Vol.2012-UBI-34 No.6
2012/5/17

バッテリレス・システム実現に向けた不揮発主記憶システムの検討

大村 廉[1,a)]　内門 裕紀[1,b)]

概要：本稿では，2 次電池を用いることなく，不安定な電源下でもユーザにストレスを与えずに動作する計算機システムを目指し，システムが不意の電源切断に面した状況においても，電源切断時の状態を高速に保存／復帰するシステムの実現に対する取組について述べる．本研究は低オーバヘッド化のため，主記憶が FeRAM や MRAM といった高速な不揮発メモリで構成されることを前提とし，システムを構成する CPU，主記憶，周辺デバイスについて一貫性を維持する状態の保存／復元を行うアプローチをとる．まず，分散システム（メッセージ・パッシングシステム）のアナロジーから導かれる一貫性維持のための要求事項について述べる．そして，オペレーティングシステムおよびデバイスドライバの改変によって要求事項を実現する手法について述べる．ソフトウェアによる手法を実用的なアプリケーション動作で評価した結果では，UART および Ethernet を用いた場合におけるオーバヘッドは多くともそれぞれ約 0.2%，1.5%程度であり，十分に低オーバヘッドでの実行状態の保存が実現できた．また，更なる低オーバヘッド化のため，現在取組中であるハードウェアによるサポート手法について述べる．

キーワード：パーシステント・システム，不揮発メモリ，オペレーティングシステム，ハードウェア・サポート

An Examination on A Non-volatile Main Memory System for EnablingBattery-less Systems

Ohmura Ren[1,a)]　Uchikado Yuki[1,b)]

Abstract: This paper illustrates a study to achieve a useful computer system that can recover the running state with no battery unit between unpredictable power failures of unstable power supply . This study assumes the usage of fast non-volatile memories as the system's main memory for decreasing overhead and takes an approach to store and recover the consistent state of system components consisting of the CPU, main memory and peripheral devices. First, the requirements to store and recover the system consistent state are illustrated based on the analogy of consistency of distributed (message-passing) systems. The methods to satisfy the requirements with the modification of the device driver and operating system are described. Evaluations with practical applications showed that the software-based methods have at most 0.2% or 0.5% overhead and store of running state can be done with low overhead. Additionally, a hardware support to reduce overhead still more, which is our work in progress, is illustrated.

Keywords: Persistent Systems, Non-volatile memory, Operating Systems, CPU Architecture, Hardware Support

1. はじめに

ユビキタス社会となった現在において，身の回りでは大型のものから小型のものまで多数の計算機システムが稼働

[1] 豊橋技術科学大学
Toyohashi University of Technology, 1-1, Hibari-ga-oka, Tenpaku-cho, Toyohashi, Aichi 442-8580, Japan
[a)] ren@tut.jp
[b)] uchikado@usl.cs.tut.ac.jp

し，生活を成り立たせるうえで必要不可欠なものとなっている．また，安心・安全な社会を構築するにあたり，環境や人の生活をモニタリングしてセンサデータを取得し，その処理によって効率的な情報提示，危険予測や回避のための情報提供をおこなうための取り組みが行われている．特に，道路の路面や建物の壁といった環境設備や，衣服やメガネなど身に着けるもの，鉛筆や食器などの日常的に使うものといった，今まで計算機能力が存在しなかったようなものにまで，センサやプロセッサが搭載され，情報収集を行うことで人の生活支援，安心・安全の保障を行うための研究が多く行われている．

このような状況において，計算機システムの電源に関する問題は，より重要性を増してくる．計算機システムは電源が無ければ動作しないため，電源の効率的な利用や電源をいかに確保するか，が問題となる．

計算機システムにおける電源の問題に対する対応としての方向性の一つは，システムの低消費電力化である．低消費電力化は，半導体の消費電力を下げることや，計算機システムにおいて使用されていない部分の動作速度を低下，もしくは動作を停止させることで行われる．しかし，現在の計算機システムでは，CPU や主記憶の状態は動作を停止している状態であっても微弱な電力を供給することによって保たれているため，計算機システム全体の電力を停止させることは難しい．ノートパソコンのスリープやスマートフォンなどの一見停止しているように見える状態は，その実通電状態であり，記憶保持のために電力を消費している．つまり，このような電力を削減することができれば，計算機システムのさらなるの大幅な低消費電力化を望むことができる．

もう一つの計算機システムにおける電力の問題に対する対応の方向性は，エネルギーハーベスティング技術と呼ばれる技術である．エネルギーハーベスティング技術は，特に環境モニタリング用のセンサノードのような組み込みシステムを対象とし，太陽光，温度差，振動などによって発電する素子を計算機システムに付加して，その素子で発電した電力を計算機システムの稼働に利用する，という技術である．この技術は有線による電力供給や一次電池の交換なしに組込システムを恒久的に動作させるための技術として期待されている．しかし，エネルギーハーベスティング技術による発電は，基本的に安定した電力とはならない．このため，一度大容量のキャパシタや二次電池に蓄えられてから用いられることが想定されており，充電のための損失や充電回路自体の電力消費量が問題になっている．

そこで我々は，エネルギーハーベスティング技術のような電力供給技術の下において，バッテリレス（一次あるいは二次電池なし）で動作する計算機システムを構築することを目標として，今まで研究をすすめてきた．より具体的には，太陽電池で駆動する「卓上電卓」の感覚で使用可能な計算機システムの構築を目標とし，不安定な電力供給に対応するため，システムは「突然の電源切断」に面したときにもその実行状態を保持し，再度電源が投入されれば高速に復帰して電源切断直前の状態から計算の再開が可能なシステムを構築するための研究を行ってきた [6].

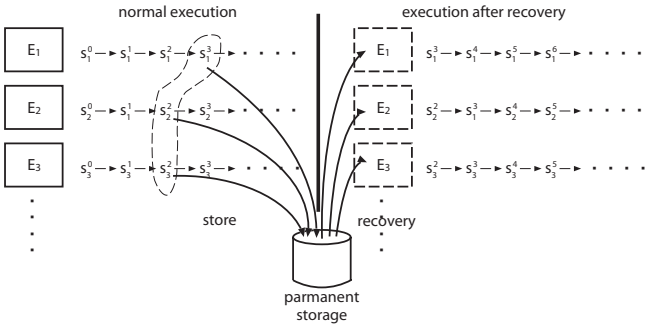

図 1 既存研究における実行状態復元の模式図
Fig. 1 Schema of state recovery on existing studies

1980 年代後半から 90 年代前半にかけて，システムの耐故障性確保のため，大型計算機や分散システムでは「Persistent System」と呼ばれ，システムの実行状態の保存／復元に関する研究が多く行われた [1], [2], [3], [5]．この時，計算機システムのメモリ階層における主記憶は揮発であり，電源切断とともに失われるとされ，主記憶や CPU の実行状態はディスク上に保存された．そして，主記憶の状態の保存／復元がその処理時間における大部分を占めていた．一方，我々の研究では，FeRAM や MRAM などの主記憶に用いることが可能であり，かつ不揮発性を有するメモリ（以下，単に「不揮発メモリ」と呼ぶ）に着目する．主記憶を不揮発とすることで，大幅な状態保存／復帰の高速化を目論むことができる．そして，主記憶を不揮発とする計算機システムにおいて，高速な実行状態の保存／復元を行うためのソフトウェアによる対応手法を提案し，そのような機能をもつオペレーティングシステムの開発を行ってきた．本稿では，まず，その基本的な手法について述べるとともに，ソフトウェアによる具体的な手法について述べる．また，今まで開発された手法はソフトウェアによる手法であったため，少なからずオーバヘッドが生じていた．このため，現在，我々は状態の保存/復元をサポートするための CPU の開発に取り組んでおり，現在キャッシュメモリに対する設計を行っている．本稿では我々が提案する CPU の設計について述べる．

2. 実行状態の復元

2.1 従来研究と本研究における実行状態の保存／復元

前述のとおり，システムの実行状態の復元に関する研究は，従来研究において主にフォールトトレランスを目的として行われてきた．**図 1** に，従来研究のシステムの実行状態の復元を摸式化した図を示す．

図 1 において，各 $E_i(i = 1, 2, 3, ...)$ はシステムを構成する要素を表わす．それぞれの要素 E_i の状態が $s_i^0 \rightarrow s_i^1 \rightarrow s_i^2 \rightarrow ...$ として遷移する最中，ある時点においてそれぞれの状態を永続記憶装置上に保存する．そして，なんらかの障害によってシステムの実行が停止し，現在の各要素の状態が失われた時，保存した状態を各要素に再構築して実行を継続する．これが実行状態の復元である．

これらの作業において，従来研究では，システムを構成する要素として主に CPU と主記憶と状態のみが扱われてきた．つまり，状態保存作業では，CPU のレジスタと主記憶の内容のみが保存され，リカバリにおいては，CPU レジスタの内容および主記憶の内容の復帰が行われた．この理由は，対象とされたのはオペレーティングシステムを含めたシステム全体ではなく，オペレーティングシステム上で動作するプロセスが対象とされることが多かったためである．プロセス内ではオペレーティングシステムによって抽象化されたデバイスが扱われる．このため，扱われたとしても，例えば read や write，lseek といった操作によるファイル内のアクセス位置を示すポインタが状態保存／復元の対象要素として扱われるにとどまる場合が多かった．本研究では，オペレーティングシステムを含めたシステム全体の実行状態を対象とするため，CPU，主記憶，周辺デバイス全ての状態を対象とし，その状態の保存と復元を行う．

また，主記憶は従来，揮発な（電源切断とともに失われる）状態として扱われてきたが，本研究では不揮発メモリを主記憶に使用することを前提とするため，主記憶の状態は不揮発な（電源切断時の状態がそのまま残される）要素，として扱う．一方，CPU の状態については，ゲートレベルで不揮発性を持った半導体はまだ実用化されていないため発な要素として扱う．周辺デバイスについては，ディスク装置などのようにそもそも状態を永続的に保存することを前提としたものもあるものの，むしろネットワークやディスプレイのように状態が保存されない種類のデバイスの方が多く，基本的には「揮発なもの」として扱う．ただし，CPU については，レジスタの容量は主記憶に比較して小容量であり，かつその容量は固定であることから，キャパシタなどを用いて電源切断時の電圧の減衰時間を調整することによって電源切断時にその内容を保存することができると考えられる．そこで，CPU の状態は揮発ではあるものの，電源切断時の状態を復元可能な要素として扱う．また，周辺デバイスについては，の動作が物理的な状況を含めた外部状態に依存する場合があり，電源切断時の状態をそのまま保存することは難しい場合が多い．このため，周辺デバイスは電源切断時の状態を保存可能な要素としては扱わない．しかし，例えばネットワークデバイスにおいてはその初期化コマンドと電源切断時に実行中であったコマンド，ディスプレイについては初期化コマンドと電源切断時に表示されていた内容を復元できれば状態が復元されるように，その状態は揮発であるもののいくつかのコマンドの発行によって復元可能な要素として扱う．

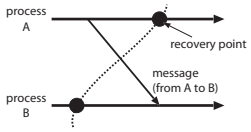

図 2 一貫性が保たれる復元状態
Fig. 2 An example of consistent cut

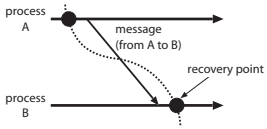

図 3 矛盾を生じる復元状態
Fig. 3 An example of inconsistent cut

このような前提において，CPU，主記憶，周辺デバイスの各状態は自動的にその状態が残されるものと，明示的にその状態を保存／復元しなければならないもの，に分かれることになる．このとき，状態復元に対して「一貫性」を考慮する必要が生じる．つまり，自動的に保存される要素（主記憶）の状態は，他の明示的に保存／復元されなければならない要素（CPU，周辺デバイス）の復元可能な状態と「一貫性のある状態」でなければならい．例えば，プログラムにおいて，デバイスになんらかのコマンドを発行し，そのあとにシステムが停止したような場合，主記憶や CPU の状態についてはコマンドを発行した後の状態が復元され，デバイスについてはコマンドが発行される前の状態が復元されるようでは，状態復元後の正常な動作が望めなくなってしまう．このため，自動的に保存される要素については状況に応じて過去の状態を復元できるよう，自動的に保存されない要素については，他の状態と一貫性のある状態を復元できるよう，それぞれ状態の保存や復元を行う必要が生じてくる．そこで，本研究では分散システムにおける一貫性の議論に着目して，システムの実行状態における一貫性の定義，並びにその時の状態保存／復元方法の要求検討を行う．

2.2 分散システムにおける一貫性

分散システムの一貫性を議論する場合において，分散システム構成する各ノードあるいはプロセスを一つのステートマシンとして扱う．つまり，外部からの入出力や内部でクロックをイベントとして自身の状態を変化していく機械としてモデル化される．このため非常に一般性を有するモデル化がなされ，システムを構成する CPU，主記憶，周辺デバイスの動作間でも同様の議論を適用することができる．

分散システムにおいて，その実行状態を復元する場合，同一時刻において全てのプロセスの状態を保存することができればよい．しかし，これは分散システムでは各プロセ

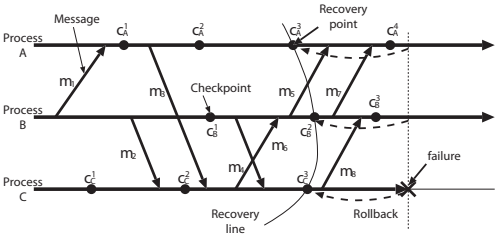

図 4 メッセージパッシングシステムにおける一貫性を維持した状態復元の例

Fig. 4 An example of a consistent cut on a running message passing system

ス間で厳密な時刻同期ができないことと同様の理由で，原理的に不可能である．そこで，Lamport の論理クロックを基に，以下のような形で各プロセスの復元状態について，その一貫性が議論される．

各プロセスにおいて，復元される状態がメッセージを横切る場合を**図 2** および**図 3** に示す．横の矢印は，各プロセスの実行を表わしている．斜めの矢印は，プロセス間でやり取りされるメッセージを表わしており，両図ともプロセス B がプロセス A からメッセージを受け取る状況を示している．図中の黒丸はそれぞれのプロセスの状態が復元される時点を示している．

これらの図において，図 2 は一貫性を保った復元状態となり，図 3 は矛盾が生じる (一貫性が保てない) 復元状態となる [2]．なぜなら，図 3 では，プロセス B についての復元状態がプロセス A が送ったメッセージを反映しているにもかかわらず，プロセス A の復元状態はそのメッセージを送る前の状態となり，これは，現実には起り得ない (Lamport の happend-before 関係が false となる) 状態となるためである．つまり，各プロセスの復元される状態を結んだ線 (図中点線) に着目すれば，この線が「X」の形となる形で交差する場合には一貫性が保たれた状態であり，メッセージを追い越すような形で交差する線は一貫性が保てない状態となる．

なお，図 2 において，復元状態を示す線がメッセージを横切るということは，「メッセージは送られたが，まだ届いていない状態」が復元されるということを意味する．このようなメッセージはイン・トランジット・メッセージ (in-transit message) と呼ばれる．イン・トランジット・メッセージは各プロセスの実行状態の復元後，受信したプロセスに再度渡される必要が生じる．

これらの一貫性の議論をもとに，メッセージパッシングシステムおけるシステム全体の実行状態復元の実際の方法は，大きく「チェックポインティングベースのロールバックリカバリ」と「ログベースのロールバックリカバリ」に分けられる．

チェックポインティングベースのリカバリは，各プロセスで行われるチェックポインティング（その時点の状態を保存する処理）のみを用いる方法である．**図 4** は，チェックポインティングベースのリカバリ手法における実行状態復元の例を示している．前述の例と同様に，各プロセスの実行は横の矢印で表し，メッセージは斜めの矢印で表している．各プロセスのチェックポイントは，図中黒丸で表している．

通常実行中，各プロセスはそれぞれチェックポインティングを行い，その時点のプロセスの状態を保存する．あるプロセスに故障が生じリカバリが必要になったとき，リカバリ作業では一貫性が保たれる各プロセスのチェックポイントを選びだし，その状態まで各プロセスの実行状態を戻す (「ロールバック」と呼ばれる)．図 4 の例では，前述の議論に基づき $(c^3_{\mathrm{A}}, c^2_{\mathrm{B}}, c^3_{\mathrm{C}})$ が一貫性が維持された状態となり，各プロセスについてこれらのチェックポイントの状態が復元されることになる．なお，チェックポインティングベースの手法では，リカバリ作業によって各プロセスが復元されるポイントは「リカバリポイント (recovery point)」と呼ばれる．また，各リカバリポイントを結んだ線は「リカバリライン (recovery line)」と呼ばれる．

ログベースのロールバックリカバリでは，各プロセスで行われるチェックポインティングに加えプロセス間で取り交わされるメッセージの情報をログとして保存する．リカバリ時には，このログを用いてメッセージを再生することにより，チェックポイント後の各プロセスの状態を再構築する．このとき再構築される状態は，やはり上記の議論において一貫性が維持された状態が構築されるよう，再生されるメッセージが選択される．

2.3 実行状態の復元における要求分析

通常，各周辺デバイスはオペレーティングシステム内の対応するデバイスドライバによって集中的に管理・制御される．周辺デバイスはそれぞれ独立に動作し，デバイス間でその状態に依存関係が発生しないとすれば，デバイスドライバとある周辺デバイスの関係は 1 対 1 の関係で考えることができる．実際，DMA や USB コントローラのような特殊なデバイスを除けば，実際のシステム（オペレーティングシステム内での関係）ではこのような関係となる場合が多い．そこで，本研究ではデバイスドライバに注目し，個々のデバイスとデバイスドライバの間で一貫性を維持するリカバリ手法を確立する．そして，その手法をシステムに接続される各デバイスのデバイスドライバにそれぞれ適用することによって，システム全体の状態を復元するアプローチを取る．

デバイスとデバイスドライバの間の一貫性を考えるにあたり，前述の分散システムの一貫性の議論を適用する．デバイスドライバと，周辺デバイスを分散システムを構成するノードとみなし，デバイスドライバ内での周辺デバイスへのアクセスをそれぞれの間で取り交わされるメッセージ

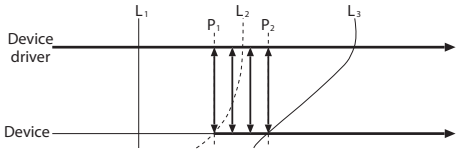

図 5 基本的なデバイスアクセス
Fig. 5 Basic device access

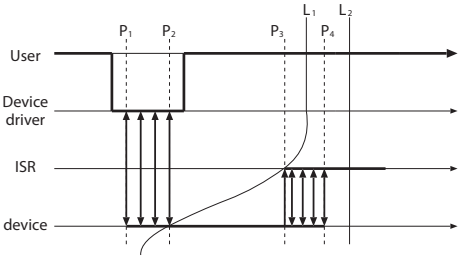

図 6 コマンドに対応する割り込み処理
Fig. 6 Interrupt handling for a request

とみなす．例えば，デバイスへの write 操作 (store 命令) はデバイスドライバからデバイスへ送られるメッセージとみなし，read 操作 (load 命令) はデバイスからデバイスドライバへのメッセージとみなすことができる．この他，デバイスからの割り込みやデバイスが直接的に主記憶へアクセスする場合なども同様である．

このようにすることで，適切な（一貫性を維持した）デバイスとデバイスドライバの関係を定義することが可能となる．すなわち，各実行中の状態において，リカバリラインが，デバイスのアクセスを横切らないよう[*1]にデバイスとデバイスドライバのリカバリポイントを定めれば，一貫性の取れた状態となる．そして，その時の各リカバリポイントを復元できるよう，デバイスドライバについては主記憶と CPU 状態の保存を，デバイスについてはデバイス状態の保存 (チェックポインティングベースの状態復元) や状態復元時に再生するためのデバイスに対するコマンドの保存 (ログベースの状態復元) を行えばよい．

なお，このときデバイスドライバの実行のうち，割り込み処理 (ISR; Interrupt Service Routine) についても一つのプロセスとして，とみなしてモデル化する．なぜなら，割り込み処理は CPU の通常実行に対して非同期に起動され，また，そのコンテキストはデバイスドライバとは独立のものとなるためである．

このようにして解析したときのデバイスドライバとデバイスの動作と代表的なリカバリラインの図を**図 5**〜**図 8** に示す．それぞれ，デバイスドライバからデバイスへコマンドを発行した場合，コマンドに対して割り込みにより完了通知が行われる場合，外部から能動的にイベント（割り込み）が生じる場合，割り込み処理ルーチン内で他のスレッドとの共有領域がアクセスされる場合，の処理を簡単化して表したものである．

[*1] デバイスへのアクセスは前述のように read と write で方向性を持つため，厳密には図 2 のようにアクセスを横切っても良い場合が存在する．しかし，実際には read と write は混在する場合が多く，単純化して検討するため，単に「アクセスを横切らないよう」とした．

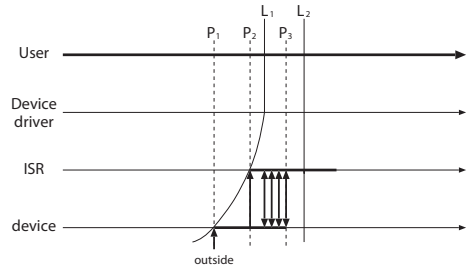

図 7 外部イベントに対応する割り込み処理
Fig. 7 An interrupt handling for an external event

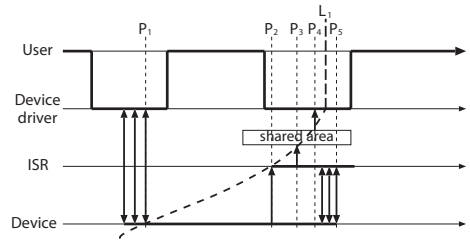

図 8 ISR 内で共有データにアクセスする場合
Fig. 8 A case of accessing shared area from the ISR

詳細は文献 [6] にゆずり本稿では割愛するものの，これらの要求分析から，以下の要求事項が明らかになる．

(1) コマンド発行中のデバイスドライバの状態が復元される場合には，デバイスへのコマンド発行は各アクセス毎にログをとるか，デバイスアクセス前にチェックポインティングを行う
(2) デバイスへ発行される要求（一連のコマンド）は発行される際に保存し，保存されたデバイスへの要求は，その要求の後処理が終了して最初にデバイスドライバの状態が復元可能となる時点で破棄することができる
(3) ISR 内でデバイスへのアクセスが完了する前のデバイスドライバの状態が復元される場合，ISR の状態を割り込み発生以前にロールバックさせる
(4) ISR 内で他の要素との共有領域へ write アクセスを行う場合，デバイスアクセス終了後に行うようにするか，ISR がデバイスアクセス終了までロックを保持し続けるようにする

2.4 状態復元の実現

図 9 にこれまで述べた手法による実行状態復元について，関連するソフトウェアコンポーネントを示す．図中，網掛け部が前述までの議論において，関連するソフトウェアコンポーネントに該当する．boot&recovery はオペレーティングシステムのブート時あるいは復帰時の処理を行うコンポーネントである．Callback Function はシステムの状態復元時に呼び出され，対応するデバイスの状態復元に

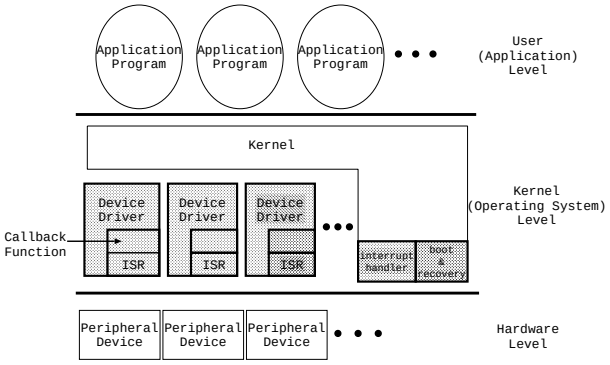

図 9 実行状態復元に関するソフトウェアコンポーネント
Fig. 9 Software components related to system state recovery

責任を持つ処理が記述される部位とする．

また，改変を加えたデバイスドライバのコードは**図 10** に示すようになる．デバイスドライバ内では，以下に述べるような処理を適宜挿入すればよい．追加される処理はマクロ化可能な程度の処理であるため，図 10 内では DEVICE_ACCESS_START のように全て大文字で名前が付けられた関数（マクロ）の形で記述してある．なお，前述のとおり，ある要素の状態を復元するには，チェックポインティングベースのの方法とログベースのの方法があるが，デバイスドライバおよびデバイスの状態を復元する場合にも同様であり，以下，チェックポインティング手法，および，ロギング手法と呼ぶ．

図 10 中，DEV_ACCESS_START および DEV_ACCESS_END (3, 6, 39, 43 行目) は，それぞれてデバイスへのアクセスの開始とデバイスへのアクセスの終了時の状態保存や保存された状態の破棄の処理である (要求事項 (1))．チェックポインティング手法の場合には，DEV_ACCESS_START においてデバイスドライバの主記憶状態，CPU 状態の保存を行い，DEV_ACCESS_END においてその情報を破棄する．この時，主記憶全ての状態を保存する必要はなく，DEV_ACCESS_START と DEV_ACCESS_END によって囲まれた区間で変更される主記憶の状態と DEV_ACCESS_START 時点の CPU 状態が記録されればよい．ロギング手法では DEV_ACCESS_START において特に何も行う必要はないが，tell_device_request 関数内でデバイスへ発行されるコマンドのログを取得し，DEV_ACCESS_END においてその情報を破棄する．

また，デバイスへ発行される要求を保存する処理に対応するのが DEV_REQUEST_SAVE(5, 42 行目) である．また，デバイスへの要求の破棄に対応するのは DEV_REQUEST_DISCARD(35 行目) である (要求事項 (2))．

ISR の状態保存を行う処理に対応するのが ISR_SATE_SAVE(32, 48 行目)，保存された情報を破棄する処理が ISR_STATE_DISCARD(34, 53 行目) である (要求事項 (3))．これらの処理は，デバイスアクセスをチェックポインティング手法でデバイスドライバの状態を復元さ

```
1: void write(void* request){
2:   if(idle == get_device_state()){
3:     DEV_ACCESS_START();
4:     tell_device_request(request);
5:     DEV_REQUEST_SAVE(request_log,request);
6:     DEV_ACCESS_END();
7:   }else{
8:     lock(request_queue);
9:     enqueue(request_queue,request);
10:    request_count++;
11:    unlock(request_queue);
12:  }
13:}
14:void read(void* return_buf){
15:  lock(event_buf);
16:  while(!event_buf_count){
17:    unlock(event_buf);
18:    sleep(read_wait);
19:    lock(event_buf);
20:  }
21:  memcpy(return_buf,event_buf);
22:  event_buf_count--;
23:  unlock(event_buf);
24:}
25:void ioctl(void* new_state){
26:  dev_state = *new_state;
27:  device_state_change(new_state);
28:}
29:void interrupt(){
30:  do{
31:    if(request_done & interrupt_reason()){
32:      ISR_STATE_SAVE();
33:      post_processing_for_request();
34:      ISR_STATE_DISCARD();
35:      DEV_REQUEST_DISCARD(request_log);
36:      lock(request_queue);
37:      if(request_count){
38:        request = dequeue(request_queue);
39:        DEV_ACCESS_START();
40:        tell_device_request(request);
41:        request_count--;
42:        DEV_REQUEST_SAVE(request_log,request);
43:        DEV_ACCESS_END();
44:      }
45:      unlock(request_queue);
46:    }
47:    if(event_arose & interrupt_reason()){
48:      ISR_STATE_SAVE();
49:      lock(event_buf);
50:      data = get(event_buf);
51:      post_processing_for_event(&data);
52:      event_buf_count++;
53:      ISR_STATE_DISCARD();
54:      unlock(event_buf);
55:      if(someone_is(read_wait))
56:        wakeup(read_wait);
57:    }
58:  }while(interrupt_reason());
59:}
```

図 10 デバイスドライバコード例
Fig. 10 An example code of modified device driver

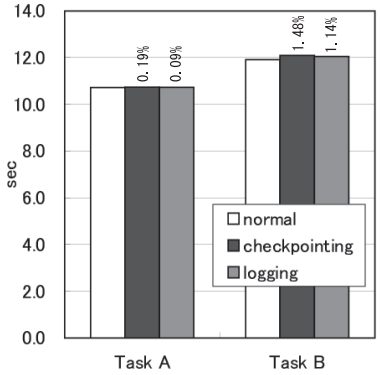

図 11 各タスクの実行時間

せる場合の対応とほぼ同様であるが，復帰後 ISR を再実行する必要はないため，CPU の状態を保存する必要はなく，ISR_SATE_SAVE においてこの区間で変更される主記憶の状態が保存されればよい．また，49 行目と 54 行目のロックの操作は要求事項 (4) に対応し，50 行目で read メソッドとの共有領域である event_buf をアクセスする．このため 49 行目で取得したロックは 54 行目まで保持される．

2.5 ソフトウェアによる手法とその性能

上記のような手法を linux 2.4.4 カーネルおよびそのデバイスドライバに対し適用し，性能評価を行った．CPU はモトローラ社の MPC860(40MHz) を用い，RAMTRON 社製 FeRAM（アクセス速度 70nsec）を用いた不揮発 RAM ボードを作成して主記憶として用いた．デバイスとして，オンチップ上に存在する UART(9600bps) および Ethernet(10Mbps) コントローラを用いた．

まず，システム上でデバイスを利用するプロセスを実行している最中に，故意にシステムの電源の切断／復帰を行いシステムの挙動を観察し，システムが適切に処理を継続することを確認した．

また，性能評価のため，提案手法を用いない場合 (通常実行)，チェックポインティング手法を用いた場合，ロギング手法を用いた場合それぞれにおいて，次の 2 つのタスクを実行した．タスク A では，UART デバイスに対し 1 文字を出力する動作を 10000 回繰り返すプロセスを実行した．タスク B では NFS 上の 2Mbyte のファイルを cp コマンドによりコピーした．コピー先も同様に NFS 上とした．これらのタスクの実行時間を測定結果を**図 11** に示す．タスク A については，0.1〜0.2%程度の性能低下，Task B については 1.1〜1.5%程度のオーバヘッドで実行状態の保存／復元が行えることを確認した．

3. ハードウェアによるサポート

以上，我々は，ソフトウェアによる対応によって突然の電源切断にも対応可能なシステム実行状態保存／復元を実現している．しかし，全てソフトウェアでの実装であるため，少なからずオーバヘッドが生じていた．また，現在までの議論において，主記憶は単に「不揮発メモリで構成」されるということを前提とし，CPU のキャッシュの存在を無視してきた．フォールトトレランス，特にソフトエラーからのリカバリを対象とした従来研究において，キャッシュを効果的に用いて実行状態の復元を行う研究が存在する [1]．キャッシュを効果的に用いたり，必要となるハードウェアの二重化など行うことで，本研究においても，より低オーバヘッドで電源切断への耐性を持ったシステムを構築することが見込める．そこで，我々は現在，不揮発主記憶システムにおいて高速な実行状態復元を行うシステムを，CPU アーキテクチャから見直し，検討を行っている．以下，ハードウェアによる状態保存／復元のサポート，特に現在我々が開発を進めているキャッシュによるサポート手法について述べる．

3.1 基本方針

まず，対象とするシステムに対する前提は 2.1 章で述べた内容をそのまま踏襲するものとする．特に，キャッシュを考慮したとしても，キャッシュを含めた CPU の状態は電源切断時に保存可能（キャッシュの内容は主記憶に全て反映させることが可能）であり，特に周辺デバイスの状態を扱わなければ電源切断時の CPU 状態，主記憶状態はそのまま復帰可能である，とする．

2.4 章にて，デバイスドライバ内の具体的な状態保存の方法を示した．具体的には，DEVICE_ACCESS_START にデバイスドライバ内の状態保存と DEVICE_ACCESS_END における破棄，DEVICE_REQUEST_SAVE によるデバイスへの要求保存と DEVICE_REQUEST_DISCARD による破棄，ISR_STATE_SAVE によるデバイスドライバ (ISR) 内状態の保存と ISR_STATE_DISCARD による破棄であった．このうち，最も時間の取られる処理は DEVICE_ACCESS_START および ISR_STATE_SAVE におけるデバイスドライバ内の状態保存に関する処理である．これらの処理では，その後改変されうる主記憶の状態を保存する．

そこで，主記憶の状態保存について，揮発なキャッシュ領域を活用してオーバヘッドの低減を行う．具体的には，キャッシュメモリを一時的な領域として用い，DEVICE_ACCESS_START と DEVICE_ACCESS_END や ISR_STATE_SAVE および ISR_STATE_DISCARD の間で変更される（リカバリ時に破棄される）内容は揮発なキャッシュ内にとどめておき，電源切断とともにその内容が失われるようにする．このため，DEVICE_ACCESS_START 時，および ISR_STATE_SAVE では，キャッシュのフラッシュ（ダーティビットが立った内容を主記憶に書きだし，主記憶にその内容を反映させる処理）を行うようにする．

3.2 キャッシュのモード切替

この処理を確実に行えるよう, CPU にはキャッシュのモードを制御する命令として，STA(Start Atomic operation) 命令および EDA(End Atomic operation) 命令を設ける．STA 命令は DEVICE_ACCESS_START や ISR_STATE_SAVE として利用されることを想定している．また，EDA 命令は DEVICE_ACCESS_END や ISR_STATE_DISCARD において使用されることを想定している．STA 命令と EDA 命令で囲まれた区間で電源切断が生じた場合には，復帰時に STA 命令時の状態が復元されるように保障するようにする．

STA 命令では，キャッシュのフラッシュおよびオプションの指定によって CPU 状態を保存するとともに，キャッシュのモードを Atomic Operation モードに切り替える．なお，前述のとおり，DEVICE_ACCESS_START で使用される場合は CPU の状態保存を行う必要があり，ISR_STATE_SAVE で使用される場合は CPU の状態保存は行わなくてよい．Atomic Operation モードの時，キャッシュはその置換動作を抑制する．つまり，STA 命令と EDA 命令内で行われた主記憶への操作は，電源切断とともに確実に破棄できるよう，自動的にキャッシュラインの置換が行われないようにする．Atomic Operation モードにおいて，キャッシュの同一エントリ全てがダーティとなり，置換を行わなければならない場合には，CPU はフォールトを起こして対応するハンドラへ飛ぶようにする．このようにすることで，Atomic Operation モード時に電源切断が生じた場合，Atomic Operation に入った直後の状態 (STA 命令が発行された時の状態) が復元され，この時点から実行が再開されるようにする．EDA 命令は，キャッシュのモードを通常動作へ戻し，以降，電源切断時の状態を復元するようにする．

Atomic Operation モードでは，キャッシュの置換を抑制することについて述べた．この場合，例えあばダイレクトマップ方式よるキャッシュなどでは同一エントリに対する競合が生じる可能性が高くなるため，置換に起因するフォールトが頻発することになる．このため，キャッシュとしては，連想度の高いキャッシュやフルアソシアティブ方式が用いられることが望ましい．あるいは，Victim キャッシュ [4] のように，連想度の高いキャッシュと低いキャッシュとを混合させる方式が用いられることが必要となる．

また，単にキャッシュのモードを Atomic Operation モードおよび通常モードに切り替えることについて述べたが，実際のオペレーティングシステム内では複数のスレッドが並行して動作することになる．このため，単純に全てのキャッシュラインを Atomic Operation モードにすることは他のスレッドに悪影響を与える可能性がある．このため，スレッド ID 毎にキャッシュのモードを管理することが必要となる．現在，これらの事項を考慮し，状態復元サポート用 CPU の設計を行っている．

4. おわりに

本稿では，2 次電池を用いることなく，不安定な電源下でもユーザにストレスを与えずに動作する計算機システムを目指し，システムが不意の電源切断に面した状況においても，電源切断時の状態を高速に保存／復帰するシステムの実現に対する取組について述べた．

本研究は低オーバヘッド化のため，主記憶が FeRAM や MRAM などの高速な不揮発メモリで構成されることを前提とし，システムを構成する CPU，主記憶，周辺デバイスについて一貫性を維持する状態の保存／復元を行う手法について述べた．まず，メッセージ・パッシングシステムのアナロジーから導かれる一貫性維持のための要求事項について述べ，オペレーティングシステムおよびデバイスドライバの改変によって要求事項を実現する手法について述べた．

また，現在取組中であるハードウェアによるサポートについて言及し，中でも現在設計を進めているキャッシュアーキテクチャについて述べた．現在，FPGA 上に CPU コアとともに上記に述べたようなキャッシュ機構を実装し，実機での実験環境を構築中である．今後，提案するハードウェアでの動作検証ならびに性能評価を行っていく予定である．

謝辞 本稿で述べた研究の一部は，堀科学芸術振興財団の研究助成によって遂行されている．

参考文献

[1] Nicholas S. Bowen and Dhiraj K. Pradhan. Processor- and Memory- Based Checkpoint and Rollback Recovery. *IEEE Computer*, pp. 22–31, Feb. 1993.

[2] Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.

[3] C. Morin and I. Puaut. A Survey of Recoverable Distributed Shared Memory Systems. *IEEE Transaction on Parallel and Distributed Systems*, Vol. 8, No. 9, pp. 959–969, 1997.

[4] Afrin Naz, Mehran Rezaei, Krishna Kavi, and Philip Sweany. Improving Data Cache Performance with Integrated Use of Split Caches, Victim Cache and Stream Buffers. In *Proceedings of the 2004 workshop on MEmory performance: DEaling with Applications , systems and architecture*, MEDEA '04, pp. 41–48, New York, NY, USA, 2004. ACM.

[5] James S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Univ. of Tennessee Computer Science, 1997.

[6] 大村廉, 山崎信行, 安西祐一郎. 主記憶に不揮発メモリを用いたシステムの実行状態復元手法. 情報処理学会 ACS 論文誌, Vol. 45, No. SIG1(ACS4), pp. 88–99, 1 月 2004.