Programming Support and Adaptive Checkpointing for High-throughput Data Services with Log-based Recovery

Jingyu Zhou
Computer Science Department
Shanghai Jiao Tong University
zhou-jy@cs.sjtu.edu.cn

Caijie Zhang
Google Inc.
Mountain View, CA 94043
caijiezhang@google.com

Hong Tang
Yahoo Inc.
Sunnyvale, CA 94089
htang@yahoo-inc.com

Jiesheng Wu
Microsoft
Redmond, WA 98052
jieshwu@microsoft.com

Tao Yang
University of California
Santa Barbara, CA 93106
tyang@cs.ucsb.edu

Abstract

Many applications in large-scale data mining and offline processing are organized as network services, running continuously or for a long period of time. To sustain high-throughput, these services often keep their data in memory, thus susceptible to failures. On the other hand, the availability requirement for these services is not as stringent as online services exposed to millions of users. But those data-intensive offline or mining applications do require data persistence to survive failures.

This paper presents programming and runtime support called SLACH *for building multi-threaded high-throughput persistent services. To keep in-memory objects persistent, SLACH employs application-assisted logging and checkpointing for log-based recovery while maximizing throughput and concurrency. SLACH adaptively adjusts checkpointing frequency based on log growth and throughput demand to balance between runtime overhead and recovery speed. This paper describes the design and API of SLACH, adaptive checkpoint control, and our experiences and experiments in using SLACH at Ask.com.*

1 Introduction

This paper studies programming support for a class of highly parallel data services, where in-memory states are frequently updated and retrieved, and these in-memory states must be kept persistent. Such services are typical and important for many data mining and offline applications at Google, Yahoo, Microsoft, Ask.com, and other Internet companies for document analysis, advertisement information mining, and user behavior studies. For example, in the offline system of Ask.com, web pages are crawled constantly and information regarding URLs is continuously updated. There are hundreds of application modules accessing various URL information services for data mining, URL string matching, URL name conversion, and property extraction. The traffic accessing such a service may reach hundreds of thousands of requests per second. Such a service can be unavailable for a short period of time during system upgrade or failure repair, but it must be available with high throughput for most of the time.

It is challenging to satisfy high throughput and data persistence at the same time for data services. Persistence can be achieved via data replication on multiple nodes [9, 11, 23] or log-based recovery [6, 7, 21]. For the class of applications we target at, fast memory access is required to deliver extremely high throughput, and log-based recovery is relatively cheaper to achieve persistence of in-memory states. This paper focuses on log-based recovery. Checkpointing [5, 15, 18, 22] can be used together with operation logging to generate a restorable execution point of a service so that old logs can be discarded. In practice, applications only need log and checkpoint a selected collection of data objects for critical recovery based on their domain-specific requirements. Programming high-throughput concurrent services with log-based recovery is complicated and our goal is to provide programming and system support that simplifies the integration of application-specific logging and checkpointing with maximized concurrency and performance.

The contribution of this work is in two areas. First, we present application-assisted programming and runtime support called SLACH to shield application programmers from the complexity of managing persistence of in-memory states. It supports and integrates previously proposed logging and checkpointing techniques for reliable production systems while considering high-throughput and application-specific flexibility in the design. SLACH is lightweight and can be integrated with legacy code using its narrow programming interface. Second, we propose an adaptive control scheme that adjusts checkpointing frequency based on log growth and throughput demand. Checkpointing can reduce the log size and shorten the recovery time, but checkpointing adds runtime overhead and reduces the system throughput. We consider a tradeoff between runtime service load and recovery speed for applications with high performance demands.

The rest of the paper is organized as follows. Section 2 discusses the background of our targeted applications and design considerations. Section 3 presents the design, system architecture, and API of SLACH. Section 4 describes an algorithm for adaptive frequency control for checkpointing. Section 5 describes two service deployments in the production system of Ask.com and a persistent key-value hash table (PHT) for in-core and out-of-core data. Section 6 presents experimental results. Sections 7 summarizes related work and concludes the paper.

2 Background and Design Considerations

We summarize the characteristics of targeted data services as follows.

Request driven: These data services adopt a request-driven model where a client sends requests to a server and the server returns a response after some processing. A server could handle requests issued concurrently from multiple clients.

High throughput and in-memory fast access: High throughput is often the most important requirement for some internal services in large-scale data mining and offline processing applications. To assist such applications, some of internal data services are required to deliver extremely high throughput and two such services are described in Section 5. As a result, data objects of our targeted applications are often kept in-memory for fast access.

High availability for most of the time: Offline applications such as ones at search engine companies can stop running for a few hours for software upgrade, hardware repairing, or data recovery. The availability of such an application is not as stringent as that for online services, but those data-intensive offline or mining applications need to accomplish on-schedule processing of a large amount of data. Therefore, targeted data services do require data persistence to survive failures.

In addition to serving read-only operations, many data services have frequent updates for a selected set of objects and these updates have to be stored on persistent media to survive crash failures. Once a failure occurs, the service state should be restored following consistent recovery [13], i.e., the restored state should be equivalent to that of a failure-free execution.

Data independence and object-oriented access model: Our targeted data services tend to host a large amount of independent data items. With these characteristics in mind, we target at partitionable data services in the sense that data manipulated by such a service can be divided into a large number of independent data partitions and each service access can be conducted independently on a single partition; or each access is an aggregate of a set of sub-accesses that can be completed independently on a single partition. Thus we focus on persistence and high throughput for a data partition hosted at each machine.

We use a data object model similar to the key-value scheme used in [3, 4]. The service state consists of a collection of independent homogeneous memory objects, and each object has a unique object ID. The targeted data service supports concurrent read and update of these objects from multiple threads. We assume each object is a continuous memory block. The middleware infrastructure at Ask.com has a generic serialization framework that is able to serialize noncontiguous C++ object to a continuous memory block and vice versa. A read or update operation access an object or part of an object.

The failure model we handle follows the fail-stop assumption. To simplify the description of proposed techniques, throughout the paper we make the following assumptions. We focus our discussion on local recovery due to application failures. Extending the scheme to support hardware failure is straightforward, by checkpointing to remote storage and restarting processes on a live machine, and is in fact implemented in our production system.

3 Design

In this section, we first present the system architecture of SLACH. Then we describe the application programming interface (API) and the underlying logging and checkpointing mechanism of SLACH.

3.1 System Architecture

Figure 1 illustrates the system architecture of an application service with SLACH. On the left side, a network-accessible data service employs multiple application threads concurrently accessing a number of objects in memory. A thread may perform read or update operations on these objects when servicing client requests. The application thread logs object update operations through a

SLACH function call. On the right side, SLACH periodically triggers checkpointing to reduce the log file size. During service recovery, SLACH loads the latest checkpoint from the disk, followed by replaying the logs from the checkpoint time till the time the service ceases its previous execution. Then the application resumes its execution from the consistent memory state just recovered.

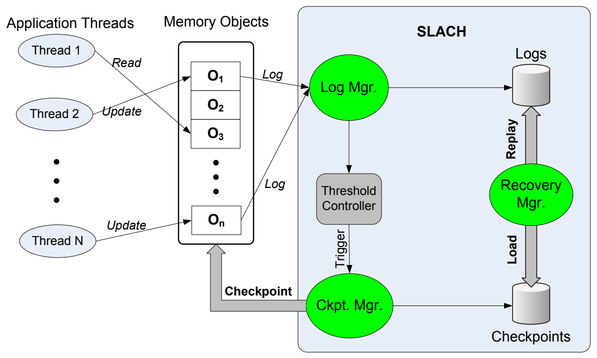

Figure 1. Architecture of SLACH.

The key modules in SLACH are discussed as follows. The log manager is responsible for appending the new update operations to the log. The checkpoint manager records a snapshot of object values to the disk. The threshold controller manages the frequency of checkpointing and triggers checkpointing when appropriate. The recovery manager uses object checkpoints to start the data service application and also replays the log to bring the service data into a consistent and up-to-date state.

3.2 Programming Interface

SLACH currently supports application services written in C++. To use SLACH, an existing service must be augmented in three ways: (1) Implementing the application service as a subclass of `SLACH::Application` and providing three callback routines to replay log records, to dump all objects into a checkpoint file, and to load one object from within a checkpoint file; (2) Creating a SLACH API object that handles physical storage of logs and checkpoints, and provides failure recovery during service start-up; (3) Issuing logging requests through SLACH API before modifying object states.

Table 1 summarizes the actions taken by various components in our implementation. We illustrate each component as following.

SLACH Managers: During a failure-free execution, the checkpoint manager periodically triggers application-defined checkpoint callback. The checkpointing frequency is adjusted dynamically by the threshold controller. During failure recovery, the recovery manager performs automatic service state recovery by loading checkpoints and replaying log records.

```
class SLACH::Application {
  /* a parameter is a pair of size and address*/
  typedef std::pair<uint32_t, void*> para_pair;
  /* a vector of parameters */
  typedef std::vector<para_pair>     para_vec;

protected:
  /* application checkpoint callback function */
  virtual void ckpt_callback()=0;

  /* callback of loading one object checkpoint*/
  virtual void load_one_callback(int64_t obj_id,
      const void *addr,uint32_t size)=0;

  /* callback of replaying one operation log */
  virtual void replay_one_callback(int64_t
     obj_id, int op, const para_vec& args)=0;
};
```

Figure 2. Callback functions defined in an application and called by SLACH during checkpointing, recovery and log replaying.

Application: An application needs to implement three service-specific callbacks by inheriting from the `SLACH::Application` class as shown in Figure 2. During a failure-free execution, the application service needs to log an operation for every update of the memory objects. A checkpoint callback, `ckpt_callback()`, is invoked periodically by the checkpoint manager to make a checkpoint of application memory objects. During failure recovery, `load_one_callback()` and `replay_one_callback()` are invoked by the recovery manager to load an object checkpoint and replay an operation log respectively.

SLACH API: This row lists two library functions that can be used by applications. Figure 3 illustrates the programming interface of these functions: `log()` and `ckpt()`. In addition, function `register_policy()` gives applications the flexibility to customize the checkpoint policy of the SLACH threshold controller (Section 4).

```
class SLACH::API {
public:
    /* register ckpt. policy and parameters */
    void register_policy(const Policy& p);

    /* log one write operation */
    void log(int64_t obj_id, int op, ...);

    /* checkpoint one object */
    void ckpt(int64_t obj_id, const void* addr,
            uint32_t size);
};
```

Figure 3. The SLACH programming interface used for logging and checkpointing.

	Failure-free Execution		Failure Recovery	
	Logging	**Checkpointing**	**Checkpoint loading**	**Log replaying**
SLACH Managers		Periodically call `ckpt_callback()`.	For each object checkpoint, call `load_one_callback()`.	For each operation log record, call `replay_one_callback()`.
Application	For each object update, call `log()`.	Define `ckpt_callback()` : for each selected object, call `ckpt()`.	Define `load_one_callback()` : recover an object from checkpoint.	Define `replay_one_callback()` : replay an update operation.
SLACH API	`log()`	`ckpt()`		

Table 1. Illustration of actions taken by various components in the SLACH framework.

3.2.1 An Example

We uses the following example to illustrate how a data service is constructed using SLACH. Assume an object `Item` is defined as:

```
struct Item {
    double price;
    int    quantity;
};
```

Figure 4 gives the skeleton of a service called `MyService` that manages 1000 such objects. In this example, the application uses the array index as object ID.

`MyService` defines the operation type for updating an item's price as `OP_PRICE`. Function `update_price()` shows the implementation of this write operation. Note that operation logging is performed before updating the object in memory and the operation log is sent to SLACH. `ckpt_callback()` iterates through all 1000 memory objects and checkpoints these objects one by one. `load_one_callback()` restores a memory object by copying saved object data from a checkpoint. During replay, `replay_one_callback()` is called to redo a logged operation on a specified object.

```
class MyService : public SLACH::Application {
private:
    Item  obj[1000]; /* Application objects */
    SLACH::API slach_;   /* SLACH API */
    static const int OP_PRICE=0;/* an op type */

public:
    /* Update an item's price: log then update */
    void update_price(int id, double p) {
      slach_.log(id, OP_PRICE, &p, sizeof(p));
      obj[id].price = p;
    }

    void ckpt_callback() {
      for (int i=0; i<1000 ; i++)
        slach_.ckpt(i, &obj[i], sizeof(obj[i]));
    }

    void load_one_callback(int64_t id,
                const void *p, uint32_t size) {
      memcpy(&obj[id], p, size);
    }

    void replay_one_callback(int64_t id, int op,
                          const para_vec& args) {
      switch (op) {
        case OP_PRICE:
         obj[id].price = *(double*)args[0].second;
         break;
        // ...
      }
    }
};
```

Figure 4. A simplified example of using SLACH.

3.3 Selective Operation Logging and Checkpointing

Our framework employs a *selective operation logging* to keep a list of operations that modify applications' memory objects. Each log record has the following format:

$(oid, op\_type, parameters, timestamp)$,

where *oid* specifies the object on which the operation is applied, *op_type* is the user-defined type of the operation, *parameters* contain the parameters for replaying the specific operation, and *timestamp* is the logical timestamp when the operation happens. SLACH is oblivious to the meanings of *op_type* and *parameters*, and directly passes them to a service-specific log replay routine during recovery. Similar to write-ahead log, logging requests are issued by service threads prior to any state change of memory objects.

The advantage of the above operation log with a customized log-replaying callback function is that an application programmer can define domain-specific strategies on how an object update should be logged and replayed when needed. As a special instance, we can use this to implement *value log*, which records the whole object state after each update operation. For applications discussed in Section 5, objects are normally partially changed and recording the entire object state has too much space overhead. Given SLACH's support, we only log operation-specific data necessary to replay update operations, which reduces log size significantly. Noted that for certain applications, if replaying an operation takes a longer time, a programmer can choose a tradeoff of implementing operation logs or value logs.

SLACH adopts an object-level fuzzy checkpointing [14,

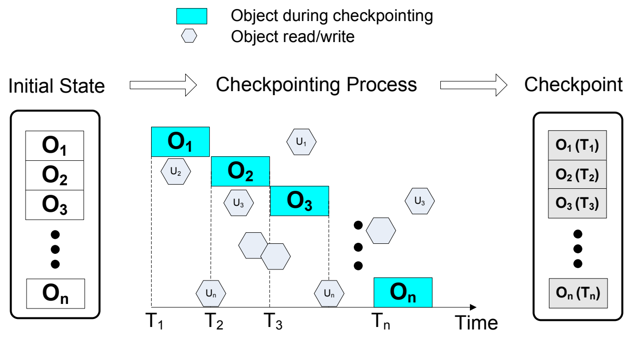

Figure 5. Fuzzy checkpointing allows access to objects that are currently not being checkpointed.

21] to avoid service disruption during the checkpointing process. In this way, when some objects are being checkpointed, all of other objects can still be accessed in parallel. As illustrated in Figure 5, during the checkpointing process objects are dumped out one at a time and the checkpointing of one object does not affect access to other objects. For instance, when object O_1 is being checkpointed, the access to object O_2 is still allowed.

Let $C = \{O_1(T_1), O_2(T_2), ..., O_n(T_n)\}$ denote the latest checkpoint before a failure. Let $C_t = \{O_1(t), O_2(t), ..., O_n(t)\}$ denote the recovered state for these objects at time t. Let timestamps $T_1 < T_2 < ... < T_n < t$ and there is no failures until time t. With two assumptions that all objects $(O_1, O_2, ..., O_n)$ are independent and all update operations applied to these objects are deterministic, we can show that our scheme has the following property.

Property 3.1 *At any time $t1$ ($t1 > t$) when there is a failure, SLACH can roll back the states of all objects to time t. The recovered state C_t is consistent to the state of objects obtained by executing updating operations up to time t.*

For any object O_k with a checkpoint as $O_k(T_k)$, any update operation after time T_k and before time t is logged. Thus SLACH will retrieve this checkpoint $O_k(T_k)$ and then apply additional update operations in the log up to time t. This results in a deterministic and consistent state for object O_k. Because each object is independent of each other, logged operations can be applied safely to produce consistent object states.

4 Adaptive Control for Checkpointing Frequency

The SLACH architecture includes an adaptive checkpointing frequency controller to strike a balance between checkpointing cost and recovery speed. If checkpointing is conducted sporadically, the operation log can become very large, leading to lengthy service recovery. On the other hand, frequent checkpointing degrades runtime service performance, because the checkpointing process competes system resources and temporarily blocks access to objects being captured.

Our scheme uses an adaptive runtime controller to dynamically adjust the checkpointing frequency. The basic idea behind this scheme is that when the service load is high, checkpointing should be done less frequently to avoid service performance degradation. On the other hand, when service load is low, the overhead of checkpointing is negligible, so we can conduct checkpointing more frequently to reduce the log size for fast service recovery.

In our current SLACH implementation, a programmer can select one of the following policies to control checkpointing frequency: the number of logged records, log file size, or checkpointing time interval, and can also specify the allowable threshold lower bound and upper bound $[LB, UB]$. For example, if the number of logged records is used, the SLACH threshold controller selects a threshold to control the number of logged records between LB and UB. When the number of logged records on disk exceeds this threshold, the checkpointing process is triggered to reduce the number of logged records.

SLACH determines the triggering threshold as follows. When the predicted server load drops below a low watermark (LW), the overhead of checkpointing is negligible and the lower bound is used as a threshold to perform checkpointing more frequently. On the other hand, when the predicted server load exceeds a high watermark (HW), the upper bound is used to adjust the checkpoint controller in order to perform checkpointing as sporadically as possible. When the predicted server load lies between LW and HW, we compute the threshold through a nonlinear function of the server load. Specifically, the controller adjusts the checkpoint threshold using the following formula:

$$Threshold = LB + F(load) \times (UB - LB),$$

where

$$F(load) = \begin{cases} 0 & load \leq LW \\ (\frac{load - LW}{HW - LW})^{\beta} & LW < load < HW \\ 1 & load \geq HW \end{cases}$$

In our scheme, *load* is an exponentially weighted moving average (EWMA) of the incoming request rate.

$$load_{curr} = \alpha \times load_{prev} + (1 - \alpha) \times sample,$$

where $load_{curr}$ is the estimated current system load, *sample* is the observed system load within the last sampling window. The parameter β can be adjusted for different applications accordingly. In addition, HW and LW can be adjusted and different applications can register the controller with different policy and control parameters.

5 Implementation and Applications

We have implemented SLACH API and its runtime support in C++ and it is part of a middleware infrastructure platform at Ask.com. Using SLACH, we have developed and deployed a number of production data services at Ask.com and we describe two data services below: *URL property service (UPS)* and *host information service (HIS)*. In addition, we have also implemented a high-throughput *persistent hash table (PHT)* using the SLACH library and compared it with an implementation using Berkeley DB for in-core benchmarks. These services are described as follows.

UPS hosts meta data for tens of billions of URLs and this meta data set is the property of each URL discovered from the Internet such as the last modified time, the last crawled time, document language and other classification features. Each URL's property is a persistent object that is updated independently and must be kept persistent after failures. These URL properties hosted by UPS are frequently read and updated by many other services in the Ask.com offline system, such as crawling service and near-duplication elimination service. UPS runs on a cluster of machines, serving hundreds of thousand requests per second generated from other offline services and sometime millions of requests per second. Meta data of different URLs is highly independent and these URLs are partitioned among machines. Thus each data access request is answered by one of UPS machines and access to different URLs is served in parallel. SLACH is used to log each memory object update at each machine and critical local disk data is periodically copied to remote storage to tolerate disk failures.

HIS, which runs a cluster of machines also, manages the meta information of all web hosts on the Internet. For each host, HIS maintains the country it belongs to, the number of URL stored, and its network delay, etc. Both UPS and HIS are high-throughput services with all the service data held in memory and partitioned on hundreds of machines. But UPS has much more write traffic on average. For example, the property of a URL gets updated immediately following the changes of its crawling time. At our production sites, we have observed its write traffic varies from 20% of the total traffic to 80% sporadically. For HIS, the host update traffic is bursty at times because the properties of all hosts are updated once a day or so.

Hash table is known to be a common building block for many network services [4, 9] and we have implemented a high-throughput persistent hash table service (PHT) that provides atomic single-element modifications using the SLACH library. The in-memory part of PHT is constructed as a number of buckets where each bucket contains multiple elements. Locks at the bucket level guarantee mutual exclusion of access to different elements. In our implementation, each bucket is constructed as a `__gnu_cxx::hash_map` object. Data persistence is ensured by the SLACH library.

Our experiences with the use of SLACH have been positive as service developers can focus on application logic, leaving persistent management to SLACH. For example, integrating existing UPS and HIS services with SLACH only takes an experienced programmer less than one day. In contrast, developing each of these two services from scratch without SLACH would take over a month to include the feature of log-based recovery while allowing high throughput.

6 Evaluations

We present an evaluation of SLACH with the following objectives: 1) Demonstrate that SLACH imposes a small run-time overhead on applications when selective logging is integrated; 2) Study the system behavior during the checkpointing process, and illustrate that services can continuously handle client requests without interruptions during checkpointing; 3) Evaluate the effectiveness of the proposed adaptive run-time checkpoint control mechanism; 4) Compare the performance of persistent hash table using SLACH with a Berkeley DB implementation for in-core benchmarks.

The performance metrics we use are throughput and response time. In measuring the sustained throughput compared with the request arrival rate, we use the throughput loss percentage defined as

$$LossPercent = 100 - \frac{SuccessfulRequests}{TotalRequests} \times 100.$$

A loss percent of zero means that all arrived requests are handled successfully. In terms of response time, we compute the average response time of all successfully processed requests.

6.1 Settings

Our evaluation studies were conducted on a cluster of 15 machines. Each machine has dual 2.8 GHz Intel Xeon processors with hyper-threading enabled, 4 GB of memory, one 130 GB Seagate ST3146707LC SCSI disk, and a gigabit Ethernet link. We run three applications described in the previous section. Because all applications are partitioned into a number of separate service instances running on different machines, the evaluation focuses on the performance and recovery of a single service instance on a node.

Table 2 summarizes the characteristics of UPS and HIS for a single partition in terms of in-memory data size and the maximum performance for read and write requests respectively. In order to determine the maximum read and write performance, we run the service on one machine and clients on another six machines. Each client sends out requests following exponentially distributed arrival intervals. We increase the request arrival rate until there are five percent throughput losses. This probed request rate is then used as the service capacity.

Table 2. Characteristics of UPS and HIS applications on a single partition.

Service	Data	Max. Read	Max. Write
UPS	1.9 GB	$110K$ Req/s	$56K$ Req/s
HIS	2.1 GB	$58K$ Req/s	$15K$ Req/s

Table 3 lists the parameter values used in the checkpoint frequency control of SLACH for these two services.

Table 3. Parameters used in checkpoint frequency control.

	Description	UPS	HIS
α	Moving avg. weight	0.8	0.8
UB	Log upper bound	8.0 M	1.8 M
LB	Log lower bound	1.0 M	0.3 M
HW	High watermark	85%	85%
LW	Low watermark	20%	35%
β	Scaling factor	3	6
w	Sampling window	5 s	5 s

6.2 Overhead of SLACH with Selective Logging

We use UPS to assess runtime overhead introduced SLACH for selective logging. In this evaluation, we compare the performance of two schemes: 1) *Base* scheme, where logging is disabled; 2) *Log* scheme, which enables selective logging. The Base scheme has better performance as it only writes data in memory, but it does not guarantee data persistence. For these two schemes, we compare their performance difference by varying the percentage of writes under different load conditions.

Figure 6 shows the results of throughput loss and response time when the write percentages are 20%, 50%, and 80% respectively. For all experiments, the Log scheme introduces some but reasonable runtime overhead compared to the Base scheme that has no disk writes. The overhead slightly increases when write traffic grows.

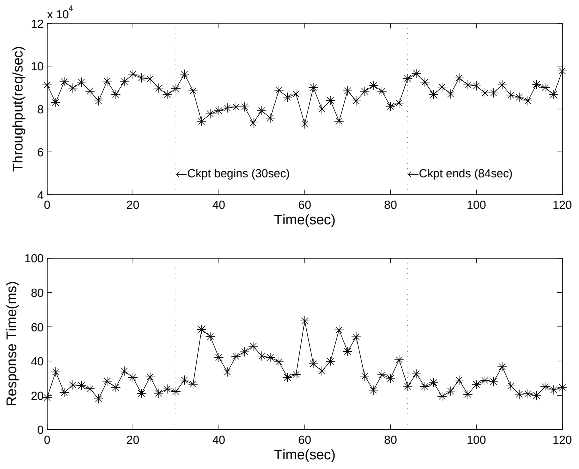

Figure 7. System behavior during checkpointing for UPS under 100% server load.

6.3 Performance of SLACH with Fuzzy Checkpointing

In this experiment, we study the system behavior during object-level fuzzy checkpointing. The primary goal is to illustrate that our checkpointing scheme achieves good throughput with no service disruption during the checkpointing period.

Figure 7 shows throughput and response time during the memory state checkpointing for UPS under 100% server load. The checkpointing happens between time 30 and 84. During this period of time, the service can continue handling client requests without interruption. During the period when checkpointing writes 1.9 GB in-memory service data to the disk (about 35 MB/s), the service has an 8.9% decrease of throughput and 57.6% increase of response time.

6.4 Effectiveness of Adaptive Threshold Selection

We evaluate the effectiveness of adaptive selection of checkpoint threshold by comparing with a fixed threshold policy.

We first evaluate service recovery time for the fixed threshold approach and the adaptive scheme. The service recovery time consists of the checkpoint loading time and the replay time of operation logs. The checkpoint loading time for a service is a constant factor that only depends on the size of service data. For instance, the checkpoint loading times are about 20 seconds and 25 seconds for UPS and

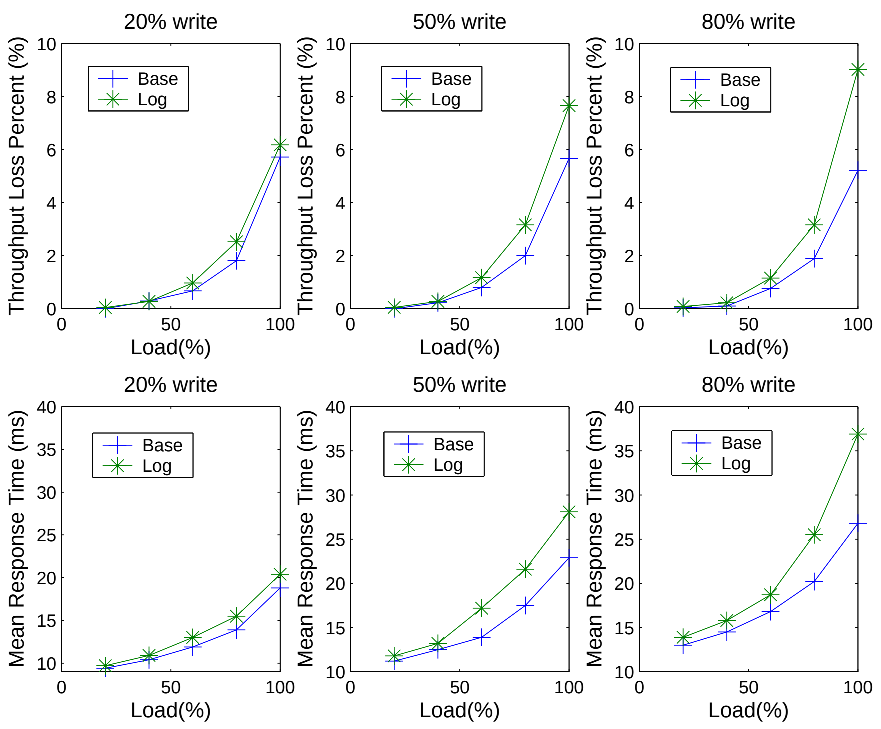

Figure 6. Runtime overhead of selective logging for UPS during normal execution.

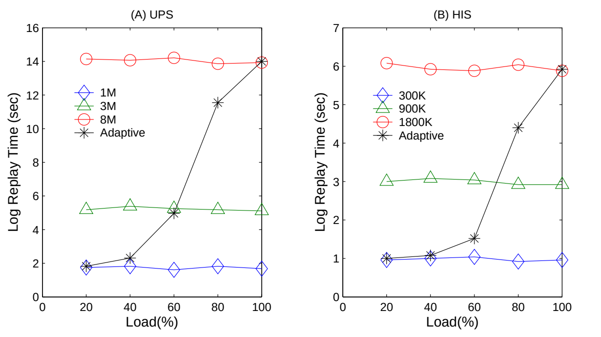

Figure 8. A recovery speed comparison for UPS and HIS using fixed checkpoint threshold with adaptive threshold control.

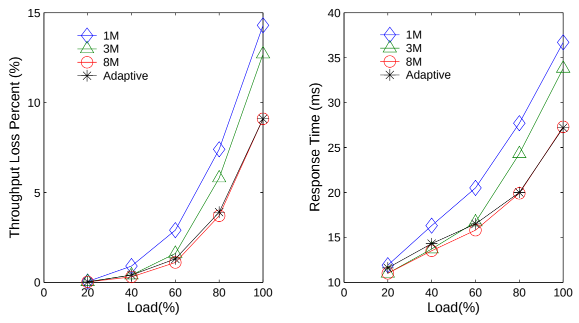

Figure 9. A performance comparison of UPS using fixed checkpoint threshold (1M, 3M, 8M) with adaptive threshold control.

HIS, respectively. Thus, this experiment focuses on the difference of log replay time.

Figure 8 illustrates the average log replay time for UPS and HIS. The replay time is almost constant for fixed threshold scheme because of the fixed log size. For the same reason, using a smaller threshold results in less replay time. The adaptive scheme selects smaller threshold values when system is lightly loaded, thus requires less replay time than the fixed threshold scheme with a bigger value.

We then evaluated the run-time overhead for these two approaches. Figure 9 compares the run-time performance of UPS using fixed checkpoint threshold and adaptive threshold control under different service load. The fixed threshold policies use 1 million, 3 million, and 8 million log entries respectively. For the fixed threshold policy, we can see that a higher threshold value always results in lower runtime overhead. This is due to the fact that a policy with a smaller threshold value would result in more frequent checkpointing, and thus higher run-time overhead. The adaptive approach performs favorably under different load conditions and has comparable performance as the best fixed threshold policy of 8 million.

6.5 Performance of PHT

This experiment evaluates the maximum throughput of lookups and updates on the persistent hash table using SLACH and using Berkeley DB's BTREE[1]. We limit the maximum memory for both schemes to be 1.5 GB. For Berkeley DB, the version being used is 4.4.2; page size is set to 32 KB; and cache size is 1.5 GB. To perform a fair comparison, we configure both schemes in a mode that guarantees no loss of data after an application crash, as long as the operating system does not also crash. For SLACH, we flush every operation log to the OS buffer. For Berkeley DB, we configure the transaction options to use both *DB_TXN_WRITE_NOSYNC* and *DB_AUTO_COMMIT* flags [1].

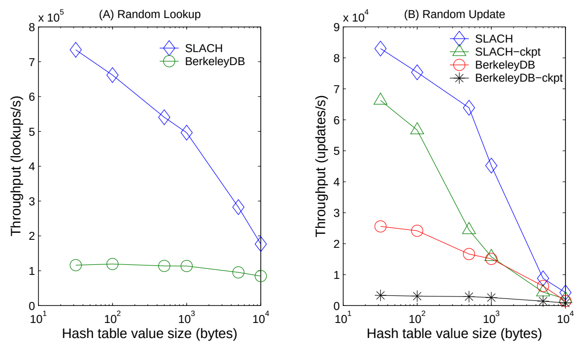

Figure 10. A comparison of maximum throughput of random lookups and updates on PHTs built out of SLACH and Berkeley DB.

This experiment studies the performance of PHT when the hash table data can be completely held in memory. We vary the size of hash table elements from 30 bytes to 10 KB and conduct random lookups and updates on the hash table.

Figure 10 (A) shows the maximum throughput of random lookups as a function of hash table value size. The SLACH scheme outperforms the Berkeley DB scheme for all value sizes, and the smaller the value size, the higher performance yielded by SLACH. For a 30-byte value size, the performance of SLACH scheme is 533.1% higher than that of Berkeley DB scheme. This is because per-operation overhead is the performance bottleneck when value size is small. The SLACH scheme uses a more efficient `__gnu_cxx::hash_map` as the internal data structure, whose single key lookup only takes about 0.5 μs, ten times faster than Berkeley DB. With the increase of value size, the overhead of memory copying gradually dominates the response time. As a result, the difference between these two schemes becomes smaller.

Figure 10 (B) shows the performance of random updates. Again, the SLACH scheme outperforms Berkeley DB scheme for all value sizes. The reasons that SLACH scheme provides better performance are two folds. First, Berkeley DB incurs more per-operation overhead; Second, Berkeley DB involves more disk I/Os than SLACH.

Figure 10 (B) also shows that SLACH checkpointing has much less overhead than Berkeley DB. This is because Berkeley DB checkpointing is not asynchronous and it has to flush all committed changes in the log to the database file. SLACH conducts object-level fuzzy checkpointing so that most requests can be served in parallel as usual.

7 Related Work and Concluding Remarks

Previous research has recognized the importance of providing infrastructure platform for building cluster-based network services [8, 17]. Distributed replication [9, 17, 20] provides reliability by replicating data on a number of servers for network services. Replication support for highly available key-value stores is the key research focus in the Dynamo [4] to achieve a 24x7 ”always-on” experience. Replication is an orthogonal strategy compared to log-based recovery and is needed for many applications.

Logging has been extensively used in database systems [14] and distributed message systems [6]. To capture the non-deterministic events of a multi-threaded network server, we choose to log operations that change the application state, which is more fine-grained than logging the incoming application requests. Process-based checkpointing is a well-known technique for fault tolerance [2, 7, 16] and migration [12, 15]. The idea is to suspend a program's execution, save the entire address space of the process, and then resume the execution [19]. Unlike these process-based checkpointing, object-level fuzzy checkpointing exploits the data independence of applications and conducts fine-grained checkpointing without service disruptions. Fuzzy checkpointing is first explored in database systems [14]. Recently, Wang et al. [21] has applied fuzzy checkpointing for middleware servers. For stream processing, SPADE language of System S [10] has been extended to support checkpointing, which allows states of user-defined operators to be saved. The targeted applications and programming interface of SPADE are different from SLACH.

SLACH provides a lightweight programming framework for supporting selective logging of update operations and fuzzy object checkpointing while achieving very high per-

[1]In theory, Berkeley DB's HASHTABLE is a more direct comparison to PHT, however, our experiments show that Berkeley DB's HASHTABLE performs even worse than BTREE.

formance and concurrency, thus simplifying the construction of high-throughput persistent data services. Another contribution of this work is adjusting checkpoint frequency dynamically to meet throughput demands as much as possible. Our experimental studies of three applications show that SLACH can successfully deliver data persistence as well as very high runtime throughput.

The experiment of the persistent hash table shows that SLACH is 3-8 times faster than an alternative approach using Berkeley DB and the performance degradation imposed by the checkpoint of SLACH is only one fourth of that of Berkeley DB. Database systems such as Berkeley DB support general persistent storage with logging and thus it is hard for them to meet high throughput demands under a limited resource budget as they require more machines to provide much more functionalities than SLACH.

Our work focuses on addressing performance challenges for a class of applications by exploiting its characteristics (e.g. data object independence) and conducting fine-grained checkpointing without service disruptions for high throughput. Our system only logs deterministic operations because these operations can be replayed deterministically. While it is a limitation, our experience at Ask, Google, Microsoft, and Yahoo is that many data mining and Internet applications have deterministic operations and the proposed support is suitable for many such applications.

Acknowledgment

The authors would like to thank anonymous reviewers for their helpful comments. Much of work described was conducted when the authors were at Ask.com. Jingyu Zhou is supported in part by National Natural Science Foundation of China (Grant No. 60811130528).

References

[1] Berkeley DB Reference Guide. `http://www.sleepycat.com/docs/api_c/env_set_flags.html`.

[2] R. Baldoni, F. Quaglia, and M. Rayna. Consistent checkpointing for transaction systems. *The Computer Journal*, 44(2):92–100, 2001.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, Seattle, WA, Nov. 2006.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Symposium on Operating System Principles*, pages 205–220, Stevenson, WA, Oct. 2007.

[5] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The Performance of consistent checkpointing. In *the 11th Symposium on Reliable Distributed Systems*, Oct 1992.

[6] E. N. Elnozahy and W. Zwaenepoel. On the Use and Implementing of Message Logging. In *the 24th International Symposium on Fault-Tolerant Computing*, pages 298–307, 1994.

[7] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computer Surveys*, 34(3):375–408, 2002.

[8] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proc. of the 16th Symposium on Operating Systems Principles*, St.-Malo, France, Oct. 1997.

[9] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *OSDI*, 2000.

[10] G. Jacques-Silva, B. Gedik, H. Andrade, and K.-L. Wu. Language-level checkpointing support for stream processing applications. In *Proc. of Int. Conf. on Dependable Systems and Networks (DSN)*, Lisbon, Portugal, June 2009.

[11] B. C. Ling, E. Kiciman, and A. Fox. Session State: Beyond Soft State. In *NSDI*, pages 295–380, 2004.

[12] M. Litzkow and M. Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Usenix Conference*, pages 283–290, Jan 1992.

[13] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring Failure Transparency and the Limits of Generic Recovery. In *the 4th Symposium on Operating Systems Design and Implementation*, Oct 2000.

[14] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[15] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *OSDI*, Dec 2002.

[16] B. Randell, P. A. Lee, and P. C. Treleaven. Reliability Issues in Computing System Design. *ACM Computer Surveys*, 10(2):123–165, 1978.

[17] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu. Neptune: Scalable Replica Management and Programming Support for Cluster-based Network Services. In *USENIX Symposium on Internet Technologies and Systems*, pages 197–208, San Francisco, CA, 2001.

[18] J. S.Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Usenix Winter 1995 Technical Conference*, Jan 1995.

[19] Y. Tamir and C. H. Sequin. Error Recovery in Multicomputers using global checkpoints. In *the International Conference on Parallel Processing*, pages 32–41, 1984.

[20] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *the 6th Symposium on Operating Systems Design and Implementation*, Dec 2004.

[21] R. Wang, B. Salzberg, and D. B. Lomet. Log-based recovery for middleware servers. In *SIGMOD*, pages 425–436, 2007.

[22] Y. M. Wang, Y. Huang, K.-P. Vo, P. Y. Chung, and C. Kintala. Checkpointing and its applications. In *Proc. IEEE Fault-Tolerant Computing Symposium (FTCS-25)*, pages 22–31, June 1995.

[23] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *OSDI*, 2000.