Linux/Alpha活用講座

清水 尚彦　nshimizu@keyaki.cc.u-tokai.ac.jp

第19回 プロファイルとチューニングのコツ

Kondara MNU/Linuxのリテールパッケージを入手してはいたものの、現役で仕事に使うマシンの環境を変える勇気は出ずにぐずぐずしていたのですが、折りよく(?)私のメインマシン(EB164のマシン)はついにハードディスクが昇天してしまいました。そこで、新しいハードディスクに交換するついでに、複数のディストリビューションを切り替えて評価できるように少し大きめのハードディスクを導入しました。といっても、まずは常用することになるDebian GNU/Linux(以下、Debianと略)の環境を構築しないことには話が始まりません。前はARCコンソールからMILOを使ってLinuxを立ち上げていたのですが、NTのサポートもなくなっていくことだし、今さらMILOでもないだろうということで、SRMにファームウェアを載せかえることにしました。今月は、まずSRMファームウェアのインストールの奮戦記からお伝えします。

SRMコンソールファームウェアのインストール

コンパックのサイトからEB164のファームウェアとファームウェアアップデートユーティリティをダウンロードします。ライセンス条項にはデジタル製品に使うためにダウンロードして使うことは許されています。ただし、ダウンロードしたソフトを他人に渡すことは禁止されていますので、各自の責任でダウンロードしてください。

```
http://ftp.digital.com/pub/DEC/Alpha/firmware/
archive/
```

ここにはAlphaPC164LXなど良く使われているマザーボードのSRMコンソールもありますので、NTをもう使わないという人は試してみてください。必要なファイルは各マザーボードのディレクトリにある*.romとfwupdate.exeです。このユーティリティは、ARCでもSRMでも利用できるものになっています。また、操作ミスによってどちらも立ちあがらなくなってしまったときには(実は私はおっちょこちょいなのでこれをやってしまいました)、ボードによってはジャンパを設定した後にシリアルポートに接続したコンソールからデバッグモニタを立ち上げて、ここからロードすることもできます。

簡単にできそうですが、私は最初うまくいかずに苦労しました。というのは、このアップデート用のフロッピーディスクを作成するのに、入手したばかりのノートPCを使ったのですが、Windows 98でフォーマットしたフロッピーディスクだと、EB164のARCコンソールもデバッグモニタも、正しいファイルシステムとしては認識しなかったのです。ARCからは手も足もでないのでデバッグモニタに切り替えて調査しました(こういったことをするときには、現役を引退しているHP100LXも、クロスケーブルでつなぐと簡単にコンソールになるので便利です)。デバッグモニタにはフロッピーディスクのディレクトリを表示するfldirコマンドがあるので、これで見たところ何やら本来のファイル以外のエントリが見えます。結局、Windows98で作成したフロッピーは使えないという結論を出しました。

なんとかならないかと考えて、ノートパソコンにインストールしておいたLinuxを立ち上げてmformatでフォーマットし直し、必要なファイルをコピーしました。もう一度ファームウェアをARCに戻して、ARCからファームウェアアップデートのメニューを起動すると、無事にファームウェアのインストールができました。再度リブートしてSRMを立ち上げたつもりだったのですが、CRTは青い画面のまま。実はデバッグモニタから使っているときにいつの間にかコンソール出力がシリアルポートになっていたのでした。例によってシ

リアルポートに接続したHP100LXから

```
# set console graphics
```

というコマンドを入力してCRTにメッセージを出すように変更しました。これでようやく普通に使えるようになります。ここまでくればしめたもので、5月号に書いた手順でインストールをすればいいのです。といっても、何をやるにも何かしらおかしなことが起こるのが常の私は、今回もfdiskで引っかかりました。BSD形式のパーティションを設定して最初のaパーティションを約4GB確保しました。次に、スワップ領域を200MBytes程度取ったつもりでいたのですが、シリンダ指定でなく" +200MBytes "と容量を指定して領域を確保したところ、インストーラがスワップの設定に失敗します。良く見るとbパーティションがとんでもなく大きくなっています。どうやらfdiskの領域確保のプログラムにlong／intの取り違いによるバグがありそうです。シリンダ番号を直接指定することで回避できますから、ソースを追いかけて追求してはいません。

CD-Rを使ってみよう

Debianにはcdrecordのパッケージがあって、これを使ってCD-Rを焼くことができます。ところが、実際には使い方によっては、メモリが確保できないというメッセージが出て動きません。これは、AlphaとPowerPCのカーネルだけは定義ファイルが古いまま放置されていて(こんなの多いですね)、必要な大きさの共有メモリが確保できないからです。こんなときには迷わずカーネルを修正して再構成しましょう。

```
/usr/src/linux/include/asm-alpha/shmparam.h
```

の中の

```
#define SHMMAX 0x3fa000
```

という定義があります。i386では0x2000000になっているのですが、コメントにAlpha では24bitを超えてはいけないとなっているので、それ以内に収まるように

```
#define SHMMAX 0xffa000
```

にしました。これでcdrecordも問題なく動くようになりました。データのCDだけでなく自分の好きな曲を集めた音楽CDも焼いてみたいと思ったので、cdparanoiaをインストールしました。どちらもDebianのパッケージに入っています。

●Debian 2.1r4/AlphaのCD入手方法

Debianのi386版のCDは、書店でもパソコンショップでも比較的簡単に手に入りますが、Alpha版の販売はなかなかなされておらず、専用線接続以外の人が使ってみるには敷居が高いものでした。Debianのサイトを見ていたら日本でもAlpha版のCDを販売しているところがあったので紹介します。

```
http://www.hypercore.co.jp/
```

Kondara MNU/LinuxにもAlpha版のパッケージが入っていますが、選択肢は多い方がいいですね。実はDebianのAlpha版の入手が困難だったので、CD-Rを導入したと同時に必要な人の間で回覧をするようにオフィシャルCDの作成を行おうと準備していたのですが、一般に販売されているものを入手するほうが簡単なので回覧は止めました。

Octaveの高速化

MATLAB互換の数値計算簡易言語であるOctaveは、ちょっとしたアルゴリズムの評価などに大変便利で、私は良く使っています。せっかくCPML(…)やCXML(…)などのライブラリがCompaqからリリースされているのですから、これらを使ってOctave を高速化することを検討してみましょう。

4月号に書きましたが、連立一次方程式の簡単なベンチマークで評価すると問題によっては数倍から10倍程度の高速化がなされるケースがあるのでCXMLの利用は魅力的です。数値計算をターゲットとしたときのCompaq Fortranは優れた性能を示すのですが、今回は利用しません。興味のある方は試してください。今回は、OctaveにCXMLをリンクできるように修正することにします。Octaveの配布サイトから最新版を取ってきてもいいのですが、ここでは簡単にDebianのパッケージシステムのソースを入手します。この方法は必ずしも最新のパッケージが入手できないのが難点ですが、それ以上に配布バイナリと同じ世代のプログラムでコンパイルができることが確実になっているソースを入手できることに意味があります。

●debian/rulesの変更

このスクリプトが$(MAKE)を起動して実際にバイナリを作成する部分であるmake-stampの引数を次のように追加変更します。

```
LDFLAGS=-s          \
SPECIAL_MATH_LIB="-lcxml -lcpml" \
```

これはmakeに渡す環境変数の追加になります。スタティックなライブラリを使いたい場合には次のように関連ファイルをすべて並べます。

```
SPECIAL_MATH_LIB="/usr/lib/libcxml.a /usr/lib/\
libFutil.a /usr/lib/libfort.a/usr/lib/libots.a\
/usr/lib/libcpml.a"
```

次にDebianのディストリビューションの不正な時間設定を修正します。rootでログインして

```
# touch /usr/lib/gcc-lib/alpha-linux/egcs*/include/*
```

とします。これをしていない場合、インクルードファイルの依存関係が指定されているにも関わらず未来の時間のインクルードファイルが存在するため、makeがループして終了しないことがありました。

liboctaveの中のlo-ieee.ccのAlpha固有処理の部分をLinuxでは使わないように変更します。新しいOctaveのソースファイルではすでに修正されていますが、この部分はOSF1用の処理になっていてLinuxでは未定義な定数を使っているためです。2カ所あるので同様に修正してください。

```
if defined(__alpha__) && ! defined(linux)
```

のようにLinuxではこの部分を参照しないように変更します。私のように新旧を比較しようと思うのでなければ、修正済みの新しいソースを取ってきたほうがいいと思います。対象とするベンチマークは次の6本です。実行時間をOctaveのtime変数で測定してグラフ1に記載します。ベンチマークは、グラフの凡例にある6種類です。これは

```
t1=time;a\b;time-t1
```

というコマンドによって実行しました。

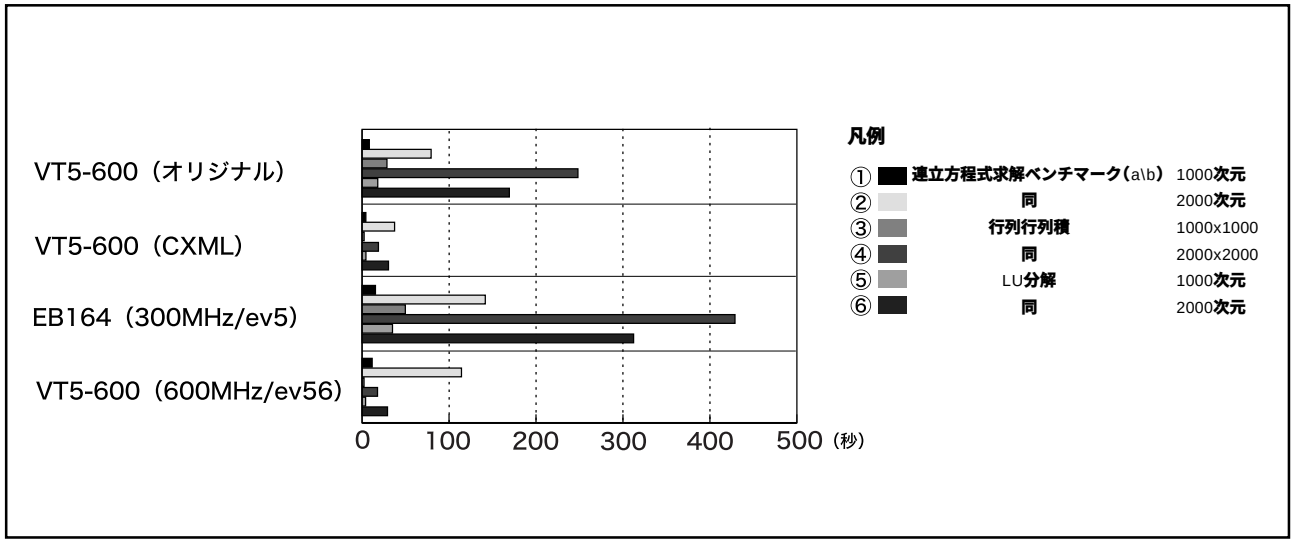
このように有意義な程度の高速化が計られています。コンパックからリリースされたFortranコンパイラを使って作成すればもっと高速化すると思い、試してみましたが、Octaveのテストが通りません。行列演算部分は通るのですが、関数のEXPと微分方程式系のテストでエラーが出てしまいました。行列演算だけを行う人は最速のOctaveとなるかもしれないので試す価値はあるでしょう。エラーの詳細は解析していないので、利用する場合には慎重に検討してからにしてください。

ベンチマークついでに最近流行っているMP3のエンコードを試してみましょう。ベンチマークの素材はGPLで配布されているLAMEです。評価に使ったバージョンは3.63です。LAMEのホームページは次のURLとなります。

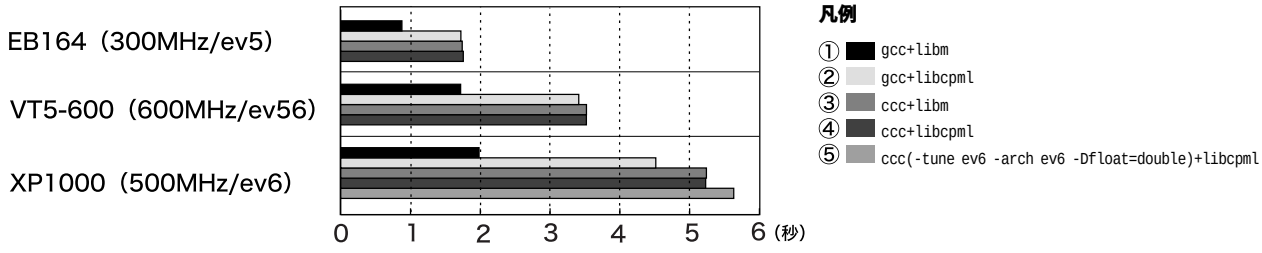
```
http://www.sulaco.org/mp3/
```

FFTなどCXMLにある処理も使っているのですが、それを使うには変更量が多くなって大変なので、ここではCPMLとCCC(…)だけを使ってみます。EV5用のバイナリを使って評価しますから、XP1000の実力を出し切ってはいないことに注意してください。評価は、約70MBytesの実際の音楽WAV

グラフ1



グラフ2



ファイルの演奏時間を、エンコードするときにかかった時間で割った倍率で表示します。プログラムはコンパイラとライブラリの組み合わせで次の5種類になります。結果はグラフ2に示します。

①gcc+libm
②gcc+libcpml
③ccc+libm
④ccc+libcpml
⑤ccc(-tune ev6 -arch ev6 -Dfloat=double)+libcpml

次に、音質を上げるオプションをつけて作成するベンチマークをしましょう。以下のように実行します。

```
$ lame -h -m s track1.wav track1.mp3
```

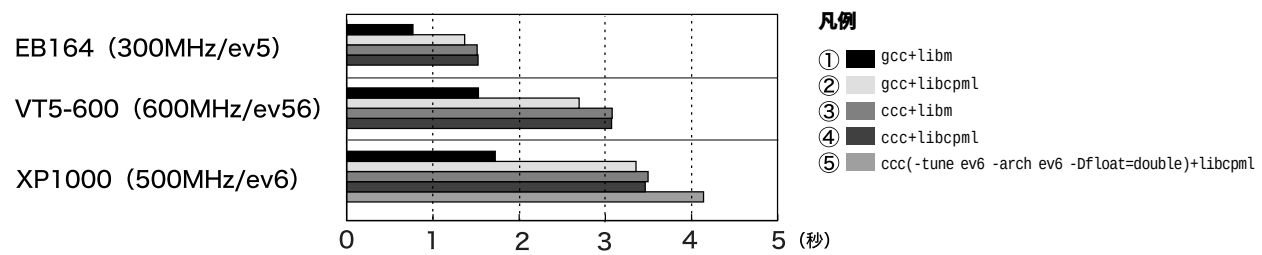
結果はグラフ3のようになります。

同じデータの情報ではないので参考までとなりますが、LAMEのホームページに載っているPentium III(600MHz)とAthlon(650MHz)の倍率はそれぞれ2.9 倍、4.05倍となっています。ほとんどキャッシュヒットすると思われるこの手の問題では、クロック周波数と性能はほぼ比例するはずなので(EB164、VT5の結果もそうなっています)、高速なクロックのマシンを持ってくればもっと速くなります。もっと速くしたいときには「午後のこ〜だ」でやっているようなSIMD命令への対応などをするか、もしくは若干のコードスケジュールをする必要があるでしょう。特にSIMD化はx86系で高い性能を実証しているので期待が持てます。この手の問題にAlphaのビデオ系のSIMD命令はどれだけ役に立つのでしょうか？　興味があるところです。21264でいくつかの拡張命令が導入されていますが、オプションの変更で高速化するのでそれらを使うとそれなりに効果があることが分かります。

時間を計って比較するだけならどのパソコン雑誌でもできるから面白くないので、例によってプロファイルを取ってプログラムの振る舞いをもう少し調べてみましょう。プロセッサによって処理の得手不得手があるので、ここではXP1000とEB164の両方でそれぞれプロファイルを取り、その違いについても検討してみます。

実用的には高音質となるオプションを選択して利用することが多いはずなので、プロファイルも高音質のものを取りま

グラフ3



す。実行速度はコンパックのコンパイラの方が速いのですが、最適化の影響からかプロファイルを正確に取れていないのでgccで取ったプロファイルをもとに検討します。

実行例1、2がプロファイルの結果です。呼びだし関数の時間割合が2つのプロセッサで若干違うことが分かると思います。両方のプロセッサで一番時間がかかっている処理は「quantize_xrpow」という関数です。この関数はテーブルを参照しながらアナログ値を離散情報に量子化する関数になりま

実行例1　XP1000でのプロファイル

```
Each sample counts as 0.000976562 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 21.67     23.52    23.52  1100652     0.02     0.02  quantize_xrpow
 16.38     41.30    17.78   995904     0.02     0.02  window_subband
  7.86     49.84     8.54  7327437     0.00     0.00  count_bit_noESC
  5.49     55.80     5.96    27664     0.22     0.52  L3psycho_anal
  5.00     61.22     5.43  9714795     0.00     0.00  ix_max
  4.89     66.53     5.31    55328     0.10     0.10  fft_long
  4.21     71.10     4.57   471810     0.01     0.01  calc_noise1
  3.25     74.62     3.53  1718880     0.00     0.00  mdct_long
  3.14     78.03     3.40  9063583     0.00     0.00  count_bit_noESC2
  3.05     81.34     3.31  1100663     0.00     0.03  count_bits
  2.98     84.58     3.24    55328     0.06     0.06  fft_short
  2.56     87.36     2.78  1084135     0.00     0.01  count_bits_long
  2.26     89.81     2.45 13111281     0.00     0.00  putbits
  1.91     91.89     2.07  2351754     0.00     0.00  count_bit_ESC
  1.78     93.81     1.93    55250     0.03     0.81  outer_loop
  1.46     95.40     1.58    13832     0.11     1.66  mdct_sub48
  1.25     96.75     1.36  9681761     0.00     0.00  choose_table
  1.07     97.92     1.16   454158     0.00     0.00  amp_scalefac_bands
  1.00     99.00     1.09 20681352     0.00     0.00  BF_addEntry
```

実行例2　EB164でのプロファイル

```
Flat profile:

Each sample counts as 0.000976562 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 17.65     53.70    53.70  1239346     0.04     0.04  quantize_xrpow
 10.57     85.85    32.15  1124136     0.03     0.03  window_subband
  8.43    111.49    25.64  1239357     0.02     0.09  count_bits
  8.23    136.53    25.04   529985     0.05     0.05  calc_noise1
  7.91    160.60    24.07    31226     0.77     1.36  L3psycho_anal
  7.59    183.69    23.09  8365812     0.00     0.00  count_bit_noESC
  4.31    196.79    13.11 10321039     0.00     0.00  count_bit_noESC2
  4.07    209.18    12.38 10946419     0.00     0.00  ix_max
  3.73    220.53    11.36    62452     0.18     0.18  fft_long
  2.90    229.35     8.81  1941184     0.00     0.00  mdct_long
  2.49    236.91     7.57  1221148     0.01     0.02  count_bits_long
  2.35    244.05     7.13    62452     0.11     0.11  fft_short
  1.77    249.42     5.37    15613     0.34     2.98  mdct_sub48
  1.75    254.75     5.32   509714     0.01     0.01  amp_scalefac_bands
  1.68    259.85     5.10 23330038     0.00     0.00  BF_addEntry
  1.54    264.54     4.69  2541216     0.00     0.00  count_bit_ESC
  1.29    268.47     3.94 14821377     0.00     0.00  putbits
  1.11    271.84     3.37 10910023     0.00     0.01  choose_table
  1.10    275.17     3.33    62374     0.05     2.27  outer_loop
```

す。この時間が一番重要なのはどのプロセッサにとっても同じで、ソースを見ると、x86ではインラインアセンブラで記述した関数を呼ぶように書かれています。Cで記述された部分は8回のアンローリングをしていますが、この関数でもっとも処理時間がかかる処理は、浮動小数点から整数への変換の部分です。実は21264からは浮動小数点レジスタから整数レジスタへの転送命令ができたのですが、それ以前のプロセッサではそういった命令はないため、一旦メモリにデータをストアしてからロードし直すことになるのです。この処理は多くのプロセッサにとって処理の難しいもので時間がかかります。LAMEのコードは単純なアンロールをしているだけなのですが、こういったコードを見るとソフトウェアパイプラインで高速化したくなるのは私だけでしょうか？ :-)

その次に実行時間のかかるwindow_subbandにしても、同じく整数と浮動少数点数の変換処理が頻繁に入っています。

では続いてXP1000で実行時間を占めている「count_bit_noESC」を見てみましょう。この関数でループを形成している部分を見てみるとデータ依存で処理を変える部分があります。

```
if(x != 0) {
  sign++;
  x *=16;
}
if(y != 0) {
  sign++;
  x += y;
}
```

こういったところは分岐予測が当たりにくく、パイプラインの深いマシンやアウトオブオーダーでペンディング命令数が多くなった場合などに、キャンセルされる命令が増えてしまうため、性能が出しにくい処理になります。アセンブラを手で書くときには、条件移動命令などを上手に組み合わせて分岐を使わずに記述できる可能性がありますから、試してみると面白いかもしれません。

このようにプロファイルをもとに実行時間の上位を占める関数を中心として、高速化の方法を探ることがチューニングの基本です。今回は具体的なチューニングにまでは踏み込んでいませんが、興味のある方はこの記事をヒントに試してください。コンパイラが上手に命令を生成できない部分では、インラインアセンブラを使うことで、かなりのことができるはずです。

具体的なプログラムをもとにプロファイラの使い方やその結果を用いたプログラムの簡単な解説を試みました。読者の皆様のプログラムの解析の助けになれば幸いです。今回の例ではコンパックのコンパイラの出力をプロファイラにうまく渡すことができず、最速のプログラムでの性能確認はできませんでしたが、動作の傾向はgccもほぼ同様であると考えられます。プロファイルの結果とプログラムをじっくり眺めると、いろいろなアイデアが浮かんでくるのではないでしょうか？（これ以上どうしようもないという洞察が得られる人もいそうですね） 今回の結果を見ると21264の新しい命令を使ってさらに処理を高速化したくなりますね。