修 士 論 文

疎結合マルチプロセッサシステム上への APR技術の実装に関する研究

指導教官　片山卓也 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

許 修寧

2000 年 2 月 15 日

目 次

図 目 次

第 1章

はじめに

1.1　研究の背景と目的

計算機システムが我々の生活に直接関係するような状況で使用される場面が増加している。特に計算機システムが我々の生命に直接関係するような状況で使用される場面が増加している。例えば、病院で患者を監視するシステム、原子力発電所の制御システム、飛行制御システム、交通制御システムなどがある。そして、生命には直接関係ないが、電話システム、銀行のオンラインシステムなども我々の生活に大きな影響を与えるような場面での利用が増加している。

上記のような状況で使用される計算機システムは、計算機システム内部で障害が発生してもシステム全体としては利用者から要求されている仕事を確実に遂行する、すなわち、耐故障性を持つことが要求される。

計算機システムの信頼性を高める為の技術にはフォールトを作り込まないようにするフォールトアボイダンス技術と、フォールトが発生しても提供するサービスにその影響が現れないようにするフォールトトレランス技術に大きく分けられる。フォールトアボイダンス技術に関しては現在までに色々な研究がなされているが、フォールトのない完全なシステムを作成することは特に大規模なシステムでは極めて難しい。

計算機システムの信頼性を高める研究は、特にハードウェアの分野で様々な研究が行われてきており、一部は実現され、我々はすでに多くの場面でこれらの恩恵を受けている。しかし計算機システムは年々大規模化、複雑化する傾向にあり、それとともに、量的にも、質的にもソフトウェアへの依存が増してきている。そこでソフトウェアによる信頼性に関する要求が増加している。

ソフトウェアによって計算機システムの信頼性を高めるための手法に関しては、計算機

システムが仕事を遂行できなくなるということがないように、正しく動作するソフトウェアを作ることによって信頼性を向上させるという研究が数多くなされている。しかし、計算機システム内部で障害が発生してもシステム全体としては利用者から要求されている仕事を遂行する、すなわち、耐故障性を持つためのソフトウェアに関する研究は、比較的新しい分野であり、まだ十分になされているとは言えない。

ソフトウェアによって計算機システムの信頼性を高めるための手法の中で、重要なものの一つとして複製技術がある。複製技術はその実装が容易であることから最も多く用いられている。

この複製技術の中で並列度の高いモジュール複製技術である APR(Active Parallel Replication) 技術が Cherif[1] によって提案されている。APR 技術は複製によってモジュールの冗長性を高めると同時に、システムに存在する並列性を利用して高速な計算を可能にするという特徴を持っている。

APR 複製技術についてはその理論的な定義が完成し、複製技術としての一般的なパフォーマンスの良さは知られているものの、特定の場合に起りうる問題とその解決法についてはまだ一般的な議論にとどまっており、その具体的な問題と解決法については検討されていない。また、APR 複製技術は疎結合マルチプロセッサシステム上への実装の容易性が指摘されているものの、その実装のためのアーキテクチャに関しては一般的な議論にとどまっており、設計および実装の具体例は与えられていない。

そこで本研究では、APR 複製技術で起りうる問題について議論を行った上でその解決法として APR 複製技術の拡張を行う。そしてその有効性について実例をあげて説明を行う。

さらに、APR 複製技術を疎結合マルチプロセッサシステム上への実装を考慮した設計を行う。そして関数型言語である Objective Caml を用いて実装を行う。

1.2 論文の構成

まずはじめに、信頼性、耐故障性などの用語の定義と、耐故障性を高めるための既存の手法と、APR 実装の基盤として用いている計算モデルである FTAG と APR 複製技術について第 2 章で説明する。第 3 章では本研究の話題の中心となる APR 複製技術が抱えている問題点と新しく提案するその解決方法について詳しく説明する。第 4 章では新しく提案する方法を実例をあげて説明を行う。第 5 章では APR 複製技術の実装について説明する。ここでは設計について詳しく説明する。

第 2章

耐故障性を実現するための技術

この章では本研究の話題の基本となる FTAG 計算モデルと APR 複製技術について説明する。はじめに、フォールト、誤り、障害など耐故障性に関する用語の定義から説明する。

IFIP ワーキンググループ 10.4 では、計算機システムの障害 (フォールト) について以下のように定義している [2]。本研究でもこれらの定義を用いる。

> 計算機システムが、利用者が期待する動作を行わないとき、正確にはシステムに対して定められた仕様から逸脱した動作を行ったときに、この計算機システムには障害 (*Failure*) が発生したという。誤り (*Error*) とは、障害をもたらす可能性があるシステム内部の状態であり、誤りがシステム外部に影響を与えたときに、それは障害となってあらわれたということになる。そして、誤りの原因はフォールト (*Fault*) と呼ばれる。

以上の定義を図にしたものを図 2.1 に示す。

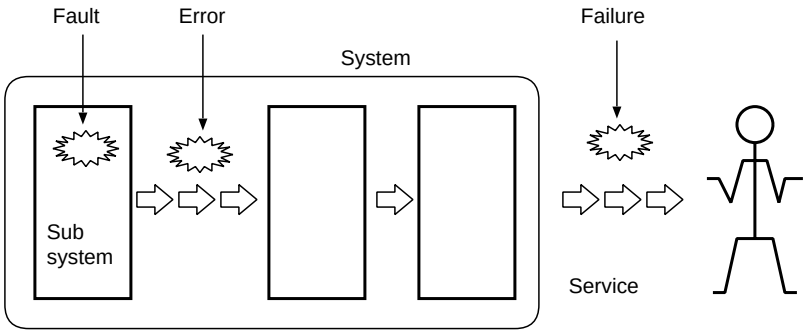


図 2.1: フォールト、誤り、及び障害の関係

また、障害は以下のように分類される。

Crash 完全に停止する／内部状態を失うような障害

Omission 外部からの入力に対して応答しなくなるような障害

Performance 出力値は正しいが時間的に遅れる障害

Timing 出力が、仕様より早い／遅いという障害

Value 正しくない出力の値を返してしまうような障害

Byzantine 任意に正しくない出力を返してしまうような障害

耐故障性というのはシステムの内部で何らかのフォールトが発生してもそれをシステムの外部に障害として影響がないようにするものである。即ち、システムの利用者から何らかの要求があったとき、システムの内部に不具合があったとしても利用者には正しい結果を返すことである。

耐故障性を持つ計算機システムを構成するための技法については、すでにハードウェア、ソフトウェア双方の領域において多くの手法が提案され、一部は実用されている。ここではこれら既存の手法や概念について、また、関連する研究について簡単に述べる。

ソフトウェア高信頼性化技術には、ハードウェアあるいはシステムを対象とする場合と同じように次の二つのアプローチがある。

- フォールトアボイダンス技術 (Fault-Avoidance)
- フォールトトレランス技術 (Fault-Tolerance)

まず、フォールトアボイダンス技術は欠陥や誤りがソフトウェアの開発過程で潜入しないようにして最初からフォールトを作り込まないようにする技術である。そして、フォールトトレランス技術はある程度の欠陥や誤りは避けられないとして、フォールトが発生しても外部に、つまり、提供するサービスにその影響が現れないようにする技術である。

従来のソフトウェア工学ではフォールトアボイダンス技術が追求されていたが、最近は特に完全なフォールトアボイダンスは一般的には不可能であるとして、安全性および信頼性のためにフォールトトレランス技術の必要性が高まっている。

ソフトウェアのフォールトはハードウェアのそれと相違点があり、ソフトウェアフォールトトレランスについて検討するときはこのようなソフトウェアの特性を十分考慮して、その対策を考えなければならない。以下でその相違点についていくつか述べる。

- ソフトウェアの場合、データが与えられた時点以降で初めてフォールトが顕在化する場合がある
- ソフトウェアの動作はユーザの意図が表現された記述内容だけに依存するのに対し、ハードウェアではそれ以外に物理的部品の故障がある
- ソフトウェアには経年変化がない

2.1 基本技術

ソフトウェアは日々複雑化しており、ソフトウェアの耐故障性を向上させることが困難な場合が多くなってきている。

耐故障性ソフトウェア (FTS : Fault Tolerant Software) を実現するための研究の多くは、1970 年頃から行われており、FTS を実現するために考えられた幾つかの基本的な手法がすでに存在するのでここに紹介する。

2.1.1 チェックポインティング

計算途中において、ある時点での計算の状態や値を記憶装置に保存することを、チェックポインティング (*checkpointing*) を行うという。また、チェックポインティングを行う時期をチェックポイントと呼ぶ。(図 2.2 で M_1 と M_2、M_3 と M_4 の間) チェックポインティングは、計算が長時間に及ぶようなアプリケーションを実行する際に利用される一般的な手段である。

図 2.2 に示しているようにチェックポインティングを行う計算の計算中に誤りが発生し、直前のチェックポイント (*M_3* の後とのチェックポイント) の計算状態を取り出す動作を、ロールバック (*rollback*) という。また、ロールバックを行い、さらに障害発生直前のチェックポイントから実行を行って、以前障害が発生した計算状態まで戻ることを、リカバリ (*recovery*) と呼ぶ。

チェックポイントを計算途中に行った後に障害が発生した場合、ロールバックを行ってリカバリを行うことによって、計算を最初からではなく、チェックポイントから再起動することができる。

チェックポインティングを行うことにはチェックポインティングを行うというプロセスと保存装置の追加によるコストが必要であることは言うまでもないが、既に述べた通り、長時間の計算途中での障害による再起動は、チェックポインティングを行わない場合は完

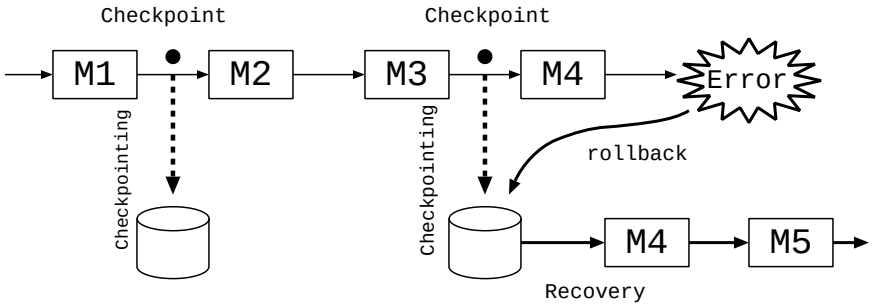


図 2.2: チェックポインティング

全に最初から計算を行う必要があり、この計算のやり直しに伴う時間的なコストは大きい。これに対してチェックポインティングを行う計算では、障害発生時にはその時点での直前のチェックポイントの状態から計算を再開することができる。

2.1.2 リカバリブロック

チェックポインティングを用いて実現される耐故障性を高める手法の一つとして、リカバリブロックがある。リカバリブロックの例を、図 2.3 に示す。

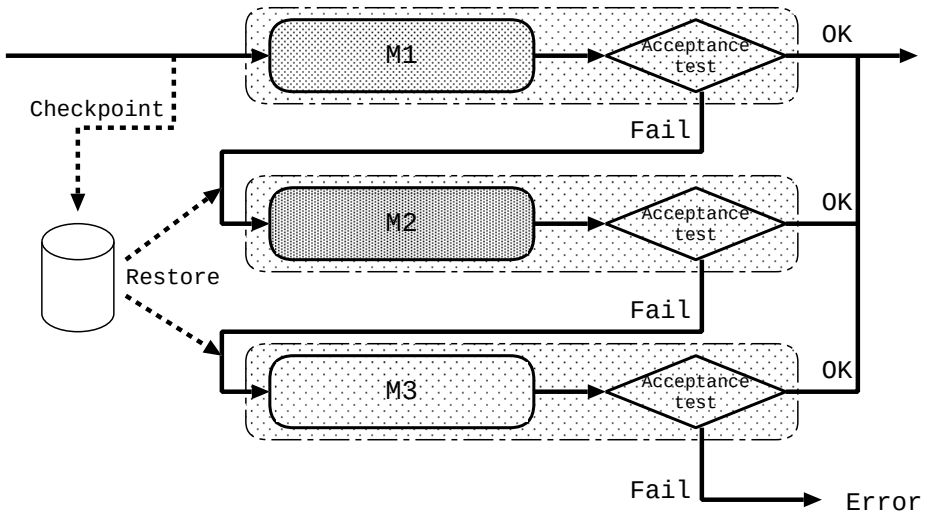


図 2.3: リカバリブロック

図 2.3 は 1 つのリカバリブロックを示している。処理の流れがリカバリブロックの中に

入るときにはまず、チェックポインティングが行われる。リカバリブロックには、同じ意味の計算を行う実行単位が複数用意されている (図中では 3 つ、M1 と M2、M3)。

チェックポインティングが終わると、まず最初に 1 つめの選択肢である M1 が実行される。M1 の実行が完了した時点で M1 の状態や出力を用いてアクセプタンステスト (検定:*Acceptance test*) が行われる。

ここで M1 からの出力がアクセプタンステストによって妥当ではないと判断されると、このリカバリブロックに入ったときに保存したチェックポイントの値を取りだし (Restore)、別の選択肢の計算である M2 を実行する。そして、M2 の実行が完了すると M1 のときと同じように M2 の最終状態がアクセプタンステストにかけられる。M2 の状態がアクセプタンステストによって妥当であると判断されると、このリカバリブロック全体の計算が終了する。もし M2 も出力がアクセプタンステストによって妥当ではないと判断されると別の選択肢の計算である M3 を実行する。

リカバリブロックはネストさせることができる。これらを図 2.4 に示す。つまり、M1 や M2、M3 をそれ以上は分割できない小さなモジュールと仮定すると、上記の例でのリカバリ-ブロック (A とする) 全体は、さらに大きなリカバリブロックである T の一部になっている。リカバリブロック T のうちの計算の選択肢の 1 つである A がアクセプタンステストで妥当でないとされた場合は、A 内の M1 や M2 の計算結果は全て破棄され、T の他の選択肢である計算が実行される。

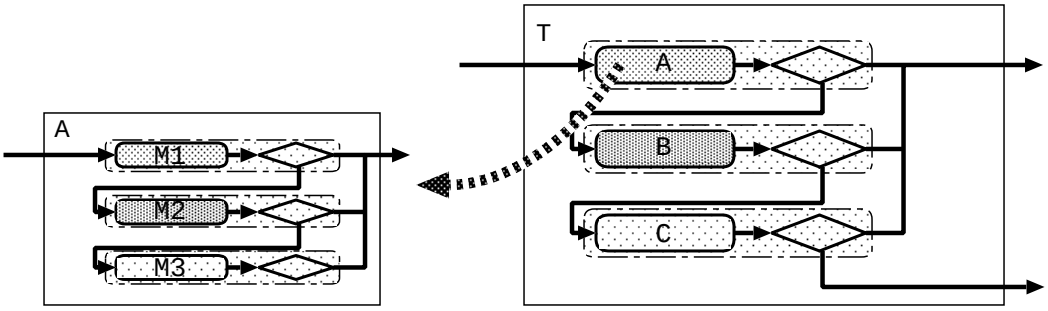


図 2.4: ネストされたリカバリブロック

2.1.3 複製

最も古くから行われている耐故障性を高めるための手法の一つとして、複製 (replication) がある (図 2.5)。

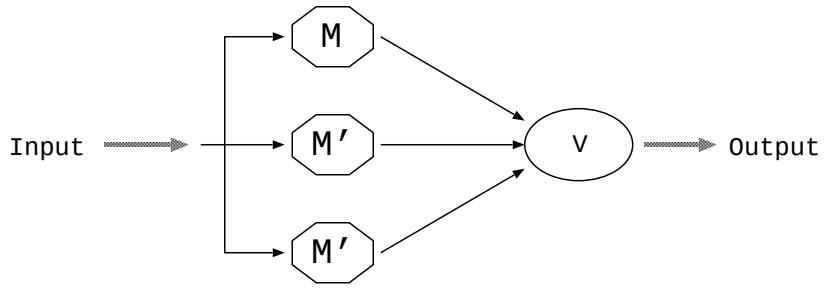


図 2.5: 複製

複製は文字通り、ある機能を実行する 1 つのモジュール M の複製を複数個作成し、これを同時に実行することにより複数の出力を得る[1] 。そして、選択関数 V に得られた複数の出力を入力することにより正しい出力を得る。特に、選択関数 V が多数決によって結果を決めている場合は、多数決関数と呼ばれることがある。

複製の方法では、複数の複製が同じフォールトを内部に持っていた場合、複数の M からは同じ出力が得られるために、ボーティング関数でそのフォールトを検出して対処するのは不可能である。つまり、複製では一過性の障害には対処できるものの、長時間内在する障害 (設計時のミスによるソフトウェアのバグや、ハードウェアの設計ミスによる決まった状況での故障) には対処することができない。

2.2 FTAG モデル

ここでは APR 技術実装の基盤となる計算モデル FTAG について説明する。計算モデル FTAG は属性文法に基づいた階層的関数型モデル HFP(Hierarchical and Functional Process)[3] を基にして、再実行や複製などの耐故障性を高めるために必要な機能を拡張したものである。ここではまず計算モデル FTAG の基本的なモデルについて述べ、次に再実行 [4], 複製について説明する。

2.2.1 モジュールとモジュール分解

計算モデル FTAG では、全ての計算を**モジュール**と呼ばれる純粋に数学的な関数の集まりとして記述する。各モジュールは複数の入力と出力を持つことができる。入力 $x_1, \ldots, x_n$ および出力 $y_1, \ldots, y_m$ を持つモジュール M を以下のように記述する。

[1]図 2.5 で M' はモジュール M の複製モジュールである

$$M(x_1, \ldots, x_n \mid y_1, \ldots, y_m)$$

$x_1, \ldots, x_n$,$y_1, \ldots, y_m$ を M の**属性**と呼ぶ。この場合 $x_1, \ldots, x_n$ を**入力 (または相続) 属性**、$y_1, \ldots, y_m$ を**出力 (または合成) 属性**と呼ぶ。

M が十分に単純な場合は、出力は直接計算される。この様なモジュールを基本モジュールと呼び、以下のように記述する。

$$M(x_1, \ldots, x_n \mid y_1, \ldots, y_m) \Rightarrow return \ \ \textbf{where} \ \ E$$

ここで E は $y_1, \ldots, y_m$ が $x_1, \ldots, x_n$ からどの様に計算されるかを表す式の並びで、**属性関係式**と呼ばれる。属性関係式 E は M_i の入力属性および M の出力属性を計算するための式の並びである。M_j のある入力属性 y が M_i のある出力属性 x を単にコピーされるだけの場合、y の代わりに x を M_j の入力として記述することで $x = y$ という関係式を省略することができる。

M が複雑な場合はより単純な複数のサブモジュールに分解される。M が $M_1, .., M_k$ に分解されることを以下のように記述する。

$$M(x_1, \ldots, x_n \mid y_1, \ldots, y_m) \Rightarrow return \ \ M_1 \ldots M_k \ \ \textbf{where} \ \ E$$

$M_1 \ldots M_k$ と E の組を M の**分解**と呼び、以後 D で表現する。

モジュールの分解には条件によって複数の方法がある場合が考えられる。条件付きの分解は以下のような一般形で記述される。

$$\begin{array}{l} M(x_1, \ldots, x_n | y_1, \ldots, y_m) \Rightarrow \\ \quad [\quad C_1 \quad \rightarrow D_1 \\ \quad | \quad \vdots \\ \quad | \quad C_n \quad \rightarrow D_n \\ \quad | \quad \textbf{otherwise} \ \rightarrow D_{def}] \end{array}$$

条件 $C_1, \ldots, C_n$ はこの順でテストされ、C_i が真になった時点で対応する分解 D_i が適用される。どの条件も充たされない場合はデフォルト分解 D_{def} が適用される。

FTAG におけるプログラムは、全てが基本モジュールで記述できるようになるまでモジュール分解を繰返し適用することによって実行される。計算の過程や結果は**計算木**と呼ばれる木構造によって表現される。入力属性は計算木を上から下へ、出力属性は下から上へ流れる (図 2.6)。

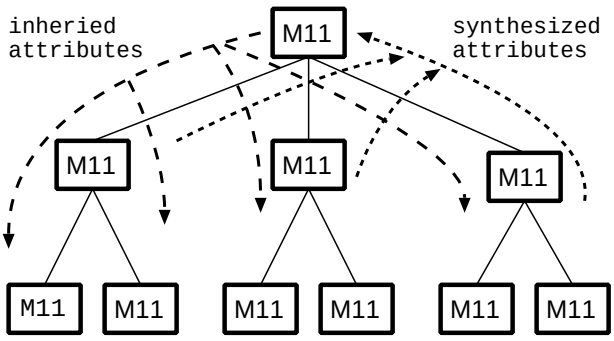


図 2.6: FTAG の計算木

サブモジュールの実行順序はサブモジュール間の属性の依存関係によってのみ決定される。依存関係が存在しない部分は並列に実行することが可能である。

FTAG では依存関係によって決定する潜在的な実行順序に加えて、実行順序を強制するための記法も用意されている。例えば、M_1 の実行を M_2 の前に行なうためには演算子 ';' とグループ構造 {} を使用して $\{M_1; M_2\}$ の様に記述する。

他に実行順序を明示的に指定するものとして異なる入力に対して同一のモジュールを複数個適用する場合の記法が用意されている。これらについては説明を省略するが、詳しいことは [7] を参考にされたい。

2.2.2 再実行

FTAG は計算木の一部を別の計算木で置換する**再実行**という機構を持っている。これは障害が発生した (正しくない結果が得られた、または結果が得られなかった) 部分を再計算するために利用される。

ここで障害は全て値の誤りに反映されることで検出可能であるという仮定を設ける。この仮定はソフトウェア障害を取扱うときに一般的なものである [5]。一方プロセッサの故障などの障害は適当な属性の値が $\perp$ となることで検出できるものとする。

図 2.7 は再実行のもっとも単純な場合を表している。モジュール M は M_1, M_2, M_3 に分解され実行するが、M_3 の実行中に入力属性のフォールトのテストが行われ、その入力属性がフォールトであることが判明したとする。このときモジュール M_3 にはフォールトのとき再計算を開始するモジュールが書かれているのでそのモジュールから再計算が行われる。この単純な例では障害は一過性、すなわち再計算の時には発生しないことを仮定している。。再計算が正しく実行された後では、再計算された部分が現在の実行履歴として保存される。

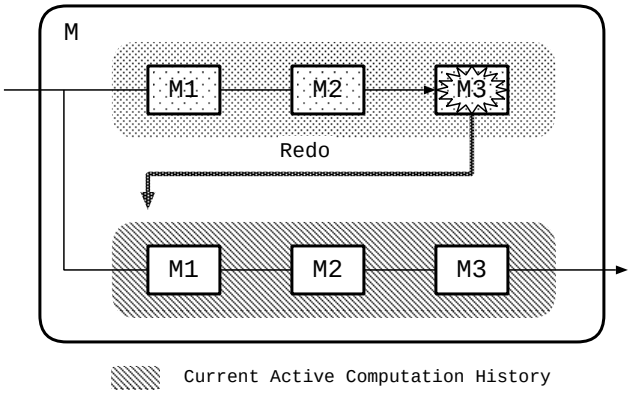


図 2.7: 再実行の概念

図 2.7 は FTAG プログラムでは以下のように記述される。

$$
\begin{array}{l}
M(x \mid y) \Rightarrow \\
\quad M_1(x_1 \mid y_1) M_2(x_2 \mid y_2) M_3(x_3 \mid y_3) \\
\quad \textbf{where} \\
\qquad \vdots \\
M_3(x \mid y) \Rightarrow \\
[\quad valid(x) \qquad \rightarrow M_3body(x \mid y) \\
\mid \quad \textbf{otherwise} \quad \rightarrow \textbf{redo M} \quad]
\end{array}
$$

M_3 において $valid(x)$ は入力 x が正しいかによって障害の発生の有無をテストする。正しければ M_3 は M_3body が実行され、実際の計算が行なわれる。さもなければ、M_3 は $redoM$ によってモジュール M が再実行され、以前の M の実行結果と置換される。

2.2.3 複製

障害時にもサービスを継続して行なうための標準的な手法として**複製** (*replication*) がある。複製はプログラムの一部分を複数個同時に実行させ、どれかに誤りが発生しても全体として正しい計算を行うための機構である。

FTAG では次のように記述することにより実現される。

$$M(x \mid y) \Rightarrow M_1(x \mid y)\ M_1(x \mid y)\ M_1(x \mid y)$$

ここで 3 つの M_1 はそれぞれが同一の入力属性 x および出力属性 y を持つ。このような形態の分解を**複製分解** (*replicated decomposition*) と呼び、分解された M_1 の集合を**複製対象**

(*replica*) と呼ぶ。複製分解の解釈は以下のようになる。各モジュールは通常の分解と同様同時に実行されるが、どれかひとつだけが M の正しい結果となる。当然ながら、耐ハードウェア故障のためには各 M_1 はすべて別のハードウェア上で実行されなければならない。

複製分解においてモジュールの名前が同一でない場合は N バージョンプログラミングを実現しているものと考える。以下の記述を考える。

$$M(x \mid y) \Rightarrow M_1(x \mid y)\ M_2(x \mid y)\ M_3(x \mid y)$$

ここで $M_1,\ M_2,\ M_3$ は M の異なった実装を表す。$M_1,\ M_2,\ M_3$ にはそれぞれ同じ入力属性 x が入る。そして同じ出力属性が返されることが期待される。

複製分解において、複製対象であるサブモジュールの合成属性の中からもとのモジュールの合成属性を選択する方法にはいくつか考えられる。ひとつはいわゆる**選択関数**、すなわちサブモジュールの出力すべてを入力とし、複数の値が一致していたらそれを正しい結果として出力する関数を使用する方法である。

他にも $\perp$ 以外の値が得られたらそれらは全て正しいとする方法もある。この仮定はサブモジュールの中に明らかにおかしいと思われる結果を検出して取り除くためのアクセプタンステストが追加されているならば、より安全に受け入れられるものとなる。当然ながら、最良の方法はアプリケーションおよび対処すべき障害の種類に依存する。

2.3 APR 複製技術

ここでは並列度の高いモジュール複製技術である APR 複製技術について説明する。APR 技術は複製によってモジュールの冗長性を高めると同時に、システムに存在する並列性を利用して高速な計算を可能にするという特徴を持っている。

APR 複製技術は FTAG 計算モデルを基盤としている。FTAG 計算モデルは全ての計算をモジュールと呼ばれる単純に数学的な関数の集まりとして記述している。そしてそのモジュール計算の過程は計算木と呼ばれる木構造によって表現され、属性は木のノード間の幹を流れるデータとして表現される。

このような FTAG 計算モデルは APR を実現するにわたって重要な三つのポイントを提供してくれる。最初のポイントは関数型であることである。関数型言語は参照透過性という特性を持っている。すなわち、モジュールの実行は属性の入力に対してモジュール実行時刻と順序に関係なく合成属性を出力する。二番目のポイントはモジュール実行順序は FTAG プログラムによって明確に定義された属性の依存関係のみで決定される。三番目のポイントは FTAG では、システム状態はそれぞれのノードについている属性の値と計算木によって示される。

ここではまず APR 複製技術の基本的な振る舞いについて述べ、次に誤り処理と APR 複製技術の特徴などについて説明する。

2.3.1 Active Parallel Computation

APR 複製技術は計算木を幾つかに複製し、それぞれの計算木で異なる計算順序で計算を行う。計算木の実行が始まると最初に計算木は幾つかに複製される。複製されたそれぞれの計算木をレプリカ (replica) と呼ぶ[2] 。APR 複製技術が基盤としている FTAG 計算モデルは関数型モデルを基にしているので**参照透過性** (*referential transparency*) という特性を持っている。即ち、計算木で分解されたモジュールの実行は属性の依存関係がなければその実行順序には関係なく結果は同じである。APR 複製技術でそれぞれのレプリカが異なる計算順序で計算を行っても同一の結果を返すことは参照透過性によって保証される。

それぞれのレプリカは違う計算順序で計算木の枝にあるモジュールを計算する。即ち、ある時刻においてはそれぞれのレプリカで違うモジュールの計算を行う。例えば以下のような FTAG プログラムコードがあるとする。

[2]FTAG のレプリカと APR のレプリカは異なることに注意されたい。FTAG では計算木の中で一部分だけが複製され、それをレプリカと呼んでいる。しかし APR では複製された計算木のことをレプリカと呼んでいる。

$$
\begin{aligned}
M(x \mid y) \Rightarrow\ & M_1(x_1 \mid y_1) \\
& M_2(x_2 \mid y_2) \\
& M_3(x_3 \mid y_3) \\
& \textbf{where} \\
& \vdots
\end{aligned}
\qquad
\begin{aligned}
M_2(x_2 \mid y_2) \Rightarrow\ & M_{21}(x_{21} \mid y_{21}) \\
& M_{22}(x_{22} \mid y_{22}) \\
& M_{23}(x_{23} \mid y_{23}) \\
& \textbf{where} \\
& \vdots
\end{aligned}
$$

$$
\begin{aligned}
M_1(x \mid y) \Rightarrow\ & M_{11}(x_{11} \mid y_{11}) \\
& M_{12}(x_{12} \mid y_{12}) \\
& M_{13}(x_{13} \mid y_{13}) \\
& \textbf{where} \\
& \vdots
\end{aligned}
\qquad
\begin{aligned}
M_3(x_3 \mid y_3) \Rightarrow\ & M_{31}(x_{31} \mid y_{31}) \\
& M_{32}(x_{32} \mid y_{32}) \\
& M_{33}(x_{33} \mid y_{33}) \\
& \textbf{where} \\
& \vdots
\end{aligned}
$$

このFTAGプログラムコードは図2.8の示しているようにレプリカ１は計算木の一番左にあるモジュール M_1 から分解され、レプリカ２は中央にあるモジュール M_2 から分解され、レプリカ３は右にあるモジュール M_3 から分解していく。このようにそれぞれのレプリカで違う計算順序に基ついて計算を行うことを我々は *Active Parallel Computation*（以下APCと呼ぶ）と言う。

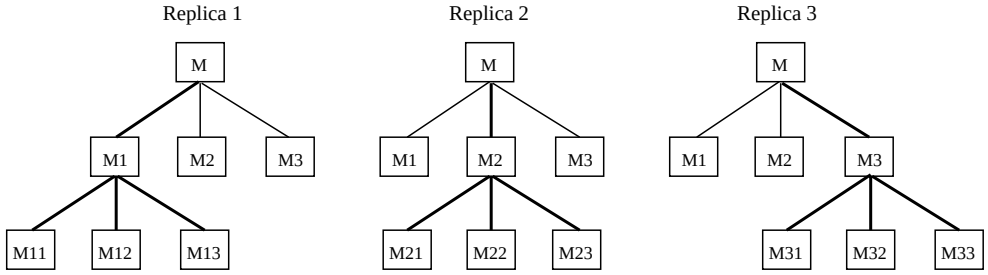


図 2.8: Active Parallel Computation

APR複製技術がそれぞれのレプリカで違う実行順序でモジュールの計算を行っていることと全体の実行時間に関する特徴を示すためにGantt Chartを利用する。ここで用いるGantt Chartはプロセッサを縦に、そしてモジュールの計算順序を時間の流れにあわせて左から右へ表現する。そのChartの中で一つの箱は一つのモジュールの計算を表している。箱の左辺はモジュールの実行開始時間を、右辺は一時停止あるいは終了時間を表している。

図 2.8 のように 3 つのレプリカがあると仮定する。そして、それぞれのレプリカは二つのプロセッサを並列に利用することができると仮定する。最初のレプリカ 1 はルートモジュール M を分解して、その分解されたモジュールの一番目のモジュール M_1 から計算を行う計算順序を用いる。その一番目のモジュール (M_1) はまたいくつかのモジュールに分解され、その分解されたモジュールの一番目のモジュール (M_{11}) から計算し始める。レプリカ 2 はルートモジュール M を分解して、その分解されたモジュールの二番目のモジュール (M_2) から計算する計算順序を用いる。その二番目のモジュール (M_2) はまたいくつかのモジュールに分解され、その分解されたモジュールの二番目のモジュール (M_{22}) から計算し始める。そして、同じようなことがレプリカ 3 でも違う計算順序でモジュールの計算が行われる。

分解されたモジュールの間でデータの依存性がない以下の FTAG プログラムコードを APC でどのように実行を行うかを Gantt Chart で表したのを図 2.9 で示す。

$$
\begin{aligned}
M(x \mid y) &\Rightarrow M_1\ M_2\ M_3 \quad \textbf{where} \quad E \\
M_1(x \mid y_1) &\Rightarrow M_{11}\ M_{12}\ M_{13} \quad \textbf{where} \quad E_1 \\
M_2(x \mid y_2) &\Rightarrow M_{21}\ M_{22}\ M_{23} \quad \textbf{where} \quad E_2 \\
M_3(x \mid y_3) &\Rightarrow M_{31}\ M_{32}\ M_{33} \quad \textbf{where} \quad E_3
\end{aligned}
$$

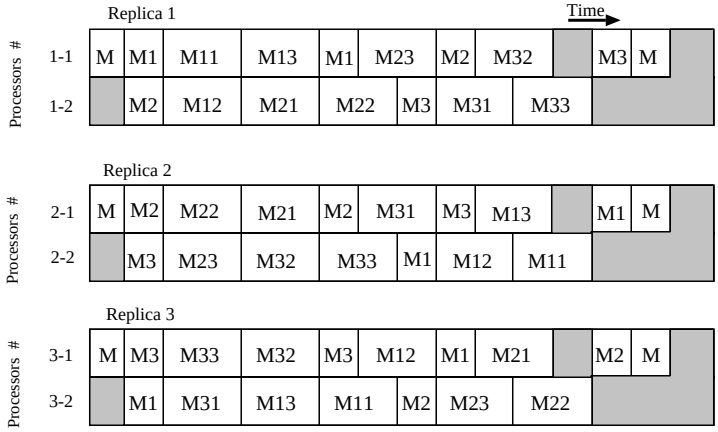


図 2.9: The Gantt Chart:Active Parallel Computation

この FTAG プログラムコードはレプリカ 1 で最初にモジュール M が実行をはじめて分解が行われる。その結果、3 つのモジュール $(M_1,\ M_2,\ M_3)$ が生成され、3 つのモジュールが計算を終了しその属性を返すまで M は一時停止 (suspend) する。M_1 はプロセッサ 1-1 で実行を始めて分解が実行され、M_1 も分解したモジュールが属性を返すまで一時停止

する。M_1 の実行と並列に M_2 はプロセッサ 1-2 で実行され、分解が行われた後 M_1 と同じように一時停止する。レプリカ 1 での計算順序はルートモジュール M の計算が最初に分解されて、その次に M_{11} と M_{12} がプロセッサ 1-1 と 1-2 でそれぞれ実行する。モジュール M_{11} と M_{12} の実行が完了するとモジュール M_{13} がプロセッサ 1-1 で実行を行う。モジュール M_{21}、M_{22}、M_{23} は以前実行された M_2 の分解より作られていて使えるプロセッサが現れるまで待機 (await) する。モジュール M_1 は M_{11}, M_{12}, M_{13} の属性が返されるまで実行することは出来ない。ない。モジュール M_{21} は M_{13} と依存関係がないので、実行のためのスケジューリングによって M_{13} と並列に実行される。

それぞれのレプリカで以上のようにモジュールの計算を行う。しかし、それぞれのレプリカは違う実行順序でモジュールの計算を行う。

2.3.2 Active Parallel Replication

APC では、それぞれのレプリカで異なる計算順序を持ってモジュールの計算を行っているので、ある時刻で計算しているモジュールはそれぞれのレプリカで異なる。即ち、一つのモジュールに対してすでに計算を終えたレプリカとまだ計算を始めていないレプリカが存在することになる。

以上のような APC を基にそれぞれのレプリカで計算を終えたモジュールの属性を他のレプリカに送るようにする。他のレプリカからあるモジュールの属性を受け取ったレプリカはそのモジュールの属性を保存する。そして、保存された同じモジュールの属性の中で同じ属性が多数存在するかを調べる。同じ属性が多数存在すると確認されたとき、そのモジュールをまだ計算していないレプリカは確認されたモジュールの属性を受け入る。そして、保存されたモジュールの属性を比べることでフォールトの検出を行う。以上のようなことを我々は *Active Parallel Replication*（APR）と呼ぶ。

複製技術の主な目的はシステム内部にフォールトが含まれていても正しい結果を出力することである。以下ではまずはじめに、APR 複製技術がどの様にモジュールの計算を行うかを説明し、フォールトを含んでいる場合に APR 複製技術がどのようにフォールトの検出と回復を行うのかについて説明する。

レプリカがあるモジュールの分解を行うとき、その分解した結果（サブモジュールの名前と入力属性）を全てのレプリカに送る。そして同じモジュールの分解結果が他のレプリカから送られてきているのかを調べる。もし分解した結果が他のレプリカから送られているのであればその結果と比べる。他のレプリカで分解された同じモジュールの結果が送られてきたときも、同じモジュールの分解結果が存在するかを調べる。もし分解の結果が異

なるときはレプリカは同一モジュールの分解が他のレプリカで行われることを待ちながら計算を続ける。全てのレプリカでそのモジュールの分解が行われて同一の結果が確認されないレプリカでは、その分解をフォールトとして扱う。

レプリカがモジュール M_i の出力属性を計算するとき、M_i は計算した出力属性を他のレプリカに送る。出力属性を受けたレプリカは属性を保存し、もし同じモジュールの出力属性がすでに存在すると同じモジュールに対して他のレプリカが送った出力属性をその出力属性と比べる。多数の同じ出力属性が確認されるとその出力属性は保存される。まだこのモジュールの計算が終わっていないレプリカは確認された出力属性を保存し、それ以上そのモジュールの計算を行わない。違う出力属性を計算したレプリカはそのフォールトを回復しなければならない。多くの同じ出力属性が得られなかった場合 M_i を実行したレプリカは他のレプリカがその出力属性を送ってくることを待ちながらモジュールの計算を続ける。もし全てのレプリカが M_i の実行を完了した後も多数の同一出力属性が得られないときはフォールト回復が行われる。

フォールトを含んでいない場合

以上の方法に基づいて以前の FTAG プログラムコードの実行状況を Gantt Chart を用いて表したものを図 2.10(b) に示す。この Gantt Chart では APR のアプリケーションの実行時間がレプリカ間の協力がない APC と比べてどれだけ短くなっているかを示している。

図 2.10(b) の Gantt Chart では、モジュール M_{32} と M_{33} はレプリカ 2 と 3 で計算されてその出力属性がレプリカ 1 に送られて比べられその出力属性が妥当であると確認されるのでレプリカ 1 では M_{32} と M_{33} は計算されないことがわかる。そして、レプリカ 2 では M_{11} と M_{13} が、レプリカ 3 では M_{21} と M_{22} が計算されないことを示している。その結果、フォールトが発生していないときの計算時間は APR 複製技術を適用した時に短くなることを示している。

フォールトを含んでいる場合

計算木の中でフォールトが発生した場合、とき APR ではどのように計算をおこなうのかを図 2.11 に示す。

図 2.11 の Gantt Chart は、レプリカ 1 と 3 でモジュール計算を終えたモジュール M_{12} で出力属性が異なる時を示している。モジュール M_{12} の出力属性が違うことが確認されて（フォールトの検出）どのレプリカでフォールトが発生したが判明できないので、レプリカ 2 でモジュール M_{12} の出力属性が送られてくるのを待ちながらモジュールの実行を続ける。レプリカ 2 からモジュール M_{12} の出力属性が送られてきてレプリカ 1 でフォー

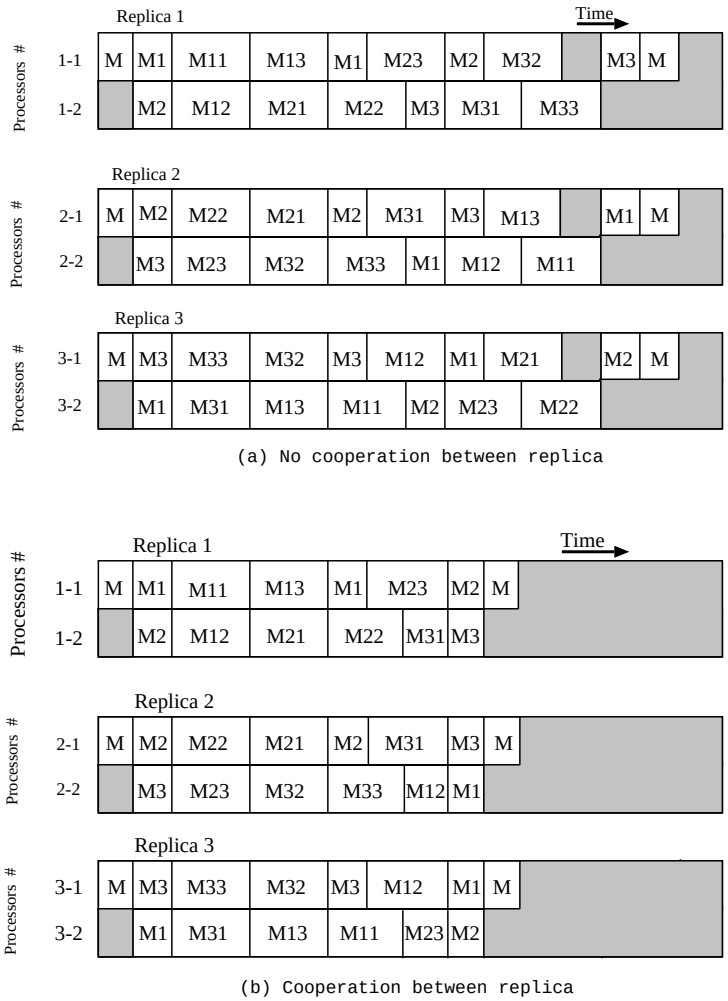


図 2.10: The Gantt Chart: APR

ルトが発生したことが確認され、レプリカ 1 はモジュール M_{12} の再計算を行う（フォールトの回復）。このときモジュール M_1 もモジュール M_{12} と依存関係を持っているので再計算される。

このようにフォールトが発生した場合の計算時間は再実行に費やした時間が増加する。

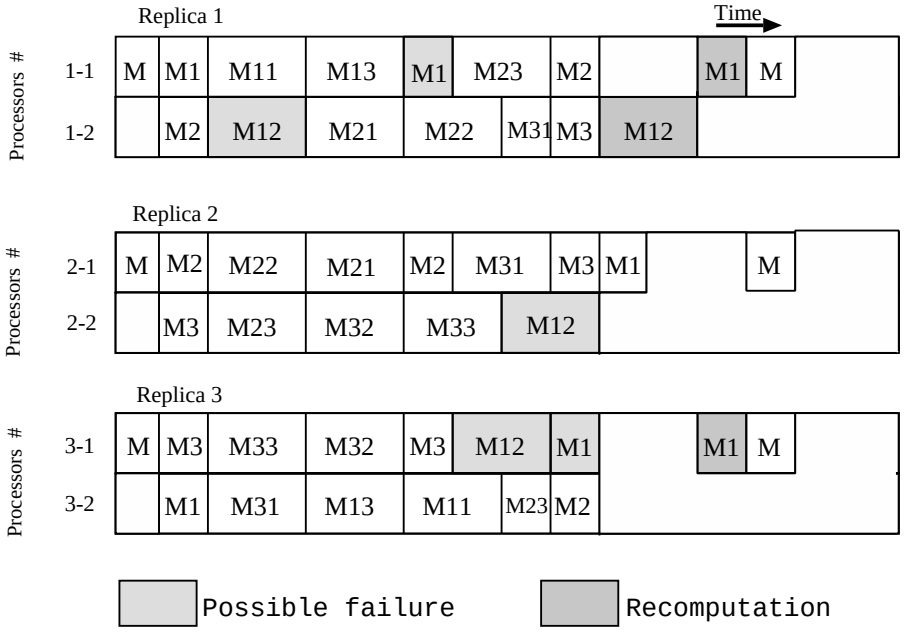


図 2.11: The Gantt Chart:Forward Recovery

第 3章

APRでの問題点と解決方法

APR 複製技術はシステムに存在する並列性を利用して高速な計算を可能にするという特徴を持っている複製技術である。しかし計算木の深さが深いときに、深さが浅いところにあるモジュールでフォールトが発生すると、そのフォールトの検出が遅れ、フォールトの伝搬が大きくなるという問題をかかえている。このフォールトの伝搬は無駄な計算を行ってしまうことで計算木全体の計算時間を長くしてしまう。この章では APR 複製技術が持っているフォールトの伝搬に関する問題について説明する。そして、APR 複製技術の拡張として、フォールトの伝搬を防ぐためにフォールトを早期に検出する新しい技術を提案する。

3.1　フォールトの伝搬

実際に実行される計算木にはいろいろな形が存在する。その中で縦に計算木の枝が深く分解されるような計算木では浅いところで発生したフォールトの伝搬は計算木全体の実行時間に大きな影響を与える。

APR では Value Failure Model の場合、n 個のフォールトに対応するためには $2n+1$ 個のレプリカが必要である。そしてフォールトを検出するためには $n+1$ 個以上の属性値が必要である。そこで計算木の枝が深くなっているとそれぞれのレプリカで計算された同じモジュールの出力属性がフォールトの検出に必要な $n+1$ 個が集まるまでにはかなりの時間がかかってしまう。もし深さのある枝での計算は枝の浅いどころでフォールトが発生してしまうとそのフォールトの検出が遅れ、その後のモジュール計算は全てフォールトを含んだことになり無駄な計算になってしまう。

例えば図 3.1 のような計算木を考えてみる。１つの Value Failure に対処出来るように

3 つのレプリカが存在すると仮定する。レプリカ１、レプリカ２、レプリカ３はそれぞれ左、中央、右の順序で計算を始めるとする。この例ではモジュールのフォールトを検出するためには出力属性が 2 つ必要である。レプリカ１のモジュール M_{11} の出力属性がフォールトを含んでいるかどうかはレプリカ３がモジュール M_{11} を計算した後に明らかになる。これはレプリカ３の右の枝にある M_3 を根とする部分木全てのモジュールが計算を終えたあとになる。もし右の枝がかなり深い場合はモジュール M_{11} の計算が遅れ、レプリカ１のモジュールの M_{11} で発生したフォールトの検出が遅れることは言うまでもない。

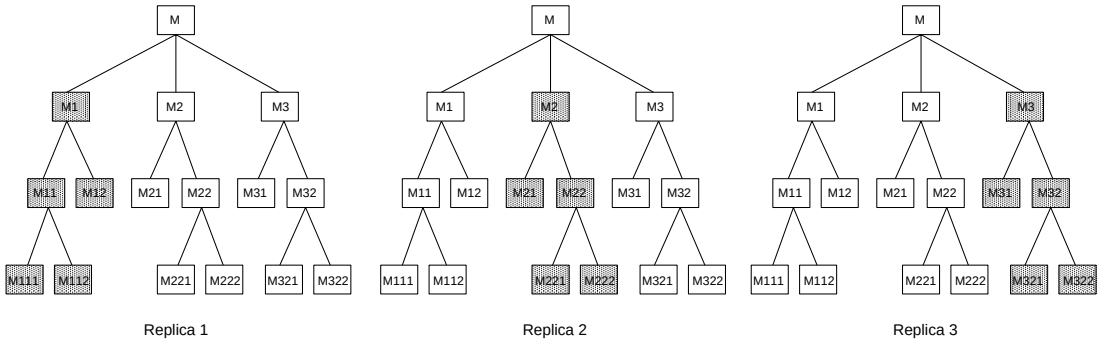


図 3.1: 深さのある計算木：その１

図 3.2 に図 3.1 の計算木でフォールトが発生し、どの様に伝搬するのかを示す。図の中にあるモジュール M_{11} でフォールトが発生したと仮定するとフォールトを持つ出力属性を発生させ、その属性は下のモジュールで入力属性として利用されるのでそのフォールトは検出されるまで伝搬してしまう（図 3.2 の (b)）。

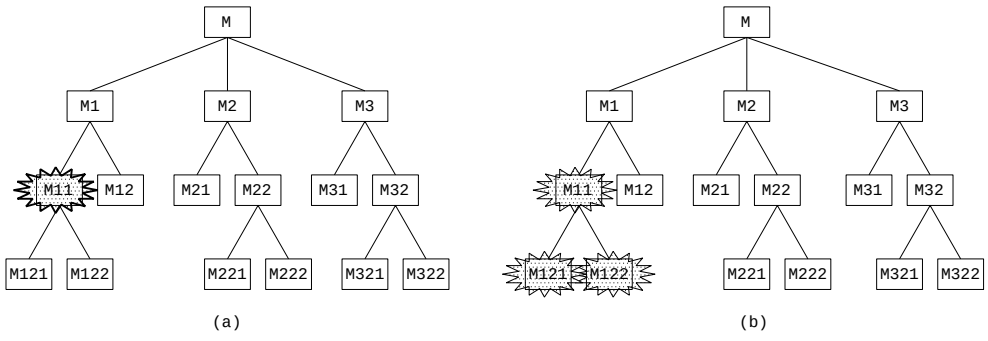


図 3.2: フォールトの伝搬

以上のようなフォールトの伝搬はそのフォールトの検出された時点で誤りのある計算と

見なされて無駄な計算に時間を費やしてしまったことになる。その結果、フォールトの回復を行ったとしても無駄な計算を行ったため計算時間全体が長くなってしまう。このようなことを防ぐため対策が必要である。

3.2 フォールトの早期検出

以上で説明したようにフォールトの伝搬は計算木全体の実行時間に大きな影響を与える。フォールトが発生したときにフォールトを早期検出することはフォールトの伝搬を防ぎ無駄な計算を減らすことで全体の計算時間を短くする。

図 3.3 のような計算木を考えてみる。この木で枝の深さが浅いところにあるモジュールでフォールトが発生した場合と、深いところにあるモジュールでフォールトが発生した場合を比べてみる。まず枝の深さが深いと見なされるモジュール M_{123} でフォールトが発生したとき、そのモジュールは依存関係を持つモジュール M_{12}, M_1 が無駄になり、無駄になるモジュール計算は少ない。

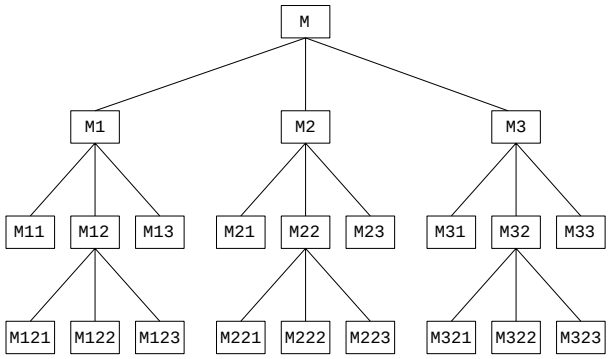


図 3.3: 深さのある計算木：その 2

しかし枝の深さが浅いところにあるモジュール M_{12} でフォールトが発生したときはそのモジュールに依存関係を持つサブモジュール M_{121}, M_{122}, M_{123}, M_{12},M_1 が全て無駄な計算になり計算時間の損失が大きくなる。そこでこのような浅いところで発生したフォールトは早期検出されるべきである。

以上のようなことが APR 複製技術ではどのようになるのかを Gantt Chart を用いて図 3.4 で説明する。図 3.4(a) は、モジュール M_{123} で発生したフォールトがいつ検出されるかを示している。レプリカ 1 のモジュール M_{123} が計算され、レプリカ 3 のモジュール M_{123} が計算された後に初めてフォールトの発生が分かる。この時レプリカ 1 のモジュール M_{123}

でフォールトが発生したとしてもその影響を受けるのはレプリカ 1 のモジュール M_{12} と M_1 だけである。

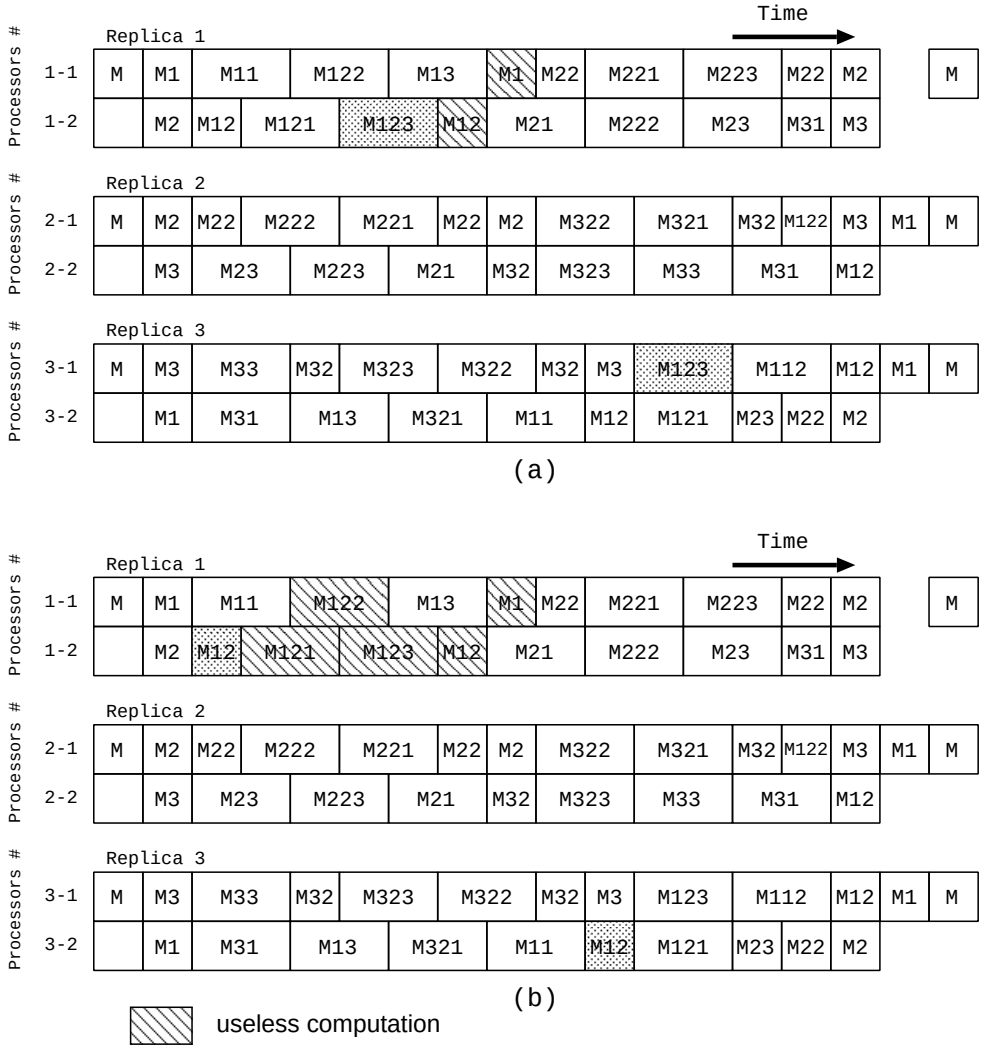


図 3.4: 無駄になる計算

図 3.4(b) の場合は枝の浅いところにあるモジュール M_{12} でフォールトが発生した場合と示している。レプリカ 1 のモジュール M_{12} が計算され、レプリカ 3 のモジュール M_{12} が計算された後にフォールトの発生が検出される。この時レプリカ 1 のモジュール M_{12} でフォールトが発生したとすると、依存関係を持っているレプリカ 1 のモジュール M_{121}, M_{122}, M_{123},M_{12}, M_1 が無駄な計算になってしまう。このようにフォールトの検出が遅れたため無駄な計算に時間を費やしてしまった結果、総計算時間が長くなってしまう。

以上で説明したようにフォールトの早期検出は重要な問題である。そこで APR 複製技術でフォールトの早期検出ができるように APR 複製技術の拡張を行う。

3.3 APR 複製技術の拡張

3.3.1 動的計算管理法

計算木でフォールトが枝の浅いところで発生したとき、そのフォールトを早期に検出してフォールトの伝搬を抑制し、計算木全体の実行時間を長くならないように APR 複製技術を拡張する。APR 複製技術はそれぞれのレプリカで違う順序によって計算を行い、モジュールの出力属性を共有することで耐故障性の実現と全体の実行時間を短くするものである。そこでそれぞれのモジュールの計算は APR 複製技術の基本的な計算順序を守りながら一定の時間で計算木の枝を変える方法を提案する。

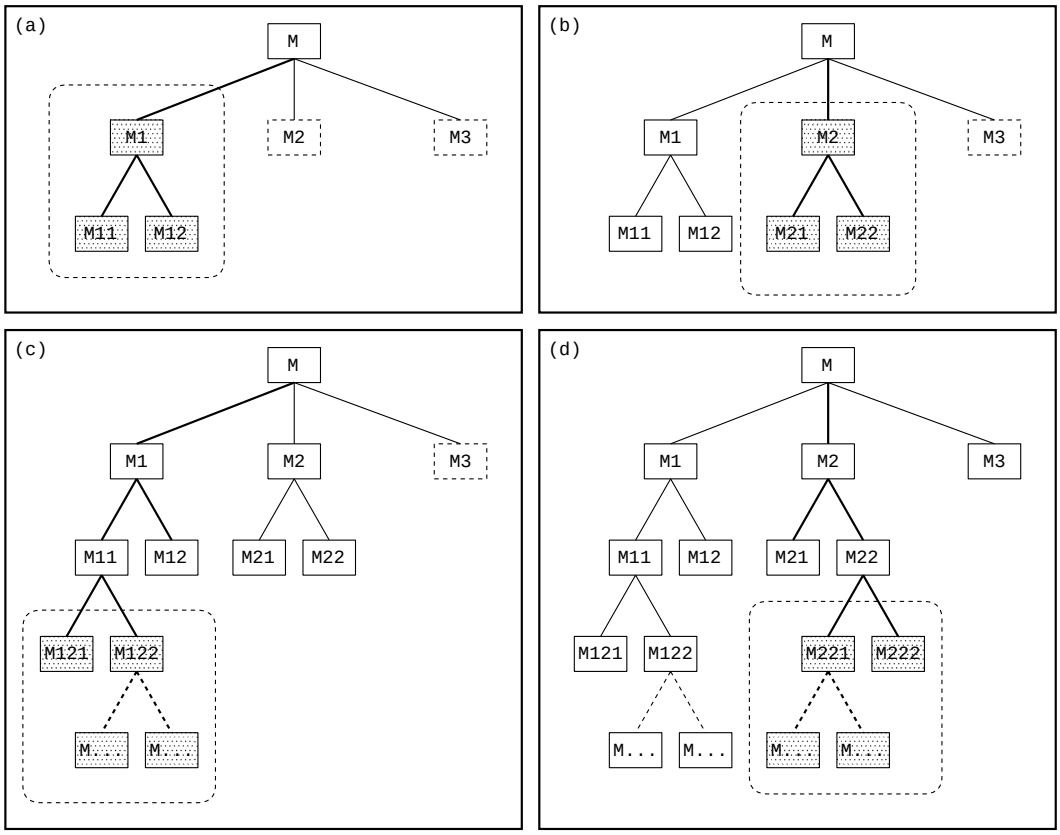


図 3.5: Adaptive Computation Managerment Scheme

図 3.5 は新しく提案する方法によって一つのレプリカの中で各モジュールの実行がどのように行われるかを示している。図の中で (a) はモジュール M が最初に分解され、APR のモジュール計算順序に基ついて最初のモジュール M_1 から実行していることを示している。そして一定の時間の間はその計算が続くのであるが、その一定の時間が過ぎるとそのときまでのモジュールの計算に関するデータを保存し、計算するべき部分木を変更する。次に計算する部分木は一時停止した部分木を除いてそのレプリカの計算順序に基ついた次の計算可能な部分木である。図 3.5 の (b) で M_2 を根とする部分木の計算を開始している。M_2 を根とする部分木の計算も一定の時間で別の部分木に変わることになる。以上のようなことを我々は**動的計算管理法** (*Adaptive Computation Management Scheme* : ACMS) と呼ぶ。

計算する部分木を変え続ける範囲はレプリカの数に依存する。APR ではモジュールの計算時間が一定であると仮定すると、フォールトを含んでいないときに計算木全体の約

$$\frac{n+1}{2n+1}$$

の（n は対応出来る Value Failure の数）モジュールがその計算を終了すると全体の計算が完了することから、$2n+1$ 個のレプリカが存在するとき $n+1$ の範囲内で部分木の実行を変える。

APR 複製技術はそれぞれのレプリカで違うモジュール計算順序の基でモジュールの実行を行う技術で、ACMS は違うモジュール計算順序を持っている一つのレプリカの中で一定の時間で実行する部分木を変えることに注意されたい。

第 4 章

解決方法の実例への適用

この章ではAPRでフォールトの早期検出する方法として提案したACMS方法の有効性について実例をあげで説明を行う。その後、もっと深刻とされる計算木での適用を行う。

4.1 ACMSの適用

モジュールの間でデータの依存性がない以下のFTAGプログラムとそれを計算木で表した図4.1を実例としてACMSの適用を行う。

$$
\begin{aligned}
&M(x \mid y) \Rightarrow M_1\ M_2\ M_3 \quad \textbf{where} \quad \mathbf{E}.\\
&\quad M_1(x \mid y_1) \Rightarrow M_{11}\ M_{12}\ M_{13} \quad \textbf{where} \quad \mathbf{E_1}.\\
&\quad M_2(x \mid y_2) \Rightarrow M_{21}\ M_{22}\ M_{23} \quad \textbf{where} \quad \mathbf{E_2}.\\
&\quad M_3(x \mid y_3) \Rightarrow M_{31}\ M_{32}\ M_{33} \quad \textbf{where} \quad \mathbf{E_3}.\\
&\qquad M_{12}(x \mid y_{12}) \Rightarrow M_{121}\ M_{122}\ M_{123} \quad \textbf{where} \quad \mathbf{E_{12}}.\\
&\qquad M_{22}(x \mid y_{22}) \Rightarrow M_{221}\ M_{222}\ M_{223} \quad \textbf{where} \quad \mathbf{E_{22}}.\\
&\qquad M_{32}(x \mid y_{32}) \Rightarrow M_{321}\ M_{322}\ M_{323} \quad \textbf{where} \quad \mathbf{E_{32}}.
\end{aligned}
$$

フォールトを含んでいないとき

図4.1の計算木でフォールトを含んでいないとき、APRとACMSを適用してモジュール計算を行ったときの様子をGantt Chartを用いて図4.2で示す。ここではそれぞれのレプリカで2つのプロセッサを利用できると仮定している。

まず図4.1の計算木でそれぞれのモジュールがAPR技複製術によってどのように計算されるかを図4.2(a)に示す。ここでもレプリカ１、２、３がそれぞれモジュールの出力属性を送りあうことで実行時間が短くなっている事が示されている。

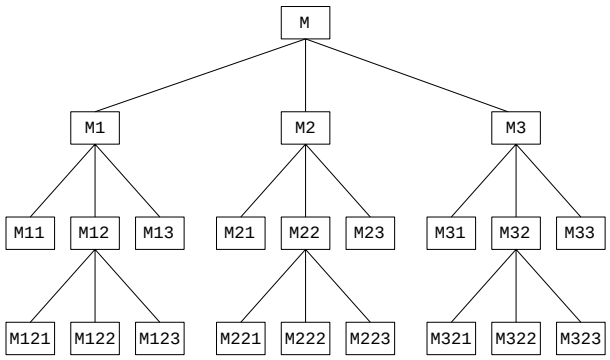


図 4.1: 深さがある計算木:実例

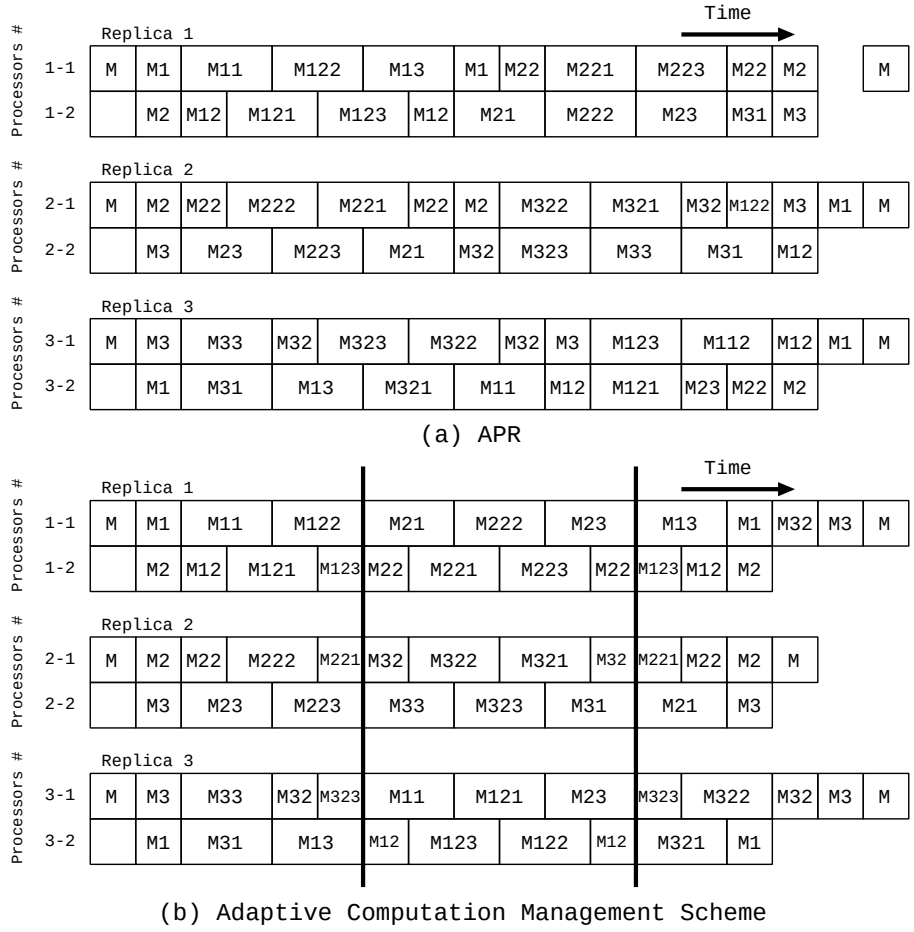


図 4.2: The Gantt Chart: APR と ACMS の比べ

同じ計算木に ACMS を適用した場合を図 4.2(b) に示す。図の中で縦線はモジュールの計算順序を変える時刻を表している。

図 4.2 で注目すべきことは、APR を拡張した ACMS を適用しても単なる APR の時とその全体の計算時間には何の影響もなく、APR の優れた点がそのまま生かされていることである。

フォールトを含んでいるとき

図 4.3 に示すようにモジュール M_{12} でフォールトが発生したとき、APR と ACMS でいつフォールト検出が行われるのかを図 4.4 で示す。

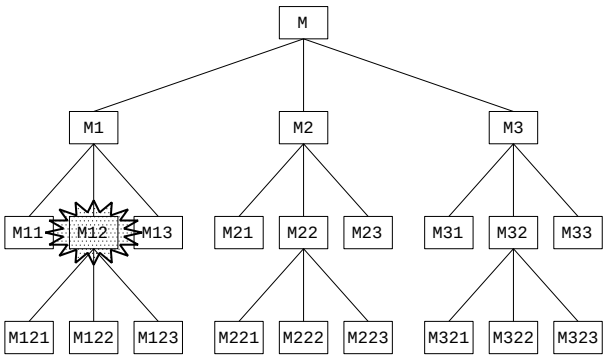
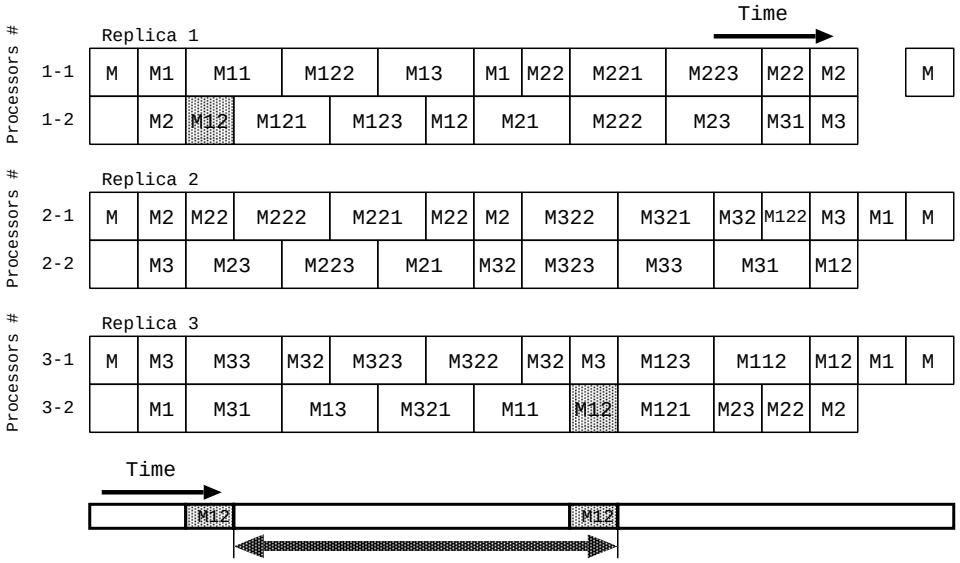


図 4.3: 深さがある計算木:実例 (fault)

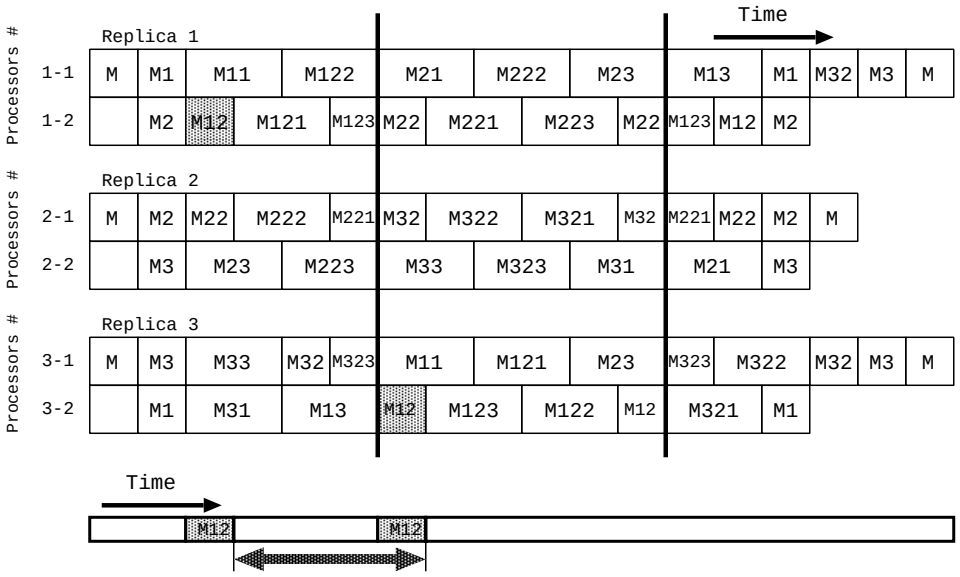
図 4.4 の下に追加したチャートは図 4.2 の上でモジュール M_{12} が実行を行っている部分を示している。

ここでは 1 つの Value failure に対処するために 3 つのレプリカが存在すると仮定しているのでフォールトを検出するためには 2 つ以上の同じモジュール属性が必要となる。図 4.4 の (a) でレプリカ 1 のモジュール M_{12} が実行された後にレプリカ 3 のモジュール M_{12} が実行されるまでの時間と、(b) でレプリカ 1 のモジュール M_{12} が実行された後にレプリカ 3 のモジュール M_{12} が実行されるまでの時間を比べてみる。これはモジュール M_{12} で発生したフォールトが検出されるまでの時間を表している。即ち、この時間が短ければ短いほどフォールトの検出が早く行われ、フォールトの伝搬を抑制することができる。

APR を拡張したを適応した (b) の方がレプリカ 3 でのモジュール M_{12} の実行が早く行われていることが分かる。これはモジュール M_{12} のフォールト検出が早いことを意味している。



(a) APR



(b) Adaptive Computation Management Scheme

図 4.4: The Gantt Chart: 一般的な実行と APR の比べ

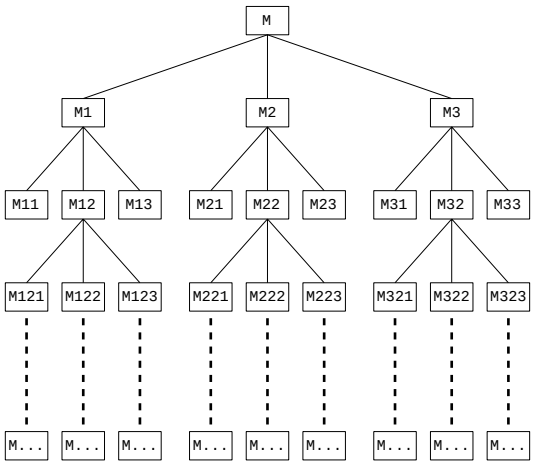


図 4.5: 深さがある計算木:その 3

ここで ACMS を適用したとき深さのある計算木で浅いところにあるモジュールで発生したフォールトが早めに検出されていることに注目されたい。フォールトが早めに検出されるということはそのフォールトの伝搬を抑制してその回復が早く行われるので全体の実行時間が短くなる。

この実例で APR を拡張した ACMS を適用したとき、フォールトがないときは既存の APR での計算木全体の計算時間と変りがないことを理論的に示した。そして、フォールトが計算木の浅いところで発生したとき、そのフォールトの検出が早期に行われることを示した。

4.2 深刻とされる計算木

以上では特定の計算木を用いて ACMS の適用を行った。ここでは図 4.5 のように計算木が深い事を表している深刻とされる計算木を用いて ACMS の適用を行う。

図 4.5 のように計算木の枝が深くなると浅いところで発生するフォールトはその検出が枝の深さに比例して遅くなってしまう。このことはフォールトの伝搬を起こして計算木全体の計算時間に大きな影響を与える。

図 4.5 のモジュール M_{12} でフォールトが発生したとき、そのフォールトがいつ検出されるのかを APR と ACMS に分けて図 4.6 に示す。ここでもレプリカは 3 つ存在し、それぞれのレプリカは 2 つのプロセッサを利用可能と仮定している。

図 4.5 の計算木はそれぞれの枝がかなりの深さを持っているので同じモジュールの出力

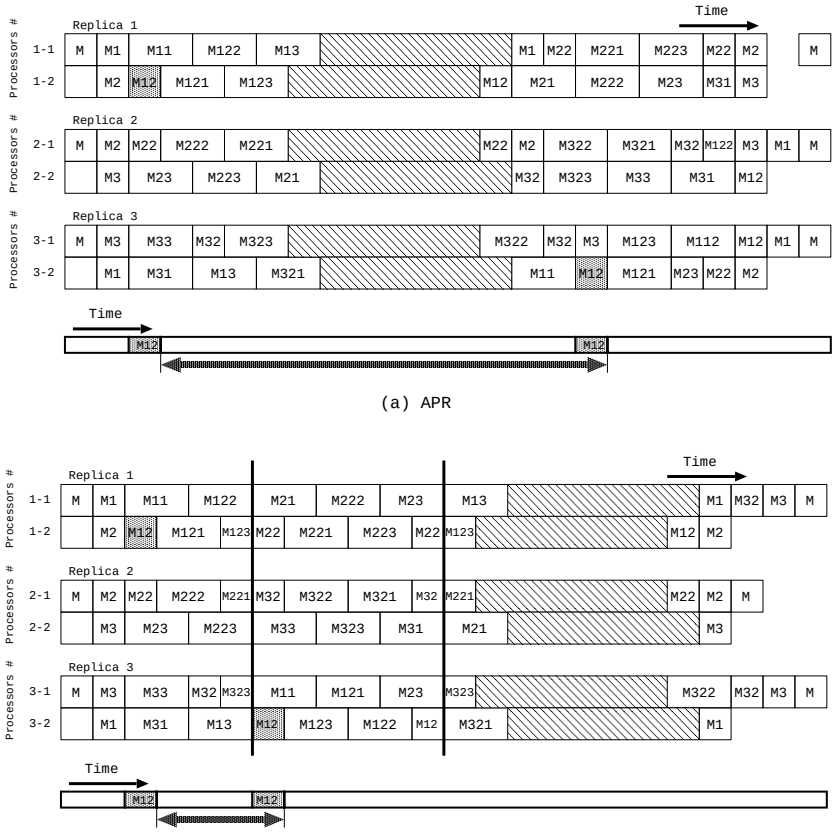


図 4.6: The Gantt Chart

属性が 2 個以上集まるまでには時間がかかる。図 4.6(a) に示したように APR では枝の深さが深ければ深いほどモジュール M_{12} のフォールト検出は遅れる。しかし、ACMS の場合は図の中で (b) に示したように枝の深さには関係なくモジュール M_{12} のフォールト検出が早めに行われる。

図 4.2 と図 4.6 に示したように APR を拡張した ACMS は APR 本来の特徴を生かしながらモジュール計算の早いときに発生したフォールトを効率良く検出していることに注目されたい。

第 5 章

APR 複製技術の実装

5.1 対象とする実装環境

近年低価額な PC やワークステーションが我々の日常生活に普及しており、これらの計算機資源は日常の業務や研究活動に欠かすことのできない存在になってきている。これらの計算機はユーザとのインタラクションを行っている時間を除いては、CPU が稼働していない、いわゆるアイドルの状態であることが多い。CPU 時間に注目してこれらの計算機を見た場合、使用可能な計算資源が多く存在する。

さらに、これらのコンピュータの多くはネットワークで接続されており、稼働率が低い多くのプロセッサがネットワークという物理的な手段で接続された、1 つの大きな計算機システムを形成している。このように分散している計算機資源を有効に利用することは望ましい。

図 5.1 に APR が実行環境の対象としているシステムの構成図を示す。複数の計算機がネットワークを介して接続されている。このように、プロセッサが低速な通信路で結合されている環境を想定している。

図 5.1 のそれぞれの計算機は 1 つ以上のプロセッサを持っており、個々のプロセッサはローカルに密に結合されたメモリとプロセッサを一つにまとめてプロセッサと呼ぶことにする。以上のように一連の処理を行ういくつかのプロセッサとそれらのものが低速な通信路で結合されているのを疎結合マルチプロセッサシステムと呼ぶことにする。そして、外部記憶装置 (StableStorage) は 1 台以上の計算機に付属している。

本研究では上記環境で APR の実装を行うための設計を確立し、その実装を行う。さらにこの環境を用いて APR 動作環境の挙動に関する実験と評価を行う。

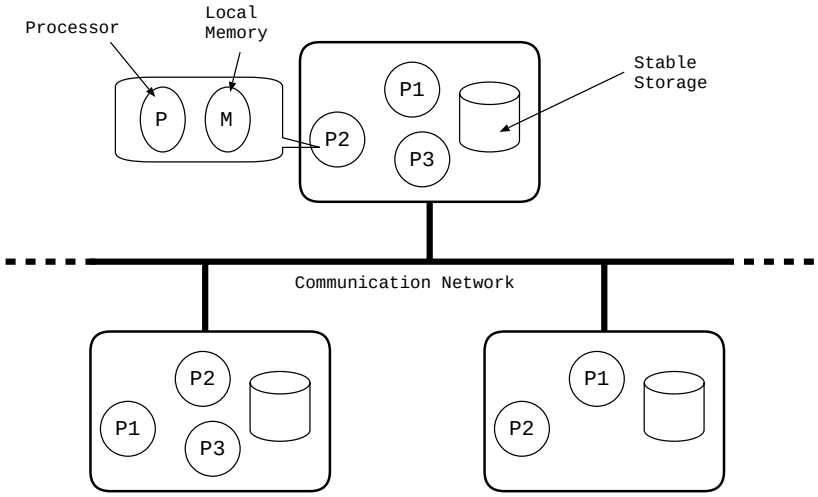


図 5.1: APR が対象とする実装環境

5.2 実装に用いる計算機言語

本実装では関数型プログラミング言語である Objective Caml（以下 OCaml）を用いて行う[1] 。我々が実装のターゲットとしている疎結合分散環境は様々なプロセッサタイプや OS を持つノードがネットワークを介して結合している環境を仮定している。OCaml はバイトコートインタプリタによる実行をサーポートしており、上記のような異機種混在環境においてもバイトコードを用いて実行モジュールを共有することができる。

APR 複製技術では耐故障性を実現するためにレプリカ間の通信が必要である。我々はレプリカ間の通信を行うため Ensemble グループコミュニケーションシステムを用いる。Ensemble は OCaml のインターフェースを持つことに加え、OCaml のユーザ定義のデータ型のデータをグループコミュニケーションのメッセージとして扱うことができる。そして、Ensemble は UNIX RPC のような remote procedure call mechanism を提供してくれる。そこで我々は属性の send/receive に Ensemble を利用する。

5.3 APR 複製技術の機能分析

APR 複製技術は以下のように 7 つの機能によって構成される [1]。

[1]OCaml は現在 Solaris, SunOS, FreeBSD, Linux, DigitalUnix, Windows95/NT などの OS 上で動作する。

Scheduling & Execution 機能 はAPR複製技術を実装するときに一番重要な機能である。APR複製技術ではそれぞれのレプリカで違う計算順序を持っていて、それに従ってモジュールの計算を行う。そのような ことをScheduling & Execution機能という。

Computation Tree 管理機能 はローカル計算木の状態を維持する。モジュールの分解が起きると新しくできたモジュールを計算木に追加する。そ して、計算木はフォールトが検出されたときなど再実行のために親モジュルの検索に利用される。そして、フォールトのあるモジュールを計算木の枝から削除する。

System Configuration 機能 は利用可能なシステムのエレメントを管理する。Scheduling & Execution機能がモジュールの実行を行うときにSystemConfiguration機能を利用して利用可能なシステムエレメントにモジュールを割り当てる。

Fault Detection と Recovery 機能 はAPR複製技術で耐故障性を実現するための機能である。フォールトの検出とその回復を行う。回復には場合によってforward Recovery, Backward Recoveryなどを行う。

Stable Storage Managerment 機能 は安全な保存装置に属性を読み書きするための機能である。安全な保存装置にはモジュールの属性とモジュルを識別するためのデータなどが保存される。

Communication 機能 は複数存在するシステムエレメントの間、そしてそれぞれのレプリカの間で通信を行うための機能である。レプリカの中で計算されたモジュールの属性は他のレプリカに送られる。そのときCommunication機能が利用される。システムエレメントの間でもその属性の通信に利用される。

以上の7つの機能の内Communication機能を除いた6つの機能は実装を行うために図5.2に示すように分けられる。Communication機能はEnsembleと言うミドルウェアを利用するのでここでは除いてある。

図5.2の左はAPRの機能を表し、右は実装を行うためにその機能ことに分けたものである。実装を考慮した機能分析は図の中に示されているようにComputation Manager, FT Manager, Attribute Storeで構成される。Computation Managerは実行状態管理部として実行順序のスケジューリング、計算木のノードの管理そしてCrash Failure Modelに対応するための機能を持つ。Computation ManagerはAPR複製技術のアーキテクチャの中で一番重要である。FT ManagerはValue Failure Modelに対応するためのものでフォール

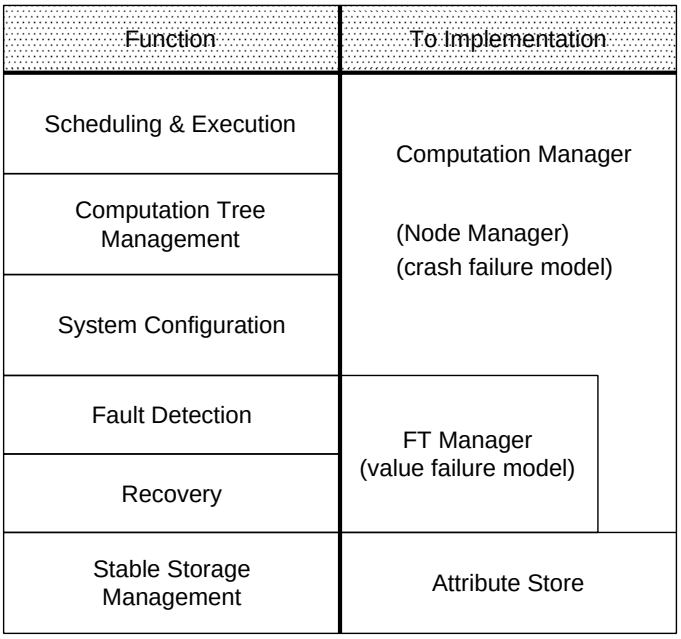


図 5.2: APR 機能の分析

トの検出などを行う。Attribute Store は計算されたモジュールの属性値とそのモジュールを識別するための情報を一緒に保存する。以下でこれらの詳しい説明を行う。

5.4 ソフトウェアの構成

APR のソフトウェアの構成を図 5.3 に示す。

APR システムはランタイムシステムとプログラムコンバータからなる。ユーザからシステムに入力されるのは APR プログラムであり、APR プログラムは FTAG プログラムを APR で幾つかのパラメータ（複製の個数に関するパラメータ、使用可能な計算機資源に関するパラメータなど）を扱えるように拡張したものである。入力されたプログラムはプログラムコンバータにより OCaml のコードに変換され、APR ランタイムシステムとリンクされる。

APR のランタイムシステムには以下の 4 つのコンポーネントが含まれている。

- Start-Up Module
- Computation Manager

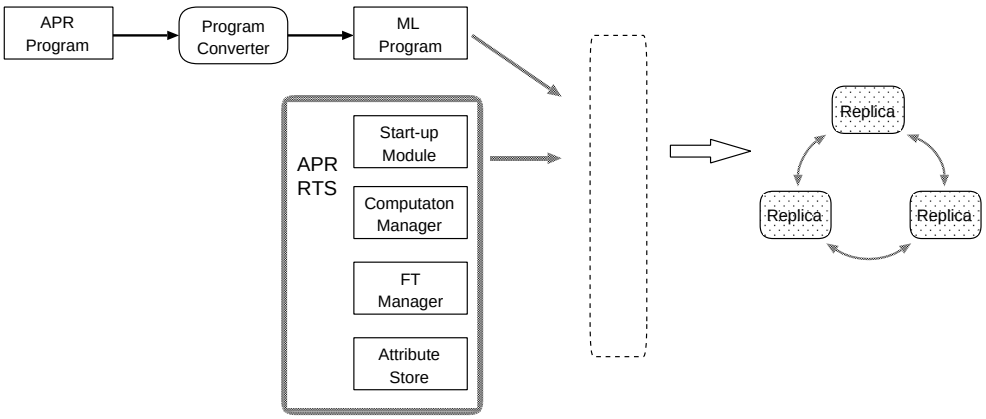


図 5.3: ソフトウェアの構成

- FT Manager
- Attribute store

この 4 つの APR ランタイムシステムのコンポーネントと変換された APR のプログラムはパラメータを基に幾つかのレプリカに複製される。そしてそれぞれのレプリカは独立にモジュールの計算を始める。

図 5.4 に一つのレプリカでデータがどの様に流れるのかを示す。そして、APR ランタイムシステムの 4 つのコンポーネントについて以下で説明する。

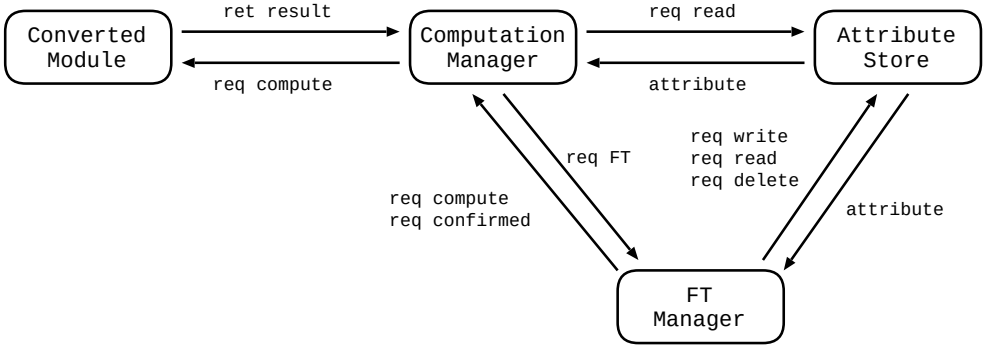


図 5.4: APR ソフトウェアのデータの流れ

5.4.1 Start-Up Module

一つのアプリケーションプログラムが実行を始めると最初に Start-Up Module はパラメータを基に必要な数のレプリカを作る。パラメータには対応可能なフォールトの数が記入されていて、$2n+1$ 個のレプリカを作る（n は対応して欲しいフォールトの数）。そして、システムの中でそれぞれのレプリカを識別できるようにするための唯一の識別番号 rid を与える。そして、それぞれのレプリカがルートモジュールの計算を始めるように Computation Manager を呼び出す。

5.4.2 Attribute Store

Attribute Store は計算によって得られる出力属性値を保存するためのものであり、APR でフォールト検出とボーティングを行うために機能する。自分のレプリカで計算によって得られた出力属性値と他のレプリカで計算され送られてきたモジュールの出力属性値を保存するためのものである。そして Attribute Store の主な機能は出力属性値の読み書きと削除である。

Attribute Store に保存される出力属性値の構造と Attribute Store が持つオペレーションを図 5.5 に示す。

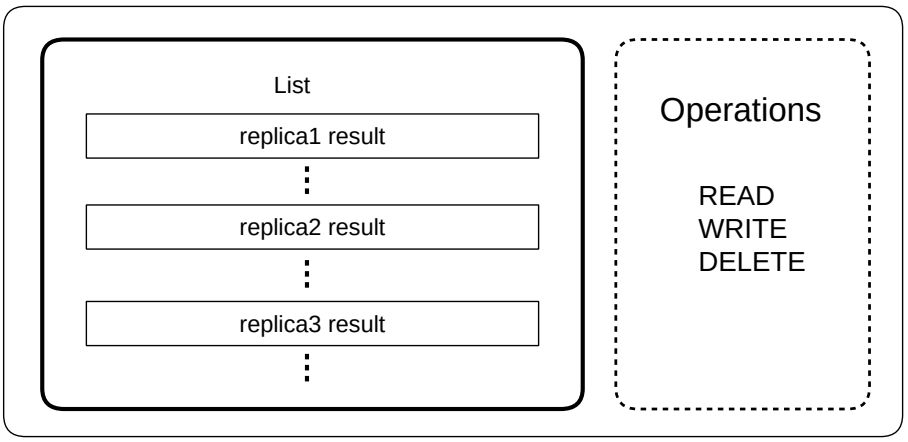


図 5.5: Attribute Store

図 5.5 で示しているように Attribute Store の属性値の構造はリスト構造をしている。そのリスト構造の中には自分のレプリカで計算によって得られたモジュールの出力属性値と他のレプリカから送られてきたモジュールの出力属性値が保存される。ここに保存される

モジュールの出力属性値には識別するためにレプリカの識別番号である rid とモジュールを識別するための uid が一緒に保存される。

プログラムの中でリスト構造とその属性値は、以下に示すタイプを用いて表される。

```
type result = { rid   : int ;          (* レプリカ識別番号 *)
                uid   : int ;          (* モジュール識別番号 *)
                attr1 : float ;        (* 入力属性 *)
                attr2 : float };;      (* 出力属性 *)

type t_list =  result list;;
```

5.4.3 FT Manager

APR 複製技術のアーキテクチャで耐故障性の機能を提供するのはFT Manager である。FTManager の振る舞いを大きく分けるとフォールト検出とボーティングの二つに分けられる。モジュールの出力属性値が送られて来るとFT Manager はアクティブになり、送られてきた出力属性値を基にフォールト検出とボーティングを行う。FT Manager は、新しい出力属性値が現れたときに Computation Manager より呼び出されてアクティブになる。即ち、自分のレプリカでモジュールの計算によって出力属性値が得られた時と他のレプリカから出力属性値が送られて来たときに Computation Manager より起動される。

FT Manager の振る舞いを図 5.6 に疑似コードを用いて示す。

FT Manager が呼び出され、属性値が入力されるとFT Manager は Attribute Store から同じモジュールの出力属性値を持つ値を読む（`read`）する。そして `read` によって読み込んだ値の長さを `n` に入れる。これによって変数 `n` には計算されたレプリカが異なる同じモジュールの属性値の個数が入ることになる。次にその `n` が多数決の数を満たしているかを調べる。もし多数決の数より小さいときは Attribute Store にレプリカ ID、モジュール名、出力属性を書き込んで実行を終了する。もし、多数決の数より大きければ、以前 read したリストのなかに同じ属性がいくつ含まれているかを調べる。これが count である。count された数が多数決を満たしているかを調べる。ここで多数決が取れたならばそのモジュールはどのレプリカでも計算をする必要がなくなり、まだ計算を行っていないレプリカにそのモジュールの計算を行わないように知らせる。

図 5.7 にFT Manager を中心として Attribute Store と Computation Manager がどの様に動作を行うのかをインタラクション図を用いて示す。実線の矢印は多数決が取れていないときの動作を示している。最初に ComputationManager がFT Manager を呼び出しモ

ジュールの出力属性を渡す（図の中ではft)。FT Managerは渡されたモジュールと同じ名前と出力属性を持つ要素をAttribute Storeに返すように要求する（read)。この要求に対してAttribute Storeは値を返す。図の例ではFT ManagerはAttribute Storeから返された値を比べて多数決に満たしていないのでComputation Managerから渡されたモジュールの出力属性値とレプリカid、モジュールidと一緒にAttribute Storeに書き込んで終了する。

図中の点線は多数決が取れたときのインタラクションを示している。多数決が取れた後、FT ManagerはComputation Managerに対して属性値の確認（confirm）を送っている。

5.4.4 Computation Manager

Computation ManagerはAPRランタイムシステムの中で一番重要なコンポーネントである。Computation ManagerではAPRの特徴であるそれぞれのレプリカで違う計算順序で計算を行うことを実現している。

Computation Managerはメッセージを他のレプリカにブロードキャストするSender関数、他のレプリカから送られてきたメッセージを受け取るReceiver関数、モジュールの計算順序を決めその順序にしたがって実行を指示するScheduling & Computation関数の３つの関数で構成される。

図5.8にComputation Managerを中心としたデータの流れを示す。

あるモジュールの計算が終わるとその出力属性はSender関数とFT Manager に送られる。Sender関数に送られてきた出力属性は他のレプリカにブロードキャストされる。FT Managerに送られてきた出力属性はFT Managerによって同じモジュールの出力属性（異なるレプリカの）が多数存在するのか調べられる。FT Managerからフォールトが検出されたモジュールの再実行の要求がSender関数に入るとSender関数は他のレプリカに対してレプリカIDとモジュール名をブロードキャストする。

他のレプリカからブロードキャストされたメッセージをReceiver関数が受け取る。Receiver関数が受け取るメッセージは他のレプリカで計算されたモジュールの出力属性とモジュール再実行の要求である。そして、そのメッセージがモジュールの出力属性であればFT Managerにモジュールの出力属性を送る。送られてきたメッセージがモジュールの再実行の要求であればScheduling & Computation関数にモジュールの再実行を要求する。

Scheduling & Computation関数にはFT Managerから再実行の要求と計算を行う必要がないと確認されたモジュール名が送られてくる。再実行の要求は、FT Managerでフォールトが検出された時、フォールトを含んでいるモジュールのレプリカIDが自分のレプリ

カ ID と一致した場合にモジュールの再実行を要求する。計算を行う必要がないモジュール名が送られてくることは、FT Manager でボーティングによって出力属性の多数決が取れたときである。

以下で Computation Manager の 3 つの関数について説明する。

Sender 関数

Sender 関数はメッセージをブロードキャストするだけの関数で、Ensemble を用いて行う。図 5.9 に Sender 関数で扱うメッセージの内容と振る舞いを疑似コードで示す。

Receiver 関数

Receiver 関数は他のレプリカがブロードキャストしたメッセージを受け取る。受け取ったメッセージの内容が計算されたモジュールの出力属性であればその属性を FT Manager に送る。受け取ったメッセージが再実行の要求であればまずレプリカ ID を調べて、自分のレプリカと一致した場合、Scheduling & Computation に再実行を要求する。レプリカ ID が一致しない場合はメッセージは捨てられる。図 5.10 に Receiver 関数の疑似コートを示す。

Scheduling & Computation 関数

Scheduling & Computation 関数はモジュールの実行を管理する関数である。Converted Module のルートモジュール M を呼び出すことで実行を始める。ルートモジュール M が幾つかのサブモジュールへの分解を行うと、ルートモジュールは分解したモジュール名と入力属性を Scheduling & Computation 関数に返す。分解されたモジュールには実行順序を表す番号を付ける。この計算順序を決める番号付けのアルゴリズムを図 5.11 に示す。

図 5.11 のアルゴリズムはそれぞれのレプリカでそれぞれのモジュールに異なる番号を付ける。その模様を図 5.12 に示す。図中で計算木の枝についている四角はモジュールを表している。そして、それぞれのレプリカで分解されたモジュールの順序と計算を行う順序（”( )”で囲んでいる）を四角の中に書いている。Scheduling & Computation 関数は計算を行う順序が小さい順でそのモジュールを呼び出して実行を行う。

```
Procedure

    val M: module name;
        m: majority;
        x: input attribute;
        y: output attribute;
        L: list;
        n: length of L;

begin
    L = read(M,x) ;          (* Attribute Storeからデータを読み取る *)
    n = length(L) ;
    if ( m > n )
        then { write(M,x,y) }              (* 多数決が取れていない *)
        else { p = count(y, L)
               if ( p >= m)
                  then { -> confirmed(M,x) }    (* ボーティング成功 *)
                  else {if (other majority)     (* フォールト検出   *)
                           then computer(M,x)   (* 再実行の要求     *)
                           else write(M,x,y) }
             } ;
end ;
```

図 5.6: FT アルゴリズム

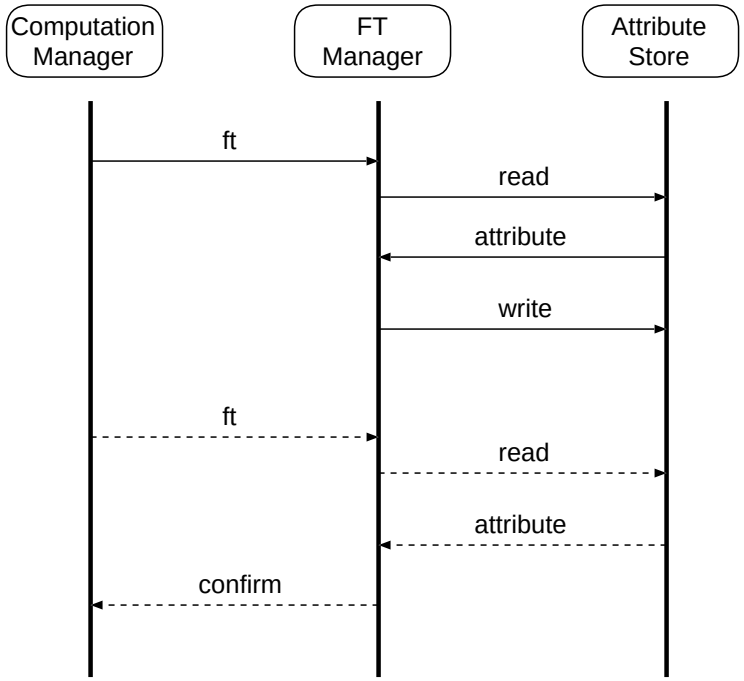


図 5.7: インタラクション図

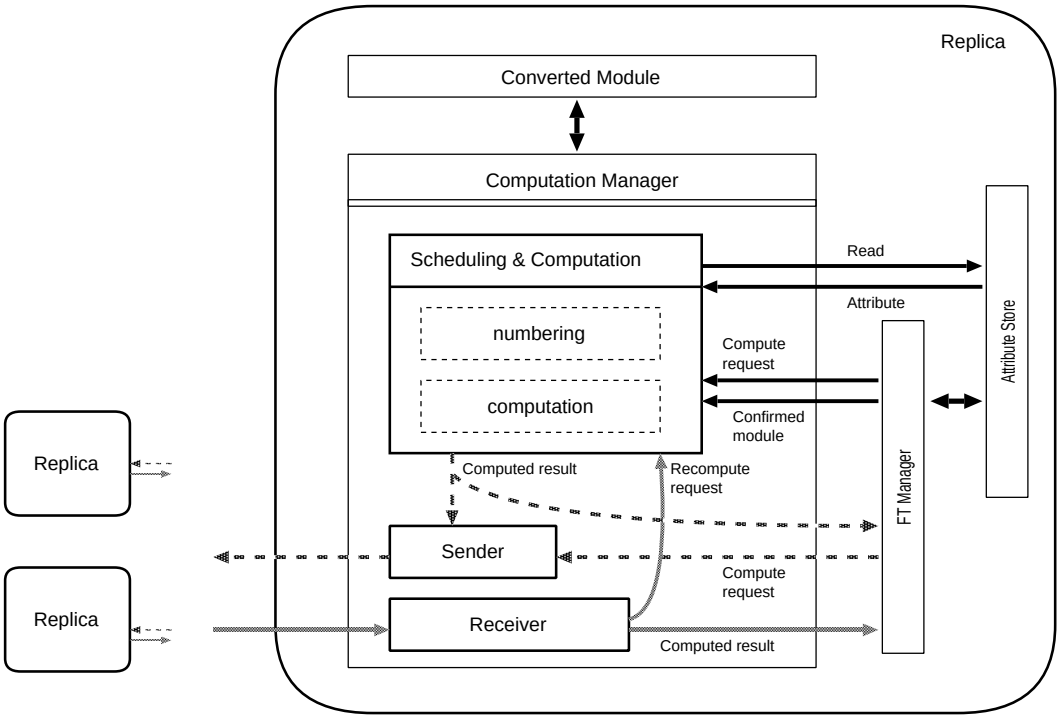


図 5.8: データの流れ：Computation manager

```
function Sender

  val  output_attribute =
              { replica name, module name, output attribute };
       decomposition_result =
              { (module name, input attribute), ... };
       computed  = output_attribute | decomposition_result;
       recompute = { replica name , module name };
       result = computed | recompute;

  broadcast(result);

end; (* Sender *)
```

図 5.9: Sender 関数の疑似コード

```
function Receiver

  val  output_attribute =
              { replica name, module name, output attribute };
       decomposition_result =
              { (module name, input attribute), ... };
       computed  = output_attribute | decomposition_result;
       recompute = { replica name , module name };
       result = computed | recompute;

  match result with
      computed  -> FT Manager (computed)
    | recompute -> scheduling (recompute);

end; (* Receiver *)
```

図 5.10: Receiver 関数の疑似コード

```
function Compute_order

  val  rid = replica ID ;;
       D_list = [ decomposed module list ] ;;

  plist := let rec proces1 D_list =
              [] -> []
            | (a::l) -> ( a - (rid-1) )::l ;;
  let rec process2 plist =
      [] -> []
    | (a::l) -> ( if (a<0) then (a + length(plist)) )::l ;;

end; (* Compute_order *)
```

図 5.11: 計算順序を決めるアルゴリズム

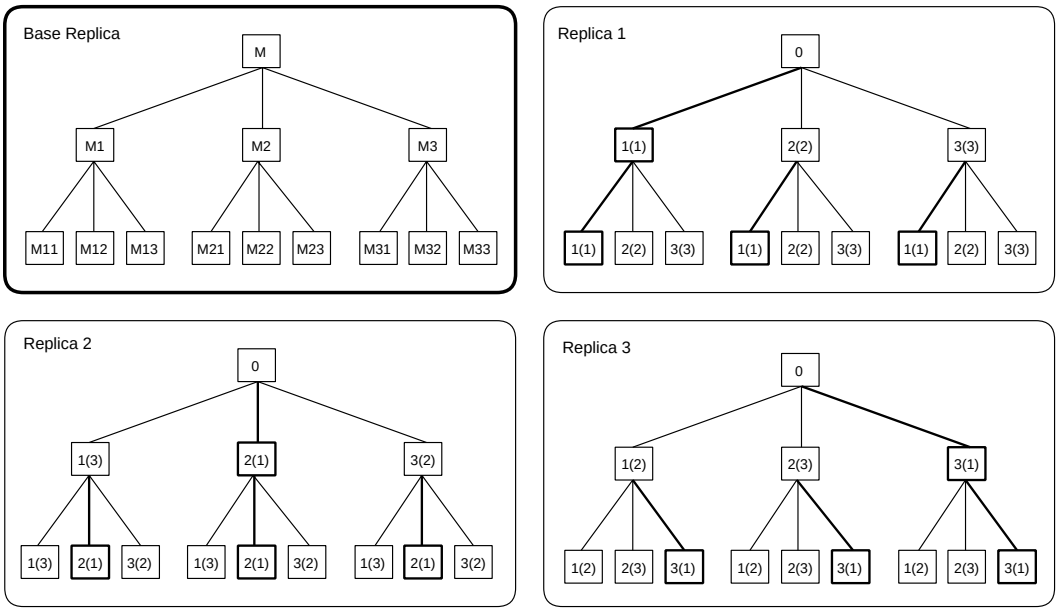


図 5.12: 計算順序

第 6 章

おわりに

6.1 まとめ

本研究では、並列度の高いモジュール複製技術である APR 技術が抱えている問題点を指摘し、その改善方法を提案した。この問題点というのは、計算木の形によっては計算木の浅いところで発生したフォールトは検出が遅れて、フォールトの回復後の計算木全体の計算時間が長くなる問題である。

この問題点に対応するために新しく提案した ACMS は APR 複製技術を拡張するもので、1 つのレプリカの中で一定の時間で計算する部分木を変えることで、計算木の浅いところで発生するフォールトを早期に検出し、その回復を行うことでフォールトによる計算木全体の計算時間に対する影響を少なくするものである。

APR 複製技術についてはその論理的な定義が完成し、マルチプロセッサシステム上への実装の容易性が指摘されているものの、設計と実装に関してはまだ具体例が与えられていなかった。そこで、本研究では APR 複製技術を疎結合マルチプロセッサシステム上への実装を考慮した設計を行った。まずはじめに、設計に関してはその具体例がないため APR 複製技術の機能分析から始めた。

APR 複製技術の設計ではプログラマが耐故障性に関しては特別に意識をしなくても耐故障性をもつソフトウェアの実現ができるように設計した。すなわち、計算プログラムは APR プログラムと APR ランタイムシステムで構成される。計算を行う APR プログラムに対して APR ランタイムシステムは耐故障性を実現するように設計し、耐故障性を持つプログラム環境を実装した。

6.2 今後の課題

疎結合マルチプロセッサシステム上での並列計算を可能にするPVMなど他の耐故障性技術とAPRを比較し、フォールト検出、フォールトからの回復に要するコストなどに関する性能評価を行う。

本研究で提案したフォールトの伝搬を抑制する方法(ACMS)の中で、計算する部分木を変えるタイミングについて検討を行う必要がある。本研究を行った段階ではそのタイミングについては、APRプログラムを書くプログラマに委せられている。すなわち、プログラマがAPRプログラムを書く時にモジュールの計算時間を予測し、パラメタとして書き込むことにしている。しかし、部分木を変えるタイミングはさまざまな計算例を用いてシミュレーションを通じて求める。これによってプログラマが耐故障性を実現するために考慮するべき要素が減ることになる。即ち、プログラマが書くべき耐故障性に関するパラメタが一つ減ることになる。

謝辞

本研究全般にわたってご指導を頂いた片山卓也教授に感謝致します。APR,FTAG そしてFTに関する様々なことを私に教えてくださったAdel Cherif助手と東京工業大学の鈴木正人助教授に感謝します。また、ゼミで私に指導してくださった研究室のみなさんに感謝します。

参考文献

[1] A. Cherif : "Replication for Fault Tolerant Software Using a Function and Attribute Grammar Based Computation Model", Doctor's thesis, Japan Advanced Institute of Science and Technology, March 1998.

[2] Jean-Claude Laprie, editor. Dependability: Basic Concepts and Terminology, pp.210-245. IFIP WG 10.4 Dependable Computing and Fault Tolerance. Springer-Verlag Wien New York, 1992.

[3] T.Katayama. HFP, a hierarchical and functional programming based on attribute grammars. In *Proceedings of the Fifth International Conference on Software Engineering,* pp.343-353, 1981.

[4] M. Suzuki, A. Iwai, and T. Katayama. A formal model of re-exection in software process. In *Proceedings of the Second Internatiocal Conference on Software Process,* pp. 84-99, Berlin, Germany, Feb 1993

[5] B.Randell. System structure for software fault tolerance. IEEE Transactions on Software Engineering, Vol. SE-1, No.2, pp. 220-232, Jun 1975.

[6] Dimiter R. Avresky, David R. Kaeli, editer. : FAULT-TOLERANDT PARALLEL AND DISTRIBUTED SYSTEMS, chapter 22: A NOVEL REPLICATION TECHNIQUE FOR IMPLEMNTING FAULT-TOLERANT PARALLEL SORTWARE (A. Cherif, M. Suzuki and T. Katayama). Kluwer Academic Publishers, 1998.

[7] 豊島 真澄, 高信頼性ソフトウェアの実装環境の構築に関する研究, 修士論文、 北陸先端科学技術大学院大学、 March 1998.