Vol.2015-ARC-215 No.5
Vol.2015-OS-133 No.5
2015/5/26

Sparkの可用性向上に向けたインメモリKVSによるデータ管理代替方法の提案

愛甲和秀[†1]　木下雅文[†1]　小島剛[†1]　畑﨑恵介[†2]

昨今，注目度が高まっている並列処理システム Spark の障害復旧処理モデルでは，データ量やタスク処理時間に連動してダウンタイムが長くなる，可用性劣化が問題となる場合がある．本研究では，Spark の可用性劣化の原因となるデータ復旧処理時間を短縮するため，Spark のメモリ上のデータ管理をインメモリ KVS で代替する方法を提案する．本方法は，データ書き込み時のノード間でのインメモリデータ複製処理と，障害時のノード内のインメモリ KVS のデータ配置情報と Spark のデータ配置情報の同期処理により，上記復旧処理時間を短縮する．その結果，従来法では障害復旧時間が数分かかる場合においても，提案技術を適用することによりダウンタイムを常に 5 秒以内に維持できるとの評価結果を得た．以上の結果から，提案方法が Spark のダウンタイムが増加する可用性問題の解決に有効であることが明らかになった．

Proposal of Data Management Method using In-Memory KVS for Availability Improvement of Spark

KAZUHIDE AIKOH[†1]　MASAFUMI KINOSHITA[†1]
GO KOJIMA[†1]　KEISUKE HATASAKI[†2]

In recent years, real-time big-data analysis of social infrastructure is anticipated. In this area, parallel analysis platform is effective to increase the performance. However, the mainstream parallel analysis platform like Spark has a problem of availability. This paper proposes the novel data management method for availability improvement of Spark. To reduce the downtime of Spark, a spark in-memory data protection method is developed which consists of auto failover function and of data location synchronization function using highly available In-Memory KVS. Existing method takes several minutes, but the method proposed in this paper only require at most 5 seconds of downtime. As a result of the evaluation, this method is effective for the availability problem of Spark.

1. はじめに

昨今，ビックデータ処理への期待の高まりを受け，電力システム，銀行システム，通信システムのような社会インフラ分野への適用が検討されている．

このビックデータ処理システムの例として，Marz らは，Lambda Architecture という，データマイニング処理のようなバッチシステム(batch layer)，バッチシステムの処理結果を低レイテンシーでアクセスするための参照システム(serving layer), ストリーム処理のようなリアルタイムシステム(speed layer)からなるシステムアーキテクチャを提案している[1]．このシステムでは，入力データに対してバッチシステムとリアルタイムシステムでそれぞれ加工した処理結果に対する低レイテンシーでの横断検索が可能となる．

さらに，この Lambda Architecture におけるバッチシステム，ストリームシステム，及び参照システムを統合し，インメモリ処理によりシステム全体の性能向上を可能にする並列処理システム Apache Spark (以下，Spark) が注目されている．

しかしながら，この Spark の障害復旧処理モデルでは，データ消失時にデータ再構築処理が発生するため，データ量やタスク処理時間に連動してダウンタイムが長くなる．したがって，例えば，ある程度処理に時間のかかるバッチシステムの処理結果のデータを消失すると，バッチ処理の再実行時間が後段の参照システムのダウンタイムとして見えてしまい，参照システムに求められる応答性能要件を満たせなくなる可能性がある．

また，我々が対象とする社会インフラ分野においては，このシステムダウンの影響が大きく可用性が最も重視されている．したがって，我々の過去の研究において，インメモリ KVS を用いたメッセージングシステムの無停止化などのシステムの可用性向上手法の提案を行ってきた[2].

そこで我々は，この Lambda Architecture 及び Spark の社会インフラ分野への適用を想定し，一般に対人向けの参照システムに求められる応答時間 5 秒以内を目標として，Spark の可用性劣化の原因となるデータ復旧処理時間を短縮するため，Spark のメモリ上のデータ管理を高信頼なインメモリ KVS で代替する方法を提案する．

†1 (株)日立製作所
Hitachi Ltd.
†2 Hitachi America Ltd.

ⓒ2015 Information Processing Society of Japan

本稿では，2 章において Spark の可用性向上に向けた課題について述べる．3 章では本稿で提案するインメモリ KVS による Spark データ管理代替方法について述べ，4 章で Spark のダウンタイムについて従来手法及び提案手法の比較結果を示す．

2. Spark の可用性向上に向けた課題分析

2.1 Apache Spark [3]

Spark は，Hadoop 型の分散並列処理におけるデータ管理をインメモリで実現することにより，分析処理のような Iteration(何度も同じデータを参照する)の多いアプリケーションの高速化を実現する OSS のソフトウェアである．また, SQL 処理(Spark SQL), Streaming 処理(Spark Streaming)，機械学習処理(MLLib)，グラフ処理(GraphX)，統計処理(SparkR)の各種ライブラリをサポートすることにより，アプリケーションを容易に開発することができる．

さらに，Master-Work による分散アーキテクチャを採用しており，アプリケーションをタスクに分割しタスク割り当て / 進捗管理 / データ配置管理を担う Driver が動作する Master ノードと，割当てられたタスクの実行 / 処理結果のメモリへの格納を担う Worker ノードから構成される．

2.2 RDD(Resilient Distributed Datasets) [4]

Spark では，タスク処理内容及びその結果を RDD というデータ形式で管理する．さらに，partition と呼ばれるタスク処理結果のデータを Worker ノードのメモリ上にキャッシュし，ディスクアクセス数を削減することにより処理性能を向上させる(Figure 2-1)．

RDD は Master ノード上で生成され，入力となる RDD(dependencies)，入力 RDD に対する処理内容(function)，処理結果の partition を Worker ノードに配置するか HDFS に格納するかを制御するための情報(storage level)，partition の Worker ノード上の配置情報(data placement)の 4 つの情報を保持する．

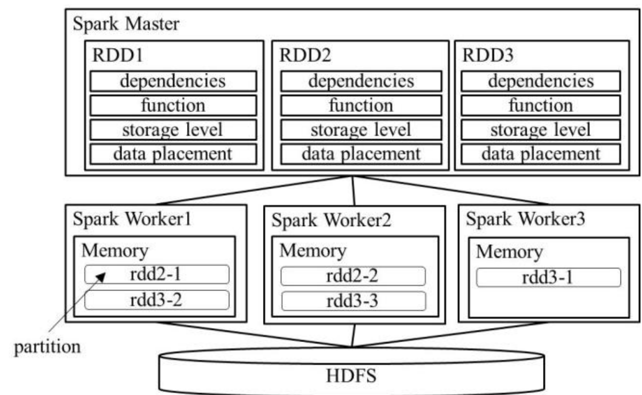

Figure 2-1 RDD

一方，Worker ノード上では，データセットの最小単位(partition)情報を管理する．

例えば，HDFS 上のログファイルを取得し，"ERROR"及び"HDFS"の文字列のある行だけを抽出してファイルに出力するアプリケーションの場合，RDD1 は HDFS 上の File １を入力として HDFS からの load 処理を実行し，load 処理結果のデータは，rdd1-1 及び rdd1-2 に分割され，それぞれ Worker ノード 1，Worker ノード 2 のメモリ上に格納されていることがわかる．なお，RDD2 と RDD3 は cache()メソッドにより，Worker ノードのメモリ上に処理結果の partition が格納され，さらに RDD2 は，ディスクにも複製が配置される(Table 2-1)．

2.3 Spark における障害復旧処理 (Linage)

Spark 及び, インメモリ HDFS 機能を提供する Tachyon[5]では，ノード障害により RDD データが消失した場合，データ複製による復旧ではなく，データ再構築による復旧アプローチを取る．これは，Spark が想定しているデータセンターのユースケースでは, 平均リソース使用量 30-50%と余剰があり，かつ，Disk や Network などの通信帯域がネックになって処理性能が劣化するため，複製よりも再構築の方が効率良いと考えたためである．

このデータ再構築を実現するために，RDD では Linage と呼ばれる処理の履歴情報を元に，ノード障害時にロストしたデータの再構築を実現している．

Table 2-1 RDD List

RDD	dependencies	function	storage level	data placement	
RDD1	File1	load	None	−	−
RDD2	RDD1	filter(_.startsWith(“ERROR”)).cache(Disk&Cache)	Disk&Cache	rdd2-1	Worker1
				rdd2-2	Worker2
RDD3	RDD2	filter(_.startsWith(“HDFS”)).cache()	Cache	rdd3-1	Worker3
				rdd3-2	Worker1
				rdd3-3	Worker 2

ⓒ2015 Information Processing Society of Japan

＜障害処理ステップ＞

1) 障害ノードを特定する
2) RDD の data placement 情報を用いて，障害で消失した RDD データを特定する．
3) RDD の dependencies 情報を用いて，消失した RDD の入力データを特定する．
4) 下記のいずれかの手段で, 消失した RDD データを再取得する．
    A) [データ再構築処理] RDD データが HDFS 上に複製されていた場合, HDFS からデータを再取得する．
    B) [データ再取得処理] RDD データの複製がなかった場合，ステップ 3 で特定した RDD と再取得する RDD の function 情報を用いて, 消失した RDD データを再構築する．

2.4 Spark 適用における課題

前述のように，Spark のインメモリ処理により定常時の性能は Hadoop に対して大幅に向上し，障害時にも，他ノードが入力となる RDD 保持している場合には，データ再構築により，すべての処理をメモリ上で完結させることができる．

しかしながら，下記の 2 つの場合において，復旧処理時の応答性能目標 5 秒以内維持を達成できない可能性がある．

1) 各タスクの処理時間が長い場合，データ再構築処理(Re-compute)の時間も長くなる．
2) RDD データのサイズが大きい場合，データ再取得処理(Data Load)の時間が長くなる．

そこで，本稿では，Spark のダウンタイムを削減する新たなデータ管理方式を提案する．

3. インメモリ KVS による Spark データ管理代替方法

本章では，2.4 節で示した，Spark 適用における課題の解決方式についての検討結果を示す．

3.1 アプローチ

まずはじめに，検討アプローチについて述べる．

Spark のダウンタイムを削減するためには，障害時のデータ再構築方式ではなく，通常時にデータ複製し障害時にもデータは常にローカルメモリアクセスで処理が完結することが必要となる．

そこで，2 つの観点から日立のインメモリ KVS 製品 EADS (Hitachi Elastic Application Data Store)[6]のレプリケーション機能に着目した．1 つ目は，Paxos アルゴリズム[7]を用いた高速・高信頼なデータ管理機能である．これは，通信分野でメッセージ基盤などの膨大なトラフィックを捌くシステムに適用されている実績を持つ．2 つ目はシステム無停止化である．これは，上述のメッセージ基盤としてEADS を活用することにより，ノード障害時のシステム無停止化を実現している[2].

このEADS を用いたシステム無停止化を実現する機能を用いることにより Spark の無停止化を実現する方式を提案する．

3.2 分散データ分析処理高信頼化技術

Spark の無停止化を実現するために，インメモリデータを EADS の多重化/高信頼化技術により管理する分散データ分析処理高信頼化技術(DDPS: Distributed Data Protection for Spark) を開発した．

3.2.1 実現アーキテクチャ

まず, インメモリ KVS による Spark データ管理代替方法の実現アーキテクチャについて述べる(Figure 3-1).

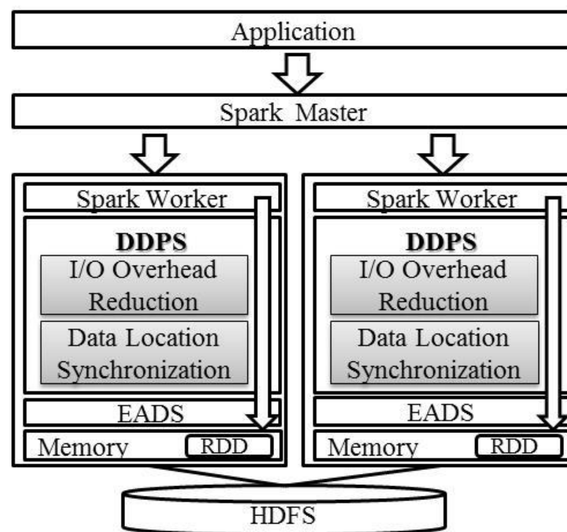

Figure 3-1 Architecture of DDPS

DDPS の実現アーキテクチャは，ユーザが Spark 向けに開発した Application，その Application をタスクに分割し，各タスクのスケジューリングと RDD データの配置を管理する Spark Master, 割り当てられたタスクの実行と, 処理結果の RDD のキャッシュ要求を発行する Spark Worker, その Spark Worker からのキャッシュ要求の EADS への転送と障害時の Spark と EADS のデータ配置情報の同期処理を担う DDPS，メモリを管理し，DDPS からのキャッシュ要求の実行，別ノードへの複製，及びサーバ障害時の系切り替え処理を行う EADS の 5 つのコンポーネントからなる．

この DDPS における I/O 性能劣化抑止技術(I/O Overhead Reduction)とキャッシュデータ不整合技術(Data Location Synchronization)により，Spark の無停止化を実現する．

3.3 I/O 性能劣化抑止技術

I/O 性能劣化抑止技術は，非同期データ多重化処理，ローカルアクセス制御処理，ステートレス化の 3 つの機能からなる(Figure 3-2).

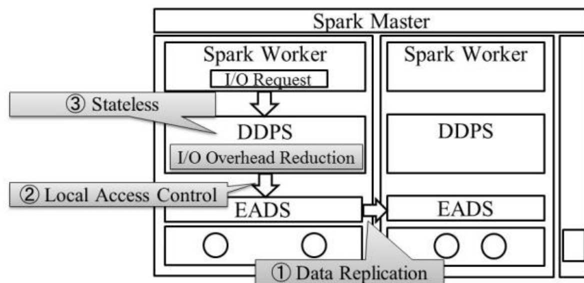

Figure 3-2 Performance evaluation function of DDPS

3.3.1 非同期データ多重化(Data Replication)処理

I/O 性能が劣化する 1 つ目の要因として，データロスト時の再構築処理に時間がかかる可能性がある．

そこで，この課題を解決するために，非同期データ多重化処理を行う．本処理では，ノード間でクラスタ構成を組み複製データを非同期で多重化して管理することにより，I/O 性能を維持しつつノード障害発生時のデータロストを抑止し，ダウンタイムレスでの障害復旧を実現する．

3.3.2 ローカルアクセス制御(Local Access Control)処理

I/O 性能が劣化する 2 つめの要因として, EADS の位置透過性を用いると，データ配置はハッシュ値によって自動的に決定するため，データ配置をアプリケーションが制御することはできない．しかしながら，Spark は各 Worker ノード内で処理が完結するようにタスクを分割することにより高性能を実現しているため，そのまま組み合わせると EADS 上のデータにアクセスする度にノード間通信が発生し，性能劣化する可能性がある．

そこで，この課題を解決するために，I/O 要求を発行した Spark Worker ノードを特定し，EADS のデータ配置制御機能を用いて I/O 要求元と同一ノードの EADS にデータを配置することにより, I/O レイテンシーの低下を抑止する．

3.3.3 ステートレス(Stateless)化

I/O 性能が劣化する 3 つ目の要因として，データ配置制御ができるようになった場合，DDPS が配置情報を管理すると，Spark，DDPS，EADS それぞれが配置情報を持つことになり，その整合性を保つ仕組みが複雑になるため，切り替え処理に時間がかかり性能劣化する可能性がある．

そこで，この課題を解決するために，データ配置情報などの制御情報をすべて EADS 側で管理し，障害時には EADS から再取得することにより，整合性を保つ仕組みを簡素化する．

以上の 3 つの解決技術により，DDPS 適用における性能劣化の問題を解消することができる．

3.4 キャッシュデータ不整合抑止技術

Spark は Worker ノードの障害を検知し，障害が発生したノードでしかかり中のタスクを別のノードに自動的に振り分ける機能を有する．同様に，EADS にも，ノード障害を検知し，障害によって消失したデータを複製からマスタ昇格する(複製データを持つ別のノードにデータアクセス要求を振り分ける)機能を有する．しかしながら，それぞれが独立に動作するため，キャッシュデータの配置情報に不整合が発生する可能性がある．

この問題を解決するキャッシュデータ不整合抑止技術は，EADS 障害検知処理，データ配置情報同期処理の 2 つの機能からなる(Figure 3-3).

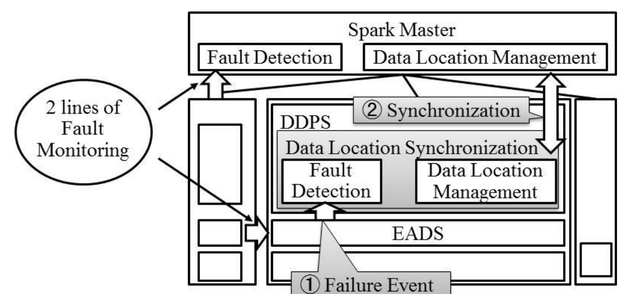

Figure 3-3 Data location synchronization function of DDPS

3.4.1 EADS 障害検知(Failure Event)処理

キャッシュデータの不整合が生ずる 1 つ目の要因としては，EADS の位置過性を用いると，アプリケーションに対して障害によるデータ配置変更を隠蔽するため，Spark がデータ配置変更の発生を検知することができないことによるキャッシュの不整合が発生する可能性がある．

そこで，この課題を解決するために，ノード障害発生時の EADS の系切り替えイベントに連動して，データ配置情報の更新処理を実行する．

3.4.2 データ配置情報同期(Synchronization)処理

キャッシュデータの不整合が生ずる 2 つ目の要因としては，障害検知の仕組みが 2 系統存在し，互いに独立に切り替え処理を実行するため，EADS データの移動先を Spark の管理するキャッシュデータ配置情報に反映できないことによるキャッシュの不整合が発生する可能性がある．

そこで，この課題を解決するために，Spark との連携において，Spark と EADS の 2 系統の障害検知機構を協調させ，データ配置情報に不整合が発生しないように一括更新する．

Vol.2015-ARC-215 No.5
Vol.2015-OS-133 No.5
2015/5/26

以上の 2 つの解決技術により，DDPS 適用におけるキャッシュデータの不整合の問題を解消することができる．

上記結果を踏まえ，5 つの解決技術のプロトタイピングを行った結果，DDPS 技術により Spark のインメモリデータを EADS で管理可能であることを確認した．

4. Spark 高信頼化技術の適用効果の検証

本章では，サンプルアプリケーションを用いた実機検証結果を示す．

4.1 検証対象

検証に当たって，下記の 3 方式を比較対象とすることとした．

- 従来方式 1：Spark RDD のデータ再構築方式
- 従来方式 2: Spark RDD のディスクアクセス方式
- 提案方式：Spark + EADS によるデータレプリケーション方式

本検証では，次の 2 点の検証を行った．1 つ目は， DDPS の処理オーバヘッドである(検証観点 1). 提案方式では従来方式に対してノード間データ複製処理が発生するため，このオーバヘッドについても評価する．2 つ目は，DDPS のダウンタイム削減効果である(検証観点 2). 提案方式の狙いであるダウンタイムの高速化効果を従来の Spark におけるデータ再構築方式（従来方式 1）及びディスクアクセス方式(従来方式 2)と比較する．

4.1.1 検証環境

ここで，検証システムの構成について述べる．本実験で使用した実機のハードウェア構成は下記の通りである(Table 4-1).

Table 4-1 Experimental environment

4 servers (1 Master node，3 Worker nodes)	
CPU	2.2GHz(6 Core, 2Socket)
Memory	96GB
Disk	500GB(SATA SSD [MLC, 3Gbps])
Network	10Gbps Ethernet

4.2 検証観点 1：DDPS の処理オーバヘッド

まず，定常時の性能劣化の度合いについて評価した結果を示す．

評価方法としては，フィボナッチ数を算出する下記の Application1 を使用した(Figure 4-1).

```
Application 1   Fib(x)
input workload x ∈ R ;
begin
  fib(x) :={ Σi fib(i)+fib(i-1) | st. 2<i<x };
end
```

Figure 4-1 Workload generator application

評価観点は， x(workload:処理負荷)を変化させた場合の従来方式(Spark 方式)と提案方式(DDPS 方式)の処理時間の差を計測ことにより，データを EADS 上に格納することによる I/O オーバヘッドを算出した．その結果，1 タスクあたりの DDPS の I/O オーバヘッドは 67ms. x(処理負荷)に寄らず I/O オーバヘッドは一定となることが分かった．

4.3 検証観点 2：ダウンタイム削減効果

次に，提案方式の障害時(サーバ 1 台ダウン)のダウンタイム削減効果について評価した結果を下記に示す．

2.4 節で示したように，Spark の障害時の性能低下の要因としては，ディスクからのデータ再取得と，RDD データの再構築の時間が大きくなることに起因する．
すなわち，Spark の処理モデルから，障害時の性能評価モデルは障害復旧時の検索処理時間を DST，通常時の検索時間を ST, 障害復旧オーバヘッドを OH とすると下記のようになる．

$$DST = ST + OH$$

また，Spark 方式の場合の障害復旧オーバヘッドは，データ取得時間を DL，データ再構築時間を RC とすると，

$$OH = DL + RC$$

となる．一方．DDPS 方式の場合の障害復旧オーバヘッドは，I/O オーバヘッド を IO，系切り替え時間を FO とすると，

$$OH = IO + FO$$

となる．

そこで，データ再取得時間が増加する場合と，データ再構築時間が大きくなる場合の２つのケースについて，提案方式とのダウンタイム削減効果についての評価を行う．

4.3.1 データ再構築時間の影響

まず，データ再構築時間が長くなる，すなわち，各タスクの処理時間の増加に対するダウンタイムへの影響を評価する．

評価方法としては，4.2 節と同様の Application1 を使用した．評価観点としては，x(処理負荷)を変化させた場合の従来方式(Spark 方式)と提案方式(DDPS 方式)について，

ⓒ2015 Information Processing Society of Japan

下記の項目を計測することによりダウンタイムを計測する．

- Spark 方式の再計算時間
- DDPS の系切り替え時間

その結果，Figure 4-2，Figure 4-3 に示す通り，Spark 方式の再計算時間は，x(処理負荷)の増加に連動するが，DDPS 方式は，処理負荷に寄らず一定であることがわかった．さらに Spark 方式で主張されている[3]ように，タスク処理負荷が低い(x=20)場合は，ネットワーク転送のオーバヘッドの割合が大きくなるため，Spark 方式の効率が良いといえる．

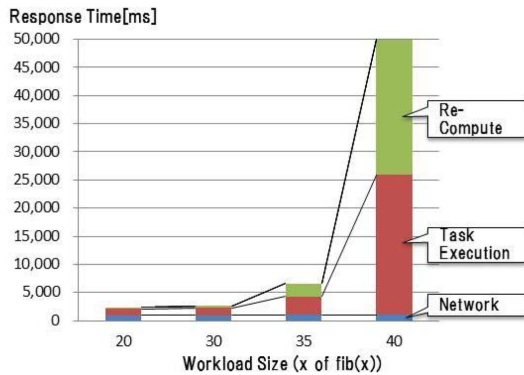

Figure 4-2　Response Time at Failure (Spark[Re-Compute])

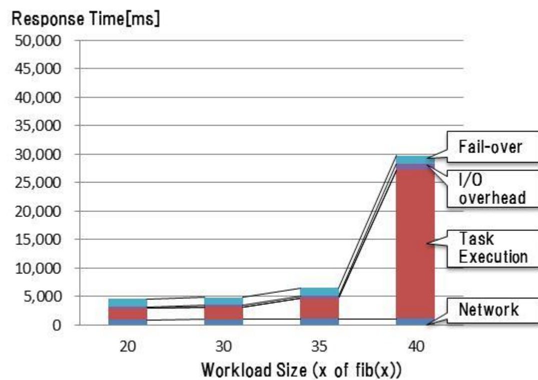

Figure 4-3　Response Time at Failure (DDPS[Fail-over])

そこで，Figure 4-4 に示すように，Spark 方式と DDPS 方式の障害復旧オーバヘッド(Recovery overhead)を比較すると，Spark 方式の復旧時間は，タスク処理時間に比例して増加し，タスク処理時間が 1.5 秒以上のタスクを実行している場合は，目標性能 5 秒を保証できなくなることがわかる．さらに，各タスクの計算処理量が 600ms を超えると障害復旧時間が逆転し，DDPS 方式が有効となるという結果が得られた．

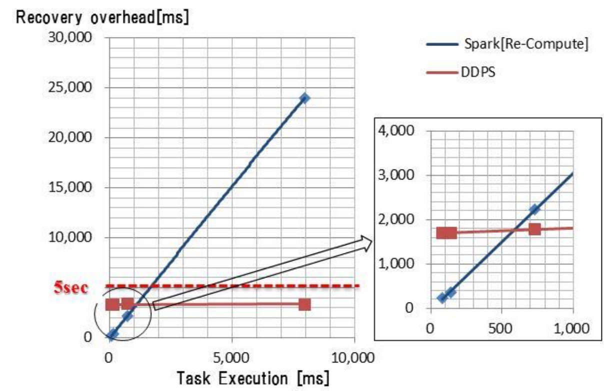

Figure 4-4　Influence of Task Execution time to Recovery overhead

4.3.2 データ再取得時間の影響

次に，データ取得時間が長くなる，すなわち，データ量の増加に対するダウンタイムへの影響を評価する．
評価方法としては，下記の Application2 を使用した (Figure 4-5).

Application 2　CreateRDD(x)

```
input text file x ;
begin
    File := load(x);
    create RDD from File;
    cache RDD on Memory
end
```

Figure 4-5　File read application

評価観点としては，入力データサイズを増加させた場合の，Spark 方式におけるデータ再取得時間を計測し，DDPS 方式との障害復旧オーバヘッド(Recovery Overhead)を比較する．

その結果，Figure 4-6 に示すように，Spark 方式と DDPS 方式の障害復旧オーバヘッド(Recovery overhead)を比較すると，Spark 方式のデータ復旧時間は，データ量の増加と比例し，データ量が 700MB を超えると，検索性能 5 秒を保証できなくなることが分かった．さらに，Worker ノードあたりのデータ量 400MB を超えると障害復旧時間が逆転し，DDPS 方式が有効となるという結果が得られた．

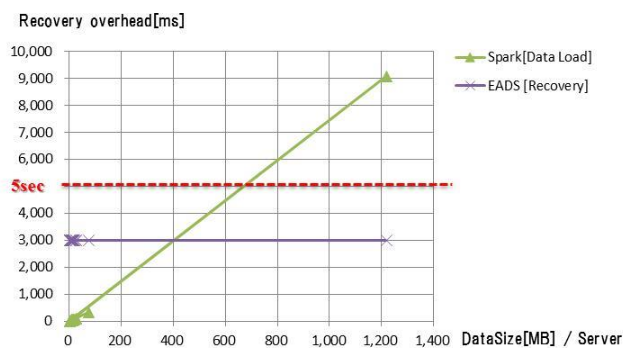

Figure 4-6　Influence of DataSize to Recovery overhead

5. おわりに

以上の結果から，提案方式適用における定常時の処理性能劣化を 67ms に抑え，障害時のダウンタイムを 5 秒以内に維持できるという評価結果を得た．さらに，各タスクの計算処理量が 600ms 以上，または Worker ノードあたりのデータ量 400MB 以上という条件で DDPS の効果が見込めるため，DDPS が汎用的に適用可能な技術であることを確認した．

ただし，Spark 環境におけるリソース管理を支援する YARN[8]や Mesos[9]の連携した構成においては，本提案手法における DDPS と YARN, Mesos のデータ配置制御方式との連携動作が検証できていないため，今後検証していく必要がある．

参考文献

[1]　Marz N., and Warren J., Big Data: Principles and Best Practices of Scalable Realtime Data Systems. (2014) Manning Publications Company.
[2]　Kinoshita, M., Takada, O., Mizutani, I., KOIKE, T., Leibnitz, K., and Murata, M., "Improved Resilience through Extended KVS-based Messaging System". (2015) IEICE TRANSACTIONS on Information and Systems, vol.E98-D, no.3, p578-587.
[3]　Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I.Spark: cluster computing with working sets. (2010). In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, p10.
[4]　Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. (2012). In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, p2.
[5]　Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. (2014). In Proceedings of the ACM Symposium on Cloud Computing, p1-15.
[6]　EADS
http://www.hitachi.co.jp/Prod/comp/soft1/cosminexus/uceads
/index.html
[7]　LAMPORT, Leslie. The part-time parliament. (1998). ACM Transactions on Computer Systems (TOCS), 16.2, p133-169.
[8]　Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., and Baldeschwieler, E. Apache hadoop yarn: Yet another resource negotiator. (2013). In Proceedings of the 4th annual Symposium on Cloud Computing, p5.
[9]　Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., and Stoica, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. (2011). In NSDI Vol. 11, p22.

ⓒ2015 Information Processing Society of Japan