Visualizing Hot Spots in Various Domains

Karel Driesen, Nagi Basha, David Eng, Matt Holly, John
Jorgensen, Georges Kanaan, Babak Mahdavi, Qin Wang.
School of Computer Science, McGill University
Montreal, Quebec, Canada
www.cs.mcgill.ca/acl

Abstract

A program hot spot is a collection of instructions which are executed repeatedly, typically in an inner loop of some program phase. The size of a hot spot indicates the minimum size of an instruction cache that is able to execute all instructions without encountering cache capacity misses[1]. The duration of a hot spot indicates whether it is worth the effort to optimize the loop, for example by a dynamic optimizing compiler such as employed in the HotSpot™ Java Virtual Machine. Hot spots are also important in value domains other than instruction addresses, such as branch addresses (Branch Target Buffer), load/store addresses (Data Cache), and allocation sites (Automatic memory management), among others.
We present a visualization technique which shows hot spots in a variety of value domains. By re-numbering values according to the order in which they first appear, hot spots tend to show as a small number of horizontal, mostly black rectangles. The vertical size corresponds to the size of the hot spot, and the horizontal size corresponds to its duration.

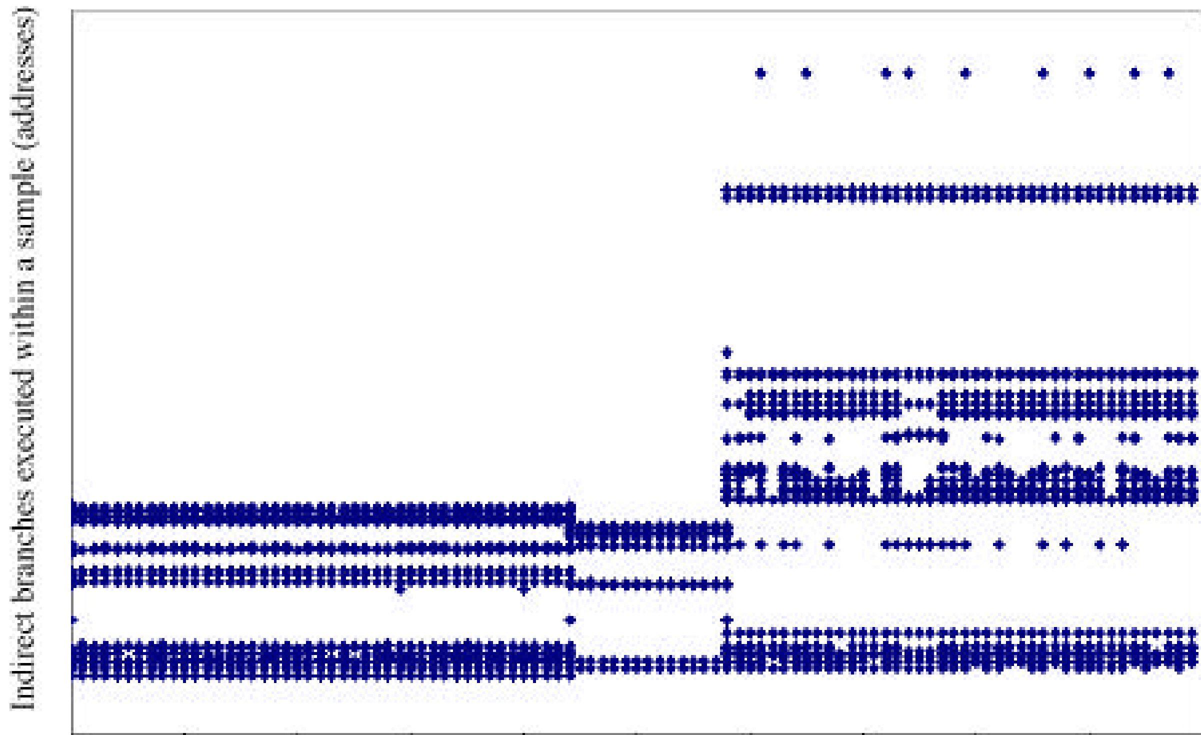
1. Methodology

For every tested domain, we obtained a trace of values. For example, the IXX-Indirect trace consists of the address of indirect branches executed in IXX, an IDL parser written in C++. The unit of time is an indirect branch execution, and hot spots are measured in samples of 20000 indirect branch executions. In Figure 1, the Y-axis shows the *address* of the indirect branches executed within each sample. This hot spot visualization is similar to that used by Merten et. al. [1].

In Figure 2, the Y-axis also shows branches, but instead of using branch addresses, they are re-numbered in order of appearance. In other words, the first executed indirect branch gets number 1, the second number 2, and so on. Every branch is looked up in a table that maps the branch address to this number. If a branch is absent from the table, it receives the next number and is inserted in the table. We called the re-numbering component which performs this operation an *id-dispenser*, since it is similar to the apparel that produces tickets with increasing numbers in waiting rooms. A person can leave the room and come back with the same number, but a new person entering the room receives a new ticket, incremented by one. This re-numbering scheme has several benefits for hot spot visualization, as shown in Figure 2:

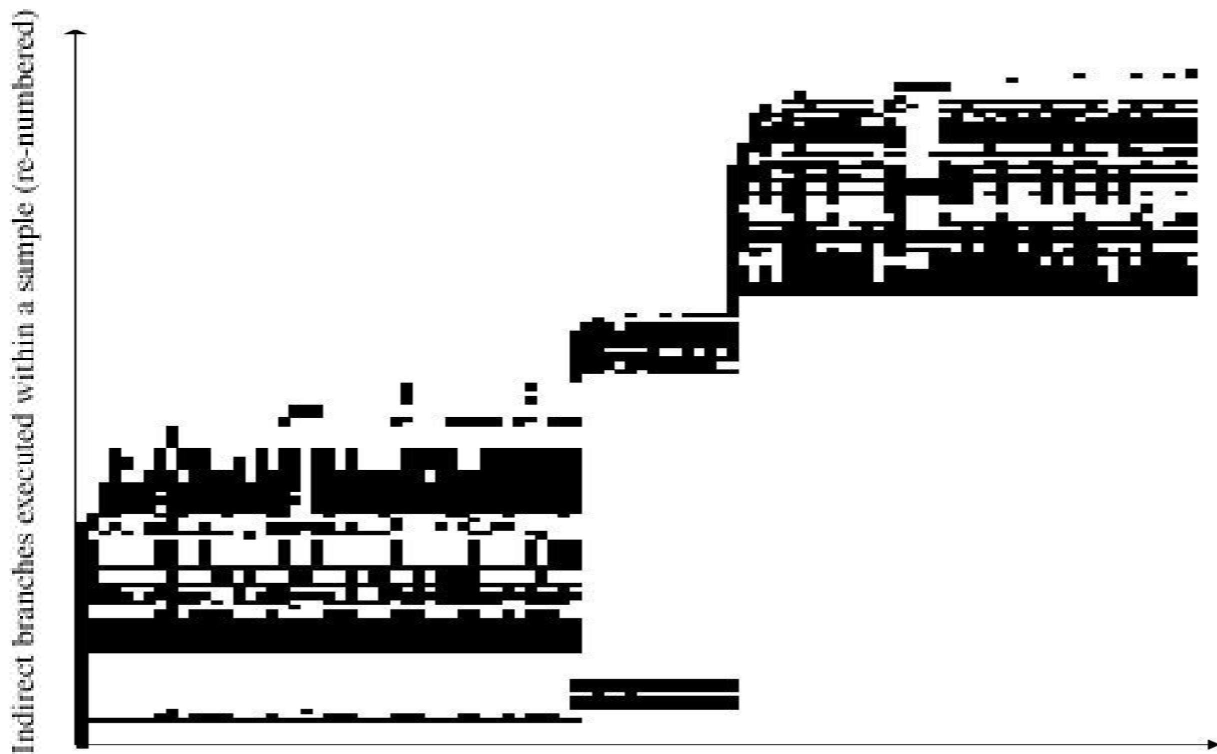
- branches appear in the sequence they are executed the first time, and this typically clusters hot spots together, making different program phases more visible.

---
[1] Other miss causes, such as cold start misses and conflict misses can still occur.



Indirect branch executions in 100 samples of 20000 executions each

Figure 1. IXX-indirect-raw: indirect branch trace of *ixx*, using branch addresses on the Y-axis.



Indirect branch executions in 100 samples of 20000 executions each

Figure 2. IXX-indirect: indirect branch trace of *ixx*, re-numbering branches in order of appearance

- space on the Y-axis is better used, since every number has at least one sample in which it appears (there are no blank horizontals). In contrast, real branch addresses leave wide unused horizontal gaps in the graphs.
- the vertical size of a hotspot is proportional to its actual size. In contrast, non-re-numbered branch addresses often seem to overlap, which makes the visual impression of a hot spot's size unreliable. For example in Figure 1, the first hot spot seems smaller than the last hot spot, while Figure 2 shows that is actually larger.
- 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.

The only disadvantage of re-numbering is that placement information is lost. However, the location of a hot spot is less important, at least for visualization purposes, than its size.

2. Visualizations

For every tested domain, we obtained a trace of values. Table 1 gives information about all traces and shows the units employed.

Name	Description	X-axis (time)	Y-axis values (location)
IXX-Indirect-raw	Indirect branches executed by ixx, an IDL parser written in C++ from OOCSB98 benchmark suite [2]	Indirect branch executions in samples of 20000	Addresses of indirect branches executed in sample
IXX-Indirect	Indirect branches executed by ixx, an IDL parser written in C++ from OOCSB98 benchmark suite [2]	Indirect branch executions in samples of 20000	Re-numbered indirect branches executed in sample
JAVAC-Methods	Methods executed in first 2M byte codes by JavaC compiler from SPECJVM98 benchmark suite [3]	Byte code executions in samples of 20000	Re-numbered methods touched in sample
JAVAC-Load/ Stores	Memory addresses touched in first 2M byte codes by JavaC compiler	Byte code executions in samples of 20000	Re-numbered memory locations loaded or stored in sample
SOOT-Allocations	Object allocation sites in the Soot compiler, measured using the Java Virtual Machine Profiling Interface	Bytes allocated[a] in samples of 1000 bytes	Re-numbered allocation sites responsible for allocated bytes in sample

Table 1. Visualizations and trace definitions.

[a]This is a measure of time typically used in garbage collection studies.

The data in this study was collected in an graduate course on run-time support for object-oriented systems at McGill. The graphs were generated using the Plumber 3.04 framework developed in the Adaptive Computation Lab (www.cs.mcgill.ca/acl). We also are also in the process of using the same technique to visualize URL hot spots (useful for server caching) and DNA hot spots (a string of a particular length serves as the Y-axis value unit).

3. Visualizations per domain

Figure 3 shows method hot spots in the execution of JavaC. The hot spots are more spread out than in Figure 2, but it is still easy to recognize different phases in the program, and long lasting hot spots such as

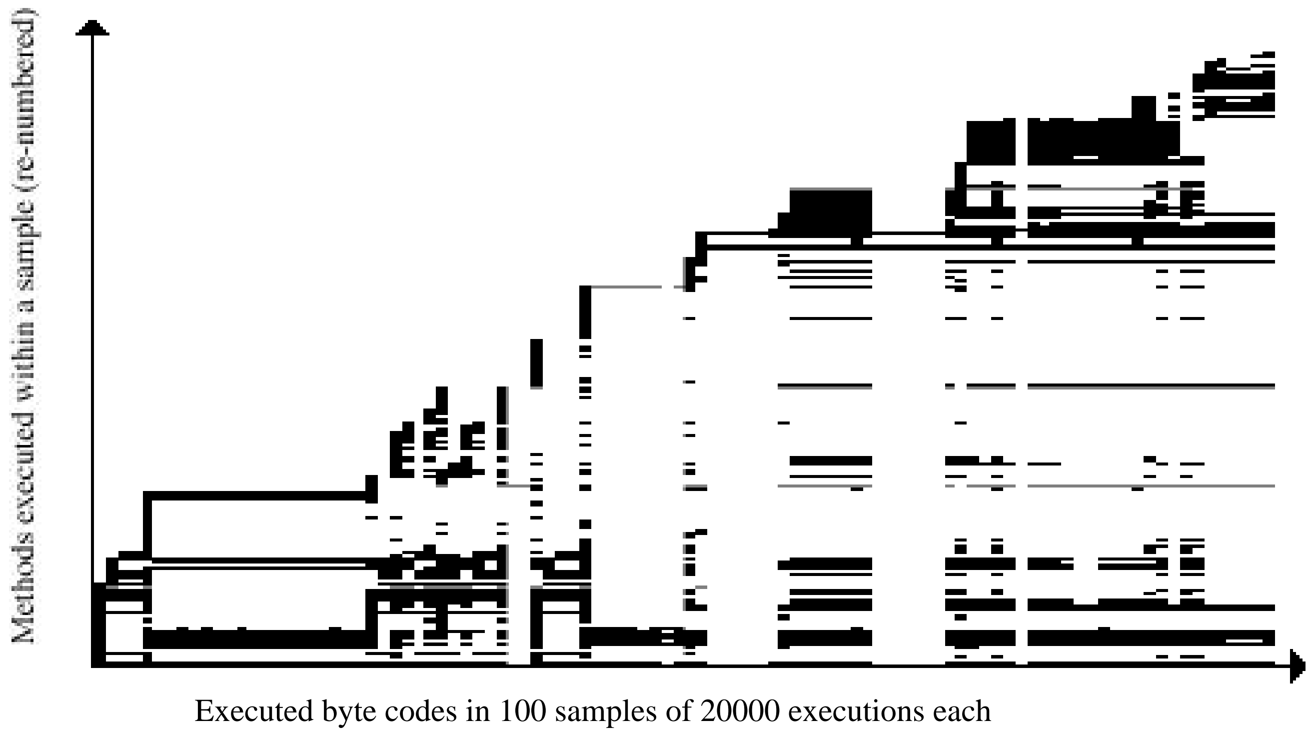


Figure 3. JAVAC-Methods: Methods executed in first 2M byte codes of JavaC

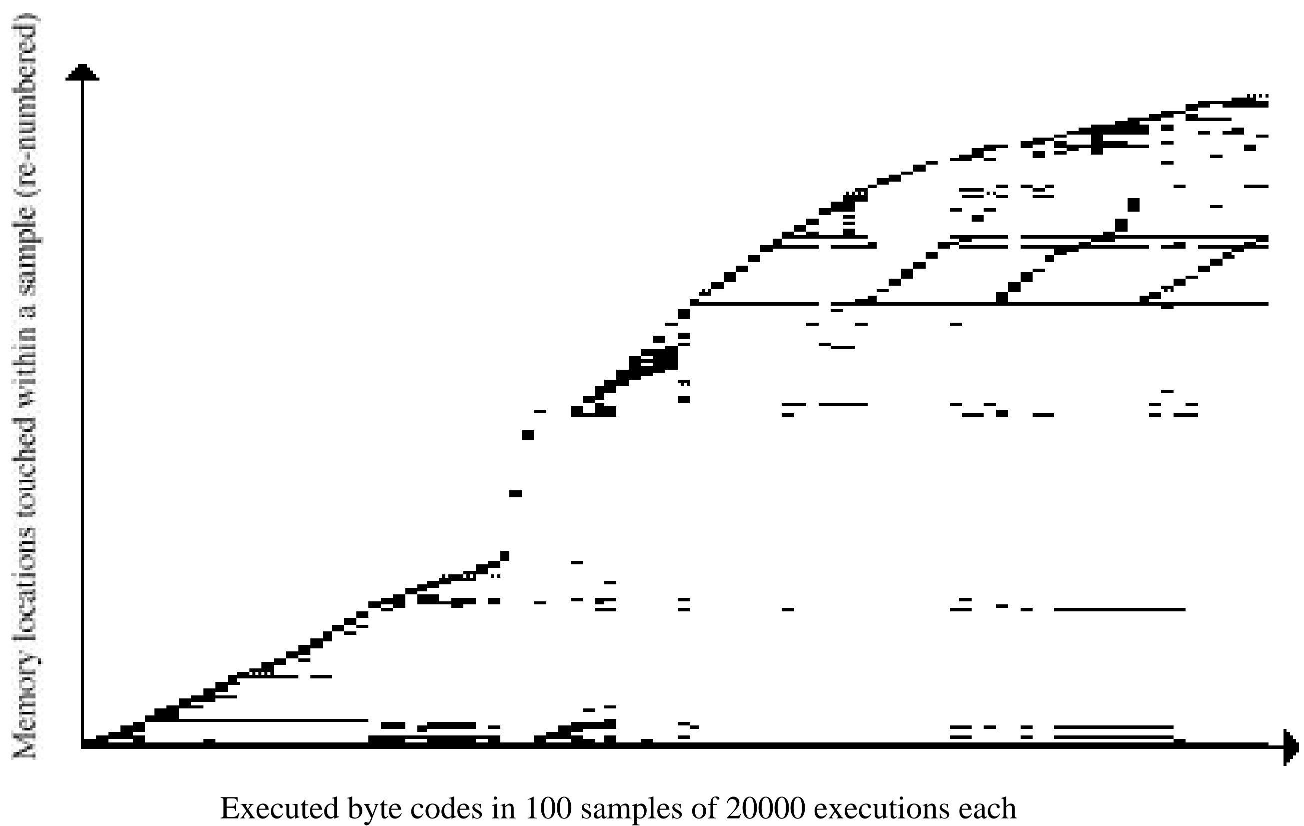
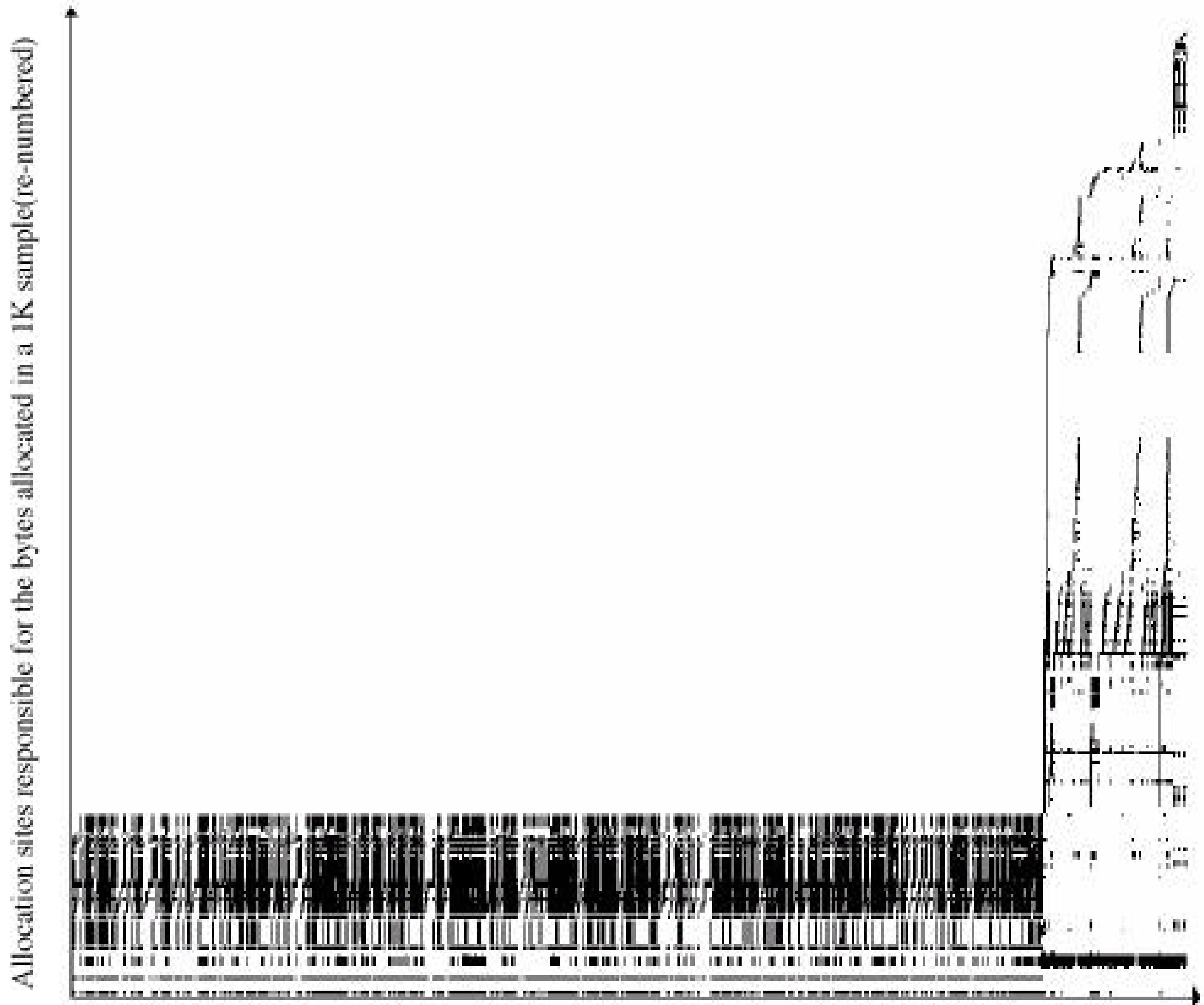


Figure 4. JAVAC-Load/Stores: memory locations touched in first 2M byte codes of JavaC

the lower horizontal bars. Figure 4 shows hot spots in memory addresses touched by load or stores in the same program. Different program phases also appear, but less pronounced than in Figure 3. Hot spots seem to be small (thin rectangles), but the program seems to touch large parts of the memory only once. In the second half of the trace, repeated iterations over the same memory locations appear as slanted lines.



Bytes allocated during entire program run, in samples of 1K each

Figure 5. SOOT-Allocations: allocation sites responsible for allocation in Soot.

Finally, in Figure 5, we trace new object allocation in Soot, a large object-oriented byte code to byte code compiler developed by the Sable group at McGill [4]. In this example, time is measured by the amount of memory allocated, as is typical in memory management studies. The Y-axis shows allocation sites. The meaning of a hot spot (a long flat rectangle) in this case corresponds to a small number of allocation sites that are responsible for a large amount of allocated memory. This enables a developer interested in memory optimization to focus on those instructions which consume the largest amount of memory.

4. Conclusions and future work

We have demonstrated a visualization technique aimed at detecting hot spots which can be applied in a variety of domains. Re-numbering of values by order of appearance tends to emphasize hot spots and allows a more accurate impression of a hot spot's size. Re-numbering also makes visualization independent of the actual encoding of values, which is typically platform-dependent.

We plan to apply hot spot visualization in other domains and to increase its effectiveness by careful use of color.

5. References

[1] 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.

[2] Karel Driesen and Urs Hölzle. Accurate Indirect Branch Prediction. *ISCA '98 Conference Proceedings*, pp. 167-178, Barcelona, July 1998.

[3] Karel Driesen, Patrick Lam, Jerome Miecznikowski, Feng Qian, Derek Rayside.
On the Predictability of Java Byte Codes.
Poster at OOPSLA 2000, October 2000
(also at www.cs.mcgill.ca/acl).

[4] URL: http://www.sable.mcgill.ca/soot/.