Vol.2018-EMB-48 No.4
2018/6/29

車載制御システムにおける時間的決定性保証手法の検討

小川 真彩高[1]　本田 晋也[1]　高田 広章[1]

概要：近年，車載制御システムは複雑化，高度化してきており，車載制御システムのマルチコア化が進められてきている．車載制御システムでは，ランナブル間のデータ通信のタイミングの決定性（時間的決定性）が求められる．しかし，マルチコアでは，異なるコアに配置されたランナブル間のデータ依存関係により制御構造が複雑となるため，時間的決定性が満たされるように設計することが困難である．時間的決定性を保証する通信手法として，LET（Logical Execution Time）が提案されている．本研究では，実際のパワトレアプリの構成を参考にして生成したパワトレアプリモデルに対し LET 通信を適用する手法について検討した．具体的にはパワトレアプリモデルに LET 通信を適用する際の課題を挙げ，それらの課題を考慮した LET の実現方法を示した．

キーワード：車載制御システム，時間的決定性，LET (Logical Execution Time)

The Study of Temporal Determinism Assurance Semantics for Automotive Control Systems

Masataka Ogawa[1]　Shinya Honda[1]　Hiroaki Takada[1]

Abstract: Currently, automotive control systems have been more complex and more advanced and thereby applying a multi-core architecture to automotive control systems have been proceeding. Additionally, automotive control systems require temporal determinism which means that execution timing of data communication between runnables is static. Although, it is difficult to design so as to ensure temporal determinism because implementation on a multi-core architecture makes data communications between tasks more complex. LET（Logical Execution Time）communication was proposed in order to resolve this problem. In this paper, we study an approach of applying LET communication to a powertrain application model, which mimics a structure of an actual powertrain application. Concretely, we propose a LET realization approach to deals with four problems caused when applying LET communication to the model.

Keywords: automotive control systems, temporal determinism, LET (Logical Execution Time)

1. はじめに

近年，排ガス規制や低燃費化などへの対応から，車載ソフトウェア，特にパワートレインを制御するパワートレインアプリケーション（パワトレアプリ）が複雑化・高度化している．しかし，今まで主流であったシングルコアでは耐熱，耐ノイズ性能から動作周波数が限界に達していることや，車内に搭載する ECU 数を削減するため，パワトレアプリにおいてもマルチコアアーキテクチャが採用されるようになってきた．

複雑化する車載ソフトウェアの設計効率を向上させるため，車載ソフトウェアの標準である AUTOSAR 仕様 [1] に準拠することが必須になってきている．AUTOSAR では実行周期が指定された各処理をランナブルとして定義し，同じ周期で実行されるランナブルを 1 つのタスクとし，タスク単位でスケジューリングされる．ここで，ランナブル間は共有データを介して通信を行うが，安定した制御を行うためにはデータ依存関係を持つランナブル間での実行順が決定的であることが望ましい．データ依存関係を持つランナブル間の関係は，タスク内，同一コアのタスク間，コア間にわけることができる．タスク内については，タスク中

[1] 名古屋大学大学院情報学研究科
Graduate School of Informatics, Nagoya University, Nagoya-shi, 464-8603 Japan

© 2018 Information Processing Society of Japan

の配置位置によってランナブルの実行順が決定的となる．同一コアのタスク間では，同じイベントで起動されるタスクの実行順は優先度によって決定される．しかし，コア間については，異なるコアに配置されたタスク間の実行順を決定的にすることは難しく，コア間で共有するデータへのアクセス時にはコア間排他やバス競合によって実行時間が変化してしまうため，ランナブルの実行順序を安定させることは難しい．AUTOSAR 仕様で定められたデータ通信方法として直接通信（Explicit 通信）と Implicit 通信があるが，これらの通信方法はこの問題を考慮していない．実行順を決定的にする方法として，イベントフラグや排他制御を用いた同期を使用することが考えられる．しかしこの方法ではクリティカルな機能を持つ高優先度のタスクの応答時間が悪化し，デッドラインミスする危険性が出てくる．

同期を使用せずにこの問題を解決する方法として，LET 通信 [2] が車載分野においても注目されるようになってきている．LET 通信は決定的な通信タイミングでデータをやり取りする方法である．LET 通信の課題としてはメモリ使用量の増加やセンサがデータを受け取ってからアクチュエータに渡されるまでの一連のタスクチェーンの時間（end-to-end latency）が直接通信や Implicit 通信より長くなること，追加のデータアクセスオーバヘッドがかかることが挙げられる．メモリ使用量を削減する手法については文献 [3] で提案されているが，この手法について実際の規模の車載ソフトウェアに適用した場合の評価は存在していない．また，実際にパワトレアプリなどの車載ソフトウェアに実装するためにはサブスケジューリングへの対応，非同期通信の考慮，タスクのデッドラインミスの考慮，LET 処理の軽量化という 4 つの課題が存在する．本論文のコントリビューションは，これら 4 つの課題を考慮した LET 通信の実装方法を検討することと実際の規模の車載ソフトウェアに文献 [3] の手法を適用した場合の評価を与えることである．

本論文の構成を以下に示す．2 章では本論文が対象としている車載制御モデルについて説明し，3 章では既存の LET 通信実装方法および LET 通信の課題について紹介する．4 章では 3 章で述べた課題を考慮した LET 通信の実現方法について述べる．5 章では実際の規模の車載ソフトウェアに対し LET 通信を実装した場合の評価を行い，直接通信，Imlicit 通信との比較を行う．6 章では，本論文のまとめと今後の課題について述べる．

2. 車載制御モデル

本章では，車載制御ソフトウェアに必要とされる性質の 1 つである時間的決定性について説明した後，AUTOSAR で提案されている共有データ通信手法である直接通信および Implicit 通信と，時間的決定性を保証することのできる共有データ通信手法である LET（Logical Execution Time）通信について述べる．また，本論文で対象とするパワートレインアプリから抽出したパワトレアプリモデルや本論文で想定している実行環境のハードウェアアーキテクチャモデルについて説明する．

2.1 時間的決定性

タスクの起動タイミングはタスクの周期によって決定されるが，ランナブルが実際に共有データにアクセスするタイミング（実行タイミング）は高優先度のタスクおよび ISR の起動やタスク内の実行パス，バス競合やコア間排他制御に影響される．ここで，ランナブル間の共有データ通信のタイミングが決定的である性質を時間的決定性と呼ぶ．時間的決定性があれば，ある実行タイミングで読み込まれる共有データを更新したランナブルの起動タイミングが決定的となり，予測可能性が向上する．しかし時間的決定性を持たない場合は逆に，ある実行タイミングで読み込まれる共有データを更新したランナブルの起動タイミングが不明なため，複数のデータ更新タイミングを想定して制御設計をする必要が出てくる．これらのことから，時間的決定性を保証できるような通信方法が求められている．

2.2 通信モデル

共有データを介したタスク間の通信方法として，直接通信，Implicit 通信，LET 通信について説明する．

2.2.1 直接通信（Explicit 通信）および Implicit 通信

直接通信（Explicit 通信）は，ランナブルが共有データを直接アクセスする方法である．直接通信は Implicit 通信，LET 通信と比べて簡素な実装であり，追加のオーバヘッドがないという利点がある．

Implicit 通信は，データの一貫性を保持するために AUTOSAR によって提案された通信方式である．データの一貫性とは，あるタスクが使用しているデータの値が，外部の要因によって変更されないようにするという性質である．Implicit 通信の例を図 1 に示す．Implicit 通信ではまず，タスクの先頭で共有データを読み込み，そのランナブルに対応したローカル領域にコピーする．この一連の処理をコピーインと呼ぶ．次にタスク実行中は共有データの代わりにローカル領域のデータにアクセスする．最後に，タスクの末尾でローカル領域のデータを共有データにコピーする．この一連の処理をコピーアウトと呼ぶ．Implicit 通信ではローカルメモリの使用量が増加することや，コピーイン，コピーアウトによるデータアクセスオーバヘッドがかかってしまうことに注意する必要がある．

直接通信では，複数のコアで共有されるデータにアクセスする際にはスピンロックによるコア間排他制御を行う必要がある．Implicit 通信では，ローカル領域へのアクセス時にはコア間排他制御は不要であるが，コピーイン，コピーアウトでは直接通信と同様にコア間排他制御は必要である．直接通信および Implicit 通信は，共有データ書き込みの実行タイミング次第でそのデータを受け取るランナブルの起動タイミングが変化する可能性があるため，時間的決定性を保証することはできない．

© 2018 Information Processing Society of Japan

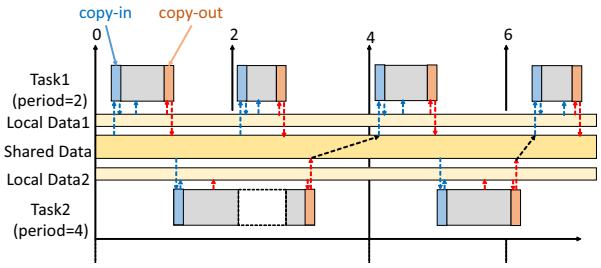


図 1　Implicit 通信
Fig. 1　Implicit communication

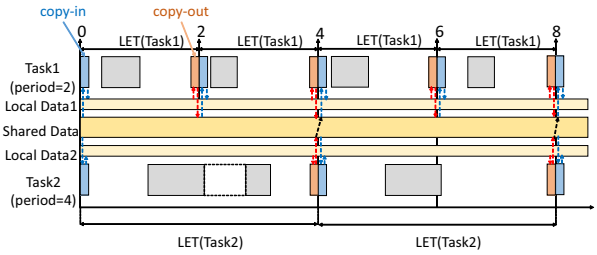


図 2　LET 通信
Fig. 2　LET communication

2.2.2　LET 通信

時間的決定性を保証する通信方法として LET（Logical Execution Time）通信 [2] がある．LET 通信ではあるタスクについて，一定の決定的な時間間隔（一般的にはタスクの周期）を LET と定め，LET の開始時にコピーインを実行し，タスクではローカル領域にアクセスし，LET の終了時にコピーアウトを実行する．

LET 通信の例を図 2 に示す．LET 通信ではすべての共有データアクセスの実行タイミングが決定的となるため，時間的決定性が保証される．また，通常，複数のコアからアクセスされる可能性のある共有データに関しては排他アクセスをする必要があるが，LET 通信では共有データにアクセスするタイミングが明確になっているため，このような排他を必要としない設計が可能である [4]．

ここで，LET 通信では，end-to-end latency の長さが Implicit 通信より長くなり，最新のデータを利用することができなくなる場合がある点と Implicit 通信と同様に追加のメモリ使用量が必要である点に留意する必要がある．

2.3　パワトレアプリモデル

2.3.1　ソフトウェアモデル

本論文で対象とするパワトレアプリモデルの構成を図 3 に示す．パワトレアプリモデルは複数のタスクで構成される．タスクはおもに時間同期タスクと回転角同期タスクに分けられる．時間同期タスクは一定の時間間隔で発生するイベント（時間イベント）によって起動されるタスクで，回転角同期タスクはクランクシャフトのある特定の角度で発生するイベント（回転角イベント）によって起動されるタスクである．タスクはイベントの発生期間に対する周期を持ち，同じイベントで起動されるタスクは同期して実行される．

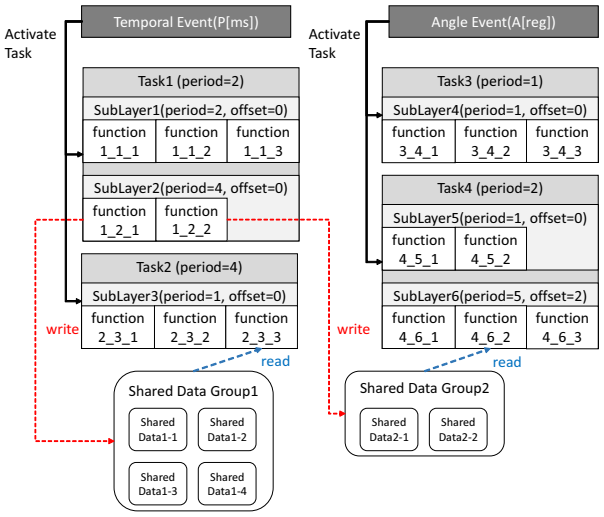


図 3　パワトレアプリモデル
Fig. 3　Powertrain application model

タスク中では複数のランナブルが実行される．ランナブルは不可分の処理を実行する関数であり，共有データを介して他のランナブルと通信を行う．ランナブルが実行される順番はタスク中の配置位置に依存する．また，ランナブルでは実際に測定して得た平均実行時間の間ループする処理を行う．

共有データは複数のランナブルによってアクセスされるデータである．共有データに書き込みを行うランナブルはただ 1 つである．読み込みおよび書き込みを行うランナブルが一致する共有データは共有データグループ（SDG）という単位でグルーピングされ，ランナブルは SDG 単位でアクセスする．

現在，パワトレアプリモデルに含まれるランナブルの数は 1000 個程度であり，合計 20 個程度の時間同期タスクと回転角同期タスクがあり，ランナブルはそれらのうちいずれかに割当てられる．共有データの数は約 10000 個であり，300 個程度の SDG に分割されている．

2.3.2　スケジューリングモデル

パワトレアプリモデルでは，タスクは RMS(Rate Monotonic Scheduling) によって周期が短いタスクほど高い優先度に割当てられる．設計時に，CPU 負荷が高く重要な処理の実行がデッドラインを満たすことができない場合，いくつかの比較的重要でないランナブルの周期を長くすることで CPU 負荷を下げるという工夫がなされる．

実行順序を変更せずに CPU 負荷を下げることができるように，サブスケジューリングという方法を用いる．サブスケジューリングでは，各ランナブルはタスクの起動間隔に対して周期とオフセットを持つ．各タスクに対し，同じ周期とオフセットを持つランナブルの集合をサブレイヤと呼ぶ．図 4 ではサブスケジューリングされたランナブルの実行の様子を示している．Task1 にサブスケジューリングを行なったタスクを Task1' とすると，SubLayer2 内のランナブルは 0ms, 4ms, 8ms,... のタイミングで実行される．サブスケジューリングされたランナブルのデッドラインは

© 2018 Information Processing Society of Japan

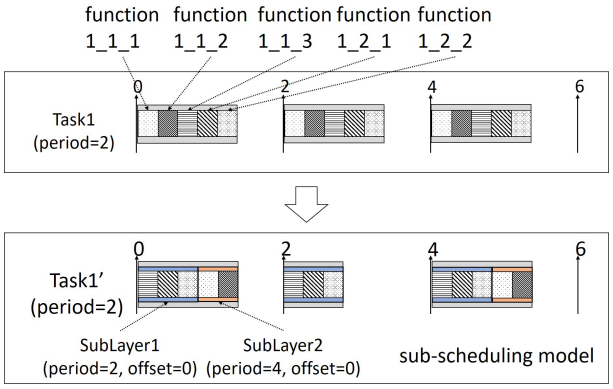


図 4　サブスケジューリング
Fig. 4　Sub-scheduling

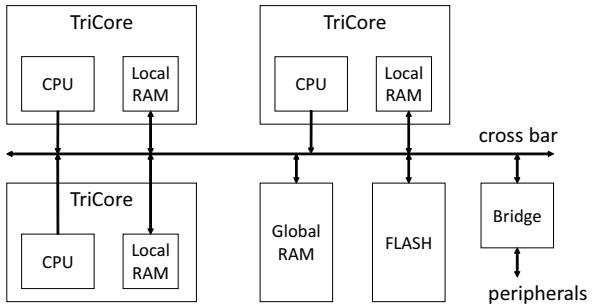


図 5　マルチコアハードウェア構成
Fig. 5　Multi-core hardware architecture

そのランナブルが実行される周期（サブスケジューリングの周期）ではなく，タスクの周期となる．また，オフセットを用いることで，ランナブルの実行タイミングを分散させ，CPU 負荷の低下に貢献することが可能である．

2.4　ハードウェアモデル

本研究で想定する実行環境のハードウェアアーキテクチャは車載システム向けに開発された AURIX [5] アーキテクチャに準じている．AURIX アーキテクチャの構成を図 5 に示す．AURIX アーキテクチャは 1 サイクルでアクセスできるローカル RAM を持つ CPU が 3 つ搭載されており，各コアはクロスバーを介してグローバルメモリおよび他コアのローカル RAM にアクセスすることができる．クロスバーを介してのメモリアクセスはローカルメモリへのアクセスと比べて 9 倍である．また，AURIX アーキテクチャの 1 つである TC29xTX では，CPU0 は 120KB のローカルデータ RAM，CPU1 および CPU2 は 240KB のローカルデータ RAM を持っており，グローバルメモリは 32KB である．

3.　既存の LET 通信の実現手法および課題

LET 通信を車載制御ソフトウェアに適用する試みはすでに行われてきている．WATERS industrial challenge 2017 [6] では，ボッシュ社が提供するパワトレアプリのモデルをもとに LET の実装方法や評価手法についての議論が行われた [7][8][9][10][11]．また，文献 [12] ではダブルバッファを用いた LET 通信の実装方法が示されており，文献 [3] では LET 通信によって追加で必要となるローカル領域の削減手法について述べられている．

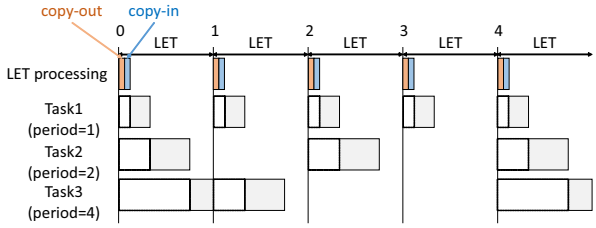


図 6　LET 処理
Fig. 6　LET processing

本章ではこれらの既存の LET 実装方法を説明した上で，実際にパワトレアプリモデルに実装する上で考慮しなければならない課題について議論する．

3.1　LET 処理

LET 通信におけるコピーイン，コピーアウトの処理を LET 処理と呼ぶ．LET 処理の例を図 6 に示す．LET 処理では，前回の LET におけるコピーアウトを実行してから，コピーインを実行する．また，LET 処理は元々のタスク（レギュラータスク）とは異なる高優先度な通信専用のタスクもしくは ISR によって実行される [7]．このようにするのは，LET 処理を必ず LET の最初で実行できるようにするためである．レギュラータスクの実行は LET 処理の後である必要があるため，LET 処理を実行するコア以外に配置されたタスクも LET 処理が終了するまで待つ必要がある．LET 処理を実現するタスクおよび ISR はイベントごとに用意される．LET 処理の周期は対応するイベントの周期とする．

3.2　ダブルバッファの利用

LET 通信を実現する方法として，ダブルバッファを用いた手法が提案されている [12]．ダブルバッファを用いた通信モデルについて図 7 に示す．本手法では，共有データの書き込み側のローカル領域の代わりに共有データをダブルバッファにする (つまり，共有データを 2 つ用意する)．この共有データのうち片方が読み込み専用のバッファ（リードバッファ），もう片方が書き込み専用のバッファ（ライトバッファ）である．また，共有データがどちらのバッファであるかを決めるため，バッファポインタを用意する．リードバッファに対するバッファポインタをリードバッファポインタ，ライトバッファに対するバッファポインタをライトバッファポインタと呼ぶ．

コピーインではリードバッファポインタを使用してリードバッファを読み込み，ローカル領域にコピーする．レギュラータスク中では読み込みはローカル領域から行い，書き込みはライトバッファポインタが示すライトバッファに対して行う．コピーアウトではリードバッファポインタとライトバッファポインタの入れ替えを行う．これによ

© 2018 Information Processing Society of Japan

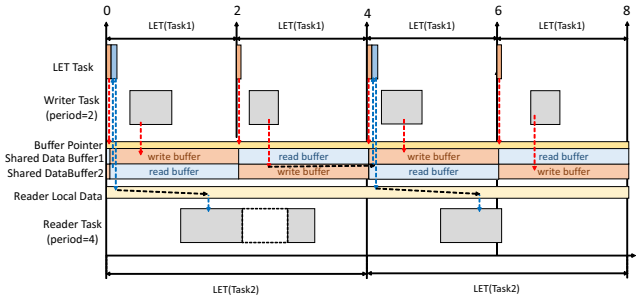


図 7　ダブルバッファによる LET 通信
Fig. 7　LET communication with double buffering

り，読み込みと書き込みのアクセス競合が起きないようにすることが可能である．

3.3　LET 通信実装の課題

ボッシュ社のモデルについて，上記の LET 通信の実現方法が提案されているが，実際に 2.3 節で述べたモデルに LET 通信を適用する場合，以下の 4 つの課題について考慮する必要がある．

(a) サブスケジューリングへの対応
(b) 非同期通信の考慮
(c) タスクのデッドラインミスの考慮
(d) LET 処理の軽量化

まず（a）について，ボッシュ社のモデルは，すべてのランナブルがタスクの周期で実行されているという想定であり，サブスケジューリングは行っていない．そのため，サブスケジューリングされたランナブルについて，LET 通信の実現方法を検討する必要がある．

次に（b）について，ここまで述べた LET 実装手法は時間同期タスク間の通信について考慮しているが，実際のパワトレアプリには回転角同期タスクが存在する．回転角同期タスクはエンジンの回転数によって周期が変動するものの，回転角同期タスク間については同期して起動することが可能であるため，既存の手法を用いて LET 通信を実装することは可能である．しかし，時間同期タスクと回転角同期タスクの通信に関しては，異なるイベントに起因して起動されるために非同期となる．ここで，非同期間の通信においても LET 通信の恩恵を受けることが可能であるかを議論する必要がある．

次に（c）について，実際のパワトレアプリでは重要度の低いランナブルは低周期・低優先度のタスクに割当てられる．そのようなタスクの中には高負荷時には安全性に影響しないソフトリアルタイムなタスクもあり，それらのデッドラインミスはある程度許容される．デッドラインミスが発生した場合，LET によって静的に決定されていた共有データ通信タイミングが変更されてしまう．そのため，パワトレアプリモデルに LET 通信を適用する場合でも，可能な限り LET による共有データ通信タイミングに影響がでないような実装方法を検討する必要がある．

最後に（d）について，LET 処理では同じイベントによって起動される複数のタスクのコピーイン，コピーアウトを行うため，実行時間が大きくなると予想される．また，LET 処理が完了するまでレギュラータスクは実行することができないため，レギュラータスクが実行可能な区間が短くなってしまう．このような問題に対処するため，軽量な LET 処理の実現方法について提案する必要がある．

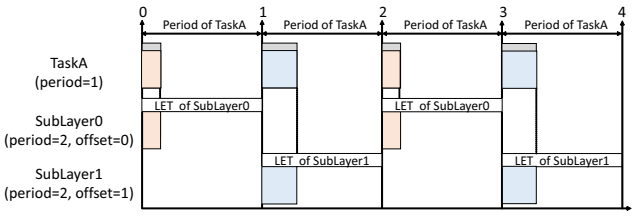


図 8　サブレイヤの LET
Fig. 8　LET of sublayers

4.　パワトレアプリモデルへの適用検討

本章では 3.3 節で述べた LET 通信を実装する際の課題に対応できるような LET 通信の実現方法について検討する．

4.1　（a）サブスケジューリングへの対応

サブスケジューリングされているランナブルが実行されるタイミングはタスクの周期ではなく，サブレイヤの周期およびオフセットに依存するため，サブレイヤごとに LET の区間を設定する必要がある．

LET の区間の設定方法として，サブレイヤの周期とする方法が考えられる．しかしこの場合，LET の区間が非常に長くなり，他のタスクに更新結果を反映させるまでの時間が長くなる可能性がある．ここで，サブレイヤのデッドラインはタスクの周期によって決定されるため，タスクの周期内にデータの更新が行われるはずである．そのため，LET の区間をサブレイヤの起動タイミングからタスクの周期経過した時刻までの区間とする．この場合，LET の区間はタスクの周期分の長さとなり，速やかに他のタスクに更新結果を反映させることができるため，本論文ではこちらの方法を採用する．

サブレイヤの LET について図 8 に示す．図 8 では SubLayer1 は周期が 2，オフセットが 1 で，SubLayer1 を含むタスクの周期が 1 であるため，LET の区間は $[1,2],[3,4],\ldots$ となる．

4.2　（b）非同期通信の考慮

LET 通信であっても，異なるイベントで起動されたタスク間では非同期通信となるため，時間的決定性を保証することは不可能である．また，スピンロックなどによるコア間排他制御も必要となってくる．その上，非同期間では元々データの送受信のタイミングが不定であるという前提で設計されるため，時間的決定性の必要性も高くない．このように LET 通信のメリットがほとんどないため，非同期通信においては，最新のデータが利用でき，データの一貫性を保証することのできる Implicit 通信を用いる．

© 2018 Information Processing Society of Japan

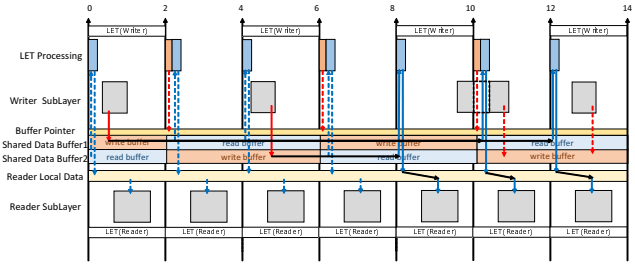


図 9　サブレイヤのデッドラインミスの例
Fig. 9　Example of deadline miss of a sublayer

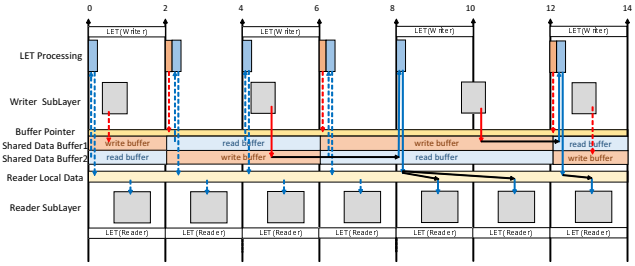


図 10　デッドラインミス対応
Fig. 10　Dealing with a problem of deadline miss

4.3　（c）タスクのデッドラインミスの考慮

ダブルバッファを用いた実装でデッドラインミスした場合の共有データ通信について，図 9 に示す．この場合，時刻 10，12 で起動した Reader SubLayer は本来時刻 8 で起動した Writer SubLayer が生成したデータを使用するべきであるが，時刻 0 で起動した Writer SubLayer が生成したデータを使用することになる．時刻 8 で起動した Reader SubLayer は時刻 4 で起動した Writer SubLayer が生成したデータを使用しており，時刻 10，12 で起動した Reader SubLayer は前々実行時より古いデータを使用することになってしまう．

デッドラインミスによるこのような問題に対処するために，以下のような機能を搭載する．まず，各 SDG について，データ更新フラグを用意する．データ更新フラグは LET 処理の起動時に False となり，ランナブルがデータの更新を行った際に True となる．ここで，LET 処理の起動時にすでにデータ更新フラグが False のとき，共有データを更新するランナブルがデッドラインミスしたと判断し，コピーアウト（ダブルバッファを用いた実装ではバッファポインタの切り替え）を行わない．コピーアウトが行われなかった場合は，次のタスク実行時にデッドラインミスしたランナブルが終了していれば LET 処理によってコピーアウトを行う．

この機能を用いた場合の例を図 10 に示す．これによって，Reader SubLayer は少なくともデッドラインしたタスクの前回の実行時に更新したデータを使用することができるようになる．また，時刻 12 で起動した Reader SubLayer については時刻 8 で起動した Writer SubLayer が生成したデータを使用できるため，想定どおりに共有データを受け取ることができる．

4.4　（d）LET 処理の軽量化

本節では，LET 処理による軽量化の方法について提案する．

4.4.1　SDG によるオーバヘッドの削減

共有データの数は 10000 個にも及ぶため，それらすべてについてバッファポインタを用意し，コピーアウト時にバッファポインタの交換を行うのは非効率的である．ここで，2.3.1 節で説明したとおり，本研究では共有データを SDG によってグルーピングしている．ダブルバッファを用いた場合，各共有データをリードバッファポインタ，ライトバッファポインタを用いてアクセスするが，SDG を共有データの配列とみなし，各共有データへのアクセスを SDG の先頭アドレス+インデックスとすることで，SDG ごとにバッファポインタを用意すれば十分となる．これにより，バッファポインタの必要数が少なくなり，バッファポインタに必要なメモリ使用量と LET 処理のコピーアウト時に行うバッファポインタの交換に必要な実行オーバヘッドを削減することが可能となる．

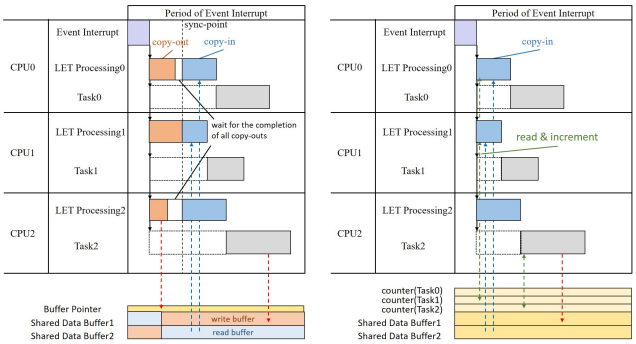
4.4.2　LET 処理の分散

ランナブルの個数は 1000 個程度あり，すべてのランナブルのコピーイン，コピーアウトを 1 つの LET 処理で実施すると非常に大きなオーバヘッドがかかる．そのため，LET 処理をコアごとに用意することで負荷を分散させる．複数のコアで LET 処理を起動する場合，ある共有データに関してコピーアウトとコピーインの順序が入れ替わらないように注意する必要がある．そのため，同じイベントに関する LET 処理はコピーアウトが終了した後，同期を行って他コアの LET 処理のコピーアウトが終了するまで待ち，その後各 LET 処理はコピーインを行うようにする．図 11(a) に同期による LET 処理の分散の例を示す．

ここで，より効率的にダブルバッファの切り替えを行うことで，LET 処理のさらなる効率化を実現する方法を図 11(b) に示す．LET によってリードバッファとライトバッファが切り替わるタイミングは静的に決まっている．よって，共有データへアクセスするランナブルや LET 処理は，その静的なタイミングに従って 2 つの共有データのうちどちらのデータを使用するのかを判断すれば良い．この静的なタイミングは，タスクごとに持つカウンタ変数によって管理する．書き込み側のタスクが実行されるたびにカウンタ変数をインクリメントし，カウンタ変数を参照してライトバッファを決定する．また，読み込み側のタスクについては対応する LET 処理の実行のたびにカウンタ変数をインクリメントし，カウンタ変数を確認してリードバッファを決定する．

この手法を用いることで，LET 処理によるバッファポインタの切り替えを行う必要がなくなり，それによりバッファポインタ自体も不要となる．また LET 処理ではコピーアウトが不要となるため，LET 間の同期も不要となる．これにより，LET 処理をさらに軽量化することが可能となることが予測できる．

© 2018 Information Processing Society of Japan



(a) 同期による LET 処理の分散　(b)LET 処理の同期の除去

図 11　LET 処理の分散

Fig. 11　Distribution of LET processings

5. 評価

本章では，パワトレアプリのモデルに LET 通信を適用する場合のメモリ使用量と共有データのアクセスに要するオーバヘッド（合計データアクセスオーバヘッド）について静的見積もりをし，直接通信および Implicit 通信と比較評価を行う．ここで LET 通信を適用する場合であっても 4.2 で説明したとおり非同期な通信に関しては Implicit 通信を行う．また，LET 通信以外では複数のコアからアクセスされる共有データについてはスピンロックによってコア間排他を行うため，スピンロックの獲得・解放に要するオーバヘッドの合計（合計スピンロックオーバヘッド）についても評価する．ただし，本来スピンロックのオーバヘッドにはスピンロックの解放待ち時間も含まれるが，スピンロックの解放待ち時間は各コアのタスクの実行状況に応じて大きく変化するため，静的見積もりで得ることは難しい．したがって本研究では合計スピンロックオーバヘッドにスピンロックの解放待ち時間は含まないこととする．

ここで，LET 通信はダブルバッファによる実現方法を想定している．また，LET 通信については，ローカル領域削減手法を用いた場合についても評価を行う．

5.1 評価環境

本研究の評価を行う実機環境として想定しているのは 2.4 節で説明した CPU を 3 つ搭載した AURIX アーキテクチャである．また，ソフトウェアモデルは 2.3 節で説明したソフトウェア構成とスケジューリングモデルを持つパワトレアプリを想定している．回転角イベントについては本論文では 4,000rpm でエンジンが動作しているという想定で発生間隔を決めている．

時間同期タスクおよび回転角同期タスクはいずれかのコアに配置されている．また，LET 通信では各コアごとに時間イベント，回転角イベントに対応する LET 処理を行う LET タスクを用意する．LET タスクでは関連するレギュラータスクについてコピーイン，コピーアウトが行われる．SDG はアクセスするランナブルのうち，最も周期が短いランナブルが割当てられているタスクが配置されているコアのローカルメモリに配置される．

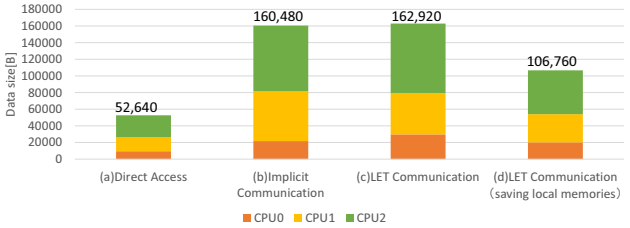


図 12　各通信方法におけるローカルメモリに置かれるデータの合計サイズ

Fig. 12　Total data size stored local memories with each communication mechanism

また，評価に用いられるスピンロック獲得・解放 API オーバヘッドと各コアからのメモリアクセス時間は実機の KIT_AURIX_TC299_TRB 上で計測した．

5.2 メモリ使用量の評価

まず，各通信方法におけるメモリ使用量の計算方法について説明する．直接通信では，追加のデータが必要ないため，共有データのサイズの合計となる．Implicit 通信では，書き込み側のランナブルと読み込み側のランナブルそれぞれについて追加のローカル領域が必要である．したがって，共有データにアクセスするランナブルが割当てられているタスクが配置されているコアのローカルメモリに共有データ元々のデータサイズ分だけ追加する．

LET 通信では，読み込み側のローカル領域については Implicit と同様である．書き込み側については，ダブルバッファを用いているため，共有データのサイズを 2 倍にし，バッファポインタに必要なメモリサイズ分さらに追加する．

結果を図 12 に示す．（a）は直接通信，（b）は Implicit 通信，（c）は LET 通信，（d）はローカル領域削減手法を用いた場合の LET 通信のメモリ使用量を示している．結果，Implicit 通信および LET 通信は直接通信と比べて 3 倍程度メモリを使用し，LET 通信はバッファポインタに必要なデータサイズ分 Implicit 通信よりも必要なデータサイズが大きかった．しかし，ローカル領域削減手法を用いることで，LET 通信に必要なメモリ使用量を約 2/3 に削減することが可能であった．これにより，LET 通信の場合であってもローカル領域削減手法によって必要なメモリ使用量を大幅に削減することが可能であることがわかった．

5.3 データアクセスおよびスピンロックオーバヘッド評価

次に，データアクセスおよびスピンロックオーバヘッドの評価について説明する．本論文ではそれぞれの通信方法について，データアクセスオーバヘッドの静的見積もりを行う．ここで，ランナブルが 1 回の実行で同じ共有データにアクセスする回数は 1 回のみとする．これは，ランナブル内では共有データからレジスタに読み込んだ後はそのレジスタを用いて計算すればよく，複数回共有データにアクセスする必要がないためである．共有データへの書き込みについても同様に，ランナブル内で 1 回だけでよい．直接

© 2018 Information Processing Society of Japan

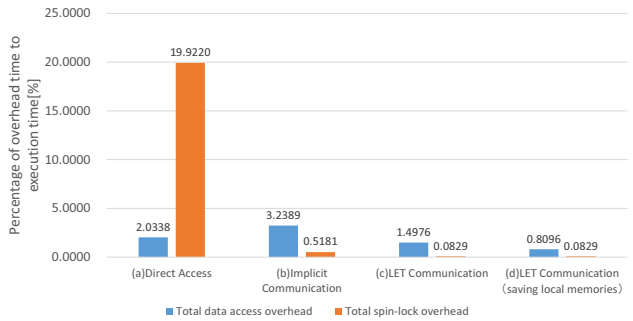


図 13 各通信方法におけるコアあたりの合計データアクセスオーバヘッドおよび合計スピンロックオーバヘッド

Fig. 13 Total data access overhead and total spin-lock overhead with each communication mechanism

通信ではランナブル内での共有データへのアクセスを対象に見積もりを行い，Implicit 通信，LET 通信ではさらにコピーインおよびコピーアウトを対象に加える．

結果を図 13 に示す．図 13 には各通信方法における 1 コアごとの実行時間に対する合計データアクセスオーバヘッドおよびスピンロックオーバヘッドの割合を示している．直接通信では，合計データアクセスオーバヘッドは大きくないものの，共有データごとにスピンロックをとっている想定であるため，合計スピンロックオーバヘッドが非常に大きくなっている．Implicit 通信では，SDG ごとにスピンロックを取るため，合計スピンロックオーバヘッドは削減できている．しかし，合計データアクセスオーバヘッドはコピーイン，コピーアウトによって追加のオーバヘッドがかかっている．LET 通信では 4.4.1 節の手法により，Implicit 通信より合計データアクセスオーバヘッドを削減できている．また，同期しているタスク間ではコア間排他が不要であるため，合計スピンロックオーバヘッドが大きく削減できている．また，ローカル領域削減手法を適用することでコピーイン，コピーアウトでアクセスされる共有データが削減されるため，合計データアクセスオーバヘッドはさらに減少する．

6. おわりに

本論文では，既存の LET 通信の実現方法について説明した上で実際にパワトレアプリモデルに適用する場合の課題として，サブスケジューリングへの対応，非同期通信の考慮，タスクのデッドラインミスの考慮，LET 処理の軽量化を挙げた．評価として，実際の規模のパワトレアプリに LET 通信を実装した場合のメモリ使用量やデータアクセスオーバヘッドについて見積もりを行った．LET 通信については文献 [3] のローカル領域削減手法を適用した場合についても評価を行った．結果として LET 通信は直接通信と比較してメモリ使用量は多くなるが，文献 [3] の手法を用いることで Implicit 通信よりも大幅にメモリ使用量を削減することが可能であることがわかった．また，LET 通信では同期しているタスク間のコア間排他が不要なことからスピンロックオーバーヘッドを大幅に削減できることと，文献 [3] の手法によってデータアクセスオーバヘッドが削減できることがわかった．

今後の課題としては，実機ボード上で LET 通信を実装し，end-to-end latency の評価を行うことが挙げられる．また，4.4.2 節で述べた手法の評価を行うために，LET 処理を分散させたときの LET 処理の実行オーバヘッドを実機によって計測することも必要である．

謝辞 本研究を進めるにあたりご協力いただいたトヨタ自動車株式会社および株式会社サニー技研に深く御礼申し上げます．

参考文献

[1] AUTOSAR SWS Operating System, 2017.
[2] Thomas A Henzinger, Benjamin Horowitz, and Christoph M Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, Vol. 91, No. 1, pp. 84–99, 2003.
[3] Stefan Resmerita, Andreas Naderlinger, and Stefan Lukesch. Efficient realization of logical execution times in legacy embedded software. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pp. 36–45. ACM, 2017.
[4] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication centric design in complex automotive embedded systems. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
[5] Infineon technologies. AURIX ™Family.
[6] The industrial challenge of WATERS 2017. `http://waters2017.inria.fr/challenge/`(Accessed 16/05/2018).
[7] Alessandro Biondi, Paolo Pazzaglia, Alessio Balsini, and Marco Di Natale. Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores. *The industrial challenge of WATERS 2017*, 2017.
[8] Frédéric Boniol, Julien Forget, and Claire Pagetti. WATERS Industrial Challenge 2017 with Prelude. *The industrial challenge of WATERS 2017*, 2017.
[9] Juan M Rivas, J Javier Gutiérrez, Julio L Medina, and Michael González Harbour. Comparison of Memory Access Strategies in Multi-core Platforms Using MAST. *The industrial challenge of WATERS 2017*, 2017.
[10] Kai Björn Gemlau, Johannes Schlatow, Mischa Möstl, and Rolf Ernst. Compositional Analysis of the WATERS Industrial Challenge 2017. *The industrial challenge of WATERS 2017*, 2017.
[11] Jorge Martinez, Ignacio Sañudo, Paoloa Burgio, and Marko Bertogna. End-To-End Latency Characterization of Implicit and LET Communication Models. *The industrial challenge of WATERS 2017*, 2017.
[12] Matthias Beckert, Mischa Möstl, and Rolf Ernst. Zero-time communication for automotive multi-core systems under SPP scheduling. In *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*, pp. 1–9. IEEE, 2016.

© 2018 Information Processing Society of Japan