BitVisorのための
OSの状態の復元機能

2013年12月6日

電気通信大学　河﨑 雄大 大山 恵弘

背景

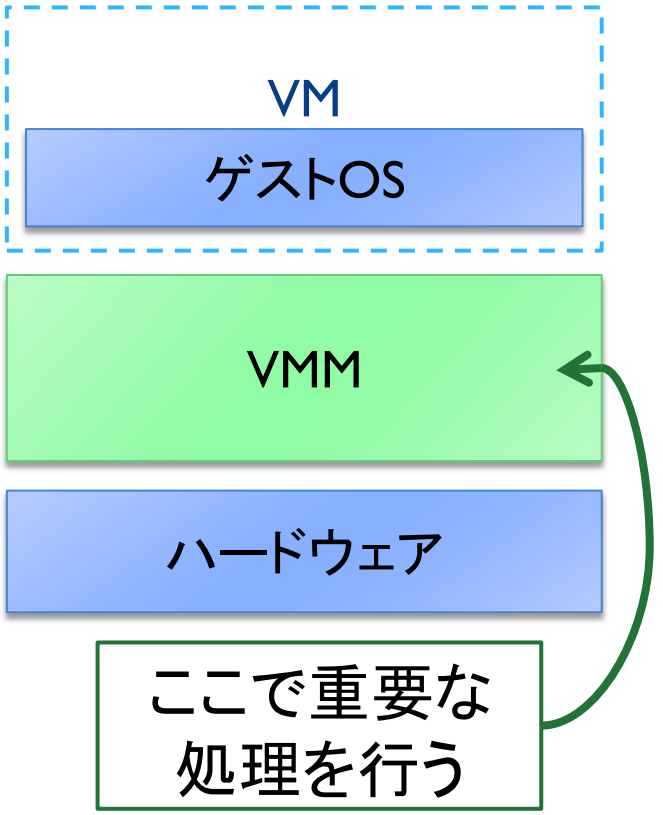
- 近年、マルウェアなどの多くのセキュリティ脅威が発見されている
- OS上のセキュリティシステムで監視や防御をするのが一般的な方法である
- しかし、OSが乗っ取られてしまうと無効化されてしまう

監視や防御などの処理はOSの外で行いたい！

- データの暗号化、マルウェアの検出処理、OSの状態の復元

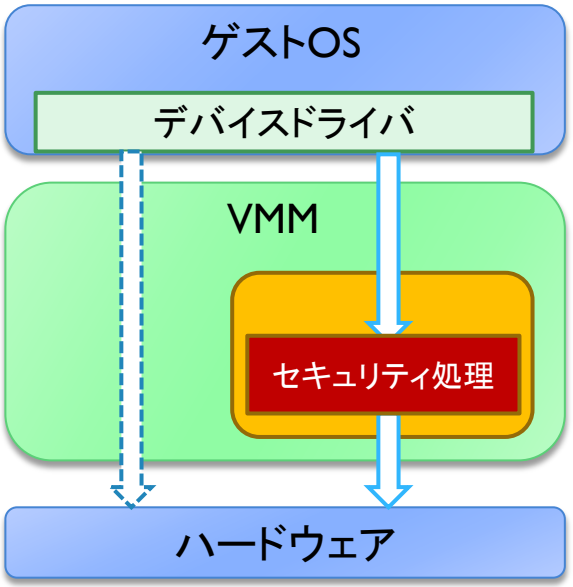
OSの外で監視や防御などの処理を行う方法

- 仮想マシンモニタ(VMM)を用いる方法
  - 仮想マシン(VM)上でOSを動かしVMM内で監視や防御を行う
    - Livewire [Garfinkelら 2003]
    - VMwatcher [Jiangら 2007]
    - BitVisor [Shinagawaら 2009]
  - 応用例
    - VM上でのマルウェアの動的解析
      - マルウェアを実際に動かして挙動の解析
      - OSの状態の復元機能が有効
      - 実環境に近いほうが望ましい



BitVisorの長所と短所

- 長所
  - ハードウェア上で直接動作する軽量なVMM
    - 実機に近い環境の提供が可能
  - 準パススルー型のVMMで最低限な部分のみを捕捉
    - 動作が高速かつセキュリティを高めることが可能
- 短所
  - ホストOSを必要とする操作は不可能
    - ファイルシステム(FS)の利用
    - ハードウェアの管理 など
  - 暗号化やVPNの機能はあるが攻撃からデータを守る機能はない
    - OSの状態の復元機能 など

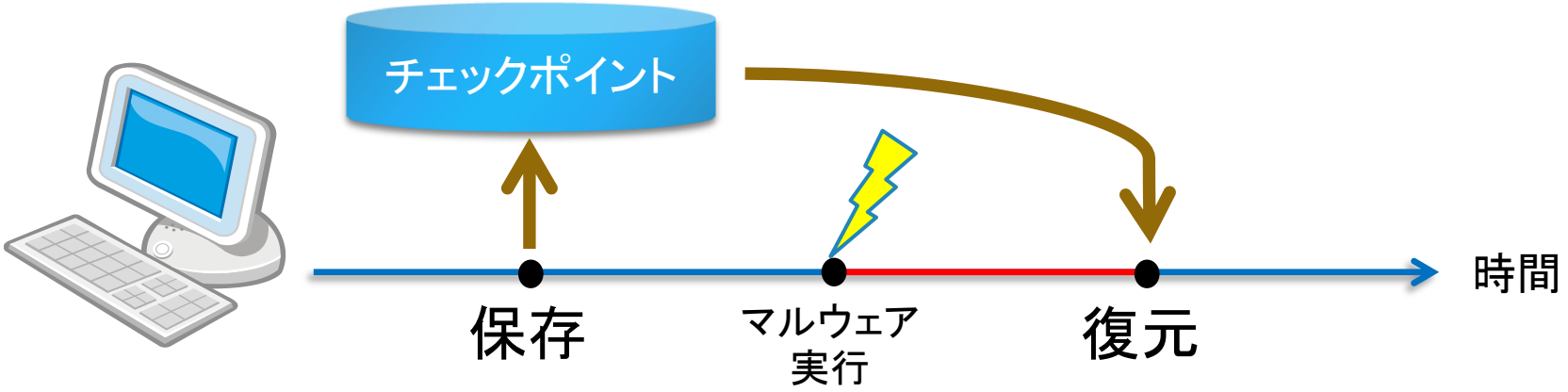


目的と方針

- 目的
  - BitVisorのためのOSの状態の復元機能の提案
- 方針
  - VMMがOSの状態を表現するもの（チェックポイント）をディスクに保存しておく
  - VMMがチェックポイントをもとにOSの状態の復元を行う
  - チェックポイントはVMMが管理する

提案機能の使用例

- マルウェアの動的解析に使用する場合
  1. VM上でゲストOSを起動する
  2. ゲストOS上でOSの状態を保存するプログラムを実行する
  3. ゲストOS上でマルウェアを実行させて挙動を観察する
  4. ゲストOS上でOSの状態を復元するプログラムを実行する



システムの概要

- 保存、復元のトリガー
  - VMCALLを呼び出す特別なプログラムをユーザがゲストOS上で実行する
  - 任意のタイミングで実行できる

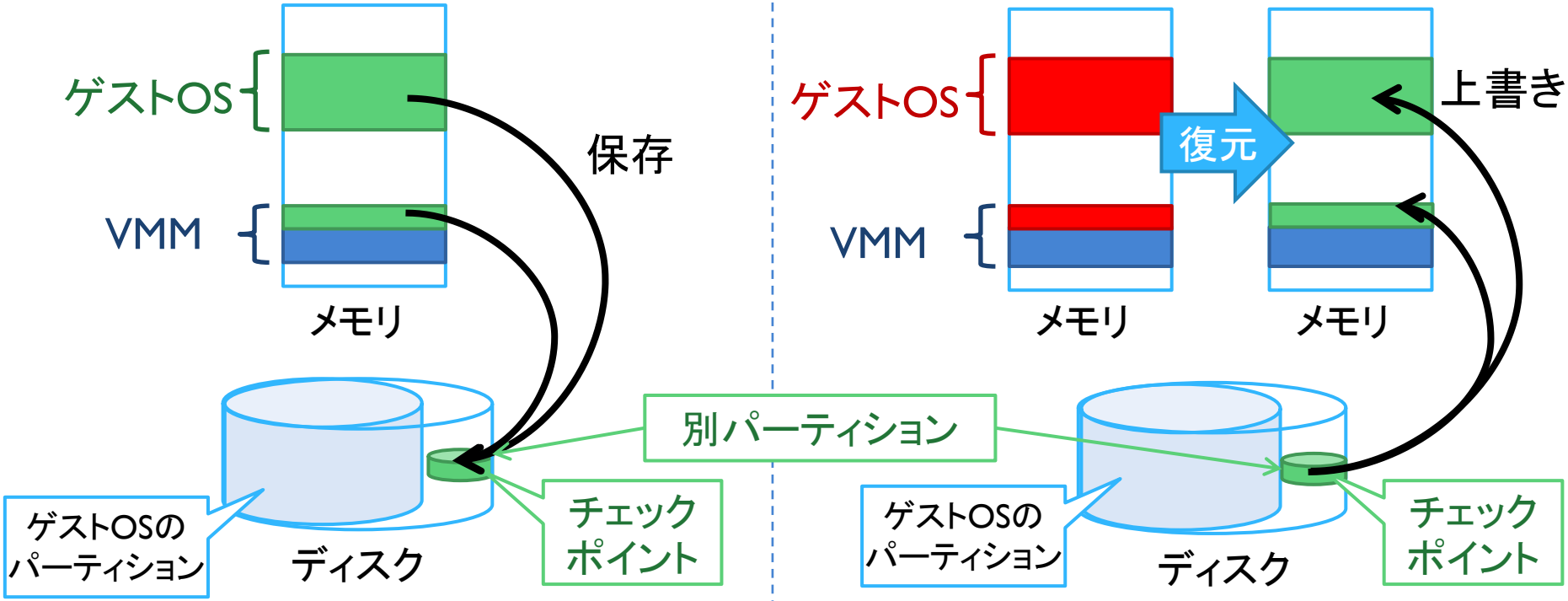
- 保存、復元するもの
  1. メモリ上のデータ
  2. ディスク内のデータ

BitVisorで実装上の問題点
- 実デバイスを扱わないといけない
- データを保存する場所がない

BitVisorで実現するのは単純ではない！

メモリデータの保存と復元（開発の途中）

- メモリの保存
  - ゲストOSの部分とVMMの一部をチェックポイントとして保存する
- メモリの復元
  - ゲストOSの部分とVMMの一部をチェックポイントで上書きする



保存するメモリデータ

- BIOS-e820から取得した実メモリの割り当て

0-9d7ff,	usable	da6de000-dadcefff,	usable
9d800-9ffff,	reserved	dadcf000-dafdcfff,	reserved
e0000-fffff,	reserved	dafdd000-dafffffff,	usable
100000-1fffffff,	usable	100000000-21e5fffff,	usable
20000000-201fffff,	reserved	db800000-df9fffff,	reserved
20200000-3fffffff,	usable	f8000000-fbffffff,	reserved
40000000-401fffff,	reserved	fec00000-fec00fff,	reserved
40200000-d9cf7fff,	usable	fed00000-fed03fff,	reserved
d9cf8000-da415fff,	reserved	fed1c000-fed1ffff,	reserved
da416000-da695fff,	ACPI NVS	fee00000-fee00fff,	reserved
da696000-da69afff,	ACPI data	ff000000-ffffffff,	reserved
da69b000-da6ddfff,	ACPI NVS		

保存するメモリデータ

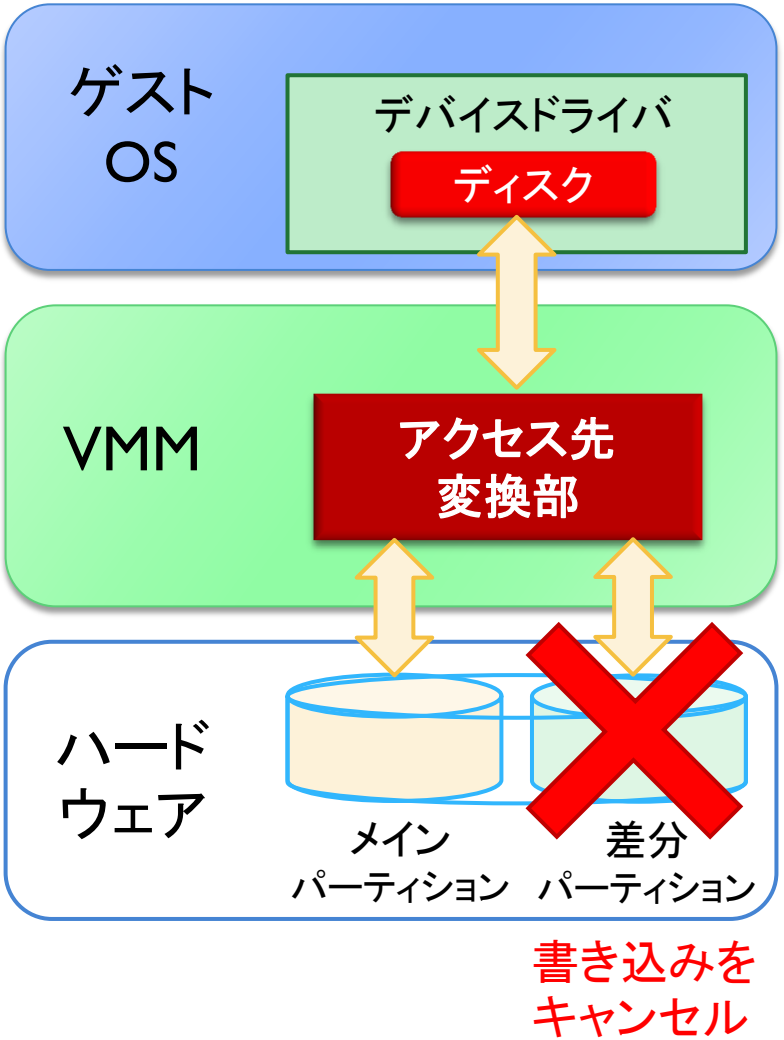
- BIOS-e820から取得した実メモリの割り当て

0-9d7ff,	usable	da6de000-dadcefff,	usable
9d800-9ffff,	reserved	dadcf000-dafdcfff,	reserved
e0000-fffff,	reserved	dafdd000-dafffffff,	usable
100000-1fffffff,	usable	100000000-21e5fffff,	usable
20000000-201fffff,	reserved	db800000-df9fffff,	reserved
20200000-3fffffff,	usable	f8000000-fbffffff,	reserved
40000000-401fffff,	reserved	fec00000-fec00fff,	reserved
40200000-d9cf7fff,	usable	fed00000-fed03fff,	reserved
d9cf8000-da415fff,	reserved	fed1c000-fed1ffff,	reserved
da416000-da695fff,	ACPI NVS	fee00000-fee00fff,	reserved
da696000-da69afff,	ACPI data	ff000000-ffffffff,	reserved
da69b000-da6ddfff,	ACPI NVS		

保存する部分は赤字の部分からVMMの領域を除いた部分
＋VMCSの領域

ディスクデータの保存と復元

- 2つのパーティションから構成
  - メインパーティション
    - ゲストOSがインストールされたストレージ
  - 差分パーティション
    - ディスクに書き込まれるデータの一時的な保存に用いるストレージ
- アクセス先を変えることでディスクの保存、復元を行う
  - 保存時: メイン→差分に変更
  - 復元時: 差分→メインに変更



アクセス先変換部

- LBA（Logical Block Addressing）の変換でアクセス先変換を実施

- 差分とメインのLBAの対応関係
  - 差分のLBA＝メインのLBA＋定数A
  - 定数A＝差分の先頭－メインの先頭

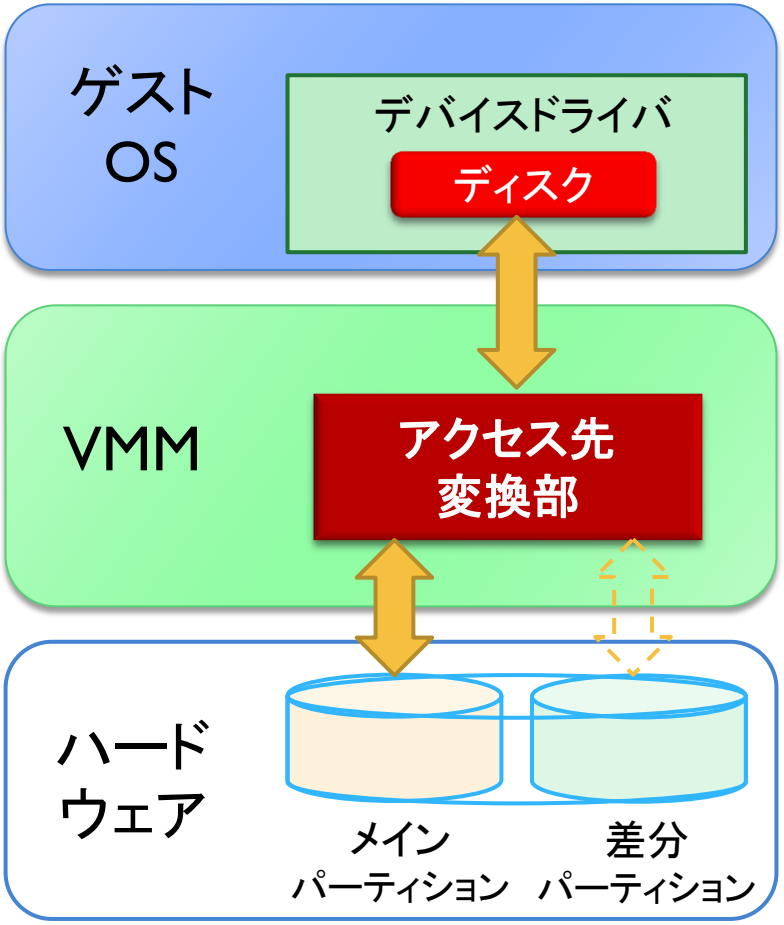
- アクセス先管理テーブル
  - 差分へのwriteを管理するためのテーブル
  - テーブルの構成要素
    - ディスクのLBA番号（セクタサイズごと）
    - ダーティフラグ
  - フラグが立つ条件
    - 差分へのwriteがあった時

lba	flag
0x120000	0
⋮	
0x12ffff	0
0x130000	0
⋮	
0x13ffff	0
0x140000	0
⋮	
0x14ffff	0

アクセス先管理テーブル

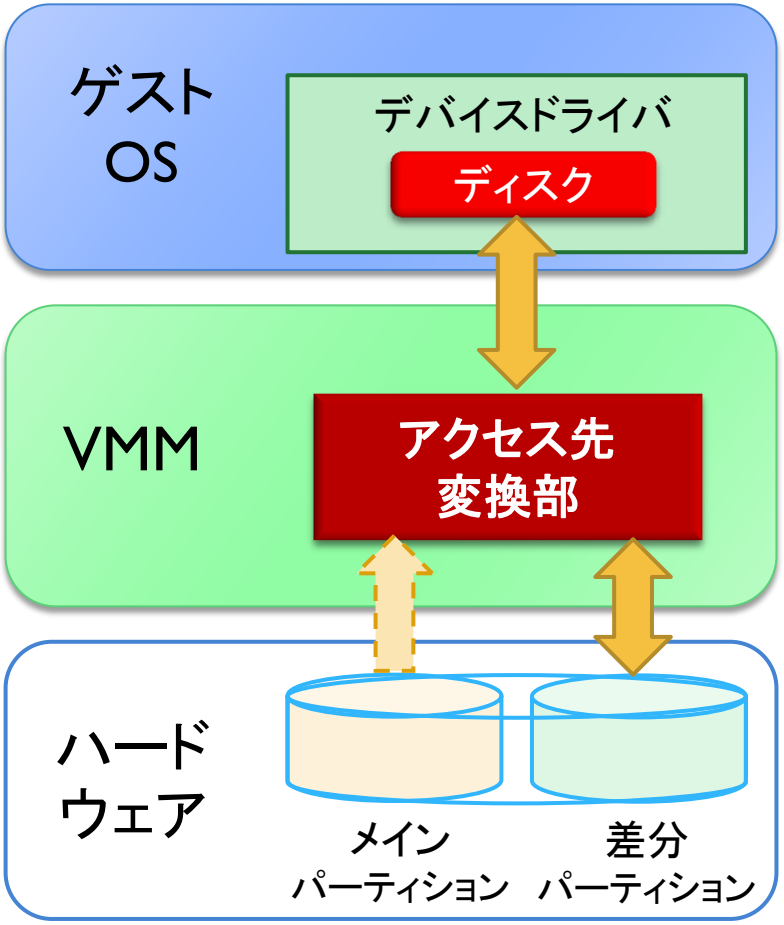
OS状態の保存前と復元後のディスクI/O

- Write
  - メインにデータを書き込む
- Read
  - メインからデータを読み込む



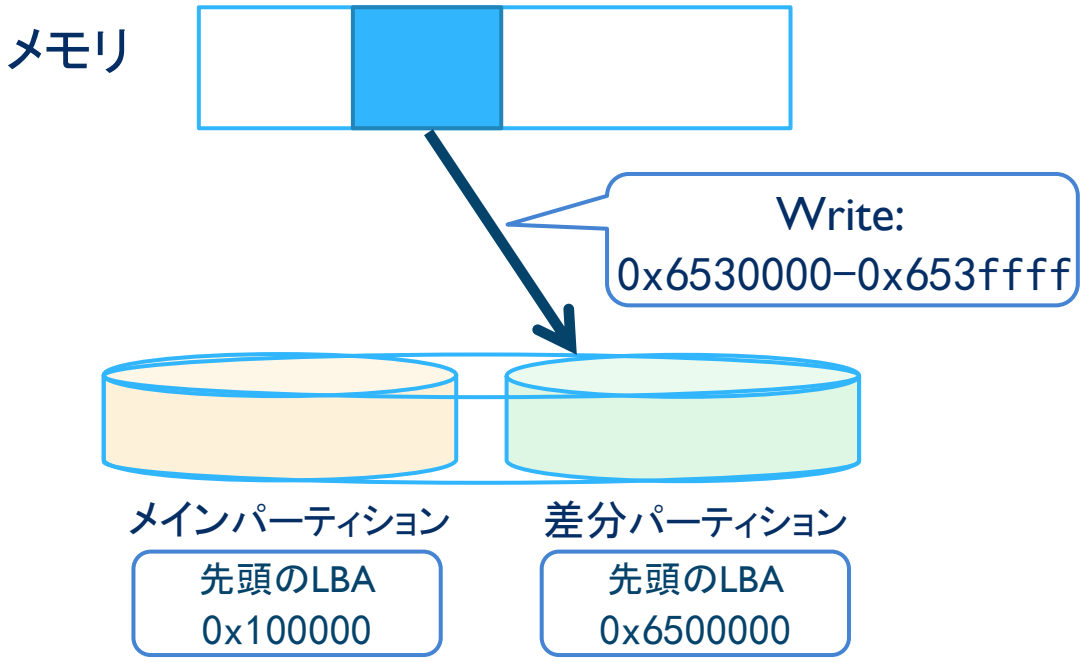
OS状態の保存後のディスクI/O

- Write
  - 差分にデータを書き込む
  - アクセス先管理テーブルにダーティフラグを立てる
- Read
  - 差分からデータを読み込む
  - ダーティフラグを立っていない所のみメインからデータ読み込む



OS状態の保存後のディスクI/Oの例 1/2

1. 0x130000から0x13ffffにデータを書き込む場合
   1. 0x6530000から0x653ffffにデータを書き込む
   2. アクセス先管理テーブルの0x130000から0x13ffffの部分にフラグを立てる

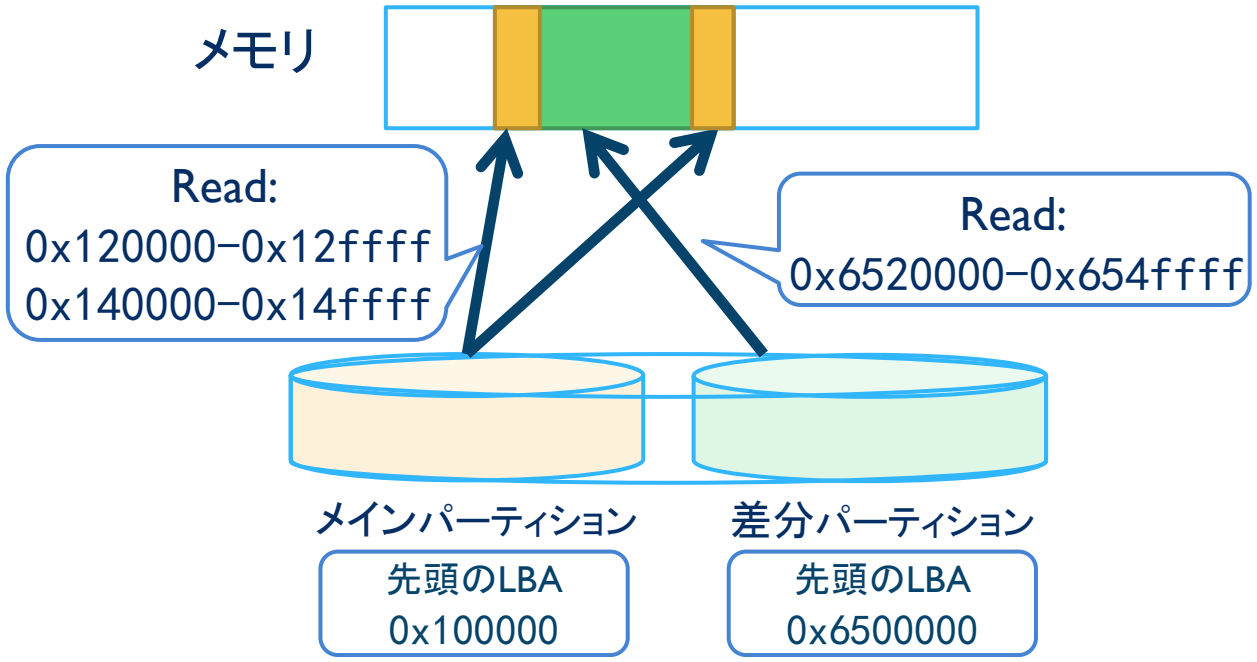


LBA	flag
0x120000	0
⋮	
0x12ffff	0
0x130000	0→1
⋮	
0x13ffff	0→1
0x140000	0
⋮	
0x14ffff	0

アクセス先管理テーブル

OS状態の保存後のディスクI/Oの例 2/2

2. 0x120000から0x14ffffのデータを読み込む場合
   1. 0x6520000から0x654ffffのデータを読み込む
   2. 0x120000から0x12ffffと0x140000から0x14ffffのデータを読み込み先ほどの読み込んだデータに上書きする



LBA	flag
0x120000	0
⋮	
0x12ffff	0
0x130000	I
⋮	
0x13ffff	I
0x140000	0
⋮	
0x14ffff	0

アクセス先管理テーブル

システムの実装

- 対象のI/O：AHCI

- アクセス先の変換を行う部分
  - drivers/ata/ahci.c
    - ahci_handle_cmd_rw_dma ()
    - ahci_handle_cmd_rw_ncq ()
- OS状態保存後のreadの処理を行う部分
  - drivers/ata/ahci.c
    - ahci_cmd_posthook ()

- 加えたコード数：約400行

アクセス先の変換を行う部分の実装例

```
void change_dst(struct ahci_port *port, int cmdhdr_index)
{
  ...
  if(dst_flag()) {
    ...
    new_lba = port->my[cmdhdr_index].dmabuf_lba + a;
    cfis = &port->my[cmdhdr_index].cmdtbl->cfis;
    port->my[cmdhdr_index].dmabuf_lba = new_lba;
    cfis->fis_0x27.sector_number = (new_lba >> 0) & 0xff;
    cfis->fis_0x27.cyl_low = (new_lba >> 8) & 0xff;
    cfis->fis_0x27.cyl_high = (new_lba >> 16) & 0xff;
    cfis->fis_0x27.sector_number_exp = (new_lba >> 24) & 0xff;
    cfis->fis_0x27.cyl_low_exp = (new_lba >> 32) & 0xff;
    cfis->fis_0x27.cyl_high_exp = (new_lba >> 40) & 0xff;
  }
}
```

提案システムで加えた部分

```
static void
ahci_handle_cmd_rw_dma (...)
{
  ...
  change_dst(port, cmdhdr_index);
}

static void
ahci_handle_cmd_rw_ncq (...)
{
  ...
  change_dst(port, cmdhdr_index);
}
```

ディスクI/Oを捕捉している部分
drivers/ata/ahci.cの一部

OS状態保存後のreadの処理を行う部分の実装例

```
void read_data_from_main(...)
{
  ...
  ret = check_flags(...);
  for(i = 0; ret; i++) {
    if(ret & 0x1) {
      lba = (a << 6) + st_main + i;
      ...
      storage_io_aread(...);
    }
    ret = ret >> 1;
  }
}
```

BitVisorのファイルread

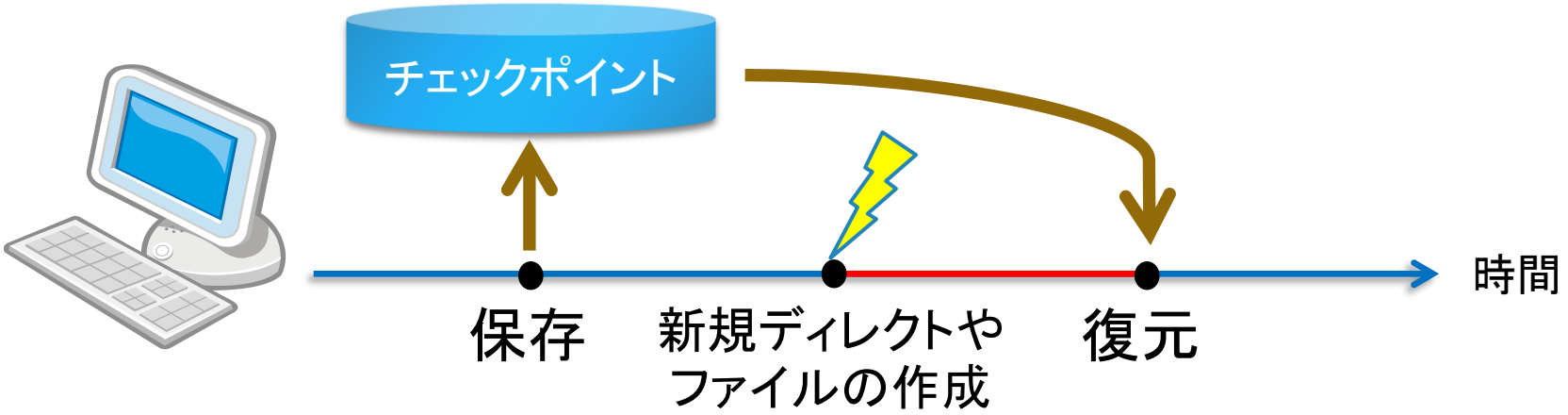
提案システムで加えた部分

```
static void
ahci_cmd_posthook (...)
{
  ...
  read_data_from_main(...);
}
```

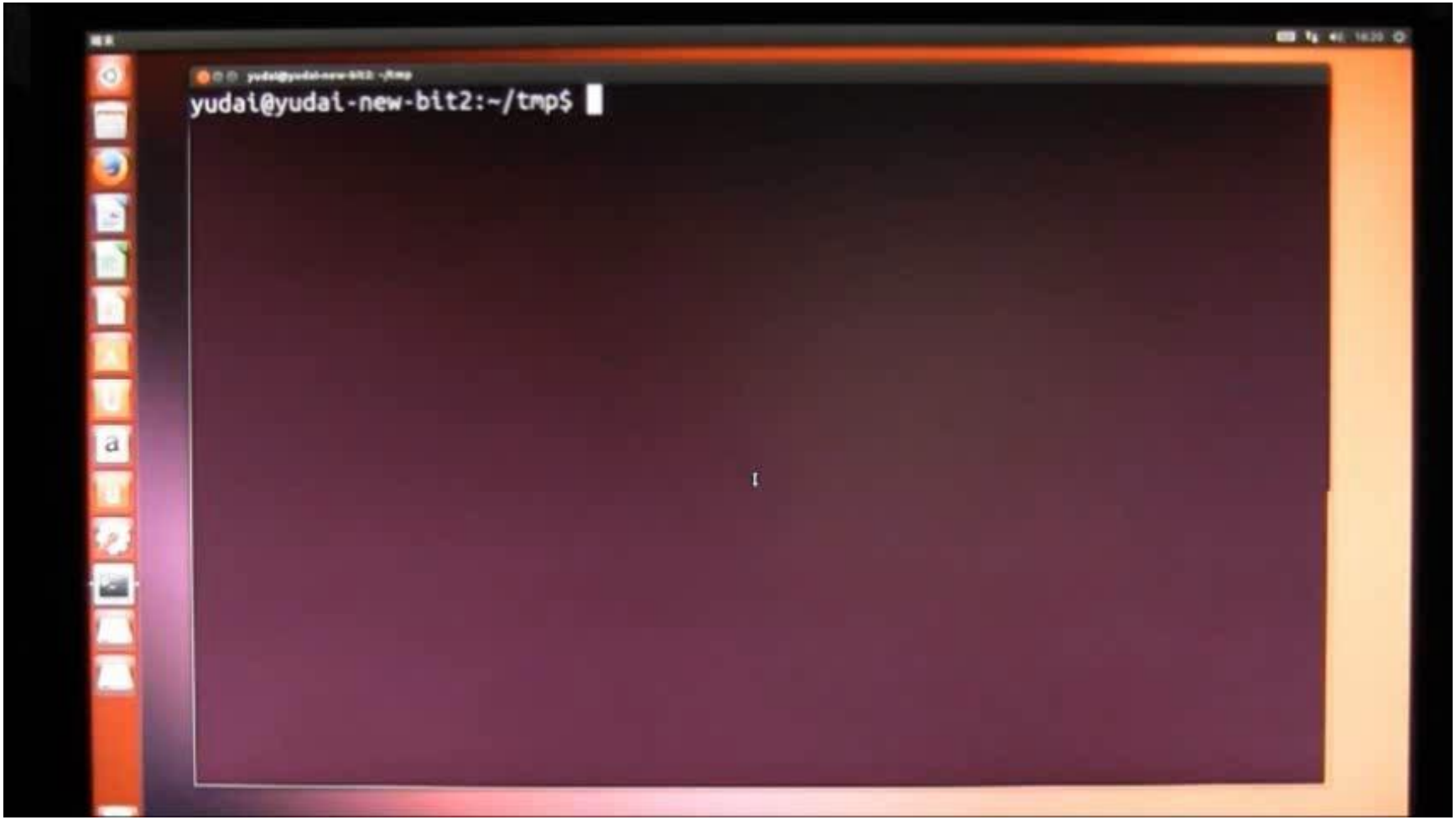
ディスクI/O終了を捕捉している部分
drivers/ata/ahci.cの一部

ディスクデータ復元のデモ 1/2

- ディスクの復元機能によってディスクの状態が元に戻るかのデモ
  - ディスクの保存後にファイルをたくさん作成するプログラムを実行し、それらがディスクの復元後にどうなっているかのデモ



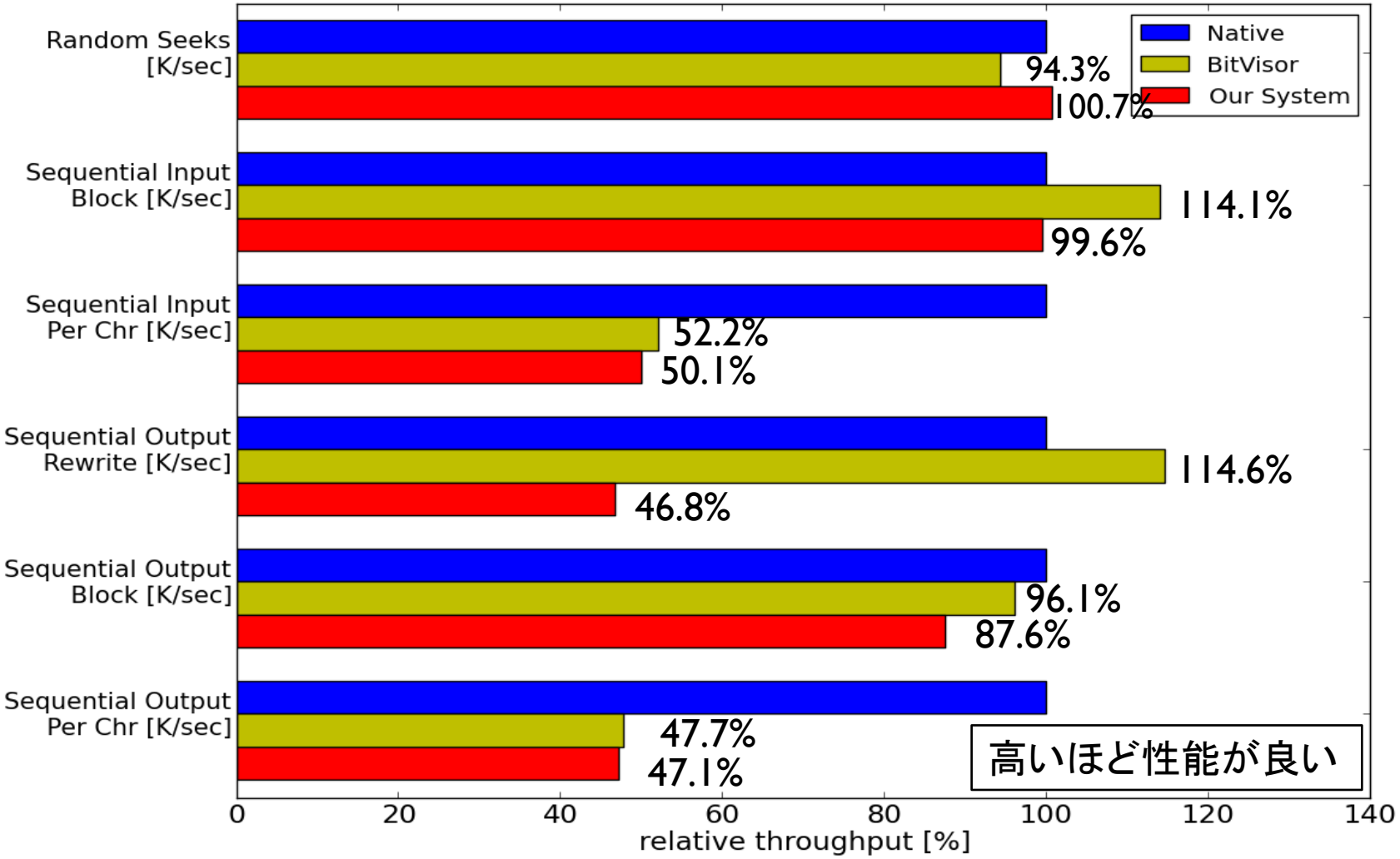
ディスクデータ復元のデモ 2/2



評価実験

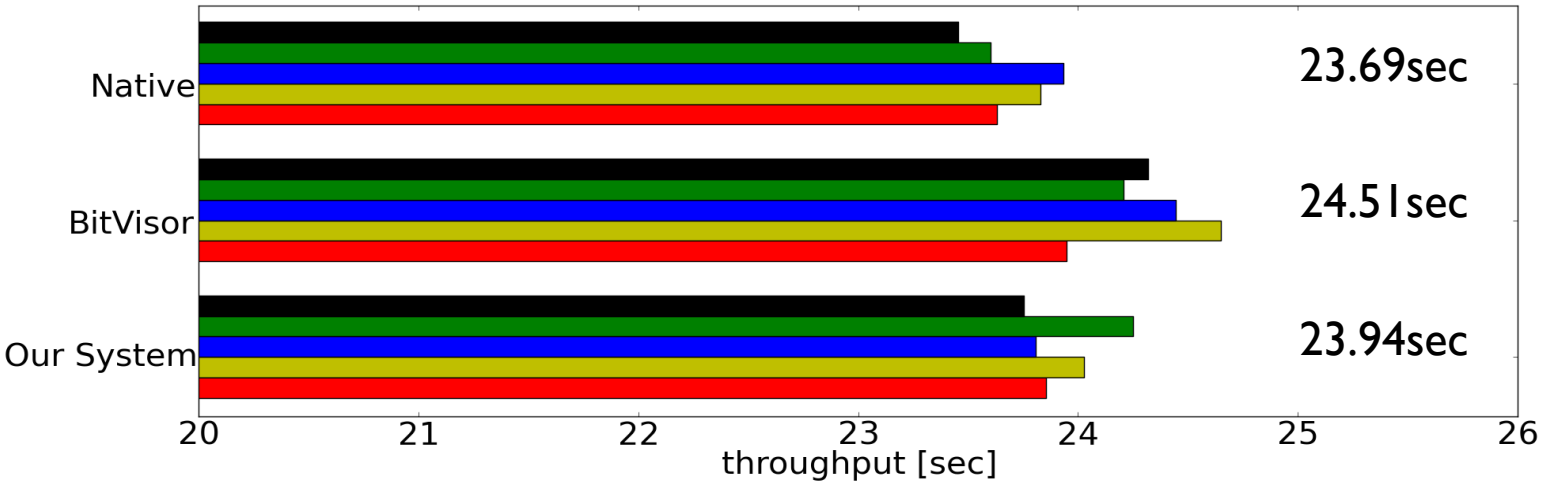
- 実験環境
  - CPU: Intel Core i3-2120 3.3 GHz × 4
  - メモリ: 8 GB
  - VMM: BitVisor 1.3
  - ゲストOS: Ubuntu 13.04 32-bit
Linux 3.8.0-32-generic
- 使用ベンチマーク
  - Bonnie++ version 1.97
  - OS上でのBitVisor1.3のビルド・ベンチマーク
- 比較対象
  - 実環境
  - BitVisor
  - 提案システム（アクセス先が差分パーティションの状態）

Bonnie++によるI/Oベンチマーク

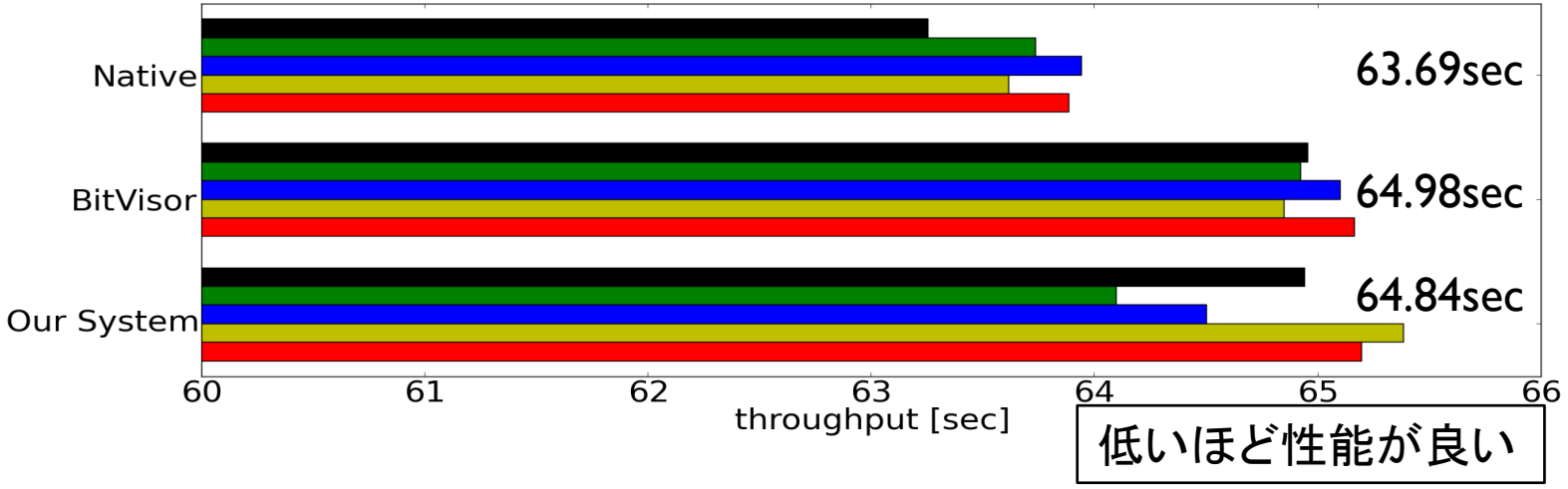


OS上でのBitVisor1.3のビルド・ベンチマーク

並列コンパイル機能をONにしてBitVisor1.3をビルドした時のベンチマーク



並列コンパイル機能をOFFにしてBitVisor1.3をビルドした時のベンチマーク



関連研究

- VMware，Hyper-V，KVM
  - OSの状態の復元機能を有したVMMである
  - ディスクやメモリの状態の保存、復元ができる
  - ホストOSがあるところが本研究とは異なる
    - ホストOSがあることで動作を止めるマルウェアも存在する

- BareBox [Kiratら 2011]
  - 動的なマルウェア解析を行うためのOSの状態の復元機能を有したVMMである
  - 本システムと同様にディスク、メモリのデータをリブートなしで復元することでOSの状態の復元ができる
  - 復元対象のOSが動いているVMと異なるVMで動いている管理OSがあるところが本研究とは異なる
    - チェックポイントの管理などで管理OSの力を借りることができる

まとめと今後の課題

- まとめ
  - BitVisorに対してディスクやメモリの状態を保存、復元できる機能を提案した
  - I/Oベンチマークを用いた結果、最大で53%のオーバヘッドが発生した
  - しかし、OS上でのBitVisor1.3のビルド・ベンチマークを用いた結果は約2%のオーバヘッドしか発生しなかった

- 今後の課題
  - ゲストOS全体のメモリの状態の復元等の未実装部分の実装
  - オーバヘッドの測定などの更なる評価や考察