プログラミング言語 7
Valgrind

田浦

Valgrind とは

- http://valgrind.org/
- Nethercote (National ICT Australia) らによって開発されたバイナリ書き換え (Dynamic Binary Instrumentation) フレームワーク
- Valgrind を用いて開発されたツール
  - memcheck：不正メモリアクセス検出ツール
  - callgrind：callgraph 構築, キャッシュシミュレーション，分岐予測シミュレーション
  - drd：競合アクセス検出
  - ...

memcheck

- 使い方:

```
1 valgrind コマンド
```

- 再コンパイルも，ソースも不要
- 例

```
1 $ valgrind hostname
2 ==7096== Memcheck, a memory error detector
3
4       ...
5
6 ==7096== For counts of detected and suppressed errors, rerun with: -v
7 ==7096== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

memcheck が検出するエラー

1. 不正なアクセス = 「割り当てられていない領域へのアクセス」
2. 初期化されていないデータへのアクセス
3. メモリリーク (malloc され，free されずに終了した領域)

バイナリ書き換えツールとは?

- バイナリ書き換えツールの目的・できること ≈ プログラム (バイナリ) の機能シミュレータの目的・できること
  - シミュレータ: 命令を一つずつ解釈実行し，その効果をシミュレートする
  - その過程で知りたい情報 (e.g., 何命令実行されたか，どのアドレスをアクセスしたか，etc.) も計算する
- 「バイナリ書き換え」は，それを高速に実行する方法 (cf. インタプリタ vs. JIT)
  - 毎回命令列を解釈実行するのではなく，元の命令列を，シミュレートする効果を「埋め込んだ」命令列に変更する
  - 例: 命令数を数えるためのバイナリ書き換え

```
mul %xmm0,%xmm1          mul %xmm0,%xmm1
add %xmm1,%xmm2    →     add %xmm1,%xmm2
                         inc $2,inst_count
jmp L1                   jmp L1
```

  - 全部を一度に書き換える代わりに，基本ブロック，スーパーブロックという単位で少しずつ変更する

memcheckの基本的な仕組み

- memcheck は以下を把握する
  - 大域変数の領域
  - スタック領域
  - malloc されている (malloc されて free されてない) 領域
- 上記のどれでもないメモリアクセスが起きたら，エラー + その究明に有用な情報を表示する
  - アクセスしたコードの場所
  - アクセスされたアドレス，それに関する情報 (スタックのそば，malloc された領域の何バイト後ろ，その malloc を呼び出した場所など)
- コードの場所をソースの行で表示するにはデバッグ情報が (-g でコンパイルされている) 必要

callgrind

以下の情報を収集・表示

- Callgraph (関数呼び出し関係の情報)
- キャッシュミス率
- 分岐予測ミス率

callgrind 使い方

- callgrind 一般

```
1 valgrind --tool=callgrind コマンド
```

- callgrind のオプション

```
1 valgrind --tool=callgrind --help
```

- キャッシュシミュレーションをする場合

```
1 valgrind --tool=callgrind --cache-sim=yes コマンド
```

callgrind キャッシュシミュレーション例

```
$ valgrind --tool=callgrind --cache-sim=yes hostname
==7253== Callgrind, a call-graph generating cache profiler
      ...
--7253-- warning: L3 cache found, using its data for the LL simulation.
nanamomo
==7253==
==7253== Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
==7253== Collected : 205069 51543 21116 914 2817 655 905 2106 598
==7253==
==7253== I   refs:      205,069  # 命令キャッシュ参照数
==7253== I1  misses:        914  # 1 次命令キャッシュミス数
==7253== LLi misses:        905  # 最終レベル命令キャッシュミス数
==7253== I1  miss rate:    0.44% # 1 次命令キャッシュミス率
==7253== LLi miss rate:    0.44% # 最終レベル命令キャッシュミス率
==7253==
==7253== D   refs:       72,659  (51,543 rd + 21,116 wr) # データキャッシュ参照数
==7253== D1  misses:      3,472  ( 2,817 rd +    655 wr) # 1 次データキャッシュミス数
==7253== LLd misses:      2,704  ( 2,106 rd +    598 wr) # 最終レベルデータキャッシュミス数
==7253== D1  miss rate:     4.7% (   5.4%   +    3.1%  ) # 1 次データキャッシュミス率
==7253== LLd miss rate:     3.7% (   4.0%   +    2.8%  ) # 最終レベルデータキャッシュミス率
==7253==
==7253== LL refs:         4,386  ( 3,731 rd +    655 wr)
==7253== LL misses:       3,609  ( 3,011 rd +    598 wr)
==7253== LL miss rate:      1.2% (   1.1%   +    2.8%  )
```

シミュレートされているキャッシュの構成

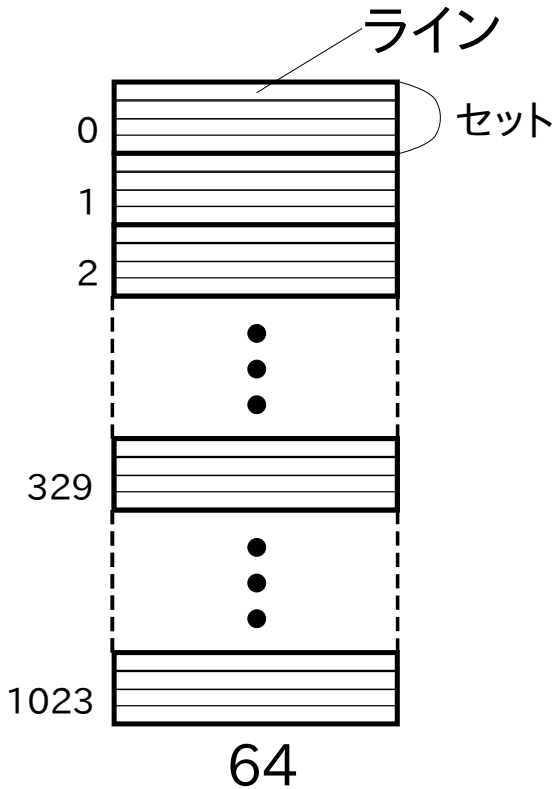
- 出力ファイル callgrind.out.*pid* を見るとわかる

```
1    ...
2                    #     容量,ラインサイズ,連想度
3  desc: I1 cache: 32768 B, 64 B, 8-way associative
4  desc: D1 cache: 32768 B, 64 B, 8-way associative
5  desc: LL cache: 4194304 B, 64 B, 16-way associative
6    ...
```

- 基本は実行した CPU のそれを使うが，レベルは 2
  - 現在の CPU の多くは 3 レベル
  - callgrind は，1 次キャッシュ(最小・最低速) と，最終レベルキャッシュ(最大・最低速) をシミュレートする

キャッシュの容量，ラインサイズ，連想度

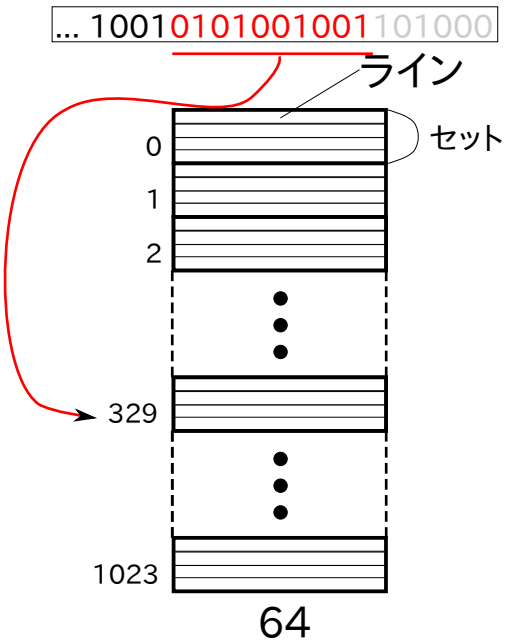
- 容量: 保持できるデータの量
- ライン: キャッシュがデータを入れ替える単位
- 連想度: あるアドレスが格納されうるラインの数



キャッシュ構成の例

容量 256KB, ラインサイズ 64B ($= 2^6$), 連想度 4 のキャッシュ

- 一つのラインは連続した 64 バイト (下位 6 bit 以外を共有する) のデータを格納
- ライン数 $=$ 256KB/64B $=$ 4096 ライン
- 4 ラインが組でひとつの「セット」
- エントリ数 $=$ 4096/4 $=$ 1024 セット
- $\therefore$ アドレス a を格納するセットは，アドレスの下位 7 bit 目から，16 bit 目までの 10 bit で決定
- 式で書けば,



$$\left\lfloor \frac{a}{64} \right\rfloor \mod 1024$$

そのセットからどのラインを追い出すかは $\approx$ LRU