2007－OS－105 (14)
2007／4／6

継続概念による割り込みなし並列I/O処理モデル

泉 雅昭† 青野 光洋† 雨宮 聡史† 松崎 隆哲††
日下部 茂‡ 谷口 秀夫‡‡ 長谷川 隆三‡ 雨宮 真人‡
†九州大学 大学院システム情報科学府
††近畿大学 産業理工学部
‡九州大学 大学院システム情報科学研究院
‡‡岡山大学 大学院自然科学研究科

我々はハードウェア割り込みなどを扱うOSを含めて，一切の処理を“細粒度走り切りスレッド”で構成し効率的に実行するFuce (FUsion of Communication and Execution) アーキテクチャを提案している．Fuceアーキテクチャはスレッド間の依存関係を継続で定め，継続により多数のスレッドを実行制御する継続モデルをマルチスレッド実行モデルに採用する．Fuceアーキテクチャは通常のプログラム処理と通信処理の融合を目的としている．本稿では，従来の割り込みモデルとは全く異なった継続概念による並列I/O処理モデルを詳述する．

Interrupt-less Parallel I/O Processing Based on Continuation Model

Masaaki Izumi†, Aono Mitsuhiro†, Satoshi Amamiya†, Takanori Matsuzaki††,
Shigeru Kusakabe‡, Hideo Taniguchi‡‡, Ryuzo Hasegawa‡ and Makoto Amamiya‡
†‡Graduate School of Information Science and Electrical Engineering, Kyushu University
††School of Humanity-Oriented Science and Engineering, Kinki University
‡‡Graduate School of Natural Science and Technology, Okayama University

We propose the Fuce architecture based on dataflow computing model, which aims at fusion of communication and execution. This architecture takes as a multiple thread execution model the continuation-based multithreading model that manages dependency among fine-grain “uninterruptible” threads. The continuation-based multithreading model can realize parallel I/O processing by cooperating with processor and operating system. Our model is different from conventional I/O processing models based on “interrupt.” We present the continuation-based parallel I/O processing model, and evaluate parallel I/O processing performance based on our model.

1 はじめに

近年のネットワーク利用の多様化や利用者の増加により，高いI/O処理能力と演算能力を備えた高性能サーバが求められている．高性能サーバではネットワークにおける低遅延と高バンド幅の実現やオペレーティングシステム (OS) におけるI/O処理のオーバヘッド削減が要求される．

そこで，I/Oバスやネットワークの高速化とバンド幅向上が試みられ，複数のNetwork Interface Card (NIC) を利用したマルチホームホストなどのネットワーク構築技術が開発されている．また，近年の高度な半導体技術によりチップ内搭載可能となった豊富なトランジスタ資源を利用して，同一チップ内にプロセッサと共にNICを搭載するアーキテクチャの研究が行われている．その研究ではシミュレーションにより高いI/O処理性能の達成を予測している [5].

プロセッサにおいては，単一命令流のみからの並列性抽出には限界があるため [3]，複数の命令流から並列性を利用するスレッドレベル並列性の利用を図ったChip MultiProcessor (CMP)[4] などが研究されている．商用プロセッサでは，最大4スレッドを並列実行可能なProcessor Element (PE) を8個搭載しチップ全体として最大32スレッドが並列実行可能なSPARC T1やIBM POWER5などが登場するに至った．上記のCMPでは，I/Oなどの割り込み処理についてスレッドレベル並列性を活かすため，マルチプロセッサシステムで用いられた割り込みの負荷分配機構を利用する．その機構により，CMP内の任意のPEでI/O処理を実行可能にし，OSも任意のPEでI/O処理を可能とした．

しかし，I/O 処理を行う PE を可変にする場合は，固定する場合と比較してスケジューリングコストの増加などのペナルティを負う．そのため，従来は I/O 処理を行う PE を固定することが多い．

今後も，CMP と OS はさらに高速なネットワークへ対応し，夥しい数の I/O 処理を短時間に行う必要がある．例えば，データパケットのサイズを最大の 1.5KB に固定した 10Gb Ethernet では，毎秒最大約 80 万個のデータパケットが CMP に届き，データパケットの数分の 1 から同数個の応答パケットもさらに届く．また，高性能サーバでは I/O 処理と同時に高い演算処理能力が求められる．

これらの要求に応えるために，我々はあらゆるプログラム，およびハードウェア割り込みなどを扱う OS を含めて，一切の処理を“細粒度走り切りスレッド”で構成し効率的に実行する Fuce (FUsion of Communication and Execution) アーキテクチャ[2] を提案している．Fuce アーキテクチャはスレッド間の依存関係を継続 [1] で定め，継続により多数のスレッドを実行制御する継続モデルをマルチスレッド実行モデルとして採用する．

本稿では割り込みによる並列 I/O 処理の問題点を整理し，Fuce アーキテクチャにおける従来とは全く異なる継続概念による並列 I/O 処理を提案する．

2 割り込みによる並列 I/O 処理

今後，高性能サーバでは多数の I/O 処理を同時に効率的に処理するため，複数の PE を搭載する CMP が利用されることが考えられる．

従来の CMP には，一つのデバイスからの割り込みを多数の PE に対して分配できる機構が搭載されている．例えば，Pentium4 では Advanced Programmable Interrupt Controller を用いて，割り込みの分配を実現する．このような機構により，割り込み処理を負荷の小さい PE に配分し，I/O 処理性能を高めることを図っている．

以下では，従来の CMP と Linux Kernel 2.6 における割り込みを用いた I/O 処理について述べる．その後に，従来の並列 I/O 処理の問題点を挙げる．

2.1 I/O 処理

Linux Kernel 2.6 では，実行可能なスレッドを保持する Run Queue を PE 毎に持ち，結果待ち状態のスレッドを保持する Wait Queue を資源毎に持つ．図 1 に Linux Kernel 2.6 での I/O 処理の流れを示す．

例えば，read システムコールによりハードディスクへのアクセスを行う際の I/O 処理は以下のようになる．

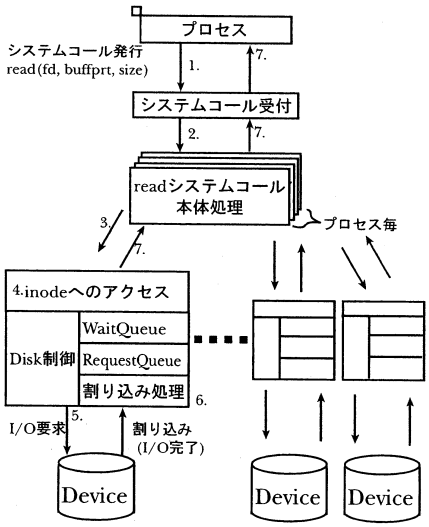

図 1: I/O Processing model in Linux.

1. プロセスからのシステムコール発行

プロセスの I/O 要求スレッドが必要な引数と共に read システムコールを発行して，OS へ制御を移行する．

2. システムコール受付

OS はシステムコールを受付て，read システムコール本体の処理に移行する．

3. read システムコール本体処理

read システムコール本体の処理を行う．ユーザから受取ったファイル記述子を用いて，ファイル管理に利用する inode 構造体を求める．I/O 要求スレッドはシステムコールの処理においてその inode 構造体へのアクセスをファイルシステムに依頼する．

4. inode 構造体へのアクセス

I/O 要求スレッドはファイルシステムにおいて inode 構造体からハードディスク上のデータ位置を求め，ディスク制御部へ移行する．

5. ディスク制御部

I/O 要求スレッドは I/O 要求を行い，同時に，Wait Queue に自身を挿入してプロセッサを明け渡し，I/O 完了を待つ．ディスク制御部において，I/O 要求情報が Request Queue に存在しない場合はハードディスクに対して I/O 要求を発行する．存在する場合は，Request Queue に I/O 要求の発行に必要な I/O 要求情報を挿入する．

6. 割り込み処理

割り込み発生後，プロセッサは割り込みハンドラ

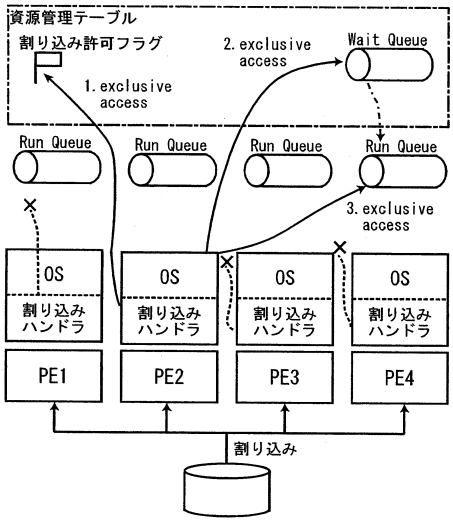

図 2: I/O Processing model on CMP.

を起動する．割り込みハンドラは DMA 転送されたデータを I/O 要求スレッドへ渡し，Wait Queue 中の I/O 要求スレッドを RUNNING 状態にし，スケジューラを呼び，割り込み処理を終了する．また，その時に，I/O 要求情報が Request Queue に存在する場合，割り込みハンドラは Request Queue から I/O 要求情報を取り出し，I/O 要求を発行する．

7. I/O 完了後の処理

I/O 要求スレッドは起床した後，I/O 完了を確認すると，Wait Queue から自身を外し，ユーザが指定したユーザ空間のバッファにデータをコピーする．それから必要な処理を施した後に，OS からプロセスへ復帰し，プロセスは処理を再開する．

2.2 CMP における I/O 処理

前節では，Linux Kernel 2.6 における I/O 処理を述べた．さらに，本節では CMP 上での I/O 処理に必要となる処理を述べる．図 2 に，CMP での割り込み処理と I/O 待ちからの起床処理を行う際に，I/O 要求と結果取得を行った PE が異なる場合について示す．

CMP において，I/O 要求と結果取得を行った PE が同一の場合と異なる場合ではコストが違う．

2.2.1 I/O 要求と結果取得を行った PE が同じ場合

CMP は一つの I/O 完了の信号を複数の PE に分配できる機構を備えるため，割り込みを複数の PE に分配する．全ての PE は走行中の処理を止め，PE 毎の割り込みハンドラが走行開始を試みる．この内一つの割り込みハンドラのみが割り込み許可フラグを立てる (1.)．他の割り込みハンドラは割り込み許可フラグが立てられたことを確認し，処理を終了する．これにより，一つの PE(図中の PE2) 上のみで割り込みハンドラが走行する．

割り込み処理終了直前，割り込みハンドラは，Wait Queue 中の対応する I/O 要求スレッドを RUNNING 状態にする (2.)．I/O 要求と結果取得を行った PE(PE2) が同一であるため，I/O 要求スレッドを自身が走行した PE(PE2) の Run Queue に挿入し (3.)，スケジューラを呼ぶ．RUNNING 状態にする際，Wait Queue 中の他の I/O 要求スレッドも Run Queue に挿入することがあり，I/O が未完了であることを確認して再び自身を Wait Queue に挿入する．

2.2.2 I/O 要求と結果取得を行った PE が異なる場合

先程と同様に，割り込み処理終了直前，割り込みハンドラは I/O 完了を I/O 要求スレッドへ通知しようとする．I/O 要求と結果取得を行った PE が異なるため，割り込みハンドラは I/O 要求を行った PE(PE4) の Run Queue をロックし (3.)，Wait Queue 中の対応する I/O 要求スレッドを RUNNING 状態にして (2.) その Run Queue に挿入し．先程と同様に，他の I/O 要求スレッドも Run Queue に挿入することがある．

その後，割り込みハンドラは I/O 完了通知のために I/O 要求を行った PE(PE4) に対してスケジューラの呼び出しを試みる．そのため，プロセッサ間通信である割り込みをその PE(PE4) に発生させる．スケジューラにより I/O 要求スレッドは機会を得て，起床し，割り込みハンドラがユーザ空間バッファにコピーしたデータを受取る．

2.3 従来の並列 I/O 処理の問題点

従来の CMP と Linux Kernel 2.6 による割り込み処理を念頭におき，従来の並列 I/O 処理の問題点について述べる．

1. 割り込みによる走行中の処理の退避・復帰
2. 割り込み許可フラグへの排他制御オーバヘッド
3. 各資源の獲得時の排他制御オーバヘッド
   CMP では，複数のシステムコールが同一資源に対してアクセスすることがあり，その資源管理情報を排他制御する必要がある．このように，各資源の獲得時に排他制御が必要になる．各資源の管理情報には Request Queue などがある．
4. Wait Queue アクセス時の排他制御オーバヘッド
   事象待ちのために資源ごとに Wait Queue を設けることにより，アクセスの集中を可能な限

り分散している．しかし，CMP では同時アクセスの可能性があるため，Wait Queue へのアクセスには排他制御が必要になる．

5. Run Queue アクセス時の排他制御オーバヘッド
CMP では PE 毎に Run Queue が存在しており，通常自 PE の Run Queue のみをアクセスする．しかし，Wait Queue から Run Queue に挿入する際に自 PE ではなく他 PE の Run Queue にアクセスする可能性があり，Run Queue へのアクセスには排他制御が必要となる．

6. プロセッサ間通信のための割り込み
割り込み処理において，I/O 要求と結果取得を行った PE が異なる場合，Wait Queue から Run Queue に I/O 要求スレッドを挿入した後に，プロセッサ間通信によりスケジューラを呼ぶ．そのために，割り込みを I/O 要求を行った PE に発生させる．

CMP と Linux Kernel 2.6 では上記の問題点が生じるため，デバイスと PE を一対一対応させて，並列 I/O 処理を実現している．

3 Fuce アーキテクチャ

3.1 Fuce プロセッサ

Fuce プロセッサ [6][7] は継続モデルを基盤とした効率のよいマルチスレッド実行を実現するために設計されている．Fuce プロセッサは現在における半導体技術の進歩を鑑みて，複数の PE を 1 チップに集約した CMP として実装されている．Fuce プロセッサにおける細粒度スレッドは “走り切り” であり，走行中は他からのいかなる干渉も受付けない．スレッドは後続のスレッドに計算結果を渡し終えた時に終了する．図 3 に Fuce プロセッサの概要を示す．

Thread Execution Unit

各 Thread Execution Unit (TEU) は，演算命令に継続を実現するスレッド制御命令を追加した単純な RISC 型の演算ユニットである Main Unit と，ロード命令のみを実行する Preload Unit の対で構成する．

Thread Activation Controller

Thread Actitvation Controller (TAC) は継続概念を実現する Fuce プロセッサの核となる機構である．TAC は スレッドの情報を保持する Activation Control Memory (ACM) と実行可能なスレッドを保持する Ready Thread Queue (RTQ) で構成される．ACM はスレッドに関する情報を保持する．図 4 に ACM の内部構造を示す．base-address とは関数インスタンスが利用するメモリ領域 (データエリア) へのポインタである．fan-in はスレッドが他のスレッドから受付ける継続数の初期値であり，sync-count はスレッドの現在の残りの継続数である．code-entry はスレッドの開始アドレスである．sync-count が 0 になった時，このスレッドは実行可能となり RTQ に投入される．あるスレッドが実行完了すると，待機中の実行可能スレッドが TEU に割り当てられる．

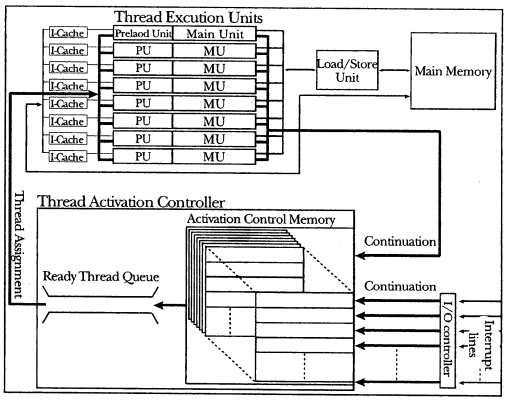

図 3: Overview of Fuce Processor.

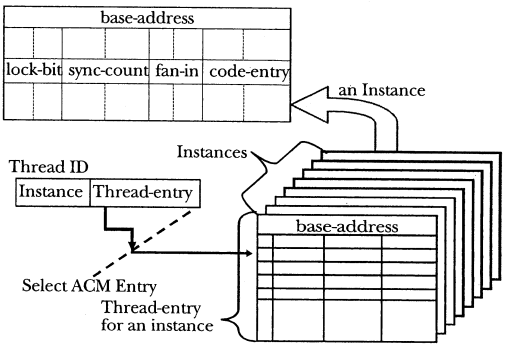

図 4: Overview of Activation Control Memory.

I/O Controller

Fuce プロセッサでは I/O Controller が割り込み信号を継続に変換する．継続先は対応するハンドラスレッド (従来の CMP における割り込みハンドラ) である．ハンドラスレッドは他のスレッドと同様に fan-in や base-address などを持つ．ハンドラスレッドは ACM 内部の特別な場所に登録され，I/O Controller と直結される．これにより，I/O Controller はハンドラスレッドへ継続可能となる．

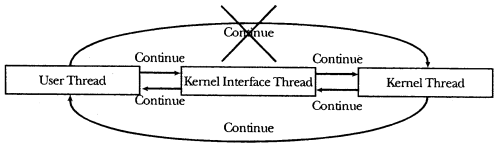

図 5: Thread Mode Change with Continuation.

3.2 Fuce-OS

Fuce-OS は，割り込み処理を含めた全プログラムを細粒度走り切りスレッドで構成し，通信処理と通常処理の融合を図る OS である．そのため，Fuce プロセッサと密に連携し，従来の OS と全く異なったマルチスレッド実行モデルである継続モデルを採用する．それにより，TAC によるスレッドスケジューラの H/W 化が行え，高速なスケジューリングを実現する．

Fuce-OS は細粒度走り切りスレッド構成により OS に内在するスレッドレベル並列性を積極的に利用する．また，I/O 処理等の遅延の大きな処理では，スプリットフェーズ方式の利用により，処理要求スレッドと結果受取りスレッドを分割し，結果受取り遅延の効果的な隠蔽を図る．

3.3 走行モード遷移

従来のプロセッサでは，ユーザモードからスーパバイザモードへ走行モードを遷移させるために特殊な命令を実行する．しかし，Fuce アーキテクチャでは全ての走り切りスレッドを継続で制御するため，継続により走行モード遷移を実現する．

Fuce アーキテクチャの走行モードは三種類あり，スレッドは User Mode，Kernel Mode と Kernel Interface Mode のいずれかのモードで走行する．それぞれのモードで走行するスレッドを User Thread，Kernel Thread，Kernel Interface Thread と呼ぶ．User Thread は従来のユーザモードである User Mode で走行し，Kernel Thread はスーパバイザモードである Kernel Mode で走行する．Kernel Interface Thread は User Thread から Kernel Thread への走行モード遷移を補助するために用意され，Kernel Interface Mode で走行し，User Thread から継続を受け Kernel Thread へ継続する．図 5 にそれぞれのモードで走行するスレッドの継続可能関係を示す．

カーネル空間をユーザ空間のアクセスから保護するため，User Thread は Kernel Thread へ直接継続してシステムコール発行を依頼することができない．そのため，User Thread から Kernel Interface Thread への継続により，システムコール発行の依頼を実現する．また，Kernel Thread と Kernel Interface Thread はあらゆるスレッドに対して継続できる．Fuce-OS では，Kernel Interface Thread の導入によりカーネル空間の保護を実現する．

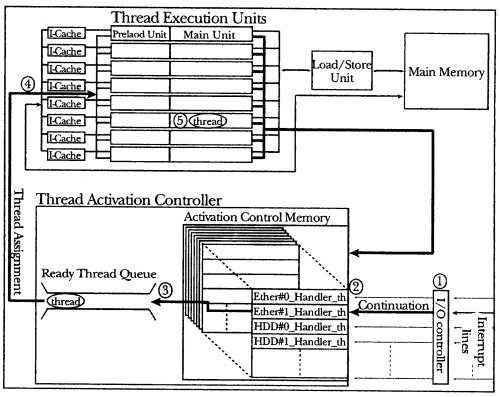

図 6: Interruption Exchanged for Continuation.

4 継続概念による割り込みなし並列 I/O 処理

4.1 割り込みの継続化

Fuce プロセッサは I/O が完了したという割り込み信号 (外部イベント) を受けると，継続により外部イベントをハンドラスレッドに通知する．そのため，継続モデルでは “割り込み” は発生せず，継続されたハンドラスレッドは実行中のスレッドが終了するまで待つ．以下，図 6 に示す Fuce プロセッサでの割り込みの継続化について述べる．

① 割り込み信号を受信した I/O Controller は，その信号を対応するハンドラスレッドへの継続に変換する．

② I/O Controller はハンドラスレッドへ継続を行うと，同一デバイスからの割り込み信号の多重受信を防ぐために対応する外部イベントの受付を禁止する．

③ 実行可能となったハンドラスレッドは RTQ へ投入され，TEU が空くのを待つ．(ある TEU で次のスレッドが実行可能になることを待つ)

④ TEU が空いたら，RTQ からハンドラスレッドを取出し，その TEU へ割当て，実行開始させる．

⑤ 実行開始直後，ハンドラスレッドはデバイスからデータを受取るか DMA 転送されたメモリ上にあるデータを読み込む．その後，対応する外部イベントの受付禁止を解除した後に I/O 処理を開始する．

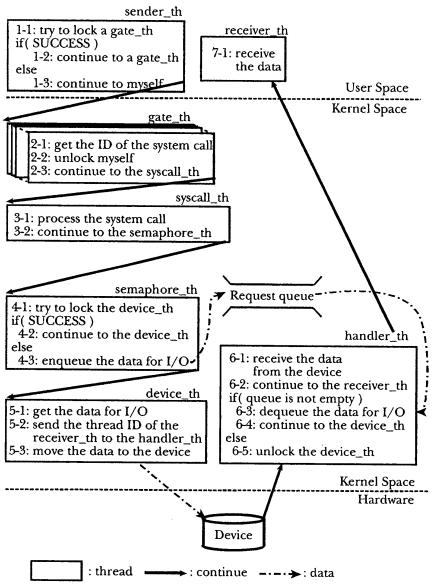

図 7: Fine-grain MultiThreading Fuce-OS.

4.2 Fuce-OS の細粒度マルチスレッド化

User Thread が Kernel Interface Thread に対してシステムコール発行を依頼し，システムコールがデバイスに対して I/O 要求を行う．その後，Fuce-OS はデバイスからデータを受取り，必要な処理を施した後に，ユーザ空間のスレッドにそのデータを渡す．以下，Fuce-OS のシステムコール処理と I/O 処理で使用される各スレッドの処理内容を説明し，外部イベントの通知である継続を用いた I/O 処理を述べる．図 7 に Fuce-OS の細粒度マルチスレッド構成を示す．

sender_th

sender_th はシステムコール発行を要求する．その要求を Kernel Interface Thread のモードで走行する gate_th に渡すために，ロック操作を試みる **(1-1)**．成功した場合，システムコール ID，システムコールに必要な引数と結果を受取る receiver_th の情報 (スレッド ID とデータ領域) を gate_th に渡し，継続する **(1-2)**．失敗した場合，再度ロックを試みるため自身に対して継続する **(1-3)**．

gate_th

システムコール ID から該当する syscall_th のスレッド ID を求め **(2-1)**，自身のロックを解除する **(2-2)**．求めたスレッド ID を用いて，必要な引数と receiver_th の情報を syscall_th に渡し，継続する **(2-3)**．

syscall_th

システムコール本体の処理を行い **(3-1)**，I/O 処理に必要なデータを semaphore_th に渡し，継続する **(3-2)**．

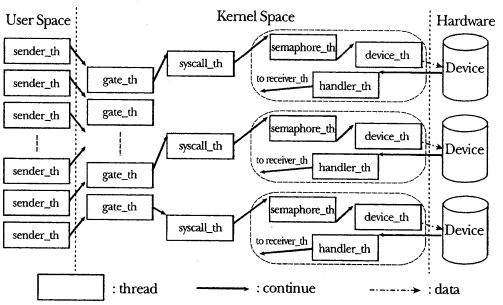

図 8: Parallel I/O Processing on Fuce Architecture.

semaphore_th

device_th に対してロック操作を試みる **(4-1)**．成功した場合，I/O 処理に必要なデータを device_th に渡し，継続する **(4-2)**．失敗した場合，そのデータを queue に挿入し，I/O 処理が完了するのを待つ **(4-3)**．

device_th

I/O 処理に必要なデータを受取り **(5-1)**，receiver_th の情報を handler_th に渡す **(5-2)**．その後，デバイスに対して I/O 発行を行う **(5-3)**．

handler_th

デバイスから継続され TEU が空くと開始する．I/O 処理結果を受取り **(6-1)**，その結果を receiver_th に渡し，継続する **(6-2)**．queue にデータがあれば，一つ取り出す **(6-3)**．そのデータを device_th に渡し，継続する **(6-4)**．queue が空ならば，device_th のロックを解除する **(6-5)**．

receiver_th

I/O 処理結果を Fuce-OS から受取る **(7-1)**．

4.3 継続概念による並列 I/O 処理

Fuce アーキテクチャでは，複数デバイスへの I/O 要求や I/O 処理を効果的に行う．図 8 に継続概念による割り込みなし並列 I/O 処理を示す．

Fuce プロセッサでは割り込みの継続化により，複数の異なる外部イベントが同時に発生する場合，4.1 節と同様に，対応するハンドラスレッドが空き TEU で並列に走行できる．従来のモデルでは S/W で構成された Run Queue が PE 毎に複数本あるが，Fuce プロセッサではスケジューリングを行う TAC に H/W で構成された RTQ が一本のみある．その結果，従来のモデルと比較して，RTQ を備えた TAC は非常に高速なスケジューリングを実現し，I/O 要求と処理結果の取得を行う TEU が異なっても，I/O 要求スレッドの起床のためにスケジューラを呼ぶ必要がな

い．そのため，あるTEUが空けば，ハンドラスレッドは即座に実行を開始する．

そのため，Fuceプロセッサは TEU とデバイスを多対多対応し，I/O処理を任意の TEU で処理する．

5 期待される効果と問題点

2.2節で述べた問題点から Fuce アーキテクチャでの期待される効果を以下に述べる．

1. 割り込みによる走行中の処理の退避・復帰

Fuce アーキテクチャでは，全てのスレッドは“走り切り”なので，退避・復帰処理を必要としない．

2. 割り込み許可フラグへの排他制御オーバヘッド

Fuce アーキテクチャでは，割り込み信号 (外部イベント) に対して一つの割り込みハンドラしか処理を開始しない．そのため，割り込み許可フラグへの排他制御オーバヘッドは生じない．

3. Wait Queue アクセス時の排他制御オーバヘッド

Fuce アーキテクチャでは事象待ちについてスプリットフェーズ方式を採用しているので，事象待ちのための Wait Queue が必要ない．そのため，Wait Queue アクセス時の排他制御オーバヘッドは生じない．

4. Run Queue アクセス時の排他制御オーバヘッド

Fuce アーキテクチャでは，スレッドを制御する命令によって H/W の RTQ を操作する．その命令の遅延はほぼ無いため，Run Queue アクセス時の排他制御オーバヘッドは生じない．

5. プロセッサ間通信のための割り込み

Fuce アーキテクチャでは，スレッドを制御する命令によってスレッドを実行可能にする．そのため，プロセッサ間通信のための割り込みは生じない．

次に Fuce アーキテクチャで生じる問題点を述べる．

Fuce アーキテクチャでは，スレッドスケジューリングを行う際に，実行可能なスレッド情報を保つ RTQ が一本のみである．そのため，全スレッドの優先度が同一である．結果として，Kernel Thread や Kernel Interface Thread のモードで走行するスレッドが優先実行されず，I/O処理の応答性が低い．

なお，Linux Kernel 2.6 と同様に Fuce アーキテクチャにおいても，複数のシステムコールが同一資源に対してアクセスする可能性があり，各資源の獲得時の排他制御オーバヘッドが生じる．

6 評価

継続概念による効果的な多種 I/O 処理の分散について評価する．評価には，VHDL で記述した Fuce プロセッサをソフトウェア ModelSim 上でシミュレートし，I/O処理とシステムコール処理をアセンブリ言語で記述した Fuce-OS を実行する．表1に実験環境を示す．

表 1: Experimental Environment.

Number of TEUs	1, 2, 3
Data Cache	N/A
Instruction Cache	4KB / TEU
Throughput of TAC	1 clock cycle
Thread Assignment from RTQ	1 clock cycle
Number of Devices	1, 2, 3

10Gb Ethernet と HDD を基に，シミュレートするデバイス遅延について次のように想定する．10Gb Ethernet では，現実的にはデータパケットが 100μsec 前後でプロセッサに到着すると予想される．また，HDD では 10msec 前後でデータをやり取りできる．本実験では，複数の HDD を RAID 構成で実装したと仮定し，2msec の遅延でデータを得ることができる．ここでは，Fuce プロセッサが 1GHz で動作すると仮定して，シミュレータでの 1 クロックあたりを 100nsec と換算して実験を行う．

本実験では以下の方法により，測定時間を 40msec とした場合の実行可能な最大システムコール数を測定する．

1. N 個のデバイス遅延 100μsec のシステムコール要求 (sender_th) と M 個のデバイス遅延 2msec のシステムコール要求を TEU に投入する．

2. M の値が常に最大になるように N の値を変化させ，単位時間内に全システムコールを処理できる N の最大値を求める．N が求めた値より大きい場合は，システムコール要求数が Fuce プロセッサの I/O 処理性能を超え，その要求が RTQ 内に留まる．

3. デバイス遅延 100μsec のデバイスを 1 個に固定し，デバイス遅延 2msec のデバイス数と TEU 数の組を (1, 1)，(2, 2)，(3, 3) と変化させて 1，2 の測定を行い，全最大システムコール数を求める．

6.1 効果的な多種 I/O 処理の分散

図 9 の棒グラフに TEU 数と遅延 2msec のデバイス数の各組での最大システムコール数を，折線グラフに TEU を 1 本と遅延 2msec のデバイス数を 1 個の組を基準とした最大システムコール数の比を示す．

遅延 2msec のデバイスでは，常にシステムコール数が最大となるように設定し実験している．そのため，TEU 数と遅延 2msec のデバイス数の組が (1, 1)

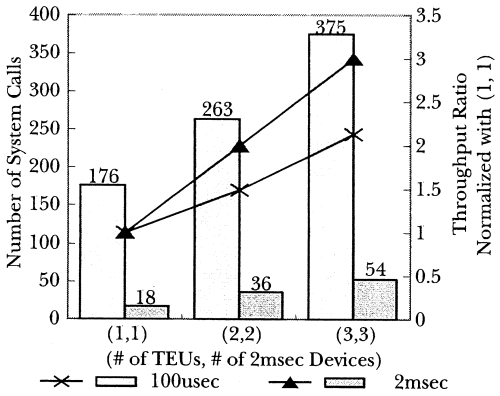

図 9: Distribution of Kinds I/O Processing.

から (3, 3) と増加したときに，最大システムコール数は 18 個から 54 個と 3 倍に増加している．また，デバイス遅延 100μsec では 176 個から 375 個と 2.13 倍に増加している．

これは，遅延 2msec のシステムコールが I/O 要求を行うと，デバイス遅延を待っている間 TEU が空き，他の I/O 処理が可能となる．その空いている TEU でデバイス遅延 100μsec の I/O 処理が処理できたためである．さらに，TEU 数と遅延 2msec のデバイス数の増加にしたがい，デバイス遅延を待っている TEU が増加するので，遅延 100μsec の I/O 処理が効果的に分配し処理できた．

また，従来の並列 I/O 処理ではデバイスと TEU を一対一対応させるため，TEU 数と遅延 2msec のデバイス数の組が (1, 1) から (3, 3) と増加したときに，遅延 100μsec ではシステムコール数が増加しないと考えられる．これに対して，Fuce アーキテクチャでの並列 I/O 処理ではデバイスと TEU を多対多対応できるため，従来の並列 I/O 処理比べて効果的な I/O 処理が行える．

7 おわりに

本稿では，継続概念を基盤とした Fuce プロセッサと Fuce-OS が協調して行う割り込みなし並列 I/O 処理について述べた。割り込みを利用する従来の I/O 処理を並列化する場合，プロセッサ間通信が発生し，スケジューリングのコストが大きい．そこで，Fuce プロセッサでは割り込みの継続化，スレッドスケジューラの H/W 化を行い，Fuce-OS ではシステムコール部と I/O 処理部の細粒度マルチスレッド化を行った．その結果，Fuce アーキテクチャでは効率的な並列 I/O 処理の実現が可能となった．

これからもネットワークはさらに低遅延化かつ高速化し，プロセッサと OS が協調する I/O 処理が重要となる．また，高性能なサーバでは複数の NIC を用いた構成が考えられ，今回提案した継続概念による並列 I/O 処理モデルは非常に有用な手段となる．

今後はさらに効果的な並列 I/O 処理の実現のために，Fuce プロセッサは OS のスレッド群を優先実行する Priority Ready Thread Queue を搭載させる．

現在，Fuce プロセッサを FPGA 上で実現しているが，I/O 処理に関する機能を有しない．提案モデルの有効性について詳細な評価を行うために，I/O 処理の機能を拡張した Fuce プロセッサを FPGA 上に実現していく．

謝辞

岡山大学大学院自然科学研究科 乃村 能成氏に，提案モデルについて多くの助言を頂きました．深く感謝いたします．

参考文献

[1] Makoto Amamiya, A New Parallel Graph Reduction Model and its Machine Architecture, Data Flow Computing, Theory and Practice Ablex Publishing Corp., pp.445-464, 1991.

[2] Makoto Amamiya, Hideo Taniguchi and Takanori Matsuzaki, An Architecture of Fusing Communication and Execution for Global Distributed Processing., Parallel Processing Letters, Vol.11, No.1, pp.7-24, 2001.

[3] D. W. Wall, Limits of Instruction-Level Parallelism, Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS), Vol.26, No.4, pp.176-189, 1991, ACM Press.

[4] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen and Kunle Olukotun, The Stanford Hydra CMP, IEEE Micro, Vol.20, No.2, pp.71-84, 2000.

[5] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz and S. K. Reinhardt, Analyzing NIC Overheads in Network-Intensive Workloads, In 8th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-8), pp.32-39, 2005.

[6] Takanori Matsuzaki, Hiroshi Tomiyasu and Makoto Amamiya, Basic Mechanisms of Thread Control for On-Chip-Memory Multi-threading Processor, Proceedings of the Fifth Workshop on Multithreaded Execution, Architecture and Compilation (MTEAC-5), pp.43-50, 2001.

[7] Takanori Matsuzaki, Satoshi Amamiya, Masaaki Izumi and Makoto Amamiya, A Multi-thread Processor Architecture Based on the Continuation Model, Proc. of 8th Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'05), pp.83-90, 2006.