継続モデルによるH/W・S/W協調並列I/O処理モデルの提案

泉, 雅昭
九州大学大学院システム情報科学府知能システム学専攻 : 博士後期課程

青野, 光洋
九州大学大学院システム情報科学府情報工学専攻 : 修士課程

雨宮, 聡史
九州大学大学院システム情報科学府知能システム学専攻 : 博士後期課程

松崎, 隆哲
近畿大学産業理工学部

他

https://doi.org/10.15017/1517817

出版情報 : 九州大学大学院システム情報科学紀要. 12 (1), pp.41-47, 2007-03-26. Faculty of Information Science and Electrical Engineering, Kyushu University
バージョン :
権利関係 :

九州大学大学院
システム情報科学紀要
第12巻 第1号 平成19年3月

Research Reports on Information Science and
Electrical Engineering of Kyushu University
Vol.12, No.1, March 2007

継続モデルによるH/W・S/W協調並列I/O処理モデルの提案

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†††

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

Masaaki IZUMI, Mitsuhiro AONO, Satoshi AMAMIYA, Takanori MATSUZAKI,
Shigeru KUSAKABE, Hideo TANIGUCHI, Ryuzo HASEGAWA and Makoto AMAMIYA

(Received December 15, 2006)

Abstract: We propose the Fuce architecture based on dataflow computing model. The goal of this architecture is fusion of communication and execution. The multiple thread execution model of the architecture takes the continuation-based multithreading model that manages dependency among fine-grain "uninterruptible" threads. In this paper, the continuation-based multithreading model constructs more resourceful parallel I/O processing cooperated with by processor and operating system. Our model is different from the conventional I/O processing model that handles "interrupt." We illustrate the continuation-based parallel I/O processing model.

Keywords: Fuce architecture, Fine-grain multithreading, Chip multiprocessor, Parallel I/O processing, Interrupt-less

1. はじめに

High Performance Computing (HPC) では数万個のプロセッサを搭載し，数十TFLOPSの性能を示すスーパコンピュータが開発されている．HPCではプロセッサ間や計算機ノード間の通信がシステム全体の性能を左右する大きな要因の一つとなるため，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，高速なI/OバスとしてHyperTransport, PCI ExpressやFSB1333が提案・利用され，高速なネットワークとしてInfiniBand QDR, 10Gb EthernetやMyrinetが利用されている．さらに複数の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の開発や，豊富なトランジスタ資源により同一チップ内にプロセッサと共にNICを搭載し高いI/O処理性能を達成することをシミュレーションにより予測した研究[6]などが行われている．

I/Oを扱う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)[5) 7)]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[8)]やIBM POWER5[4)]などが登場するに至った．上記のCMPでは，I/Oなどを扱う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散機構を利用する．その機構により，I/O処理をCMP内の任意のPEで実行可能にし，OSにおいても任意のPEでI/O処理を扱えるようにした．

しかし，I/O処理を行うPEを固定しない場合は，固定する場合と比較してキャッシュの乱れや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．

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について考慮すると，毎秒最大約80万個のデータパケットがCMPに届き，データパケットと同数から数分の1の応答パケットもさらに届く．このとき，I/O処理を行うPEを固定する場合，CMP内に複数のPEが存在しているにもかかわらず，単体PEのI/O処理性能によりCMP全体のI/O処理性能は制限される．

そこで，我々はプログラムや割り込みなどを扱うOSを

平成18年12月15日受付
* 知能システム学専攻博士後期課程
** 情報工学専攻修士課程
*** 近畿大学産業理工学部
† 情報工学部門
†† 岡山大学大学院自然科学研究科
††† 知能システム学部門

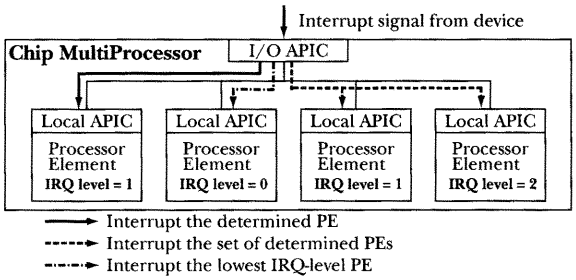

Fig. 1 Distribution of Interrupt Signal on x86 CMP with APICs.

含めて，全ての処理を“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に採用する．Fuceアーキテクチャは“Fuce”の名が示すとおり，通信処理と通常処理を融合することを目的とする．

本稿では，最初に従来の並列I/O処理について述べて，Fuceアーキテクチャにおける従来とは全く異なるH/W・S/W並列I/O処理を提案する．

2. 従来の並列I/O処理

プロセッサとOSの組み合わせにより，多数のI/O処理が存在する．本稿では従来のI/O処理として，x86プロセッサアーキテクチャのCMP (以後，x86 CMPと述べる) の割り込み発生機構と，OS (Linux Kernel 2.6) の割り込み処理を用いて議論する．その後，従来の並列I/O処理について述べる．

2.1 x86 Chip MultiProcessorの割り込み発生機構

x86 CMPはAdvanced Programmable Interrupt Controller (APIC) を用いて，割り込みをOSに通知する．APICには，外部から割り込み信号を受信しCMP内のPEに送信するI/O APICと，I/O APICから割り込み信号を受信しPEに割り込みを発生させるLocal APICの二種類がある．I/O APICは，(割り込みの優先度を示す) Interrupt Request (IRQ) レベルが最も低いPE，特定の単一PE，特定の複数PEというPE指定方法で割り込みを発生させる．**Fig. 1**に割り込み発生時のPE指定方法を示す．

特定の単一PE

割り込みを利用した従来のI/O処理では，スレッドスケジューリングのオーバヘッドを削減し，キャッシュを効果的に利用するため，I/O APICが割り込みを特定の単一PEに発生させる．しかし，10Gb Ethernetでは毎秒最大約80万データパケットがプロセッサに届く．そのため，単体PEのI/O処理性能が高速なI/Oに追いつかず，単体PEがCMP全体のI/O処理性能を制限する．その結果，I/Oが高速化するに従って，応答性の確保が困難になる．

最低IRQレベルのPE

I/O APICは最低IRQレベルのPEに割り込みを発生させると，割込み処理を行うOSの割り込みハンドラは実行を開始する．しかし，I/O要求と処理結果の取得を行ったPEが異なる場合，PE毎に搭載する命令キャッシュと命令TLBにヒットしない．

特定の複数PE

この方式ではI/O APICは割り込みを複数のPEに同時に発生させる．それぞれのPEで割り込みハンドラが実行開始を試みるが，実際は実行する割り込みハンドラが一つのPEに限定される．実現方法として以下の手法で，test&setによる割り込み用のフラグ参照を試みる．

- 最初にフラグを参照したとき，割り込みハンドラはフラグを変更し，実行開始する．
- フラグを参照し，他のPEで割り込みハンドラが走行中であることを知ると割り込み処理を終了する．

割り込みハンドラを実行できなかったPEでは，直前の処理を一時中断し次の処理を再開するまでの時間と，命令キャッシュが乱れるペナルティを負う．さらに，I/O要求と処理結果の取得を行ったPEが異なる場合は，最低IRQレベルのPEの場合と同様に命令キャッシュと命令TLBにヒットしない．

2.2 Linux Kernel 2.6の割り込み処理

Linux Kernel 2.6は割り込みをCMPの各PEに分散させるirqbalanceを用いて，割り込み処理を負荷分散できる．また，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つなぐWait Queueを資源毎に持つ．そのため，I/O要求と結果取得を行ったPEが同一である場合と異なる場合では，スレッドスケジューリングのコストが異なる．

I/O要求と結果取得を行ったPEが同一である場合

割り込み処理終了直前，割り込みハンドラは外部デバイスから受取ったデータをI/O要求スレッドへ送ることを試みる．I/O要求スレッドは自身をWait QueueにつなぎI/O完了を待っている．そのため，割り込みハンドラ(と一連のカーネル内ルーチン)は指定されたユーザ空間のアドレスにデータをコピーした後，Wait Queueにつながっている対応するI/O要求スレッドをRUNNING状態にしてRun Queueにつなぎ，スケジューラを呼ぶ．

I/O要求と結果取得を行ったPEが異なる場合

I/O要求と結果取得を行ったPEが同一である場合と同様に，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

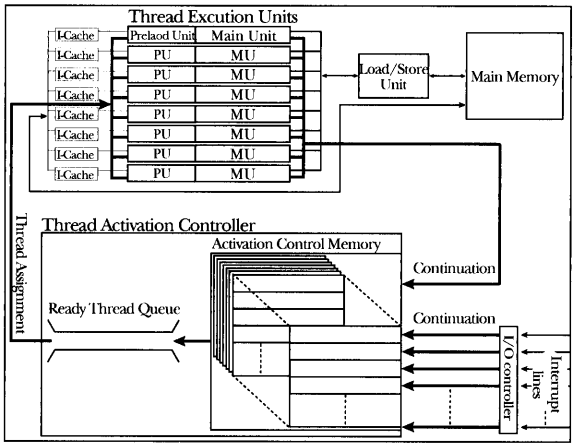

Fig. 2 Overview of Fuce Processor.

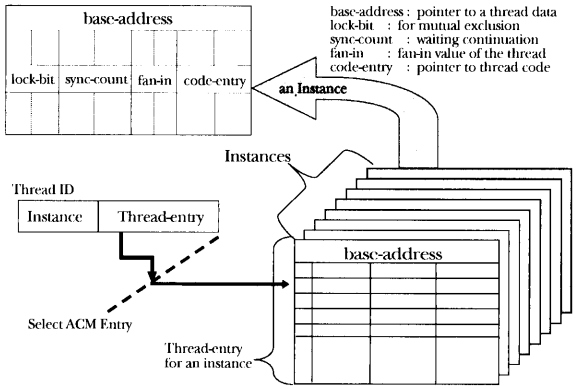

Fig. 3 Overview of Activation Control Memory.

なるため，割り込みハンドラがI/O要求を行ったPEのRun Queueをロックし，Wait Queueにつながっている対応するI/O要求スレッドをRUNNING状態にしてそのRun Queueにつなぐ．その後，I/O完了を通知するためにI/O要求を行ったPEに対してスケジューリングの実行を試みる．割り込みハンドラはスレッドスケジューリングの実行をそのPEに対して依頼するため，必要ならばプロセッサ間通信をそのPEに発生させる．スケジューリングによりI/O要求スレッドは実行の機会を得て，起床し，割り込みハンドラがユーザ空間のアドレスにコピーしたデータを受取る．

2.3 従来の並列I/O処理

x86 CMPの割り込み発生機構とLinux Kernel 2.6の割り込み処理を念頭におき，従来の並列I/O処理について述べる．複数の異なる割り込みが発生し，割り込み毎にI/O要求と結果取得を行ったPEが異なる場合，Linux Kernel 2.6はスレッドスケジューリングの実行を他のPEに対して依頼するため，必要ならば他のPEへのプロセッサ間通信を発生させる．その結果，プロセッサ間通信とスレッドスケジューリングが割り込み毎に実行される．

3. Fuce アーキテクチャ

3.1 Fuce プロセッサ

Fuce プロセッサ[9)10)]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．**Fig. 2**にFuce プロセッサの概要を示す．

Thread Execution Unit

各Thread Execution Unit (TEU) は，通常の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非常に単純な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．これらユニットは2つのレジスタファイルをダブルバッファ方式で用いることにより，ロード命令で発生するデータアクセスを隠蔽し，できるかぎりパイプラインストールを起こさずに動作するように設計されている．これに伴い，スレッドは２つの部分から成り，先頭部分はロード命令列のみで，本体となる後半部分は任意の命令列となる．

Thread Activation Controller

Thread Actitvation Controller (TAC) は継続モデルを実現する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．Fuce プロセッサにおけるプログラムは複数の関数群として定義され，実行時には高速なメモリで構成するACMに関数インスタンスとして割り当てられる．各関数インスタンスはその内部構造にいくつかのスレッドの情報を含んでおり，ACMはこれらのスレッドに関する情報を保持する．**Fig. 3**にACMの内部構造を示す．ACM内部の各ページはそれぞれ関数インスタンスに一対一対応し，各スレッドも関数インスタンスのページ番号と任意のスレッドエントリの組と一対一対応する．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) 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．

I/O Controller

Fuce プロセッサではI/O Controllerが割り込み信号を

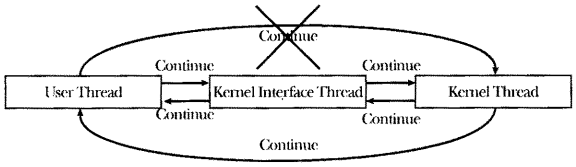

Fig. 4　Thread Mode Change with Continuation.

継続に変換する．継続先は対応するハンドラスレッド(x86 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内部の特別なページに登録され，そのページは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り，スレッドIDでデバイスと，code-entryでハンドラスレッドを関連付ける．

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．例えば，TACによるスレッドスケジューラのH/W化により，高速なスレッドスケジューリングを実現している．

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，遅延の大きな処理，特にI/O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

x86 プロセッサアーキテクチャでは，ユーザモードからスーパバイザモードへの走行モード遷移においてint 0x80やsysenter命令を用いる．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Kernel Interface Modeで走行し，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，User Threadから継続を受けKernel Threadへ継続する．**Fig. 4**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．

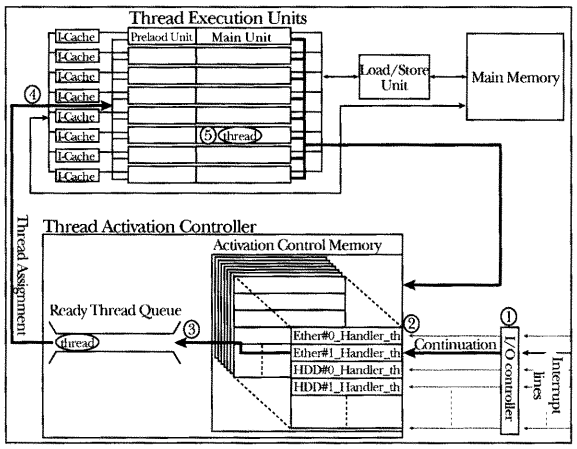

Fig. 5　Interruption Exchanged for Continuation.

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へ直接継続できない．しかし，User Threadは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．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．

4. 継続モデルによるH/W・S/W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，継続を用いて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現在実行中のスレッドが終了するまで待つ．以下，割り込みの継続化について述べる．**Fig. 5**にFuce プロセッサでの割り込みの継続化を示す．

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．

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り，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．

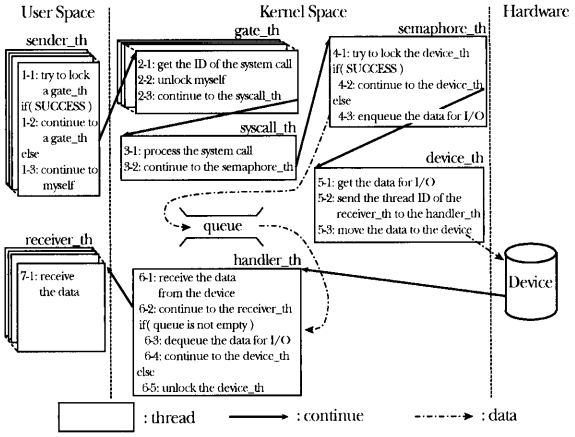

Fig. 6 Fine-grain MultiThreading Fuce-OS.

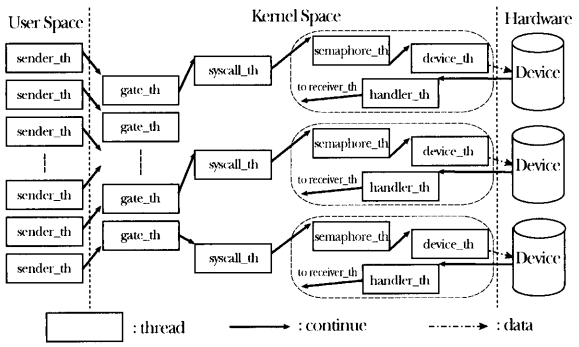

Fig. 7 Parallel I/O Processing on Fuce Architecture.

4.2 Fuce-OSの細粒度マルチスレッド化

User Threadがシステムコール発行をKernel Interface Threadに対して依頼し，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処理での各スレッドの処理内容を説明し，Fuce プロセッサからの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．**Fig. 6**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である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受取りスレッドである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)**．

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につなぎ，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 H/W・S/W協調並列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，非常に容易にかつ効果的に行う．**Fig. 7**にH/W・S/W協調並列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で並列に走行する．従来のモデルでは，Run QueueがPE毎にありS/Wで構成され，Fuce プロセッサではRTQが一本のみでありH/Wで構成される．その結果，従来のモデルと比較して，RTQを備えたTACは非常に高速なスレッドスケジューリングを実現し，I/O要求を行うTEUと処理結果の取得を行うTEUを同一にする必要がない．そのため，I/O要求を行ったTEUに限らずTEUが空いたら，ハンドラスレッドは即座に実行する．

従来の並列I/O処理ではI/O完了を通知するため，Wait Queueにつながっているスレッドも含めて全てを起床させるが，Fuce-OSではWait Queueにつなぐ必要がない．ハンドラスレッドは継続先であるスレッドIDを保持するだけで結果受取りスレッドを直接起床できる．

そのため，Fuce アーキテクチャではTEUとI/O処理を多対多として対応付け，I/O処理を固定せずに一つのTEUで処理する．それにより，従来の並列I/O処理に比べFuce アーキテクチャによる並列I/O処理性能のスケーラビリティは高い．

Table 1 Experimental Environment.

Number of TEUs	1, 2, 3, 4
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

5. 評　　価

Fuce アーキテクチャによる，複数I/O処理の効率化と，並列I/O処理性能のスケーラビリティについて評価する．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．**Table 1**に実験環境を示す．

ソフトウェアシミュレータでは実デバイスによる評価ができないため，実デバイスをシミュレートするvirtual_hw_thをTEUで実験中常に実行し続けることで評価を行う．実装した処理ではdevice_thが実デバイスにI/O発行を行うが，ここではdevice_thがvirtual_hw_thにメモリを経由してデータを送る．データを受取ったvirtual_hw_thは，ループ処理によりI/Oデバイス遅延をシミュレートする．次に，virtual_hw_thはhandler_thへの継続により実デバイスをシミュレートする．そのため，Fuce プロセッサは最大8本のTEUを利用できるが，virtual_hw_thを常に実行する複数本のTEUと，単位時間測定用に一本のTEUが必要なため，本実験では最大TEU本数を4本，最大Device数を3個とした．

10Gb Ethernetを基に，シミュレートするI/Oデバイス遅延について考慮する．パケットサイズを最大の1.5KBに固定した場合，データパケットが1〜2μsecという最短間隔でプロセッサに到着できる．仮に，Fuce プロセッサが1GHzで動作すると，データパケットの到着間隔は1〜2Kクロックとなるため，シミュレートするI/Oデバイス遅延を2，4，6Kクロックとする．

また，ネットワークの負荷が大きい高性能サーバでは，専用H/WでTCP/IP処理を全て行うTCP/IP Offload Engine (TOE) を備えた高性能なNICを利用する．本実験においてFuce プロセッサは，TOEを備えたNICを利用すると仮定して，割り込みの継続化，スレッドスケジューラのH/W化とOSの細粒度マルチスレッド化の効果に焦点を当てる．本実験では以下の方法により，単位時間を100Kクロック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．この最大システムコール数をFuce アーキテクチャにおける最大スループットとする．

1. N個のシステムコール要求(sender_th) をTEUに投入する．
2. 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，システムコール要求がRTQ内に留まる．
3. デバイス数とTEU数を変化させて1，2の測定を行い，全ての最大システムコール数を求める．

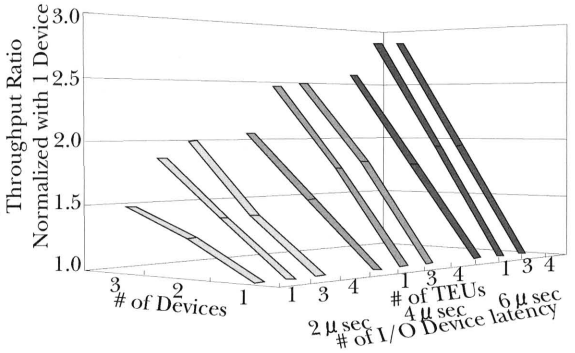

Fig. 8　Efficiency on I/O Processing.

5.1 複数I/O処理の効率化

Fig. 8に，各TEU本数のデバイス数1個を基準としたシステムコールのスループット比を示す．同一I/Oデバイス遅延についてTEU数を増加させると全ての場合において，システムコールのスループット比の向上率が増加している．これは，Fuce アーキテクチャの割り込みの継続化とスレッドスケジューラのH/W化によりI/O要求を行ったTEUに限らず空いたTEUに対して即座にI/Oを処理できるからである．さらに，従来のI/O処理とは異なり，処理結果をユーザスレッドへ渡す際に必要なスレッドスケジューリングも非常に小さなコストで実行できるからでもある．

また，I/Oデバイス遅延が増加するにつれて，スループット比の向上率が増加している．これは，遅延が小さい場合では，CMP全体のI/O処理能力の要因がCPUボトルネックであるため，デバイス数を増加してもスループット向上が困難である．これに対して，遅延が大きい場合は，CMP全体のI/O処理能力の要因がI/Oボトルネックであるため，デバイス数が増加するとスループット向上が顕著に得られる．

5.2 並列I/O処理のスケーラビリティ

Fig. 9に全てのI/Oデバイス遅延におけるTEU本数とデバイス数の各組での最大スループットと，TEUを1本とデバイス数を1個の組を基準としたスループット比を示す．

全てのI/Oデバイス遅延について，最大スループットは増加している．これは，TEUとI/O処理を多対多として

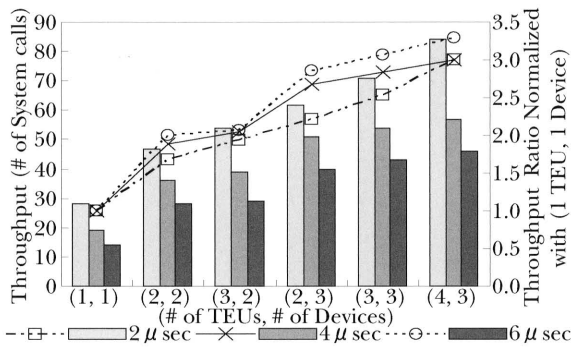

Fig. 9 Scalability on Parallel I/O Processing.

対応付け，I/O処理が空いたTEUで効率的に処理でき，並列I/O処理性能が向上したためである．また，従来の並列I/O処理ではデバイスとTEUを一対一として対応させて，全てのデバイスが同一のI/Oデバイス遅延である場合に，TEU数とデバイス数の組が(1，1)から(3，3)と増加したときに約3倍のスループット比を示すと考えられる．これに対して，Fuce アーキテクチャによる並列I/O処理ではデバイスとTEUを多対多として対応させてI/O処理を行うため，I/Oデバイス遅延が6μsecでは(3，3)で3.07倍のスループット比を示し，2μsecでは2.54倍を示す．

6. おわりに

本稿では，Fuce アーキテクチャを基盤としたFuce プロセッサとFuce-OSが協調して行う並列I/O処理について述べた。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，スレッドスケジューリングのコストが大きい点を述べた．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では自身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非常に単純で効率的な並列I/O処理の実現が可能となった．

今後も，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の協調したI/O処理が重要となる．また，高性能なサーバやHPCにおける計算機では複数のNICを用いた構成が考えられ，今回提案した並列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処理を実現するために，Fuce プロセッサは，Kernel ThreadとKernel Interface Threadを優先的に実行させるPriority Ready Thread Queueをさらに搭載させる．Fuce-OSはLinux Kernel 2.6との詳細な比較のために，TCP/IPプロトコルスタックを含めた細粒度マルチスレッド構成による実装を進める．

現在，Fuce プロセッサをFPGAに設計しているが，I/O処理に関する機能を有していない．提案モデルの有効性について詳細な評価のために，Fuce プロセッサをI/O処理の機能に関して拡張したFPGAを設計していく．

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) R. Kalla, B. Sinharoy and J. M. Tendler, IBM power5 chip: a dual-core multithreaded processor, IEEE Micro, Vol.24, No.2, pp.40-47, 2004.
5) 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.
6) 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.
7) N. Nishi, et al., A 1GIPS 1W Single-Chip Tightly-Coupled Four-Way Multiprocessor with Architecture Support for Multiple Control Flow Execution, Proc.ISSCC2000, 2000.
8) Poonacha Kongetira, Kathirgamar Aingaran and Kunle Olukotun, Niagara: A 32-Way Multithreaded Sparc Processor, IEEE Micro, Vol.25, No.2, pp.21-29, 2005.
9) 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.
10) 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.