Vol.2018-OS-142 No.9
2018/2/28

システム障害発生時における障害情報収集と再起動の同時実行による高信頼化

山本 遼介[1,a)]　片山 吉章[2]　水口 武尚[2]　明田 修平[1]　毛利 公一[1]

概要：近年の組込み機器は，ハードウェアの高性能化，アプリケーションの多様化により，汎用 OS を広く採用している．汎用 OS は，他の組込み向け OS と比較し，ソフトウェアとしての規模が大きく，動作が複雑である．そのため，汎用 OS 自身に障害が発生する可能性があるだけでなく，障害発生時の再現が難しいため，その障害原因の特定が困難である．障害の原因を特定するために，障害発生後からシステム再起動前に障害情報を収集する方法がある．しかし，この方法は障害情報の収集に時間を要し，システムの再起動が遅れるという課題がある．本論文では，組込み機器で動作する汎用 OS と収集用 OS を同時に動作させることで，システムの再起動と障害情報の収集を同時に行う手法を提案する．提案手法の実現にあたり，ARM プロセッサの拡張機能である TrustZone を利用する．これにより，収集用 OS を保護された環境（Secure World）で動作させることが可能であり，システム障害発生時でも障害情報を安全に収集できる．また，提案手法を実現することで，障害情報を収集し，再起動に要する時間を短縮できることを確認した．

キーワード：組込みシステム，信頼性，クラッシュダンプ，LPAR

Ryosuke Yamamoto[1,a)]　Yoshiaki Katayama[2]　Takehisa Mizuguchi[2]　Shuhei Aketa[1]
Koichi Mouri[1]

Abstract: Recently, general purpose operating systems (OSes) are widely used in embedded systems to access internet and to execute many applications. System faults are apt to occur on them because they are larger and more complicated than embedded OSes. Furthermore, it is difficult to specify cause of the faults because reproduction of them are difficult. Generally, crash dump is acquired between the crash and rebooting, and is analyzed to find the error. However, this may take a long time to acquire the crash dump. This may be cause of delay of restarting the services that are provided on the system. To solve this problem, we propose a method that reboots a general purpose OS and acquires crash dump at the same time by a "collector OS." To achieve the method, we used TrustZone that is an extension of ARM architecture. By using TrustZone, the collector OS can run in a protected area, that is called "secure world", and can acquire crash dump after the fault. We confirmed that the proposed method can acquire crash dump and reboot faster.

Keywords: Embedded Systems, Reliability, Crashdump, LPAR

1. はじめに

近年，カーナビ，POS システム，スマートフォンなど多くの組込み機器は，アプリケーションの多様化，インターネットへの接続などの理由から汎用 OS を利用している．しかし，汎用 OS は他の組込み向け OS と比較し，ソフトウェアとして規模が大きく，動作が複雑である．それゆえに，OS 自身にバグが潜在し，障害発生の原因となることがある．特に，OS 内のバグとしてデバイスドライバが原因となるものが挙げられる．このような OS 内のバグは機器の機能停止を引き起こし，復旧のために再起動を必要とする場合がある．さらに，汎用 OS の障害は再現が難しく，障害の原因を特定することが困難である．このような障害の原因を特定するために，障害発生後から再起動前にメモリダンプやレジスタダンプといった障害情報を収集する方

[1] 立命館大学
Ritsumeikan University
[2] 三菱電機株式会社 情報技術総合研究所
Information Technology R&D Center, Mitsubishi Electric Corporation
[a)] ryamamoto@asl.cs.ritsumei.ac.jp

法が挙げられる．この障害情報を障害原因の分析に利用し，原因を特定することで，システムの信頼性を向上させる．

既存の障害情報を収集するツール [1][2] は，メモリやCPU から障害情報を収集後，OS を再起動させる．そのため，収集する障害情報の量に比例してサービスの再開が遅れるという課題がある．この理由から，迅速な復旧が求められる組込み機器では，障害情報を収集することを諦めなければならない場合がある．

以上の課題を解決するために，障害情報収集と汎用 OS の再起動を同時に行う手法を提案する．これにより，障害情報を収集しつつ迅速な再起動を実現する．障害情報収集と汎用 OS の再起動を同時に行うために，汎用 OS の外部に収集用 OS を別途動作させ，障害情報を収集する．これにより，汎用 OS の状態に依存しない障害情報収集が可能になり，確実に障害情報を収集できる．さらに，収集用 OS が汎用 OS を再起動させることで，ハードウェアリセットによる障害情報の消去を回避できる．

収集用 OS を動作させる場合，障害が発生した汎用 OS の動作は保証されないため，汎用 OS から収集用 OS を保護する必要がある．この課題を解決するために，ARM アーキテクチャの拡張機能である TrustZone[3] を利用する．TrustZone が提供する保護された空間（セキュアワールド）で収集用 OS を動作させることで，収集用 OS の保護を実現する．一方で，障害情報収集中，汎用 OS は障害情報が残存するメモリ領域を使用できないため，汎用 OS の性能低下が考えられる．これに対して，収集完了したメモリ領域を随時汎用 OS に割り当てることで，汎用 OS の性能低下を最低限に抑える．

提案手法の手順として，障害発生時に汎用 OS は収集用 OS に対して障害の通知を行う．通知を受け取った収集用 OS は，汎用 OS の再起動に必要なメモリ領域のダンプを収集後，収集し終えた領域を用いて汎用 OS を再起動させる．その後，収集用 OS は継続してメモリダンプを収集し，空いたメモリ領域を随時，汎用 OS へ割り当てる．

以上の提案手法により，障害情報を収集しつつ迅速な OS の再起動を実現する．提案手法を実現するために，収集用 OS として椿を実装した．椿は，汎用 OS である Linux と同時に動作させることが可能であり，障害情報収集，Linux の再起動を行う．実装した提案手法を用いて，障害発生時における Linux のダウンタイムの計測を行なった．結果，提案手法の適用により，障害情報収集後に再起動した場合よりもダウンタイムを短縮できていることが確認できた．また，Linux 再起動後，利用できるメモリサイズ，CPU 利用率を確認したところ，提案手法の適用による性能低下を最低限に抑えることができていた．

2. 既存の障害情報収集ツールと課題

2.1 Linux Kernel Crash Dump(LKCD)

既存の障害情報収集ツールとして，Linux Kernel Crash Dump（LKCD）[1] が存在する．LKCD は，Linux において Panic や Oops が発生した際に，クラッシュダンプを一時退避領域（外部ストレージの swap 領域や専用のパーティション）に退避する．その後，Linux を再起動させ，一時退避領域からダンプファイルやシンボルテーブルなどをディスクに保存する．

LKCD は，Linux に標準搭載されている機能を用いてクラッシュダンプを収集する．障害発生により動作が保証されない Linux の機能を用いているため，正確に障害情報を収集できない場合がある．

2.2 kdump

kdump[2] は，障害が発生したカーネルとは別に障害情報を収集するキャプチャカーネルを動作させる．キャプチャカーネルを用いてメモリダンプを収集することにより，LKCD のようにクラッシュしたカーネルがダンプを収集しない．そのため，LKCD と比較して，より確実にダンプを収集することができる．

kdump は，kexec を利用してキャプチャカーネルを起動させる．kexec は，BIOS やブートローダの処理をスキップすることで高速に再起動する Linux の機能である．BIOS やブートローダは，Linux が利用していたメモリ領域を上書きする可能性がある．そのため，kexec を用いることで，クラッシュしたカーネルのメモリ領域にアクセスすることなくキャプチャカーネルを動作させることができる．

LKCD 同様，kdump は障害情報を収集後に Linux を再起動させる．そのため，障害情報の量に比例して Linux の再起動が遅れる課題がある．

3. 提案手法

既存の障害情報収集ツールは，障害情報収集後に OS を再起動させるため，OS の再起動が遅れる課題がある．そこで，障害情報収集と OS の再起動を同時に行うことで，障害情報を収集しつつシステムを復旧させる手法を提案する．本章では，提案手法の実現に向けた課題について述べ，提案手法の処理手順について述べる．

3.1 提案手法の実現に向けた課題

提案手法の概要を**図 1** に示す．提案手法では，障害情報収集と汎用 OS の再起動を同時に行うことで，障害情報収集後に再起動させるよりも高速にシステムを復旧させる．具体的には，汎用 OS と収集用 OS を同時に動作させることで，汎用 OS の再起動と障害情報収集を同時に行う．こ

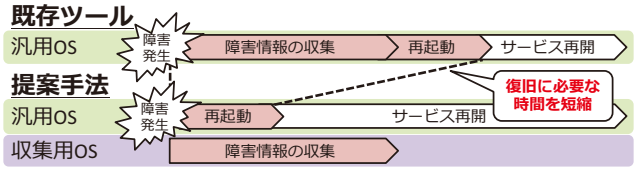

図 1 提案手法の概要

の手法を実現に向けた課題として，(1) 障害情報収集と再起動の同時実行，(2) 複数 OS の同時実行 が挙げられる．

3.1.1 障害情報収集と再起動の同時実行

障害情報収集と迅速な復旧を両立させるために，障害情報収集と汎用 OS の再起動を同時に行う．そのための課題として，(1) 汎用 OS に依存しない障害情報収集，(2) ハードウェアリセットを伴わない再起動 が挙げられる．障害が発生した汎用 OS が障害情報を収集する場合，障害情報の正確性が保証されない．そのため，汎用 OS に依存しない障害情報収集が必要である．一方で，汎用 OS 再起動時にハードウェアリセットを行うことで，デバイスやメモリに残存する情報が消去されるため，障害情報を収集できない課題もある．

以上の課題を解決するために，汎用 OS の外部に収集用 OS を動作させる．収集用 OS は汎用 OS 再起動時に障害情報を収集する．これにより，汎用 OS に依存しない障害情報収集が可能となる．また，収集用 OS が汎用 OS を再起動させることで，ハードウェアリセットなしで汎用 OS を再起動させる．これにより，残存する障害情報を失わずに汎用 OS を再起動させることが可能になる．

3.1.2 複数 OS の同時実行

複数の OS を動作させる技術として計算機の仮想化，Logical Partitioning（LPAR）[4] が挙げられる．仮想化は仮想計算機上でゲスト OS を動作させる手法であり，命令のエミュレーションやデバイスの抽象化によりオーバヘッドが生じる．LPAR は，メモリや CPU などのハードウェア資源を分割して複数の OS を動作させるため，仮想化のようなオーバヘッドは発生しない．そのため，提案手法は LPAR 方式で複数の OS を動作させる．LPAR を採用して複数の OS を動作させた場合，(3) 収集用 OS の保護，(4) 汎用 OS における性能低下の抑制 といった課題がある．

障害が発生した汎用 OS が収集用 OS の資源にアクセスした場合，収集用 OS の処理を妨害する可能性がある．そのため，汎用 OS から収集用 OS を保護する必要がある．収集用 OS を保護するために，収集用 OS と汎用 OS を異なる権限の動作環境で動作させる．収集用 OS を汎用 OS よりも高い権限で動作させることにより，汎用 OS による収集用 OS へのアクセスを制限することができる．また，LPAR では，CPU やメモリを各 OS に分配するため，汎用 OS が利用できる資源が制限され，汎用 OS の性能低下が課題となる．汎用 OS の性能低下を最低限に抑えるため

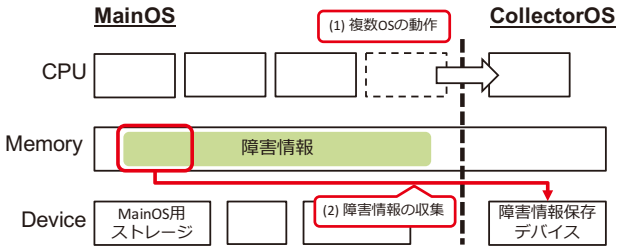

図 2 障害発生時の動作

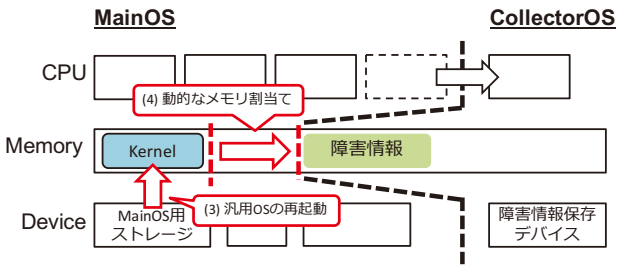

図 3 汎用 OS 再起動時の動作

に，ハードウェア資源の割当てを動的に行う．具体的には，汎用 OS が正常に動作しているときは，汎用 OS に最大限ハードウェア資源を割り当て，障害発生時は 2 つの OS が動作できるように割り当てる．

3.2 処理手順

提案手法では，汎用 OS と収集用 OS を同時に動作させる．提案手法における障害発生時の動作を**図 2**，汎用 OS 再起動時の動作を**図 3** に示す．汎用 OS が正常に動作しているとき，全ての CPU コアやデバイスを汎用 OS に割り当て，収集用 OS が利用しないメモリ領域の全てを汎用 OS に割り当てる．障害発生時の処理内容を以下に示す．

(1) 各 OS にハードウェア資源を分配し動作させる．汎用 OS の障害発生時に，汎用 OS は収集用 OS に対して障害の発生を通知する．
(2) 収集用 OS は，汎用 OS が利用していたメモリ領域からメモリダンプの取得と障害情報保存デバイスへの保存を開始する（1 次ダンプ収集）．
(3) 汎用 OS の再起動に必要な容量分のメモリダンプの取得・保存が完了後，そのメモリ領域を用いて汎用 OS を再起動させる．メモリダンプの取得・保存も継続する（2 次ダンプ収集）．
(4) 汎用 OS の再起動中もメモリダンプの収集が完了した領域を随時汎用 OS に割り当てる．

以上の手順で，メモリダンプを取得しつつ汎用 OS を再起動させ，障害発生前と同等の状態へと回復させる．

4. 提案手法の実装

本章では，提案手法の実装について述べる．動作環境を**表 1** に示す．実装環境として，TrustZone をサポートし

表 1　動作環境

項目	内容
ボード	Jetson TK1
CPU	Cortex-A15 4Core
汎用 OS	Linux 3.10.24（NVIDIA 社による改変あり）
収集用 OS	椿

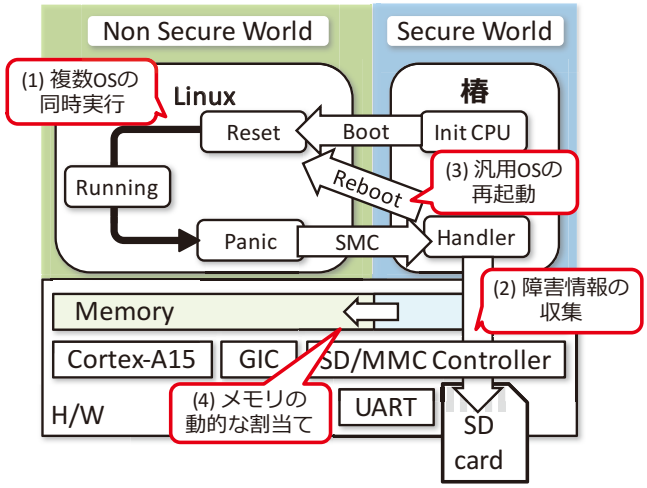

図 4　ソフトウェア構成

た Cortex-A15 を搭載したボードである Jetson TK1[5] を使用した．また，汎用 OS は，組込み機器で広く採用される Linux を採用した．また，収集用 OS として自製 OS の「椿」を実装することで，提案手法を実現する．

本実装におけるソフトウェア構成を**図 4** に示す．前章で述べた提案手法の処理手順における (1) 複数 OS の動作，(2) 障害情報の収集，(3) 汎用 OS の再起動，(4) 動的なメモリの割当てについて実装した．本章では，それぞれの実装について述べる．

4.1　複数 OS の動作

提案手法の実現にあたり，複数 OS の同時実行，収集用 OS の保護を実現する必要がある．収集用 OS の保護を実現するために，収集用 OS を汎用 OS よりも高い権限で動作させる．具体的には，ARM アーキテクチャの拡張機能である TrustZone を利用する．TrustZone は，ソフトウェアの動作環境としてセキュアワールドとノンセキュアワールドを提供する．セキュアワールドはノンセキュアワールドよりも高い権限を持ち，ノンセキュアワールドから保護された空間である．セキュアワールドで収集用 OS を動作させることで，収集用 OS の保護を実現する．

提案手法では，LPAR 方式で汎用 OS と収集用 OS を同時実行する．具体的には，CPU，メモリ，割込み，デバイスを分配する．また，OS 間で連携をとるために，OS 間通信も必要である．以下，TrustZone を用いた LPAR の実装について述べる．

4.1.1　CPU の分配

TrustZone を利用して収集用 OS と汎用 OS に CPU を分配する．TrustZone による拡張が実装されている ARM プロセッサは，セキュアワールドかノンセキュアワールドの状態を持つ．椿はセキュアワールドの CPU を利用し，Linux はノンセキュアワールドの CPU を利用する．電源投入時，ARM プロセッサはセキュアワールドで動作する．そのため，椿が先に起動し，Linux をノンセキュアワールドで起動させる．また，Linux をマルチコアで動作させるために，椿が Application Processor（AP）の起動をフックし，初期化したあとで Linux に渡す．

4.1.2　割込みの分配

割込みコントローラ（GIC）のグルーピング機能を用いて各 OS へ割込みを分配する．グルーピングは，割込みをグループ 0，グループ 1 に分ける機能である．グループ 0 はセキュアワールドで利用され，グループ 1 はノンセキュアワールドで利用される．グループ 0 のみ，割込みをすべて FIQ（高速割込み）として利用することができる．これにより，セキュアワールドで動作する椿は FIQ を利用し，ノンセキュアワールドで動作する Linux は IRQ（通常割込み）を利用するといった割込みの分配が可能となる．

4.1.3　メモリの分配

複数の OS を動作させるために，各 OS にメモリを割り当てる．具体的には，Linux が利用するメモリ領域を制限し，空いたメモリ領域を椿が利用する．Linux のメモリ領域を制限するために，Linux が利用するメモリ領域の開始アドレス，サイズを指定する必要がある．

開始アドレスはデバイスツリーで指定する．デバイスツリーには，物理アドレス空間上のメモリがマップされる場所を記述することができ，メモリの開始アドレスを変更することで，Linux が認識するメモリの開始アドレスを指定することができる．また，メモリサイズは，Linux のブートパラメータである mem パラメータで指定する．

4.1.4　デバイスの分配

各 OS で使用するデバイスが競合しないように Linux が使用するデバイスを制限する．Linux 再起動中，椿はクラッシュダンプを収集するために，SD カードのホストコントローラである SD/MMC Controller を使用する．そのため，Linux が SD/MMC Controller を利用できないように制限する必要がある．そこで，デバイスを登録する処理をスキップすることで，Linux が SD/MMC Controller を認識しないようにした．

4.1.5　OS 間通信

提案手法を実現するために，OS 間での通信を行う必要がある．OS 間でデータを渡す方法として，共有メモリを利用する方法が考えられるが，障害発生時に共有メモリ内のデータの正確性が保証されない．共有メモリを利用しない OS 間通信方法として，SMC 命令を実行することで障害発生を通知する方法を提案する．SMC 命令は，ノンセキュアワールドからセキュアワールドに遷移する命令である．

しかし，SMC 命令はセキュアワールドからノンセキュアワールドに対して制御を移すことができない．そのため，椿から Linux の呼出しは Software Generated Interrupt（SGI）を用いて行う．SGI は CPU 間通信として利用される割込みであり，セキュアワールドの CPU からノンセキュアワールドの CPU に対しても通信可能である．しかし，異なる CPU への通信であるため，レジスタによる値渡しができない．そのため，Linux が占有する CPU が SGI による割込みを受信した後，SMC で椿に制御を移し，再度 Linux 側の制御に移る際に値を渡す．これらを実装することにより，OS 間での相互通信が可能となる．

4.2 障害情報の収集

障害情報をより確実に収集するために，汎用 OS に依存しない障害情報収集が必要である．これを実現するために，収集用 OS がクラッシュダンプを取得し，障害情報保存デバイスに対して書込みを行う．本実装では，SD カードを障害情報保存デバイスとして採用し，椿にメモリダンプを書き込む機能を実装した．

ストレージへのデータ転送方式として，ポーリングと DMA が存在する．ポーリングは，データ転送中に CPU コアを 1 コア分占有し続けるため，汎用 OS の性能低下の原因となる．そのため，本実装では，DMA によるメモリダンプ収集機能を実装する．具体的には，汎用 OS 再起動中に，DMA 転送を行い，SD カードからの割込みに対して椿が処理を行う．これにより，障害情報収集による CPU コアの占有は，割込み発生時のみとなる．

4.3 汎用 OS の再起動

汎用 OS の再起動を行う際，障害情報を保持するためにハードウェアリセットを利用できない．そのため，収集用 OS による汎用 OS の再起動処理を実装する必要がある．ハードウェアリセットを伴わずに OS を再起動させるために必要なことは，ソフトウェアによるハードウェアの初期化である．具体的には，CPU，割込みコントローラ，デバイスの初期化が必要である．

CPU の初期化として，Linux が使用する MMU の無効化，キャッシュのフラッシュ，Linux を BSP で起動させるための CPU コア切替えの 3 つの処理を行う必要がある．これらの処理を行わない場合，Linux の起動に失敗する．また，割込みコントローラの初期化として，各割込みの状態の初期化，各割込みの無効化を行う必要がある．

デバイス初期化の必要性は再起動対象の OS が利用するデバイスドライバに依存する．デバイスドライバが以下の条件を満たす場合，OS の再起動前にデバイスの初期化を行う必要がある．

(1) デバイスの初期状態が想定外 (正常に動作しない状態) であることを検知していない
(2) デバイスの初期状態が想定外であることを検知し，起動を中止する

（1）はデバイスが想定外の状態であり，ドライバがこれを検知しなかったため，OS が正常に動作しなくなる場合がある．（2）は，デバイスドライバがレジスタ値を読み，デバイスが想定外の状態であることを検知した際にパニックを発生させる場合である．実装環境では，（1）が 1 箇所，（2）が 2 箇所存在しており，いずれも Clock and Reset Controller のレジスタ値を初期化することで対処できた．

4.4 動的なメモリ割当て

障害情報収集と汎用 OS の再起動を同時に行なった際に，汎用 OS は障害情報収集中のメモリ領域を使用できない．そのため，汎用 OS が使用できるメモリ領域が制限される課題がある．提案手法では，障害情報を収集し終えたメモリ領域を汎用 OS に割り当てることでこの課題を解決する．

4.4.1 動的なメモリ割当て処理の設計

提案手法では，Linux 再起動時に利用できるメモリ領域を制限し，椿からのメモリ利用許可を受信した際に当該メモリを割り当てる．よって，この処理を実現するために，指定したメモリ領域の制限・割当て処理が必要である．既存の動的なメモリ割当て手法としてメモリホットプラグがある．メモリホットプラグは，指定したメモリ領域について制限・割当てができる．しかし，実装環境では，メモリホットプラグを利用できなかったため，メモリホットプラグと同様の方法でメモリ割当て処理を実装する．具体的には，page 構造体の reserved フラグと free リストを操作することでメモリ割当て・制限を行う．page 構造体の reserved フラグをセットすることで，カーネルが当該ページを利用しないようになる．また，free リストにはアプリケーションに割り当てる候補になるページが登録される．

メモリホットプラグ方式では，制限できないメモリ領域が存在する．制限できないメモリ領域は Linux によって内容が上書きされるため，Linux 再起動後に収集できない．制限できないメモリ領域として，（1）カーネルに予約されたメモリ領域，（2）フレームバッファ が存在する．（1）のメモリ領域は memblock 構造体で管理されており，予約された領域を取得することができる．（2）のフレームバッファはディスプレイに描画されている内容を格納しているため，障害情報として収集する必要がない．そのため，（1）の領域のみ 1 次ダンプ収集で収集することで対処する．

4.4.2 実装

処理の手順を図 5 に示す．Linux 初回起動時，Linux はあらかじめ予約された領域を伝える．椿は，この予約領域を記憶し，Linux に対して利用可能なメモリ領域を伝える．障害発生後から Linux 再起動前に，椿は 1 次ダンプ収集を行う．1 次ダンプ収集では，Linux 初回起動時に伝えられた予約領域と Linux が起動直後に利用する領域のメモリダ

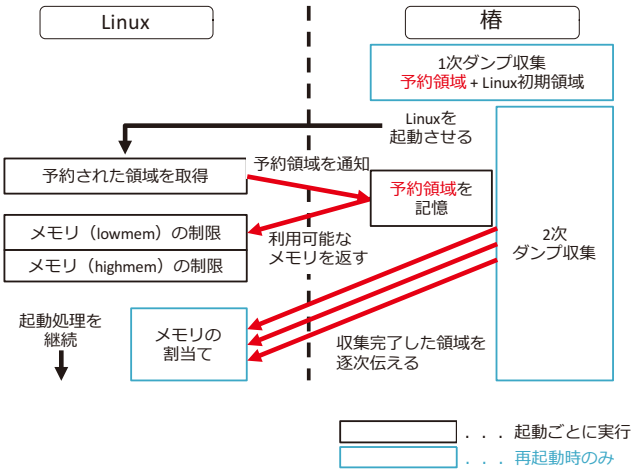

図 5 動的なメモリ割当て処理手順

ンプを収集する．その後，椿は Linux を再起動させ，2 次ダンプ収集を行う．再起動時，メモリダンプを収集し終えた領域のみ，Linux が利用できる．Linux は許可されたメモリ領域内で動作するために，メモリの制限を行う．その後，一定量のダンプ収集ごとに椿がメモリ割当て許可を行い，Linux は許可されたメモリ領域を割り当てる．

5. 評価

前章で述べた提案手法の実装を用いて評価を行なった．提案手法は，障害情報収集と汎用 OS の再起動を同時に行うことで，障害情報を収集しつつ迅速なシステム復旧を実現する．汎用 OS のダウンタイムを計測することで，障害情報収集後に汎用 OS を再起動させるよりも高速にシステムが復旧できることを示す．また，提案手法を適用した際の汎用 OS の性能低下を確認するために，再起動時の汎用 OS が利用できるメモリサイズ，CPU 利用率に関する評価を行なった．

5.1 汎用 OS のダウンタイム

汎用 OS のダウンタイムを計測することで，提案手法の適用により，障害情報収集後に再起動するよりも早く汎用 OS を再起動できることを示す．提案手法を適用した際の汎用 OS のダウンタイムは，1 次ダンプのサイズに依存する．そこで，1 次ダンプのサイズを変えながら，汎用 OS のダウンタイムを計測した．

Linux が利用できる最大メモリサイズを 1024MB に設定し，1 次ダンプのサイズを 0MB（障害情報を収集しない），128MB，256MB，512MB，1024MB（障害情報収集完了後に再起動）の 6 通りに設定し，ダウンタイムを計測する．

具体的には，**図 6** に示す 4 点における時刻を椿が取得する．それぞれの区間に要した時間を計測することで，ダウンタイムとその内訳を示す．また，提案手法を適用せず，Linux 単体での再起動完了までの時間も計測する．

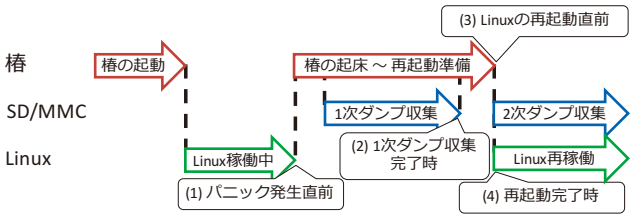

図 6 ダウンタイム計測タイミング

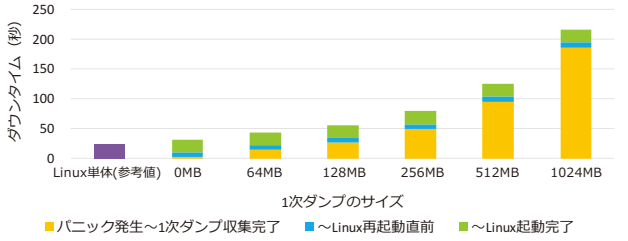

図 7 各メモリサイズにおけるダウンタイム

5.1.1 評価結果

評価結果を**図 7** に示す．

提案手法を適用した場合の各ダウンタイムから．1 次ダンプのサイズに比例してダウンタイムが増加していることが確認できた．そのため，障害情報収集後に再起動させた場合のダウンタイムが最も長い．この結果から，提案手法を適用することで，障害情報収集後に再起動するよりも早く汎用 OS を再起動させることを確認できた．

5.1.2 考察

評価結果より，提案手法を適用した場合のダウンタイムは 1 次ダンプのサイズに比例することが確認できた．ダウンタイムを短縮するためには，1 次ダンプのサイズを減らす必要がある．提案手法を適用した場合，再起動時に Linux が利用できるメモリ領域を制限することで，1 次ダンプのサイズを減らすことができる．そのため，Linux 再起動直後に必要なメモリサイズが小さいほど，ダウンタイムが短くなる．また，Linux 再起動後に 2 次ダンプ収集を行うため，提案手法を適用した場合のダウンタイムは，2 次ダンプのサイズに依存しない．これらより，収集したクラッシュダンプの総量に関わらず，1 次ダンプのサイズが小さければ，汎用 OS のダウンタイムを短縮できる．

5.2 汎用 OS 再起動時に利用可能なメモリサイズ

提案手法は，汎用 OS が再起動直後に利用するメモリ領域を制限し，障害情報を収集する．そのため，再起動直後の汎用 OS は，利用できるメモリ領域が制限され，性能が低下する．提案手法では，汎用 OS の性能低下を最低限に抑えるために，収集完了したメモリ領域を汎用 OS に割り当てる．本節では，Linux 上で動作するプロセスが動的に割り当てられたメモリ領域を逐次利用できることを確認し，メモリ制限による性能低下を最低限に抑えられていることを示す．

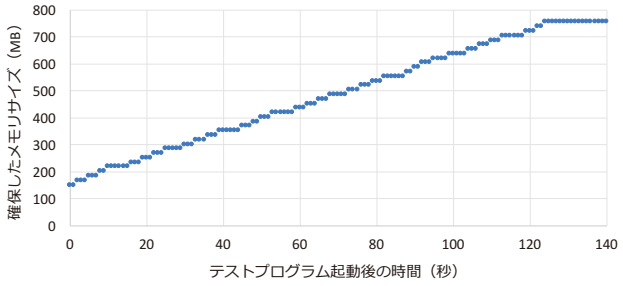

図 8 テストプログラムが確保できたメモリサイズ

5.2.1 評価方法

再起動した Linux 上でテストプログラムを動作させることで評価を行う．テストプログラムは，malloc 関数を用いてメモリの確保を繰り返し行うことで，逐次確保可能なメモリを全て確保する．また，1 秒ごとに確保できたメモリサイズを出力する．これにより，テストプログラムの出力は Linux 上で動作するプロセスが確保可能な最大メモリサイズを示す．

評価手順として，Linux の初期メモリサイズを 64MB，最大メモリサイズを 1024MB に設定し，Linux を再起動させる．Linux 再起動後，テストプログラムを動作させ，確保できたメモリサイズを確認する．

5.2.2 評価結果

テストプログラムが確保できたメモリサイズを**図 8** に示す．テストプログラムは，実行直後に 64MB 以上のメモリを確保できている．これは，Linux 再起動処理中であっても，収集用 OS がメモリの割当て許可を行なっているためである．テストプログラム起動後もメモリの割当て処理が行われており，時間経過と共にメモリが確保できていることが確認できる．テストプログラム起動から 124 秒ほどで全メモリの割当てが完了しており，その後，確保したメモリサイズが変動していない．この結果より，汎用 OS 上で動作するプロセスが，動的に割り当てられたメモリ領域を利用できていることが確認できる．よって，汎用 OS のメモリ領域の制限による性能低下を最低限に抑えられている．

5.2.3 考察

図 8 より，テストプログラムがメモリを確保した直後の数秒間，確保可能なメモリサイズが変動しておらず，グラフが階段状になっていることが確認できる．これは，椿によるメモリ割当て許可の間隔が数秒であることを示している．この数秒の間，メモリダンプが収集されているにも関わらず，Linux に割り当てられていないメモリ領域が存在する．これに対して，メモリ割当て許可通知の粒度を細かくすることで，より効率的にメモリを利用できることが考えられる．しかし，メモリ割当て許可の頻度を高めることで，Linux 上で動作するアプリケーションの CPU 利用率が低下することが考えられる．そのため，提案手法を適用するシステムごとに通知の頻度を検討するべきである．

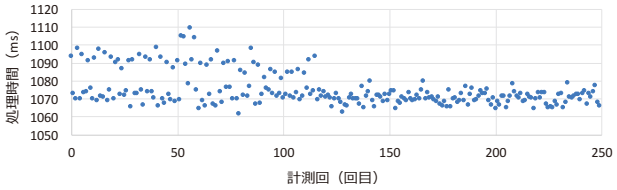

図 9 1 億回のインクリメントに要した時間

5.3 CPU 性能評価

汎用 OS 再起動時に，収集用 OS は障害情報を収集し，汎用 OS に対してメモリ割当て許可を通知する．また，メモリ割当て許可を受信した汎用 OS は，アプリケーションがメモリを利用できるように割り当てる．これらの処理により，汎用 OS 上で動作するアプリケーションの CPU 利用率が低下する．そこで，提案手法適用時の CPU 利用率の低下がどの程度か確認する．

5.3.1 評価方法

メモリサイズの評価と同様に，再起動した Linux 上でテストプログラムを動作させることで評価を行う．評価方法として，Linux を 1 コアで再起動させ，テストプログラムを動作させる．テストプログラムは，1 億回のインクリメントを繰り返し行い，インクリメント完了までの経過時間を保持する．テストプログラムは一定回数の計測を行なったあと，各回の計測結果を出力する．また，このテストプログラムは障害情報収集完了後も動作させる．これにより，提案手法の適用による CPU 利用率の変動を確認できる．

5.3.2 評価結果

各計測において 1 億回のインクリメントに要した時間を**図 9** に示す．図 9 から，おおよそ 116 回目の前後でアプリケーションのパフォーマンスに差があり，116 回目以前の計測結果にばらつきがあることが確認できる．これは，一回の計測中におけるメモリ割当て処理の有無が影響していることがわかる．この結果から，116 回目の計測前後でメモリの割当て処理が終了していることがわかる．116 回目以前の平均処理時間は 1080.035ms であり，116 回目以降の平均処理時間は 1070.720ms であった．この結果から提案手法の適用による性能低下は 0.87%程度である．

5.3.3 考察

Linux 再起動時，提案手法による追加処理は，(1) SD ホストコントローラからの割込みの受信，(2) Linux によるメモリ割当て処理の 2 種類である．Linux 再起動後，メモリ割当てを行なった際に，これらの処理によって発生する平均オーバヘッドを計測したところ，**表 2** に示す結果となった．(1) は DMA 転送による割込みと転送完了割込みの 2 種類があり，前者は特に処理を行わないため，12μs 程度のオーバヘッドが発生する．後者は次の DMA 転送要求やメモリ割当て許可を行なっており，4313μs 程度のオーバヘッドが生じる．(2) は Linux 内部でメモリ割当て処理を行なっており，この処理によるオーバヘッドを計測した結

表 2　提案手法の処理ごとのオーバヘッド

処理	オーバヘッド（μs）
DMA 転送に伴う割込み	12
転送完了割込み	4313
動的なメモリの割当て	14817

表 3　提案手法の実現に必要な Linux の改変量の内訳

機能	行数
OS 間通信	65
障害の通知	3
デバイスの制限	13
動的なメモリの割当て	213
計	294

果，14817μs 程度であることが判明した．これらの時間と処理回数から概算したところ，提案手法による CPU 占有時間は 0.75%程度であった．評価結果の性能低下とほぼ同程度であるため，評価結果が妥当であると考えられる．

6. 今後の課題

6.1 OS 間でのデバイス共有

現在の提案手法では，障害情報を収集するための専用デバイスが必要である．そのため，提案手法を適用した場合，機器に障害情報収集用デバイスを搭載する必要があり，コストがかかる．この課題を解決するためには，OS 間でデバイスを共有する仕組みが必要である．

OS 間でデバイスを共有する方法として，片方の OS に処理を依頼する方法が考えられる．この方式では，汎用 OS，収集用 OS それぞれに専用のデバイスドライバが必要であり，提案手法の適用コストが高い．別のデバイス共有方法として，デバイスドライバを切り替えながら利用する方法が考えられる．具体的には，デバイスに対するアクセスを各 OS のドライバで排他制御することで実現する．この方法も各 OS のドライバに改変が必要であるが，比較的低コストであることが考えられる．

6.2 汎用 OS の変更量

提案手法の実現にあたり，汎用 OS である Linux を改変する必要があった．汎用 OS の改変量を減らすことで，提案手法の適用コストを減らすことができる．**表 3** に提案手法の実現に必要な Linux の改変量の内訳を示す．

表 3 から，動的なメモリ割当てを実現するための変更量が最も大きいことが確認できる．提案手法は，メモリホットプラグと同等の方式で実装していることから，この機能が利用できる環境の場合，改変量が少なくなると考えられる．

7. 関連研究

提案手法では，複数の OS を同時実行させることで，組込み機器の信頼性を向上させる．本章では，信頼性向上，複数 OS の同時実行に関する関連研究について述べ，それぞれの課題を示す．

7.1 信頼性向上

Orthros[6] は，障害発生後，プロセスを障害発生前の状態に復帰させ，継続して動作させる．Orthros は，ActiveOS と BackupOS の 2 つの OS を動作させる．BackupOS は，常に ActiveOS を監視し，ActiveOS に障害が発生した場合に，自動的にフェイルオーバを開始する．フェイルオーバ時にプロセスやファイルキャッシュを BackupOS にマイグレーションすることで，障害発生時にプロセスの処理を継続させる．このアプローチでは，プロセスの動作を障害発生後も継続させることができる．しかし，障害の発生原因を突き止めることができず，同様の障害が繰り返し発生する可能性がある．

リブートは，ソフトウェアを起動時の状態に戻すため，バグ復旧方法として有効である．Phase-based Reboot[7] は，OS の障害発生時，リブートにより障害から復旧させる．OS のブートシーケンスを 3 段階のフェイズに分割し，それぞれのフェイズでスナップショットをとる．このスナップショットを用いて OS をリブートをさせることで，ブートシーケンスの一部を省略してリブートできる．Phase-based Reboot は，VM のスナップショット機能を利用しているため，VM 上で動作する OS に対して有効である．そのため，動作させるホスト OS において，仮想化による実行時のオーバヘッドが発生する．

7.2 複数 OS の同時実行

OS を複数動作させる一般的な技術として計算機の仮想化 [8][9] が挙げられる．仮想化は，Virtual Machine Monitor(VMM) が Virtual Machine(VM) と呼ばれる計算機をエミュレーションし，VM 上で OS を動作させる．複数の VM を用意することで，複数の OS を動作させることが可能である．しかし，仮想化は命令をトラップし処理を行うため，オーバヘッドが発生する．

Logical Partitioning(LPAR) は，ハードウェア資源を各 OS に分割し，複数の OS を動作させる技術である．SoftwareLPAR[4] は，プロセッサのアーキテクチャ依存の機能を利用せず，ソフトウェアにより，メモリや CPU などのハードウェア資源を各 OS ごとに分割している．この手法では，OS が直接ハードウェア上で動作するため，仮想化によるオーバヘッドがほとんど発生しない．しかし，ハードウェア資源が各 OS に固定的に割り当てられる．そのため，各 OS の処理の優先度が動的に変わる場合，動的にハードウェア資源を割り当てることができないため，効率的にハードウェア資源を利用できない．

8. おわりに

本論文では，障害情報収集と OS の再起動を同時に行うことで，組込み機器の信頼性を向上させる手法について述べた．組込み機器の信頼性を向上させるために，障害情報の収集が有効であるが，既存の障害情報収集ツールは，障害情報収集に時間を要しシステムの復旧が遅れる課題があった．そこで，汎用 OS と収集用 OS を動作させ，障害情報収集と OS の再起動を同時に行う手法を提案した．

評価では，提案手法を適用することで，障害情報収集後に汎用 OS を再起動させた場合よりも，汎用 OS を高速に再起動できることを示した．また，汎用 OS へメモリを動的に割り当てることができており，汎用 OS の CPU 利用率の低下が約 0.87%であった．このことから，提案手法による汎用 OS の性能低下を最低限に抑えることができていた．

参考文献

[1] LKCD: `http://lkcd.sourceforge.net/` (2005).
[2] Goyal, V. et al.: Kdump, A Kexec-based Kernel Crash Dumping Mechanism (2005).
[3] ARM: *ARM Security Technology Building a Secure System using TrustZone Technology* (2009).
[4] Shimosawa, T. et al.: Logical Partitioning without Architectural Supports, *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pp. 355–364 (2008).
[5] NVIDIA Corporation: Jetson TK1 組み込み開発キット，, 入手先 ⟨http://www.nvidia.co.jp/object/jetson-tk1-embedded-dev-kit-jp.html⟩ （参照 2018-2-5）.
[6] 吉田健二ほか：プロセスの耐障害性向上のための多重 OS の開発と評価，情報処理学会論文誌コンピューティングシステム（ACS），Vol. 7, No. 2, pp. 11–24 (2014).
[7] Yamakita, K. et al.: Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery, *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pp. 169–180 (2011).
[8] VMware: `http://www.vmware.com/jp` (2015).
[9] Barham, P. et al.: Xen and the Art of Virtualization, *SIGOPS Operating Systems Review*, Vol. 37, No. 5, pp. 164–177 (2003).