C-021

フォールトトレラントコンピュータにおける計算機再組込みのためのメモリ再同期手法
Memory Resynchronization Method for Node Recovery in FTC

吉田 雅徳† Masanori Yoshida　関山 友輝† Tomoki Sekiyama　大島 訓† Satoshi Oshima　大平 崇博† Takahiro Ohira

1. 背景

社会インフラ分野における情報システムでは、極めて高い可用性と信頼性が要求される。これを実現するための一つの方式がフォールトトレラントコンピュータ（ＦＴＣ）である。ＦＴＣはコンピュータの各コンポーネントを多重化し、多重化された各コンポーネントに同じ処理内容を同時に実行させ、その結果得られる多重化された出力に対して多数決を実施し、最終的な出力内容を決定するコンピュータ構成方式である。単一コンピュータと比べ、多数決による非常に高信頼な出力が得られるとともに、一部のコンポーネントで障害が発生しても残りのコンポーネントが処理を継続することにより、一切サービス停止が発生しない高可用なシステムが構成出来る。

近年、これら情報システムが提供するサービスが大規模化及び多様化した結果、高可用性・高信頼性に加えて、高い性能と機能も要求されるようになった。従来、高可用性・高信頼性の要求へは専用ハードウェアを用いることで対応することが一般的だったが、高性能・高機能の要求への対応が困難となってきている。

こういった背景から、複数の計算機をソフトウェアによって連携させることによりＦＴＣを実現する「ソフトウェアＦＴＣ」への期待が高まっている。図 1に、本研究で想定するソフトウェアＦＴＣの構成を示す。

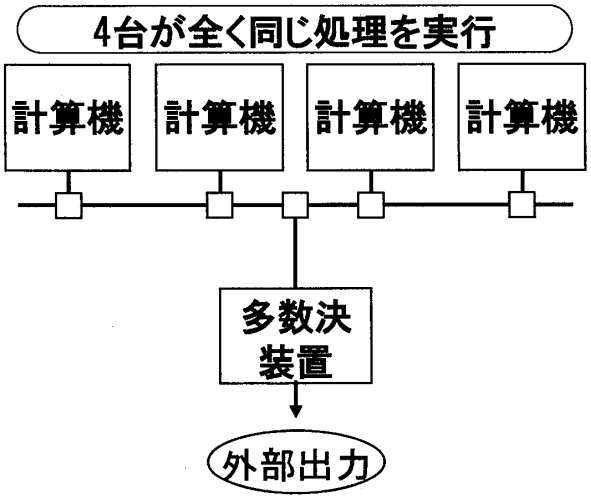

図１　ソフトウェアＦＴＣ

2. 目的

本研究では、ソフトウェアＦＴＣで計算機再組込みを実現するために必要となる「メモリ再同期」の手法を提案する。

計算機再組込みとは、ソフトウェアＦＴＣを構成する計算機群の中の１台で障害が発生した場合に、同障害計算機を修理もしくは交換した後、同計算機（以下、組込み計算機と呼称）と、稼働継続中の他計算機（以下、稼働中計算機と呼称）との同期状態を回復させる機能である。稼働中計算機の処理を停止せずに、組込み計算機へアプリケーション状態をコピーし、両計算機のアプリケーション状態を一致させる。

メモリ再同期とは、計算機再組込み機能を構成する機能の一つであり、アプリケーション状態の中でも特にメモリ上のデータを一致させる。稼働中計算機のアプリケーションはメモリを絶えず更新し続けるため、まずコピー対象のメモリ領域全体をコピーし、その後はメモリ更新差分を繰り返しコピーすることが必要となる。十分短時間で送信可能なサイズまでメモリ更新差分のサイズが小さくなった段階で、稼働中計算機のアプリケーションを一時停止し、最終の更新差分をコピーして一致化完了となる。メモリ再同期処理の全体の流れを図 2に示す。

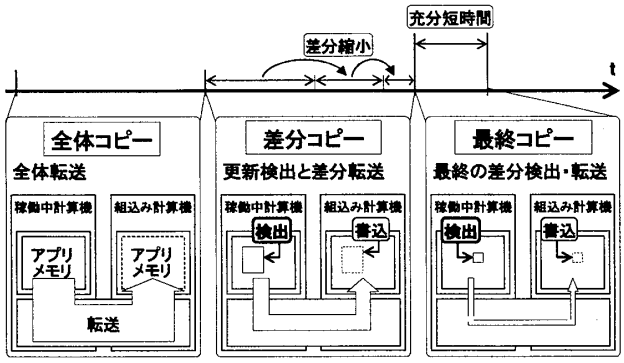

図２　メモリ再同期処理

本研究では特に、メモリ更新差分コピーを実現するために必要となる、メモリ更新箇所の検出について、稼働中計算機上のアプリケーションへのオーバヘッドを従来より大幅に低減しつつ、最終コピー時の一時停止時間を短く保つことが出来る方式を提案する。同方式により、特に、巨大なメモリ空間を使用するアプリケーションについて、効率的にメモリ更新箇所を検出することが出来る。

† （株）日立製作所　Hitachi, Ltd.

3. メモリ更新検出方式

3.1 概要

メモリは多くのシステムリソースと異なり、アプリケーションはカーネルが提供するシステムコールを介さずに直接メモリの読み書きを行うため、システムコールにフックを埋め込む方法では更新を検出することは出来ない。そこで本研究では、メモリの更新を検出するためにメモリ管理ユニット（MMU：Ｍｅｍｏｒｙ　Ｍａｎａｇｅｍｅｎｔ　Ｕｎｉｔ）を利用している。

MMUを利用してメモリ更新検出を行う既存技術としては、Ｘｅｎ®のライブマイグレーション[1]がある。しかし同技術では、ライトプロテクションを使用してメモリ更新のたびに例外を発生させており、アプリケーション処理へのオーバヘッドが高い。

本研究では、ライトプロテクションとページテーブルのＤｉｒｔｙフラグを組み合わせて活用することで、アプリケーション処理へのオーバヘッドを削減しつつ、メモリ更新検出を効率的に行う方式を提案する。

3.2 実証環境

本研究で提案するメモリ更新検出方式は、**表 1**に示す環境を使用して実装を行い、動作を実証した。本稿の以降の説明では、同環境を想定として説明を行う。但し、本方式は基本的にＯＳカーネルの種別には依存せず、ＣＰＵについても、Ｉｎｔｅｌ®　ｘ８６アーキテクチャはもちろん、階層型ページテーブル構造を持つ他社の多くのＣＰＵに対しても同様に適用可能と考える。

表 1　実証環境

(1)	CPU	Intel Xeon® 5140
(2)	OS カーネル	Linux® 2.6.12.5

3.3 MMUとページテーブル

MMUとは、中央演算装置（ＣＰＵ：Ｃｅｎｔｒａｌ　Ｐｒｏｃｅｓｓｉｎｇ　Ｕｎｉｔ）のメモリアクセスを助けるハードウェア機構である。ＣＰＵが仮想アドレスを指定してメモリにアクセスする時に必要となる諸処理（仮想アドレスから物理アドレスへの変換など）を実行する。

MMUの動作は、ページテーブルと呼ばれるデータ構造により決定される。ＣＰＵのアーキテクチャによって異なるが、多くの場合、ページテーブルは他のデータと同様にメモリ上に配置されるテーブルデータであり、カーネルはページテーブルを操作することでMMUの動作を制御することが出来る。ページテーブルは、その名が示すとおり、メモリを一定サイズのブロック、すなわちページの列として管理しており、各ページに対応する設定を格納するページテーブルエントリ（ＰＴＥ：Ｐａｇｅ　Ｔａｂｌｅ　Ｅｎｔｒｙ）により構成される。従って、MMUの諸処理はページサイズの粒度で行われることになる。

多くのMMUは、メモリの仮想アドレス-物理アドレス変換を行うだけでなく、メモリページのアクセス権や更新有無を管理する機構を持っており、ＰＴＥを適切に設定することでこれらの制御が可能である。これらの機構はメモリ更新検出に活用することが出来る。

（1）Ｒ／Ｗフラグ

ＰＴＥが持つ「Ｒ／Ｗフラグ」をクリアしてページへの書き込みを禁止することを、ライトプロテクションと呼ぶ。ライトプロテクションが設定されたページにアプリケーションがアクセスすると、例外が発生してカーネルが用意した例外ハンドラが呼び出される。この機構を使用すれば、例外ハンドラ処理の中でメモリ更新を検出することが出来る。

（2）Ｄｉｒｔｙフラグ

ＰＴＥが持つ「Ｄｉｒｔｙフラグ」は、一旦クリアした後、当該ＰＴＥに対応するページへアプリケーションが書込みを最初に行う時に、MMUによって再度セットされる。この機構を使用すれば、例外を発生せずにメモリ更新を検出することが出来る。

3.4 ページテーブル階層構造

Ｉｎｔｅｌ®社ＩＡ３２アーキテクチャ[2]では、ページテーブルは階層構造を取る。同階層構造では、ページを管理する下位構造を「ページテーブル」、それらページテーブルを管理する上位構造を「ページディレクトリ」と呼称する。そこで以降では、下位構造としてのページテーブルと区別するために、階層構造全体をまとめた呼称としては「ページテーブル階層構造」を使用する。

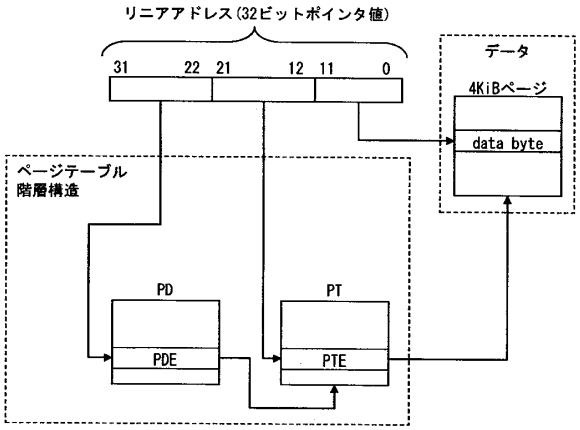

図 3 ページテーブル階層構造

ページテーブル階層構造の概要を**図 3**に示す。アドレス空間全体（４ＧｉＢ）は、１０２４個の４ＭｉＢのブロックの配列としてページディレクトリで管理される。ページディレクトリはページディレクトリエントリ（ＰＤＥ：Ｐａｇｅ　Ｄｉｒｅｃｔｏｒｙ　Ｅｎｔｒｙ）の配列である。各ＰＤＥは各４ＭｉＢブロックに関する設定と、下位構造であるページテーブルの物理アドレスを格納する。ＣＰＵが指定した３２ビットのリニアアドレス

の上位１０ビットは、ページディレクトリ内のＰＤＥを指定するためのオフセットとして使用される。

ページテーブルは、各４ＭｉＢブロックを４ＫｉＢページの配列として管理する。ページテーブルはページテーブルエントリ（ＰＴＥ：Ｐａｇｅ　Ｔａｂｌｅ　Ｅｎｔｒｙ）の配列である。各ＰＴＥは各４ＫｉＢページに関する設定と、同４ＫｉＢページの物理アドレスを格納する。ＣＰＵが指定した３２ビットのリニアアドレスの次の１０ビットは、ページテーブル内のＰＴＥを指定するためのオフセットとして使用される。

ＰＴＥとＰＤＥは構成が似ており、特に両者共にＲ／Ｗフラグを持つ。ＰＴＥのＲ／Ｗフラグをクリアした場合は４ＫｉＢページ毎に、ＰＤＥのＲ／Ｗフラグをクリアした場合は４ＭｉＢブロック毎に、ライトプロテクションが設定される。

一方、ＰＴＥにはＤｉｒｔｙフラグが存在するが、ＰＤＥには存在しない。そのため、Ｄｉｒｔｙフラグを使用したメモリ更新検出を４ＭｉＢ単位で行うことは出来ない。

本研究によるメモリ更新検出方式では、ＰＴＥのＤｉｒｔｙフラグと、ＰＤＥのＲ／Ｗフラグを組み合わせて使用する。

3.5 制約条件

本研究によるメモリ更新検出方式では、更新検出対象アドレス範囲に対応するＰＴＥ及びＰＤＥを操作する。一方、本来のＯＳカーネルの処理でもＰＴＥ及びＰＤＥを操作する場合があり、両者が競合するとメモリ更新を正常に検出出来ないだけでなくＯＳカーネルが正常に動作しなくなる。

メモリ更新検出方式とＯＳカーネルの競合を避けるため、更新検出対象として指定可能なメモリ領域に、**表 2**に示す制約条件を設ける。次項以降の説明では、この制約条件を前提とする。

表 2　制約条件

(1)	対象領域に対し mlock 済みであること
(2)	対象領域に対し msync を発行しないこと

(1)の条件はページ回収によるＰＴＥ全体のクリアを、(2)の条件はｐａｇｅ構造体のＰＧ＿ｄｉｒｔｙフラグ設定後のＰＴＥのＤｉｒｔｙフラグのクリアを、それぞれ避けるためのものである。ＰＴＥのＤｉｒｔｙフラグ以外のフラグを書き換える操作については特に制限は不要である。また、Ｌｉｎｕｘ®２．６．１２．５ではアプリケーションの動作中（ｆｏｒｋ～ｅｘｉｔ）にＰＤＥを変更する処理は存在しないため特に対処していない。

3.6 機能定義（入出力定義）

本研究によるメモリ更新検出方式により実現する機能を**表 3**に示す。

表 3　機能定義

入力	(1)対象プロセス (2)対象アドレス範囲
入力条件	対象アドレス範囲が、開始・終了アドレス共に４ＭｉＢにアラインされていること。
出力	対象プロセスの対象アドレス範囲において、更新されたページの開始アドレスの配列

3.7 処理構成

メモリ再同期処理としては、稼働中計算機のアプリケーションからメモリ内容を読み込み、計算機間で転送した後に組込み計算機のアプリケーションへメモリ内容を書き込む処理も必要になるが、具体的にアプリケーションが使用するメモリ領域へアクセスしてメモリ内容を読み書きする方法については言及しない。これが容易に可能な対象としてはＩＰＣ共有メモリがある。それ以外の、アプリケーション固有のメモリ領域（スタック等）へのアクセス方法については、ＯＳカーネル毎あるいはシステム毎に、別途検討する必要がある。

本研究によるメモリ更新検出方式では、メモリ再同期専用のデーモンが処理フローを管理する。一方で、稼働中計算機上のアプリケーションのプロセスも処理を継続する。両者の処理の概要を**表 4**にまとめた。

表 4　メモリ更新検出処理構成

呼出元	処理	概要
メモリ再同期デーモン	更新検出開始	PTE/PDE に変更を加え、以降の「更新箇所取得」呼出で、更新箇所(※)が取得出来るよう準備する。
	全体転送	「更新検出開始」の後、対象プロセスの対象アドレス範囲全体を一度コピーする。
	更新箇所取得(初回)	PTE/PDE を走査し、更新検出開始～更新箇所取得(今回)の間に発生した更新箇所を取得し、呼出元へ返す。 PTE/PDE に変更を加え、次回の更新箇所取得呼出で、更新箇所が取得出来るよう再度準備する。
	更新箇所取得(2回目以降)	PTE/PDE を走査し、更新箇所取得(前回)～更新箇所取得(今回)の間に発生した更新箇所を取得し、呼出元へ返す。 PTE/PDE に変更を加え、次回の更新箇所取得呼出で、更新箇所が取得出来るよう再度準備する。
	差分転送	「更新箇所取得」の結果に応じて、メモリ更新差分をコピーする。

	更新検出終了	PTE/PDE の状態を更新検出開始以前の状態に戻す。
アプリ	ページフォルト例外ハンドラ	PDE に設定したライトプロテクションが原因で例外が発生した場合、PDE の R/W フラグをセットする。

※更新発生ページの先頭アドレス

次節以降で**表 4**に示した各処理の詳細を説明する。

3.8 更新検出開始処理

更新検出開始処理は、**表 5**に示す処理を順番に実行することにより実施される。

表 5　更新検出開始処理

#	ステップ	処理
1	PDE 処理	対象プロセスの対象アドレス範囲に対応する PDE の R/W フラグをクリアする。
2	PTE 処理	対象プロセスの対象アドレス範囲に対応する PTE の Dirty フラグをクリアする。
3	TLB フラッシュ	(1) 対象プロセスが現在アクティブ(※)か調べる。 (2) アクティブならば、当該 CPU コアに対して TLB フラッシュを命令する IPI を送信する。

※CPU コアを占有して実際に動作中であること

最初に PDE にライトプロテクションを設定し、次に PTE の Dirty フラグのクリアを行う。この順序は、次節で説明する更新箇所取得処理においては、同処理実行中に発生したメモリ更新の検出漏れを防ぐために重要である。しかし、更新検出開始処理においては重要ではなく、逆でも問題無い。更新検出開始処理については、同処理実行中に発生した更新は検出する必要はなく、同処理が完了して呼出元へ戻った後の更新が検出出来れば充分であるためである。

更新検出開始処理の呼出元へ戻る直前に、TLB フラッシュ処理を実行している。TLB は MMU が持つページテーブル構造のキャッシュであり、この内容を破棄させることにより、以降に対象プロセスが対象アドレス範囲を読み書きした時に、変更済みの PDE 及び PTE を MMU に再読み込みさせることが出来る。これにより、これ以降に発生したメモリ更新を確実に検出出来る。但し、対象プロセスがアクティブで無い場合、TLB をフラッシュさせる必要は無い。プロセスがアクティブになる時、コンテキストスイッチにより TLB がフラッシュされるからである。

3.9 ページフォルト例外ハンドラ処理

PDE の R/W フラグをクリアしたことにより、対象プロセスが対象アドレス範囲に書き込むと、ページフォルト例外が発生してページフォルト例外ハンドラが呼び出されるため、これを適切にハンドリングする必要がある。

本研究によるメモリ更新検出方式では、ページフォルト例外ハンドラに変更を加え、例外発生原因が PDE の R/W フラグをクリアしたことであった場合、当該 PDE の R/W フラグをセットして戻るようにする。

次節で説明する更新箇所取得処理では、PDE の R/W フラグのセット有無により、当該 PDE に対応する 4 MiB アドレス範囲での書込み有無を判別する。従って、ここでは R/W フラグをセットするだけで充分である。

3.10 更新箇所取得処理

更新箇所取得処理は、**表 6**に示す処理を順番に実行することにより実施される。

表 6　更新箇所取得処理

#	ステップ	処理
1	PDE 処理	対象プロセスの対象アドレス範囲に対応する PDE の R/W フラグをテストし、セット済ならばクリアする。
2	TLB フラッシュ	(1) 対象プロセスが現在アクティブか調べる。 (2) アクティブならば、当該 CPU コアに対して TLB フラッシュを命令する IPI を送信する。
3	PTE 処理	上記 PDE 処理において、R/W フラグがセット済だった範囲について、PTE の Dirty フラグをテストし、セット済ならばクリアする。

最初に PDE の走査を行う。対象プロセスが PDE に対応するアドレス範囲へ 1 回以上書き込んだ場合、最初の書込み時にページフォルト例外が発生し、例外ハンドラ処理が当該 PDE の R/W フラグをセットしているはずである。従って、PDE の R/W フラグのセット有無によって 4 MiB 単位での書込み有無が判別出来る。R/W フラグがセット済だった PDE については、再度 R/W フラグをクリアする。これにより、当該 PDE に対応する 4 MiB アドレスでの新たな更新検出を可能とする。

次に、PDE の R/W フラグのクリアを有効化するため、TLB フラッシュ処理を実行する。

最後に、更新有と判別された 4 MiB アドレス範囲に対応する PTE を走査する。この、PTE 走査範囲を限定する「**枝狩り**」は、重要である。PTE は 4 KiB ページ毎に 1 個対応するため、更新検出の対象アドレス範囲が巨大である場合、走査時間が長くなるという問題があった。枝狩りによってこの走査時間を大幅に削減出来ている。

PTE の Dirty フラグがセット済ならば、当該 PTE に対応する 4 KiB ページが更新有と判別出来る。Dirty フラグがセット済だった PTE については、再度 Dirty フラグをクリアする。これにより、当該 PTE に対応する 4KiB での新たな更新検出を可能とする。

更新箇所取得処理における注意点として、TLB フラッシュ処理は、PTE の Dirty フラグの走査を実行する前に、必ず実行しておく必要がある。仮に TLB 未フラッシュの状態で PTE の走査をした場合、走査範囲に対応する PDE または PTE が TLB 内に残留している可能性がある。

PDE が残留していて R/W フラグがセット済である場合、対象プロセスが当該 PDE に対応する４ＭｉＢアドレス範囲に書き込んでもページフォルト例外が発生せずに４ＭｉＢ単位での更新検出が検出漏れを起こす。

PTE が残留していて Dirty フラグがセット済である場合、これはメモリ上のページテーブル構造には未反映であるかもしれず 4KiB ページ単位での更新検出が検出漏れを起こす。

3.11 更新検出終了処理

更新検出終了処理は、**表 7**に示す処理を実行することにより実施される。

表７　更新検出終了処理

#	ステップ	内容
1	PDE 処理	対象プロセスの対象アドレス範囲に対応する PDE の R/W フラグをセットする。

PDE に設定したライトプロテクションを解除するだけでよい。更新検出漏れを考慮しなくてよいので、TLB フラッシュ処理も不要である。

4. 性能考察

4.1 アプリケーションへのオーバヘッド

Ｘｅｎ®のライブマイグレーション[1]では、メモリ更新検出のために、４ＫｉＢページ毎にライトプロテクションを設定する方式を用いている。但し、更新頻度の高い領域であるＷｒｉｔａｂｌｅ　Ｗｏｒｋ　Ｓｅｔｓ（ＷＷＳ）を動的に検出し、ＷＷＳについては更新検出の対象から外して最終コピー時　に転送するという効率化を行っている。

ＷＷＳにより、更新が頻繁に発生する箇所でのページフォルト例外発生は抑えることが出来る。しかしＷＷＳの特定以前や、ＷＷＳ以外の領域について、４ＫｉＢの粒度でライトプロテクションを設定していることによりページフォルト例外が頻繁に発生するため、アプリケーションへのオーバヘッドは大きい。特に制御分野では、アプリケーションの処理時間が推測困難となり、リアルタイム性能要件が満たせなくなることが問題となる。

本研究によるメモリ更新検出手法では、ライトプロテクションを４ＭｉＢ単位で設定するため、Ｘｅｎ®のケースと比べ例外発生頻度は１／１０２４と、無視できる程小さくすることが出来ている。

4.2 最終コピー時の一時停止時間

最終コピー時の一時停止時間は、最終のメモリ更新箇所取得処理と、最終のメモリ更新差分転送処理の、合計所要時間で決まる。

後者（転送処理）については、本方式では実際に更新されたページのみを転送するため、最適（最小）な所要時間が達成出来ている。

前者については、アプリケーションのメモリ更新パターンに応じてその所要時間が変化する。更新箇所取得処理の所要時間で支配的な要素はＰＴＥ走査時間であり、ＰＴＥ走査範囲は、Ｒ／Ｗフラグセット済のＰＤＥの個数に比例する。つまり、更新箇所取得処理の所要時間は、アプリケーションが１回以上更新した４ＭｉＢブロックの個数に比例する。

最終コピー時の更新箇所取得処理では、１回前の更新箇所取得処理から最終コピー開始までの期間の、アプリケーションによるメモリ更新箇所が取得される。この期間は非常に短く（最終コピーを開始する条件はこの期間が充分短くなることである）、一般的には、同期間の短さに比例してアプリケーションによるメモリ更新箇所も局所性を持つ（更新された４ＭｉＢブロックの個数が少ない）と考えられる。

このように、本研究による更新箇所取得処理の所要時間はアプリケーションのメモリ更新パターンに依存して変化するが、通常は最終コピー時の更新箇所取得処理の所要時間は充分短くなっている。

5. 結論

本研究では、ＰＤＥ階層でのライトプロテクションとＰＴＥ階層でのＤｉｒｔｙフラグ走査を組合せることにより、アプリケーションへのオーバヘッドを従来より大幅に削減しつつ、最終コピー時の一時停止時間を短く抑えることが出来る、メモリ更新検出方式を提案した。

現時点での動作実証においては、多くの制約をアプリケーションに課すことで単純な実装による実現を得ているが、今後はこれら制約を外して適用対象を広げることが課題と考えている。

参考文献

[1]Christopher Clark, et al., “Live Migration of Virtual Machines”, NSDI05(2005).

[2]インテル社, “IA-32 インテル アーキテクチャ ソフトウェア・デベロッパーズ・マニュアル”

[3]大島 訓, 他, “ソフトウェアによる制御システム向け Fault Tolerant Computer の提案”, FIT2010 第 9 回情報科学技術フォーラム (2010).