Optimal Web Cache Sizing: Scalable Methods for Exact Solutions[*]

Terence Kelly Daniel Reeves
{tpkelly,dreeves}@eecs.umich.edu
Electrical Engineering & Computer Science
University of Michigan
Ann Arbor, Michigan 48109 USA

April 29, 2000

Abstract

This paper describes two approaches to the problem of determining exact optimal storage capacity for Web caches based on expected workload and the monetary costs of memory and bandwidth.

The first approach considers memory/bandwidth tradeoffs in an idealized model. It assumes that workload consists of independent references drawn from a known distribution (e.g., Zipf) and caches employ a "Perfect LFU" removal policy. We derive conditions under which a shared higher-level "parent" cache serving several lower-level "child" caches is economically viable. We also characterize circumstances under which globally optimal storage capacities in such a hierarchy can be determined through a *decentralized* computation in which caches individually minimize local monetary expenditures.

The second approach is applicable if the workload at a single cache is represented by an explicit request sequence and the cache employs any one of a large family of removal policies that includes LRU. The miss costs associated with individual requests may be completely arbitrary, and the cost of cache storage need only be monotonic. We use an efficient single-pass simulation algorithm to compute aggregate miss cost as a function of cache size in $O(M \log M)$ time and $O(M)$ memory, where M is the number of requests in the workload. Because it allows us to compute arbitrarily-weighted hit rates at *all* cache sizes with modest computational resources, this algorithm permits us to measure cache performance with no loss of precision.

The same basic algorithm also permits us to compute *complete* stack distance transformations in $O(M \log N)$ time and $O(N)$ memory, where N is the number of unique items referenced. Experiments on very large reference streams show that our algorithm computes stack distances more quickly than several alternative approaches, demonstrating that it is a useful tool for measuring temporal locality in cache workloads.

1 Introduction

In the Internet server capacity planning literature, monetary cost is often regarded as the objective function in a constrained optimization problem:

> The purpose of capacity planning for Internet services is to enable deployment which supports transaction throughput targets while remaining within acceptable response time bounds and minimizing the total dollar cost of ownership of the host platform [33].

Web cache capacity planning must weigh the relative monetary costs of storage and cache misses to determine optimal cache size. As large-scale Web caching systems proliferate, the potential savings from making this tradeoff wisely increase. Calculating precisely the optimal size of an isolated cache might not be worth the bother, but deployments on the scale of Akamai and WebTV raise the stakes to the point where careful calculation is essential. While the monetary costs and benefits of caching do not figure prominently in the academic Web caching literature, they are foremost in industry analysts' minds:

[*]Published in the proceedings of the Fifth International Web Caching and Content Delivery Workshop, 22–24 May 2000, Lisbon, Portugal. Details at `http://www.terena.nl/conf/wcw/` The most recent version of this paper is available at
`http://ai.eecs.umich.edu/~tpkelly/papers/`
`$Id: wcp.tex,v 1.36 2000/04/29 19:18:00 tpkelly Exp $`

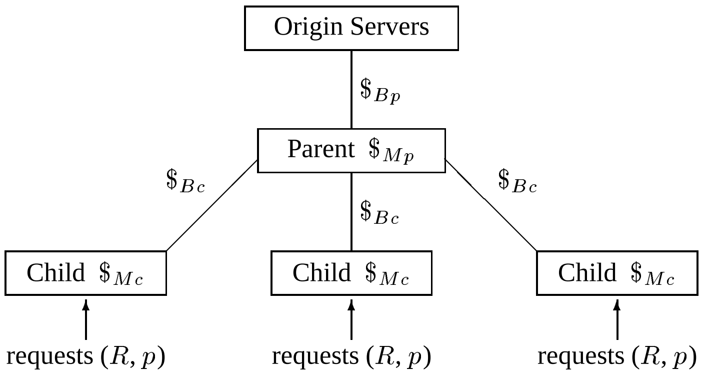

Figure 1: Two-level caching hierarchy of Section 2.

> CacheFlow is targeting the enterprise, where most network managers will be loath to spend $40,000 to save bandwidth on a $1,200-per-month T1 line. To sell these boxes, CacheFlow must wise up and deliver an entry-level appliance starting at $7,000 [24].

This paper considers approaches to the problem of determining optimal cache sizes based on economic considerations. We focus exclusively on the storage cost vs. miss cost tradeoff, ignoring throughput and response time issues, which are covered extensively elsewhere [31, 15]. Performance constraints and cost minimization may safely be considered separately in the cache sizing problem, because one should always choose the larger of the two cache sizes they separately require. In other words, if economic arguments prescribe a larger cache than needed to satisfy throughput and latency targets, an opportunity exists to save money overall by spending money on additional storage capacity; we might therefore say that our topic is optimal cache *expansion* rather than optimal sizing.

We begin in Section 2 with a simple model that includes only memory and bandwidth costs. We believe that the memory/bandwidth tradeoff is the right one to consider in a highly simplified model, because the monetary costs of both resources are readily available, and because bandwidth savings is the main reason why organizations deploy Web caches.[1] The analysis of Section 2 is reminiscent of Gray & Putzolu's "five-minute rule" [19], but it extends to large-scale hierarchical caching systems. We show how the economic viability of a shared high-level cache is related to system size and technology cost ratios. We furthermore demonstrate that under certain conditions, globally-optimal storage capacities in a large-scale caching hierarchy can be determined through scalable, decentralized, local computations. Section 3 addresses the shortcomings of our simple model's assumptions, describing an efficient method of computing the optimal storage capacity of a single cache for *completely arbitrary* workloads, miss costs, and storage costs. We employ an algorithm that computes *complete* stack distance transformations and arbitrarily-weighted hit ratios at *all* cache sizes for large traces using modest computational resources. We provide a simple implementation of our fast simultaneous simulation algorithm [25] and present results demonstrating that it computes stack distances and hit rates more quickly than alternative methods. Section 4 concludes by discussing our contributions in relation to other work.

2 A Simple Hierarchical Caching Model

In this section we consider a two-level cache hierarchy in which C lower-level caches each receive identical request streams at the rate of R references per second as depicted in Figure 1. Requests that cannot be served by one of these "child" caches are forwarded to a single higher-level "parent" cache. A document of size S_i bytes may be stored in a child or parent cache at a cost, respectively, of $\$_{Mc}$ or $\$_{Mp}$ dollars per byte. Bandwidth between origin servers and the parent costs $\$_{Bp}$ dollars per byte per second, and bandwidth between the parent and each child costs $\$_{Bc}$. Our objective is to serve the child request streams at minimal overall cost in the long-term steady state (all caches "warm"). The tradeoff at issue is the cost of storing documents closer to where they are requested versus the cost of repeatedly retrieving them from more distant locations in response to requests.

Request streams are described by an independent reference model in which document i is requested with relative frequency p_i where $\sum_i p_i = 1$; the rate of request for document i is therefore $p_i R$ requests per second. The model

[1]According to a survey of Fortune1000 network managers who have deployed Web caches, 54% do so to save bandwidth, 32% to improve response time, 25% for security reasons, and 14% to restrict employee access [22].

Table 1: Notation of Section 2.

M	total number of requests	N	total number of distinct documents
C	number of child caches	R	request rate at children (requests/second)
i	index of a typical document	p_i	relative popularity of document i, $\sum_i p_i = 1$
S_i	size of document i (bytes)	$\$_{Mc}$	cost of storage at a child cache (\$/byte)
$\$_{Mp}$	cost of storage at parent cache (\$/byte)	$\$_M$	cost of storage when $\$_{Mc} = \$_{Mp}$ (\$/byte)
$\$_{Bc}$	child-parent B/W cost (\$/(byte/sec))	$\$_{Bp}$	parent-server B/W cost (\$/(byte/sec))

of Breslau et al. [13] (independent references from a Zipf-like popularity distribution) is an example of the class of reference streams we consider. Given independent references drawn from a fixed distribution, the most natural cache removal policy is "Perfect LFU", i.e., LFU with reference counts that persist across evictions [13] (Perfect LFU is *optimal* for such a workload only if documents are of uniform size). We therefore assume that all caches use Perfect LFU replacement.

2.1 Centralized Optimization

Because we ignore congestion effects at caches and on transmission links, we may compute optimal cache sizes by determining optimal dispositions for each *document* independently, and then sizing caches accordingly. A document may be cached 1) at the parent, 2) at *all* children, or 3) nowhere. These alternatives are mutually exclusive: By symmetry, if it pays to cache a document at any child, then it ought to be cached at all children; and if a document is cached at the children it is pointless to cache it at the parent. The costs of the three options for document i are

cache at children	cache at parent	don't cache
$CS_i\$_{Mc}$	$S_i\$_{Mp} + Cp_iRS_i\$_{Bc}$	$Cp_iRS_i(\$_{Bp} + \$_{Bc})$

The document should be cached at the children if and only if this option is cheaper than the alternatives (we break ties by caching documents closer to children, rather than farther):

$$CS_i\$_{Mc} \leq S_i\$_{Mp} + Cp_iRS_i\$_{Bc} \quad \Rightarrow \quad p_i \geq \frac{C\$_{Mc} - \$_{Mp}}{CR\$_{Bc}} \tag{1}$$

$$CS_i\$_{Mc} \leq Cp_iRS_i(\$_{Bp} + \$_{Bc}) \quad \Rightarrow \quad p_i \geq \frac{\$_{Mc}}{R(\$_{Bp} + \$_{Bc})} \tag{2}$$

Each child cache should therefore be exactly large enough to accommodate documents i whose popularity p_i satisfies Equations 1 and 2; Perfect LFU replacement ensures that, in the long-term steady state, precisely those documents will be cached at the children. By similar reasoning, the parent cache should be just big enough to hold documents for which parent caching is the cheapest option:

$$p_i < \frac{C\$_{Mc} - \$_{Mp}}{CR\$_{Bc}} \tag{3}$$

$$S_i\$_{Mp} + Cp_iRS_i\$_{Bc} \leq Cp_iRS_i(\$_{Bp} + \$_{Bc}) \quad \Rightarrow \quad p_i \geq \frac{\$_{Mp}}{CR\$_{Bp}} \tag{4}$$

Taken together, the requirements for parent caching (Equations 3 and 4) imply that a parent cache is only justifiable if there are enough children:

$$\frac{C\$_{Mc} - \$_{Mp}}{CR\$_{Bc}} > p_i \geq \frac{\$_{Mp}}{CR\$_{Bp}} \quad \Rightarrow \quad C > \frac{\$_{Mp}\$_{Bc}/\$_{Bp} + \$_{Mp}}{\$_{Mc}} \tag{5}$$

Equation 5 is a *necessary* condition for a shared parent cache to be economically viable, as is the existence of at least one document whose popularity satisfies Equations 3 and 4. Together, the two conditions are *sufficient* to justify a parent cache under our model assumptions, provided that a parent cache entails no fixed costs. In practice, of course, the fixed cost of purchasing and installing a cache is often substantial. In such cases, the proper procedure for determining whether a shared parent cache is economically justifiable is as follows: Compute overall cost (of memory, bandwidth, and fixed costs) in a system with an optimally-sized parent cache, i.e., one capable of holding all documents that satisfy Equations 3 and 4. Compare this with total costs in a system without a parent cache, and choose the cheaper option.

Of particular interest is the special case where per-byte memory costs at parent and children are equal, and the number of children is large. If $\$_{Mp} = \$_{Mc} = \$_M$ then Equation 5 simplifies to

$$C > \frac{\$_{Bc}}{\$_{Bp}} + 1 \tag{6}$$

If in addition to uniform memory costs we furthermore assume that C is very large, the criteria for caching at a child (Equations 1 and 2) simplify to

$$p_i \geq \left(\frac{C-1}{C}\right) \frac{\$_M}{R\$_{Bc}} \approx \frac{\$_M}{R\$_{Bc}} \quad \text{and} \quad p_i \geq \frac{\$_M}{R(\$_{Bc} + \$_{Bp})}$$

If the first of these inequalities is satisfied, then the second must also be satisfied, because R and all costs are strictly positive. Therefore in the case where the number of children is large and memory costs are identical at parent and children, document i should be cached at children iff

$$p_i \geq \frac{\$_M}{R\$_{Bc}} \tag{7}$$

2.2 Decentralized Optimization

We now consider circumstances under which a *decentralized* computation that uses only local information yields the same result as the centralized computation of Section 2.1.

Imagine that the parent and child caches are operated by independent entities, each of which seeks to minimize its own operating costs ($\$_{Mp}$ and $\$_{Bp}$ for the parent, $\$_{Mc}$ and $\$_{Bc}$ for the children). Each child's decision whether or not to cache each document is independent of whether the document is cached at the parent, because the transmission and storage costs facing children are unaffected by caching decisions at the parent. The higher-level cache in turn bases its caching decisions solely on the document requests submitted to it and the costs it must pay in order to satisfy them. A child will cache document i iff

$$S_i \$_{Mc} \leq S_i p_i R \$_{Bc} \quad \Rightarrow \quad p_i \geq \frac{\$_{Mc}}{R\$_{Bc}} \tag{8}$$

After the lower-level caches have sized themselves to accommodate documents whose rate of request satisfies Equation 8, requests for those documents will not reach the parent. The parent will, however, receive requests for all other documents j at the rate of Cp_jR, and will choose to cache all documents that satisfy

$$S_j \$_{Mp} \leq C p_j R \$_{Bp} \quad \Rightarrow \quad p_j \geq \frac{\$_{Mp}}{CR\$_{Bp}} \tag{9}$$

The condition of Equation 9 is identical to that of our previous centralized-optimization result (Equation 4). Furthermore, when memory costs are uniform Equation 8 becomes the child-caching criterion for large numbers of children (Equation 7). Therefore the caching decisions—and hence cache sizes—determined independently through (literally) greedy local computations are the same as those that a globally-optimizing "central planner" would compute.

2.3 Cost Calculations

In practice bandwidth costs rarely have the convenient dimensions we have thus far assumed, because they typically involve fixed installation costs as well as periodic maintenance and service fees. However, we can convert periodic costs into a single cost using a standard present-value calculation [12]; in the simplest case, $\text{PV} = \text{payment}/\text{interest rate}$. For example, if the annual interest rate is 5%, the present value of perpetual yearly payments of \$37 is \$37/.05 = \$740. Slightly more sophisticated calculations can account for finite time horizons (depreciation periods) and variable interest rates.[2]

In order to put the model of this section in perspective, we briefly consider the actual costs of bandwidth in our area (the midwestern United States). Table 2 presents prices charged by a major Internet Service Provider near our home institution and corresponding bandwidth costs assuming a 5% annual interest rate.

[2]A back-of-the-envelope PV calculation sheds light on the industry analysts' negative remark about CacheFlow cited in Section 1: If the appliance yields a 15% bandwidth savings on a \$1,200/month line (\$180/month in cost savings) and if the annual interest rate is 5%, then the product's present value exceeds \$40,000. However, if we assume a finite product life, we find that PV exceeds purchase price only for lifetimes of roughly 7 years or more assuming 50% bandwidth savings.

Table 2: Merit Networks Inc. prices of Internet connectivity for commercial and educational customers in U.S. dollars.

Technology & Bandwidth	Installation	Annual costs Edu	Annual costs Comm	$\$_B$ ($/(byte/sec)) Edu	$\$_B$ ($/(byte/sec)) Comm
Private Line					
56 Kbps	6,602	8,395	9,520	24.93	28.14
ISDN					
64 Kbps	3,763	7,484	8,609	19.18	21.99
128 Kbps	3,763	8,504	10,609	10.87	13.50
256 Kbps	9,880	10,377	13,217	6.79	8.57
384 Kbps	10,224	11,996	15,326	5.21	6.60
Fractional T1					
128 Kbps	7,307	14,077	16,182	18.05	20.68
256 Kbps	7,307	14,842	17,682	9.50	11.28
384 Kbps	7,307	15,352	18,682	6.55	7.93
768 Kbps	7,307	16,882	20,682	3.59	4.38
Full T1 line(s)					
1.5 Mbps	7,307	19,942	24,682	2.17	2.67
3.0 Mbps	9,962	35,344	40,163	1.91	2.17

Table 3: LAN bandwidth costs of 10 Mbps shared Ethernet at the University of Michigan.

number of clients	installation cost ($)	bandwidth per client (bytes/sec)	bandwidth cost ($/(byte/sec))	number of clients	installation cost ($)	bandwidth per client (bytes/sec)	bandwidth cost ($/(byte/sec))
1	25803	1250000.0	0.020643	30	40414	41666.7	0.969936
5	27819	250000.0	0.111276	40	45452	31250.0	1.454464
10	30338	125000.0	0.242704	50	50490	25000.0	2.019600
15	32857	83333.3	0.394284	75	63085	16666.7	3.785100
20	35376	62500.0	0.566016	100	75680	12500.0	6.054400
25	37895	50000.0	0.757900				

As a crude estimate of LAN bandwidth costs, we consider the cost of 10 Mbps shared Ethernet installations at our home institution. Table 3 presents LAN bandwidth costs based the University of Michigan's internal prices. Prices shown are determined by the following formula:

$$\text{price} = 1.1 \times (\text{number of hosts} \times \$458 + \$23,000)$$

Consistent with the assumptions of this section, we compute available bandwidth per LAN client for the idealized case of identical client behavior. Note that if we take any $\$_{Bp}$ from Table 2 and any C and $\$_{Bc}$ from Table 3, these will satisfy Equation 6 for any $C > 1$. If this seems counter-intuitive, recall that we assume *identical* child workloads, i.e., we assume perfect sharing in lower-level caches' reference patterns. Furthermore, note that Equation 6 is a necessary but not a sufficient condition for a parent cache to be economically justifiable.

Some readers may object that technology costs fluctuate too rapidly to guide design decisions. While it is true that memory and bandwidth prices change rapidly, engineering principles and rules of thumb based on technology price *ratios* have remained remarkably robust for long periods [18, 20], and the main results of this section are stated in terms of ratios.

In order to apply the methods of Section 2.1 or Section 2.2 in an optimal cache size computation, we require both detailed workload data (R and p_i) and technology costs ($\$_M$ and $\$_B$) for the same site at which the workload is recorded. Web proxy workloads are readily available, but they are not accompanied by technology cost information, and our efforts to obtain cost data for the traces we use in our empirical work failed. Similarly, we were unable to obtain large, high-quality workloads for the one site where we do have access to cost data, because Web caches are not widely deployed on our University campus. We choose not to mix and match data from different sources by, for example, combining workload and cost data from different times and sites, and therefore we do not use the methods of

Table 4: Notation of Section 3

M	total number of requests	N	total number of distinct documents requested
x_t	document requested at virtual time t	S_i	size of document i (bytes)
$\$_t$	cost incurred if request at time t misses (\$)	$\$_M(s)$	storage cost of cache capacity s (\$)
D_t	set of documents requested up to time t	$P_t(i)$	priority of document $i \in D_t$
δ_t	priority depth of documents in D_t (bytes)	$\$_A(s)$	total miss cost over entire reference sequence (\$)

this section or the next to compute actual cache sizes. We do not regard this as a serious deficiency. Our main intent is to describe general methods for computing the optimal value of an important parameter, not to share anecdotes about the specific values that we obtain when we apply these methods to particular inputs.

3 A Detailed Model of Single Caches

The model assumptions and optimization procedures of Section 2 are problematic for several reasons: The workload model assumes an idealized steady state, ignoring such features as temporal locality and the creation of new documents at servers. Production caches use variants of LRU; many cache designers reject Perfect LFU because of its higher time and memory overhead. Storage and miss costs are not simple linear functions of capacity.

In this section we describe a method that suffers from none of these problems. We assume that 1) workload is described by an *explicit sequence* of requests; 2) associated with each request is an *arbitrary* miss cost; 3) the cache uses one of a large family of replacement policies that includes LRU and a variant of Perfect LFU; and 4) the cost of cache storage capacity is an arbitrary monotonic function. It is straightforward to extend the algorithm of this section to multi-level storage hierarchies in which each cache has at most one parent or child, as described in Mattson et al. [30]. It is not clear, however, that the method can be extended to the more interesting case in which shared high-level caches serve multiple children.

Our cache workload consists of a sequence of M references $x_1, x_2, \ldots, x_M$ where subscripts indicate the "virtual time" of each request: if the request at time t is for document i, then $x_t = i$. Associated with each reference is a nonnegative miss cost $\$_t$. Whereas document sizes are constant, the miss costs associated with different requests for the same document need not be equal: if $x_t = x_{t'} = i$ for $t \neq t'$ we require $S_{x_t} = S_{x_{t'}} = S_i$, but we permit $\$_t \neq \$_{t'}$ (e.g., miss costs may be assessed higher during peak usage periods). Finally, the cost of cache storage $\$_M(s)$ is an arbitrary nondecreasing function of cache capacity s; this permits us to consider, e.g., fixed costs.

The set of documents requested up to time t is denoted $D_t \equiv \{i : x_{t'} = i \text{ for some } t' \leq t\}$. A scalar *priority* P_t is defined over documents in D_t; two documents never have equal priority: $P_t(i) = P_t(j)$ iff $i = j$. Informally, the *priority depth* δ_t of a document $i \in D_t$ is the smallest cache size at which a reference to the document will result in a cache hit. Formally,

$$\delta_t(i) \equiv S_i + \sum_{h \in H_t} S_h \quad \text{where} \quad H_t \equiv \{h \in D_t : P_t(h) > P_t(i)\} \tag{10}$$

The priority depth of documents not in D_t is defined to be infinity. Priority depth generalizes the familiar notion of LRU stack distance [30] to the case of non-uniform document sizes and general priority functions. Let

$$\$_A(s) \equiv \sum_{t=1}^{M} \$_t I_t(s) \quad \text{where} \quad I_t(s) \equiv \begin{cases} 0 & \text{if } s \geq \delta_t(x_t) \\ 1 & \text{otherwise} \end{cases}$$

denote total miss cost over the entire reference sequence as a function of "size" parameter s. For every input sequence, $\$_A(s)$ is equal to the aggregate miss cost incurred by a cache of size s whose removal priority is defined by P if and only if 1) $s \geq \max_i S_i$, and 2) the cache removal policy satisfies the *inclusion property*, meaning that a cache of size s will always contain any smaller cache's contents. The second requirement is familiar from the literature on stack distance transformations of reference streams; replacement policies with this property are sometimes known as "stack policies" [30, 11, 32, 37]. LRU and the variant of Perfect LFU that caches a requested document only if it has sufficiently high priority ("optional-placement Perfect LFU") are stack policies; FIFO and mandatory-placement LFUs are not [30].[3] The first requirement is necessary because aggregate miss cost is monotonic only for cache sizes capable of holding any document.

[3]The distinction between mandatory- and optional-placement policies is important. Whereas models of processor memory hierarchies typically assume mandatory placement (e.g., Sleator & Tarjan on paging policies [34]), in Web caching we need not require that a requested document

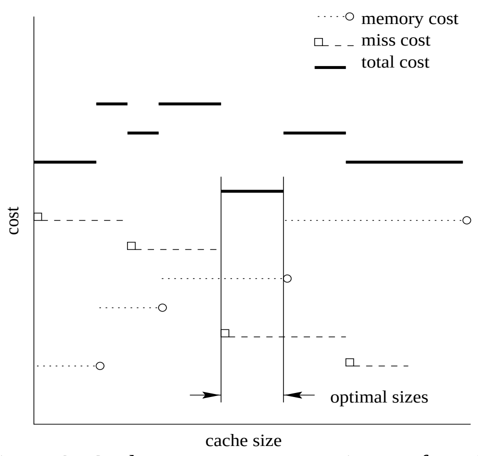

Figure 2: Cache costs as monotonic step functions.

Given $\$_A(s)$ we can efficiently determine a cache size s that minimizes total cost $\$_A(s) + \$_M(s)$. Because storage cost is nondecreasing in cache capacity, we need not consider total cost at all cache sizes: $\$_A(s)$ is a "step function" that is nonincreasing in s, with at most M "steps," and minimal overall cost must occur at one of them (see Figure 2). We may therefore determine a (not necessarily unique) cache size that minimizes total cost in $O(M)$ time.

At first glance, it might appear that the bottleneck in our overall approach to computing optimal cache size is the computation of priority depth (Equation 10). A straightforward implementation of a priority list, e.g., as a linked list, would require $O(N)$ memory and $O(N)$ time per reference for a total of $O(MN)$ time to process the entire sequence of M requests. For reasonable removal policies, however, it is possible to perform this computation in $O(M \log N)$ time and $O(N)$ memory using an algorithm reminiscent of those developed for efficient processor-memory simulation [11, 32, 37]; we describe our priority-depth algorithm in Section 3.1. Given a pair $(\delta_t(x_t), \$_t)$ for each of M requests, we can compute $\$_A(s)$ after sorting these pairs on δ in $O(M \log M)$ time and $O(M)$ memory. This "post-processing" sorting step is therefore the computational bottleneck for any trace workload, in which $M \geq N$. By contrast, a simulation of a *single cache size* would require $O(M \log N)$ time for practical removal policies.

3.1 Fast Simultaneous Simulation

In this section we briefly outline an algorithm which computes δ_t for each of M references in $O(M \log N)$ time and $O(N)$ memory by making a single pass over the input sequence. Because it allows us to compute $\$_A(s)$ at the additional cost of sorting the output, in effect this algorithm enables us to simulate *all* cache sizes of possible interest simultaneously. An efficient method is necessary in order to process real traces, in which M and N can both exceed 10 million [27]. To make the issue concrete, whereas a naïve $O(MN)$ priority depth algorithm required over five days to process 11.6 million requests for 5.25 million documents, our $O(M \log N)$ algorithm completed the job in roughly three minutes on the same computer.

In order for our method to work, we require that the priority function P corresponding to the cache's removal policy satisfy an additional constraint: The relative priority of two documents may only change when one of them is referenced. This is not an overly restrictive assumption; indeed, some researchers regard it as a requirement for a practical replacement policy, because it permits requests to be processed in logarithmic time [8].

We represent documents in the set D_t as nodes of a binary tree, where an inorder traversal visits document records in ascending priority. We require one node per document, hence the $O(N)$ memory requirement. At each node we store the aggregate size of all documents in the right (higher-priority) subtree; we can therefore recover $\delta_t(i)$ by traversing the path from document i's node to the root. To process a request, we output the referenced document's priority depth, remove the corresponding node from the tree, adjust its priority, and re-insert it. Tree nodes are allocated in an N-long array indexed by document ID, so locating a node requires $O(1)$ time. All of the other operations require $O(\log N)$ time, for a total of $O(M \log N)$ time to process the entire input sequence.

always be cached (as in Irani's discussion of variable-page-size caching [23]). Optional-placement Perfect LFU is optimal for infinite sequences of independent references from a fixed distribution, if document sizes are uniform. Limited empirical evidence, however, suggests that optional-placement variants of LFU perform worse than their mandatory-placement counterparts on real Web workloads [27]; the subject has not been investigated thoroughly. GD-Size [14] and mandatory-placement variants of LFU such as GDSF [4], swLFU [26, 27], and LUV [8] do not satisfy the inclusion property, and therefore the one-pass simulation methods described in Section 3.1 cannot be applied to them.

Table 5: Traces derived from access logs recorded at six NLANR sites, 1–28 March 1999. Run times shown are wall-clock times to compute given quantities, in seconds. The run times sum to under four hours, ten minutes.

	BO1	PA	PB	SD	SV	UC
# docs (millions)	5.25	4.90	9.82	8.64	9.38	7.62
# reqs (millions)	11.58	13.55	19.80	37.09	23.74	26.02
max S_i (MB)	218.6	104.9	218.7	175.0	107.4	175.0
unique bytes (billions)	104.5	76.0	188.3	204.9	159.1	150.1
bytes req'd (billions)	236.2	220.7	383.1	620.3	412.9	397.5

	BO1	PA	PB	SD	SV	UC
run times (sec):						
stack distances	249	341	403	1117	581	716
priority depths	230	288	399	872	547	587
HR(size)	309	414	497	1439	712	903
BHR(size)	314	423	522	1461	740	913

For all removal policies of practical interest, a document's priority only *increases* when it is accessed. A simple binary tree would therefore quickly degenerate into a linked list, so we use a splay tree to ensure (amortized) logarithmic time per operation [35, 28]. It is possible to maintain the invariant that each tree node stores the total size of all documents represented in its right subtree during insertions, deletions, and "splay" operations without altering the overall asymptotic time or memory complexity of the standard splay tree algorithm. A simple ANSI C implementation of our priority depth algorithm is available [25].

We devised our efficient priority depth algorithm before we became aware of similar techniques dating back to the mid-1970s [11, 32, 37], which appear not to be widely used in Web-related literature. To the best of our knowledge, no recent papers containing stack depth analyses (e.g., References [1, 2, 7, 10, 9, 29]) cite the most important papers on efficient stack distance computation (References [11, 32, 37]). The idea of using splay trees as we do is suggested by Thompson, who used AVL trees in his own work [37]. Our algorithm is simpler than those described in the processor-memory-caching literature because we ignore associativity considerations and assume that cached data is read-only. It is more general and better suited to Web caching because it handles variable document sizes, arbitrary miss costs, and a wide range of optional-placement cache policies.

3.2 Numerical Results

To illustrate the flexibility and efficiency of our priority depth algorithm, we used it to compute *complete* stack distance transformations and LRU hit rates at *all* cache sizes for six four-week NLANR [17] Web cache traces summarized in Table 5 and described more fully in Reference [27]. Similarly detailed results rarely appear in the Web caching literature.[4] Perhaps this is because such complete and exact calculations have been viewed as computationally infeasible. All of the results presented here, however, were computed in a total of under five hours on an unspectacular machine—far less time than was required to download our raw trace data from NLANR.[5] Finally, we describe a timing test conducted outside of our research group that shows that our priority depth implementation computes stack distances substantially faster than two alternatives.

Figure 4 shows LRU hit rates and byte hit rates at all cache sizes for our six Web traces, computed by our splay-tree-based priority depth algorithm. For the workloads considered, exact performance measurements at all cache sizes appear to offer little *visual* advantage over the customary technique of interpolating measurements taken at regular intervals (e.g., 1GB, 2GB, 4GB, etc.) via single-cache-size simulation. However, since exact hit rate functions may be obtained at very modest computational cost, it is not clear that a less precise approach offers any advantage, either.

LRU stack distance, a standard measure of temporal locality in symbolic reference streams, is a special-case output of our priority depth algorithm when all document sizes are 1. Mattson et al. is the classic reference on stack distance analysis [30]; Almeida et al. [2] and Arlitt & Williamson [7] apply the technique to Web traces. The frequency distribution of stack distances from our six traces is shown on the left in Figure 3. Frequency distributions visually exaggerate temporal locality, particularly when (as is common in the literature) the horizontal axis is truncated at a shallow depth. The situation does not improve if we aggregate the observed stack distances into constant-width bins, because as Arlitt & Williamson have noted, the visual impression of temporal locality created depends on the bin sizes we choose [7]. Perhaps the clearest and least ambiguous way to present these data is with a cumulative distribution, as on the right in Figure 3, from which order statistics such as the median and quartile stack distances are directly apparent.

[4]Almeida et al. present complete stack distance traces for four Web server workloads ranging in size from 28,000–80,000 requests [1, 2]. They furthermore note that the marginal distribution of a stack distance trace is related to cache miss rate, but their discussion assumes uniform document sizes. Arlitt et al. present the only stack depth analysis of large traces (up to 1.35 billion references) of which we are aware [5, 6].

[5]We used a Dell Poweredge 6300 server with four 450-MHz Intel Pentium II Xeon processors and 512 MB of RAM running Linux kernel 2.2.12-20smp.

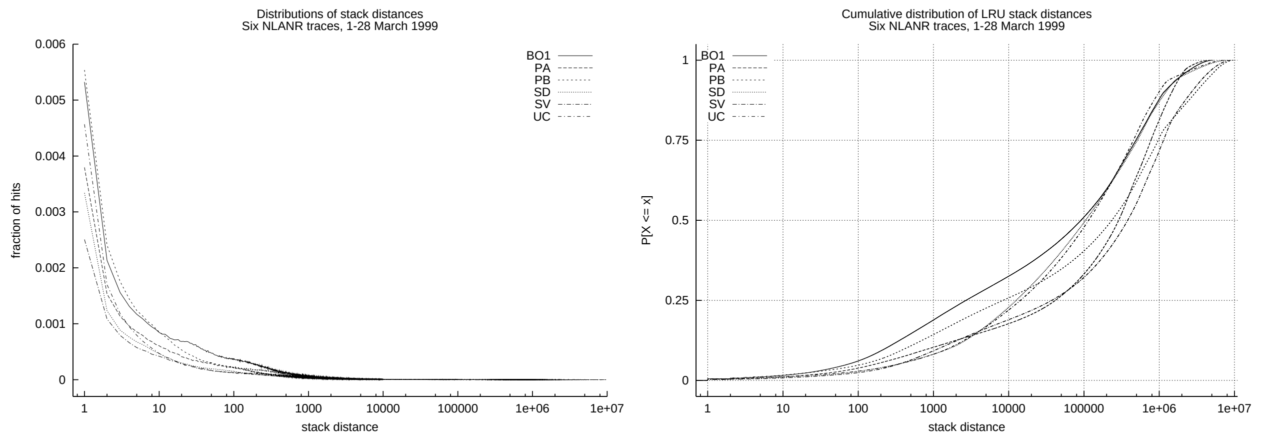

Figure 3: Frequency distribution (left) and cumulative distribution (right) of LRU stack distances in six traces. Compare these data with Table 10 and Figure 8 of Arlitt & Jin [6]; temporal locality is far weaker in our network cache workload than in their very large server workload.

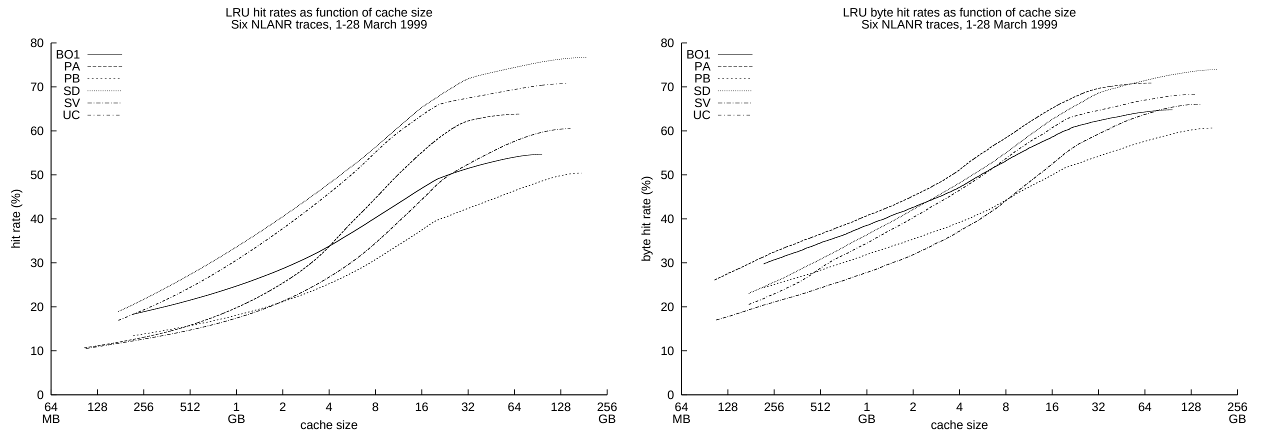

Figure 4: Exact hit rates (left) and byte hit rates (right) as function of cache size for six large traces, LRU removal. Fast simultaneous simulation method yields correct results only for cache sizes $\geq$ largest object size in a trace; smaller cache sizes not shown.

Martin Arlitt of Hewlett-Packard Labs recently compared the speed of three stack distance programs: The first author's publicly-available implementation of the fast splay-tree-based priority depth algorithm of Section 3.1 [25], a simple $O(MN)$ linked-list implementation supplied with Kelly's fast code, and Arlitt's own program [3]. Arlitt's implementation divides the LRU stack into a number of equal-sized "bins," each of which contains 50 items. The advantage of this approach is that the worst-case number of operations to process a reference is proportional to the number of bins plus the number of items in a bin, rather than to the number of items. In the asymptotic analysis, however, this strategy still requires $O(MN)$ time to process its entire input. The trace used for this test is the largest of which we are aware: 1,352,804,108 references to 2,770,108 unique items derived from the World Cup Web server workload described in Reference [6] and available from the Web Characterization Repository [21]. Temporal locality is strong in this trace (the median stack depth is 179); it is therefore "friendly" to the simple linked-list implementation. Run times were as follows: 446 hours 24 minutes for the simple list, 45:52 for the bin/list hybrid, and 18:40 for the splay tree algorithm. Breaking the LRU stack into bins yields a nearly tenfold speedup over a simple linked list (from roughly 18 days to under two days). The splay-tree algorithm runs 24 times faster than the linked list and more than twice as fast as the hybrid list/bin approach. Other results, too tentative and incomplete to report here, confirm our intuiton that the performance advantages of our splay-tree algorithm are inversely related to locality. Arlitt found that both the list and bin/list implementations outperform the splay tree code on a Web server trace with very high locality, whereas the splay tree offers very dramatic advantages on a Web proxy workload with relatively low temporal locality.

4 Discussion

The main contribution of Section 2 is to generalize a familiar principle—the five minute rule—to multi-level branching storage hierarchies. Simple rules of thumb such as this often support illuminating back-of-the-envelope calculations; see Gray & Shenoy [20] for a good recent summary of time-tested rules of thumb updated for the capabilities and costs of modern technologies. The results of Section 2.2 suggest that under some circumstances centralized design may yield few benefits, because decentralized computation may lead to equally good outcomes. More sophisticated results of a similar flavor can be found in the literature on microeconomic approaches to distributed resource allocation [38]. The main contribution of Section 3 is to present a fast simultaneous simulation technique adopted from the processor-memory literature but more suitable to Web-related research, and demonstrate its application to the optimal cache sizing problem.

The idealized model of Section 2 is useful for computing optimal cache sizes only to the extent that its underlying workload and cost assumptions are valid. Breslau et al. have argued that the assumption of independent references is approximately correct [13], but Almeida et al. have described several shortcomings of this model and have proposed more accurate alternatives [1, 2]. Section 2 assumes homogeneous lower-level caches; Wolman et al. explore the implications of sharing among heterogeneous client aggregates, and furthermore consider document modification rates, which we have ignored [39, 40]. The main formal weakness of our hierarchical caching model is the simple linear cost model. In many cases of practical interest, memory and bandwidth costs are step functions that do not admit accurate linear approximations. From a practical standpoint, another serious weakness is that we ignore the low-level aspects of Web operation. Feldmann et al. report that details such as bandwidth heterogeneity and aborted transfers can negate the bandwidth savings that proxy caching would otherwise yield [16]. An important goal of our ongoing work is to determine how much detail can be added to the model of Section 2 without sacrificing scalable decentralized computation of optimal cache sizes.

The major limitation of the single-cache optimization method of Section 3 is that in its present form it cannot account for document expirations. However, the problem of modeling a cache that evicts stale documents is quite similar to that of modeling write invalidations in processor cache hierarchies, and the stack algorithm literature considers this problem [30, 32, 37, 36]. Our simulator implementation [25] does not currently support document expirations, but it is likely that this feature will be added to support our future research.

Acknowledgments

Professors Michael Wellman, Jeff MacKie-Mason and Peter Honeyman gave valuable guidance to this project. Our research was funded by DARPA/ITO grant F30602-97-1-0228. Trace data courtesy NLANR (NSF grants NCR-9616602 and NCR-9521745). Jeff Kopmanis made available computational resources essential to our empirical work. Martin Arlitt of HP Labs offered feedback on a draft and performed a lengthy comparison test of our stack distance program with his own. Jim Gray of Microsoft Research reviewed our work, referred us to relevant literature, and shared an interesting forthcoming paper with us. Michael Sparks of the University of Manchester supplied detailed information on practical issues surrounding Web cache management. We are particularly grateful for the generous assistance of Merit Network, Inc. and the University of Michigan's IT Division: Jeff Ogden provided bandwidth pricing data for Table 2, and JoElla Coles supplied the data in Table 3.

References

[1] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. Technical Report TR-96-11, Boston University Computer Science Department, 1996. `http://www.cs.bu.edu/techreports/`.

[2] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS96)*, December 1996. This is a shorter and more recent version of Reference [1].

[3] Martin Arlitt. Personal communication.

[4] Martin Arlitt, Ludmila Cherkasova, John Dilley, Richard Friedrich, and Tai Jin. Evaluating content management techniques for Web proxy caches. In *Proceedings of the Second Workshop on Internet Server Performance (WISP '99)*, May 1999. Also available as an HP Labs tech report at `http://www.hpl.hp.com/techreports/98/HPL-98-173.html`.

[5] Martin Arlitt, Rich Friedrich, and Tai Jin. Workload characterization of a Web proxy in a cable modem environment. Technical Report HPL-1999-48, Hewlett-Packard Laboratories, 1999. `http://www.hpl.hp.com/techreports/1999/HPL-1999-48.html`.

[6] Martin Arlitt and Tai Jin. Workload characterization of the 1998 World Cup Web site. Technical Report HPL-1999-35R1, Hewlett-Packard Labs, September 1999. `http://www.hpl.hp.com/techreports/1999/HPL-1999-35R1.html`.

[7] Martin F. Arlitt and Carey L. Williamson. Internet Web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–644, October 1997.

[8] Hyokyung Bahn, Sam H. Noh, Kern Koh, and Sang Lyul Min. Using full reference history for efficient document replacement in Web caches. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS99)*, November 1999. `http://www.cs.hongik.ac.kr/˜dnps/research/pub.html`.

[9] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web client access patterns: Characteristics and caching implications. *World Wide Web Journal, Special Issue on Characterization and Performance Evaluation*, 1999. Also available as Boston U. CS tech report 1998-023 at `http://www.cs.bu.edu/techreports/`.

[10] Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the 1998 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 151–160, July 1998. `http://www.cs.bu.edu/faculty/crovella/paper-archive/sigm98-surge.ps`.

[11] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.

[12] Richard A. Brealey and Stewart C. Meyers. *Principles of Corporate Finance*. McGraw-Hill, sixth edition, 2000. ISBN 0-07-117901-1.

[13] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom99*, March 1999. Tech report version available at `http://www.cs.wisc.edu/˜cao/papers/`.

[14] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pages 193–206, December 1997. `http://www.cs.wisc.edu/˜cao/papers/gd-size.html`.

[15] John Dilley, Rich Friedrich, Tai Jin, and Jerome Rolia. Web server performance measurement and modeling techniques. *Performance Evaluation*, 33:5–26, 1998.

[16] Anja Feldmann, Ramón Cáceres, Fred Douglis, Gideon Glass, and Michael Rabinovich. Performance of Web proxy caching in heterogeneous bandwidth environments. In *Proceedings of IEEE INFOCOM '99*, March 1999.

[17] National Laboratory for Applied Network Research. Anonymized access logs. `ftp://ftp.ircache.net/Traces/`.

[18] Jim Gray and Goetz Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. Technical Report MSR-TR-97-33, Microsoft Research, September 1997. `http://www.research.microsoft.com/scripts/pubs/trpub.asp`.

[19] Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *Proceedings of ACM SIGMOD*, May 1987.

[20] Jim Gray and Prashant Shenoy. Rules of thumb in data engineering. Technical Report MS-TR-99-100, Microsoft Research, December 1999. Revised version dated February 2000 `http://www.research.microsoft.com/scripts/pubs/trpub.asp`.

[21] W3C Web Characterization Activity Working Group. Web Characterization Repository. `http://researchsmp2.cc.vt.edu/cgi-bin/reposit/index.pl`.

[22] Brendan Hannigan, Carl D. Howe, Sharon Chan, and Tom Buss. Why caching matters. Technical report, Forrester Research, Inc., October 1997.

[23] Sandy Irani. Page replacement with multi-size pages and applications to Web caching. In *29th ACM STOC*, pages 701–710, May 1997.

[24] Ted Julian and Brendan Hannigan. The cache appliance opportunity. Technical report, Forrester Research, Inc., January 1998.

[25] Terence Kelly. Priority depth (generalized stack distance) implementation in ANSI C, February 2000. `http://ai.eecs.umich.edu/˜tpkelly/papers/`.

[26] Terence Kelly, Yee Man Chan, Sugih Jamin, and Jeffrey K. MacKie-Mason. Biased replacement policies for Web caches: Differential quality-of-service and aggregate user value. In *Fourth International Web Caching Workshop*, March 1999. `http://ai.eecs.umich.edu/˜tpkelly/papers/wlfu.ps`.

[27] Terence Kelly, Sugih Jamin, and Jeffrey K. MacKie-Mason. Variable QoS from shared Web caches: User-centered design and value-sensitive replacement. In *Proceedings of the MIT Workshop on Internet Service Quality Economics (ISQE 99), Cambridge, MA*, December 1999. `http://www.marengoresearch.com/isqe/agenda_m.htm`.

[28] Dexter C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, 1992. ISBN 0-387-97687-6.

[29] Anirban Mahanti and Carey Williamson. Web proxy workload characterization. Technical report, Department of Computer Science, University of Saskatchewan, February 1999. `http://www.cs.usask.ca/faculty/carey/papers/workloadstudy.ps`.

[30] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[31] Daniel A. Menascé and Virgílio A. F. Almeida. *Capacity Planning for Web Performance: Metrics, Models, and Methods*. Prentice Hall, 1998. ISBN 0-13-693822-1.

[32] Frank Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Electrical Engineering and Computer Science Department, University of California, Berkeley; and Computer Science and Mathematics Department, Lawrence Berkeley Lab, May 1981. This is the author's Berkeley Masters thesis.

[33] Morgan Oslake. Capacity model for Internet transactions. Technical Report MSR-TR-99-18, Microsoft Research, April 1999.

[34] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[35] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Number 44 in CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1983. ISBN 0-89871-187-8.

[36] James G. Thompson and Alan Jay Smith. Efficient (stack) algorithms for analysis of write-back and sector memories. *ACM Transactions on Computer Systems*, 7(1):78–117, February 1989.

[37] James Gordon Thompson. Efficient analysis of caching systems. Technical Report UCB/CSD 87/374, Computer Science Division (EECS), University of California at Berkeley, October 1987. This is the author's Ph.D. dissertation.

[38] Michael P. Wellman. Market-oriented programming: Some early lessons. In S. Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, 1996. `http://ai.eecs.umich.edu/people/wellman/Publications.html`.

[39] Alec Wolman, Geoff Voelker, Nitin Sharma Neal Cardwell, Molly Brown, Tashana Landray, Denise Pinnel, Anna Karlin, and Henry Levy. Organization-based analysis of Web-object sharing and caching. In *Proceedings of the Second USENIX Conference on Internet Technologies and Systems (USITS '99),* October 1999. `http://www.cs.washington.edu/homes/wolman/`.

[40] Alec Wolman, Geoffrey M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy. On the scale and performance of cooperative Web proxy caching. *Operating Systems Review*, 34(5):16–31, December 1999. Originally in 17th ACM Symposium on Operating Systems Principles (SOSP '99). `http://www.cs.washington.edu/homes/wolman/`.

Vitae

Terence Kelly is a Ph.D. candidate at the University of Michigan, Ann Arbor, Michigan, USA. His current interests include the economics of distributed storage/retrieval systems and cache simulation techniques. His past research includes heuristic optimization, automotive traffic simulation, and Web caching/prefetching at Princeton University, the Santa Fe Institute, RAND Corporation, and Microsoft Research. He received his Master's degree in Computer Science from the University of Michigan in 1998 and his Bachelor's degree from Princeton University in 1992.

Dan Reeves is a Ph.D. student of Computer Science at the University of Michigan, having received his Master's degree there in 1999. His research interests are in Artificial Intelligence for E-commerce. He is actively involved with the Michigan Internet AuctionBot and has analyzed large data-sets from eBay auctions. His recent work involves automating the negotiation of declarative (rule-based) contracts. He is an avid Mathematica programmer and has built various libraries to support web communication in Mathematica programs.