Duplication with Temporary Triple Modular Redundancy and Reconfigurationのための タスク割り当て手法

齋藤　寛[†1,a)]　米田　友洋[†2,b)]　今井　雅[†3,c)]

概要：本稿では，Duplication with Temporary Triple Modular Redundancy and Reconfiguration (DTTR) のためのタスク割り当て手法を提案する．提案手法は，ネットワークオンチップモデル (NoC) における故障コア数の上限値より故障パターンを列挙し，パターン毎にタスクスケジューリングを行うことでタスク割り当てを決定する．実験では，NoC やタスクグラフのサイズ，故障コア数を変えながら提案手法を適用することで，タスク割り当て時間を評価した．現実的な時間制限の中，提案手法は 4x4 の NoC であれば，故障コア数が 6 つ，100 のタスクを持ったタスクグラフにも対応することができる．

A Task Allocation Method for Duplication with Temporary Triple Modular Redundancy and Reconfiguration

Hiroshi Saito[†1,a)]　Tomohiro Yoneda[†2,b)]　Masashi Imai[†3,c)]

Abstract: In this paper, we propose a task allocation method for Duplication with Temporary Triple Modular Redundancy and Reconfiguration (DTTR). The proposed method determines the task allocation for a given network-on-chip (NoC) model from the task scheduling results of fault patterns of cores enumerated from the upper bound of fault cores on the NoC. In the experiments, we evaluate the task allocation time while changing the size of NoC, task graph, and upper bound of faulty cores. From the experimental results, we found out that the proposed method can deal with task allocation of the task graph with 100 tasks for 4x4 NoC with 6 faulty cores in a realistic time limit.

1. はじめに

最近の組込みシステムにおける集積回路は，半導体微細化技術の向上によって複数のプロセッシングコアからなるマルチコア，メニーコアシステムとして実現されている．しかしながら，製造プロセスや劣化等によって生じる欠陥も顕著となるため，システムの信頼性に大きな影響を及ぼす．

集積回路は，主に冗長な回路を持たせることで信頼性を確保する．最も一般的な手法の 1 つは，車載アプリなどで採用されている Lock-Step と呼ばれる二重化方式である．２つのコアで同じ処理を行った後比較を行い，結果が異なれば異常を知らせる．Lock-Step は，回路構造がシンプルで故障検出が速いのが特徴である．しかしながら，故障したコアを特定することができないため，故障した後に動作を継続することができない．もう 1 つは，Triple Modular Redundancy (TMR) と呼ばれるコアの三重化である．３つのコアで同じ処理を行った後に多数決を行うことで，故障したコアを特定することができる．しかしながら，コアの三重化はコストが高い．

マルチコア，メニーコアシステムは，多数のプロセッシングコアより構成されているので，多重実行がしやすく故障検出や故障個所の特定がしやすい．著者らは [1] で，高信頼なマルチコア，メニーコアシステムを実現するために Duplication with Temporary Triple Modular Redundancy

†1 現在，会津大学
Presently with The University of Aizu
†2 現在，国立情報学研究所
Presently with National Institute of Informatics
†3 現在，弘前大学
Presently with Hirosaki University
a) hiroshis@u-aizu.ac.jp
b) yoneda@nii.ac.jp
c) miyabi@eit.hirosaki-u.ac.jp

and Reconfiguration (DTTR) と呼ばれる手法を提案した．DTTR では，通常はタスクを二重で実行し，結果を比較する．結果が異なる場合は，故障個所の特定のために，もう 1 つのコアを使って三重実行を行う．Lock-Step によるシステムや常時 TMR を行うシステムと比べ DTTR では，信頼性，処理時間，コストの面でバランスの良いものを実現することができる．DTTR を実現するためには、あらかじめタスクを異なるコアに多重で割り当てておく必要がある．

本稿では，DTTR のためのタスク割り当て手法を提案する．提案手法は，入力として与えた故障コア数の上限値の範囲で，任意のコアの故障を表した故障パターンを列挙する．次に，故障パターン毎にタスクスケジューリングを行い，どのコアを使うかを求めることによってタスク割り当てを決める．タスクスケジューリング結果を基にタスク割り当てを決めるので，アプリケーションの実際の実行時間が短くなることが期待できる．また，タスクスケジューリングより，故障パターン毎に DTTR におけるタスクの属性を決める．本稿では，DTTR を実現するメニーコアプラットフォームとしてネットワークオンチップ (NoC) [2] を用いる．

本稿の構成は，以下のとおりである．2 節では，関連研究の解説を行う．3 節では，アプリケーションを表すタスクグラフ，NoC，DTTR の解説を行う．4 節では，提案手法の詳細を述べる．5 節では，実験結果を述べる．最後に 6 節では，まとめと今後の課題を述べる．

2. 関連研究

本節では，関連研究としてコアが故障した時にタスクを再割り当てする手法，あらかじめ故障を想定してタスクを多重に割り当てる手法を解説する．

Ababei らは [3] で，あるコアが故障した時，故障前後の通信時間の差が最小となるよう残りのコアに動的にタスクを再割り当てる手法を提案した．Derin らの手法 [4] は，整数線形計画法を用いて計算時間と通信時間の両方が最小となるようにタスク割り当てを行う．あるコアが故障したら，発見的手法を用いて動的に残りのコアにタスクの再割り当てを行う．Lee らの手法 [5] は，前もって全ての故障パターンに対してスループットが最大になるようタスクスケジューリングとタスク割り当てを行う．仮にもしあるコアが故障したら，前もって計算された情報を基に動的に割り当てを行う．最初の 2 つの手法は，故障検出後にタスク割り当ての計算とタスクのローディングが必要となり，最後の手法はタスクのローディングが必要となる．しかし，リアルタイム性が重要なアプリケーションにおいては，こうしたオーバーヘッドは深刻な問題となることが予想される．

著者らは [6] で，コアの故障後ではなく，あらかじめ空いているコアにタスクを冗長に割り当てる手法を提案した．この手法は，故障コア数の上限値の範囲でコアの故障パターンを全列挙し，アプリケーションの実行時間の最小化や実行可能な故障パターンの最大化を整数線形計画法を用いて解く．また，規模の大きなアプリケーションや沢山のコアを含んだシステムを扱うために，著者らはヒューリスティックを用いた手法を [7] で提案した．しかし，故障パターンが増えると，問題を解くために多大な時間を要する．

提案手法は，[7] に対して 2 つの改良を加える．1 つは，DTTR の動作を考慮してタスク割り当てを行う．もう 1 つは，故障パターンの選び方を変えることによってタスク割り当てを効率的に行う．

3. 背景

3.1 NoC

NoC は図 1 のように，コア，ネットワークインターフェース（NI），ルーター，チャネルから構成される．コアはプロセッサなどから構成されタスクを実行し，必要に応じて他のコアと通信を行う．送受信データはパケットとして表され，パケットは NI を介してルーターから構成されるネットワークに送られ，ルーターでルーティングアルゴリズムの下，受信先のコアにパケットを転送する．各コアがデータを並列に転送することができ，コアの数が増えても性能が良いということで，マルチコア，メニーコアシステムの通信手段として注目を浴びている．

提案手法で用いる NoC モデル N は，コア c_u $(0 \leq u \leq |C|-1)$ の集合 C と通信コストを概算するためのパラメータ，および故障コア数の上限値 $fnum$ からなる．コア c_u は 4 つの要素 $\langle type, mem, port, Comm\rangle$ から構成される．$type$ はコア c_u のタイプでプロセッサやアクセラレータの他に，I/O 処理，DTTR のための故障診断やコアの再構成を管理する Diagnostics and Reconfiguration (DnR) コアを表す．mem はコア c_u のローカルメモリサイズ，$port$ はコア c_u の I/O ポート数を表す．なお，対象となる NoC はコアごとに分散メモリを持ち，共有メモリを持たないことを想定している．$Comm$ は，コア c_u から他のコア全てに対する通信パラメータ $comm$ の集合を表す．$comm$ は 2 つの要素 $\langle c_v, cdelay\rangle$ から構成され，コア c_u からコア c_v までの通信時間 $cdelay$ を表す．$cdelay$ は，コア c_u からのヘッダーフリットがコア c_v に届くまでの時間を想定している．フリットは，パケットをチャネルのビット幅に合わせて分けたデータ転送における最小単位を表す．提案手法では，コア c_u と c_v 間の通信時間は，$cdelay + (\lceil s/cb \rceil - 1) * interval$ より求める．s はパケット量，cb はチャネルのビット幅，$interval$ はヘッダーフリット以降のフリットが受信先コアに届くまでの時間を表す．

3.2 タスクグラフ

提案手法は，アプリケーションモデルとしてタスクグラフを用いる．本稿で用いるタスクグラフ $G\langle T, E, period\rangle$ は，3 つの要素から構成される．T はタスクの集合を表す．あるタスク $t_i\langle io, r, cmp, CS\rangle$ $(0 \leq i \leq |T|-1)$ は，4 つの要素から構成される．io が $true$ の場合，そのタスクが

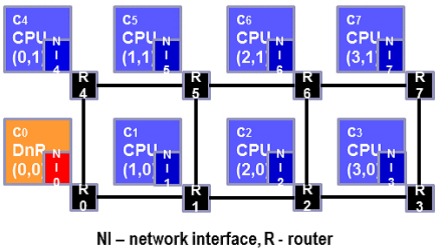

図 1　NoC モデル

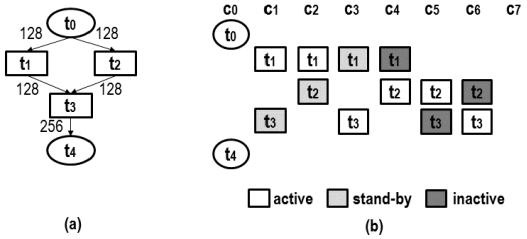

図 2　(a) タスクグラフと (b) タスク割り当て例とタスクの属性

I/O タスクであることを表し，さもなければそのタスクは処理タスクであることを表す．I/O タスクは，I/O ポートを介してデータの送受信を行う．一方，処理タスクは演算を行う．r はタスクの多重度，すなわちタスク t_i のコピー数を表す．cmp が $true$ の場合，処理タスク t_i の 2 つのコピーを異なるコアで実行し，その結果を比較することを表す．CS は，t_i を実行することができるコアの集合を表す．t_i を実行することができるコア $c_u \in CS$ は，3 つの要素 $\langle type, delay, code \rangle$ から構成される．$type$ はコアのタイプを表す．$delay$ と $code$ はそれぞれ，タスク t_i を $type$ で示されたコアで実行した時の時間とコード量を表す．E はエッジの集合を表す．あるエッジ $e_k\langle t_i, t_j, s \rangle (0 \leq k \leq |E|-1)$ は，タスク t_i からタスク t_j にデータ依存があることを表す．s はタスク t_i からタスク t_j へのデータ転送におけるパケット量を表す．$period$ は，アプリケーションの実行周期を表す．なお，本稿では周期タスクによるアプリケーションのみを扱う．周期 $period$ は，タスクスケジューリングの際に時間制約として用いる．

図 2(a) は，タスクグラフの例を表す．丸で表されたタスクは I/O タスクを表し，四角で表されたタスクは処理タスクを表す．エッジに付加されたラベルは，データ転送におけるパケット量 s を表す．

3.3　DTTR

DTTR の説明の前に，タスクの属性を定義する．提案手法によって，各タスク t_i の $r(\geq 3)$ 個のコピーが NoC 上の異なるコアに割り当てられると仮定する．r 個のコピーの内，DTTR では 2 つのコピーが実際に実行されるが，これら 2 つを active タスクと呼ぶ．一方，active タスクを含むあるコアが故障したことによって，active でないタスクを含んだコアの内のいずれか 1 つが実行されることになるが，このタスクのコピーを stand-by タスクと呼ぶ．それ以外のタスクのコピーは inactive タスクと呼ぶ．なお，$r = 3$ の時は inactive タスクは存在しない．図 2(a) はタスクグラフを，図 2(b) は図 1 に示された NoC に対するタスク割り当て例とタスクの属性を表す．この例では，各タスク t_i の r は 4 である．

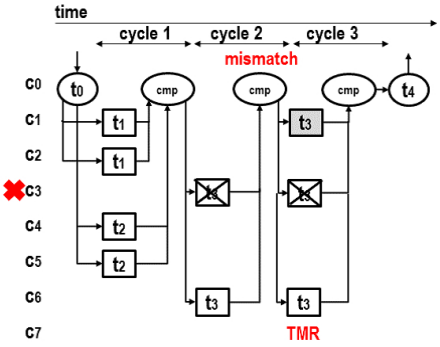

図 3　DTTR の動作

図 3 を用いて DTTR の説明を行う．ここでは，図 2(a) のタスクを実行する．また，タスク割り当てとタスクの属性は図 2(b) で示されたものを用いる．まず，I/O タスクが c_0 で実行される．c_0 は，I/O タスクや故障診断，制御を行う DnR コアである．c_0 は，入力信号を t_0 で受信し，そのデータをタスク t_1 と t_2 が active タスクとして含まれるコア c_1，c_2，c_4，c_5 に転送する．DTTR では，active タスクが二重で実行される．そのため，タスク t_1 がコア c_1 と c_2，タスク t_2 がコア c_4 と c_5 で実行される．各コアの計算結果は，DnR である c_0 に集約され，結果が比較される．次に，タスク t_3 が active タスクとして含まれるコア c_3 と c_6 で実行される．この結果の比較を c_0 で行った時に結果が一致しなかったと仮定する．その場合，DnR コアである c_0 は，タスク t_3 を stand-by タスクが入ったコア c_1 を含む 3 つのコア c_1，c_3，c_6 で実行し結果を比較することで，故障コア（この場合，コア c_3）を特定する．その後，DnR コア c_0 は c_3 が故障したとして，c_3 以外のコアでタスクの処理を行うようタスクの属性を再構成する．すなわち，コア c_1 のタスク t_3 が active タスクとなり，コア c_5 の inactive タスクが stand-by タスクとなる．

コア c_0 だけでなく，c_2 と c_5 もタスク t_3 を再実行することで，DTTR は永久故障だけでなく，一過性故障にも対応することができる．一過性故障の場合，三重実行の結果は全て一致することになる．なお，DTTR は連続する実行サイクル（例えば t_3 の二重実行のサイクルと三重実行のサイクル）でコアが故障することはないといった仮定を設けている．また，タスクにおける処理の結果が DnR に帰ってこないような場合は，別途タイムアウトを DnR に含める必要がある．本稿では DnR は故障しないと仮定しているが，故障を想定した場合 DnR の多重化が必要となる．DTTR の詳細は，[1] を参照されたい．

4.　提案手法

本節では，DTTR のためのタスク割り当て手法を解説する．提案手法は，タスクグラフ，NoC モデルを入力に，

あらかじめ r 個のタスクのコピーを異なるコアに割り当てる．タスク割り当ての結果がアプリケーションの実行時間に影響を及ぼすので，提案手法では，タスクの実行時間，タスク間通信時間，タスクの比較時間を考慮した上でタスクスケジューリングを行い，どのコアを使うかを決めることによってタスクを割り当てる．まず，故障コアのない故障パターンにてタスクスケジューリングを行う．時間制約 $period$ の範囲でタスクスケジューリングが出来た場合は，故障コアのない故障パターンにおいて任意の 1 つのコアが故障した場合の故障パターンを全列挙する．列挙された故障パターン毎にタスクスケジューリングを行い，新たな故障パターンを生成する．全てのタスクにおいて r 個のタスクのコピーが割り当てられた段階でタスク割り当ては終了となる．なお，故障が実際にいつ起こるかはタスク割り当ての段階では判断できない．そのため，タスクスケジューリング時には Temporal TMR は考慮しない．

4.1 故障パターン

NoC モデルと故障コア数の上限値 $fnum$ より，コアの故障パターン $p_l(0 \leq l \leq |P|-1)$ の全てが定まる．P は故障パターンの集合を表す．なお，提案手法では DnR コアは故障しないと仮定する．例えば，図 1 の NoC の場合，$fnum = 2$ とすると故障パターンの総数は ${}_7C_0 + {}_7C_1 + {}_7C_2 = 29$ となる．内訳は，故障コアのないパターン 1 つ，任意の 1 コアの故障パターン 7 つ，任意の 2 コアの故障パターン 21 つである．各故障パターンでは，故障しているコア（故障コア）だけでなく，生存しているコア（生存コア）もわかる．例えば，図 1 の NoC で，コア c_1 が故障した場合の生存コアは，$c_2 \cdots c_7$ である．

次に，パターンの属性を定義する．パターン p_l の生存コアの任意の 1 つのコアが故障したパターンを p_k とする．この時，p_l は p_k に対する親パターン，p_k は p_l に対する子パターンと呼ぶ．一方，あるパターン p_m と p_k の親パターンのうちの 1 つが一致する場合がある．この時，p_m は p_k の兄弟パターンと呼ぶ．なお，あるパターンに対して親パターン，子パターン，兄弟パターンは複数存在する．

4.2 タスクスケジューリングにおける制約

DTTR においてある故障パターンのタスクスケジューリングは，親パターンや兄弟パターンのタスクスケジューリングに制限される．

親パターンの生存コアで動作している最中にあるコアが故障すると仮定する．この時，故障したコアの中にある active タスクだけが影響を受けるが，他のコアに入った active タスクは影響を受けない．こうした背景から，子パターンのタスクスケジューリングは親パターンのタスクスケジューリングをベースとする．親パターンのある生存コアが故障した時，タスクスケジューリングでは故障コアで実行される active タスクが他のコアで実行されるようにスケジューリングし，もう 1 つの active タスクは親パターンと同じコアを使うようにする．図 4(a) の左は，故障なしのパターン p_0 の active タスクが入ったコアを表す．図 4(a) の右は，コア c_1 が故障した時のパターン p_1 におけるスケジューリングの制約を表す．c_1 に入った active タスク t_1 のみが影響を受けるので，t_1 は c_3 から c_7 のいずれかを使ってスケジューリングされる．それ以外のタスクは親パターン p_0 と同じコアを使ってスケジューリングされる．親パターンが複数ある場合は，タスクスケジューリングが最も早く終わった親パターンを子パターンのベースとする．

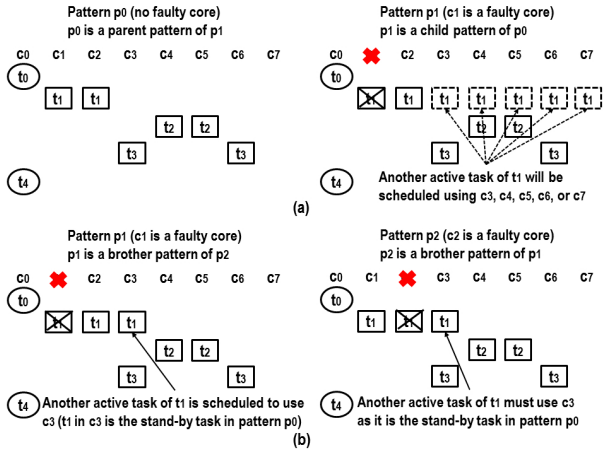

図 4　タスクスケジューリングにおける (a) 親パターンからの制約，(b) 兄弟パターンからの制約

一方，active なタスクが入ったコアのうちいずれか 1 つが故障した子パターンでのタスクスケジューリングによって代わりに使用するコアが決まったら，active なタスクが入った残りのコアが故障した子パターンでも同じコアを使用するようタスクスケジューリングする．これは，兄弟パターンによるタスクスケジューリングの制約である．図 4(b) の左は，p_1 の active タスクを含んだコアを表す．ここでは，パターン p_1 のタスクスケジューリングの結果，c_3 に含まれる t_1 が active タスクになったと仮定する．この場合，パターン p_0 において，c_3 に入った t_1 は stand-by タスクとなる．そのため，コア c_2 が故障したパターン p_2 のタスクスケジューリングでは，パターン p_0 で stand-by タスクとなった c_3 に含まれる t_1 を用いる．図 4(b) の右は，パターン p_2 における active タスクを表す．なお，パターン p_1 と p_2 は共に兄弟パターンである．

4.3 タスクスケジューリングに基づいたタスク割り当て手法

タスクスケジューリングに基づいたタスク割り当て手法の疑似コードを Algorithm 1 で表す．Algorithm 1 の TaskAllocation はメインとなる関数である．始めに，CopyTaskGraph にてタスクグラフ G より二重実行と結果の比較を表したタスクグラフ G' を生成する．図 5 は，図 2(a) のタスクグラフのタスクを二重実行，および比較を挿入したものである．なお，I/O タスクは DnR での処理で二

Algorithm 1 Proposed task allocation algorithm for DTTR

```
1: procedure TaskAllocation(G, N)
2:     G' ← CopyTaskGraph(G)
3:     p_0 ← GenInitialPattern(N)
4:     p_0.Parent ← null
5:     P ← {p_0}
6:     TaskAlloc ← InitializeTaskAlloc(G', N)
7:     while P ne ∅ do
8:         p_l ← SelectPattern(P, TaskAlloc)
9:         p_l.Brother ← ScheduledBrotherPattern(p_l, P)
10:        p_l.valid ← Scheduling(p_l, G', N, TaskAlloc)
11:        if p_l.valid eq true then
12:            if CheckAlloc(G', TaskAlloc) eq true then
13:                return TaskAlloc
14:            end if
15:            p_l.Child ← GetChildPattern(p_l, N)
16:            for all p_c ∈ p_l.Child do
17:                p_c.Parent ← p_l
18:            end for
19:            P ← P ∪ p_l.Child
20:        end if
21:        P ← P − {p_l}
22:    end while
23: end procedure
```

重実行は行わない．次に，故障コアのない故障パターン p_0 を GenInitialPattern にて生成する．故障パターン p_0 の親パターンはないものとする．$TaskAlloc$ は，タスクがどのコアに割り当てられたのかを表したテーブルで InitialTaskAlloc で初期化する．タスク t_i がコア c_u に割り当てられると $TaskAlloc_{t_i,c_u}$ は”allocated”となる．次に，故障パターン集合 P が空になるまでの間，故障パターン毎にタスクスケジューリングを行う．まず，SelectPattern で，これまでに列挙した故障パターンの内，故障コア数が少なく，タスクが割り当てられた故障パターン p_l を選択する．[7] では，タスク割り当ての有無に関係なく故障コア数が少ないパターンを選んでいくが，タスクが割り当てられていないコアが故障コアとして含まれた故障パターンのタスクスケジューリングは意味がない．タスクが割り当てられたコアのみからなる故障パターンを選ぶことで，無駄な故障パターンのスケジューリングを行わずにタスク割り当てを完了することができる．ScheduledBrotherPattern では，これまでにスケジューリングされた兄弟パターンの全てを列挙し，p_l の兄弟パターンの集合 $Brother$ とする．次に，Scheduling で故障パターン p_l のタスクスケジューリングを行う．タスクグラフに示した周期 $period$ の範囲でタスクスケジューリングが完了した場合，CheckAlloc で全てのタスクに関して r 個のコピーが割り当てられたら，$TaskAlloc$ を出力し終了する．さもなければ，故障パターン p_l の生存コアの 1 つが故障したパターンを全列挙し，p_l の子パターンの集合 $Child$ とする．また，子パターン p_c の親パターンの集合 $Parent$ に故障パターン p_l を加える．最後に故障パターンの集合 P を更新する．

Scheduling は，リストスケジューリング [9] をベースにタスクの開始時間と使用する生存コアを決める．ある時間 $time$ において，実行可能なタスクの集合 T_{avail} を求める．

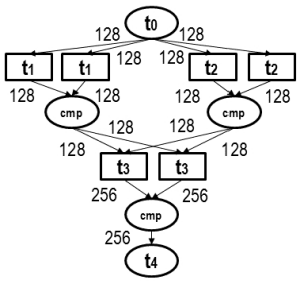

図 5 図 2(a) のタスクグラフから生成された二重実行と比較を含んだタスクグラフ G'

エッジによって依存関係のある直前の全てのタスクが実行を完了し，コア間通信時間が経過した段階でタスクは実行可能となる．同様に，ある時間 $time$ において，利用可能でかつ I/O ポートやメモリが空いているコアの集合 C_{avail} を求める．更に，C_{avail} のコアを 4.2 節で解説した親パターンと兄弟パターンからの制約を考慮し制限する．T_{avail} のタスクを C_{avail} に含まれるコアの範囲で割り当てる．割り当て後，$p_l.Alloc_{t_i}$ を更新する．$p_l.Alloc_{t_i}$ は，パターン p_l でタスク t_i を実行するコアを表す．以上の操作をタスクグラフの周期 $period$ の範囲で行い，スケジューリングできた場合 true を，さもなければ false を返す．また，スケジューリングができた場合，$p_l.Alloc$ より $TaskAlloc$ を更新する．$p_l.Alloc_{t_i}$ が c_u の場合，$TaskAlloc_{t_i,c_u}$ を”allocated”とする．

5. 実験結果

実験では，パターンの選び方によるタスク割り当て時間への影響，および NoC サイズを固定しタスクのコピー数やタスクグラフのサイズを変えたときのタスク割り当て時間への影響を評価する．提案手法は，Java と Eclipse を用いて実装した．実験環境は，Intel Core i5 と 8GB のメモリを含んだ Windows 8 PC である．

NoC は，2 種類用いる．1 つは 4x4，もう 1 つは 6x6 である．うち，各 NoC とも 1 つの DnR コアを含む．コアあたりのメモリサイズは 16K と 32K とする．cb は 32 ビット，コア間の $cdelay$ は，$md = 0$ の時は 0，$1 \leq md$ の時は $(5 * md + 10)$ とする．md は，コア間のマンハッタン距離を表す．タスクグラフの各エッジの重み s を 32 とすることで，コア間の通信時間 $comm = cdelay$ とする．提案手法は現在のところ $fnum = 6$（任意の 6 コア故障) まで対応することができる．

タスクグラフは，TGFF[10] を用いてランダムに生成したものを用いる．graph20, graph40，graph60，graph81，graph100 はそれぞれ，タスク数が 20，40，60，81，100 個のタスクグラフである．入力エッジがないタスク，および出力エッジがないタスクを I/O タスクとする．各タスクの実行時間とコード量はそれぞれ，1 から 200 まで，1 から 1000 までのランダムな値を用いる．I/O タスクのコピー数 r は 1，処理タスクのコピー数 r は 3 から 8 まで変化させる．各タスクグラフの周期 $period$ は 5,000 とする．

表 1　パターンの選び方による影響

		allocation time [s]		patterns	
	r	small	proposed	small	proposed
4x4 16K graph60	3	3.2	3.0	12	12
	4	18.8	18.7	114	114
	5	89.0	90.5	558	558
6x6 16K graph81	3	8.1	7.1	20	15
	4	85.1	41.0	285	115
	5	1,283.2	597.3	4,350	890

表 1 は，パターンの選び方によるタスク割り当て時間とタスク割り当てが決まるまでにタスクスケジューリングが行われたパターン数を表す．proposed は提案手法を表し，タスクが割り当てられたコアのみからなる故障パターンからタスクスケジューリングを行っていく．small は [7] のように，故障コア数の少ないパターンからタスクスケジューリングを行っていく．この結果は，パターンの選び方に効果がある時とない時を示している．4x4 でコア数が 16K の NoC の場合，タスク数やコード量に対してコアの数に余裕がほとんどないため，どのコアにもタスクが割り当てられる．その結果，small と proposed に差がないことが分かる．一方，6x6 でコア数が 16K の NoC の場合，コアの数に余裕があるためタスクが割り当てられないコアがあり，タスク割り当て時間が短くなっている．

図 6(a) は，NoC を 4x4，各コアのメモリを 32K と固定し，図 6(b) は，NoC を 6x6，各コアのメモリを 16K と固定し，タスクグラフやタスクのコピー数を変えたときのタスク割り当て時間を表す．この実験では，時間制限を 2 時間とした．提案手法は，コア数，$fnum$ 数，タスク数，タスクのコピー数が増えると割り当てに掛かる時間が増える．特にコア数と $fnum$ 数が増えることによってパターン数が激増し，6x6 の NoC では 2 時間見てもタスク割り当てが終わらないものがあった．4x4 の NoC で故障コア数が 6，タスク数 100 のタスクグラフであれば，現実的な時間でタスク割り当てを完了することができる．それ以上の場合，例えば複数のアプリケーションを扱いたい場合は，アプリケーション毎に NoC の領域を分割する，タスクの並列性が高い場合はタスクを機能毎にまとめて、4x4 の NoC で扱えるようにすることが望ましい．前者は，大規模な NoC では一般的な方法である．後者は一見大きな制約にみられがちだが，タスクを細かくしすぎると処理時間より通信時間の方が長くなってしまう可能性があるため，現実的な対応策であると考えている．

6. まとめ

本稿では，DTTR のためのタスク割り当て手法を提案した．提案手法は，NoC における故障コア数の上限値より故障パターンを列挙し，全てのタスクが指定したコピー数割り当てられるまで，故障パターン毎にタスクスケジューリングを行うことでタスク割り当てを決定する．実験では，NoC，タスクグラフ，故障コア数の上限値を変えながら提案手法を適用することで，タスク割り当て時間や DTTR に

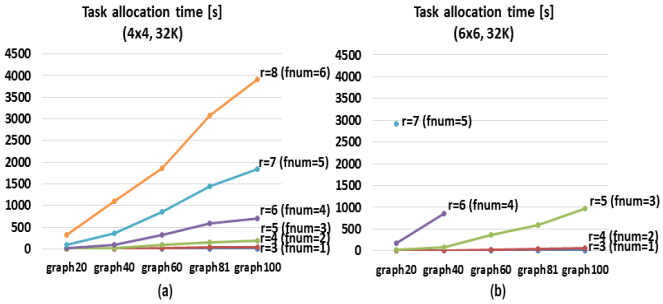

図 6　タスク割り当て時間 (a)4x4 (各コアのメモリは 32K)，(b)6x6 (各コアのメモリは 16K)

おけるオーバーヘッドを評価した．現実的な時間制限の中で提案手法は，4x4 の NoC であれば，故障コア数が 6 つ，100 のタスクを持ったタスクグラフにも対応することができる．今後は，DTTR におけるタスクの属性の決定法，信頼度を考慮したタスク割り当て手法を検討する．

謝辞　本研究は JSPS 科研費 15H02254 の助成を受けたものである．

参考文献

[1] 今井，米田，マルチコアシステムにおける信頼度向上手法のマルコフモデルによる性能評価，電子情報通信学会技術研究報告 DC2015-19, pp.19–24, 2015.

[2] G.De Micheli and L.Benini, *Networks on Chips*, Morgan Kaufmann, 2006.

[3] C.Ababei and R.Katti, Achieving network on chip fault tolerance by adaptive remapping, *Proc. IPDPS*, pp. 1–4, 2009.

[4] O.Derin, et al., Online task remapping strategies for fault-tolerant network-on-chip multiprocessors, *Proc. NoCS*, pp. 129–136, 2011.

[5] C.Lee, et al., A task remapping technique for reliable multi-core embedded systems, *Proc. CODES+ISSS*, pp. 307–316, 2010.

[6] H.Saito, et al., An ILP-based Multiple Task Allocation Method for Fault Tolerance in Networks-on-Chip, *Proc. MCSoC*, pp.100–106, 2012.

[7] H.Saito, et al., A Redundant Task Allocation Method for Reliable Network-on-Chips, *Proc. SASIMI*, 2015.

[8] W J. Dally and B.Towles, *Principles and practices of interconnection networks*, Morgan Kaufmann, 2004.

[9] G.De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[10] K.Vallerio, *Task Graphs for Free*, 2008.