Linux における CPU/Memory/IO の Hot Plug サポート

菅沼公夫* 河内隆仁* 青野寛**

* NEC Solutions (America), Inc. Santa Clara Technology Center

** 日本電気株式会社 第一コンピュータソフトウェア事業部

我々は Itanium プロセッサ搭載サーバを主なターゲットとし、CPU/Memory/IO の Hot Plug 機能の実装作業を行っている。これらの機能は、順次 2.5 カーネルに採用されており、2.6 カーネルでは本格的な Hot Plug 機能が利用可能となる予定である。本論文では、Hot Plug の目的と活用方法を紹介し、ACPI(Advanced Configuration and Power Interface)との関わりと CPU/Memory/IO Hot Plug 機能それぞれの実装方式を解説する。

1 はじめに

Linux は 2.4 カーネルにおいてマルチプロセッサシステムのスケーラビリティ改善などにより、大型サーバでの活用が現実的なものになりつつある。Linux は今後ますますエンタープライズ領域へ進出すると期待されているが、そこで問題となっているもののひとつが RAS (Reliability, Availability, Serviceability) である。企業の基幹システムに Linux を導入するためには、ハードウェア障害やカーネルの不具合によるシステムダウンを極限まで小さくする必要がある。しかしながら現在の Linux は、商用 UNIX に比べると各種の RAS 機能が十分とは言えない。

このような背景から、Linux の RAS 機能強化を推し進める動きが各所で出始めている。NEC (日本電気株式会社) では、エンタープライズ用途で利用できる Linux カーネルの実現を目指し、他社 Linux 技術者や Linux コミュニティの技術者と協力し、RAS 機能の強化を含め様々なプロジェクトに参画している。CPU/Memory/IO Hot Plug サポートプロジェクトは当社を含む複数の企業で立ち上げたオープンソース・プロジェクトである Atlas Project[1]の一環として作業を進めている。

2 Hot Plug の目的と活用方法

Hot Plug とはシステムを停止する事なくシステムの構成デバイスを追加・削除する機能であり「活線挿抜」とも呼ばれる。Hot Plug の中でも、デバイスの動的組み込みを Hot Add、動的切り離しを Hot Remove と呼ぶ。USB や PC カードなどのデバイスにおいては Hot Plug はすでに一般的となった機能だが、CPU や Memory、PCI カードや PCI ホストバスブリッジなどのシステムの中核に位置するデバイスに対しても同様に Hot Plug をサポートするのが本プロジェクトの目的である。また、CPU/Memory/IO Hot Plug 機能は、RAS 機能としての役割に加え、システム内での論理的なパーティション分割といった、システムの柔軟な運用につながる技術でもある。

2.1 対象プラットフォーム

NEC は Itanium2 プロセッサを搭載し、CPU/Memory/IO の Hot Plug が可能な TX7/i9000 シリーズを提供している。本プロジェクトでは各種プラットフォームで利用可能な Hot Plug 機能の開発を目標としているが、我々の開発作業、動作検証作業は TX7/i9000 シリーズを用いて行っている。

開発作業は他社の技術者や Linux コミュニティ技術者と共同で行っている。例えば、Itanium 用 CPU Hot Plug 機能の開発は、CPU Hot Plug サポートプロジェクト[2]の一部として進めており、このプロジェクトは Intel 社製 Pentium や IBM 社製 Power4 なども開発の対象となっている。

TX7/i9000 シリーズは 4 つの Itanium2 プロセッサと Memory からなる Cell カードを最大 8 枚接続した ccNUMA (Cache-Coherent Non-Uniform Memory Access) システムであり、Cell 単位の Hot Plug を行う機能がハードウェア及びファームウェアに実装され

[1] http://atlas-64.sourceforge.net/

[2] Linux Hotplug CPU Support, http://sourceforge.net/projects/lhcs/

ている。システムの構成図を図 1 に示す。また、PCI/PCI-X カードを搭載可能な IO サブシステムは最大 8 個実装可能であり、同様に Hot Plug が可能となっている。

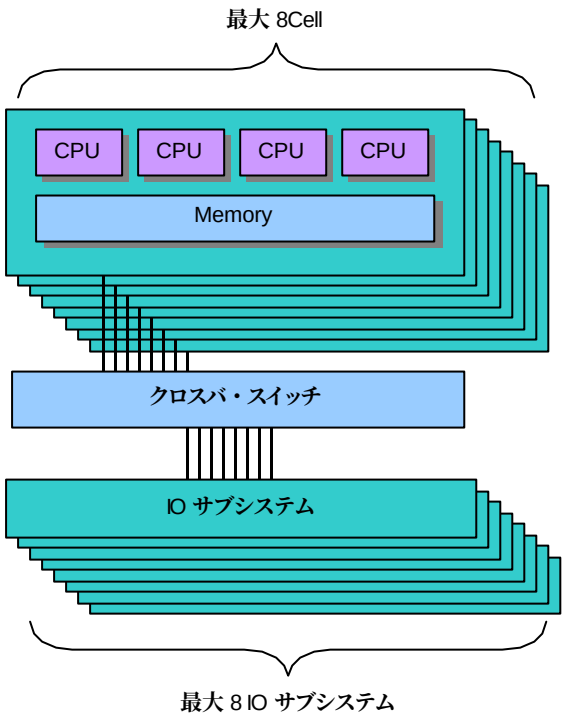

図 1　TX7/i9000 シリーズ システム構成図

2.2 Hot Plug の活用方法

CPU/Memory/IO Hot Plug 機能は以下のような目的で利用される。

- Hot Swap
  障害部品切り離しによるシステムダウン防止
  システムを停止せずに不良部品を交換
- Hot Add
  システムを停止せずにシステム資源を増強
- ダイナミック・パーティショニング
  システムを停止せずにパーティションのサイズ(CPU 数、Memory 量)を変更
- 省電力
  CPU の切り離しによる消費電力の抑制
- その他
  マルチプロセッサシステムで Suspend-To-RAM、ハイバネーションを実装するための基盤など

2.2.1 Hot Swap

ミッションクリティカルな業務を遂行しているシステムにおいては、いかなる目的の運用停止も許されない場合がある。このような場合に Hot Plug 機能を利用することによって、システムの運用を停止することなく故障部品を交換 (Hot Swap) することが可能となる。

また、Itanium プロセッサファミリには MCA (Machine Check Architecture) という CPU や Memory の故障を検出し、OS に通知する自己診断機能が備わっている。Atlas Project では、Itanium 用 Linux での MCA のサポート作業[3]を進めており、障害ログを取得する機能の 2.4 カーネルへの実装を行った。また、障害 Memory を page 単位で切り離す機能などの開発も現在進められている。

MCA の機能を使用すれば、ハードウェアによって自動訂正された Memory の 1bit エラーや CPU キャッシュの 1bit エラーなどの発生を検出することができる。また、ハードウェアによる訂正不可能な障害が発生した場合でも、障害箇所を特定し影響範囲を最小限に抑えシステムダウンを回避する機能が実装可能となる。

キャッシュや Memory の 1bit エラーはハードウェアの機能により自動的に修正されるが、障害が多量に発生する部品を継続的に使用し続けるべきではない。自動訂正が不可能なエラーが発生する前に、問題となる部品を交換しシステムダウンを防止すべきである。Hot Plug による障害部品の切り離しを行えば、システムを停止することなく故障部品を運用中のシステムから切り離すことができる。また、MCA ログを解析し、自動的に部品の切り離しを行うといった運用も可能である。

このように、Hot Plug 機能は、MCA 機能と組み合わせることによって、システムの可用性の向上に多いに貢献するものと期待できる。

2.2.2 Hot Add

システム負荷の増加に伴って、CPU や Memory、IO デバイスなど、システム資源の増強が望まれる場合がある。Hot Add を利用すれば、CPU/Memory や IO

[3] IA64 Linux MCA recovery, http://sourceforge.net/projects/mca-recovery/

デバイスなどを運用を停止せずに追加することができる。

最初は必要最小限のシステムで運用し、システム負荷の増加にあわせて、必要な資源の追加を行うという運用方法も実現できる。

2.2.3 ダイナミック・パーティショニング

単一システム上のリソース(CPU/Memory/IO など)を複数のパーティションに分割し、それぞれのパーティションを独立したシステムとして運用する機能がパーティショニング機能（図 2）である。システムを停止した状態（静的状態）でのみパーティションの構成を変更可能な機能をスタティック・パーティショニング、システム運用中に動的に構成を変更可能な機能をダイナミック・パーティショニングと呼ぶ。

ダイナミック・パーティショニングを実現するためには、各パーティション上で実行している OS が、CPU や Memory などのリソースを Hot Plug する機能を有している必要がある。

ダイナミック・パーティショニング機能を活用すれば、各パーティションのリソース量を再起動なしに変更するといった運用が可能となる。例えば、時間帯によってシステム負荷が増減するような運用において定期的にパーティションを最適な構成に変更したり、複数のグループで共同運用しているシステムのリソース分割を需要に応じて変更したりする、といった利用方法が考えられる。

パーティショニングと同等な機能は、メインフレームのようなプラットフォームでは VM (Virtual Machine) によって実装される場合もある(図 3)。VM とはシステム上に仮装的なシステムを複数構築する機能である。上述のパーティショニングとは異なり、ハードウェア的に分割するのではなく、OS とハードウェアの中間に位置する VM の層で論理的にシステムを分割する。VM によるリソースの設定をシステム運用中に変更する場合も、VM 上で実行中の OS の CPU/Memory Hot Plug 機能が必要である。

コンピュータシステムは、従来ハードウェアと OS が一体となって静的に構成されるものであった。しかし、ダイナミック・パーティショニングや VM 技術を導入することによって、コンピュータシステムは仮想化され、その構成も動的に変更が可能なものとなる。CPU/Memory/IO Hot Plug 機能はこのようにシステムの仮想化技術を前進させる基盤技術のひとつとしての役割も持っている。

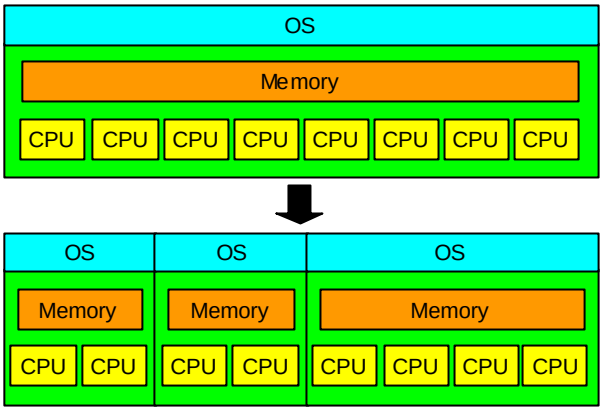

図 2　パーティショニング

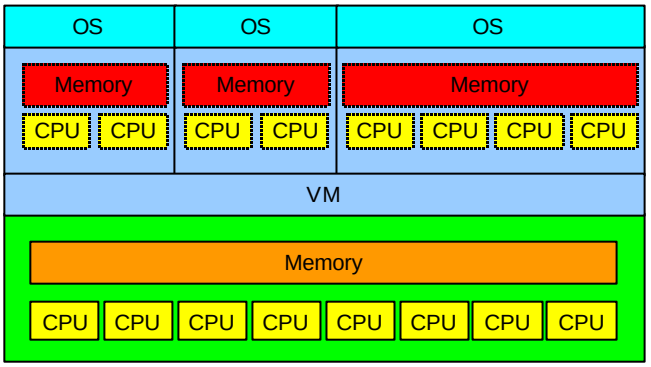

図 3　Virtual Machine

2.2.4 省電力機能

CPU Hot Plug は、省電力機能の一環としても活用が可能と考えられる。大型のマルチプロセッササーバでも、常に全ての CPU が処理を実行しているとは限らず、長時間 CPU が idle 状態になっている場合がある。負荷状況に応じて CPU を Hot Remove できれば、CPU の消費する電力を節約することが可能と考えられる。システム負荷を監視する機能と CPU Hot Plug を組み合わせることで、システムが必要以上の電力を消費することを防止できる。

2.2.5 その他の活用方法

2.5 カーネルでサポートされた swsusp[4]は、Suspend-to-RAM やハイバネーションといったシス

[4] http://sourceforge.net/projects/swsusp/

テム停止機能をソフトウェアで実現している。しかし、これらの機能はユニプロセッサでしか動作しない。

マルチプロセッサシステムでシステムを停止させるためには、停止作業を行っている CPU 以外の CPU の動作を管理する必要があり、ユニプロセッサシステムに比べて複雑である。この問題を CPU Hot Plug を活用することで解決しようとする動きがある。CPU Hot Plug 機能を活用し、システム停止処理内で稼動 CPU を 1 つにしてしまえば、ユニプロセッサシステムと共通の方式でシステムの停止処理を実現することが可能になる。

Hot Plug 機能は、このように RAS 機能の強化という側面だけではなく、様々な機能実装の基盤となり得る機能と考えられる。

3 機能と実装

3.1 ACPI と Hot Plug

ACPI (Advanced Configuration and Power Interface) 仕様[1] は、ノートパソコンなどで利用されている電源や省電力管理だけではなく、コンピュータの構成の記述や制御に関しても規定している。Hot Plug の機能に関しても ACPI を利用することにより、対象デバイスの Hot Plug イベントのハンドリング、電源制御、状態の取得ができるようになっている。

ACPI 仕様には、システムハードウェアの構成を記述するための言語としての AML (ACPI Machine Language) と、イベントの通知の機構が定義されている。AML は特定の CPU の命令セットに依存しない一種のプログラミング言語であり、これによってデバイスの構成が記述できるとともに、動的な構成の変更にも対応できるようになっている。AML によって、ハードウェアベンダーは固有の実装の詳細を隠蔽し、OS 毎に別のドライバを開発する手間を省くことができる。

システムのデバイス構成は AML によってツリー状の名前空間上に表現され、これをもとに OS はシステム内の各コンポーネント間の依存関係を知ることができる。例えば、TX7/i9000 シリーズの場合、Cell や IO サブシステムは module device と呼ばれる複数のデバイスのコンテナとなるデバイスによって表現され、Hot Plug はそれらに対するイベントとして表現されている。

Hot Plug の契機がカーネルの外的要因 (例: イジェクトボタン押下など) によって発生する際には ACPI 互換ハードウェアでは SCI (System Configuration Interrupt) と呼ばれるイベントが発生する。カーネルに組み込まれた ACPI ドライバは、このイベントを契機に Hot Plug 処理を起動することができる。このイベント機構を使わず、OS の内部からの契機 (例: ユーザのコマンドラインからの Hot Remove 要求) から Hot Plug 処理を開始することも可能である。

Linux は、Intel 社による ACPI ドライバのリファレンス実装である ACPI CA[5]をもとにした ACPI ドライバを採用している。ACPI ドライバは、Hot Plug などのイベントの管理や名前空間へのアクセスなどの API 群を提供している。

また、2.5 カーネル以降では Linux Driver Model[2] によりカーネル内部にシステム全体のデバイスツリーを持つようになった。Linux 向け ACPI ドライバも、このモデルへの統合が進められている。

本論文執筆時点 (2002 年 7 月末) では ACPI イベントと CPU/Memory/IO Hot Plug との間のインターフェースが完成していないが、2.5 カーネルに機能を盛り込めるよう開発中である。

3.2 CPU Hot Plug

3.2.1 設計

CPU Hot Plug サポートの開発内容は、次の 3 項目に大別できる。

(a) CPU 数不変の前提の撤廃
(b) CPU Hot Add 処理
(c) CPU Hot Remove 処理

以下に各項目についての必要機能を説明する。

(a) CPU 数不変の前提の撤廃

2.4 カーネルでは、CPU 数はシステム初期化時に `smp_num_cpus` に記録した後、その情報が不変であることを前提にしていた。また、論理 CPU 番号は `0,1,2, ..., smp_num_cpus-1` と連続していることも前提としていた。

[5] http://developer.intel.com/technology/iapc/acpi/

CPU Hot Plug で CPU を削除した場合、連続していた CPU 番号に抜けが生じる場合がある。CPU 番号の連続性を維持するために全 CPU の番号を再割り当てする方式も考えられるが、運用中の CPU 番号の変更はカーネル内部のマルチプロセッサに関する大部分のロジックを変更しなければならない。むしろ、CPU 番号の連続という前提を撤廃し、非連続な CPU 番号を前提とした構造に変更する方が変更箇所は少なく、現実的である。

ドライバなどが CPU 毎の領域を確保している箇所にも変更が必要になる。CPU の追加の可能性を考え、あらかじめマクロ NR_CPUS で定義されている CPU の最大数分の領域を確保しておくか、CPU 追加時に動的に領域を割り当てる処理を追加する、といった対応が必要となる。

CPU 番号が非連続になる影響は、カーネルだけではなくアプリケーションやライブラリにも存在する可能性がある。CPU 数の変更をプロセスに通知する手段を用意しておく必要がある。

(b) CPU Hot Add 処理

カーネルの初期化処理を行う関数や、初期化時にしか参照されないデータ領域は init セクションと呼ばれる箇所に配置され、初期化終了時に解放される。CPU 初期化処理も init セクションの一部であったが、CPU Hot Add を実現するためには、CPU 初期化に必要な関数やデータ構造を全て init セクションから切り出す必要がある。また、CPU の初期化処理を他のシステム初期化処理から独立させ単独で実行可能な構造にする必要がある。

カーネルは CPU の初期化時に以下の作業を行う。

- idle の生成
- 各種 register の初期化
- タイマーの設定
- IRQ の設定
- CPU 毎に所持する構造体・変数の確保・初期化
- migration_thread, ksoftirqd などの CPU 毎に必要なカーネルスレッドの起動

(c) CPU Hot Remove 処理

CPU Hot Remove を実現するためには、以下の機能が必要となる。

- CPU 毎に所持する構造体・変数の解放
- idle, ksoftirqd, migration_thread などのカーネルスレッドの停止
- CPU 停止に伴う IRQ の設定変更
- 対象 CPU の runqueue 上のプロセスを他 runqueue へ移動
- CPU 停止時のキャッシュの flush と CPU の HALT 処理
- 同期処理

図 4 に CPU Hot Remove 処理の流れを示す。

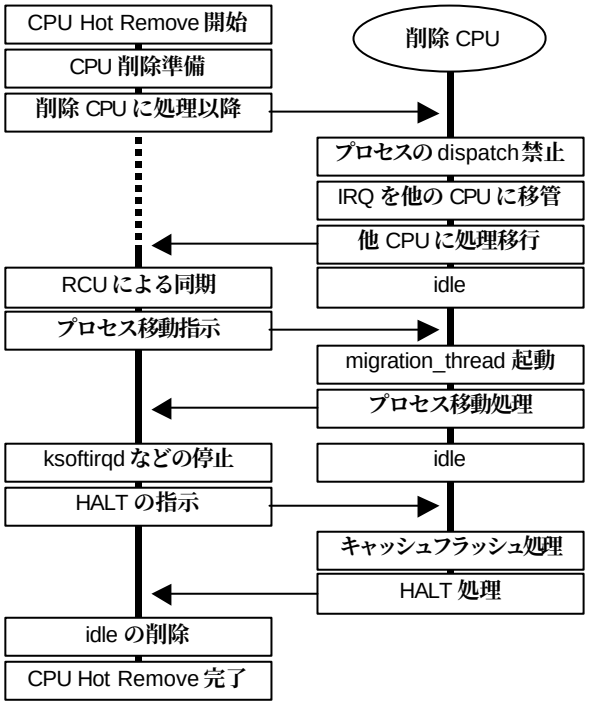

図 4　CPU Hot Remove 処理の流れ

3.2.2 実装

(a) CPU 数不変の前提の撤廃

稼動 CPU 数を示す変数 smp_num_cpus を参照している箇所を全て書き換える作業を行った。例えば以下のようなコードはカーネルソースの至るところに見受けられる。

```
for (cpu = 0; cpu < smp_num_cpus; cpu++) {
```

```
        /* do something for each cpu */
}
```

このような箇所を全て、以下のように書き換える必要があった。

```
for (cpu = 0; cpu < NR_CPUS; cpu++) {
        if (!cpu_online(cpu))
                continue;
        /* do something for each cpu */
}
```

この変更によって 2.5 カーネルでは smp_num_cpus という変数は廃止されることになった。また、稼動中の CPU を確認するためのビットマップ (cpu_online_map)、稼動中の CPU 数を確認するためのマクロ (num_online_cpus()) なども用意した。

CPU 毎に領域を確保している箇所に関しては、CPU 数の最大数分の領域を確保するように変更した。これを CPU 追加時に動的に確保するように変更するかは、該当部分のコード管理者の判断に委ねることとした。

(b) CPU Hot Add 処理

Linus Torvalds の発案により、2.5 カーネルでは CPU の初期化方式を大きく変えることになった。BSP (Boot Strap Processor：最初に立ち上がる CPU)が全ての初期化処理を実施した後で、各 AP (Application Processor：BSP 以外の CPU)を CPU Hot Add の手順で起動するように変更した。この変更により、通常の AP 立ち上げ処理と CPU Hot Add の処理という二系統の類似した初期化処理を別々に実装する必要が無くなり、カーネルの保守性を維持しながら CPU Hot Add を実現することが可能となった。2.4 カーネルと 2.5 カーネルの初期化手順の違いを図 5,6 に示す。

(c) CPU Hot Remove 処理

3.2.1 節で記述した必要な機能の各項目の実装について全てを記載することは困難なので、特徴的な開発項目を 3 点だけ紹介する。

(1) プロセスの移動

2.5 カーネルでは Ingo Molnar による O(1)スケジューラの採用によって runqueue が CPU 毎に確保されるようになった。CPU を停止する際は、停止する CPU の runqueue 上のプロセスを全て他の CPU に移動させる必要があった。

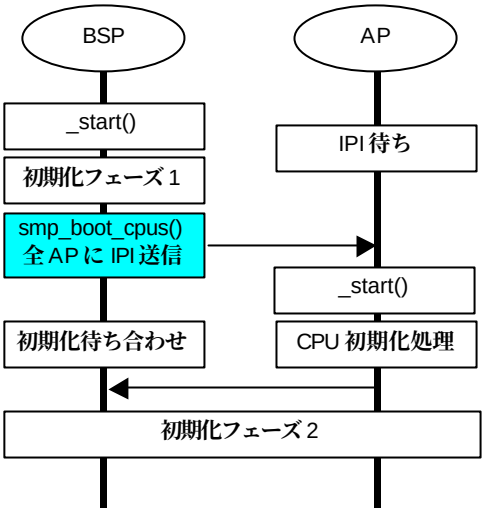

図 5　2.4 カーネルの CPU 初期化処理

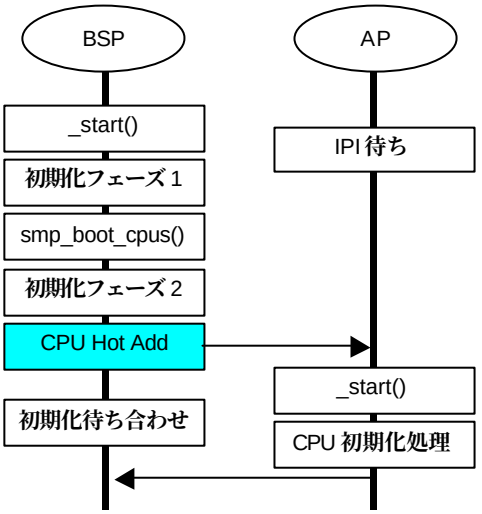

図 6　2.5 カーネルの CPU 初期化処理

O(1)スケジューラではプロセスの CPU 間の移動は migration_thread という CPU 毎に起動されるカーネルスレッドにのみ許されている。この migration_thread 自体も他のプロセスと同様に runqueue に繋がっている存在である。このため migration_thread のみは特殊な扱いをする必要が生じた。他にも、ksoftirqd のようにカーネルスレッドは場合によって特定 CPU に強く結び付けられている場合があり、CPU Hot Remove 機能を実装するにあたり、

全てのカーネルスレッドについて、CPU 停止の影響を判断しながら実装を行った。

(2) RCU (Read-Copy-Update)[3] による同期処理

CPU の有無は cpu_online_map という運用中に変更される可能性のあるビットマップで示されるようになったため、CPU Hot Plug の処理全体とこの変数の参照を排他制御する必要が生じた。一般的には、グローバル変数はロックプリミティブを利用して排他制御しなければならないが、高頻度で参照される部分を排他アクセスすることはスケーラビリティに悪影響を及ぼす。cpu_online_map はまれにしか変更されない変数であるため、そのような変数を変更するときに有用な同期方法である RCU を採用した。RCU により、cpu_online_map は排他制御を行うことなく読み込むことができる。RCU により、CPU Hot Remove は、性能に一切悪影響を与えない実装を行うことができた。

(3) CPU の HALT 処理

物理的に CPU を抜くためには、対象 CPU 上のリソースに対するアクセスが一切発生しないことを OS が保証する必要がある。なぜなら、もしキャッシュデータが Hot Remove 中の CPU 内に残っていると、他 CPU が同一キャッシュラインを参照した際に Remove 対象の CPU に対する通信が発生してしまう可能性があるからである。

Itanium プロセッサ搭載システムにおいて Remove 対象の CPU を完全に沈黙させるために、キャッシュの完全な flush と invalidate を行い、さらにファームウェアの持つ HALT 処理を実行するアーキテクチャ依存の関数を実装した。

3.2.3 開発作業状況と課題

CPU Hot Plug の開発は Rusty Russell と共同して進めており、現在、i386, ia64 (Itanium), ppc, s390 といったアーキテクチャをサポートしている。CPU Hot Plug の成果は 2.5 カーネルに順次取り込まれており、2.6 カーネルでは正式にサポートされる予定である。また 2.4.18 カーネルをベースにした Itanium 用コードを Atlas Project の成果として公開している[6]。

以下に残作業・検討項目を示す。

- 残項目の 2.5 カーネルへの取り込み
  CPU Hot Remove とユーザインターフェース部分が未提出である。
- アーキテクチャ依存部分のマージ
  Itanium 用コードは、別パッチで管理されており、最終的なマージが完了していない。
- インターフェース
  ACPI 経由のイベントをサポートするインターフェースと、ユーザレベルからのインターフェースを実装する必要がある。後者に関しては、2.5 カーネルで採用された driverfs[4]との連携も検討しなければならない。

3.3 Memory Hot Plug

3.3.1 設計

Memory Hot Add に関しては、既存の Memory 初期化処理と同様の方式を用いて実装することができるが、Memory Hot Remove に関しては使用中の page を再配置する必要であり実装は容易でない。特にカーネルが使用している page を再配置にするには、Linux カーネルの基本構造に大幅な改造が必要となり、実装は非常に困難である。現段階では、Memory Hot Remove は設計の検討段階であり、実装は行っていない。

(a) Memory Hot Add

カーネルの Memory 初期化処理の流れを以下に示す。

1. 利用可能な Memory アドレスの範囲の確認
2. 全 Memory を bootmem として初期化
   bootmem 用 mem_map を作成し、簡易な構造の bootmem allocator を使用可能とする。bootmem allocator は Memory 初期化処理が完了するまでの期間でのみ利用される。
3. zone 情報の構築
   Memory を DMA/NORMAL/HIGHMEM に分割し、zone allocator[5]の参照するデータ構造を作る。
4. mem_map の構築
   各 page を管理する page 構造体の初期化を行う

[6] Atlas Project, http://atlas-64.sourceforge.net/

5. 全 page の解放
   カーネルで使用している以外の page を全て解放し利用可能にする。同時に bootmem allocator を使用不可にする。

Memory Hot Add は、この手順と同様の方式で、追加された Memory を初期化する。また、追加された Memory は独立した pg_data 構造体および独立した mem_map で管理する。この方式には次のようなメリットがある。

- 通常の Memory 初期化処理を最大限流用できる
- 初期化処理を、既存の Memory 管理構造から隔離して実行できるので排他制御が容易
- 2.4 カーネルの共通部分では、すでに複数の pg_data 構造体の存在が前提になる構造になっており、必要となる変更の箇所は少ない。

(b) Memory Hot Remove

Memory Hot Remove を実装するためには主に以下の機能が必要となる。

- 削除が必要な page の選別と利用状況の確認
- ユーザ空間で使用している page の再配置/解放
- カーネルで使用している page の再配置/解放
- page のフリーリストからの削除
- pg_data, mem_map, zone 情報などの削除
- 全 CPU のキャッシュのフラッシュと TLB の更新
- ハードウェアの観点で Memory との接続の切断

削除対象となった page が、ユーザ空間で使用されているのか、カーネルで使用されているのかは 2.5 カーネルで Rik van Riel によりサポートされた reverse mapping によって判別可能となった。この機能をベースに page 単位の再配置/解放処理の実装を検討する。

削除対象となった page がユーザ空間で使用されている anonymous page であった場合、通常のページアウトと同様の手段でページアウトするか、他の未使用 page に再配置することが可能である。ただし、入出力処理のために page がロックされていた場合などの対応は別途考慮する必要がある。

対象となった page がカーネルで使用されている場合、page が page_cache や buffer などの解放可能な page であれば既存の解放処理を利用することができる。しかし、slab やカーネル text/data などに使用されている Memory の場合、単純な再配置によって領域を解放することはできない。page の再配置を行う処理期間は、その page には一切アクセスしないという保証が必要であり、現在のカーネルの構造では実現は困難である。

カーネルで使用する Memory は Hot Remove 不可とする制限を設ければ、実装に関する制約はかなり緩和される。カーネルで使用する page を特定の領域に限定し、その領域以外を Hot Remove 可能にするという方式で実装するのが現実的だと考えられる。

3.3.2 実装

Memory Hot Add の実装は試作段階ではあるが、2.4 カーネルに discontigmem[7]という ccNUMA 対応のパッチを加えたものをベースに行った。discontigmem は、CPU/Memory/IO Hot Plug サポートと同様に Atlas Project 内で開発が進められている機能である。discontigmem では ccNUMA システムにおける性能向上を実現するために、pg_data/mem_map は Cell 単位に構築する構造になっている。試作カーネルでは、Hot Add する Memory の単位を Cell 単位に固定し、Hot Add Memory 機能を実装した。

処理の流れは以下のようになる。

1. Hot Add 起動
2. 対象となる Cell の Memory アドレス範囲の取得
3. bootmem の作成
   - init_bootmem_node()
   - free_bootmem_node()
   - reserve_bootmem_node()
4. zone 構造体の生成
5. mem_map の作成
6. page の解放
   - free_all_bootmem_node()

[7] Linux discontiguous memory support, http://sourceforge.net/projects/discontig/

7. zonelist の再構築

ここで記述した関数名は 2.4 カーネルで実存する関数で、これらの関数は__init 宣言を削除する以外の変更を加えずに Memory Hot Add で利用することが可能であった。

3.3.3 開発作業状況と課題

試作した Memory Hot Add のパッチは本プロジェクトの成果物として SourceForge のプロジェクトサイト[8]にて公開している。今後も、2.4 カーネル + discontigmem をベースに作業を続ける予定だが、最終的な目標は 2.5 カーネルへの取り込みである。

以下に残作業・検討項目を示す。

- Hot Remove の設計
  カーネルで使用している page の存在しないメモリ領域の Hot Remove 実装
- Add/Remove の単位の見直し
  Cell 単位ではなく、より細かい単位による追加・削除が望ましい
- インターフェース
  CPU Hot Plug と同様に ACPI 経由のイベントをサポートするインターフェースと、ユーザレベルからのインターフェースを実装する必要がある。後者に関しては、2.5 カーネルで採用された driverfs との連携も検討する必要がある。

3.4 IO Hot Plug

3.4.1 機能

ディスクや Ethernet といった IO デバイスは、通常 CPU/Memory との間にペリフェラルバスやアダプタを経由して接続される。今日のシステム構成では、PCI バスを経由して接続されるのが一般的である。一部のサーバクラスのシステムでは、ハードウェアの機能として、OS 稼働中に PCI カードの交換を可能にする PCI Hot Plug[6]が利用可能である。また、PCI バスはホストバスブリッジを通して CPU/Memory と接続されているが、システムによってはこのホストバスブリッジの Hot Plug も可能である。これらの IO 系全体の Hot Plug を総称して、IO Hot Plug と呼ぶ。

IO Hot Plug では、すでに 2.4 カーネルに存在している Hot Plug に関する機能に加えて、以下の機能を実装した。

- ACPI を利用した PCI Hot Plug
- 割り込みコントローラの Hot Plug
- その他チップセットデバイスの Hot Plug

以下の節では、それぞれの詳細について解説する。

3.4.2 ACPI を利用した PCI Hot Plug

ACPI は、3.1 節に記述したように、システムの構成制御に利用されており、ACPI を利用することによって特定のハードウェア実装に依存しない形での PCI Hot Plug が実現可能である。

カーネル 2.4.16 以降では PCI Hot Plug のドライバ[9]が取り込まれている。しかし、これは Compaq (現 HP) 社製の PCI Hot Plug コントローラ専用の PCI Hot Plug ドライバである。我々はこの PCI Hot Plug ドライバをベースに、ACPI を利用した特定のハードウェアに依存しない ACPI PCI Hot Plug ドライバを実装した。これにより、Itanium アーキテクチャのプラットフォームだけでなく、i386 プラットフォームにおいても、単一のドライバで PCI Hot Plug を実現することができた。

PCI デバイスの Hot Plug をサポートするためにカーネルが必要とする主な機能は、以下の通りである。

(a) Hot Plug 可能な PCI スロットの構成情報取得
(b) バス番号/IO ポート/メモリマップド IO 空間の割り当て/解放
(c) スロット単位の電源制御
(d) カーネル内部の pci_dev 構造体の追加/削除

以下に各項目について説明する。

(a) Hot Plug 可能な PCI スロットの構成情報の取得

AML の名前空間はシステム内の Hot Plug 可能な PCI スロットに関する各種の情報を提供する。この情

[8] Linux Hotplug Memory Support, http://sourceforge.net/projects/lhms/

[9] PCI Hot Plug for Linux, http://sourceforge.net/projects/pcihpd/

報を利用して、カーネル内で Hot Plug 可能な PCI スロットを特定するよう実装した。

(b) バス番号/IO ポート/メモリマップド IO 空間の割り当て/解放

PCI Hot Plug をサポートするシステムでは、起動時に BIOS が予め余裕を持ったリソース(バス番号、IO ポート、メモリマップド IO 空間)をホストバスブリッジ単位で割り当てる。ACPI PCI Hot Plug ドライバは、BIOS が割り当てたリソースを ACPI を通じて取得し、各デバイスへの割り当て/解放を行うよう実装している。

(c) スロット単位の電源制御

ACPI 仕様によりデバイス単位の電源制御に関する標準的なメソッド群が定義されており、ハードウェア個別の実装は抽象化されている。我々の実装でも、それらのメソッド呼び出しを利用し、PCI スロットの電源制御を行うよう実装している。

(d) カーネル内部の pci_dev 構造体の追加/削除

カーネル内部では PCI デバイス毎に pci_dev 構造体を持っており、Hot Plug 時にこれらを追加/削除する必要がある。ACPI PCI Hot Plug ドライバではこれらを動的に追加/削除する機能を実装した。

以上のように実装を行った ACPI PCI Hot Plug ドライバは、本論文執筆時点 (2002 年 7 月末) で、Alan Cox のパッチ 2.4.19-rc3-ac2 以降に含まれており、2.4.20 以降の正式リリースで利用可能になる予定である。

3.4.3 割り込みコントローラの Hot Plug

割り込みコントローラとは、周辺機器からの割り込みを CPU に配送するための割り込みのルーティングなどを行うデバイスである。割り込みコントローラの詳細はアーキテクチャ毎に異なり、Linux でもアーキテクチャ毎に別々の実装を行っている。

Itanium プラットフォームでは、PCI デバイスからの割り込みに関して、IO SAPIC (IO Streamlined Advanced Programmable Interrupt Controller)を割り込みコントローラとして利用しているが、従来の実装では起動時に見つけたコントローラのみを初期化し、その後は、その数は変化しないことを前提としていた。

IO Hot Plug の実装に際して、この実装の見直しを行い、システムに新たに IO SAPIC が Hot Add されたときにも正しく割り込みがプログラムされるよう改良を行った。Hot Remove に関しては、各デバイスを停止した状態ではすでに割り込みが発生しないため、IO SAPIC に対して特別な処理を行う必要はない。

3.4.4 チップセットデバイスの Hot Plug

システムを構成するチップセットは、電気的には実際に PCI バス上に存在していなくても、便宜上そのレジスタを PCI のコンフィグ空間からアクセスできるように設計されていることが多い。例えば TX7/i9000 シリーズでは、各 Cell に、Cell Configuration Space と呼ばれるレジスタ群が定義されており、PCI コンフィグ空間からアクセスが可能となっている。

このようなデバイスが IO ポートやメモリマップド IO 空間を使うことや、OS のデバイスドライバが直接アクセスを行うことは稀であるが、カーネルはこれらを通常の PCI デバイスとして扱っている。そのため、IO Hot Plug 時にはこれらに対しても pci_bus 構造体や pci_dev 構造体の割り当て/解放を行う必要がある。これは、PCI Hot Plug の一種ではあるものの、通常の PCI Hot Plug が行っているようなスロットの電源の制御などは伴わない。

カーネルにおいてこのような Hot Plug に対応するために、PCI Hot Plug ドライバのコードから、PCI バスおよびデバイスの再検出/切り離しを行う部分を切り出し、スロットを持たない PCI デバイスの追加/削除を可能にする変更を行った。

3.4.5 開発作業状況と課題

本節で論じた IO Hot Plug は、TX7/i9000 シリーズ固有の実装ではなく、PCI Hot Plug や ACPI などのオープンな仕様をもとに実装されたものである。同等のハードウェアが開発された場合、本プロジェクトの成果をそのまま利用して Hot Plug が実現できると考えている。

PCI Hot Plug に関しては、現在、PCI SIG による標準的な PCI Hot Plug コントローラの仕様[7]がすでに存在しており、将来的にはこの仕様に準拠したハードウェアが一般的になると考えられる。この仕様では、

現在 ACPI で抽象化されている部分が、より細かい制御の可能なハードウェア仕様として記述されている。ACPI PCI Hot Plug ドライバも、この仕様に準拠するよう拡張を行う必要がある。

4 まとめ

本論文では CPU/Memory/IO の Hot Plug 機能の目的と活用方法を紹介し、実装方式の解説を行った。また、Hot Plug 機能は RAS 向上に貢献するだけでなく、運用性向上など様々な可能性を持っていることを示した。

我々は現在も Hot Plug 機能の開発作業を継続しており、2.6 カーネルでの正式なサポートを目指している。

References

[1] Intel, Microsoft, Toshiba, Phoenix, Compaq, ACPI spec 2.0a, http://www.acpi.info/, 2002

[2] Patrick Mochel, Linux Driver Model, http://www.kernel.org/pub/linux/kernel/people/mochel/doc/driver-model.txt

[3] Read-Copy-Update, http://lse.sourceforge.net/locking/rclock.html

[4] Patrick Mochel, Device Drivers, http://www.kernel.org/pub/linux/kernel/people/mochel/doc/driver.txt

[5] The Zone Allocator, http://home.earthlink.net/~jknapka/linux-mm/zonealloc.html

[6] PCI SIG, PCI Hot-Plug Specification Rev. 1.1, 2001

[7] PCI SIG, PCI Standard Hot-Plug Controller and Subsystem Specification Rev. 1.0, 2001