2006−OS−103　(4)
2006／7／31

Zero-Wait 方式による多数の I/O 要求に対する処理の Fuce 上での実現と評価

青野光洋[†1]　泉　雅昭[†1]　松崎隆哲[†2]
日下部　茂[†3]　乃村能成[†4]
谷口秀夫[†4]　雨宮真人[†3]

Zero-Wait 方式のスレッドは，実行開始後 wait なしで走り切るプログラム片であり，スレッド間の実行制御は継続概念によって行われる．この方式にもとづいて，OS 機能の一つである多重の I/O 要求に対する処理を Fuce プロセッサ上で実現する．Fuce プロセッサはデータフローの概念を基盤とし，Zero-Wait 方式と親和性の高いチップマルチプロセッサである．I/O 要求に対する処理は，OS 機能の中でもオーバヘッドが生じやすい処理の一つである．Zero-Wait 方式にもとづいてそのような処理を Fuce プロセッサ上で実現することにより，既存の方式と比較して直接的な継続点の起動によるスループット向上に加え，カーネル内処理の並列性を高めることが可能となる．本稿では，Zero-Wait 方式によって Fuce プロセッサ上で実現した処理について説明し，実測を用いた評価について述べる．

The Implementation and Evaluation of handling Many I/O Requests Based on Zero-Wait Threads for The Fuce Processor

Mitsuhiro Aono,[†1] Masaaki Izumi,[†1] Takanori Matsuzaki,[†2]
Shigeru Kusakabe,[†3] Yoshinari Nomura,[†4] Hideo Taniguchi[†4]
and Makoto Amamiya [†3]

Zero-Wait Thread is a continuation-based thread and supposed to run to completion without suspention. We have implemented handler routines for multiple I/O requests based on Zero-Wait Threads for the Fuce processor. The Fuce processor supports a continuation model which is a varient of dataflow computing model. Handling I/O requests causes high latencies in operating systems, implementation based on Zero-Wait Thread allows performance improvement and effective in threads parallel execution. This paper explains our impelementation and shows evaluation for effectiveness of Zero-Wait Thread on the Fuce processor.

1. はじめに

近年，ハードウェア技術の進歩により SMT(Simultanious MulitiThreading) や CMP(Chip MultiProcessor)[1] のように複数のスレッドを並列実行するハードウェアを安価に入手可能になってきた [2]．しかしながら，こういった並列実行可能なハードウェア上で動作するオペレーティングシステム (以下，OS と略す) であっても，逐次計算モデルから派生したプログラム実行モデルをそのまま利用している．そのため，ハードウェアレベルで利用可能な並列度が今後さらに高くなったとしても，複数スレッドの同時実行に伴う同期処理や複雑なスレッドスケジューリングが性能向上の妨げになるという問題点が考えられる．

一方，並列処理との親和性の高い計算モデルとしてデータフロー計算モデルがある．プロセッサアーキテクチャにおいてはデータフローの概念を取り入れた技術の提案がなされてきており，近年でもデータフロー方式のプロセッサが開発されたり，非データフロー方式のプロセッサに対しても高速化技術の一部として取り入れられたりしている [3][4]．一方，本質的に多重並行処理を扱う OS に対してはデータフローの概念が積極的に取り入れられているとは言いがたい．そこで，我々はデータフロー計算モデルの概念を取り入れ

†1 九州大学大学院システム情報科学府
Graduate School of Information Science and Electrical Engineering, Kyushu University
†2 近畿大学産業理工学部
School of Humanity-Oriented Science and Engineering, Kinki University
†3 九州大学大学院システム情報科学研究院
Graduate School of Information Science and Electrical Engineering, Kyushu University
†4 岡山大学大学院自然科学研究科
The Graduate School of Natural Science and Technology, Okayama University

た Zero-Wait 方式を実行モデルとする OS を提案し，その有効性に関する研究を行っている [5]．

Zero-Wait 方式では，スレッドは実行開始後 wait なしで走り切り，スレッド間の実行制御は継続概念によって制御されるという特徴を持つ．また複数のスレッドで構成されたプログラムは，スレッドをノードとし継続関係をエッジとするデータフローグラフを形成する．こういった特徴を持つ方式を利用することで，OS 内の効率の良い高多重処理の実現を目指す．

今回，実現例として OS 機能の一つである I/O 要求に対する処理について取り上げ，Fuce(Fusion Communicaion and Execution) プロセッサ [6] 上で実現した．Fuce プロセッサはデータフロー計算モデルを基盤とするチップマルチプロセッサである．

I/O 要求に対する処理は，OS 機能の中でもオーバヘッドが生じやすい処理の一つである．例えば汎用 OS における I/O 要求に対する処理では，I/O 要求スレッドを処理待ちの状態として wait キューを用いて管理する方式がある．この場合，I/O 要求スレッドが I/O 処理結果を受け取る際に，wait キューにより管理されている待ち状態の I/O 要求スレッドを再実行する必要がある．したがって，要求数が増えるとスレッドの起床処理のオーバヘッドが大きくなると考えられる．

一方 Zero-Wait 方式では，データフロー方式によりカーネル内処理の並列性を高めることが期待できる．また，継続概念によるスレッド間の実行制御を行うことにより，wait キューを用いずに処理結果受け取りを行うスレッドを直接起動することが可能となり，低オーバヘッドの実現が可能である．処理結果受け取りスレッドの直接起動によるオーバヘッドの削減効果は，直接起床方式と wait キューを用いた方式との比較で確認されている [5]．そこで，本稿ではカーネル内処理の並列性について効果を検証する．

本稿の構成を述べる．まず，第 2 節では Zero-Wait 方式について概要を述べる．次に第 3 節で多重の I/O 要求処理における問題点について説明する．さらに，第 4 節では実現した I/O 要求に対する処理について述べる．また，第 5 節で効果の検証としてシミュレータによる評価を行い，最後に第 6 節でまとめる．

2. Zero-Wait 方式

2.1 Zero-Wait スレッド

Zero-Wait 方式に基づく Zero-Wait スレッドは以下のような特徴を持つ．

- スレッドはいったん実行が開始されると wait なしで走り切る命令列である．スレッドの実行は実行開始点から必ず実行され，途中から実行されることはない．
- スレッドの実行順序は半順序関係であり，スレッドが次に実行すべきスレッドを継続命令により指定する．
- 複数のスレッドで構成されたプログラムは，スレッドをノードとし継続関係をエッジとするデータフローグラフになる．

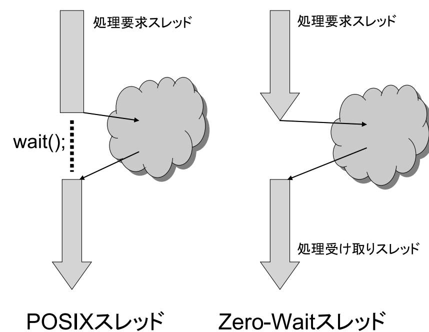

図 1 Zero-Wait スレッド

Zero-Wait スレッドは wait なしで走り切る命令列なので，例えば POSIX スレッドのように途中で明示的に待ち状態を記述することはない (図 1)．ただし，実際の実行時に継続処理が完了して実行可能状態になってからプロセッサ資源が割り当てられるまでの実行待ち状態はある．また，Zero-Wait スレッドでは遅延が生じる可能性のある処理においては，処理要求と処理受け取りをそれぞれ別のスレッドに分離するスプリットフェーズ実行方式を用いる．この方式により，処理結果を受け取るまでの間に別のスレッドが実行できることから，遅延を隠蔽することによる処理スループットの向上が期待できる．

2.2 Zero-Wait 方式における実行制御機構

Zero-Wait 方式では，スレッド間の実行制御を行うために以下のような機構が必要である．

2.2.1 同 期 機 構

Zero-Wait スレッドでは，一つのスレッドが複数の先行スレッドから継続を受ける場合がある．したがって，先行スレッドからの継続処理と同時に対象スレッドが実行可能状態になるとは限らない．そこで各スレッドは同期カウンタを持ち，原則としてその初期値は先行スレッドの数になる．

あるスレッドが継続処理を行う場合，継続先のスレッドの同期カウンタをデクリメントする．そして，継続先スレッドの同期カウンタが 0 になるとスレッドは実行可能状態になる． この実現方法により，複数スレッ

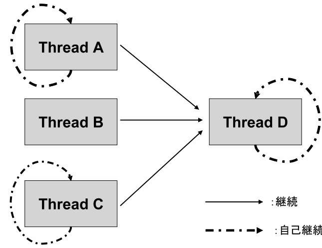

図 2　排他制御

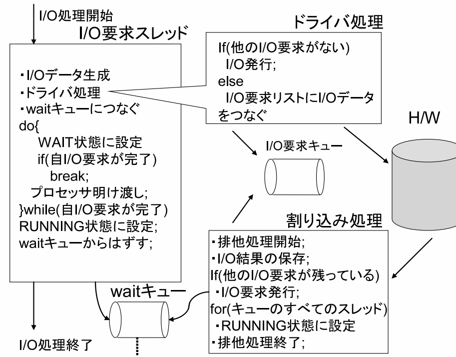

図 3　wait キューを用いた方式

ドからの継続処理に対処することができる．

2.2.2　排 他 制 御

多重並行処理で排他的な処理を実現するために，排他的な処理が必要な箇所をスレッド化し，そのスレッドの排他的な実行を保証する機構が必要である．

例えば，複数回実行されるが同時には実行できないスレッドがある場合を考える (図 2)．この例では，3 つのスレッド (Thread A，Thread B，Thread C) が Thread D に対して継続を行っている．しかし，Thread D を同時に実行可能にできる先行スレッドは 1 つだけとし，排他制御によってスレッドの継続を制限する必要がある．例えば，後述する Fuce プロセッサでは，スレッドをロックする機構があり，その機構を以下のように利用する．3 つのスレッドは Thread D に対してロック操作を試みる．ただし，ロックが成功できるスレッドは 1 つだけであり，ロックが成功したスレッド (例えば，Thread B) は Thread D へ継続する．また，ロックに失敗したスレッドは，再びロック操作を行うために自分自身への継続処理として自己継続処理を行う．自己継続処理は，自スレッドの同期カウンタ値を指定された値にリセットし，自スレッドに対して継続処理を行う．また，排他制御を行う対象である Thread D も自己継続処理を行い，先行スレッドからの継続を待つ．

3.　既存の I/O 要求に対する処理

本節では，I/O 要求に対する処理における問題点について述べる．

3.1　wait キューを用いた方式

Linux では wait キューを用いた方式を採用している． 処理の概要は図 3 のようになる．I/O 要求スレッドは I/O 処理に必要なデータを生成し，ドライバ処理を呼び出す．ドライバ処理では，キューに I/O 要求がなければ HW に対して I/O を発行し，あればデータをキューへつなぐ．ドライバ処理から復帰すると，I/O 要求スレッドは wait 状態になり自分自身を wait キューにつなぐ．そして，自分の出した I/O 要求が完了すると wait キューから取り除く．

割り込み処理では，割り込みレベルを設定し I/O 結果を受け取る．次にキューに I/O 要求があれば I/O を発行する．さらに wait キューにつながっているスレッドを起床し，RUNNING 状態にする．

Linux では、HW からの処理結果を受け取るスレッドは HW に対して要求を行うスレッドと同一であり、処理結果を受け取るまで wait 状態にしてキューで管理されている。また，自分の出した I/O 要求が完了したかどうかを wait キューのプロセスをいったん起床して確認する必要があり，完了していない場合にはプロセッサを明け渡す処理をしなければならない．Linux カーネル 2.6 では，ディスク I/O 時の処理において，wait キューを複数用意し，ハッシュ関数を用いて I/O 要求スレッドを分散することで，オーバヘッドを削減している．しかしこういった改善を行っても，自分の I/O 要求が完了したかどうかを確認する処理は，依然として行う必要がある．

3.2　直接起床方式

直接起床方式の処理の流れは図 4 のようになる．直接起床方式特有の処理は下線を引いた部分になる．直接起床方式では，wait キューを用いた方式のように I/O 完了待ちスレッドを wait キューで管理しない．その代わりに I/O 要求スレッドが自スレッドと自 I/O データの情報の組を，例えば I/O 要求スレッド情報保存テーブルに保持する．これにより，割り込み処理の中で当該 I/O データを発行したスレッドは一意に指定し，起床することができる．また，自分の I/O 要求が完了したかどうかを確認する処理を行う必要がな

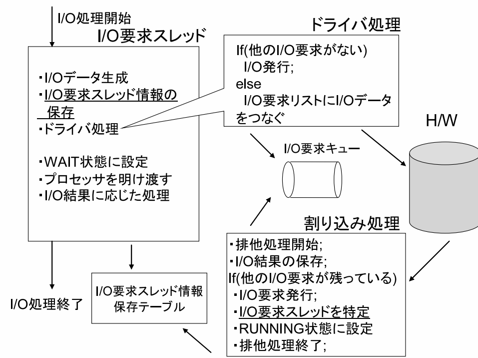

図 4　直接起床方式

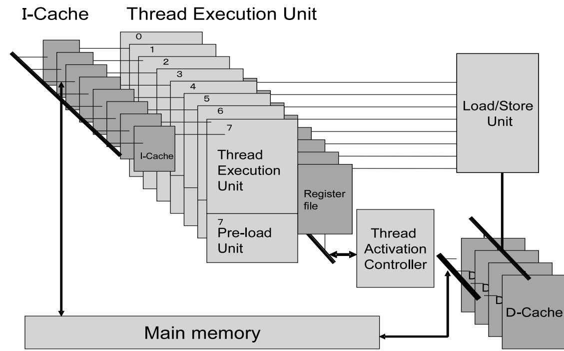

図 5　Fuce プロセッサ

くなる．したがって，wait キューを用いた方式と比較して，オーバヘッド削減が実現できる．

3.3　既存の方式と Zero-Wait 方式との比較

直接起床方式によるオーバヘッドの削減効果については，文献 [5] で wait キューを用いた方式との比較で確認されている．直接起床方式は，Zero-Wait 方式の派生方式とも言え，Zero-Wait 方式においても同様のオーバヘッドの削減効果が期待できる．しかしながら，文献 [5] での直接起床方式は処理自体が逐次実行方式にもとづいており，評価も逐次実行方式にもとづいたプロセッサ上で行っている．それに対し，Zero-Wait 方式は処理自体はデータフロー計算モデルの概念にもとづいており，データフロー方式のような細粒度並列プロセッサ上でより効果的と考える．

データフロー計算モデルは並列処理と親和性が高く，プログラムに内在する並列性を引き出すことが可能である．したがって，そのような計算モデルと親和性の高いプロセッサ上で，Zero-Wait 方式はよりカーネル内処理の並列性を引き出し，プロセッサの並列度を上げた時により高い性能向上を出すことが可能であると考える．

4.　実現した I/O 要求に対する処理

本節では，Zero-Wait 方式にもとづく Fuce プロセッサ上での I/O 要求に対する処理について述べる．まず，今回用いた Fuce プロセッサについて概要を述べる．次に実現した処理の概要について説明する．

4.1　Fuce プロセッサ

4.1.1　構　　成

Fuce プロセッサは，並列処理との親和性の高いデータフロー計算モデルをもとにした並列プロセッサであり，スレッドの実行を行う命令実行ユニット (スレッド実行ユニット：Thread Execution Unit) を複数個搭載しているチップマルチプロセッサである．その概要図を図 5 に示す．継続概念に基づいたスレッド実行を実現するために，プロセッサ内部にスレッド管理機構 (Thread Activation Controller) を持ち，第 2 節で説明した Zero-Wait 方式と親和性が高い．

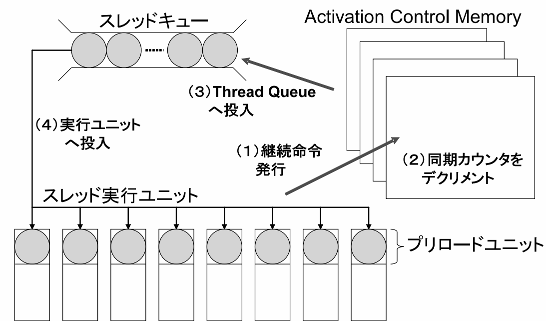

図 6　Fuce プロセッサにおけるスレッド管理機構

4.1.2　スレッド管理機構

プロセッサ内部に用意されたスレッド管理機構は，制御ユニット，Activation Control Memory(ACM)，スレッドキューによって構成される (図 6)．スレッドの同期情報は，ACM を利用することで管理する．ACM はスレッドの情報を保持するためにプロセッサ内部に構成された高速メモリである．Fuce プロセッサは，スレッド実行ユニットで継続やスレッド登録などのスレッド制御に関する命令を発行することで，ACM に対してスレッドの同期処理や登録処理を行う．スレッドの

情報をスレッドキューが受け取り，スレッドキューはその情報をもとにしてスレッドを実行する．スレッドキューは同期が完了したスレッドの情報を保持している．スレッドキューが持つスレッドは FIFO 順に実行される．実行ユニット中のスレッド実行と並行して，プリロードユニットでは次のスレッドの実行コンテキストを先読みする．

4.1.3 Fuce プロセッサ上における特徴

OS を実装することを考えた場合，Fuce プロセッサには以下のような点で特徴があると考えられる．

(1) スレッド管理
既存のプロセッサでは，スレッドの実行制御をソフトウェアが行っていた．この場合，スレッド管理コストが高くなってしまうという問題点がある．一方，Fuce プロセッサでは，スレッド管理機構を利用してスレッド管理の一部をハードウェアで行っている．これにより，Zero-Wait スレッドのような細粒度スレッドを用いた場合に問題となる，スレッド管理コストを削減することができる．

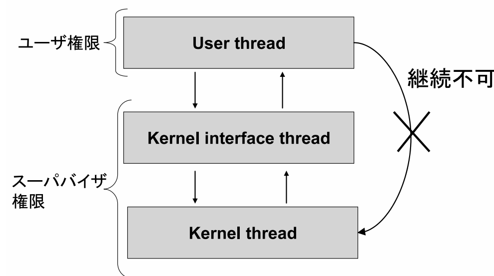

図 7 Fuce プロセッサにおけるモード切替え

(2) モード切替え
既存のプロセッサでは，プロセッサ自身にユーザモードとスーパバイザモードといった複数の状態を持たせている．例えば，ユーザプロセスからシステムコールを発行すると，プロセッサはスーパバイザモードに切り替わり，カーネル関数の実行を始める．
Fuce プロセッサでは，スレッド単位でユーザモードとスーパーバイザモードの二つの特権レベルを設けている．原則として，ユーザ権限で走行するスレッド (User thread) は，スーパバイザ権限のスレッド (Kernel thread) に継続することができない．しかしながら，スーパバイザ権限のスレッドのうち，User thread から継続可能なスレッド (Kernel interface thread) を設定する (図 5)．例えばユーザプロセスからシステムコールを発行する場合，User thread が Kernel interface thread に対して継続処理を行う．このようにすることで，Fuce プロセッサでは，ユーザ空間とカーネル空間をまたぐ処理も継続概念にもとづいて行われる．

4.2 実現した処理の概要

Zero-Wait 方式による I/O 要求に対する概要を図 6 に示す．以下，各スレッドの処理内容について説明する．

4.2.1 sender_thread

ユーザ空間からシステムコール発行を要求する．ユーザ空間からのシステムコール要求は Kernel interface thread である gate_thread が受け取る．最初に gate_thread に対してロック操作を行う (1-1)．ロックに成功した場合，gate_thread に対して継続処理を行う (1-2)．継続処理では，システムコール番号，システムコール処理に必要な引数，ユーザ側の結果受け取スレッド (receiver_thread) の情報を渡す．ロックに失敗した場合には，自己継続処理を行い再び gate_thread に対してロック操作を行う (1-3)．

4.2.2 gate_thread

sender_thread から受け取ったシステムコール番号から該当するシステムコールを特定し，その処理本体であるスレッド (syscall_thread) の ID を求める (2-1)．求めた syscall_thread に対して継続処理を行う (2-2)．継続処理では，システムコール処理に必要な引数，receiver_thread の情報を渡す．

4.2.3 syscall_thread

システムコール本体の処理を行い，処理内容に該当するデバイスに対応する semaphore_thread へ継続処理を行う (3-1,3-2)．

4.2.4 semaphore_thread

まず，継続先のスレッドである device_thread に対してロック操作を行う (4-1)．ロックに成功した場合，device_thread に対して継続処理を行う (4-2)．ロックに失敗した場合には，HW 処理に必要なデータをキューへ格納する (4-3)．これは，HW では I/O 処理が完了していないと次の処理を行うことができないからである．

4.2.5 device_thread

semaphore_thread からデータを受け取り (5-1)，de-

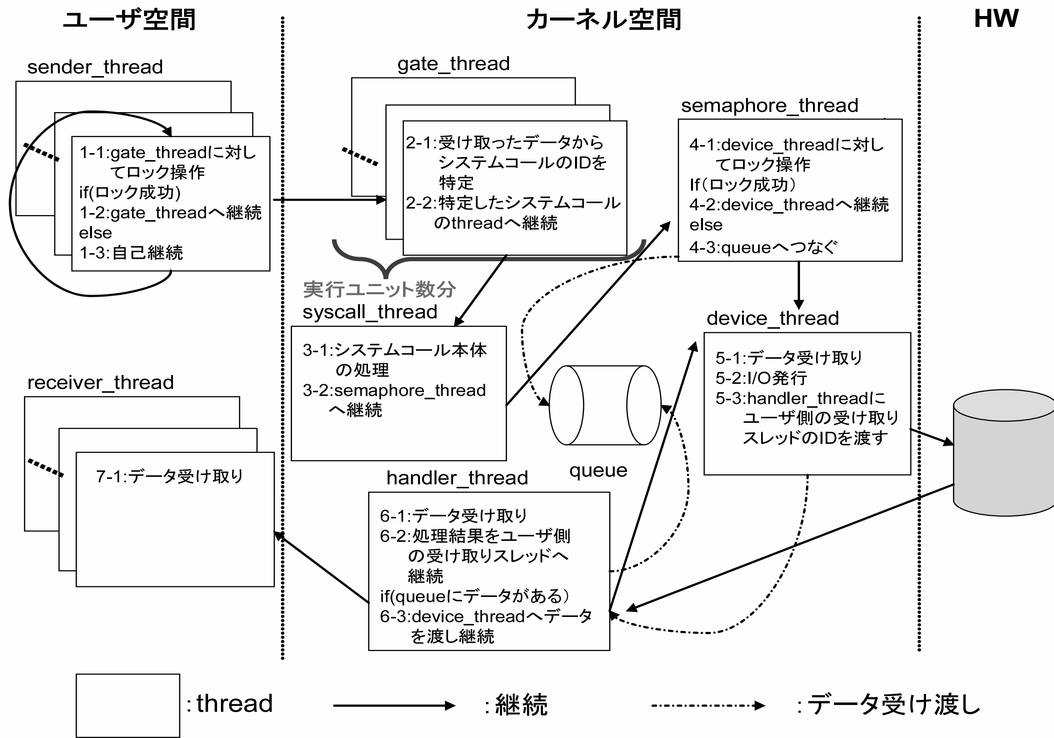

図 8 Zero-Wait 方式による多数の I/O 要求に対する処理

vice_thread は HW に対して I/O 発行を行う (5-2)．また，handler_thread に対して継続処理を行い，receiver_thread の情報を渡す (5-3)．

4.2.6 handler_thread

HW からの割り込みによって起動し，HW から処理結果を受け取る (6-1)．次に receiver_thread に対して継続処理を行い，受け取った処理結果を渡す (6-2)．また，キューの中身を調べ，他の I/O 要求があれば device_thread に対して継続処理を行い，queue から取り出したデータを渡す (6-3)．

5. 評　　価

本節では前節で述べた処理に関して，カーネル内処理の並列性についてシミュレータを用いた評価を行う．評価においては，直接起床方式との比較を行う．

5.1 評価環境

実装した処理の評価は，VHDL で記述した Fuce プロセッサを ModelSim 上でシミュレートし，その上でプログラムを実行することで行った．今回評価に用いた Fuce プロセッサは，最大 8 本の実行ユニットを持ち，スレッド管理機構 (TAC) のアクセスレイテンシは 1 サイクルである．ただし，データキャッシュは搭載していない．

5.2 評価方法

評価に際してソフトウェアシミュレータを用いているので，実デバイスを用いた評価をすることができない．そこで，デバイスはシミュレートすることで評価を行った．具体的には，デバイスをシミュレートするためのスレッド (virtual_hw_thread) を用意する．実装した処理では device_thread がデバイスに対して I/O 発行を行っているが，ここでは device_thread が virtual_hw_thread に対して継続処理を行う．次にデバイス処理が終了すると本来であればデバイスからの割り込みで handler_thread が起動して処理を開始するが，この部分は代替として virtual_thread が handler_thread に対して継続処理を行うことで handler_thread の処理を開始するようにする．

5.3 カーネル内処理の並列性に関する検証

まず，Zero-Wait 方式では，以下の方法で評価を行った．

(1) 単位時間 (T) あたりシステムコール要求 (sender_thread) を N 個実行ユニットに投入する．そして，それを M 回行い総クロック数を測定する．

(2) Nの値を増やしていくと，ある値で総クロック数が上昇する．これは，単位時間内に要求を処理することができなくなり，Fuceプロセッサ内のスレッドキューにスレッドが溜ってしまうためである．そこで，上昇する直前のNの値を最大スループット数とする．

(3) (1),(2)の測定を実行ユニット数を変化(1，2，4，6本)させて，各実行ユニット数における最大スループット数を求める．

ここで，Zero-Wait方式にもとづくFuceプロセッサ上の実装における並列性の効果を示すために直接起床方式[5]をLinuxカーネル2.6.16，Pentium 4 1.6GHz上で実現し，両者の比較を行う．

直接起床方式は，Zero-Wait方式と比較するとスレッド切替えの回数とコンテキスト切替えの回数に関しては，傾向が同じであることが考えられる．しかしながら，処理自体は逐次実行ベースであり，プロセッサも逐次実行ベースである．継続概念にもとづくFuceプロセッサ上で動作するZero-Wait方式と，逐次実行方式にもとづく汎用プロセッサ上で動作する直接起床方式で，それぞれプロセッサの並列度を上げたときの性能向上比を比較する．これにより，カーネル内処理の並列性についてZero-Wait方式による効果を検証することができると考える．

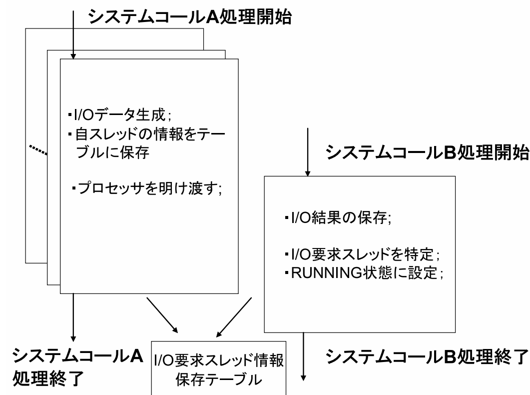

図9 測定プログラム

今回は，カーネル内処理の並列性に着目して測定を行う．そこで，実際にI/O処理に関わる部分(ドライバ処理，割り込み処理におけるI/O発行処理)を除いたものをソースファイルとして抜き出し，抜き出した処理を直接呼ぶシステムコール(AおよびB)を作成する．その処理の流れを図9に示す．システムコールAはI/O要求スレッドに相当する．また，I/O処理に関わる部分を抜き出しているので，割り込み処理に該当する部分をシステムコールBとして実現している．次に，以下のような処理を行う測定用のプログラムを作成する．

(1) 子スレッドを指定個生成する．
    (a) 子スレッドは，システムコールAを発行し，wait状態になり，プロセッサ資源を明け渡す．

(2) (1)で生成した子スレッドの個数分次の処理を繰り返す．
    (a) システムコールBを発行し，子スレッドを一つ起床する．
        (i) 子スレッドは，起床されると一定時間後にそのまま終了する．なお，子スレッドの終了処理は測定には含まれない．

測定プログラム中の子スレッドは第4節でのsender_threadに相当する．また，システムコールAおよびBはsemaphore_threadからhandler_threadまでの一連の処理に相当する．

上記のプログラムを利用して，Zero-Wait方式での測定と同様にSMTプロセッサを用いて，各論理プロセッサ数(1，2，4個)における単位時間(T)あたりの最大スループット数を求め，論理プロセッサ数を1とした時の最大スループット数を基準として，各論理プロセッサ数(2，4)における性能向上比を計算する．ただし，論理プロセッサ数が1というのは，シングルプロセッサが1個の時のことである．(以下，論理プロセッサを実行ユニットと呼ぶ)

Zero-Wait方式における各実行ユニット数での性能向上比を比較したグラフを図10に示す．実行ユニットが6本の時に最大2.89倍の性能向上が得られている．また，直接起床方式との比較を図11に示す．いずれの実行ユニット数においてもZero-Wait方式の方が高い性能向上が得られている．

今回は，1つのデバイスを想定して評価を行っている．例えばデバイスを複数にした場合，カーネル内処理においてさらに多重処理の部分が増加するので，さらなる並列性を抽出することが可能であると期待できる．

6. おわりに

本稿では，並列処理と親和性の高いデータフロー計算モデルの概念を取り入れたZero-Wait方式を用いて，Fuceプロセッサ上でのOS機能の実現例として

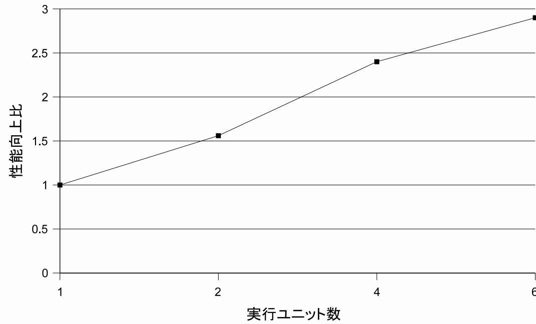

図 10　各実行ユニット数における性能向上比

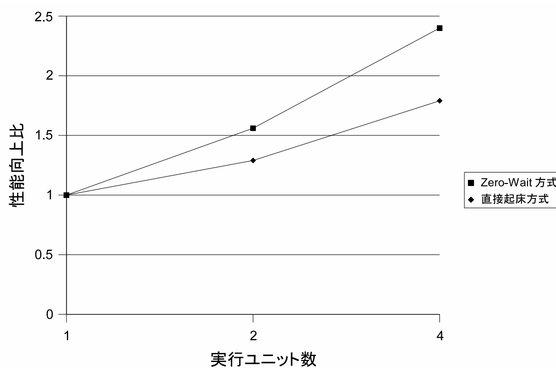

図 11　Zero-Wait **方式と直接起床方式の比較**

I/O 要求に対する処理を取り上げた．シミュレータによる評価では，カーネル内処理の並列性についてその効果を検証した．評価結果では，実行ユニット数が増加した場合に性能向上が得られた．また直接起床方式との比較を行い，Zero-Wait 方式の方が実行ユニット数の増加に伴った性能向上が得られた．このことにより，今回実現した方式ではカーネル内処理においてより並列性を引き出すことが可能であることを示した．

今後は，実現した処理について詳細な評価を行い，複数のデバイスに対して要求が発生した場合について今回の実現した処理を適用し，評価を行う．また，他の OS 機能に対して Zero-Wait 方式の適用について検討し，実装・評価を行っていく予定である．

参　考　文　献

1) Masato Edahiro, Satoshi Matsushita, Masakazu Yamashina, and Naoki Nishi. A Single-Chip Multiprocessor for Smart Terminals. IEEE Micro, vol.20, No.4, pp.12-20, 2000.
2) Deborah T.Marr, Frank Binns, David L.Hill, Glenn Hinton, David A.Koufaty, J.Alan Miller, and Micheal Upton. Hyper-threading technology architecture and microarchitecture: A hypertext history. Intel Technology Journal, Vol.6, No.1, 2002.
3) Karthikeyan Sankaralingam, Ramadass Nagarajan, Hamaing Liu, ChangkyuKim, Jaehyuk Huh, Doug Burger,Stephen W.Keckler, Robert G.McDonald, Charles R.Moore. TRIPS:A polymorphous architecture for expoliting ILP,TLP and DLP. ACM Transaction on Architecture and Code Optimization(TACO), Vol.1, NO 1, pp.62-93, March 2004
4) Steven Swanson, Ken Michelson, Andrew Schwerin, MarkOskin. WaveScalar. Proceeding of the 36th annual IEEE/ACM International Symposium on MicroArchitecture, pp291-302, 2003
5) 日下部 茂，乃村 能成，谷口 秀夫，雨宮 真人．継続概念を用いた Zero-Wait 方式による OS 構成法の提案．情処研報，OS-99，pp.69-76，2005
6) 雨宮　聡史，松崎　隆哲，雨宮　真人．排他実行マルチスレッド実行モデルに基づくオンチップ・マルチプロセッサの設計．情処研報，ARC-155，pp.51-56，2003.