Heavy Tails in Program Structure

Hiroshi Sasaki Fang-Hsiang Su Teruo Tanimoto† Simha Sethumadhavan

Department of Computer Science, Columbia University
{sasaki,mikefhsu,teruo,simha}@cs.columbia.edu

Abstract—Designing and optimizing computer systems require deep understanding of the underlying system behavior. Historically many important observations that led to the development of essential hardware and software optimizations were driven by empirical observations about program behavior. In this paper, we report an interesting property of program structures by viewing dynamic program execution as a changing network. By analyzing the communication network created as a result of dynamic program execution, we find that communication patterns follow heavy-tailed distributions. In other words, a few instructions have consumers that are orders of magnitude larger than most instructions in a program. Surprisingly, these heavy-tailed distributions follow the iconic power law previously seen in man-made and natural networks. We provide empirical measurements based on the SPEC CPU2006 benchmarks to validate our findings as well as perform semantic analysis of the source code to reveal the causes of such behavior.

Index Terms—Program characterization, statistical distribution, empirical studies.

1 Introduction

Understanding programs is fundamental to enhancing computer performance. Empirical observations about programs such as the "90-10" rule, presence of spatial and temporal locality and biased branches have influenced designs of computer systems. In this paper, we report a new way to look at dynamic program execution that reveals a surprising and undiscovered property.

We view dynamic program execution as a changing network where the nodes of the network are static instructions in a program and a directed edge appears whenever one node produces a value for another. With this network view we observe that communication among static instructions in a program follows a heavy-tailed distribution: a small number of static instructions (network nodes) in a program communicate with a large number of instructions, while most static instructions communicate only with a few instructions. Further, in the programs we study, we observe that a large fraction of the heavy-tailed distributions follow the power law distribution; some that do not obey power law follow the closely related lognormal distribution.

What is the importance of this observation? Broadly speaking, a heavy-tailed distribution indicates the presence of events that rarely happen but dominate the process described by the distribution [9]. For instance, heavy-tailed latency distributions have been shown to impact the system performance of the whole data center [3], and generated interest in "tail-tolerant" techniques. Similarly our observation about the presence of heavy tails in dynamic program structures may help us not only enhance the science of computer architecture but also open up opportunities for new types of computer architectures. In this paper we provide empirical measurements to support the occurrences of power laws in dynamic program structures and an examination of programs to understand why heavy-tailed distributions occur.

†*Teruo Tanimoto is with Kyushu University. This work was done while he was a visiting student at Columbia University.*

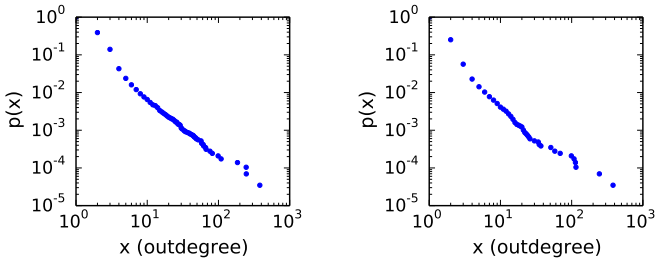
2 Programs as Information Flow Networks

Since program execution is essentially a collection of data communications and computations performed through instructions, we can view them as an information flow network. Once we view a program as a network, we can perform link analysis. For instance, we can use centrality measures to quantify the relative importances of each node in the network (analogous to the importance of a person within a social network), or determine how the communication volume is distributed across the nodes.

In our case, each vertex (or node) is a static instruction at x86-64 ISA level*, and each edge represents either control or data dependency. We define a control dependence as a connection between a source instruction and its immediate successive instruction. A data dependence is defined as a producer-consumer relationship through registers or memory. Specifically, when a source instruction writes to register(s) and/or memory, and the following destination instruction reads the value before it gets overwritten.

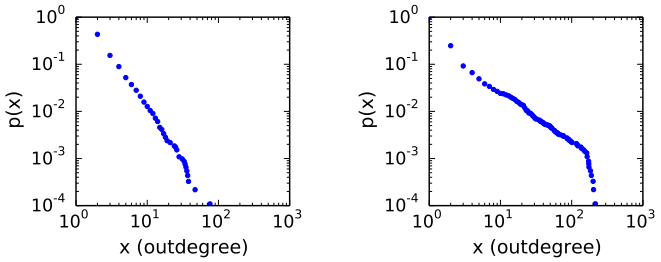
While many analyses can be performed on such networks, in this paper we focus on outdegrees of instructions to understand how instructions influence the whole program execution. Specifically, we study whether the distribution of the network fits the power law. A distribution obeys a power law if it is drawn from a probability distribution $p(x) \propto x^{-\alpha}$. A power law distribution has two parameters: the scaling factor α and the minimum value x_{min}. The α controls how sharply the probability decreases, and the x_{min} decides where the heavy tail of the distribution begins.

* Our analysis can be applied to other ISAs (e.g., ARM, MIPS etc) and low level machine instructions (e.g., micro-operations or µops).



(a) Register data dependency. (b) Memory data dependency.

Fig. 1: Outdegree-based log-log plot of the CCDF for `cactusADM`'s communication networks. Both register and memory networks follow the power law.



(a) Register data dependency. (b) Memory data dependency.

Fig. 2: Outdegree-based log-log plot of the CCDF for `sjeng`'s communication networks. Both register and memory networks do not follow the power law ($p < 0.1$).

3 Methodology

We use the SPEC CPU2006 benchmark suite [6] with `test` inputs for our analysis. All benchmarks are compiled by GCC 4.6.3 with `-O2` optimization flag. We dynamically construct the network using Pin [8]. In order to understand the difference of communication characteristics between register and memory, we generate and analyze two sets of networks per benchmark: one having only register dependencies as data flow edges and the other having only memory dependencies as data flow edges. Also, in order to understand the pure program behavior, we construct the network using only the instructions within the program binary. In other words, all the nodes and edges that account for other images (e.g., shared library) are not recorded.

We apply a standard statistical testing procedure to verify if the degree distributions align with power laws [2]. This procedure has three steps: (1) estimate the parameters (α and x_{min}) of the power law model by the commonly used Hill estimator; (2) perform a goodness-of-fit statistical test to obtain a p-value; (3) compare the power law against other distributions* via a likelihood ratio test to see which distribution(s) is a better alternative than the power law.

Step 2 is a Monte Carlo procedure which synthesizes testing datasets by the estimated α and x_{min}. We fit each synthetic dataset to its *own* power law model and count what fraction of the time the model is a poorer fit (i.e., the model from our empirical dataset is a better fit). This fraction becomes our p-value. We use the p-value as a measure of the hypothesis we are trying to verify but not to confirm a null model we would like to reject. Hence higher values are better. We compare our empirical dataset with 5,000 synthetic datasets. If the p-value ≥ 0.1, we consider the power law is a plausible fit. Otherwise, we conduct step 3 to see which alternative distributions are better.

4 Results on Program Structure

Fig. 1 represents the complementary cumulative distribution function (CCDF) of `cactusADM` benchmark from the SPEC CPU2006 suite on doubly logarithmic axes, where x represents the outdegree and $p(x)$ represents the complementary cumulative probability. When we take the logarithm of the probability distribution $p(x)$ of the power law

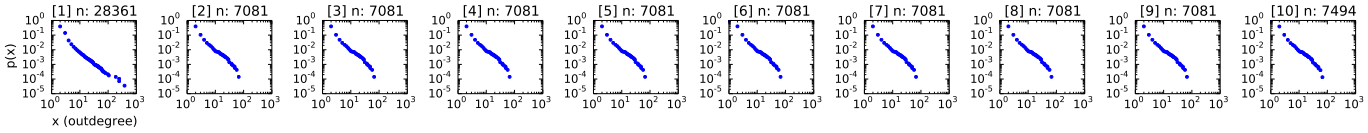
* Lognormal and exponential distributions. Interested readers are referred to the relevant papers [1, 2].

TABLE 1: Basic parameters of the outdegree distributions of the register and memory networks (n is the number of nodes, α is the scaling parameter and p is the p-value – statistically significant values are denoted in **bold**).

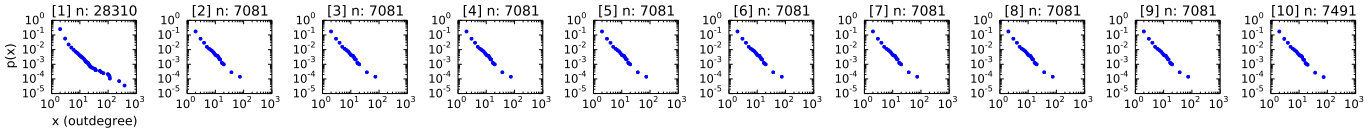
	Benchmark(input)	n	Register			Memory		
			α	x_{min}	p	α	x_{min}	p
Int	astar	5674	3.17	3	**0.90**	2.72	1	**0.28**
	bzip2(dryer.jpg)	7999	2.99	2	0.08	2.69	1	**0.43**
	bzip2(input.program)	7378	2.98	2	0.04	2.69	1	**0.43**
	gobmk(capture)	14757	2.98	2	**0.78**	2.64	1	0.00
	gobmk(connect)	19176	2.94	2	**0.13**	2.51	16	**0.22**
	gobmk(connection_rot)	16500	3.01	2	**0.84**	2.65	1	0.00
	gobmk(cutstone)	18423	2.93	2	**0.88**	2.67	1	0.00
	h264ref	27450	3.18	2	**0.95**	2.94	1	**0.68**
	hmmer	5627	3.49	2	**0.28**	2.92	3	**0.71**
	libquantum	3041	3.42	2	0.00	2.95	1	**0.26**
	mcf	1737	2.74	2	0.04	2.75	1	**0.70**
	omnetpp	17787	2.76	2	0.00	2.50	6	**0.82**
	perlbench(attrs)	43554	2.91	2	0.08	2.24	3	**0.27**
	perlbench(gv)	37151	2.91	2	0.06	2.21	4	**0.59**
	perlbench(makerand)	15619	3.32	4	**0.18**	2.45	3	**0.61**
	perlbench(pack)	60844	2.52	10	**0.48**	2.27	10	**0.32**
	perlbench(redef)	25552	2.99	2	0.05	2.37	3	**0.71**
	perlbench(ref)	30492	2.93	2	**0.12**	2.24	4	0.09
	perlbench(regmesg)	41263	2.94	2	0.07	2.25	4	**0.73**
	sjeng	9213	3.03	2	0.05	2.52	1	0.03
	xalancbmk	113373	2.93	2	0.00	2.63	3	**0.65**
FP	bwaves	5043	3.71	2	**0.20**	2.68	1	**0.18**
	cactusADM	28795	2.50	8	**0.98**	2.65	5	**0.88**
	calculix	34803	3.41	3	0.00	2.93	1	**0.12**
	gromacs	23156	3.35	2	**0.11**	2.96	1	**0.29**
	leslie3d	15360	3.65	2	**0.61**	2.86	1	**1.00**
	milc	8359	2.93	2	0.08	2.90	1	**0.32**
	namd	16078	3.20	2	0.00	2.92	1	**0.81**
	povray	37128	3.01	2	0.01	2.57	3	**0.34**
	soplex	21687	3.73	4	0.03	2.89	1	**0.31**
	sphinx3	15000	3.37	2	**0.70**	2.90	4	**0.83**

($\ln p(x) = \alpha \ln x +$ constant), it implies that the distribution follows a straight line on a log-log plot. We can see from Figures 1(a) and 1(b) that the outdegrees of both register and memory networks have heavy-tailed distributions. This means that the majority of instructions have a small number of outdegrees (i.e., consumers) whereas a small fraction of instructions have a large number of outdegrees. Also, the shape of the curve tells us that it is clearly not a random distribution. Moreover it suggests that the distribution might follow the power law. In fact, by performing the analysis described in §3, we find that both the networks obey the power law (p-value of 0.98 and 0.88) with their parameters of $\{\alpha, x_{min}\}$ as $\{2.50, 8\}$ and $\{2.65, 5\}$, respectively.

Fig. 2 shows the result of `sjeng` which is a counter example where neither of the register and memory networks follows the power law. Both curves have convex shapes which is qualitatively different from the shapes seen in Fig. 1. We investigate these two observations in more detail in the next section.



(a) Register data dependency.



(b) Memory data dependency.

Fig. 3: Outdegree-based log-log plot of the CCDF divided into 10 time series for `cactusADM`'s communication networks.

TABLE 1 presents a summary of the results (α and x_{min}, along with the p-value) from the fitting of a power law to each of the network. Among 21 unique benchmarks we examined, only one benchmark (`sjeng`) does not follow power law for either register- or memory-based communication; further 18 of the 21 benchmarks follow power law for the memory-based network, while 12 of the 21 benchmarks do not follow power law for register-based networks. Because of the compiler's register allocation mechanism we would expect only short-lived communication; hence the presence of power law in register networks is surprising. For register networks that do not obey the power law, we find that a closely related heavy-tailed distribution, the lognormal distribution offers a better fit for six of them: `milc`, `namd`, `perlbench(attrs)`, `povray`, `sjeng` and `xalancbmk`.

5 Causes of Heavy Tails

To uncover the reasons that the SPEC benchmarks exhibit power law distribution, we conduct two preliminary analyses in this section.

5.1 Network Evolution

One possible hypothesis that explains a small number of high outdegree instructions in program structures is that few instructions account for initializing read-only and heavily used data. In order to investigate this hypothesis, we present a time series of the outdegree distribution of `cactusADM`. We divide the execution of the program into 10 epochs (each epoch has $\approx$ 900M dynamic instructions) and show the log-log plot of each CCDF $p(x)$ in Fig. 3. The subfigures represent consecutive execution epochs from left to right. On top of each figure, the number in the square bracket presents the i-th epoch along with n, the number of vertices in the network. The dependencies which cross the epoch boundary are removed from the network, which means that each subfigure contains only the producer-consumer relationships within the epoch.

We can see from Fig. 3(a) that the scale of the first epoch is different from the rest of the epochs. It has more number of instructions, and the maximum outdegree is an order of magnitude higher. Similar trend holds for memory data dependency as seen from Fig. 3(b). This supports our hypothesis that the high outdegree instructions are indeed executed in the first epoch, although what is more interesting is the fact that the other computation phases of the execution (i.e., epochs 2 to 10) still follow the power law. This might indicate that the power law distribution is a fundamental property in program structures. We leave further investigation for future work.

```
1 template <class eobj>
2 inline i32 largesolidarray <eobj>::add(const eobj& e)
3 {
4   ...
5   ep[elemqu] = e;
6   i32 elemcell = freecellholder.get();
7   dccellar.ep[elemcell] = elemqu;
8   rccellar.ep[elemqu] = elemcell;
9   // elemqu, a member of largesoliadarray is updated
10  elemqu++;
11
12  return elemcell;
13 }
```

(a) A writer method in the `largesolidarray` class.

```
1 void regmngobj::createregions(i32 region1)
2 {
3   ...
4   // rarp is a largesolidarray object
5   // elemqu is read to decide when to terminate the loop
6   for (i=0; i<rarp.elemqu; i++)
7     rarp[i]->flredefine=true;
8   ...
9   for (i=0; i<rarp.elemqu; i++)
10    rarp[i]->fillnum=regfillnum;
11 }
```

(b) A reader method in the `regmngobj` class.

Fig. 4: A writer-reader example in `astar`.

5.2 Code Patterns for Power Laws

We conduct a semantic analysis for two benchmarks, namely `astar` and `sjeng`. We select these two benchmarks because both the register and memory networks follow power law for `astar` while those for `sjeng` do not. Further, their small program sizes (Lines of Code) reduce the burden of understanding source code to perform the analysis.

Astar: the astar algorithm searches for the path with the minimum cost (distance) from a start node to an end node on a map. Given a start node, the algorithm iteratively finds the set of next intermediate nodes having the lowest cost to reach the end node.

`Astar` passes a small number of central data structures that hold the configuration data across the whole

```
1 void make(move_s moves[], int i) {
2   // update board for a white pawn's move
3   if (board[from] == wpawn) {
4     if (promoted) {
5       board[target] = promoted;
6       board[from] = npiece;
7       ...
8     }
9     if (ep) {
10      board[target] = wpawn;
11      board[from] = npiece;
12      ...
13    }
14    ...
15  }
16  // statements for other chess pieces follow
17 }
```

(a) Multiple writes in the `make` function.

```
1 void gen(move_s moves[]) {
2   for (...) { // for each chess piece on the board
3     // pieces holds the location of each chess piece
4     i = pieces[j];
5
6     switch (board[i]) {
7       case (wpawn): // white pawn
8         // check the destination square by reading board
9         if (board[from+12] == npiece) { ... }
10        ...
11      case (wknight): // white knight
12        ...
13  }
14 }
```

(b) Multiple reads in the `gen` function.

Fig. 5: A writer-reader example in `sjeng`.

application. Concretely, the information in few data structures such as `largesolidarray` in Fig. 4(a) is propagated through the whole benchmark. This object stores the information that is directly used in search such as the regions on the map. Such data structures are read by multiple functions once it is updated. We observe this type of producer-consumer communication in the memory network across multiple functions. Fig. 4(a) shows an example at line 10 where `elemqu`, a member of `largesolidarray` which contains its size, gets updated. Whenever a method traverses `largesolidarray`, this value is referenced. A reader example in `createregions` is shown at line 6 in Fig. 4(b).

The program structure responsible for high outdegree instructions for register network is different from that of memory network. A representative pattern in `astar` is that a base address of an object is written to a register and successive instructions access members of this object. A power law distribution results as the object size increases and the number of reads increases. Since the register values are typically short-lived, we believe that this is a common pattern that accounts for the existence of high outdegree instructions in register networks.

Sjeng: `sjeng` is a program that plays chess games. Similar to `astar`, `sjeng` also has a central data structure `board`, which is an integer array that represents a chessboard. However, there are two notable differences between `sjeng` and `astar`: (1) `board` is a global array and its elements can be accessed via direct addressing where the base address is known at compile time and (2) `board` continuously gets updated throughout program execution.

Because of the first difference, frequent accesses to `board` and its base address does not involve register access. This is the major reason why the register network does not even have a few instructions with very high outdegrees as shown in Fig. 2(a), as opposed to `astar`.

The second difference gives us a hint where the truncated shape we see in Fig. 2(b) comes from. The code in Fig. 5(a) shows an example of writers to `board`. The function `make` is called for every move of the chess piece. For each move, `board` is written by different instructions from different conditions, e.g., which chess piece to move, which player is making a move, etc. The code starting from line 5 and 10 in Fig. 5(a) is an example. These writers have high volumes of consumers because every time the algorithm searches for the best move, function `gen` loops through chess pieces on the chessboard where each loop involves the read of `board`. An example reader is shown at line 9 in Fig. 5(b). Because of these "too many" writers with high outdegrees, the memory network of `sjeng` does not follow the power law.

6 Related Work

Power laws have been observed in many man-made and naturally occurring phenomenon from a distribution of incomes to a distribution of word frequencies to a study of sizes of living things [9]. In computer systems, phenomena such as the structure of the Internet [5], the distribution of file sizes [4], and the distribution of latencies in a data center [3] have been modeled by heavy-tailed distributions. Software engineering researchers have observed power laws in references to Java classes or shared libraries in Unix distributions [7]. Our work is the first to report power laws in dynamic program structure, which is relevant to computer architects.

7 Conclusions

Meaningful and useful program characterizations are always beneficial to design efficient computer systems. With this in mind, in this paper we characterized program structures with the help of a new way of viewing dynamic program execution. We modeled the communication between instructions as information flow networks, and analyzed the distribution of its outdegrees. We have found that the communication networks of the SPEC CPU2006 benchmarks we evaluated follow the heavy-tailed distribution, where the majority of them follow the power law distribution.

Acknowledgements

This work is sponsored in part by JSPS Postdoctoral Fellowships for Research Abroad, NSF award number 1302269 and Alfred P. Sloan Fellowship.

References

[1] J. Alstott, E. Bullmore, and D. Plenz, "Powerlaw: a Python package for analysis of heavy-tailed distributions," *PloS one*, vol. 9, no. 1, Jan. 2014.

[2] A. Clauset, C. R. Shalizi, and M. Newman, "Power-law distributions in empirical data," *SIAM review*, vol. 51, no. 4, pp. 661–703, Nov. 2009.

[3] J. Dean and L. A. Barroso, "The tail at scale," *CACM*, vol. 56, no. 2, pp. 74–80, 2013.
[4] A. B. Downey, "The structural cause of file size distributions," in *MASCOTS '01*, 2001, pp. 361–370.
[5] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *SIGCOMM '99*, Aug. 1999, pp. 251–262.
[6] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM CAN*, vol. 34, no. 4, pp. 1–17, Sep. 2006.
[7] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM TOSEM*, vol. 18, no. 1, pp. 2:1–2:26, 2008.
[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05*, Jun. 2005, pp. 190–200.
[9] M. Mitzenmacher, "A brief history of generative models for power law and lognormal distributions," *Internet Mathematics*, vol. 1, no. 2, pp. 226–251, 2003.