Short Paper: Improving the Responsiveness of Internet Services with Automatic Cache Placement

Alexander Rasmussen
University of California San Diego
arasmuss@cs.ucsd.edu

Emre Kıcıman Benjamin Livshits
Madanlal Musuvathi
Microsoft Research
{emrek,livshits,madanm}@microsoft.com

Abstract

The backends of today's Internet services rely heavily on caching at various layers both to provide faster service to common requests and to reduce load on back-end components. Cache placement is especially challenging given the diversity of workloads handled by widely deployed Internet services. This paper presents FLUXO, an analysis technique that automatically optimizes cache placement. Our experiments have shown that near-optimal cache placements vary significantly based on input distribution.

Categories and Subject Descriptors H.3.5 [*Online Information Services*]: Web-based services; D.4.2 [*Storage Management*]: Distributed memories; D.4.7 [*Operating Systems*]: Distributed systems; D.3.2 [*Language Classifications*]: Data-flow languages

General Terms Algorithms, Design, Experimentation, Languages, Management, Performance

Keywords Caching, simulation, optimization, dataflow model, Internet services, cloud computing

1. Introduction

The backends of today's Internet services rely heavily on caching at various layers both to provide faster service to common requests and to reduce load on back-end components. In the context of a large-scale Internet service, a cache bypasses the computation and/or I/O performed by one or more components or tiers of the system. For best performance, cache contents are usually stored in memory, making them an expensive and scarce resource.

To receive the optimal benefit, a cache must be placed with careful consideration of incoming workload distributions, cached data sizes, consistency requirements, component performance, and many other issues. Unfortunately, many of these are cross-cutting concerns, and fully understanding the implications of a particular cache placement requires a deep understanding of the entire system.

Today's deployed systems largely rely on developer's intuition accompanied by localized profiling to select between cache placements [Henderson 2008]. For instance, a service architect might use cross-tier caching to alleviate a locally observed performance bottleneck. However, he or she might not realize the implications of this local decision on the remainder of the system. Making matters more difficult, both the service itself and the input workload are subject to relatively frequent changes. In our own experiments, shown in Section 4, we see that the most effective cache placement varies significantly depending on input workload.

This paper presents FLUXO, an analysis technique that automatically optimizes cache placement:

- FLUXO uses a *dataflow model* of an Internet service's coarse-grained behavior and performance to abstract our reasoning about cache placement from the specific details of the online service's implementation. In recent years, we have seen the emergence of a variety of dataflow-based programming models for large-scale data analysis and service composition [Dean 2008, Isard 2007, Yahoo!, Inc. 2008, Microsoft 2008, Kohler 2000].
- FLUXO uses *runtime request tracing* of an Internet service to capture performance characteristics as well as input and intermediate data distributions. Request tracing has been used for many analysis techniques in Internet services, but especially for fault diagnosis [Reynolds 2006, Chen 2002, Szeredi 2005] and performance modeling [Abd-El-Malek 2005, Barham 2004].
- FLUXO uses a *simulation system* combined with a hill-climbing optimization algorithm to rapidly experiment with a large space of possibilities and converge on reasonably good cache placement decisions.

EuroSys'09, April 1–3, 2009, Nuremberg, Germany.

2. The Cache Placement Problem

The problem of placing caches in an Internet service can be viewed abstractly as an optimization problem: a programmer must choose the locations and sizes of caches in the most beneficial way possible and is constrained by the total amount of resources available for caches.

2.1 Assumptions and Goals

FLUXO's primary goal is to minimize overall request latency by allocating the cache space provided by a cluster of caching servers. We assume that services are deployed within a data center and that all back-end components are within a single administrative domain. This is in contrast to a large body of previous work on general, cross-enterprise web service composition [Rao 2005]. While we do not believe this is a fundamental limitation, issues of wide-area network latencies, security, accounting, etc. are outside the scope of this short paper. Further, we assume that a service runs on a single machine and that its components make calls to external services as part of their computation. At larger scales, we speculate that FLUXO could operate in an analogous way using whole cache servers as its unit of allocation, but we do not evaluate this case in this paper. We also do not consider the effects of interaction between our services' caches and caches that may exist in front of external services.

2.2 Service Representation

To simplify our analysis of the service, we use a dataflow model (or *service graph*) to represent its coarse-grained request handling behavior. An incoming request is modeled as data that enters the service graph via a SOURCE node and travels through the service graph until exiting at a SINK node. All other nodes in the service graph are *application components*, which are allowed to access external services such as local databases, platform services, or remote Web services. An example service, whose purpose is to produce a weather report given an IP address and/or a zip code within the United States, is shown in Figure 1. In this service, if an IP address is provided, it is converted by an external service into the city name where that IP address is located. A second external service then converts that city name into a weather report for that city. If a zip code is provided, that zip code is passed to an external service which resolves it directly into the weather report for that zip code. In the event that both the IP address and zip code are provided, both of these chains of execution are processed simultaneously and the result of the first chain to complete is returned.

In this representation, computation performed by a node can be cached by inserting a *cache* at the incoming edge of the node. The cache stores the results of the computation for a subset of inputs received by the node in the past. On a hit, the cache forwards the results to the outgoing edge of the node. On a miss, the cache forwards the input to the node that then performs the computation. The cache then updates its state based on a prescribed replacement policy before forwarding the result. This process of inserting a node can be generalized to a subgraph of the dataflow graph. An example of this is shown in Figure 2. As the figure shows, the cache node that is inserted has exactly one incoming and one outgoing edge. FLUXO can be extended to support caches with multiple incoming and outgoing edges, but this extension is the subject of future work. If a subgraph has multiple incoming or outgoing edges, we choose the two edges that comprise the beginning and end of a path through all the nodes in the subgraph. For example, in Figure 2, the edge from *C* to *D* is a valid outgoing edge, but not the edge from *B* to *E*.

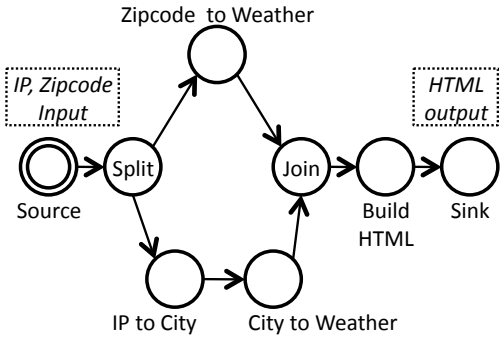

Figure 1: The service graph for a weather service application, used in our case study. The component labeled "split" copies its input to its outputs, while the component labeled "join" accepts and forwards whichever input it receives first.

A cache will only accept one update per user request to prevent other incoming edges to a cached subgraph from incorrectly triggering a cache update. If multiple caches share an incoming edge, the caches are checked in decreasing order by subgraph size and when a larger cache hits, all smaller caches are skipped. Care must be taken to ensure that the insertion of a cache does not compromise the service's ability to complete when a cache hit occurs. We omit a discussion of the intricacies due to space.

2.3 Potential Cache Locations

In all but the most simple of systems, several interrelated concerns shape the decision of where to place caches and how large to make them:

Component properties: The first concern in cache placement is *component performance*. Good cache placements should bypass one or more slow components in a system to mitigate their latencies. Also, cache placements must avoid violating *semantic constraints*, such as bypassing components that have side-effects or inappropriately caching data with strong consistency guarantees.

Workload properties: The workload distribution largely determines the *cache hit rate*. The more repetitive the workload, the more effective the cache. Another key aspect of the

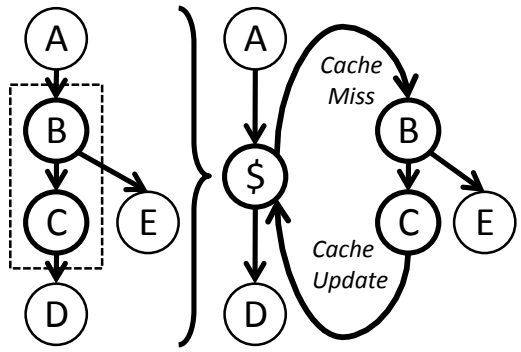

Figure 2: An example dataflow transformation that caches the computation performed by components B and C.

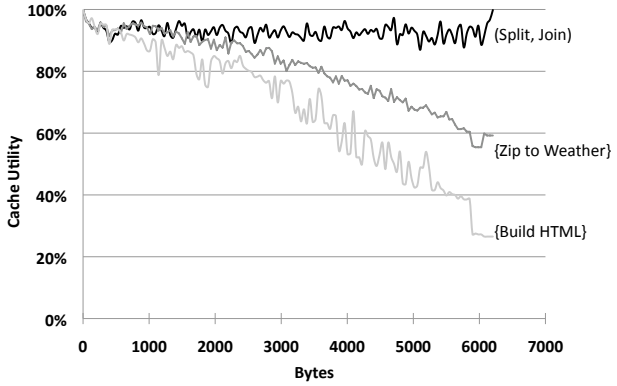

Figure 3: This graph show how the cache utility changes as we modify the caching policy suggested by FLUXO. The lines represent capacity reallocation from the cache for the entire graph ("All-Encompassing") to 3 other potential cache locations. (Split, Join) is the cache for the subgraph containing the nodes {Split, IP To City, City To Weather, Zip To Weather, Join}. The other two lines are caches for the individual nodes "Zip To Weather" and "Build HTML" respectively.

workload is the *data size* of the objects being cached, which affects how many objects can fit into a cache of a given size.

Interdependencies: Further complicating the matter of cache placement are the dependencies between the components and workloads of a service. When multiple caches are being added to a service graph, one cache can often reduce the effectiveness of the other caches. For example, each of the parallel branches in the service shown in Figure 1 can execute independently and the "Join" component will forward data from the branch that produces its output first. If its hit rate is high, a cache for one branch can substantially decrease the need for a cache for the other branch, since the cache would frequently produce output before the other branch. Similarly, adding a cache for the subgraph consisting of "IP to City" and "City to Weather" may significantly change the workload distribution (and cache-ability) of the individual "City to Weather" component.

2.4 Complexity and Scale of State-Space

Analytically modeling the space of potential cache locations is a challenging proposition. Given T units of memory capacity, a *caching policy* involves assigning these units to potential cache locations in the service graph. Given n such locations, there are

$$\binom{T+n-1}{n-1}$$

different caching policies. Explicitly enumerating this exponential space to determine the *optimal* caching policy is a daunting task. Even for the simple graph in Figure 1 with 7 possible cache locations, distributing 100 units of capacity requires searching over a billion policies. Accordingly, unless more efficient analytical methods are found, finding an approximate solution is the only feasible possibility. In FLUXO, we experiment with a simulation based approach that uses a hill-climbing algorithm to search the state space for reasonably good caching policies.

To reduce the size of our search space, we make the following observation. The difference between allocating B bytes to a cache and $B+1$ bytes to a cache is likely to be negligible unless the objects stored in the cache are very small. When allocating sizes to cache locations, FLUXO restricts the size of each cache to a multiple of the average size of data output by that cache's output component.

Figure 3 shows that the cache utility as a function of the caching policy is highly non-linear, a significant challenge to our hill-climbing algorithm. For our experiments, we determine the utility of a caching policy by the net savings in latency for a simulation of a given input stream. Starting from the policy suggested by FLUXO, Figure 3 shows the change in cache utility (y-axis) as the cache-capacity (x-axis) is reallocated from a single cache for the entire graph ("All-Encompassing") to 3 other potential cache locations. The figure reports the utility as a percentage of the utility of the suggested policy. In our experiments, the service in Figure 1 processes tens of bytes for each input request. We expect realistic Internet services to process many orders of magnitude larger amount of data. Accordingly, the unit of reallocation (x-axis) used in the simulation of such services will be larger.

3. Design and Implementation

FLUXO consists of two main parts, the FLUXO runtime and the FLUXO optimizer. Although the analysis techniques presented in this paper can apply to any sufficiently decoupled and instrumented service, our current tracing and simulation tools assume that services have been written for the FLUXO runtime. The FLUXO runtime is responsible for running services, and is invoked with a description of the service to run and an optional *caching policy* that states the size and position of the service's caches. As the service receives and processes a stream of user requests, the FLUXO runtime pro-

duces a stream of runtime observations or *events*. This ordered stream of events is then processed periodically by the FLUXO optimizer. The optimizer's responsibility is to generate a new and effective caching policy based on the observed event stream. FLUXO does not attempt real-time migration of system configurations, though others have achieved such migrations in other contexts [Anderson 2002].

The FLUXO runtime is aware of the service structure and has the ability to rewire the service graph to insert caches in accordance with the caching policy, as discussed in detail in Section 2. The optimization step may be repeated with arbitrary frequency, limited only by computing power. Alternatively, caching policy regeneration can be triggered by a change to the service itself or the typical workload. For instance, if workloads differ significantly from week to week, caching policies may be generated on Sunday nights.

3.1 Service Simulator

The FLUXO optimizer contains a service simulator that simulates a given caching policy to determine its utility, since to our knowledge no analytic caching model has been developed that takes into account arbitrary cache interference and replacement policies in the context of data transformations. The list of events generated in response to a given user request constitute a *session*, and events are processed by the optimizer one session at a time. The time between the first event (data exiting the SOURCE node) and the last event (data entering the SINK node) in a session is that session's end-to-end latency.

To simulate a cache hit during a given session, the simulator temporarily adjusts the event stream for that session. Events that would not have occurred because of the cache hit are removed from consideration, and the times on all other events are adjusted to simulate the time savings produced by the cache hit. The simulator records the net decrease in end-to-end latency across all sessions and reports this number as the simulated caching policy's utility.

The FLUXO runtime and the FLUXO simulator utilize the same execution runtime and cache implementation, so we believe our simulations to be accurate. However, such simulation is time-consuming, and exploring tradeoffs between simulator accuracy and runtime remains future work.

3.2 Policy Generator

Examining the entire space of possible caching policies to determine the best one is infeasible for all but the smallest services. For realistic services, we apply hill-climbing search algorithms to identify reasonably good caching policies. A hill-climbing search starts from a random caching policy and iteratively improves the solution by choosing the next policy with best cache utility in the current neighborhood. This iteration proceeds until it finds a local optimum. If we pick a large number of points in the space of possible policies and perform a hill-climbing search from those points to local optima in the solution space, the largest local optimum policy so discovered is likely to be a good policy to recommend.

Input	Cache Locations	Allocation
Thin-Tail	{City to Weather}	65%
	{Split, IP to City}	31%
	{IP to City}	4%
Fat-Tail	{Split, IP to City, City to Weather, Zipcode to Weather, Build HTML}	62%
	{Split, IP to City}	22%
	{IP to City}	9%
Flat	{IP to City, City to Weather, Join, Build HTML}	13%
	{IP to City, City to Weather, Join}	13%
	{IP to City}	52%
	{Zipcode to Weather, Join}	13%

Figure 4: Cache placements recommended by FLUXO for the case study's input distributions. Each cache location is described by the subgraph of the service shown in Figure 1 over which it applies. Caches in this list with the same input edge are checked from top to bottom.

During hill-climbing, successor states are generated by the following method: for each pair of cache locations for which it is possible, modify the original state by adding the mean observed output data size in bytes to the first cache's allocation and removing that same amount of bytes from the second cache's allocation. Each such modification generates a new successor state.

The more points that are examined by the above method, the more likely it is that a close-to-optimal solution will be discovered. Most of the time spent in running the policy generator is spent evaluating successor states during the hill-climbing search, since each evaluation requires a complete simulation. If hill-climbing searches are started from very low points in the space, a great deal of time will be wasted climbing toward what are likely poor policies. As an optimization, we evaluate the benefit of a large number of points and then hill-climb from the top k of those points.

4. Case Study

To evaluate FLUXO in a repeatable manner, we evaluate its performance on a simple service whose components are linked to simulated external services. To make experimentation easier, we use a relatively small service with appropriately scaled-down workload and cache capacity. In our implementation, we analyze 20,000 random points in the space of possible cache placements and choose the top 200 points for hill-climbing. We estimate the performance of a point through a simulated execution of 2700 requests, which requires approximately 50 ms per point. The total amount of memory available for caching in this simulation is 10 KB, a size chosen so that the average size of a cache could hold around 50 average-sized intermediate data items. We neglect cache warm-up effects since an average-sized cache warms up within the first 200 requests. The total execution time

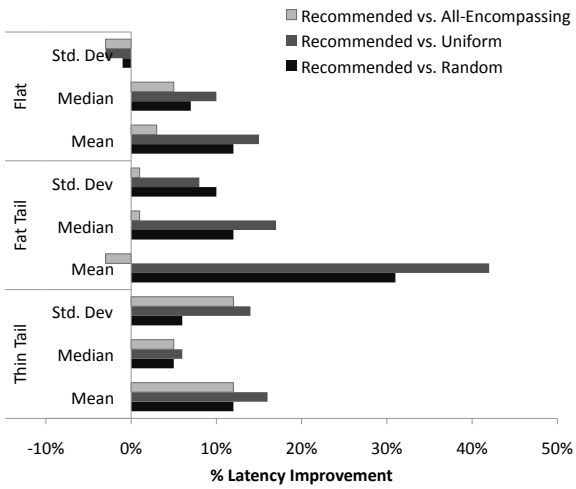

Figure 5: Latency improvements with different cache placement strategies for different input distributions.

of our analysis, including hill-climbing, is approximately 90 minutes. Unfortunately, this analysis does not scale with the size of the application, and we plan to explore the use of faster analytical methods in future work.

The request stream given to the optimizer was also used to evaluate the service's performance, so the performance numbers given here are optimistic and are meant to convey the best-case performance of FLUXO's recommendations. Future evaluation will use separate request streams for training and evaluation.

We consider the following three input distributions in our experiments: **Thin-Tail**, the original, heavy-tailed distribution of IP addresses observed in our production web service logs; **Fat-Tail**, the thin-tail distribution, modified so that no input is queried fewer than five times; and **Flat**, querying each input an equal number of times.

We compare the choice made by the FLUXO optimizer with the following naïve policies: **Random**, cache bytes are distributed randomly across all valid cache locations; **Uniform**, cache bytes are distributed evenly across all valid cache locations; and **All-encompassing**, a single cache is created from SOURCE node to SINK node.

4.1 Experimental Setup

Initially, the weather service was backed by actual external services. We found that these services could not support the query volume required for a thorough evaluation, and so we endeavored to simulate these services and to approximate their functionality as closely as possible.

The simulated external services are backed by an IP and zip code geolocation database; responses to database queries are delayed by a random period of time. The sample from which these random time periods is drawn is given by a Pareto distribution[Newman 2005], a power-law probability distribution, chosen because it closely matched the observed distribution of actual service response times. The parameterization of the distribution varies depending on the external service being queried. These parameters were estimated by recording request latencies for each external service across several hundred queries.

All inputs given to the weather service are IP address-zip code pairs so that both branches of the service graph can be exercised. The input set for each experiment consists of a set of such pairs. Each unique input pair p is queried r_p times, where r_p is determined by one of our input distributions.

4.2 Experimental Results

For each of the workload distributions, FLUXO identifies cache locations in the service graph that provide the highest cache hit rate and performance benefit. As shown in Figure 4, the chosen cache locations vary greatly based on the type of input distribution. In the thin-tail case, the biggest cache is placed around the City to Weather node because of its higher hit rate than in the case of IP address-city mapping. In the fat-tail case, the cache hit rate of the all-encompassing cache is high enough to make caching the entire computation more effective than other strategies. In the flat case, half of the total cache space is allocated to the component with the highest latency along the most frequently taken path, which is IP Address to City. The rest of the space is allocated evenly among commonly-taken paths. The chart in Figure 5 shows the improvements in end-to-end latency achieved by using these recommendations compared to the naïve strategies. Note that no single policy is most appropriate for all input distributions, although the all-encompassing cache performs well in all cases. We speculate that an all-encompassing cache may be sub-optimal for services whose internal components have a higher degree of temporal locality than the service as a whole. Testing FLUXO on such services remains future work.

5. Related Work

Previous successful work in automated design includes storage system design for performance and reliability [Anderson 2002; 2005], recovery planning in storage systems [Keeton 2006], index and view selection in database systems [Agrawal 2000], and model-based resource provisioning in Internet services [Doyle 2003, Shivam 2005]. It is future work to incorporate many of these techniques into FLUXO, such as algorithms for efficiently exploring the configuration space [Anderson 2005], and improving the performance models of simulated components [Stewart 2008].

Other work has studied the problems of sizing and placement for document caches in wide-area information dissemination [Kelly 2001, Chu 2008, Wang 1999]. FLUXO, however, focuses on the automated design of caches for the internal computation and I/O of a system. The most significant difference is that FLUXO must analyze cache behaviors in

the context of data transformations, such as the transformation from IP addresses to zip codes to weather data, complicating the modeling of interference between caches.

6. Future Work

The challenges we are currently addressing include:

Applying FLUXO to imperative programs: The next step in making FLUXO more applicable to real-world services is to apply its analysis to dataflow models automatically extracted from coarse-grained traces of real-world services.

Scalability of analysis algorithms: Promising approaches to improving our analysis scalability include 1) machine learning algorithms that are more robust to searching discontinuous spaces; 2) a simple parallelized analysis to exploit multiple cores or clusters; and 3) caching of common graph and simulation results across the rounds of our analysis.

Other future work includes using FLUXO's analysis to optimize additional caching parameters, such as cache lifetimes and replacement policies; and applying FLUXO to adapt cache placement decisions on-line as workload changes.

7. Summary

While today, Internet service architects and developers perform cache placement based on intuition or rules of thumb, all the necessary data to fully automate this decision process is available through runtime tracing of workload distributions and component performance.

This paper presents FLUXO, a system to automatically place caches to improve the latency of large-scale online Internet services. Our results show that, using our dataflow-based system model and information obtained from runtime traces, an automatic system can be constructed that produces good caching policy recommendations. Challenges remain, however, in improving the scalability of FLUXO's analytical techniques and broadening their application.

References

[Abd-El-Malek 2005] M. Abd-El-Malek, II W. V. Courtright, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile Cluster-Based Storage. In *FAST*, 2005.

[Agrawal 2000] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, pages 496–505, 2000.

[Anderson 2002] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *FAST*, 2002.

[Anderson 2005] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly Finding Near-Optimal Storage Designs. *TOCS*, 23(4):337–374, 2005.

[Barham 2004] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modeling. In *OSDI*, pages 18–18, 2004.

[Chen 2002] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *DSN*, pages 595–604, 2002.

[Chu 2008] D. C. Chu and J. M. Hellerstein. Automating Rendezvous and Proxy Selection. Technical Report UCB/EECS-2008-84, University of California Berkeley, 2008.

[Dean 2008] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[Doyle 2003] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *USITS*, 2003.

[Henderson 2008] C. Henderson. Scalable Web Architectures: Common Patterns and Approaches, September 2008.

[Isard 2007] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, pages 59–72, 2007.

[Keeton 2006] K. Keeton, D. Beyer, E. Brau, A. Merchant, C. Santos, and A. Zhang. On the Road to Recovery: Restoring Data After Disasters. *SIGOPS OSR*, 40(4):235–248, 2006.

[Kelly 2001] T. Kelly and D. Reeves. Optimal Web Cache Sizing: Scalable Methods for Exact Solutions. *Computer Communications*, 24:163–173, 2001.

[Kohler 2000] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Frans Kaashoek. The Click modular router. *TOCS*, 18:263–297, 2000.

[Microsoft 2008] Microsoft. Microsoft PopFly, 2008. URL `http://www.popfly.com/`.

[Newman 2005] M.E.J. Newman. Power Laws, Pareto Distributions and Zipf's Law. *Contemporary Physics*, 46(5):323–351, 2005.

[Rao 2005] J. Rao and X. Su. *A Survey of Automated Web Service Composition Methods*, volume 3387 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg, 2005.

[Reynolds 2006] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, pages 9–9, 2006.

[Shivam 2005] P. Shivam, A. Iamnitchi, A. R. Yumerefendi, and J. S. Chase. Model-Driven Placement of Compute Tasks and Data in a Networked Utility. *International Conference on Autonomic Computing*, pages 344–345, 2005.

[Stewart 2008] C. Stewart, T. Kelly, A. Zhang, and K. Shen. A Dollar from 15 Cents: Cross-Platform Management for Internet Services. In *USENIX*, 2008.

[Szeredi 2005] M. Szeredi. Filesystem in USEr space. 2005. `http://fuse.sourceforge.net/`.

[Wang 1999] J. Wang. A Survey of Web Caching Schemes for the Internet. *SIGCOMM Computer Communication Review*, 29(5): 36–46, 1999.

[Yahoo!, Inc. 2008] Yahoo!, Inc. Yahoo! Pipes, 2008. URL `http://pipes.yahoo.com/pipes/`.