Mobile App Acceleration via Fine-Grain Offloading to the Cloud

Chit-Kwan Lin
UpShift Labs, Inc.

H. T. Kung
Harvard University

Abstract

Mobile device hardware can limit the sophistication of mobile applications. One strategy for side-stepping these constraints is to opportunistically offload computations to the cloud, where more capable hardware can do the heavy lifting. We propose a platform that accomplishes this via *compressive offloading*, a novel application of compressive sensing in a distributed shared memory setting. Our prototype gives up to an order-of-magnitude acceleration and 60% longer battery life to the end user of an example handwriting recognition app. We argue that offloading is beneficial to both end users and cloud providers—the former experiences a performance boost and the latter receives a steady stream of small computations to fill periods of under-utilization. Such workloads, originating from ARM-based mobile devices, are especially well-suited for offloading to emerging ARM-based data centers.

1 Introduction

Despite the arrival of power-efficient multi-core CPUs and GPUs to smartphones and tablets, increasingly sophisticated mobile apps routinely push against the battery and processor limits of modern mobile devices. One way to alleviate these constraints is to offload computations to the cloud, where computing resources are effectively unlimited. This strategy is already manifest in special-purpose systems such as Siri and Google Now, but such web services cannot shift *arbitrary* workloads to the cloud at runtime. Instead, these apps must be statically partitioned into device- and cloud-specific components. Once implemented in this way, the partitioning cannot be changed easily or dynamically, rendering runtime optimization infeasible.

One solution to this problem casts the mobile device and the cloud as a *tightly coupled* distributed shared memory (DSM) system, in which memory on the mobile device is continuously replicated to a cloud server. Any in-memory object on the device will thus have a replica on the server, and any object method invoked on-device could be transparently and dynamically redirected to the remote replica for faster execution on more capable cloud hardware (Figure 1).

Recent efforts in code offloading [15, 12, 13] have explored related designs, but our approach of continuous memory replication has the advantage of allowing *fine-grain* workloads to be shifted up to the cloud at runtime. We emphasize fine granularity for two reasons. First, on-the-go mobile users demand high system responsiveness. A fine partitioning of work (i.e., at the level of an object method invocation) incurs less disruption to the user in the event the device becomes disconnected and a local restart of the task is required. Second, fine-grain workloads give low-latency cloud service providers a new class of work that can help maximize infrastructure utilization [16]. Both parties derive significant benefit from such workloads due to the disparity in the hardware resources each commands: the device owner sees the work as computationally complex and is glad to have it accelerated by someone else, while the cloud provider perceives it as small and computationally cheap and thus well-suited for flexibly filling troughs in utilization.

However, implementing a DSM, especially in a mobile setting, is known to be hard [17, 6, 10] due to latency, network bandwidth, power, and computation overhead constraints. Further complicating matters is that memory I/O, which is direct and random-access, is not naturally amenable to transaction logging techniques [21] that are typically used to maintain data consistency. This leaves existing replication methods that rely on communicating and comparing hashes to generate delta encodings (e.g., rsync) as the only available options for synchronizing the DSM. Unfortunately, such methods have computation and network overheads that do not respect the resource constraints of the mobile setting.

We propose *UpShift*, a platform that takes advantage

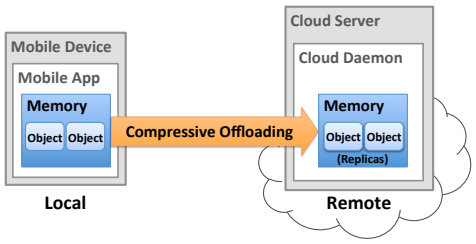


Figure 1: Compressive offloading uses compressive sensing to tightly replicate a memory page and its resident objects from device to server. On-device method invocations are redirected to a remote object replica for faster execution on more capable cloud hardware.

of recent advances in compressive sensing theory to realize a DSM with the requisite performance characteristics and thus the tight coupling needed to support offloading arbitrary, fine-grain work from a mobile device to the cloud. Below, we describe the platform's core compressive offloading mechanism, a prototype implementation on iOS, and initial performance evaluations.

2 Compressive Offloading

Compressive offloading is based on *compressive sensing* [14, 8, 9], an efficient random sampling technique in which a k-sparse signal $\mathbf{s} \in \mathbb{R}^N$ (i.e., only k coefficients in $\mathbf{s}$ are non-zero) is sampled or *encoded* by an $M \times N$ linear operator $\mathbf{\Phi}$ (the *sampling matrix*) to produce samples $\mathbf{y} \in \mathbb{R}^M$. When $\mathbf{\Phi}$ is a random matrix and $M \geq O(k \log(N/k))$, with high probability, $\mathbf{y}$ can be *decoded* to exactly recover $\mathbf{s}$ by solving

$$\min_{\mathbf{s} \in \mathbb{R}^N} \|\mathbf{s}\|_1 \quad \text{subject to} \quad \mathbf{y} = \mathbf{\Phi}\mathbf{s}, \tag{1}$$

even though the system is underdetermined ($M < N$).

UpShift relies on a novel application of compressive sensing to achieve the fast and network-efficient memory replication needed to support compressive offloading. *The key insight is that memory I/O (i.e., deltas to memory) typically constitutes a sparse signal that can be compressively sampled.* This approach, which we call *compressive replication*, has two major advantages: (1) it requires no communication cost to determine the deltas and their offsets because these are automatically recovered during decoding; and (2) it is resource-commensurate because the encoder on the mobile device has low computational complexity while the decoder on the cloud server has higher complexity.

On system startup, we assume that both the local and remote ends know the sampling matrix $\mathbf{\Phi}$ and that a length N memory page, with initial byte values represented by $\mathbf{s}_0$, is already synchronized; i.e., both local and remote can calculate $\mathbf{y}_0 = \mathbf{\Phi}\mathbf{s}_0$. At some later time, a process on the mobile device modifies the contents of the local memory page. When k elements have changed, we denote the state of the page as $\mathbf{s}_1$ and encode it by calculating $\mathbf{y}_1 = \mathbf{\Phi}\mathbf{s}_1$. This encoding is then transmitted over the network to the cloud server. On receipt, the cloud server calculates $\mathbf{y}' = \mathbf{y}_0 - \mathbf{y}_1$, which satisfies

$$\begin{aligned} \mathbf{y}' &= \mathbf{y}_0 - \mathbf{y}_1 \\ &= \mathbf{\Phi}\mathbf{s}_0 - \mathbf{\Phi}\mathbf{s}_1 \\ &= \mathbf{\Phi}(\mathbf{s}_0 - \mathbf{s}_1), \end{aligned} \tag{2}$$

where the unknown quantity $\mathbf{s}_0 - \mathbf{s}_1$ is the delta encoding we wish to recover. With high probability[1], this can be obtained from Equation 1 using convex optimization [1, 11] or greedy methods [20, 19, 7]. $\mathbf{s}_0 - \mathbf{s}_1$ can then be subtracted from $\mathbf{s}_0$ to obtain $\mathbf{s}_1$. By extension, a subsequent set i of k new updates to the local page will generate a new sample $\mathbf{y}_i$. Upon its receipt, the remote end calculates $\mathbf{y}_{i-1} - \mathbf{y}_i$ and applies the same decoding scheme above to recover $\mathbf{s}_i$.

Compressive replication and computation offloading are managed by two UpShift system components. On the mobile device, we introduce a shim layer into the runtime environment that (1) serves as the replication agent and (2) intercepts and redirects object method invocations to the cloud server. The replication agent operates continuously in the background, though updates can also be generated on-demand by the app via an API call to the shim layer. On the cloud server, a daemon (1) decodes and applies updates to its replica memory page(s) and (2) services the redirected object method invocations. In this early prototype, we disallow methods from modifying object state since this might occur on either the device or the server and would thus require bidirectional replication to enforce data consistency. For now, this simplification means we need only support unidirectional replication from device to server and simple versioning for each memory page (encoding the page increments the version). On the mobile device, the shim layer tags method invocations with the version at the time of invocation. At the daemon, redirected invocations and replica updates are queued and serviced in version order. Return values are passed back from the daemon to the shim layer and then finally to the app.

Our offloading mechanism shares some similarities with traditional RPC systems, except there is no object marshaling, which is typically slow and negatively impacts perceived system responsiveness. Since objects in memory are already replicated in the background, and since UpShift components control the entire replication and offloading process at both local and remote end-

[1] Exact recovery always occurs in practice when M is set properly.

points, we are able to pass pointers and perform address translation wherever appropriate.

The shim layer takes into account several cues when deciding when to offload. At the most basic level, offloading only occurs when proper network conditions permit. But the decision to offload can also be informed by the device's battery levels (offloading helps stretch the battery budget; see Section 4), as well as backpressure from the cloud service provider when its data centers are heavily loaded. Additionally, the end user can exercise direct control by either disallowing offloading altogether or forcing it to be "always on".

3 Prototype Implementation

The UpShift architecture described above is device-agnostic and can work with applications written in interpreted languages such as JavaScript or compiled languages such as Objective-C. We have implemented a prototype for the iOS ecosystem, using an iPad 3 running iOS 6.1.3 as the mobile device and an Amazon EC2 `g2.2xlarge` instance in us-east-1a running Ubuntu 12.04LTS as the cloud server. Here, we highlight some design choices for our prototype.

Offloading. Aside from having a rich ecosystem, we chose to target iOS because its lingua franca, Objective-C (a superset of C), allows us to perform our own memory management conveniently. The UpShift shim layer is implemented as a software library (`libupshift`) against which an iOS app links. The shim implements a memory manager which makes an initial block allocation out of the app's process heap and then privately manages this block as the UpShift memory page(s). Replicating this memory is possible because (1) modern ARM processors (e.g., the iPad 3's Cortex-A9) are bi-endian and are therefore byte order-compatible with x86 Amazon servers; and (2) we manage our own memory, so we have control over byte alignment and padding. At present, the prototype supports only pure Objective-C objects, which are allocated out of the UpShift memory by a call to `upshift_alloc` instead of the Objective-C root object's `alloc` method. In the future, we can transparently override the default `alloc` by using a replacement Objective-C category method. `libupshift` handles method invocation redirection at runtime via method swizzling: Objective-C is late-binding, so we can replace method implementations at runtime with a `libupshift` method that forwards the invocation over the network to the cloud daemon.

When an iOS app is compiled, any objects allocated with `upshift_alloc` are also cross-compiled for the Amazon EC2 environment. In our prototype system, we abstract app code requiring this cross-compiling into separate modules and perform the cross-compiling manually.[2] The resulting library is loaded by the daemon and provides class definitions for objects that are in the UpShift server's replica memory. Since Objective-C objects are actually just C `structs` under the covers, they can be made accessible on the daemon after address translation and pointer casting.

Replication. One major goal in building our prototype was determining the parameters that minimize replication latency. This is key since latency dictates the granularity of work we can offload. For example, if a replication update takes 5s to complete, then all workloads that complete in fewer than 5s would not benefit from offloading. In practice, the choice of the sampling matrix $\mathbf{\Phi}$ hugely impacts the encoding time on mobile device hardware (and thus the latency). We have found that a random partial discrete cosine transform (pDCT), i.e., a type-II DCT matrix with $N - M$ random rows deleted, performs best since it uses the FFT and is thus much faster than matrix multiplication on iPad hardware.

Similarly, we have found that an accelerated iterative hard thresholding (AIHT) [7] decoding algorithm offers the shortest decoding time, mainly because it eschews costly matrix factorizations at each iteration, unlike matching pursuit algorithms. To extract even greater decoding speed, we have implemented AIHT in CUDA in order to take advantage of GPU instances on Amazon EC2. This provides another attractive category of computations that cloud providers could use to improve utilization of their more expensive GPU hardware.

4 Evaluation

4.1 Resource Trade-Offs

The lower we drive replication latency, the wider the range of workload sizes that can be offloaded and the more responsive the system will feel. However, minimizing latency is not straightforward because its constituent parts—encoding/compression time, network transmission time, and decoding/decompression time—are not independent. For example, a faster encoding time can give worse compression and thus drive up network cost. This is the case with zlib [5] and snappy [2], two well-known compressors against which we compare pDCT encoding. Our comparison has some nuance: pDCT is tantamout to compressing a delta encoding, whereas zlib/snappy are just general data compressors. Were we to require zlib/snappy to also compress a delta encoding, we would incur additional computation and network costs to generate it first (e.g., via rsync). Instead,

[2] Ultimately, when both device and cloud server use the same hardware architecture (e.g., ARM or x86), these extra steps can be skipped.

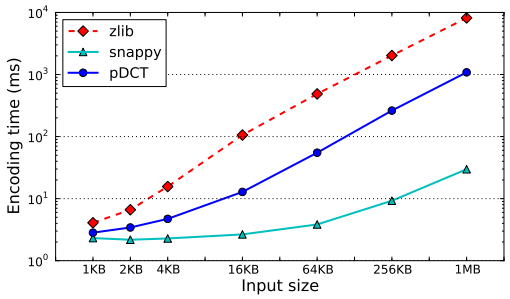


Figure 2: Mean encoding time for memory pages of various sizes on an iPad 3 using zlib, snappy, and pDCT.

we replicate a memory page with zlib/snappy by simply compressing the entire page on-device and then overwriting its server replica. This comparison is fair in that no costs beyond generating and sending the encodings are incurred by any of the three candidate methods.

Figure 2 shows the mean encoding time of zlib, snappy, and pDCT on an iPad for various memory page sizes N, when $k/N = 0.01$ (i.e., 1% of the page has changed). For pDCT, we took $M = 5.5k$ samples, a conservative number that guarantees recovery. snappy encoding is fastest, and zlib is slowest, with pDCT falling in between. For example, when $N = 64$KB, snappy requires 4ms, zlib 487ms, and pDCT 53ms.[3] snappy is particularly fast because it is a byte-level LZ77-type compressor, whereas zlib includes computationally expensive bit-level Huffman coding. pDCT, in contrast, is largely dominated by a single pass of the FFT.

Next, we consider decoding time. Recall that compressive replication trades a lower complexity encoder for a higher complexity decoder. Whereas zlib and snappy have negligible decoding times on an Amazon server, pDCT decoding takes on average 70ms for $N = 64$KB. Table 1 summarizes the average total latency estimates for snappy, zlib and pDCT, assuming a 54Mbps 802.11g uplink and a 10ms one-way Internet routing delay per 1500-byte packet from the iPad to our Amazon server. While snappy indeed gives the lowest total latency, pDCT nonetheless beats zlib handily, while providing the best compression ratio overall.[4] The compression gain over snappy is most significant: bandwidth utilization is reduced some 52% while giving up only 116ms in latency. Thus, pDCT gives the best latency/compression trade-off because its 135ms replication latency is low enough to admit even the shortest workloads for which users might perceive a benefit from offloading (user-perceptible application response time is ∼100ms [18]), while its bandwidth cost is lowest overall.

[3]Hereafter, we use $N = 64$KB since it is a reasonable page size and gives fair encoding and decoding times across all methods.

[4]To be clear, pDCT outperforms because it encodes and compresses the deltas, whereas zlib and snappy must compress the entire data block (they cannot obtain the deltas without additional costs).

	Enc	Network	Dec	Tot	CR	Size
snappy	4	15	-	19	3.8:1	17.2
zlib	487	13	-	500	6.0:1	10.9
pDCT	53	12	70	135	7.3:1	9.0

Table 1: Breakdown of average-case update latency (in ms) of replicating a 64KB memory page, assuming a 54Mbps 802.11g uplink and a 10ms one-way routing delay. Also shown is the compression ratio (CR) and size in KB of an update for each scheme. Metrics are averaged over 1,000 trials.

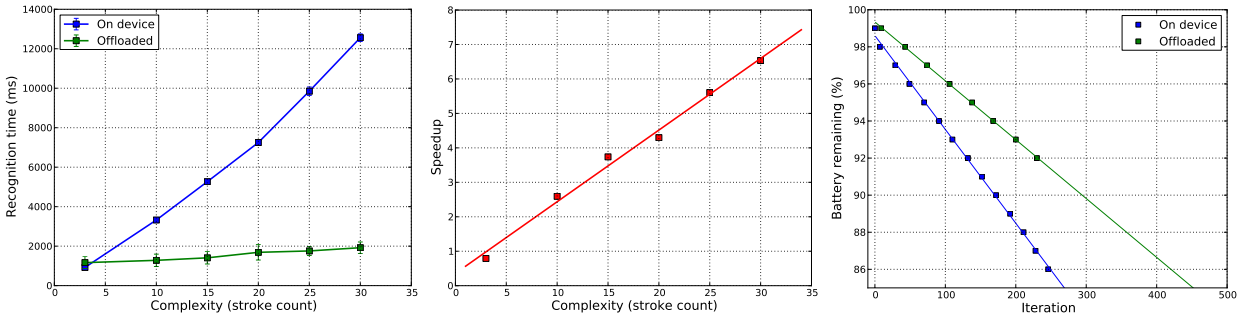
Yet another advantage of pDCT is that the compression ratio for each replication update is fixed and predictable (i.e., determined by M), whereas those given by zlib or snappy depend on the Kolmogorov complexity of the input and can therefore vary widely across updates. Compressive replication thus not only consumes less bandwidth, but also produces a stabler traffic stream, which is friendlier to other network applications. Furthermore, since pDCT is a sampling operation, it can encode high Kolmogorov complexity inputs that would confound zlib and snappy, e.g., when the memory page holds an array of floating point numbers. This is an increasingly common scenario, as data generated by device-integrated sensors are held in-memory during costly (and thus offloadable) signal processing routines.

4.2 Performance Gains

To demonstrate that our prototype system produces practical performance gains, we implemented an example iOS application that performs handwriting recognition of Chinese characters. We chose this problem domain because each character has a prescribed number of strokes, providing us a simple, quantifiable measure of the complexity of each recognition task.

Our app is based on the open source Zinnia/Tegaki [4, 3] projects, which provide a trained support vector machine model for recognizing traditional Chinese characters. The user handwrites a character on the tablet screen and the app captures the strokes as a list of stroke vectors in an in-memory object. These stroke vectors are then used by a model evaluation object method to produce a classification that results in a Unicode character being returned. Each written stroke causes the stroke vectors to grow or their values to change, giving rise to deltas to the memory occupied by the object. When the stroke vector object is `upshift_alloc`'d, our platform takes care of replicating these deltas and offloading the object's model evaluation method to the cloud server.

We compared the time required to recognize handwritten characters of increasing complexity locally on the



(a) Recognition Time (b) Speedup (c) Battery Depletion

Figure 3: Comparison of handwriting recognition performed on an iPad 3 (blue) vs. offloaded to an Amazon server (green). (a) Recognition time on-device increases sharply with increased problem complexity, but remains relatively flat when offloaded. (b) Speedup due to offloading increases with problem complexity. This is another view of (a). (c) Battery depletion rates when recognizing handwritten characters of high complexity (25 strokes). Overall, with the same battery budget, compressive offloading allows 60% more work to be performed.

iPad vs. offloaded to a cloud server. The offloading experiments were performed in a typical office environment in New York City, where the iPad was a client on an 802.11g Wi-Fi network with an average measured round trip time of 19ms to the Amazon server. Similar to the experiments in Section 4.1, we set pDCT parameters to $N = 64$KB, $k/N = 0.01$, and $M = 7k$, which gave similar replication latencies to Table 1. When we compare the on-device (blue) and offloaded (green) recognition times in Figure 3a, we find that on-device recognition time scales poorly with complexity; as character complexity increases 10-fold (from a stroke count of 3 to 30), the average on-device recognition time increases 13.62-fold. In contrast, the increase is just 1.65-fold under offloading. This is because the low overhead of compressive replication (Section 4.1) allows us to make effective use of the raw computing power of the server.

How much speedup can a user expect from compressive offloading? Even for moderately complex 20-stroke characters, the on-device recognition time averages 7,249ms; compressive offloading averages just 1,687ms, a substantial 4.2-fold speedup. Better still, the acceleration increases as the complexity increases, as shown in Figure 3b. For highest-complexity, 30-stroke characters, the speedup due to offloading is more than 6.5-fold (on-device: 12,560ms; offloaded: 1,922ms). The difference to the app user would be striking, especially when more than one character must be recognized at a time (e.g., in a tract of handwritten text).

While the performance acceleration is substantial, achieving it cannot be at the expense of increased battery consumption. We thus evaluated the battery efficiency of compressive offloading, taking into account the power drawn for computing the encoding and transmitting it over Wi-Fi. Figure 3c compares battery depletion over 250 consecutive high-complexity recognition tasks run on-device (blue) vs. offloaded (green). With compressive offloading, the battery depletion rate is reduced substantially. In fact, as the linear regression lines show, with the same battery budget, compressive offloading allows the user to perform 60% more recognition tasks.

Taken together, these results show that compressive offloading can provide end users with significant advantages in both speed and battery efficiency in real-world mobile apps. For the cloud provider, computations that take the iPad an excruciatingly long 10 seconds to execute take barely a few hundred milliseconds. At scale, these small workloads can be load-balanced to fill slack anywhere in the data center.

5 Conclusions & Future Work

In this paper, we have described a proof-of-concept platform for dynamically offloading arbitrary, fine-grain workloads from mobile devices to the cloud. At a high level, we believe that this case study points the way towards a tighter integration between mobile devices and cloud resources, possibly at the level of the operating system. This is a particularly exciting near-term possibility in light of the emerging trend of ARM-based data centers. Since the overwhelming majority of mobile devices are also ARM-based, there is the potential to greatly reduce the friction of directly offloading machine code to the cloud. Compressive offloading can play a critical role in such future systems because it provides a principled way to do this transparently and efficiently.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. IIP-1315617.

References

[1] ℓ_1-magic. http://users.ece.gatech.edu/∼justin/l1magic/.

[2] snappy: A fast compressor/decompressor.

[3] Tegaki project: Open-source chinese and japanese handwriting recognition.

[4] Zinnia: Online hand recognition system with machine learning.

[5] zlib technical details.

[6] BERSHAD, B., ZEKAUSKAS, M., AND SAWDON, W. The midway distributed shared memory system. In *Compcon* (1993), IEEE.

[7] BLUMENSATH, T. Accelerated iterative hard thresholding. *Signal Processing 92*, 3 (2012), 752–756.

[8] CANDÈS, E. J., ROMBERG, J., AND TAO, T. Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information. *IEEE Transactions on Information Theory 52*, 2 (2006), 489–509.

[9] CANDÈS, E. J., AND WAKIN, M. B. An Introduction To Compressive Sampling. *IEEE Sig. Proc. Mag. 25*, 2 (2008), 21–30.

[10] CARTER, J. B., BENNETT, J. K., AND ZWAENEPOEL, W. Implementation and performance of munin. *SIGOPS Oper. Syst. Rev. 25*, 5 (Sept. 1991), 152–164.

[11] CHEN, S., DONOHO, D. L., AND SAUNDERS, M. A. Atomic decomposition by basis pursuit. *SAIM Journal of Scientific Computing 20*, 1 (1998), 33–61.

[12] CHUN, B.-G., IHM, S., MANIATIS, P., NAIK, M., AND PATTI, A. Clonecloud: elastic execution between mobile device and cloud. In *EuroSys* (2011).

[13] CUERVO, E., BALASUBRAMANIAN, A., CHO, D., WOLMAN, A., SAROIU, S., CHANDRA, R., AND BAHL, P. Maui: Making smartphones last longer with code offload. In *MobiSys* (2010).

[14] DONOHO, D. L. Compressed sensing. *IEEE Transactions on Information Theory 52*, 4 (2006), 1289–1306.

[15] GORDON, M., JAMSHIDI, D., MAHLKE, S., MAO, Z., AND CHEN, X. COMET: Code Offload by Migrating Execution Transparently. In *OSDI* (2012).

[16] GREENBERG, A., HAMILTON, J., MALTZ, D. A., AND PATEL, P. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM Computer Communication Review 39*, 1 (2008), 68–73.

[17] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst. 7*, 4 (Nov. 1989), 321–359.

[18] MILLER, R. B. Response time in man-computer conversational transactions. In *Proc. AFIPS Fall Joint Computer Conference* (1968), ACM, pp. 267–277.

[19] NEEDELL, D., AND TROPP, J. Cosamp: Iterative signal recovery from incomplete and inaccurate samples. *Applied and Computational Harmonic Analysis 26*, 3 (2009), 301–321.

[20] TROPP, J., AND GILBERT, A. Signal recovery from random measurements via orthogonal matching pursuit. *IEEE Transactions on Information Theory 53*, 12 (Dec 2007), 4655–4666.

[21] WIESMANN, M., PEDONE, F., SCHIPER, A., KEMME, B., AND ALONSO, G. Database replication techniques: a three parameter classification. In *IEEE Symposium on Reliable Distributed Systems* (2000).