WIND RIVER　ホワイトペーパー

マルチコアの課題と選択：アプリケーションに適したソリューションの決定

チーフ・テクノロジー・オフィサー　トーマス エヴァンソン

目次

要約 …… 1
ビジネスと技術のトレンド …… 1
　周波数の制限 …… 1
　マルチコアを使用したハードウェア・オフロード …… 2
　マルチプロセッサ・システムのマルチコアへの統合 …… 3
　組込みハイパーバイザ …… 3
　ユースケース …… 4
マルチコアおよびマルチOS開発者が直面する問題 …… 6
　ランタイムサポートとOSの選択 …… 6
　ベアメタル／小さい実行型オブジェクト …… 6
　コア間またはパーティション間通信 …… 6
　開発ツールのサポート …… 6
結論 …… 8

要約

将来、さまざまなデバイスの構造や開発を変えると予測される技術やビジネスのトレンドは数多くあります。周波数の制限、マルチコアを使用したハードウェア・オフロード、マルチプロセッサ・システムのマルチコアへの統合、組込み市場におけるハイパーバイザの採用、性能目標の達成を支援するユースケース、特定機器への機能集中、高度なセキュリティと安全性への懸念による認証済みシステムへの流れが、これらに含まれます。

マルチコアと仮想化は、こうしたトレンドを実現する有望な技術です。本資料では、これらのトレンドについてより詳しく解説します。

ビジネスと技術のトレンド

周波数の制限

ほとんどの場合、チップベンダのプロセッサ・ロードマップの右上には多くのマルチコアプロセッサが示されています。マルチコアとは、1つのチップ上に複数のプロセッサコア、つまりCPUが配置されていることを意味します。なぜ、あらゆるベンダが突然マルチコア化を採用するようになったのでしょうか。

これは結局のところ物理的な問題です。チップベンダは、これまで長期にわたり、常にクロック周波数を増加させることによってプロセッサのさらなる高速化を可能にしてきました。これは電力消費を著しく増加させますが、一方でクロック周波数増加のトレンドと同時にサイズの一層の小型化というトレンドを生み出し、これが電力消費の抑制につながっています。しかし、ここ数年、これらのトレンドは鈍化の傾向にあります。同時に、より高い性能への需要がなくなったわけではないので、プロセッサアーキテクトは、より多くのシリコンをこの問題に投入することにより、スーパー・パイプライニング（あらゆる命令が、より高速で実行可能な段階をより多く通過）、スーパー・スケーリング（各サイクルで複数の命令を発行）、分岐予測（ジャンプ先を予測することによって長いパイプラインの影響を最小限に抑制）、レジスタリネーミング（中間のパイプラインステップで仮想レジスタを使用し、より多くの並列実行を行う）といった種々の巧妙な技法を駆使して、より高い性能の実現を図るようになりました。

現在、周波数の増加とシーケンシャル・コード・ストリームを並列化する試みに要するコストの回収が難しくなってきています。また、実現できる性能向上の幅が小さく段階的なものであるにも関わらず、電力消費量は劇的に増加しています。

実際のところ、このような事態を招く要素は3つあります。1つは、電力消費が周波数に正比例するということです。2つ目は、周波数を増大させるにはより高い電圧が必要であり、電力消費量は電圧の二乗に比例するということ。そして3つ目の理由は、プロセッサのクロック周波数を上げるほどCPU速度とバス速度の差が大きくなることです。これは、前述の技法や大きなキャッシュには、多数のトランジスタを使用する必要があることを意味します。追加されたこれらのトランジスタは、すべて電力消費の増大につながります。

図1は、周波数の増加によって電力消費がどのように変化するかを示したものです。最初の組み合わせは、プロセッサを最大周波数で動作させた場合の性能と電力消費です。2つ目の組み合わせは、クロック周波数を最大値の80%まで下げれば、ほぼそのままの性能を維持しながら、電力消費をほぼ半分に減らせることを示しています。最後の組み合わせは、やはり80%で実行するコアをもう1つ追加した場合にどうなるかを示しています。この場合の電力消費は最大周波数で動作するコアとほぼ同じですが、性能は60%向上しています。これは理論上の話ですが、少なくとも以上のような効果が得られるはずです。

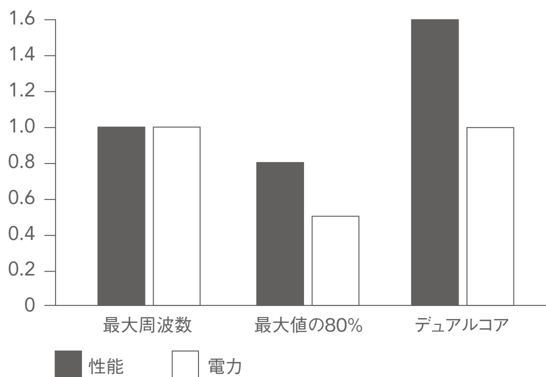


図1:周波数の変化に伴う性能と電力消費

これは素晴らしいことのように思われます。ではなぜ、これまでこのような試みがなされなかったのでしょうか。

実は、このような試みはなされていたのです。実際、マルチプロセッシング使用の起源は数十年前にさかのぼります。ただ、当時の主な目的はより高い性能を得るということであり、そのために複数のチップや、場合によっては複数のボードが使われていた、という点が現在と異なっています。多くのコアを単一チップ上に置くことが必要とされ、また、そのための技術が実現されたのはごく最近のことです。

ハードウェア的な観点からすると、これは素晴らしいソリューションです。MIPSが増加し、消費電力は低減されます。問題は、この新たな性能を活用するのが難しいということです。過去に使われた技術では世代ごとにプログラムを変更する必要はなく、実行速度がどんどん速くなっていきました。高性能を得るためにプロセッサコアを追加する場合は、これらのコアを利用するために複数のスレッドを並列で実行しなければなりません。ほとんどではないにしろ多くの組込みアプリケーションでは、プロセッサはその時間の大部分を1つのスレッドの実行に費やします。プロセッサをもう1個追加した場合でも、既存のスレッドが1個目のCPUを占有している間、2個目のプロセッサはアイドル状態のまま放置されることになります。

マルチコアの性能を最大限に引き出すという課題は、アプリケーションに並列タスクを同時に実行させるということです。並列性を見出すという基本的問題を解決するのは難しいことですが、よく理解されている特別なケースもいくつか存在します。

マルチコアを使用したハードウェア・オフロード

特に成功しているマルチコア利用例の1つが、これまで専用プロセッサ、FPGA(FieldProgrammableGateArray)、ASIC(Application Specific Integrated Circuit) といった専用ハードウェアによって実行されていた特定のタスクを実行することです。これらのタスクはすでに個別のユニットにより実行されていたものなので、並列性に関する問題は既に解決されています。極めて特殊なタスクに対し、比較的低速な複数のコアを低レベルのハードウェアとともに使用することによって、非常に良好な並列性能を実現することができます。しかも、旧来の方法に比べると設計やプログラムも比較的容易です。一例としてルータを考えてみましょう。ルータの主なタスクは、受信されたパケットを確認して、それをどのポートに転送するかを決定することです。バッファ管理や暗号処理といった比較的小さなタスク専用のハードウェアを使用することにより、複数のコアを並列で実行させて非常に高いスループットを実現することができます。

図2は、専用ハードウェアと複数コアを高効率のルータの基礎とする方法を示したものです。

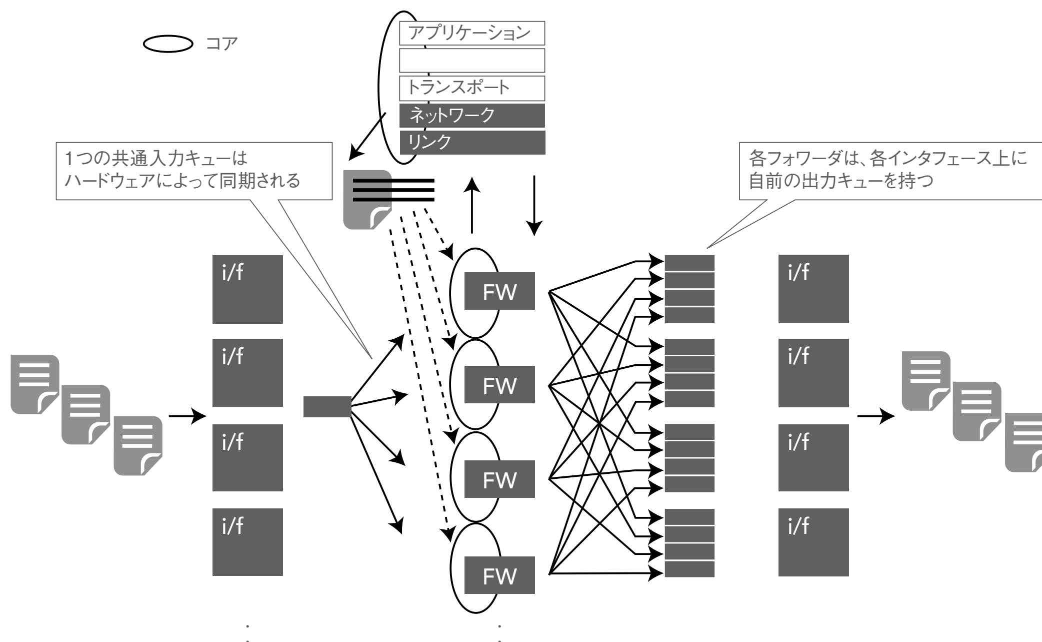


図2:ルータ・アプリケーションでのマルチコアを使用したハードウェア・オフロード

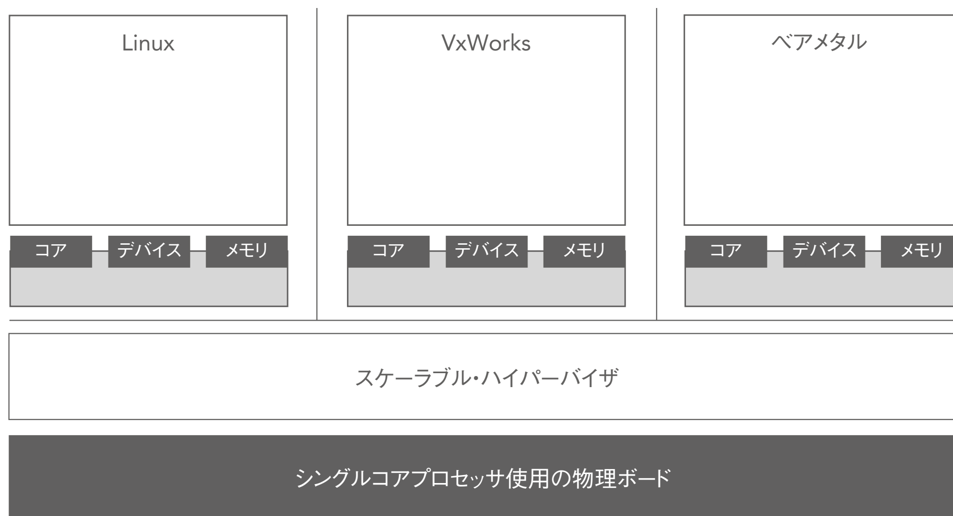


図3：ハイパーバイザを使用して1つのコアを複数の仮想ボードで共有

マルチプロセッサ・システムのマルチコアへの統合

すでに数多くのデバイスが、高性能の実現や異なる要件を持つシステムの統合などのためにマルチプロセシングを使用しています。マルチコアプロセッサが一般的になった現在、さまざまなデバイスが安価で電力消費の少ないマルチコアを使い始めた理由は容易に理解できます。同時に、マルチコアは複数のプロセッサユニットを単に1つのダイ上にまとめたものではない点を理解することが重要です。実際、マルチプロセシングシステムからの移行を困難にしている多くの重要な違いがあります。

マルチプロセシングシステムとマルチコアの主な違いの1つは、マルチプロセシングシステムではCPUの独立性がはるかに明確だということです。通常、各CPU間にははっきりとしたバスが存在しており、外付けメモリが共有されている場合でもCPUは独立して動作します。マルチコアではこのようなことはなく、共有のレベルがはるかに高くなっています。アーキテクチャに応じて、通常、マルチコアシステム内のコアは、割り込みコントローラ、デバイス、キャッシュなど、多くのものを共有しています。共有しているOSが1つ（対称型マルチプロセシング）の時は、通常これは利点となりますが、リアルタイム・OS（RTOS、ウインドリバーのVxWorksなど）と汎用OS（Linuxなど）というように、2つの異なるOSを実行する時はネックとなります。1つのマルチコア上で複数のOSを実行する時は、たとえばマスタ／スレーブの関係において両方のOSをうまく連携させる必要があります。もしくは、OSを管理するコードの一種であるスーパーバイザあるいはハイパーバイザを使用すれば、状況を簡素化することができます。

組込みハイパーバイザ

ハイパーバイザ（スーパーバイザとも呼ばれます）は、それに続いて動作する1つまたは複数のOSを管理する一連のコードです。ハイパーバイザに続いて動作するパーティション（あるいは仮想ボード）を作成することによって、ハイパーバイザは管理を行います。この作成は通常、システムの特定部分を仮想化することによって行われます。一般的に、仮想化できる要素は3つあります。

1. **実行ユニット（CPU）**：CPUを仮想化することによって、1つの物理CPUまたはコアに続いて複数の仮想CPUを実行することができます。これは、実際のCPUをタイムスライシング（つまり優先度に基づくアルゴリズムを使用）し、各仮想CPUに実際のプロセッササイクルの一部を使わせることによって行われます。
2. **メモリ**：メモリを仮想化することによって、複数のパーティションが実際のメモリの一部を使用できるように物理メモリを分割することが可能です。これらのパーティション内で実行されるOSは、仮想メモリを利用してプロセスを完了させることができます。このようなケースにおけるメモリ階層は実際には3つあります。1つがハイパーバイザ用、1つが仮想ボードで動作するOS用、そしてもう1つがプロセス内で動作するアプリケーション用です。
3. **デバイス**：複数の仮想ボードが、シリアルポート、Ethernetポート、グラフィック、その他のデバイスを共有する必要がある場合は、それらのデバイスも仮想化する必要があります。通常、パーティション内で動作するOSが物理的にデバイスにアクセスする代わりにAPI呼び出しを行えるようにするためにパーティション内に明確なインタフェースを置くことによって、仮想化は実現されます。デバイスを扱う実際のコードは2つの場所、ハイパーバイザ内か別のゲストOS内に置くことができます。

マルチコアとともにハイパーバイザを使用する最も一般的なケースは、1つのコア上に複数のパーティションを置かずに、1つまたは複数のコアのすべてを1つの仮想ボードに割り当てることです。このタイプのハイパーバイザはスーパーバイザと呼ばれます。スーパーバイザを使用する場合はパーティション間でコアをタイムスライスする必要はなく、対応するOSをはるかに効率的に実行することができます。OS間での完全な保護を実現してOSがデバイスを共有できるようにするためには、他の2つの側面であるメモリとデバイスも仮想化することがさらに必要です。

ハイパーバイザには2つのタイプがあります。タイプ1は小型の専用ハイパーバイザで、これはハードウェア上で直接動作します。タイプ2は、通常、フル装備のOS上で、あるいはそれと連携して動作し、そのホストOSのリソースを使用します。

組込みハイパーバイザに最適なのはタイプ1ですが、ほとんどのサーバのハイパーバイザはタイプ2です。組込みハイパーバイザ（たとえばWind River Hypervisor）とサーバハイパーバイザ（たとえばVMwareやKernel-Based Virtual Machine）の主な違いは、その要件が大きく異なることに起因しています。組込みハイパーバイザでは、重要な2つの要件は性能と分離であり、一方サーバハイパーバイザでは、後方互換性（ゲストOSを修正せずに実行できること）がより重視されます。

これらの要件によって、以下のような組込みハイパーバイザのデザインが決定されます。

- 最大限の性能と分離を実現するために、可能であればデバイスはゲストOS内に直接マップされる。
- ハイパーバイザはスケーラブルである。つまり、性能と分離の二者

択一を可能にするために、さまざまなエンティティの仮想化を選択できる。

- ハイパーバイザは、実行速度を上げるため、通常は非常に小さい。
- ゲストOSは、通常、準仮想化（paravirtualize）される。つまり、ハイパーバイザに続いて効率よく動作できるように、ゲストOSには変更が加えられる。

ハイパーバイザの実装方法は、ハードウェアによるサポートの度合いに応じて大きく異なります。最近のプロセッサの中には、ハードウェアで3レベルのメモリシステムを実装しているものがあります。これを利用すれば、ハイパーバイザの性能への影響を最小限に抑えられる可能性があります。しかし、ハイパーバイザは、適切なレベルの準仮想化によって、特別なハードウェアのサポートなしでもプロセッサ上で大幅に高速化することが可能です。

利用例

マルチコアとOSはさまざまなコンフィギュレーションで組み合わせることができますが、より一般的な利用例（ユースケース）がいくつかあります。

マルチコアを利用する際にOSがコアを管理する方法は、基本的に2つあります。対称型マルチプロセシング（SMP）を使用する時は、1つのOSが複数のコアを管理します。コアが利用可能になると、そのコアが、待機中のキュー上にある次のスレッドを実行します。これに対して非対称型マルチプロセシング（AMP）では、1つのコアごとに1つのOSを使用します。これには、同じOSの2つのコピーか、Linuxのような汎用OSとRTOSといった2つのまったく異なるOSを使うことができます。

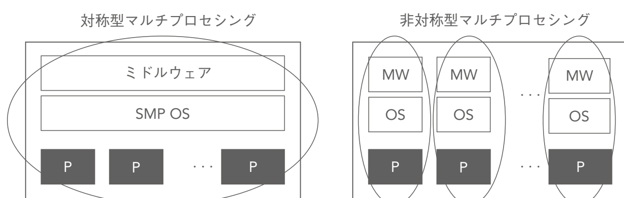


図4：SMPとAMPの比較

いずれのアプローチにも長所と短所があり、最良の選択はアプリケーションによって異なります。SMPには、存在するプロセッサよりも通常動作可能なスレッドが多い限り、コアにまたがるスレッドのロードバランスを取ることができるという利点があります。一方、コア間における同期の必要性が低いので、AMPはより効率的です。

SMPシステムでの結果の予測性と性能をより高めるために、通常、アフィニティを使用して特定のスレッドを特定のコアに結びつけることが可能です。これは1つのコアから別のコアへの移行によるキャッシュへの影響を最小限に止めますが、一方で、そのスレッドのロードバランスを取ることができなくなります。

通常、AMPはスーパーバイザ／ハイパーバイザとともに使われます。スーパーバイザ／ハイパーバイザは、共有リソースを処理するとともにOSを互いに保護し、一方の障害が他方に及ぶのを防ぎます。

ユースケース1:SMP

SMPは多くの異なるユースケースに使用することができます。この方法はネットワークスイッチを使用するので、同じデバイスのAMP実装と比較対照することができます。図5に示すデバイスの目標は、マルチコアによって実現される並列性を利用してできるだけ多くのパケットをギガビットスイッチにより転送することです。OSには、VxWorksのようなRTOSやLinuxのような汎用OSを使用することができます。アフィニティ（実行スレッドを特定のコアにロック）を使用することにより、データプレーン上の転送エンジンはキャッシュをまったく使わずに動作することができ、ハードウェア内の特定のオフロード機能を利用してバッファ管理や暗号化などの処理を行うことができます。

このシステムのツールに関する側面については、本資料の「開発ツールのサポート」の項に述べます。

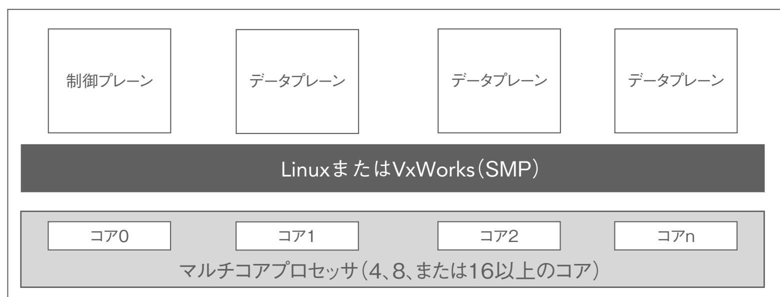
次の項では、SMPではなくAMPを使って実装した同じデバイスとトレードオフについて述べます。

ユースケース2:Supervised AMP

図6は、sAMP（supervised　AMP）を使用して実装した同じネットワークスイッチを示したものです。このシステムでは1つのコアを制御プレーン専用に使用し、そのコア上で1つの完全なOSを実行しています。そのシステムが、より強力な制御プレーンの処理能力を必要とするものであ

特性

- 性能重視
- 4以上のコアからなるマルチコア
- 分割されたシステム
  - 制御プレーン
  - データプレーン（アフィニティの使用）



ネットワーク機器

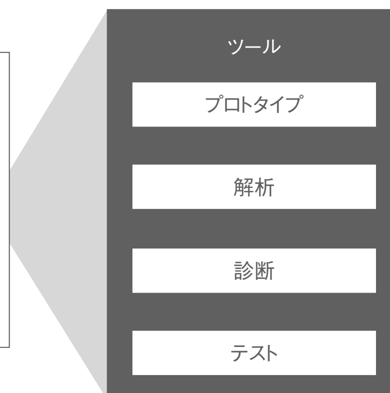


図5：性能のためのSMPソリューション

特性
- 性能重視
- スーパーバイザによるスケーラブルな保護
- 4個以上のコアからなるマルチコア
- 分割されたシステム
  – 制御プレーン
  – データプレーン(専用の実行型オブジェクトを使用)

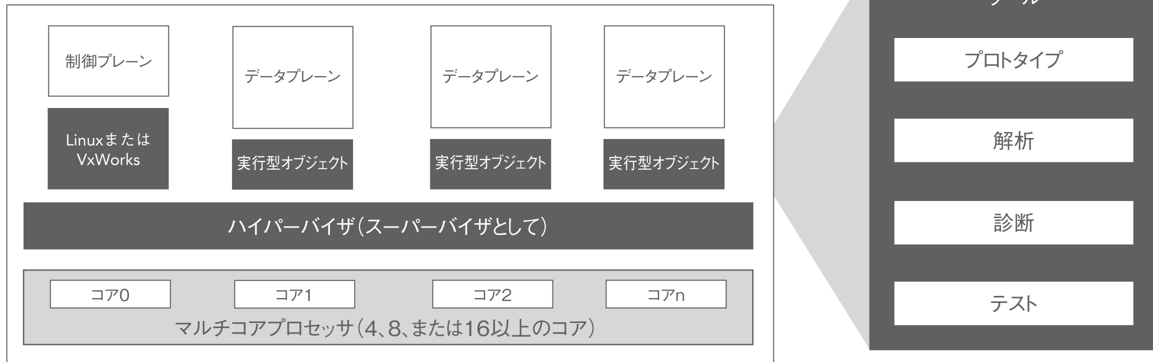


図6:性能と保護のためのsAMPソリューション

る場合は、1つのSMP OSが複数のコアを使用することもあります。他のコアはパケット転送などの専用タスクに使われます。これらのコアは、小さいOSや、転送アプリケーションを極めて効率的に行う実行型オブジェクトを実行します。実行型オブジェクトについては、後述の「開発ツールのサポート」で詳しく述べます。

割り込みコントローラなどの共有デバイスを管理するためにコアの最下層部分で実行されるスーパーバイザが存在します。このシナリオでは、このスーパーバイザは必要な場合(通常はセットアップ時)のみ作動します。それ以降、それぞれのコア上でOSと実行型オブジェクトは最大速度で実行されます。

SMPバージョンのスイッチにはいくつかの明確な利点があります。特に、OSが1つだけなので一般に管理と開発が容易です。これに対しAMPバージョンには、以下に挙げる点を含む別の利点がいくつかあります。

- **障害分離**:1つのコアがダウンした場合でも、残りのコアは動作を継続できます。
- **GPL分離**:メインのOSがLinuxの場合、AMPは、データプレーンコア上で動作する企業独自のソフトウェアを分離するための優れた方法を提供します。
- **スケーラビリティ**:通常、AMPはコア数の増加に合わせて拡張が可能です。

ユースケース3:ハイパーバイザ

特性の異なる複数のOSを混用する必要がある場合のユースケースは、以上とは多少異なったものとなります。提供される豊富なエコシステム(例えばグラフィックやネットワーク接続など)を利用するために、Linuxなどの汎用システムが必要となる場合があります。さらに、他のOSがLinuxでは提供できないリアルタイム特性を必要としたり、新しいOSへの移植が困難な既存コードが存在したりするといった状況も考えられます。もう1つの一般的な状況として、システムの一部を、Linuxにおいて可能なレベルよりも高いレベルで認証する必要があることも考えられます。OSを混在させる理由がどのようなものであっても、ハイパーバイザはパーティションの管理と分離にとって非常に効果的な技術です。これは、パーティションがシングルコア上で動作する場合(コアスケジューリングを使用)もマルチコアシステム上で動作する場合も同様です。

図7は、これと同じタイプのシステムを示したものです。この場合、両方のOSを互いに連携させる必要はなく、一方がクラッシュした場合でも、一度リブートすれば他方は実行を継続することができます。

特性
- 複数の異なるOS
  – リアルタイム(VxWorks)、汎用(Linux)、…
- ハイパーバイザによる完全な保護
- 1つまたは複数のコア
- Certification Possible

図7:保護を伴う統合のためのハイパーバイザ・ソリューション

マルチコアおよびマルチOS開発者が直面する問題

並列実行のためにコードを分割するという基本的な課題に加えて、マルチコアチップをプロジェクトに取り入れる際に直面する問題は他にも数多くあります。そのうちのいくつかを以下に挙げます。

- OSコンフィギュレーション、リソース共有、ブーティングのランタイムサポート
- コア間通信（IPC）
- コンフィギュレーションとプロトタイピング、解析、診断、テストのための開発ツールサポート

システムを適切なタイミングで使えるようにするには、これらすべての問題を解決する必要があります。この項では、問題のいくつかについてもう少し詳しく述べます。

ランタイムサポートとOSの選択

OSとそのコンフィギュレーションの選択は非常に重要です。多くの場合、OSはすでに選択されています。その理由としては、レガシーシステムによる制約や、非常に特殊な要件などが考えられます。RTOSのリアルタイム動作や性能が必要な場合は、他に選択の余地はありません。また、より一般的な汎用OSで使用できるソフトウェアが大量に必要な場合も、選択肢は自然と決まってきます。容易でないのは、マルチコアを使って1つまたは複数のOSをどのように実行するかということです。OSは1つとすべきか複数とすべきか、SMPとAMPのどちらを使用すべきか、あるいは両方を使用すべきか、また、非対称型システム上ではNUMA（Non-Uniform Memory Access）とハードウェアマルチスレッディングのどちらを使用するのが有利か、ハイパーバイザを使う利点はあるか、1つの特定タスクに特化されたコアにはどのOSを使うべきか、等々。これらの面を組み合わせる必要のあるシステムにおいて、マルチコアとハイパーバイザには、開発者にとってその組み合わせを選択できるという利点があります。

ベアメタル／小さい実行型オブジェクト

SMP、AMP、ハイパーバイザ、ハイパーバイザの長所と短所についてはすでに述べましたが、特定タスク専用のコアを使用することについて、もう少し詳しく見てみましょう。コア上で実行するタスクが例えばポーリング・ループのような単純なものだけである場合、開発者はOSの必要はないと考えるでしょう。これはその通りです。完全なOSは必要ありません。ハードウェアへのアクセスを容易にするAPIを持った小さい実行型オブジェクトでも十分です。多くの場合、これらはチップベンダから入手することができます。しかし、その実行型オブジェクトの上位にコードを置こうとすると、それ以上のサポートが必要であることがわかります。例えば、コード内にprintf()ステートメントを加えて、Linuxパーティションにマップされたシリアルポートにそのコードを転送することができれば便利ではないでしょうか。また、デバッギングについてはどうでしょう。

非常に簡単な実行型ファイルが提供可能なサポート以上のものが必要となるはずです。より良い解決策は、キャッシュに合った極めて小さいダイナミック・フットプリント（アクティブ命令コード）にまでスケールダウンできる一方で、ネットワーク接続、デバッグ機能、解析ツールなどを提供することのできるOSを使用することです。最も理想的なのは、半導体ベンダ作成の抽象化コードを使用して緊密に統合された小さいOSを入手することです。

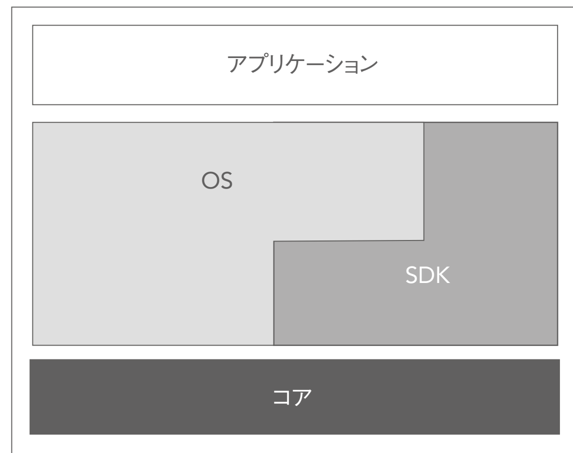


図8：ベアメタル実行型オブジェクトや小さい実行型オブジェクト用のアーキテクチャ

図8は、ハードウェアへの極めて特殊なアクセスと汎用機能の両方を実現するためにチップベンダのソフトウェア開発キット（SDK）と統合された、小さいOSを示したものです。

コア間またはパーティション間通信

コアおよびパーティションの間に高速の通信プリミティブを置くことは、アプリケーションによるデータ交換にとって重要なだけではなく、一連の機能を使用可能にするためにも重要です。これらの機能には、リモートI/O、printf、ファイルシステム、ネットワーキング、デバッギングなどが含まれます。I/Oリソース数よりもコア数の方が多い場合は、I/Oリソースを共有できるようにする必要があります。例としてデバッガを考えてみましょう。複数のパーティションを同時にデバッグしたい場合でも、通常はコアごとに個別のデバッグデバイスを使用することはできません。代わりにOSの1つにだけ接続できるようにして、他のパーティションにはプロキシを介して接続する必要があります。

ニーズによっては、異なるレベルでの通信が必要になることもあります。ロッキングプリミティブなどの少ない機能や共有メモリで十分な場合もありますが、それ以外の場合はより高いレベルのメッセージシステムが必要です。コードを移植しやすくするために、標準化されたプロトコルとAPIを探してください。多くのIPCメカニズムは標準ソケットインタフェースを使用しています。これは、緊密に組み合わさったシステムから、TCP/IPを使用するより大きな分散化システムへのプログラムの移植を容易にします。

開発ツールのサポート

マルチコア使用時には、開発を支援するためのツールが一層重要になります。マルチコアシステムは複雑なので、従来の開発方法をそのまま用いることは非常に困難です。また、編集／コンパイル／デバッグというサイクルだけでは不十分です。

コンフィギュレーション、プロトタイピング、診断、解析、テストに関わる新しい課題が、新たなレベルのツールサポートを必要とします。

プロタイピングとシミュレーション

実際のハードウェア以外のものを使用してデバイスをシミュレートすることが一般的になってきている理由はいろいろあります。1つの理由として、実際のハードウェアがまだ利用できないこと、あるいは、すべての開発者に実際のハードウェアを割り当てたのでは費用がかかりすぎてしまうことが考えられます。また、実際のハードウェアでは行うことができないような方法でシステムをデバッグする必要があるということも考えられます。しかし、多くの理由は、ほとんどのソフトウェア開発者にとってハードウェアに関する問題を処理する必要がない方が容易である、というもの

です。一度に1つのことに集中せよ。早期に継続的にテストを行なうことが重要になっているというトレンドによって、シミュレータがより大きな役割を負うようになっています。

シミュレータにはさまざまなタイプがあり、すべてのニーズを満たす万能のシミュレータというものはありません。例えば、OSの動作をシミュレートする高速機能シミュレータで十分な場合もあります。開発用ホストマシンの命令セットを使用することによって、開発中のアプリケーションは、これらのシミュレータ上で非常に高速で実行することができます。

これと対照的な例が命令セットシミュレータ（ISS）です。マルチコア用の開発を行う際に命令セットシミュレータを使用することには、多くの利点があります。ハードウェアが使用可能になる前でもデバッグを行うことができるという明らかな利点の他にも、マルチコアデバッギングに威力を発揮する2つのISSの特性があります。それは正確な実行、そしてシステムシミュレーションです。

優れたISSでシステムをシミュレートすると、ISSは毎回正確に同じようにコードを実行します。実際のハードウェアの場合はそういう訳にはいかず、アプリケーションの動作はわずかなタイミングの差によって変わってしまいます。予測可能な形でタイミングに関する問題が発生するという利点は、問題が毎回常に同じように発生するので、デバッグがはるかに容易であるということです。システムシミュレーションとは、システム全体をまとめてシミュレートし、すべてのプロセッサとデバイスを同時に停止して、何が起こっているのかを確認できることを意味します。これらの機能の組み合わせは、マルチコアのデバッギングにとって非常に有効です。

一部のシミュレータは、ソフトウェアトレースの提供や逆方向のステップ実行といった、さらに高度な機能を備えています。特に後者は、デバイスの相互動作を含むあらゆる状況下で逆方向ステップ実行を行うことができるのであれば、極めて有効です。

診断とデバッギング

マルチコアの取り扱いでもう1つの難しいタスクがデバッギングです。他と連携して動作するスレッドのデバッギングは、通常のシリアルコードのデバッギングよりもはるかに困難です。別のスレッドやプロセッサが、ある一定の時点で共有データを操作する点を考慮する必要があるからです。コードのすべての行を新たな観点で見る必要がにわかに生じてきます。

マルチスレッドのバグで最も一般的な2つが、デッドロックと競合状態です。デッドロックは2つ以上のスレッドがそれぞれセマフォを持ち、それらのスレッドが、他のスレッドにロックされたセマフォを待っている時に発生します。競合状態は、複数のスレッドがインターリーブされた形で同じグローバルデータにアクセスした時に発生し、予測できない結果をもたらします。多くの場合、デッドロックに関する問題を1つ解決すると、今度は競合状態が発生します。また、その逆の場合もあります。

競合状態に関する最も難しい部分は、この状態がタイミングに依存しているため、通常はごく稀な一定の状況下でしか発生しないということです。通常、タイミングは実行ごとに異なるので、この問題の症状は、プログラムを実行するたびにまったく違ったものに見えるのが普通です。図9に示すように書き込みがマスクされている場合は、問題が表面化するまでに数百万サイクルを経ることがあるため、データの破損を招いたり、最悪の場合はシステムがクラッシュしたりする恐れがあります。

このような状況では、ランタイム解析ツールが非常に有効な手段となります。これらのツールは、通常、イベント発生時に情報を記録するので、開発者は問題の表面化後にデータを解析することができます。これらのツールの多くは開発者が固有のユーザイベントを追加することが可能なので、他のイベントに関連してアプリケーションの特定部分がいつ実行されたのかを確認することができます。共有データを操作するコードにユーザイベントを挿入することにより、ミューテックスによる適切な保護なしにデータへのアクセスが行われるケースの有無を確認することができます。

タイミングに関する問題をチェックするためにコードを挿入するという作業は単調なものになりがちで、アプリケーションを再コンパイルして再起動し、挿入したコードを後で削除する必要があります。実行中のシステム内で、アプリケーションに動的にコードを挿入することができる優れたツールもあります。その考え方は、ブレークポイントにコードを添えて挿入することに似ています。

システムを停止することなく何が起こっているのかを確認できるこのスタイルのデバッギングは、多くのデバイスにおいて大きな威力を発揮します。しかし、本当に必要なのはハードウェア内で何が起こっているのかを知ることだ、という場合もあります。ここで述べる他のツールには、計測コードを挿入したり実際のハードウェアをシミュレートしたりすることによって、動作が少し変わってしまうという欠点があります。今日のほとんどのプロセッサは、オンチップデバッギング機能を備えています。通常、オンチップデバッギング機能はボード上のいくつかのピン（例えばJTAG）を通じて使用されます。これらのピンによって、オンチップデバッガがソフトウェアエージェントを使用せずに、プロセッサを直接制御できるようになります。マルチコアの状況で、このデバッガはスキャンチェーンと呼ばれるコマンドの単一チェーンを通じ、すべてのプロセッサとデバイスを制御することができます。

今日のオンチップデバッガは、同じスキャンチェーン上の複数のプロセッサを非常に小さなタイムクリープで同時に制御（開始、停止等）することができます。タイムクリープとは、1つのコアが応答を開始してからすべてのコアが応答を完了するまでの時間を言います。このことは、1つのコアを停止している間に、他のコアが実行を継続して状態を変えたりバッファがオーバーフローしたりしないようにするために重要です。また、1つの統合開発環境（IDE）からコアを制御し、コアを並列にチェックできることを確認してください。

すべてのコアを同時に停止できることにより、オンチップデバッガは、システム全体を停止する非常に強力なシステムレベルデバッガとなっています。

ランタイム解析ツールやオンチップデバッガに加えて、従来型のエージェ

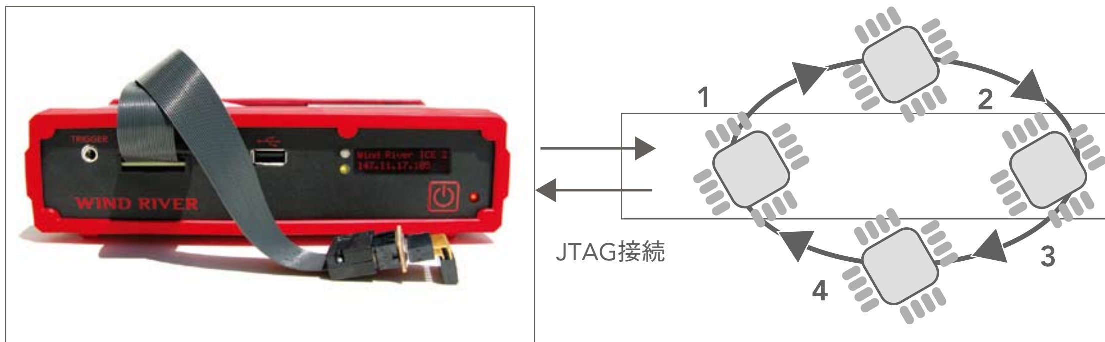


図9：JTAGによるオンチップデバッギング接続

ントベース・デバッガも使われます。特に、システムの他の部分を中断させることなく停止できるようなアプリケーションコードの場合、そのコード内の問題特定には従来型のデバッガが使われます。

解析

開発者がマルチコアの使用を開始する際の共通の問題は、予想通りの性能が得られないことです。すでにお分かりのように、マルチコアシステムの性能を予測して解析し、これを理解することは非常に難しいことです。同期に関する問題（例えばセマフォ待機）、キャッシュに関する問題、メモリやI/Oの制約を受けるアプリケーションなどを含め、アプリケーションがなぜ期待通りの性能を発揮できないのかについては多くの理由があります。実際のデータなしで何が起こっているのかを推測することは非常に困難ですが、ツールは、システムの速度を低下させるボトルネックを理解する助けとなります。

プロファイラは、コードのどの部分で時間がかかっているのかを理解する上で非常に有効なツールです。個々の関数でどれだけの時間が費やされているのかしか分からないフラットプロファイラではなく、呼び出しツリー全体でどの程度の時間が費やされているのかを知ることができる階層プロファイラを使用してください。

開発者はプロファイラの情報を活用することにより、並列化を進めながら、どこに時間をかけるかということを十分な情報に基づいて決定することができます。

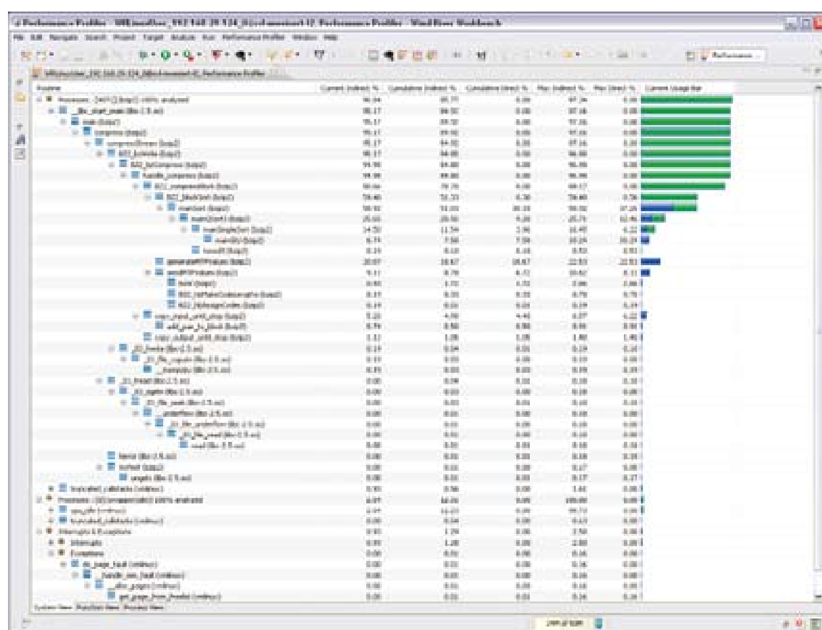


図10:Wind River Workbenchの階層プロファイリング機能

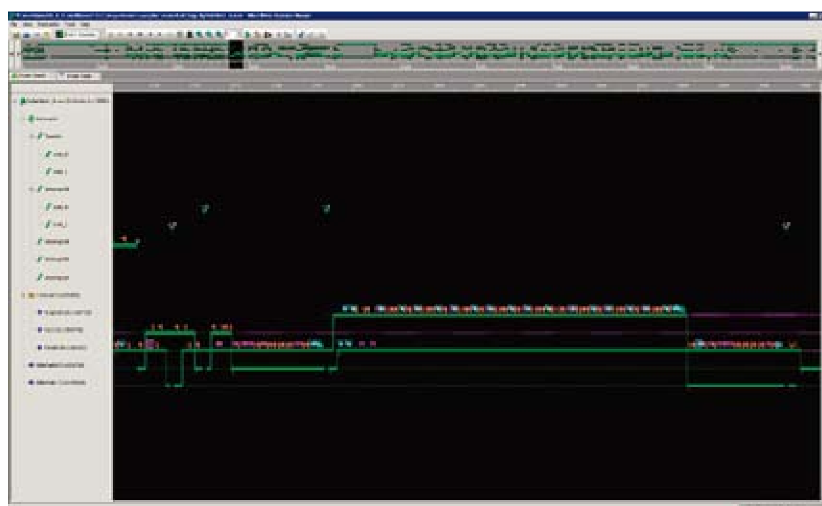


図11:Wind River Workbenchのシステムビューア機能

マルチコアアプリケーションの中で実際に何が起こっているのかを理解するには、時間に沿ってイベントのシーケンスを示すことのできるランタイム解析ツールも必要です。その出力は、図11に示すスクリーンショットのようになります。

図11は並列で実行されている2つのプロセッサを示したもので、ここでは1つのスレッドから別のスレッドへの移行が行われています。時系列に沿ったマーカーは、セマフォの取得や返却などのイベントに対応しています。開発者はこれらのイベントのいずれかを掘り下げて、その基礎となるデータを確認することができます。

結論

これからはマルチコアの時代です。これは物理的な問題であり、新しいマルチコアプロセッサを効果的に利用するには、必要なツールやランタイムを最初から作り上げるのではなく、商業ベースで入手できるものを使うほうが賢明です。マルチコア開発の方法は、これまでシングルコア開発に使われてきた方法とは大きく異なります。

すべてのマルチコア・ユースケースを解決できるような万能の解決策は存在しません。シングルコアプログラムをマルチコアプログラムに変換する魔法のような方法もありません。開発者に必要なのは、開発するアプリケーションの効率的な実行を助けるランタイム・コンフィギュレーションとツールを選択すること。開発者は、これらのツールとランタイムをうまく統合する必要があります。この統合はツールやランタイム同士だけではなく、アプリケーション用に選択した特定のシリコンに対しても必要となります。これが実現されて初めて、マルチコアが約束することをアプリケーションが完全に利用することができます。

ウインドリバーはスマートデバイス搭載ソフトウェアの最適化（DSO）をワールドワイドに提供するリーディングカンパニーです。
企業がスマートデバイスに搭載するソフトウェアを、品質および信頼性のさらなる向上を実現しつつ、
リーズナブルなコストで開発することを可能にし、早期にマーケットへ投入することを支援します。

WIND RIVER ウインドリバー株式会社

東京本社 〒150-0012 東京都渋谷区広尾1-1-39 恵比寿プライムスクェアタワー TEL.03-5778-6001（代表） FAX.03-5778-6002
大阪営業所 〒532-0011 大阪市淀川区西中島7-5-25 新大阪ドイビル TEL.06-6100-5760（代表） FAX.06-6100-5761
E-mail:info-jp@windriver.com http://www.windriver.co.jp

登録商標： Wind River、Wind Riverロゴ、Tornado、VxWorksは、Wind River Systems, Inc.の登録商標または商標です。記載されているすべての名称は、各社の登録商標、商標またはサービスマークです。

091119WRKK