Cybozu® Labs

ビット演算による最適化の妙味とJITアセンブラ

サイボウズ・ラボ

2008/4/16 光成滋生

目次

Cybozu Labs

- 自己紹介
- トリッキーなビット演算
  - Range Coderの復号処理
  - 圧縮コーデックでの例
  - FFTのビット反転(時間があれば)
- JITアセンブラ紹介
  - ペアリング暗号での使用感
  - toy VMで遊ぶ

自己紹介

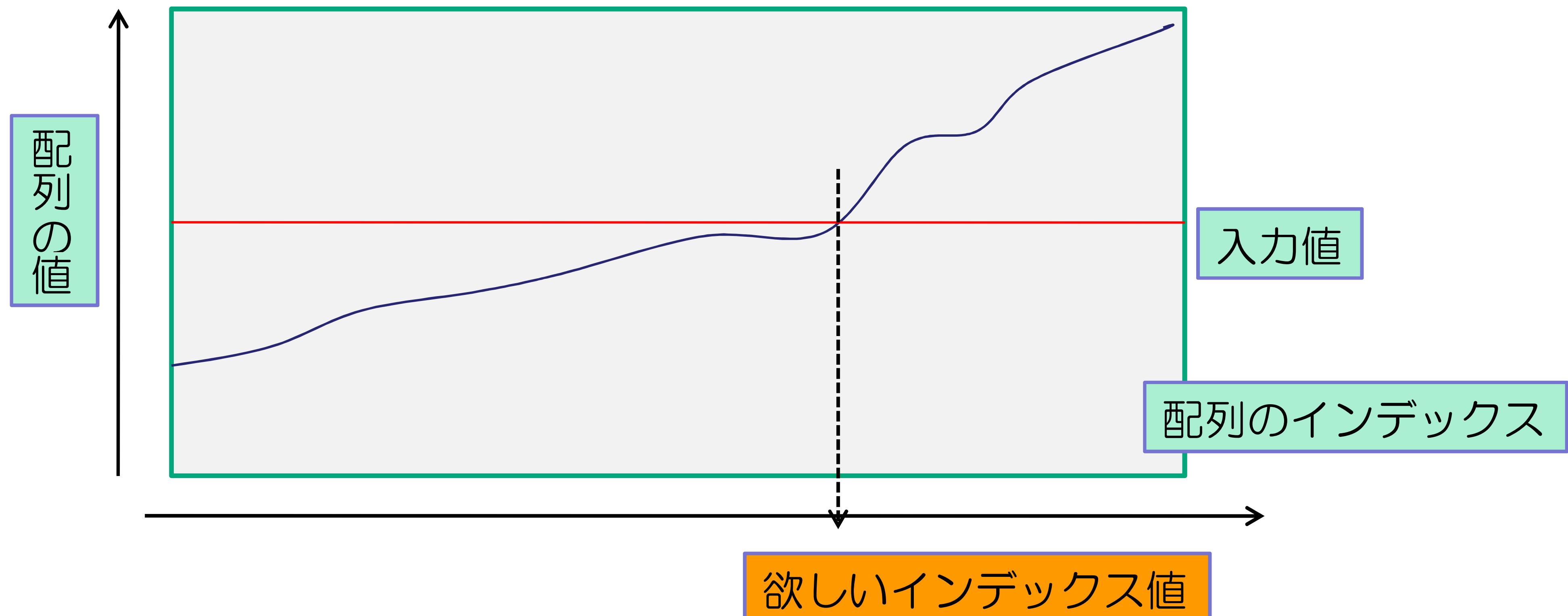
CybozuLabs

- 各種コーデックの開発&最適化
  - 午後のこ〜だ(mp3エンコーダ)
  - MPEG 2 Video, MPEG4 その他
  - CPU
    - Pentium, Athlon, PowerPCでのアセンブラ
    - MIPS, ARM, その他特殊プロセッサ
- 組み込みLinuxのブートローダ
- Linux GigabitキャプチャボードのPCI-Xドライバ
- toypcrypt(ストリーム暗号の)解読(IPAの依頼)
- ペアリング暗号の実装

Range Coderの復号処理

Cybozu Labs

- Range Coderとは
  - バイト単位で処理できる算術圧縮の一つ
  - 弊社サービスのPathtraqではURLの圧縮に用いる
- 復号の核となる部分
  - 配列の中から与えられた値を超える初めての場所を探す

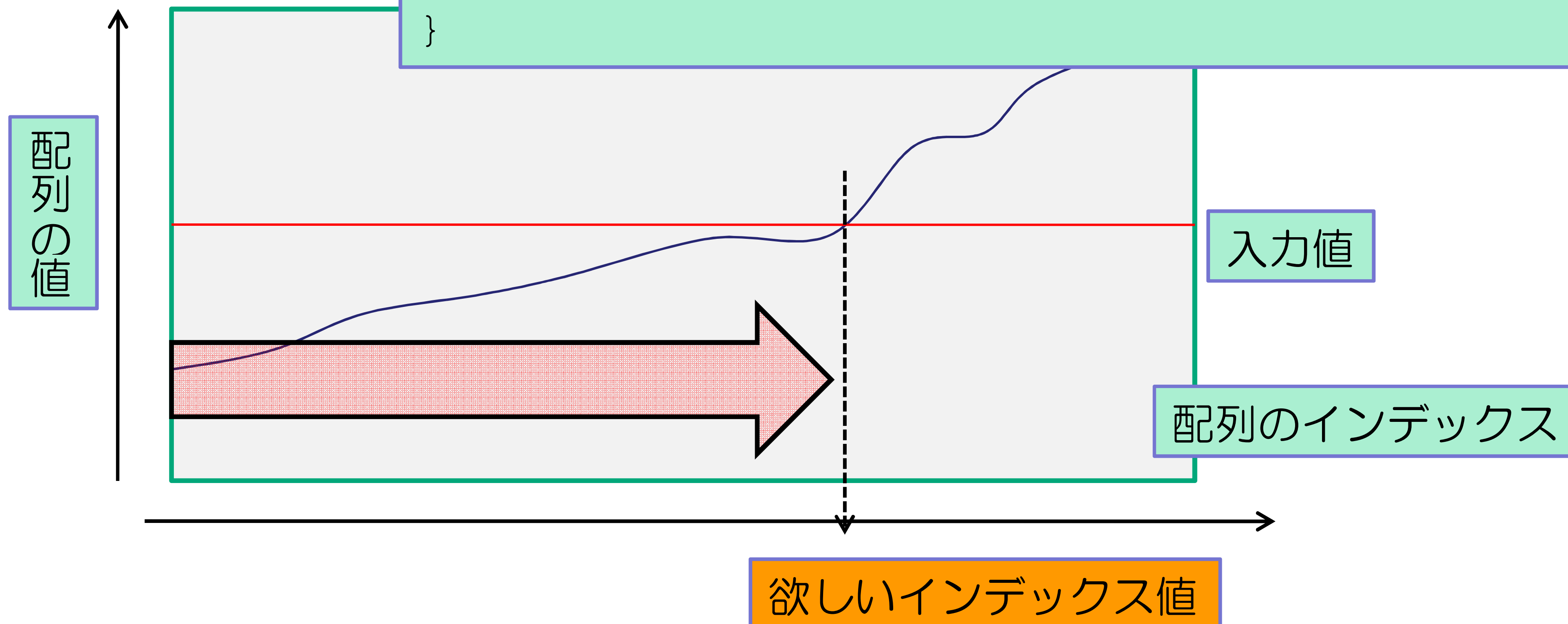


線型探索

Cybozu Labs

- 配列の先頭から順番にチェックする
  - 単純
  - 遅い

```
int getIndex(const short *inBuf, size_t num,
                    short val)
{
      for (size_t i = 0; i < num; i++) {
            if (inBuf[i] > val) return i;
      }
      return -1;
}
```



二分木探索

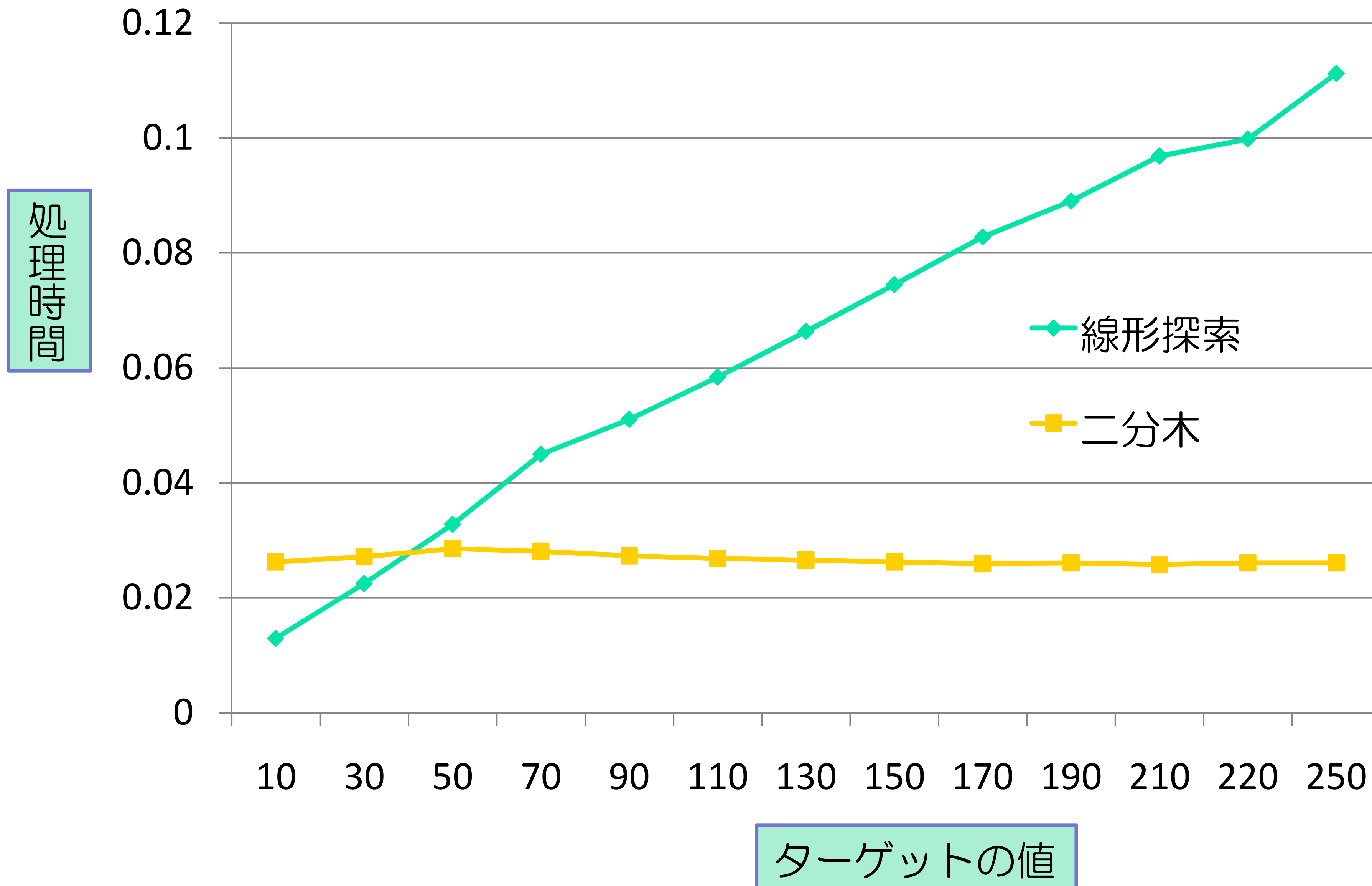
- 比較回数がO(N)からO(logN)に
  - やや複雑だが高速

```
/* 二分木探索版 */
int getIndex2(const short *inBuf, size_t num, short val)
{
	size_t i = 0, j = num;
	if (inBuf[0] > val) return 0;
	while (i < j) {
		size_t k = (i + j) / 2;
		if (inBuf[k + 1] <= val) {
			i = k + 1;
		} else {
			j = k;
		}
	}
	return i + 1 >= num ? -1 : i + 1;
}
```

ベンチマーク(その1)

Cybozu Labs

■ 線型探索は入力値が大きいほど遅くなる



Cybozu Labs

二分木は最速か?

- 二分木探索はCPUのパイプラインが止まりやすい
  - 最近のCPUには不向きなアルゴリズム
- 探索値の分岐に偏りがある
  - URLに使われる文字はアルファベットが大半で偏りもある
- Range Coderで使う場合はN = 256
  - 小さなNの場合，O記法では無視される定数項が重要に
  - 256(線型) vs log(256) = 8(二分木)

SIMDを使って線型探索を試みる

Cybozu Labs

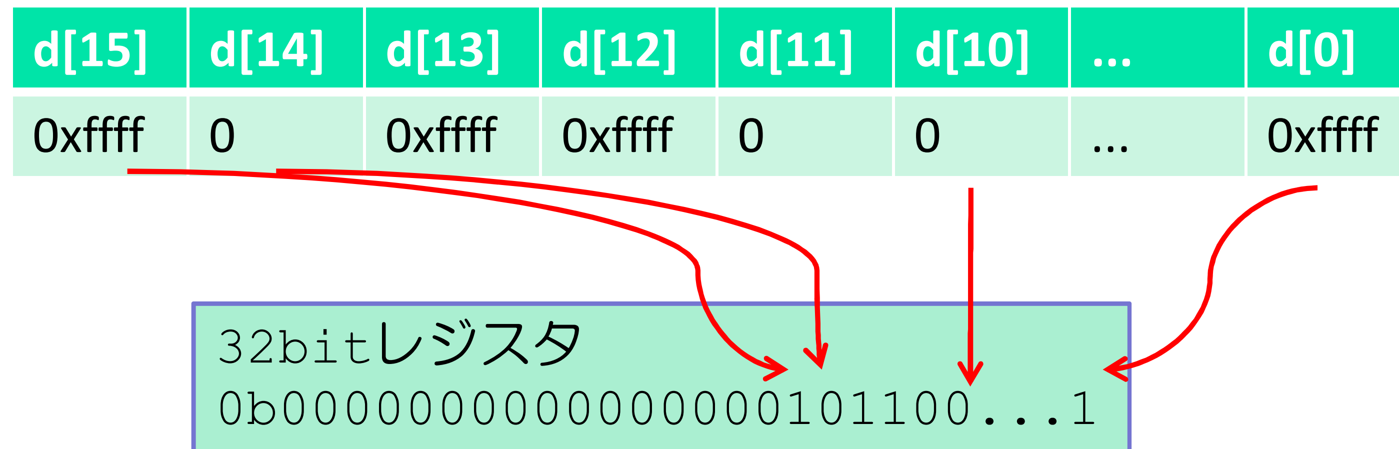
- SIMDとは
  - 複数個のデータを並列に処理する命令群
  - x86は8個(32bit)，16個(64bit)の128bitレジスタを持つ
  - short x 8個同時処理可能
    - 分岐処理には向かない
- 比較命令(cmplt)
  - 各要素に対してresult = reg1 < reg2 ? 0xffff : 0

cmplt	data[0]	data[1]	data[2]	... data[7]
reg1	1234	234	5553	34
reg2	2000	2000	2000	2000
result	0	0	0xffff	0

SIMDの命令

CybozuLabs

- MSBの集約命令(pmobmskb)
  - バイト単位での各要素のMSBを集めてくる



- ビット位置取得命令(bsf：non SIMD)
  - レジスタのbit0から初めて1になった位置を返す
- これらの命令を組み合わせて線形探索

SIMDによる線型探索

- 処理概要
  - 128x2ごとに処理すれば分岐は最大16回

Range Coderの復号処理

Cybozu Labs

■ 実際のコード

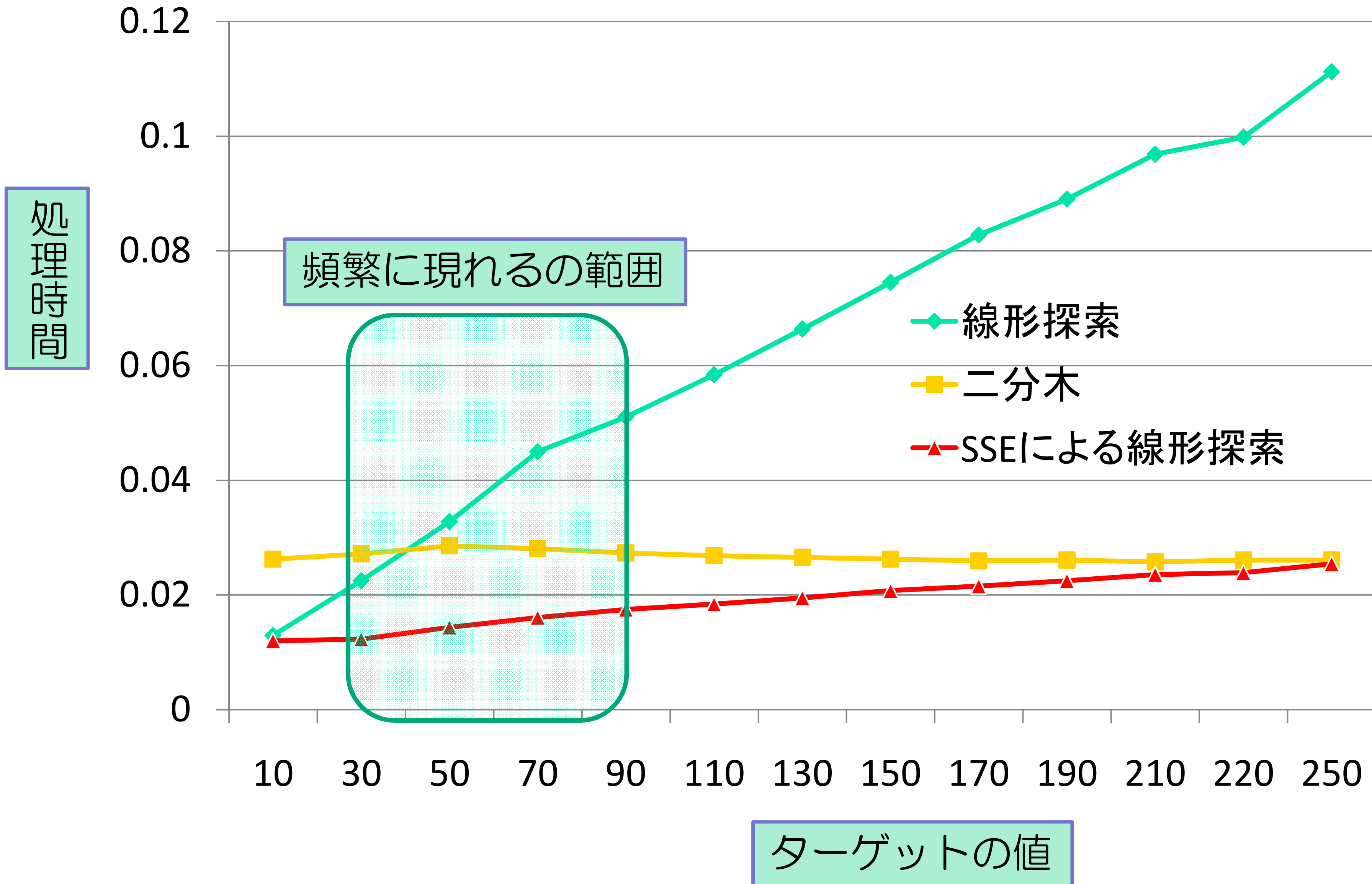
```
int getIndex2(const short *inBuf, size_t num, short val)
{
    if (inBuf[0] > val) return 0;
    __m128i v = _mm_set1_epi16(val);
    size_t i, mask = 0;
    for (i = 0; i < num; i += 16) {
        __m128i x = *(const __m128i*)&inBuf[i];
        __m128i y = *(const __m128i*)&inBuf[i + 8];
        __m128i a = _mm_cmplt_epi16(v, x);
        __m128i b = _mm_cmplt_epi16(v, y);
         mask = (_mm_movemask_epi8(b) << 16)
                     | _mm_movemask_epi8(a);
        if (mask) {
            return i + (bsf(mask) >> 1);
        }
    }
    return -1;
}
```

大雑把な比較

詳細な値の取得

ベンチマーク

Cybozu Labs



まとめ

Cybozu Labs

- 無条件にO(logN)がO(N)よりよいわけではない
  - Nが小さいとき
  - 前の係数が大きいとき
- SIMDはintrinsic関数を使うとお手軽に利用できる
  - 先のコードも小一時間で作成(検証時間込み)
  - 移植性も(x86限定だが)よい
    - intrinsic関数はVC/gcc共通．デフォルトで利用可能
    - 32bit/64bit共通

圧縮コーデックでの例

Cybozu Labs

■ 下記ループが全処理の70%を占めていた

```
int calc(int a, int b, int s) {
    const int Q = 1 << s, Q2 = Q * 2, Q3 = Q * 3;
    assert(s <= 16 && b < a && a < Q * 4);
    int n = 0;
    for (;;) {
        if (a < Q2) {
            n = n * 2;
        } else if (b >= Q2) {
            n = n * 2 + 1; b -= Q2; a -= Q2;
        } else if (b >= Q && a < Q3) {
            b -= Q; a -= Q;
        } else {
            break;
        }
        b = b * 2;
        a = a * 2 + 1;
    }
    return n;
}
```

コードの意味

- (a, b)の移動パターンと出力例
  - フラクタル的

quit

move

x 2

a

move

b

拡大

```
0
0 16
0   8   8
0   8   8 17
0   4   4   4   4
0   4   4   4   4 18
0   4   4   4   4   9   9
0   4   4   4   4   9   9 19
0   2   2   2   2   2   2   2   2
0   2   2   2   2   2   2   2   2 20
0   2   2   2   2   2   2   2   2 10 10
0   2   2   2   2   2   2   2   2 10 10 21
0   2   2   2   2   2   2   2   2   5   5   5   5
```

最適化への試行錯誤

- 斜辺に規則性は無いか見てみる

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
v	0	0	1	0	2	1	3	0	4	2	5	1	6	3	7	0	8	4	9	2	10

  - 2のべきのところは値が半分
    - これは一体なんの値だろう
  - 何度シフトすれば i が v になるのか調べてみる

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
s	0	1	1	3	1	2	1	4	1	2	1	3	1	2	1	5	1	2	1	3	1

  - なんとなく規則性がありそう

2進数で表記してみる

- 再掲

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
s	0	1	1	3	1	2	1	4	1	2	1	3	1	2	1	5	1	2	1	3	1

i	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000	10001	10010	10011	10100
z	1	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	0	1	0	2

- iを2進展開したときの下から連続する0の個数に注目
  - z + 1とｓに関係が!

bsfとbsr

- 斜辺の場合は
  - 求める値 = i >> "iの下からみて初めて1の位置(bsf) + 1"
- 一般の場合は?(同様に頑張って眺める)
  - a ^ bの上からみて初めての1の位置(bsr)
    - (斜辺のときと違う…. 統合できないのか.
- 斜辺のときは a = i, b = i - 1
  - x ^ (x - 1)は何を意味する

下から見て初めて1になるところ(bsf)

```
x       = ? ? ? 1 0 0 ... 0 0    ; ? は 0 または 1
x - 1   = ? ? ? 0 1 1 ... 1 1
 ---------------------------
x^(x-1)= 0 0 0 1 1 1 ... 1 1
```

上から見て初めて1になるところ(bsr)

```
bsr(x ^ (x - 1)) = bsf(x)
```

まとめ

Cybozu Labs

- 斜辺のときと一般のときを統合可能

```
int calc(int a, int b, int s)
{
    return b >> (bsr(b ^ a) + 1);
}
```

- 最適化の結果
  - ループが消滅
  - sに依存しないこともわかった

SIMD化への道

- bsr相当の命令はSIMDには無い
- $bsr(x) = \lfloor \log_2(x) \rfloor$
- 整数 x を浮動小数で表すと x = 2^e * f (1 <= f < 2)
- e, fは以下のbit表現で格納される(floatの場合)

bit	31	30...23	22...0
x	sign	e + 127	f

- $bsr(x) = \lfloor \log_2(x) \rfloor = e$
- floatへのcastはcvtpi2ps命令を使う(レジスタ操作)

```
/* bsrのSIMD用疑似コード */
int bsr(uint32 x)
{
    float a = (float)x;
    return (*(uint32*)&a >> 23) - 127;
}
```

注意1．x < (1<<23)の制約

注意2．左コードは非ANSI

FFTにおけるビット反転

Cybozu Labs

- FFTでのバタフライ演算では配列のアクセス時にビット反転によるアクセスが必要になることがある

```
int bitRevTbl[] = {
   0b00000000, // 0のビット反転
   0b10000000, // 1のビット反転
   0b01000000, // 2のビット反転
   0b11000000, // 3のビット反転
   ...
};

for (int i = 0; i < n; i++) {
    flot x = data[bitRevTbl[i]];
    ....
}
```

- 与えられたデータのビット反転命令はないため，上記のようにテーブルを使うことが多い

bitRevTbl[]相当のデータ生成

Cybozu Labs

- 音声処理ではn = 256, 1024などを使うことが多い
- 眺めてみる

idx	b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0
4	0	0	1	0	0	0	0	0
5	1	0	1	0	0	0	0	0
6	0	1	1	0	0	0	0	0
7	1	1	1	0	0	0	0	0
8	0	0	0	1	0	0	0	0
9	1	0	0	1	0	0	0	0

０と１が交互

００と１１が交互

００００と１１１１が交互

pmovmskbによるトリック

Cybozu Labs

- 各byteのMSBがそのパターンになるデータを生成

mm0	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
mm1	0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01

```
.lp:
    pmovmskb eax, mm0 // mm0のMSBを集約してeaxに代入
    paddb    mm0, mm1 // byte単位での並列加算(mm0 += mm1)
    // [eax]を使ってアクセス
    ...
    jmp .lp
```

Xbyak

- C++ヘッダのみで提供されるクラスライブラリ
  - Windows/Linux/Intel Mac OS X32bit専用
  - MMX/SSE/SSE2/SSE3/SSSE3対応
  - JITアセンブラ
  - C++内DSLアセンブラ
    - Xbyak::CodeGeneratorを継承してその中でニーモニックを記述
    - getCode()で生成された関数へのポインタを取得できる

```
class AddFunc : public Xbyak::CodeGenerator {
public:
    AddFunc(int y)
    {
        mov(eax, ptr[esp+4]);
        add(eax, y);
        ret();
    }
};
(int (*)(int)(AddFunc(5).getCode()(3); // 5 + 3
```

C++内DSLとしての魅力(その1)

Cybozu Labs

- gas/NASMなどのマクロを覚える必要がない
  - すべてC/C++の文法を利用できる
  - たとえばデバッグ時にコード中にあるレジスタの値を表示させたいときdebug(eax);で任意の位置で表示可能

```
void debug(const Reg32& r)
{
    static const char *str = "%s 0x%08x¥n";
    /* eax, edx, ecxがprintfで壊されないために退避 */
    push(eax); push(edx); push(ecx);
    push(r);
    push(r.toString());
    push((int)str);
    call((void*)printf);
    add(esp, 3 * 4);
    pop(ecx); pop(edx); pop(eax);
}
```

- Makefileやprojectファイルの管理が楽

C++内DSLとしての魅力(その2)

Cybozu Labs

- レジスタのaliasも分かりやすくできる
  - Const Reg32& data(eax);として以後dataを使う
- 複雑なループもアンロールせずに自然に記述できる

```
for (int j = 0; j < bit; j++) {
    movaps(t0, x);
    int s = pos + bit - 1 - j;
    if (s) {
        psrlq(t0, s);
    }
    andps(t0, mask0);
    paddd(out, t0);
    if (j < bit - 1) {
        movaps(t0, out);
        paddd(out, t0);
        paddd(out, t0);
    }
}
```

シフト量が0ならシフトさせない

ループの最後だけ実行させたくない

JITとしての魅力(その1)

Cybozu Labs

- 関数のパラメータを利用した最適化を実現しやすい
  - yを入力するとx → x + y を返す関数を作る

```
class AddFunc {
    const int y_;
public:
    AddFunc(int y) : y_(y) { }
    int add(int x) const { return x + y_; }
};
```

  - AddFunc::addの想定されるコード

```
proc addFunc::add
    mov     eax, dword [esp + 4]
    mov     edx, dword [ecx + 4] ; // y_
    add     eax, edx
    ret
```

JITとしての魅力(その2)

- Xbyakを使えばyを定数としたコードを生成できる

```
class AddFunc : public Xbyak::CodeGenerator {
public:
    AddFunc(int y)
    {
        mov(eax, ptr [esp + 4]);
        add(eax, y);
        ret();
    }
};
```

```
proc addFunc::get
    mov     eax, dword [esp + 4]
    add     eax, 5 // y = 5の場合
    ret
```

ペアリング暗号の実装

CybozuLabs

- ペアリング暗号
  - ペアリング
    - 楕円曲線暗号と有限体上にまたがって定義される写像
  - 既存のRSAなどでは不可能な機能が実現できるとして期待
- Xbyakを使ってペアリング写像を実装した
  - 既存の世界最速記録(ソース/バイナリ非公開)
    - 479usec@Opteron 2.2GHz(using SSE2)
      - 400程度usec@Core2Duo 1.8GHzとの情報(未確認)
  - Xbyakによる実装(ソース公開)
    - 175usec@Core2Duo 2.6GHz(using SSSE3)

実装しての感想(その1)

Cybozu Labs

- Xbyakを使ったメリット
  - SIMDのシフト命令は即値指定なものがあり，パラメータを変更させながら最速値を見るのが難しい
    - パラメータを自由に変更できるコードを書きやすい
  - やはりデバッグがしやすい
    - VCを使うと関数入力時に自動的に変数名ヒントがでるのも便利

```
void shrBit(const Xmm& L, const Xmm& H, const Xmm& t, int nBit)
{
    const int r = nBit / 8;
    const int q = nBit & 7;
    movaps(t, L);
    palignr(L, H, r);
    palignr(t, H, r + 1);
    psrlq(L, q);
    psllq(t, 8 - q);
    orps(L, t);
    shrBit(H, t, nBit);
    shrBit(|
}         ▲1 / 2▼  void shrBit(const Xbyak::Xmm &L, const Xbyak::Xmm &H
/*
```

感想(その2)

- 論文のアルゴリズムと実装の乖離
  - add/sub/mul/div/cb(3乗)/cbr(3乗根)を組み合わせ
  - それらの速度比から適切なアルゴリズムを選択
    - しかし論文で基にしている速度比が実情にあってない

	mul	cb	cbr	div
論文	500.9	39.4	1039.4	7711.1
自分	199	31.9	94.2	4807

    - cbrが遅いとされているため，各種論文では cbr を避けている
    - その選択は cb : cbr = 1 : 4以上のとき意味がある
      - 現状では cb : cbr = 1 : 3 のため逆に遅くなる
  - レジスタサイズが32bit前提から抜けられていない
    - 128bitを前提としたパラメータ選択が考察されていない
      - 例：SIMDではbit単位のシフトは遅いがbyte単位はそこそこ速い

toy VMでJITを体感

- フィボナッチ数列を計算できるtoyVMを作ってみる
- toyVMの仕様(2時間で作れるもの)
  - レジスタ　A, B：32bit, PC：program counter
  - メモリ 4byte x 65536：uint32 mem[65536];
  - すべての命令は4byte固定
  - 即値はすべて16bit
- 命令群

命令(R = A or B)	意味
vldiR, imm	R = imm
vldR, idx / vstR, idx	R = mem[idx] / mem[idx] = R
vaddiR, imm / vsubiR, imm	R += imm / R -= imm
vaddR, idx / vsubR, idx	R += mem[idx] / R -= mem[idx]
vputR	print R
vjnzR, offset	if (R) { PC += (signed)offset; }

フィボナッチ数列

Cybozu Labs

- 再帰は面倒なのでループにする

  Cによるコード

```
void fibC()
{
    int p, c, n, t;
    p = 1;
    c = 1;
    n = 10000;
lp:
    t = c;
    c += p;
    p = t;
    n--;
    if (n != 0) goto lp;
    printf("%d¥n", c);
}
```

```
/* 変数の割り当て */
A : c, B : temporary
mem_[0] : p
mem_[1] : t
mem_[2] : n
    vldi(A, 1); // c
    vst(A, 0); // p(1)
    vldi(B, n);
    vst(B, 2); // n
// lp
    vst(A, 1); // t = c
    vadd(A, 0); // c += p
    vld(B, 1);
    vst(B, 0); // p = t
    vld(B, 2);
    vsubi(B, 1);
    vst(B, 2); // n--
    vjnz(B, -8);
    vput(A);
```

通常のC++によるVMの実行部

Cybozu Labs

```
void run()
{
    bool debug = false;//true;
    uint32 reg[2] = { 0, 0 };
    const uint32 end = code_.size();
    uint32 pc = 0;
    for (;;) {
        uint32 code, r, imm;
        decode(code, r, imm, code_[pc]);
        switch (code) {
        case LDI:
            reg[r] = imm;
            break;
        case LD:
            reg[r] = mem_[imm];
            break;
        case ST:
            mem_[imm] = reg[r];
            break;
        case ADD:
            reg[r] += mem_[imm];
            break;
            ...
```

4byte命令を読み込んで命令セットに分解

各命令を実行

プログラムカウンタが終わるまでループ

```
        ....
        pc++;
        if (pc >= end) break;
    } // for (;;)
```

XbyakによるVMのリコンパイル

Cybozu Labs

- リコンパイルの方針
  - A：esi, B：edi, mem：ebxを利用することにした

```
const Reg32 reg[2] = { esi, edi };
const Reg32& mem(ebx);

xor(reg[0], reg[0]);
xor(reg[1], reg[1]);
mov(mem, (int)mem_);
const uint32 end = code_.size();
uint32 pc = 0;
uint32 labelNum = 0;
for (;;) {
    uint32 x = code_[pc];
    uint32 code, r, imm;
    decode(code, r, imm, x);
L(toStr(labelNum++));
    switch (code) {
    case LDI:
        mov(reg[r], imm);
        break;
```

レジスタのalias

実際のx86のコードを記述

生成されたx86コード(その1)

Cybozu Labs

- メモリアクセスが多い(当然)

```
 xor    esi,esi
  xor    edi,edi
  mov    ebx,0EFF20h
  mov    esi,1
  mov    dword ptr [ebx],esi
  mov    edi,2710h
  mov    dword ptr [ebx+8],edi
.lp:
  mov    dword ptr [ebx+4],esi
  add    esi,dword ptr [ebx]
  mov    edi,dword ptr [ebx+4]
  mov    dword ptr [ebx],edi
  mov    edi,dword ptr [ebx+8]
  sub    edi,1
  mov    dword ptr [ebx+8],edi
  test   edi,edi
  jne    .lp
```

リコンパイラの改良

CybozuLabs

- mem[0], mem[1], mem[2]をレジスタにする
  - 修正箇所
    - immが0, 1, 2の時のみメモリではなくレジスタ参照

メモリの変わりに
使うレジスタ

```
const Reg32 memTbl[] = { eax, ecx, edx };
const size_t memTblNum = NUM_OF_ARRAY(memTbl);
for (size_t i = 0; i < memTblNum; i++) {
    xor(memTbl[i], memTbl[i]);
}

case ADD:
    if (imm < memTblNum) {
        add(reg[r], memTbl[imm]);
    } else {
        add(reg[r], ptr [mem + imm * 4]);
    }
```

生成されたx86コード(その2)

CybozuLabs

■ メモリアクセスを減らせた

改良前

```
.lp:
  mov    dword ptr [ebx+4],esi
  add    esi,dword ptr [ebx]
  mov    edi,dword ptr [ebx+4]
  mov    dword ptr [ebx],edi
  mov    edi,dword ptr [ebx+8]
  sub    edi,1
  mov    dword ptr [ebx+8],edi
  test   edi,edi
  jne    .lp
```

改良後

```
.lp:
  mov    ecx,esi
  add    esi,eax
  mov    edi,ecx
  mov    eax,edi
  mov    edi,edx
  sub    edi,1
  mov    edx,edi
  test   edi,edi
  jne    .lp
```

ベンチマーク

Cybozu Labs

- n = 10000のときにかかった時間(clk@Core2Duo)

	通常VM	JIT	改良版JIT	native C
clk	1216K	136K	101K	84K

- 通常VMに対してリコンパイル(JIT)版は1桁違う
  - case + jmpによるVMはパイプラインが乱れる
- 改良版JITはnative Cにも遜色ない
- 簡単(JIT版VMは100行未満)に作れて遊べる
  - これはもちろんVMが簡単だからだが…