DCPI for OpenVMS -- システム・マイクロスコープのご紹介

Anders Johansson 著

Anders は、OpenVMS 開発エンジニアリング・グループのカーネル・ツール・チームの主席ソフトウェア・エンジニアで、DCPI for OpenVMS のプロジェクト・リーダーを務めています。ほかに関わっているプロジェクトとしては、OpenVMS I64 の LIBRTL および SDA の分野の開発作業があります。スウェーデンのストックホルムを本拠に 16 年勤続している Anders は、スウェーデンの OpenVMS Ambassador でもあります。

はじめに

ご自身のアプリケーションのパフォーマンスや、システムのどの部分が最も頻繁に使用されているか、あるいはコードのどの部分がボトルネックになっているか知りたいと思ったことが何度もあるのではないでしょうか。OpenVMS システムでパフォーマンスを測定するための製品はいくつかあります。これらの製品の大半はソフトウェアのパフォーマンス・カウンタを使用していますが、それらのカウンタにはいくつかの制約があります。

HP (Digital) Continuous Profiling Infrastructure (DCPI) for OpenVMS では、これらの制約を回避するために、Alpha チップのハードウェア・パフォーマンス・カウンタを使用しています。DCPI によって、システムのより詳細な状況を知ることができます。データの分析時に、DCPI は個々の実行イメージで消費された時間から、実行イメージで実行された命令に至るまで、幅広い情報を生成します。また、命令のストールやデータ・キャッシュ・ミスなどが生じた位置に関する情報も提供します。600 MHz の Alpha システムの場合、DCPI の通常のサンプリング頻度はシステム内の各 CPU において 1 秒あたり 10000 回です。DCPI はこれを、CPU に対する最小限の負担（通常は、利用可能な CPU 時間の 5% 未満）で実現します。

DCPI for OpenVMS について

DCPI は、Alpha プロセッサにおけるプログラムの実行を高速化するための最適化手法を探る研究プロジェクトに端を発しました。「Where have all the cycles gone?」（サイクルはいったいどこへ消えたのか？）と名付けられたこのプロジェクトの成果として、DCPI の最初のバージョン（Tru64 UNIX で利用可能）と、次の Web サイトから入手できる「SRC Technical Note 1997-016A」が得られました。

http://h30097.www3.hp.com/dcpi/documentation/SRC%20Technical%20Note%201997-016a.htm

OpenVMS への DCPI の移植の可能性を探る調査は 1999 年の末に始まりました。移植作業の大部分は、2000 年に行われ 2001 年の初めに完了しました。その後、DCPI は OpenVMS エンジニアリング・グループ内でパフォーマンス問題の特定のために利用されました。2002 年の初めに、DCPI for OpenVMS が一般に公開されました。次の OpenVMS Web サイトから、フィールド・テスト・ライセンス条項への同意を前提として、「advanced development kit」として入手できます。

http://h71000.www7.hp.com/openvms/products/dcpi/

DCPI は、インストラクション・レベルのシステム・プロファイリングのための基盤を提供します。通常は、DCPI を使用するためにプロファイル対象のコードを変更する必要はありません。Alpha チップ自身のハードウェア・パフォーマンス・カウンタに基づいて動作するからです。ユーザ・アプリケーションやサードパーティ製アプリケーションの他、実行時ライブラリ、デバイス・ドライバ、VMS エグゼクティブ自身など、システム全体のデータが収集されます。その後、収集したデータを使用して、プログラム・レベル、ルーチン・レベル、およびインストラクション・レベルの環境のプロファイリングを実行できます。

DCPI for OpenVMS の動作の仕組み

DCPI は、次の主要なコンポーネントで構成されています。

1. データ収集サブシステム。これには、デバイス・ドライバ、デーモン・プログラム、および DCPI デーモンに対するユーザ介入のためのデーモン制御プログラム（dcpictl）が含まれます。
2. データ分析ツール。収集したデータをイメージ、ルーチン、コード行、インストラクションの各プロファイルにブレークダウンするために使用します。
3. OpenVMS デバッガの特別なバージョンの共有イメージ。
4. 動的なコードのデータ収集のための API を含んだ共有イメージ。

DCPI デバイス・ドライバ（DCPI$DRIVER）の役割は、Alpha チップのハードウェア・パフォーマンス・モニタリングを制御するチップ・レジスタとのインタフェースをとり、Alpha チップのパフォーマンス・カウンタがオーバフローしたときに生成されるパフォーマンス・モニタ割り込みを処理することです。割り込みハンドラは、割り込み時に取得したデータを常駐メモリに格納します。このドライバによって格納されるデータは、イベントのタイプ、PC および PID です。

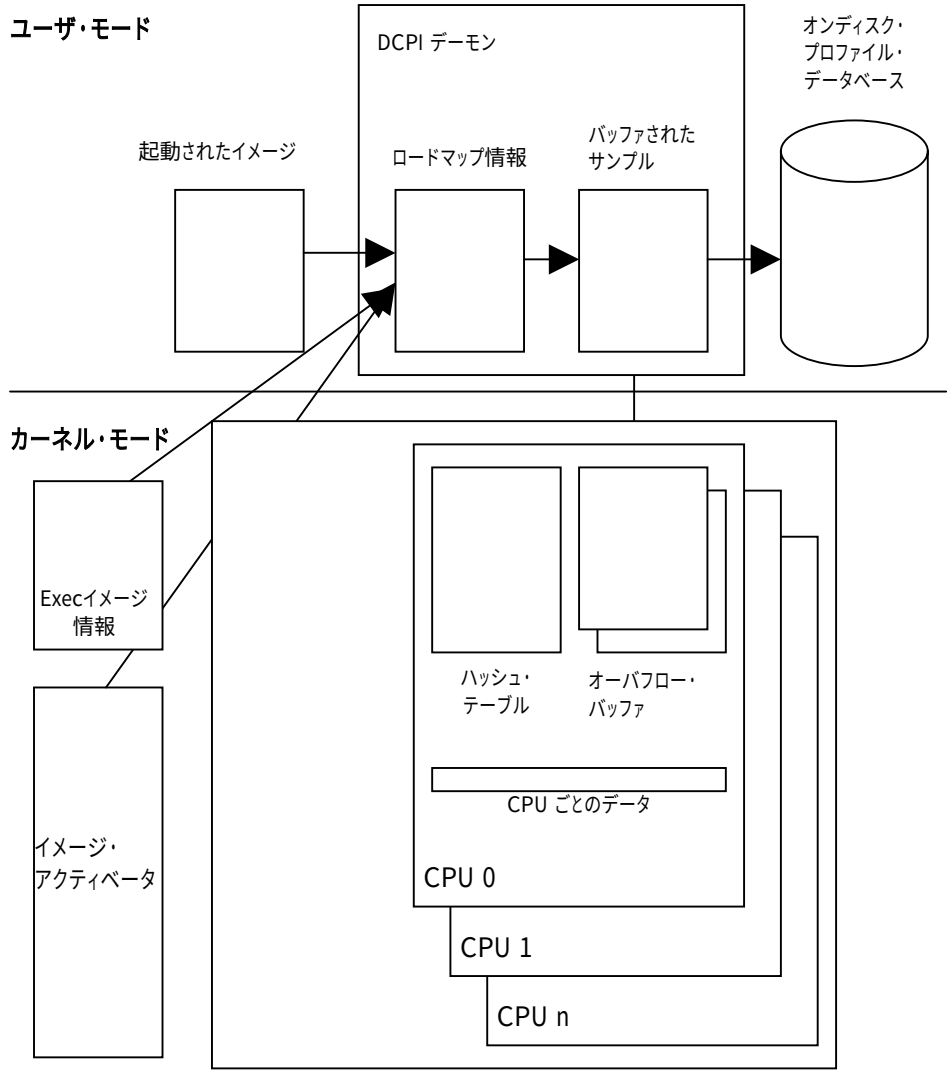
OpenVMS の対話型プロセスとして実行される DCPI デーモンは、実行するモニタリングの種類の指定や、データ収集の開始と停止を行います。データ収集時の DCPI デーモンの主な作業は次のとおりです。

- ドライバ・バッファからのデータの読み取り
- イベント、PC、PID をイメージ名とイメージ・オフセットのペアにマッピングする
- それらを後の分析のために、オンディスク・プロファイルへ格納する

このマッピングを行うために、DCPI デーモンは、システム上の各プロセスで実行中の稼動イメージのインメモリ・マップの他、exec ロード・イメージ・リストのマップを必要とします。DCPI デーモンは初期化時に、exec ロード・イメージ・リスト、およびシステム内で実行中のすべてのプロセスの稼動イメージを含んだ「ロードマップ」を作成します。さらに、稼動中のシステム上でマッピングを正しく行うためには、DCPI デーモンは、システム上で実行中の全プロセスにおける以降のイメージ起動を追跡する必要があります。イメージ起動の追跡は、OpenVMS V7.3 以降で利用可能になった「イメージ・アクティベータ・フック」を使用して行われます。この追跡機能は、OpenVMS イメージ・アクティベータと DCPI デーモンとの間のメールボックス・インタフェースによって実現されています。このインタフェースでは、イメージ・アクティベータが DCPI デーモンに対して、システム上のすべてのイメージ起動に関する詳細な情報を提供します。

データ分析ツールは、収集されたデータをさまざまな角度から分析します。これらのツールでプロファイル・データに対するルーチン名やソース・コードの対応付けを可能にするために、DCPI は独自バージョンの OpenVMS デバッガ共有イメージ（DCPI$DBGSHR.EXE）を使用します。ルーチンとソースの対応付けのために、DCPI は、実行イメージ(LINK/DSF)に対応するデバッグ情報を含んだイメージ(LINK/DEBUG)またはデバッグ・シンボル・ファイル(.DSF ファイル)も必要とします。つまり、システムで実行されているすべてのイメージのデータをデータ収集サブシステムが収集する一方で、分析ツールを使用して収集データの詳細な分析を実施するためには、デバッグ・シンボル情報が必要となります。

次の図は、DCPI のデータ収集の仕組みを示します。



DCPI のデータ収集の仕組み

Alpha チップのパフォーマンス・モニタリング

Alpha チップのハードウェア・パフォーマンス・カウンタを使用してパフォーマンス・データを収集する方法には次の 2 つの方法があります。

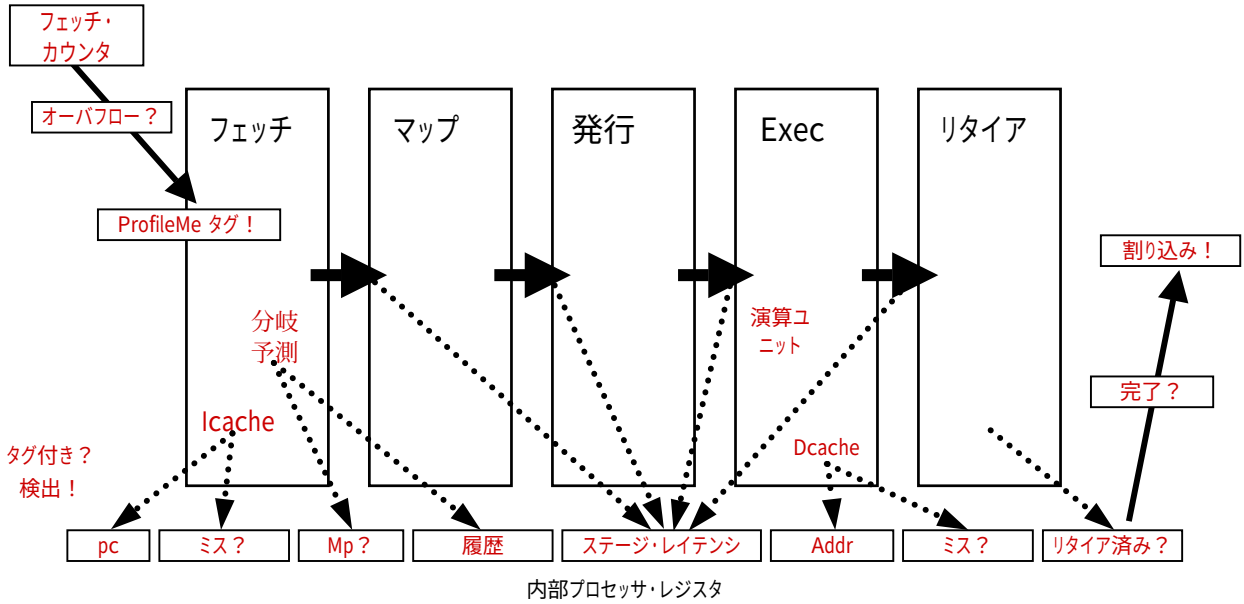
- 集約イベント
  この方法は、程度はさまざまですが既存のすべての Alpha チップで利用できます。この方法を使用して、DCPI は、Alpha プロセッサのパフォーマンス・モニタ制御レジスタに値（サンプリング周期）を設定し、サンプリング対象イベントも指定します。たとえば「CPU サイクルが実行された」など、指定したイベントが発生するたびに、そのイベントがカウントされます。イベント数がサンプリング周期として指定した値に達すると、割り込みが生成され、DCPI によって PC、PID、およびイベントのタイプが保存されます。たとえば、DCPI がサンプリング周期を 63488 に設定して CPU サイクル・イベントをサンプリングしている場合、Alpha チップは 63488 サイクルごとに割り込みを生成します。

  集約イベントの収集は、ソフトウェア・パフォーマンス・カウンタのデータを収集するよりも、信頼性の高いプロファイリング・データを収集する方法としてははるかに優れていますが、集約イベントの収集には不利な点もいくつかあります。この方法は、割り込み発生時にアクティブであった命令が、割り込みを発生させたイベントであることを前提にしています。つまり、パフォーマンス・モニタ割り込みが正確であることに依存しています。しかし、これは必ずしもそうではありません。少数のパフォーマンス・モニタ・イベントのみが正確な割り込みを生成するに過ぎません。また、EV6 以降の最近のプロセッサでは、これらの割り込みのいずれも正確ではありません。これは重大な問題のように見えるかもしれません。しかし、割り込みが正確でなくても、比較的予測可能であり、DCPI はそれを計算に入れています。

  この方法の別の問題として、「盲点」があるということが挙げられます。たとえば、IPL29 以上で実行されるコードはプロファイルの対象になりません。なぜなら、パフォーマンス・モニタ割り込みが IPL29 だからです。この盲点には Alpha のすべての PAL も含まれます。PAL コードは割り込みオフで実行されるからです。これらの場合、パフォーマンス・モニタ割り込みは、 PAL 呼び出し終了後の最初の命令、または IPL が 29 未満になったときに生成されます。

- ProfileMe
  EV67 以降で利用可能なこの方法は、多くの面で「集約イベント」よりも優れています。ProfileMe は、Alpha チップ上の固有の ProfileMe レジスタを使用します。DCPI が ProfileMe の周期を設定すると、CPU は命令フェッチをカウントします。設定した周期が経過すると（つまり、命令フェッチ・カウンタがオーバフローすると）、フェッチされた命令がプロファイル対象命令としてタグ付けされます。以降、この命令に関する情報は、命令の実行フェーズ全体を通して ProfileMe レジスタに記録されます。命令がリタイアすると割り込みが生成されます。この時点で、DCPI はオンチップの ProfileMe レジスタからすべての情報を読み出します。この方法によるパフォーマンス・データの収集ははるかに信頼性が高く、個々の命令のパフォーマンスについて、より詳細な実態を知ることができます。

  次の図は、ProfileMe を使ったサンプリングの流れを示します。



DCPI for OpenVMS のデータ分析ツール

分析ツールを使用するには、データ収集時に実際に使用されたイメージにアクセスする必要があります。なぜなら、分析ツールはイメージ・ファイルから命令を直接読み取るからです。読み取った命令が実行された命令でなければ、その分析は意味がないものになります。分析対象イメージがプロファイリング対象イメージと同じものであるかどうか確認のためのテストが行われます。この確認は、イメージ・ヘッダのさまざまなフィールドを調べ、それらの情報とデータ収集時に DCPI プロファイル・データベース（dcpidb）に格納されている情報とを突き合わせることで行われます。

分析ツールは、イメージまたはルーチン名に基づいてブレークダウン分析を実施できます。正常な分析を行うために、分析ツールは分析対象イメージに対応するデバッグ・シンボル情報を必要とします。この要件は 2 つの方法で満たすことができます。

- /DEBUG を指定してリンクされたイメージの使用。実行システムにイメージが INSTALL されている場合には該当しません。なぜなら、INSTALL するイメージは /NODEBUG リンクされている必要があるからです。

- /NODEBUG/DSF を指定してリンクされたイメージの使用。この方法でイメージをリンクすると、すべてのデバッグ・シンボル情報を含んだ独立のファイル（*imagename*.DSF）が作成されます。デバッグ・シンボル・ファイルをイメージ・ファイルと同じディレクトリに置きます。あるいは、デバッグ・シンボル・ファイルを別の場所に置き、そのディレクトリを指す論理名 DBG$IMAGE_DSF_PATH を定義します。

各データ分析ツールは、収集データについて少しずつ異なる情報を示します。厳密なプロファイリングを実施するには、次の表に示す複数の分析ツールを使用する必要があります。

ツール	説明
DCPIPROF	最上位の分析を行います。収拾されたサンプルについて、システムからイメージへのブレークダウンまたはイメージからプロシージャへのブレークダウンを提供します。通常、DCPIPROF は、どのイメージの処理にシステムの時間を使っているかを把握するために最初に使用するツールです。その後、DCPIPROF を使用して、イメージのどのルーチンに時間を使っているかを把握します。DCPIPROF 分析は、ProfileMe 方式および集約イベントにより収集されたデータを対象に実行できます。
DCPILIST	指定したイメージの特定のプロシージャ内の詳細な分析に使用します。DCPILIST は、収集されたサンプリング・データをコードのソース行または実行された Alpha 命令あるいはその両方と照合することができます。ソース・コードの照合を行うには、分析対象ルーチンの実際のソース・ファイルも必要です。DCPILIST 分析は、ProfileMe 方式および集約イベントによる収集データを対象に実行できます。
DCPICALC	イメージの指定された 1 つ以上のプロシージャについて制御フロー・グラフを生成します。DCPICALC は、基礎ブロックの推定実行頻度、1 命令あたりのサイクル数、その他の情報をグラフに追加します。DCPICALC は、集約イベントにより収集されたデータのみを対象とします。
DCPITOPSTALLS	1 つ以上のイメージにおいて、最も多くのストールの原因となった命令の特定に使用します。DCPITOPSTALLS は、集約イベントにより収集されたデータのみを対象とします。
DCPIWHATCG	DCPICALC と同様の制御フロー・グラフを生成しますが、プロシージャ・レベルではなく、個別のイメージを対象にします。DCPIWHATCG は、集約イベントにより収集されたデータのみを対象とします。
DCPITOPS	DCPICALC の出力を受け取って、イメージの実行についての PostScript™ 形式の出力を生成します。基礎ブロックの実行頻度を視覚的に表すために異なるフォント・サイズを使用します。DCPITOPS は、集約イベントにより収集されたデータのみを対象とします。
DCPICAT	未加工のプロファイル・データを読みやすい形式で提示します。通常は DCPI の開発者のみが使用しますが、便宜上 DCPI for OpenVMS キットに含まれています。DCPICAT は、すべてのタイプの DCPI イベントを処理します。
DCPIDIFF	プロファイルのセットを比較し、相違点を一覧にします。複数のテスト・ケースを比較するのに大変便利です。

表に示したように、ツールのうちいくつかは、集約イベントによってサンプリングされたデータのみを対象としています。このことは ProfileMeデータの解析を扱いにくいものにしている面もありますが、ProfileMeデータはこれらのツールなしでも全ての原因を特定するのに十分な内容であるといえます。DCPICALC、DCPIWHATCG、DCPITOPSTALLSはいずれも Alpha CPUの実行特性を熟知しており、ストールがどこで起こったかといったことに関して確度の高い推測をすることができます。ProfileMEの場合、プロファイル対象の命令ごとにすべてのデータが収集されます。したがって、ProfileMeでこれらのツールがサポートされないのは、データがハードウェア自身によって収集されており、より信頼のおけるものであるからです。

また、DCPI ツールはすべて UNIX スタイルのコマンド構文を使用する点に注意してください。DCL スタイルの構文を追加しても DCPI の分析機能を強化することにはならないため、ツールに DCL インタフェースを追加するという時間のかかる開発作業を行うことは見合わせました。

動的に生成されるコードのプロファイリング

Java™ のようなアプリケーションは、コードを動的に生成した上で実行されます。標準の DCPI は、イメージから命令を読み取って分析を行うため、分析には永続的なオンディスク・イメージが必要です。また、データ収集時にプロファイルを作成する際、DCPI はイメージ・オフセットなどを算出するためのロード済みイメージ・マップの作成も必要とします。しかし、動的に生成されるコードの場合、どちらも存在しません。

しかし、DCPI for OpenVMS には、生成されたコードについて DCPI デーモンに情報を提供するための API が含まれています。この API は、生成されたコードについてデバッグ・シンボル・テーブル（DST）を生成する手段も提供します。生成されたコードと、それに対応する DST はオンディスクの永続的な「擬似イメージ」に書き込まれ、それが分析時に使用されます。これは、Tru64 UNIX 用の DCPI と比べて DCPI for OpenVMS が進化した点です。

DCPI の使用方法

DCPI データ収集を実行するコマンドの一般的な実行順序は次のとおりです。

まず *dcpid*、続いていくつかの *dcpictl* コマンド、そして最後に *dcpictl quit* でデータ収集を停止します。

DCPI デーモンを DCPI ドライバと組み合わせて使用することで、まずデータをオンディスク・プロファイルに収集します。これらは、ディスク上のエポックに格納されます。エポックは、DCPI データに時間軸を適用する唯一の手段です。これは非常に重要な点です。なぜなら、エポック内のどのデータがたとえばピーク負荷時に収集されたかを知るのは不可能だからです。DCPI を使用したときによい結果を得るための一般的にお勧めの方法は、エポック内の負荷を安定した状態に維持することです。これが、何がプロファイルされたかを知るための唯一の方法だからです。ランプアップやランプダウン、ピーク、システムの活動が低い状態を含む実行は別々のエポックに分けて、どのプロファイルがどのテスト・ケースに基づくものか正確にわかるようにしておきます。プロファイルの名前は、作成時の GMT 時間に由来します。

データ収集時にエポックを操作するコマンドは次のとおりです。

- ***dcpid*** は省略時では現在の DCPI データベースに新しいエポックを作成します。*dcpid* 起動時にコマンド行で –epoch スイッチを指定した場合、新しいエポックは作成されず、DCPI データベースの最新のエポックが使用されます。

- ***dcpictl*** は、データ収集時の DCPI デーモンに対するインタフェースです。エポックを次のように操作できます。

    - *dcpictl flush*。ユーザによるフラッシュ操作により、DCPI デーモンおよび DCPI ドライバのインメモリ・プロファイル・データを DCPI データベース内の現在のエポックに掃き出します（論理名 DCPIDB が DCPI データベースを指しています）。掃き出しは、データ収集の間にも現在のエポックに対して自動的に行われます。
    - *dcpictl epoch*。DCPI デーモンと DCPI ドライバのインメモリ・プロファイル・データを現在のエポックに掃き出し、新しいエポックを開始します。

データ収集の実行

データ収集の一般的な開始方法は次のとおりです。

```
$ dcpid cmoveq$dka100:[dcpi.test]
dcpid: monitoring cycles
dcpid: monitoring imiss
dcpid: logging to comveq$dka100:[dcpi.test]dcpid-COMVEQ.log
```

DCPI デーモンは OpenVMS の対話型プロセスとして実行されるため、*dcpid* を実行しているターミナルをロックしないように、次のコマンドを使用するとよいでしょう。

```
$ spawn/nowait/input=nl: dcpid cmoveq$dka100:[dcpi.test]
%DCL-S-SPAWNED, process SYSTEM_187 spawned
dcpid: monitoring cycles
dcpid: monitoring imiss
dcpid: logging to comveq$dka100:[dcpi.test]dcpid-COMVEQ.log
```

EV67 以前のプロセッサでは、収集する省略時のイベントは *cycles* と *imis* です。EV67 以降のプロセッサでは、省略時のイベントは *pm*（ProfileMe）と *cycles* です。

データ収集を終了するには、次のコマンドを入力します。

```
$ dcpictl quit
```

データの分析

データの収集が完了したら（または、データの掃き出しを行った場合にはデータの収集中に）、*dcpiprof* を使用して、収集されたプロファイル・データの最初の確認をします。

```
$ dcpiprof
dcpiprof: no images specified.Printing totals for all images.
Column             Total  Period (for events)
------             -----  ------
cycles           1755906   65536
imiss              41991    4096

The numbers given below are the number of samples for each
listed event type or, for the ratio of two event types,
the ratio of the number of samples for the two event types.
=======================================================
 cycles       %    cum% imiss       % image
1349002  76.83%  76.83%  8154  19.42% DISK$CMOVEQ_SYS:[VMS$COMMON.SYSLIB]DECC$SHR.EXE
 176821  10.07%  86.90%   919   2.19% DISK$CMOVEQ_SYS:[VMS$COMMON.SYSLIB]LIBRTL.EXE
  65432   3.73%  90.62%   426   1.01% DISK$ALPHADEBUG1:[DEBUG.EVMSDEV.TST.TST]LOOPER.EXE;1
  45788   2.61%  93.23%  8651  20.60% SYS$SYSROOT:[SYS$LDR]SYSTEM_SYNCHRONIZATI
  27039   1.54%  94.77%  4598  10.95% SYS$SYSROOT:[SYS$LDR]SYSTEM_PRIMITIVES.EX
  16045   0.91%  95.68%   844   2.01% DISK$CMOVEQ_SYS:[VMS$COMMON.SYSEXE]DCPI$DAEMON.EXE;2
   7727   0.44%  96.12%  1969   4.69% SYS$SYSROOT:[SYS$LDR]RMS.EXE;
   6993   0.40%  96.52%  2102   5.01% SYS$SYSROOT:[SYS$LDR]SYS$PEDRIVER.EXE;
   6741   0.38%  96.91%  1762   4.20% SYS$SYSROOT:[SYS$LDR]PROCESS_MANAGEMENT_M
   6587   0.38%  97.28%  1215   2.89% SYS$SYSROOT:[SYS$LDR]F11BXQP.EXE;
   5742   0.33%  97.61%  1079   2.57% SYS$SYSROOT:[SYS$LDR]SYS$BASE_IMAGE.EXE;
   5385   0.31%  97.92%  1434   3.42% SYS$SYSROOT:[SYS$LDR]SYS$EWDRIVER.EXE;
   5344   0.30%  98.22%  1371   3.26% SYS$SYSROOT:[SYS$LDR]IO_ROUTINES_MON.EXE;
   5015   0.29%  98.51%  1024   2.44% unknown$MYNODE
```

最初の例は、最上位レベルの *dcpiprof* の実行の出力を示します。次のステップは、注目に値するイメージを特定し、*dcpiprof* を使用してそのイメージをさらに詳しく見ることです。

```
$ dcpiprof DISK$ALPHADEBUG1:[DEBUG.EVMSDEV.TST.TST]LOOPER.EXE;1
Column             Total  Period (for events)
------             -----  ------
cycles             84210   65536
imiss                540    4096

The numbers shown below are the number of samples for each
listed event type or, for the ratio of two event types, the
ratio of the number of samples for the two event types.
=======================================================
cycles       %    cum% imiss       % procedure                image
 65774  78.11%  78.11%   334  61.85% get_next_random          disk$alphadebug1..
```

```
 16892  20.06%  98.17%    93  17.22% analyze_samples           disk$alphadebug1..
  1543   1.83% 100.00%   112  20.74% collect_samples           disk$alphadebug1..
     1   0.00% 100.00%     1   0.19% main                      disk$alphadebug1.
```

通常、次のレベルの分析を行うには、*dcpilist* を使用してサンプリングに対応する実際のコードまたは Alpha 命令を確認します。

```
$ dcpilist –both -f dbg$tstevmsdev:[tst]looper.c get_next_random -
disk$alphadebug1:[debug.evmsdev.tst.tst]looper.exe
cycles imiss
     0     0           static int get_next_random (void)
     0     0           /*
     0     0           ** We want to get the next random number sample.
     0     0           ** The samples are SAMPLE_ITERATIONS calls apart.
     0     0           */
     0     0           {
     0     0            long int   i;
     0     0            int        sample;
    21     0
    21     0  0x2045c         STL             R31,#X000C(FP)
 42355   174            i = 0;
  3875    27  0x20460         LDL             R1,#X000C(FP)
 11536    31  0x20464         LDA             R1,#XFF9C(R1)
  3923    19  0x20468         LDL             R0,#X000C(FP)
 12001    50  0x2046c         ADDL            R0,#X01,R0
  3764    13  0x20470         STL             R0,#X000C(FP)
  3772    21  0x20474         BGE             R1,#X000006
                        . . . .
  3484    13  0x2048c         BR              R31,#XFFFFF4
 22013                  while (i++ < SAMPLE_ITERATIONS)
  4248    19  0x20478         BIS             R31,R31,R25
    37     0  0x2047c         LDQ             R26,#X0028(R2)
  3689    17  0x20480         LDQ             R27,#X0030(R2)
  7325    28  0x20484         JSR             R26,(R26)
  6714    38  0x20488         STL             R0,#X0008(FP)
     0     0              sample = rand ();
   195     2  0x20490         LDL             R0,#X0008(FP)
     0     0  0x20494         BIS             R31,FP,SP
    40     0  0x20498         LDQ             R26,#X0010(FP)
    44     1  0x2049c         LDQ             R2,#X0018(FP)
    85    37  0x204a0         LDQ             FP,#X0020(FP)
     0     0  0x204a4         LDA             SP,#X0030(SP)
     0     0           return (sample);
$
```

この詳細な情報の実際の難関は、なぜシステムがそのように動作しているのか理解すること、あるいは特定のルーチンが情報に示されているような頻度で使用されている理由を理解することです。高い頻度で使用する必要のあるルーチンにパフォーマンスの問題があるようであれば、ルーチンを詳しく見る必要があります。

先の例では、最上位のイメージは LOOPER.EXE ではありません。監視システムを見ると LOOPER.EXE を実行しているプロセスが最も CPU を消費しているので、明らかにこれが最上位であると考えがちです。しかし、実際の最上位のイメージはむしろ DECC 実行時ライブラリです。*get_next_random()* ルーチン呼び出し内の *rand()*呼び出しが DECC 実行時ライブラリを呼び出しており、そこからおそらく LIBRTL 内の 1 つ以上のルーチンを呼び出しているために、この 2 つが最上位イメージとなっています。この例は、比較的単純ながら、問題の根本原因に迫るのが難しい理由の 1 つを示しています。ここでは「問題」の原因は LOOPER.EXE であるように見えます。なぜなら、DECC 実行時ライブラリに対して多数の呼び出しを行っているからです。最初の *dcpiprof* の結果、DECC 実行時ライブラリに問題があると信じるのはたやすいことです。この例は単純ですが、システムの動作の根本原因を知るためには、さらなる分析がしばしば必要になることを示しています。

DCPI 分析のための基本的ないくつかのヒント

次に、DCPI 分析のためのヒントを示します。アプリケーションの綿密な分析を実施するには、アプリケーションそのものについてよく分かっている必要があります。

- 先に述べたように、DCPI における時間軸はエポックと呼ばれます。新しいエポックを作成する以外、個々のサンプルに「タイムスタンプ」を対応付ける方法はありません。DCPI の使用時に予測可能な結果を得るためには、エポックの全体にわたって安定した負荷を維持することを目標とします。後からエポックより細かい精度で分析する手段はありません。エポックの作成はデータ収集時に行います。

- DCPI は、「注目」しているプログラムだけではなく、OpenVMS システム全体のサンプリングを行います。共有イメージに対する呼び出しは、システムで現在実行されているすべてのプログラムによるすべての呼び出しの総計です。これを、個別のプロセス内で実行されるイメージの単位に分割するには、データ収集を *$dcpid –bypid 'imagename'* で開始します。

- 結論は慎重に出すようにしてください。先の例で示したように、結論を急ぐと間違った結論を出してしまいます。また、イメージまたはルーチンに関するサンプルが多くても、必ずしもそのイメージまたはルーチンにパフォーマンスの問題があるということにはなりません。サンプルが多いということは、単にそれらのイメージまたはルーチンの使用頻度が高いというだけのことです。サンプル数の根本原因を知るにはさらに分析を行う必要があります。

- DCPI データ収集の中核をなす DCPI デーモンは、通常のユーザ・プロセスです。負荷の高いシステムでは DCPI デーモンが CPU 時間を獲得できない場合があります。その場合、DCPI データベースの DCPI デーモン・ログ・ファイル（dcpid-*nodename*.log）には、"dropped samples"として記録されることがあります。DCPI データベースは、論理名 *dcpidb* によって定義されます。場合によっては、DCPI デーモン・プロセスの優先順位を上げるために、*dcpid –nice 'priority'* を使用してデータ収集を始めるのが適切なことがあります。

- DCPI デーモンは初期化時に、利用可能なすべてのプロセスを走査し、それぞれのプロセス内のイメージに関するイメージ・マップを作成します。多数のプロセスが存在する負荷の高いシステムでは、この初期走査に非常に時間を要することがあります。OpenVMS V7.3 以降が稼動しているシステムでは、システムの負荷が比較的低いときに事前に DCPI デーモンを開始しておくと、DCPI デーモンの初期化に要する時間を大幅に短縮できます。

- システムの活動もしくは活動のない状態はすべて収集データに反映されます。システムがアイドル状態の場合、総時間のほぼ 100% が PROCESS_MANAGEMENT.EXE の SCH$IDLE() に占められます。この exec ロード・イメージには他の重要なコードが含まれています。したがって、サンプル数が多い場合には別の何かを示している可能性があります。サンプル数のパーセンテージがアイドル時間のパーセンテージと同等であれば、このように推定してもほとんど間違いありません。

- データ収集時に余計な情報が混入するのを最小限に抑えるために、未使用の CPU または不要なソフトウェア、あるいはその両方を停止してください。ただし、これは問題の原因の絞込みを行うための、二次レベルのデータ収集を実施するときにします。

- 分析の対象にしているイメージまたはルーチンで、サイクル数が多いのが通常の状態である場合もあります。ProfileMe を使用して問題のありそうなイメージまたはルーチンを探す場合、ルーチンの RETIRED/CYCLES の比率を確認します。理想的には、Alpha チップは 1 サイクルで 4 命令処理します。比率の値が 3 であるルーチンを向上させるのはおそらく不可能でしょう。しかし、比率が 1 未満であるルーチンは、パフォーマンスの問題が生じているルーチンである可能性が高いでしょう。