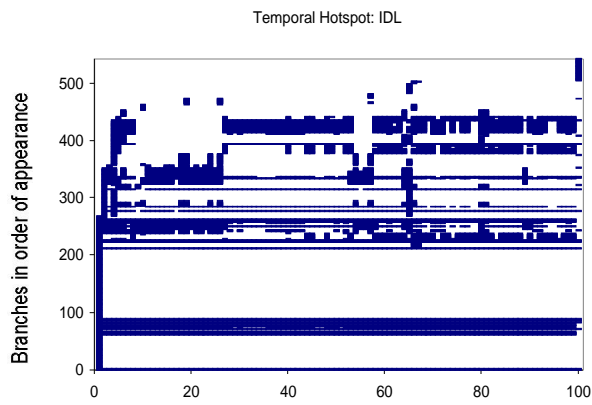
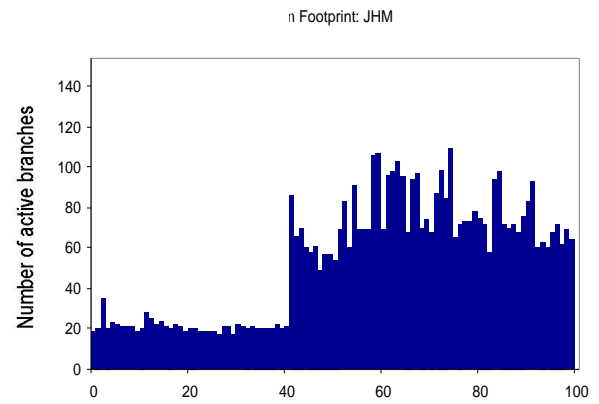
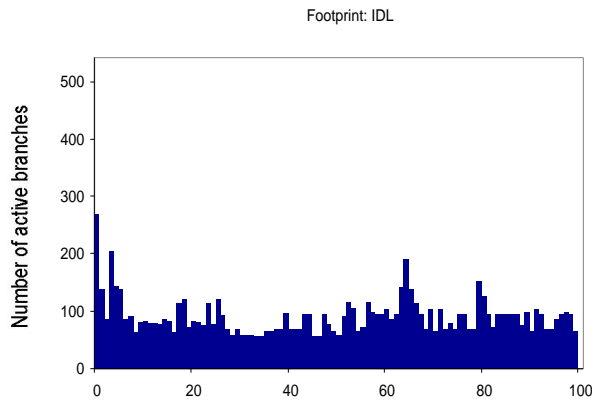
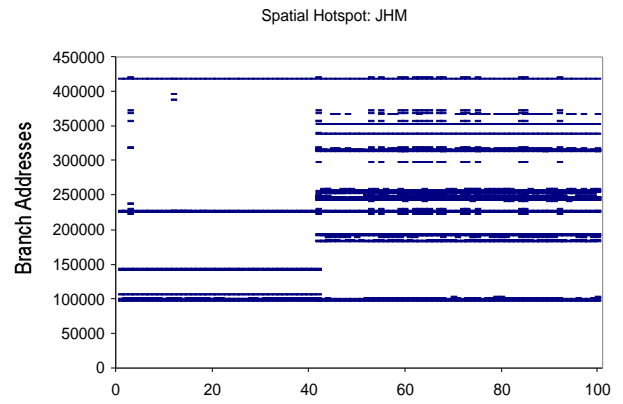
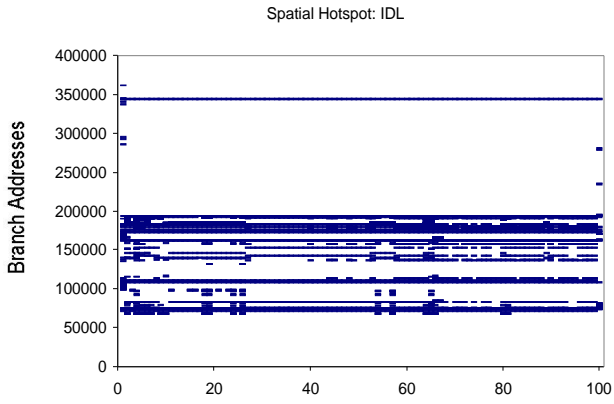


Figure 2. IDL: 543 static 1883641 dynamic

Figure 3. JHM: 155 static 6000000 dynamic

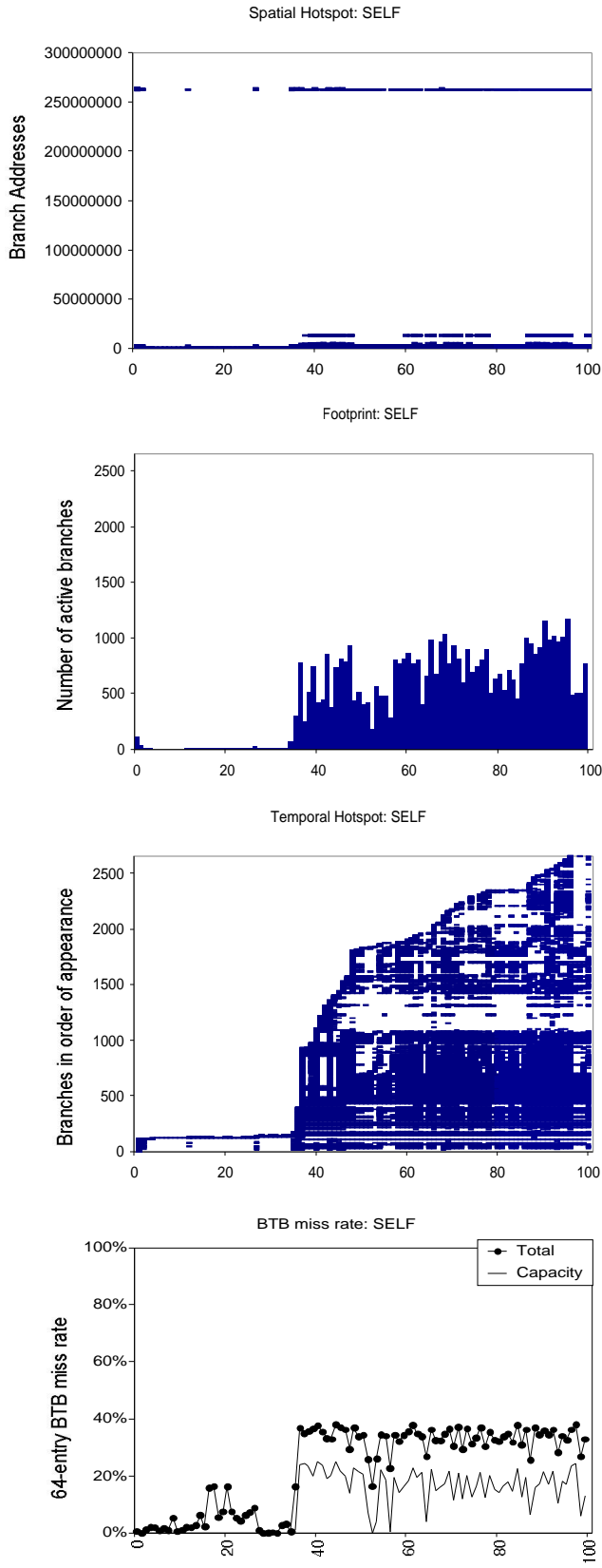


Figure 4. SELF: 2658 static 6000000 dynamic

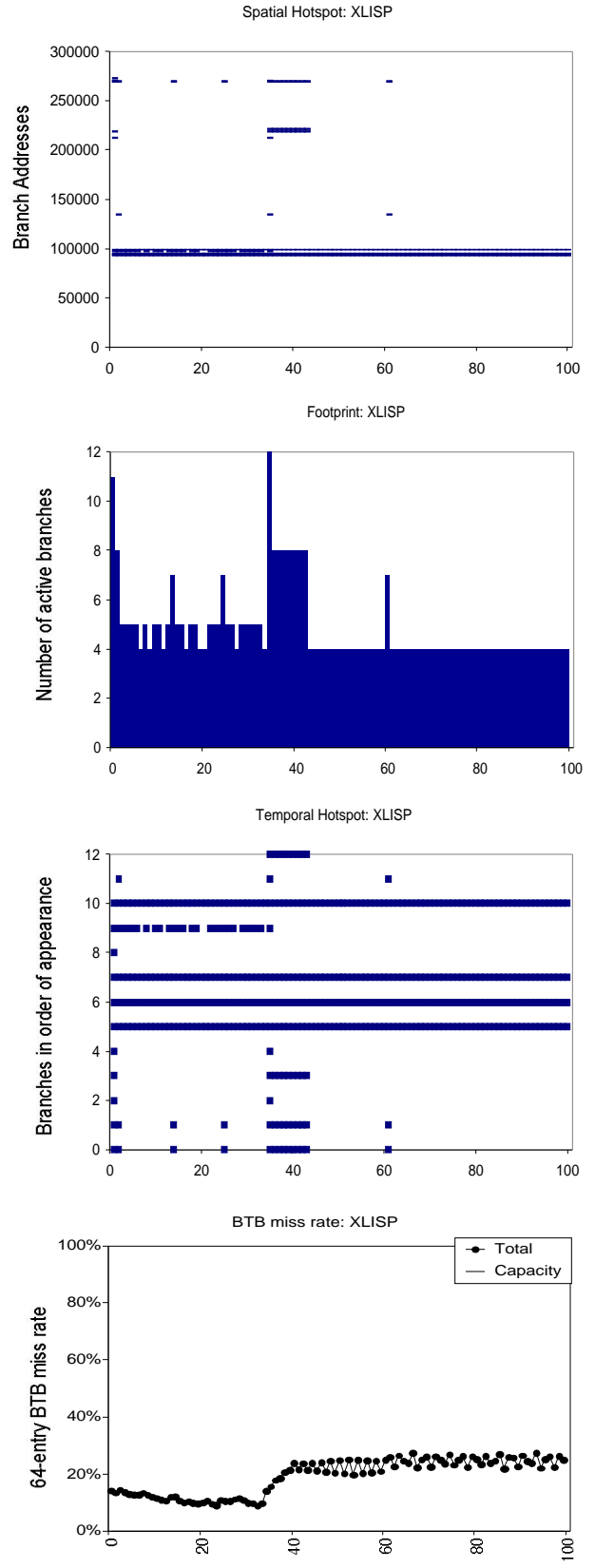
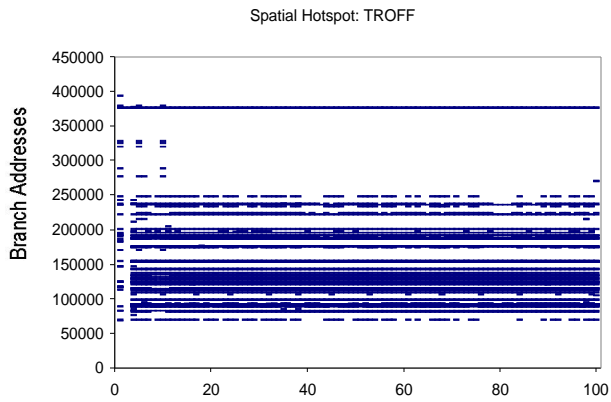
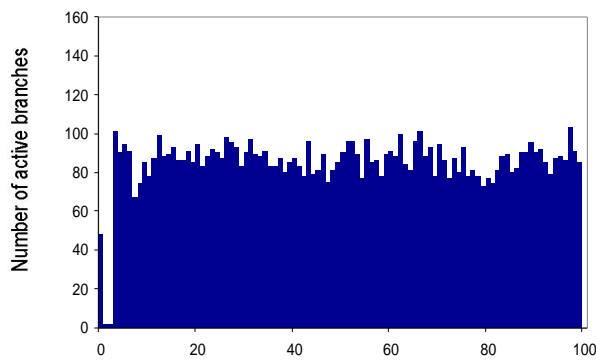


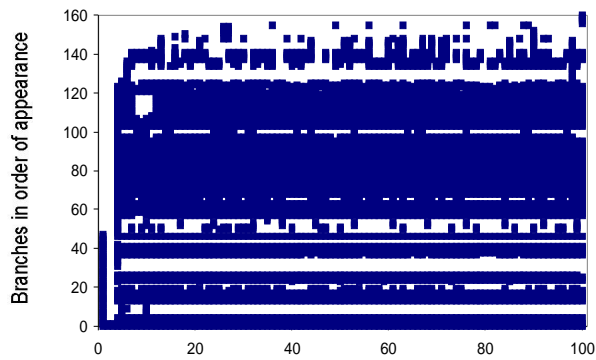
Figure 5. XLISP: 13 static 6000000 dynamic



Footprint: TROFF



Temporal Hotspot: TROFF



BTB miss rate: TROFF

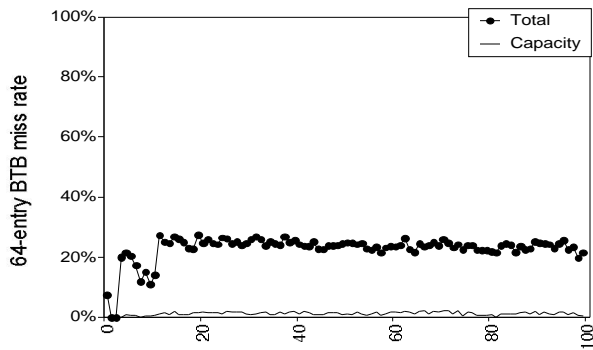
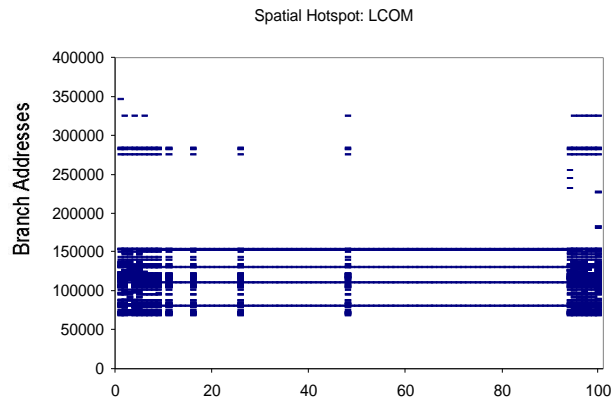
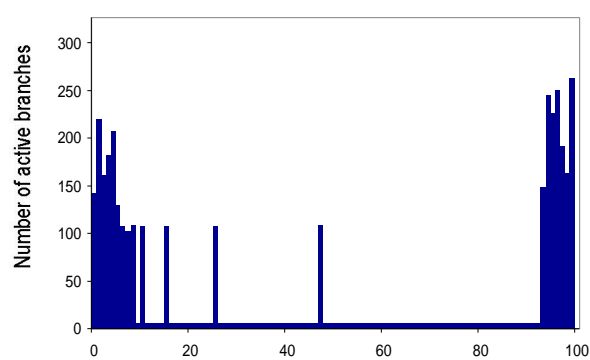


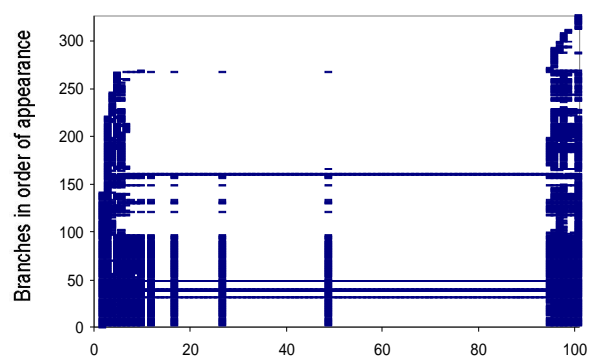
Figure 6. TROFF: 161 static 1110592 dynamic



Footprint: LCOM



Temporal Hotspot: LCOM



BTB miss rate: LCOM

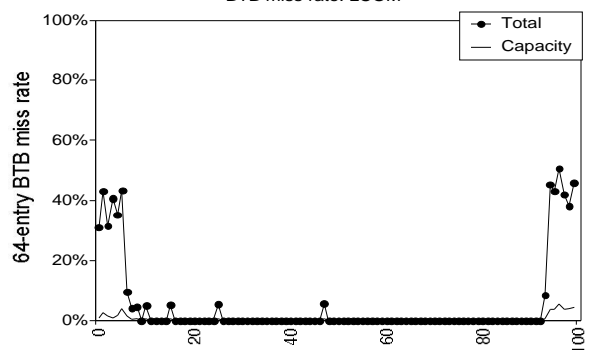


Figure 7. LCOM: 328 static 1737751 dynamic

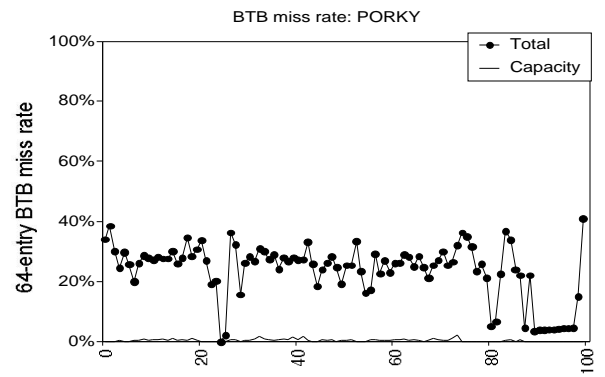
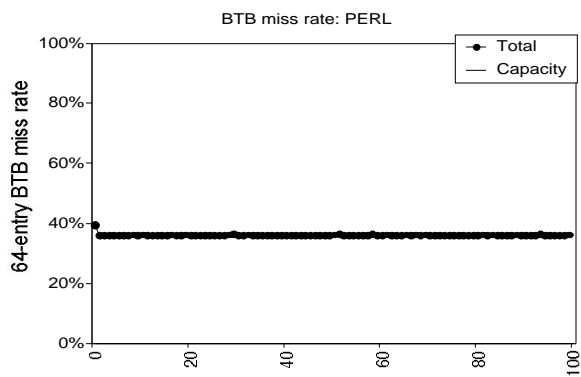
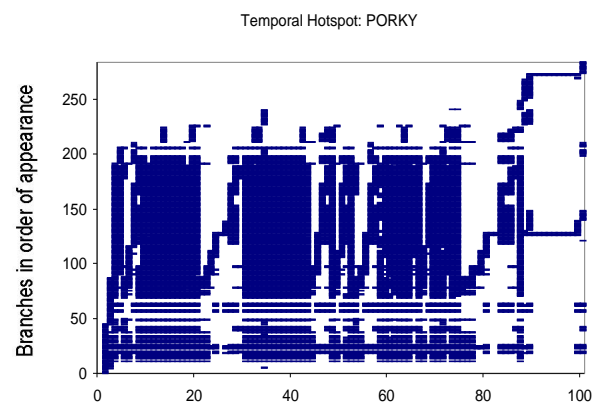
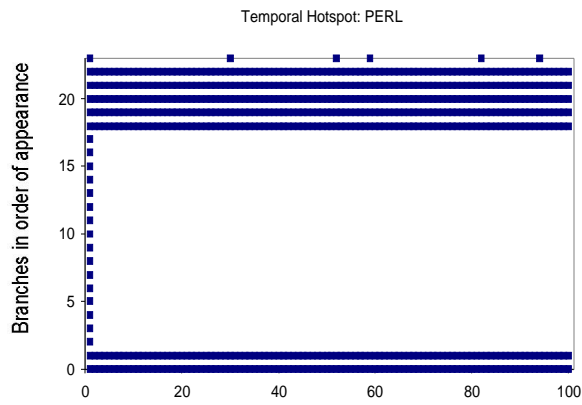
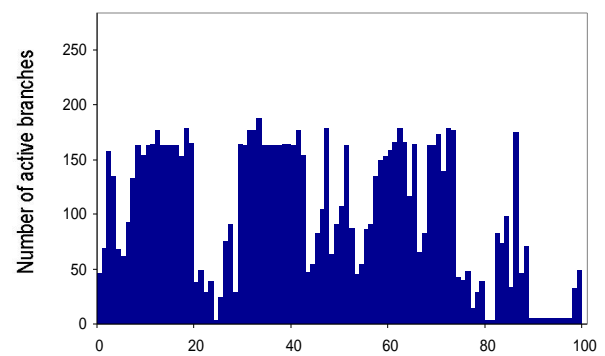
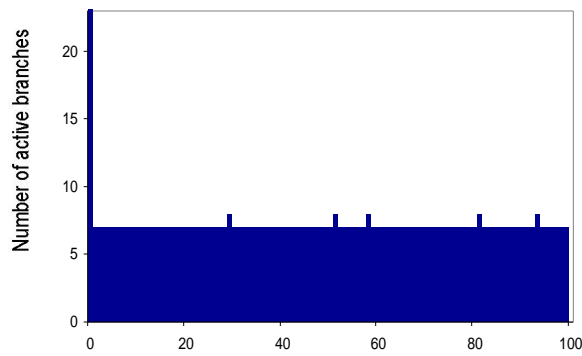
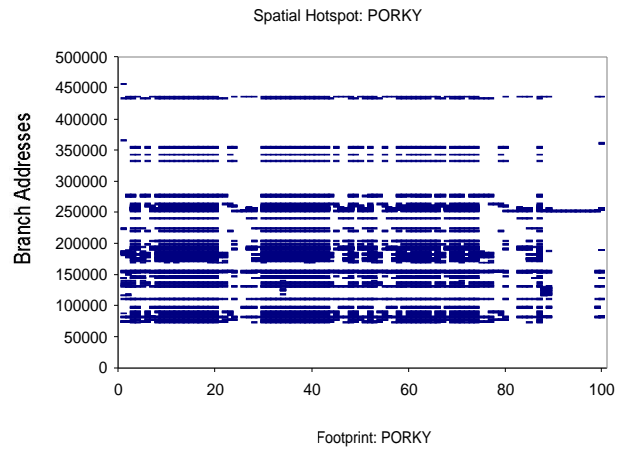
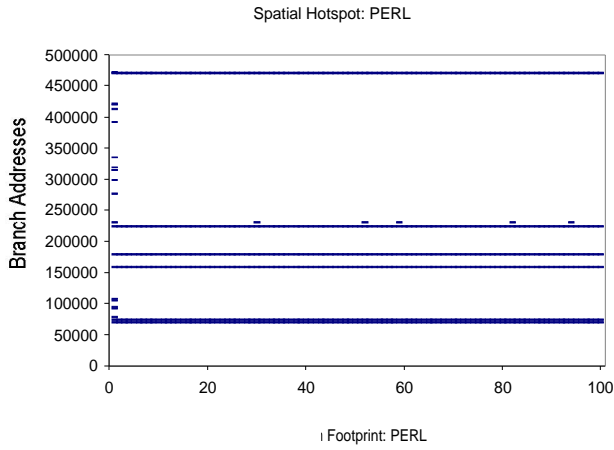


Figure 8. PERL: 24 static 300000 dynamic

Figure 9. PORKY: 285 static 5392890 dynamic

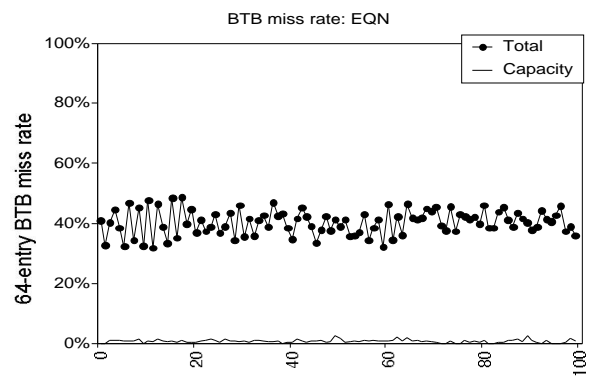
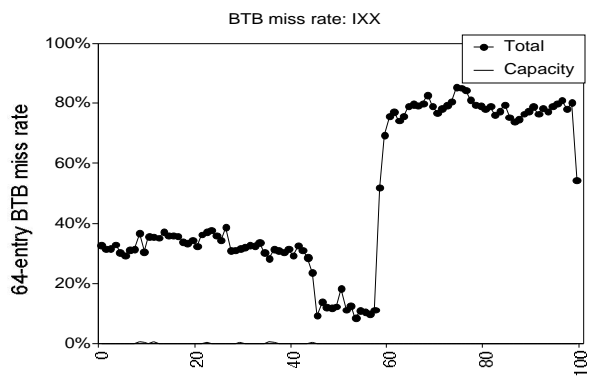
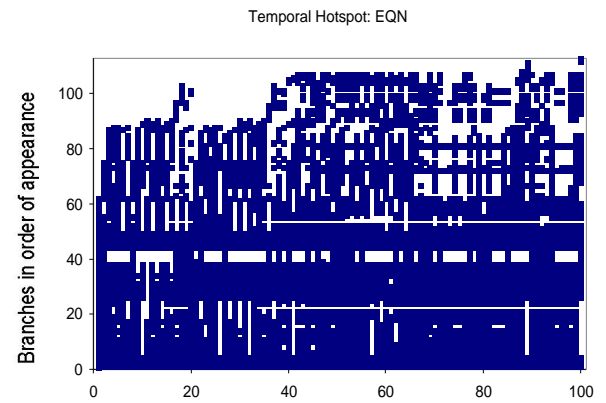
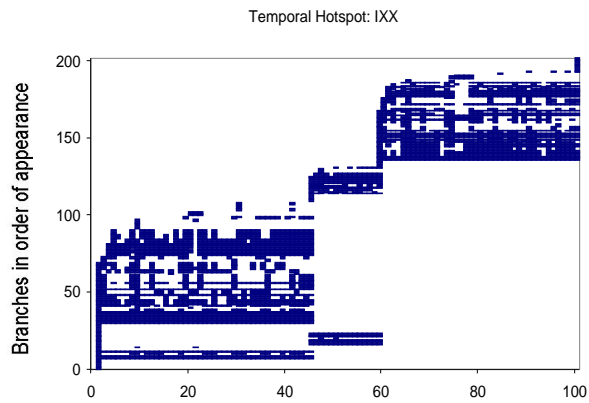
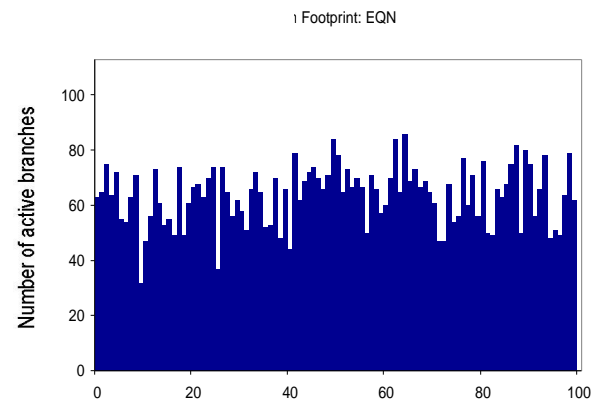
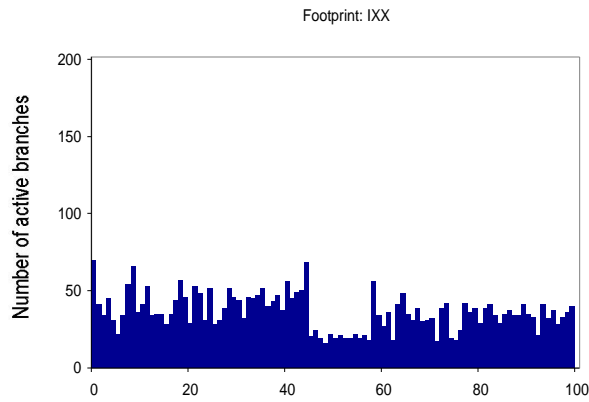
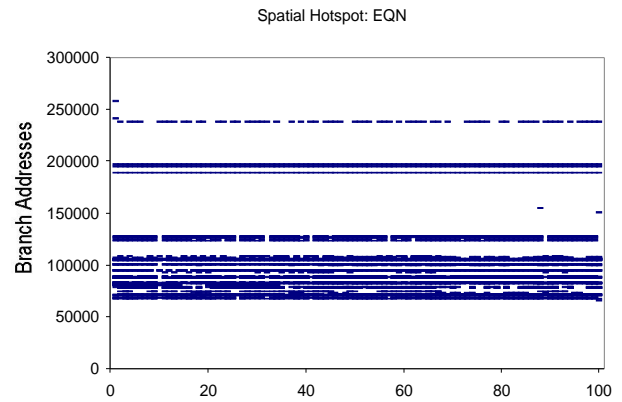
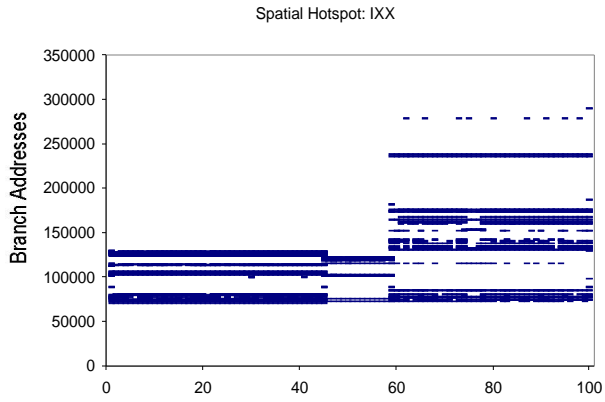


Figure 10. IXX: 203 static 212035 dynamic

Figure 11. EQN: 114 static 296425 dynamic

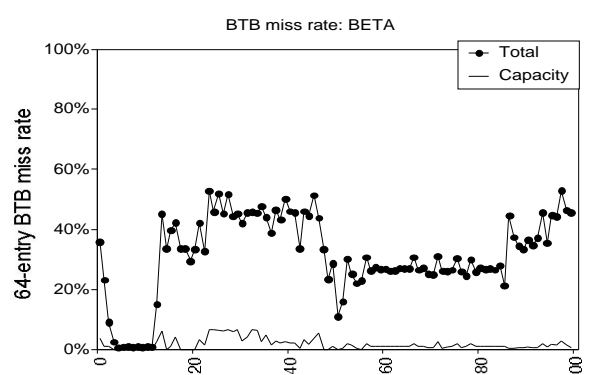
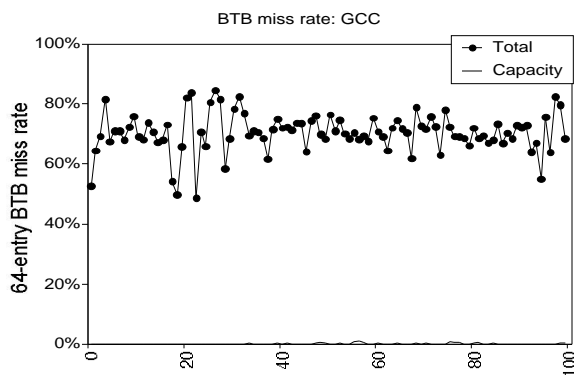
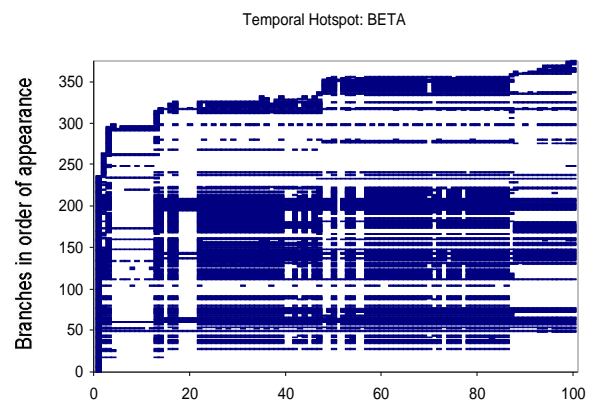
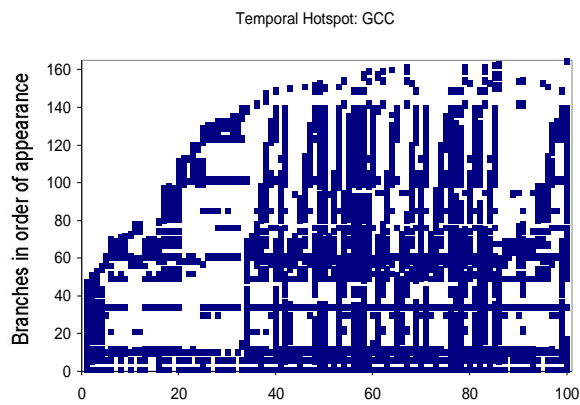
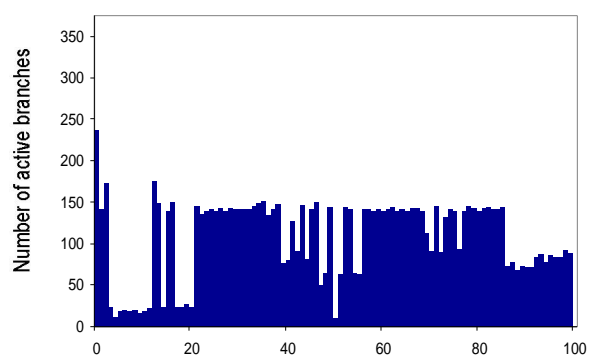
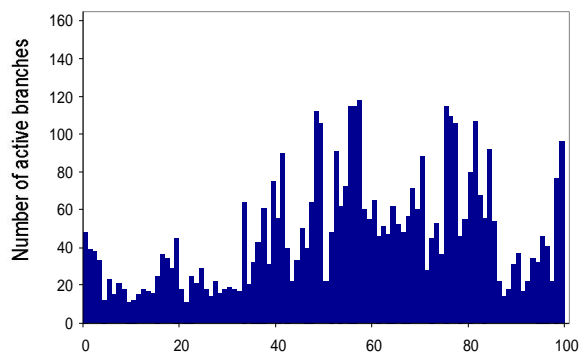
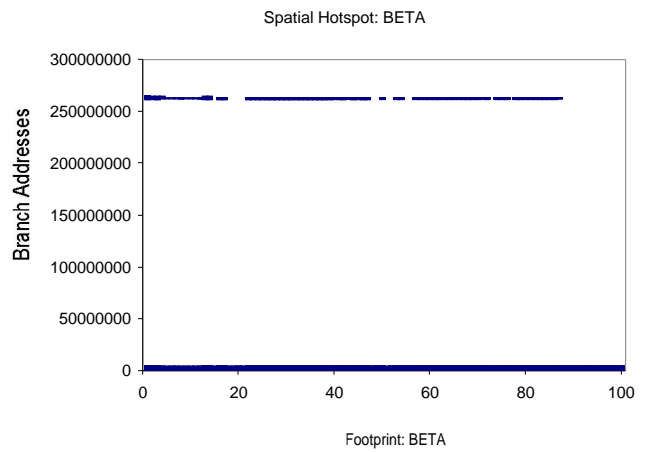
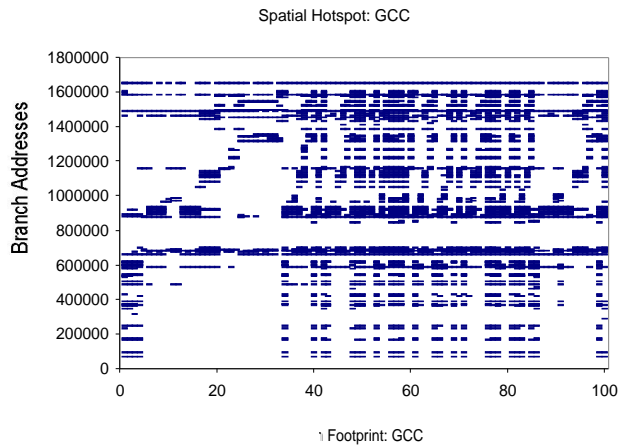


Figure 12. GCC: 166 static 864838 dynamic

Figure 13. BETA: 376 static 1005995 dynamic

