SLIM: Mmap from the Cloud to Device, and Back

Jinghao Shi° Mingyuan Xia† Ming Wu‡ Lintao Zhang‡ Zheng Zhang‡
° University of Science and Technology of China † McGill University, Canada ‡ Microsoft Research Asia
jhshi89@gmail.com mingyuan.xia@mail.mcgill.ca
{miw,lintaoz,zzhang}@microsoft.com

ABSTRACT

In the era of cloud computing, mobile applications often need to leverage a new storage hierarchy that includes not only the traditional main memory and secondary storage on devices, but also storage and computation capabilities from the cloud. In this paper we propose a new data structure library called SLIM. SLIM provides familiar STL-like interfaces and abstractions while accommodates the storage hierarchy that transcends device/cloud boundary. Initial evaluation shows that using SLIM can greatly simplify application development and reduce network and energy cost compared with traditional approaches.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Client/server*; C.4 [**Performance of Systems**]: Design studies; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*

General Terms

Design, Performance

Keywords

Mobile, cloud, data structure, STL, storage hierarchy

1. INTRODUCTION

More computing devices are becoming mobile. These devices derive much of their values by being connected not only to each other but also to the cloud. Leveraging the cloud for mobile devices is becoming an important research topic. Broadly speaking, cloud is seen not only as a data source and sink, but increasingly as computation backbone as well.

Splitting computation across device and cloud boundary is complex and challenging. It touches on many known hard problems in distributed computing, including state maintenance, heterogeneous computing environment and programming model. The common theme in much of the previous work is to preserve legacy compatibility as much as possible.

However, interesting mobile applications often starts anew on the mobile devices, whereas time-tested important desktop applications (e.g. email and browser) are worth the effort to be rewritten. Therefore, we do not believe that preserving backward compatibility is a pressing issue. The SLIM project is motivated by the observation that programming abstraction evolves much slower than the hardware trend. We argue that deriving program logic from data structures is a "constant" despite hardware and software changes. Indeed, modulo various marshaling and serialization steps in between, mobile applications all start their life as data structures manipulated by logic in the cloud and end up as data structures on the devices.

We attempt to re-examine the notion of data structures in light of the new storage hierarchy. Mosting existing data structure libraries like STL (Standard Template Library [12]) considers only a single layer: the volatile main memory. In the mobile world, there are actually two more layers of storage: flash and the cloud. The characteristics of these three layers are markedly different. The main Memory is fast but volatile, flash is large and persistent, while cloud has infinite computing and storage capacity (with respect to the device), but connectivity to it can be intermittent and unreliable. The goal of the SLIM project is to investigate the feasibility of building a high performing set of data structures that transcend these boundaries while preserving the semantics of existing libraries (e.g. STL) as much as possible.

Our contributions include:

- We propose an STL-like data structure abstraction for mobile/cloud application development. While preserving most of the STL abstractions and interfaces, SLIM gives programmers simple directives to explicity control the data structures such as the storage layers involved, the data flow direction of updates, and the consistency bounds.
- We propose several techniques to make SLIM efficient and easy to use. SLIM employs an efficient pointer swizzling algorithm to handle the pointer-rich nature of data structures across memory hierarchy boundaries. To deal with the unreliable nature of communication between cloud and device, we leverage advanced asynchronous programming model called "promoise" to simplify failure handling for the programmers.
- By building an RSS reader as an example, we demonstrate that SLIM data structures provide a performance benefit, and have the potential to enable computation offloading to the cloud.

The rest of the paper is organized as follows. Section 2 describes the programming model. Section 3 gives a high-level overview of the system. Section 4 presents preliminary results. Section 5 and 6 covers related work, discussion and future work.

2. THE PROGRAMMING MODEL

The goal of the SLIM family of data structures is to create as close as possible the illusion of dealing with the traditional and local STL data structures, whereas the runtime hides the interactions with the memory hierarchy. To begin the discussion, we will start with a piece of pseudo-code for a RSS reader using the STL library, as follows:

```
//Step1: fetch RSS xml file to local
XMLObject xmlObj = XML::Load("RSS URL");
//Step2. parse XML to construct feed array
std::vector<FeedItem> feeds = parse(xmlObj);
//Step3. find and display new feeds
foreach (feed in feeds) {
    if (/*feed is new*/)
        display(feed);
}
Sleep(freshInterval);
```

The code snippet above retrieves an XML object that contains the content of the current feeds, and then parse it into an STL vector for display. The reader is stateless, as such redundant feeds must be removed. The corresponding SLIM version is shown below:

```
//SLIM vector of all feeds, parsed and to be updated by the cloud
SLIMVector<FeedItem, L3, C2D, OnDirtyBound<1> > feeds("RSS URL
    ");
//set callback function for new feeds (use lambda function for simplicity)
feeds.onUpdate([] (FeedItem newFeeds[]) {
    //all new feeds, just display
    foreach(feed in newFeeds)
        display(feed);
});
```

A SLIM data structure takes three directives at the declaration time to tell the runtime the desired behavior of interacting with multiple storage layers: `Level`, `Direction` and `Consistency`:

- `Level` can be L1, L2 and L3. An L1 vector is exactly the same as its volatile in-memory STL counterpart. An L2 vector corresponds to the external-memory model [13] where the local persistent storage (usually Flash) is included as the bottom layer. L3 indicates that the cloud is the final layer, which is the case for the SLIM-based RSS reader. In our current design, the layers are inclusive. In other words, an L3 SLIM vector also can store data in local Flash storage if it cannot all fit in memory. We do not see immediate need to exclude local secondary storage explicitly in the L3 case, though implementing such a policy is trivial.

- `Direction` expresses the data flow direction for updates. `D2C` is for cases where the updates are coming from devices towards the cloud, and is useful for scenarios such as the outgoing queue for an Instant Messenger client, or a vector that stores data obtained by a sensor node for cloud to analyze. `C2D` is the opposite of D2C, where updates are first collected in the cloud, and then pushed to the device. This is the directive that the sample RSS reader uses. Finally, we reserve `DandC` in case the data structure might be updated by both the device and the cloud. Resolving potential update conflicts in general is tricky. Therefore, currently we do not allow DandC in SLIM. However, DandC could be useful in practice. For example, it is needed if we model an email folder as a vector of individual email messages, where user can delete emails while newly received emails may be added by the cloud. We are currently experimenting with different designs and the DandC scheme may be supported in future work.

- `Consistency` gives a few options as when the updates should show up at the other end. Currently we support three options: `OnTimeBound` and `OnDirtyBound` are bound by refreshing cycles and number of dirty items, respectively. Changing the consistency directive in the above sample code to `OnTimeBound` with the parameter `freshInterval` would make our RSS reader refreshing in the same manner as the STL-based RSS reader. Finally, `OnDemand` simply forces the update at user's chosen point.

These set of directives transform the stateless implementation of the RSS reader into an event-driven one, where the logic to display the feeds is triggered by incoming new items. Notably, the logic of parsing the XML object is no longer needed on the device side. This logic is instead residing on the cloud side. The cloud will perform the parsing and generation of the native vector that is now mapped onto the device. This leads to both network and computation savings (Section 4) with little additional programming effort.

Multiple data structures can be linked together to enable rich functionality. Indeed, a generic mailbox will not only include the main vector to contain email bodies but also multiple sorted vectors on various attributes, as well as index that allow search and lookup. In the RSS example, the feeds can derive an index in the form of a map at the cloud, and we can then map it into the client, allowing keyword-based search:

```
typedef SLIMVector<PolyPointer<FeedItem>, L3, C2D, OnDirtyBound
    <1> > FeedIndex;
//cloud—built inverted index on words, derived from the feeds vector
SLIMMap<string, FeedIndex, L3, C2D, OnDirtyBound<1> > index("
    RSS URL/index");
Promise<FeedIndex> resultPromise = index[key];
resultPromise.continueWith([] (FeedIndex result) {
    //display the search result for the key
    foreach(feed in result)
        display(*feed);
}).ignore();
```

The code snippet above relies on an L3 inverted index that is built at the cloud, each entry of the index contains pointers to the feeds containing the keyword in question. The code contains two new notions. The first is *PolyPointer*, which is a mechanism to allow pointer access across memory hierarchies (including the cloud), and will be explained in detail in Section 3. The second is *promise* [5].

The rationale of using promise here is that including cloud as a memory hierarchy introduces the unreliable network connectivity. Completely hiding this from programmer is impossible, as this moves too much complexity into the runtime. One significant complexity stems from the fact that, since SLIM is a family of generic data structures, its instance can contain pointers. As such it is possible that the target of the pointer is absent from the device and reside in the cloud at the time of pointer dereferencing. To alleviate this problem, we borrow the interface of promise to deal with asynchronous programming.

An instance of promise represents a future result value from an operation. Initially, the promise is returned by certain operation immediately (such as the `operator[]` in our case) in the *unresolved* state, expecting to receive a result at some unspecified future time. When the result is received, the promise becomes *fulfilled* and the result becomes the value of the promise. Promise will then call the continuation routine bound by the `continueWith` interface. In the above case, we bind a lambda function to deal with the resolved result value. If an error occurs, either in the calculation or network transmission, the promise becomes *broken*. User can either attach an exception routine to handle the error or ignore the exception, as we have done with the `ignore()` interface in the example code. Note that if the entries of the index, as part of the L3 SLIMMap, are available locally, the query will be satisfied immediately.

3. SYSTEM DESIGN

This section focuses on the design of the SLIM vector. Just like that of a STL vector, an element in a SLIM vector can be any fixed size data, and can even contain pointers. We omit the details of L1 SLIM vector, as it is simply a regular STL vector that provides an in-memory dynamic array of data items.

Even though we use vector as a representative data structure to describe SLIM, SLIM actually supports a wide variety of other generic data structures. Data structures such as map and linked list can be easily implemented on top of vectors since vector can be simply regarded as a continuous block of memory. Data structures build on top of a vector can also inherit the same set of policies of the base vector data structure they are based upon. However, this is often not optimal or even correct. For example, when a new item is inserted into a linked list that has a consistency policy of 1-dirty bound, we should send the update after the entry itself and both its previous and next entries are updated.

3.1 L2 SLIM vector

An L2 SLIM vector follows the external-memory model [13], in that its total size can be greater than its allocated memory. The user of the L2 vector doesn't need to explicitly manage memory space, with the trade-off of some performance overhead. Our goal is to maximize the performance such that the operations over the L2 vector approaches that of L1 when hit memory.

SLIM vectors are logically laid out in a non-overlapping 64bit *universal address* space. A SLIM vector occupies a continuous range in the space, logically with a starting pointer and an end pointer. Opaque to the programmer, a SLIM vector is located with an internal root pointer. The root pointer must be persistent and available during the entire lifetime of the vector. An L2 vector is given a pool of memory pages at its birth, but that pool may shrink or grow transparent to the programmer.

The most significant complexity arises because of pointers: a user can instantiate a pointer to walk through the vector in arbitrary fashion. Likewise, internal operations such as push_back relies on pointer as well. An L2 vector, by definition, is not guaranteed to have all of its elements memory resident.

For efficiency reason, a pointer in SLIM should contain the in-memory address of the point-to object (in our case, another SLIM vector) if the pointed-to object is in memory. Like normal C++ pointer, the content of the L2 pointer should allow direct access to memory address, instead of relying on extra translation of the logical universal address each time. The process of translating a universal pointer to a local address is commonly known as *pointer swizzling*. Similarly, *unswizzling* is the reverse process to make sure that direct access is disallowed, when the point-to object has been removed from memory. Enabling these two processes takes extra bookkeeping. There is a rich body of literature on pointer swizzling, we refer reader to [9] which classifies different proposals along the dimensions of the timing when swizzling occurs and the behavior when an object is displaced from memory.

A pointer in SLIM is a *PolyPointer* which, along with other meta data such as size and capacity of the vector it points to, includes two fields, UA and MA, for universal address and physical memory address, respectively. The state machine is depicted in Fig. 1, assisted by two auxiliary SLIM-internal data structures. The first is the PPT (PolyPointer Table) that records the MA-to-UA mapping, with the granularity of a page (typically 4KB). Given a memory address, PPT allows direct inspection of the UA page of the corresponding memory page. The second data structure is RHT (Reverse Hash Table) that enables the reverse lookup, returning the MA for a given UA. These two tables are always updated atomically to ensure their

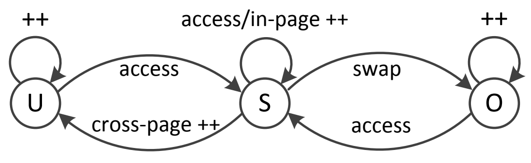

Figure 1: PolyPointer state machine. Pointer primitives that may cause state transition include self-increment (++), dereference(access) and swapping. Also an in-page self-increment is also distinguished from a cross-page one.

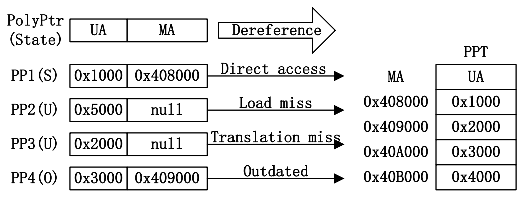

Figure 2: Four possible events upon the dereference of PolyPointers. Dereferencing unswizzled PolyPointers (its MA field is null) will raise a translation miss to fill the MA or a load miss if the UA is also not available. Swizzled PolyPointer will have a registered UA value in the PPT, which matches its MA field. To identify an outdated PolyPointer, SLIM checks if the UA field equals the value stored in the PPT entry that maps the MA.

mutual consistency. RHT also records the disk location for pages that are not swapped out. RHT is persistent, whereas PPT can be derived from RHT.

A new PolyPointer is initialized with a universal address, but its MA is NULL. Such a PolyPointer is in state *Unswizzled* (shown as "U" in Fig. 1), dereferencing such a pointer causes a translation miss. Handling such misses can follow a few different paths. Assuming that the allocated memory has been used up. One of the used pages will be evicted, written to disk if dirty. The PPT entry now records UA of the dereferenced PolyPointer, the RHT is updated, and the MA field is calculated from the PPT entry position and the offset in the page (Fig. 2). It is also possible that another PolyPointer within the same UA page suffers a miss as well. In this case, the PPT entry already exists, and the only thing needs to be done is to update the MA field of that PolyPointer. In this and the earlier case, the pointer now becomes *Swizzled* (shown as "S" in Fig. 1). A Swizzled PolyPointer can become Unswizzled, this happens if it steps beyond a page boundary. The next dereferencing will suffer a translation miss (and a possible load miss).

If a target memory is replaced, the corresponding PPT entry will record the UA of a different PolyPointer. All PolyPointers whose MAs point to this entry without the matching UA are now in the state *Outdated* (shown as "O" in Fig. 1). There are several approaches to discover and fix a PolyPointer in this state. For instance, the state can be explicitly changed to *Unswizzled* at the time of replacement, requiring either a reverse mapping table or scanning.

In the current implementation, we leave the pointer unchanged, but require consulting PPT at the time of dereferencing the pointer. This strategy works well if vector contains a large amount of data in its body, so that such overhead is amortized. Handling load miss is straightforward, RHT is first consulted with the requesting UA to locate the page off the stable storage, and PPT is then updated to record the memory page that the page is loaded into.

We set an upper limit on the secondary storage that can be used by SLIM. Of course, potentially Flash may fail or may run out of space. We handle these by throwing exceptions, similar to traditional STL when it run out of memory.

3.2 L3 SLIM vector

We model L3 as a pair of synchronized L2 vectors, one at the device and one at the cloud. To keep it simple, our current design focuses on C2D and D2C, in which cases *application updates* happen only at one end of the pair, and are reflected to the other end with *SLIM updates*, based on its consistency policy. An update expands the *view*. The one receives application updates is called the *master view*, whereas the one receives updates is called the *slave view*. For the time being we do not consider the shrinking of a view (i.e. delete) by simply tag the deleted entry rather than removing it.

Updates as well as load miss at the device must be resolved with the same universal address. Therefore in L3 the management of the universal address space is in the cloud, independent of where the addresses are consumed. In the case of D2C, the device will ask a chunk of space from the cloud, and then assign them at the device when it receives application updates. The corresponding SLIM update will include the universal address that was applied in the device.

Handling SLIM updates can take different approaches. Our current implementation records application updates in a log, and ship the log to be replayed at the other end. An alternative is to perform updates through a shared key-value store, where multiple updates can be merged. Some scenarios may require the instantiation of a new SLIM vector in one shot. For instance, if a vector is to be sorted, it is easier to derive the sorted vector in the cloud and map it to device while destroying the old one, than updating the entries as the sorting is progressing.

We require all data in the view to be available at the cloud. This is justified because storage is cheap in the cloud, and some of the data needs to be kept indefinitely anyway (e.g. gmail). Since cloud has all the data, it is possible for the device to freely reclaim storage resources as it sees fit. The only additional bookkeeping is to mark in RHT that the page is now accessible only from the cloud. When the device suffers a load miss, the universal address enclosed in the request allows the cloud to retrieve the data.

Cloud may issue update events when the device is accessing the data structure and vice versa, thus causing race conditions. SLIM allows user to specify policy to coordinate concurrent operations when update arrives while the user is accessing the data. The user can choose a more restricted access policy using critical sections, or more relaxed policy as long as the application semantic can tolerate benign races. One such case is when the application loops through the vector while SLIM updates adds new elements at the tail.

4. EVALUATION

In this section we report some initial results of a prototype SLIM implementation. We first use a micro benchmark to evaluate the performance of SLIM vector when no network traffic is incurred, to validate our pointer swizzling algorithm and measure its overhead. We then conducted a case study using RSS reader as an example to demonstrate the advantage of SLIM when applied in real applications. All the experiments are performed on a Windows 7 machine with 2.13GHz Intel Core2, 2GB main memory, and 1Gb ethernet.

4.1 Microbenchmark

We measured the performance of SLIM vector under microbenchmarks with sequential read/write, random read/write, and push_back operations. We compared it with the performance of regular STL vector under the same microbenchmarks. The push_back benchmark appends elements one by one to the tail of the vector until the total data size reaches 512KB. Other benchmarks perform 512K operations, each for one element, on an initialized vector with 512KB of total data. In the experiments, we configured SLIM vector to be L3. In order to avoid network traffic during the measurement, all the data in the vector were touched once to prime the data into main memory, and we set the consistency policy to OnDemand so that updates will not propagate to the cloud.

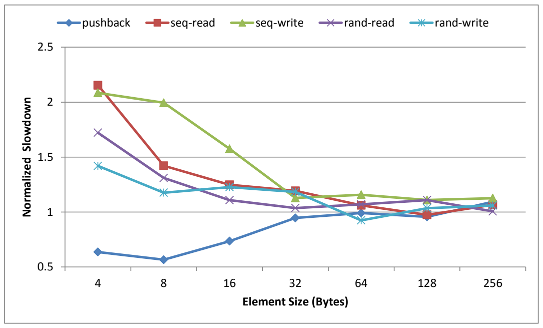

Figure 3: Microbenchmark results on SLIM vector primitives, normalized to regular vector performance.

Fig.3 reports the normalized performance of SLIM vector relative to that of regular STL vector. The x-axis represents the element size of the vector in bytes. As expected, SLIM vector generally performs worse than STL due to the inherent overhead[1]. As shown in the figure, the larger the element size, the smaller the performance difference between SLIM and STL. As elements get larger, the cost of the operations on the payload becomes larger, and hence the relative cost for maintaining the states of the SLIM metadata becomes smaller. When element size is larger than 32 bytes, the performance difference between SLIM and STL vectors ($\sim$5%) is no longer significant. In many real world applications that share data between cloud and devices, element size is often relatively large (e.g. an RSS feed, or an email message). We believe that for many such real world scenarios, the overhead of SLIM vectors relative to STL is insignificant.

4.2 Case Study on RSS Reader

We implemented an RSS reader prototype using SLIM vector. It essentially is a traditional RSS reader partitioned into a cloud part and a device part. The cloud part fetches XML files from the original RSS content provider, parses them into feed items and stores the feeds in a SLIM vector. The device part directly accesses the SLIM vector to get the updated RSS feeds. There are two major benefits for RSS reader implemented this way: 1) the job for parsing XML files is migrated to the cloud, which saves energy in the device side; 2) the network traffic is also greatly reduced because there is no duplicated feed items in the SLIM vector, and hence only the newly updated feeds need to be transferred. In contrast, a traditional RSS reader needs to fetch the entire XML file, which often contains old feed items.

To measure the computation savings, we profiled the execution of XSD [2], an open source XML parser written in C++, for parsing an XML file from Yahoo News [3] which contains 20 feed items. The parsing task takes about 24 million processor instructions. The total overhead is proportional to the number of times the feed is redundantly transfered in the stateless implementation.

[1]An exception is the push_back benchmark, which is caused by more aggressive function inlining by the compiler for SLIM

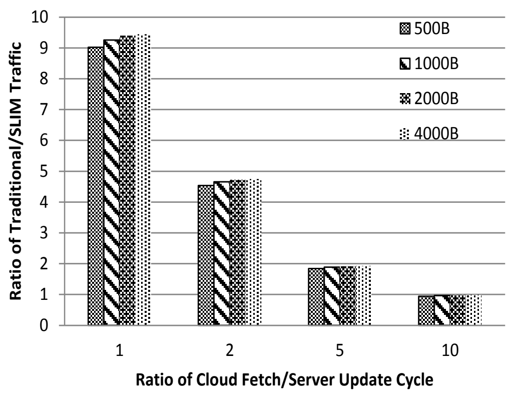

Figure 4: Comparison of network traffic between SLIM and normal RSS reader. The traffic is normalized to SLIM reader. The feed item size varies from 500B to 4000B

We also evaluated the device traffic saving compared to the traditional RSS reader. In the experiment, we built an RSS provider that maintains 100 channels and produces new feed items for each channel periodically and notifies the subscriber. Each channel organizes the latest 10 feed items in an XML file. For SLIM RSS reader, we measured the network traffic between the device and the cloud, while for the traditional one, we just measured the network traffic between the SLIM cloud service and the RSS content provider, since the cloud service itself requires the same traffic as the traditional RSS reader.

Fig. 4 shows the results. We change the number of newly generated feed items that triggers one notification to the subscriber (as shown in the x axis). The y axis represents the ratio between the network traffic of the traditional and the SLIM RSS reader.

In most cases, SLIM RSS reader's network traffic is much smaller than the traditional one. This is because SLIM breaks down the granularity of network data transfer from XML file to feed item, thus device only needs to fetch new feed items. In particular, if the cloud fetches the XML file every time when a new feed is generated, the traditional RSS reader's network traffic is more than 9 times of SLIM's. On the other extreme, when cloud service fetches XML file only when all its contents are updated, the traffic is about the same because SLIM RSS reader's meta-data traffic overhead is no more than the traditional RSS reader's update notification traffic.

5. RELATED WORK

Mobile computing leveraging cloud has been a extremely popular topic, and we can only discuss a few works with limited space. One dimension to compare is the granularity and partition point of device/cloud computation. Existing proposals include virtual machine (Internet Suspend/Resume [1] and CloudLet [11]), method-/procedure call (Cyber foraging [4], CloneCloud [6] and MAUI [7]), and explicit event such as tuple space [10]. SLIM adds to this landscape with the data structure angle.

Extending memory hierarchy beyond the single main memory hierarchy is an old idea. Still, the external memory model [13] and existing implementations such as STXXL [8], usually include secondary storage only. Our design leverages and extends the pointer swizzling technology across all three levels of memory hierarchy, transcend the device and cloud boundary. We also borrow the latest advancement of asynchronous programming to deal with network unreliability.

6. DISCUSSION AND FUTURE WORK

Instead of building a monolithic application, SLIM requires the application to be partitioned into a cloud and a device parts which interact with each other by sharing the SLIM data structures. In our prototype, the partition is done manually. For future work, we are looking at building a SLIM-aware compiler that can automatically partition the code with the help of some user annotations.

Currently, SLIM is implemented as a library and is linked into the applications. Therefore, its allocation policies are managed in a per-application basis. It is potentially beneficial to build SLIM as a middleware and being managed by the device OS. The flexibility of freely modifying memory allocation transparently to the application affords the possibility of controlling and prioritizing memory usage in mobile device depending on application states (e.g. foreground or background) and system requirements (e.g. shutting down part of the main memory to get into low energy mode).

In this paper, we assume the cloud is reliable and it holds the "golden" copy of the application state. In reality, a machine in the cloud can fail, and the process in the cloud may lose its in-memory state. It is important that the clould part can be restarted and quickly recover to a consistent state with the device. For future work, we plan to add automatic logging of SLIM data structure updates to the cloud. The log needs to be stored in a reliable cloud file system so that individual machine failure will not cause the loss of user data.

Our initial evaluation is encouraging. However, significant work remain to fully demonstrate the power and validate the vision. To be pragmatic, we plan to examine the design of a few demanding applications, including browser, file system and mailbox, each of which may contain multiple data structures. This exercise can then guide the building of the SLIM library. At the system level, an interesting direction is to install policies to control the footprint of various data structures so as to minimize energy consumption.

References

[1] The internet suspend/resume project. http://isr.cmu.edu/.

[2] Xsd. http://www.codesynthesis.com/products/xsd/.

[3] Yahoo news. http://rss.news.yahoo.com/rss/topstories.

[4] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H. i Yang. The case for cyber foraging. In *the 10th ACM SIGOPS European Workshop*, pages 87–92. ACM Press, 2002.

[5] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin. Orleans: A Framework for Cloud Computing. Technical report, Technical Report MSRTR-2010-159, Microsoft Research, 2010.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the 6th European conference on Computer systems*, EuroSys '11, New York, NY, USA, 2011. ACM.

[7] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[8] R. Dementiev and L. Kettner. Stxxl: Standard template library for xxl data sets. In *In: Proc. of ESA 2005. Volume 3669 of LNCS*, pages 640–651. Springer, 2005.

[9] A. Kemper and D. Kossmann. Adaptable pointer swizzling strategies in object bases: design, realization, and quantitative analysis. *The VLDB Journal*, 4:519–567, July 1995.

[10] A. L. Murphy, G. P. Picco, and G. catalin Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, 15:2006, 2006.

[11] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8:14–23, October 2009.

[12] A. Stepanov and M. Lee. The standard template library. Technical Report HPL-95-11(R.1), HP Laboratories, 1995.

[13] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33:209–271, June 2001.