Vol.2019-ARC-235 No.4
Vol.2019-SLDM-187 No.4
Vol.2019-EMB-50 No.4
2019/3/17

車載制御システムを対象としたLogical Execution Timeの実現手法

小川 真彩高[1,a)]　本田 晋也[1]　高田 広章[1]

概要：近年，パワートレインアプリケーションのマルチコア化が進められてきている．しかし，コア間通信はタイミングが不定であることから検証が難しい．そのため，決定的なタイミングで通信を行うLogical Execution Time (LET) が注目されてきている．LETの実現手法として，単一の処理単位（LET処理）でタスク間通信を行う手法があるが，これには実行オーバヘッドが大きいという問題がある．また，安全性に影響しないタスクは，エンジンの高回転時にデッドラインミスする可能性があるが，既存のLETの実現手法ではそれを考慮していない．本研究では，デッドラインミスを考慮したLETの実装方法を提案し，その上でLET処理を複数コアに効率的に分散する手法を提案する．評価の結果，本手法は単一のLET処理の場合と比較してCPU負荷の削減が可能であることがわかった．

The Realization of Logical Execution Time for Automotive Control Systems

Masataka Ogawa[1,a)]　Shinya Honda[1]　Hiroaki Takada[1]

Abstract: In recent years, adapting a multicore architecture for powertrain applications has been proceeding. However, it is difficult to verify timings of inter-core communication of tasks because they are variable for each instance of a task. For this reason, Logical Execution Time (LET), which realizes communication at determined timings, is attracting a lot of attention now. An existing approach for LET realizes inter-task communication via a single dedicated process (LET process), however this approach causes a large execution overhead. In addition, existing approaches do not assume the possibility that safety-unrelated tasks miss their deadline when an engine rotates at a high speed. In this paper, we propose a realization approach for LET assuming the possibility, moreover an approach that efficiently distributes operations of an LET process into several cores. As a result of an evaluation, our approaches can reduce a CPU load of an LET process compared to a single LET process.

1. はじめに

近年，排ガス規制や低燃費化などの要求を満たすため，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．それらの要求に対処するために，CPUの動作周波数を増加させる必要があるが，すでに耐熱，耐ノイズ性能から動作周波数は限界に達している．この理由から，マルチコアアーキテクチャの適用が進められている [1]．現状は3コアの車載マイコン[*1]，ハイエンドなもので6コアの車載マイコン[*2]がリリースされている．

車載ソフトウェアの生産性を向上させるために，車載ソフトはAUTOSAR仕様 [2] に準拠するのが一般的になっている．AUTOSAR仕様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つのタスクとする．ランナブル間は共有データを用いて通信を行うが，通信の順序は設計時に決定される必要がある．しかし，コア間通信では，異なるコアに配置

[1] 名古屋大学大学院情報学研究科, 名古屋市
Graduate School of Informatics, Nagoya University, Nagoya-shi, 464-8603 Japan
[a)] masa-bach@ertl.jp

[*1] Infineon AG, 32-bit TriCore™ Aurix™ TC2xx
[*2] Infineon AG, 32-bit TriCore™ Aurix™ TC3xx

Vol.2019-ARC-235 No.4
Vol.2019-SLDM-187 No.4
Vol.2019-EMB-50 No.4
2019/3/17

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高優先度のタスクの実行状況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．しかし，AUTOSAR 仕様ではコア間でランナブルの順序を定める仕様がないため，実行順序を決定することはできない．

このような問題を解決するためには，ランナブルの振る舞いにかかわらずランナブル間の共有データ通信の実行タイミングを一定にする必要がある．通信のタイミングを一定にする方法として，Logical Execution Time (LET) [3] が注目されている．LET では，LET 区間によって定められた一定のタイミングで共有データにアクセスすることで通信の遅延の長さを一定にすることができる [4]．また LET では，ランナブルのコア配置が変更されたり，機能を追加してもデータ通信の振る舞いが変わらないため，マルチコアアーキテクチャへの移行が容易となる．

文献 [5] では LET で行う共有データ通信を専用のタスクもしくは ISR に担当させることで LET を実現する手法が提案されている．本論文ではこの専用のタスクもしくは ISR を LET 処理と呼ぶ．また，Beckert らは，ダブルバッファリングを用いて LET を効率的に実現する手法について提案した [6]．しかし，LET 処理では膨大な量のデータ通信を行うため実行時間が非常に長く，より効率的な LET 処理の実装が求められる．また，実際のパワトレアプリに LET を適用するためには，安全性に影響しないタスクがデッドラインミスすることを想定した LET の実装が必要であるという課題がある．一般的に車載システムはハードリアルタイムシステムであるため，デッドラインミスが許容されていない，しかし，パワトレアプリは回転数に応じて CPU 負荷が変動し，高回転時には一部の安全性に影響しないタスクはデッドラインミスする可能性がある．そのため，そのようなタスクについてはデッドラインミスが生じる可能性があることを前提として制御設計がなされている．しかし，既存研究ではデッドラインミスが起きた場合について考慮されていない．その結果，不整合なデータが他のタスクに公開されてしまう可能性がある．

本研究では，タスクがデッドラインミスしたときの影響を軽減できるような LET 通信の実装方法として，Deadline Miss Tolerant LET (DMT-LET) を提案する．次に CPU あたりの LET 処理の負荷を軽減するために，LET 処理を複数の CPU に分散する手法を提案する．同期を用いない LET の分散手法としてハードリアルタイムな処理を対象とした Asynchronized Distributed LET process (ADLP)，同期を用いた LET の分散手法としてソフトリアルタイムな処理を対象とした Synchronized Distributed LET process (SDLP) を提案する．また，SDLP と ADLP を組み合わせてハードリアルタイムおよびソフトリアルタイムな処理が混在するシステムを対象とした Hybrid Distributed LET Process (HDLP) を提案する．最後に，HDLP を用いたときのメモリ使用量と CPU 利用率についての評価を行う．

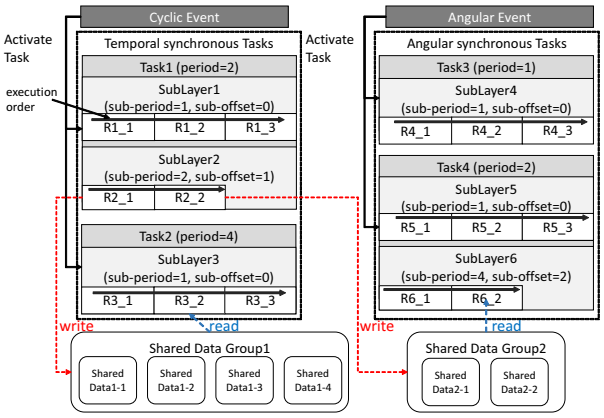

図 1　対象パワトレアプリの構成

本論文のコントリビューションは，デッドラインミスの可能性があるような実際のパワトレアプリに対して適用できる LET の実装方法を提案し，かつ CPU ごとの LET 処理の実行時間を軽減するための LET 処理の分散手法を提案することである．

2. 車載制御モデル

本章では，パワトレアプリの実装を検討するために，本研究で前提としているパワートレインアプリケーション（対象パワトレアプリ）について述べる．さらに，本研究で用いる通信方法として，Logical Execution Time について説明する．

2.1 対象パワトレアプリ

2.1.1 ソフトウェア構成

対象パワトレアプリの構成を図 1 に示す．対象パワトレアプリはパワトレアプリの実装を検討するために，実際のパワトレアプリのタスクとランナブルの関係を継承しているが，ランナブルの実際の計算や処理については含まれていない．タスクは周期イベントによって起動される時間同期タスクと，クランクシャフトの特定の角度で生じる回転角イベントによって起動される回転角同期タスクに分類される．ランナブル (図 1 の R1_1-6_2) はタスク内で事前に決められた順序で実行され，複数のランナブル間でアクセスされるグローバル変数（共有データ）を使用して他のランナブルとの通信を行う．ここで，読み込み，書き込みを行うランナブルが同じ共有データの集合を共有データグループ (SDG) と呼ぶ．現在のパワトレアプリに含まれるランナブルの数は 1000 個程度であり，合計約 20 個のタスク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．

2.1.2 サブスケジューリング

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，いくつかの重要でな

Vol.2019-ARC-235 No.4
Vol.2019-SLDM-187 No.4
Vol.2019-EMB-50 No.4
2019/3/17

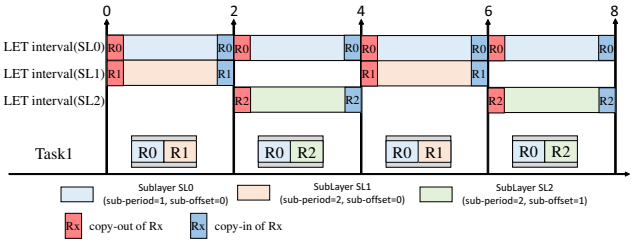

図 2　サブスケシューリングおよび LET 区間

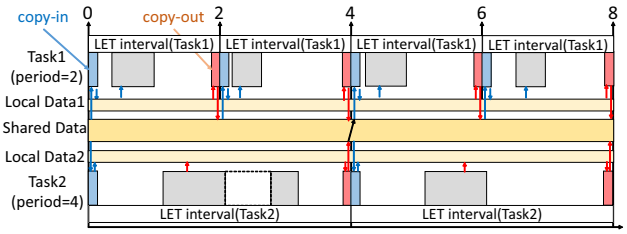

図 3　Logical Execution Time

いランナブルのタスク配置を変更する．タスクのコア配置が変更された場合，元々の設計と異なるタスクに配置されたランナブルの順序は想定されていた順序とは異なる可能性がある．また，周期の長さごとにタスクを生成するとタスク数が増加し，OS 実行オーバヘッドが大きくなるという問題もある．

このような問題に対処するために，サブスケジューリングという方法を用いる．図 2 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．サブスケジューリングではランナブルはそれぞれ個別の実行タイミングを持つことが可能となる．各ランナブルはサブ周期とサブオフセットを持つ．サブ周期はランナブルが配置されているタスクの起動に対するそのランナブルの実行頻度である．一方，サブオフセットはランナブルの最初の実行タイミングである．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 2 では，タスク Task1 の周期が 2 で，サブレイヤ SL2 のサブ周期が 2，サブオフセットが 1 であるため，SL2 内のランナブルは 2ms，6ms，10ms のタイミングで実行される．サブスケジューリングでは，ランナブルの実行周期を長くすることで CPU 全体の負荷を低減することが可能であり，その上サブ周期が異なるランナブルも同じタスク内で実行されるため，実行順序は維持されている．

2.2　Logical Execution Time

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，タスクが実際に通信を行うタイミングは高優先度のタスク（特に非同期なタスク）の起動やコア間排他制御に影響されるため，タスクの通信の検証は難しい．

タスクの通信のタイミングを一定にする手法として LET（Logical Execution Time）[3] がある．LET の例を図 3 に示す．LET では，タスクごとに一定の時間間隔を LET 区間と定める．また，ランナブルごとにそのランナブルがアクセスする SDG と同等のサイズのローカルデータを用意する．各ランナブルは SDG ではなく，自身のローカルデータに書き込みを行う．LET ではコピーインとコピーアウトがそれぞれ LET 区間の先頭と末尾で行われる．コピーインは SDG を読み込み，ランナブルのローカルデータにコピーする操作である．コピーアウトはランナブルが更新したローカルデータを SDG にコピーする操作である．これらの操作は，LET 区間中に実行されるランナブルが使用する SDG すべてに対して行われる．LET では LET 区間の先頭と末尾でのみ通信が行われるため，通信のタイミングが一定となる．

サブスケジューリングが適用されているパワトレアプリでは，ランナブルの実行周期はサブレイヤの起動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 区間が長いほど他のタスクがデータを受け取るまでの遅延が長くなるため，LET 区間の長さはデータ通信の決定性を保証できる範囲で短く設定されることが望ましい．我々は，サブスケジューリング下のパワトレアプリにおける LET 区間の設定方法をすでに提案した [7]．書き込み側のランナブルは，デッドラインミスが発生しない限り，そのランナブルが配置されているタスクのデッドライン以内に共有データの更新を行うはずである．したがって，LET 区間をサブレイヤの起動タイミングからタスクの周期だけ経過した時刻までの区間に設定する．これにより，サブレイヤの LET 区間はタスクの周期分の長さとなり，タスクの完了とともに他のタスクに更新結果を反映させることができるようになる．

サブレイヤの LET についても図 2 に示されている．図 2 ではサブレイヤ SL2 はサブ周期が 2，サブオフセットが 1 で，SL2 を含むタスクの周期が 2 であるため，LET の区間は $[2, 4], [6, 8], \ldots$ となる．

3.　既存手法の適用

ここでは，既存の LET 通信の実装方法である LET 処理とダブルバッファリングについて述べる．

3.1　LET 処理

Biondi らは，LET のコピーインおよびコピーアウトを専用のタスクによって実装することを提案している [5]．ここで，LET におけるコピーイン，コピーアウトを行う処理単位を LET 処理 (LET-P) と呼ぶ．LET 処理の例を図 4 に示す．LET 処理はタスク起動イベントごとに用意され，高優先度タスクもしくは ISR として実装される．LET 処理の周期は対応するイベントの周期とする．例えば，周期 1ms の時間イベントに対応する LET 処理の周期は 1ms となる．LET 処理はその LET 処理に対応するイベントに関連するランナブルによってアクセスされるローカルデータ

Vol.2019-ARC-235 No.4
Vol.2019-SLDM-187 No.4
Vol.2019-EMB-50 No.4
2019/3/17

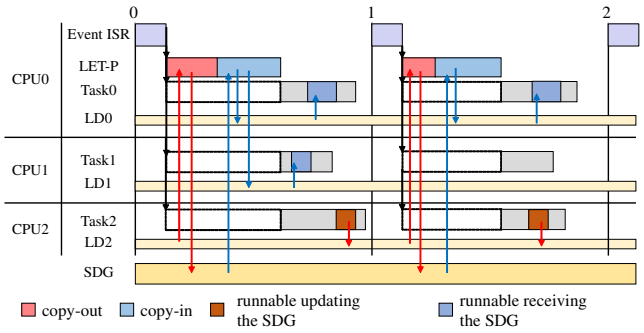

図 4　LET 処理

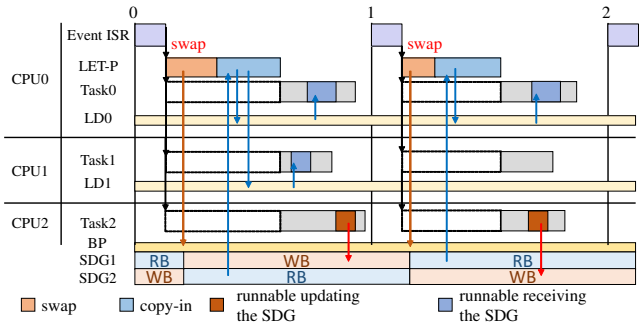

図 5　ダブルバッファリングを用いた LET

について，コピーイン・コピーアウトを行う．LET 処理には，前回の LET 区間中のローカルデータ書き込みに対するすべてのコピーアウトの後にコピーインを行うという制約が必要である．これは，コピーアウトによる共有データの更新を行う前にコピーインが行われないようにするためである．この制約によって，LET 処理はコピーアウトとコピーインの順序を守るために単一の CPU によって実行されなければならない．しかし LET 処理では 10000 個にも及ぶ共有データのアクセスを行うため，LET 処理の実行時間は大きい．また，コピーインを行う前にタスクによって共有データが利用されるのを避けるため，すべてのタスクは LET 処理が完了するまで待つ必要がある．これらの理由から，LET 処理による実行オーバヘッドは大きく，より効率的に LET 処理を実装することが必要である．

3.2　ダブルバッファリング

LET を効率的に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 [6]．ダブルバッファを用いた通信モデルについて図 5 に示す．本手法では SDG ごとに 2 つの分離されたバッファ (ダブルバッファ) を用意する．このバッファのうちの片方をリードバッファ（RB），もう片方がライトバッファ（WB）とする．RB，WB は SDG ごとに用意されたリードバッファポインタ（RBP），ライトバッファポインタ（WBP）によって参照される．読み込み側のランナブルは LET 処理のコピーインによって RB からデータを受け取り，一方書き込み側のランナブルは直接 WB を更新する．そして LET 処理はコピーアウトの代わりに SDG ごとに用意された 2 つバッファポインタのスワップを行う．このようにすることで 2 つのバッファに対するアクセスは競合せず，WB に書き込まれた値はバッファポインタのスワップによって読み込み可能となる．ダブルバッファによって，多数の共有データのコピーアウトがそれより少数の SDG のバッファポインタのスワップに置き換わるため，LET 処理の実行時間を削減することが可能である．本論文の手法はダブルバッファを用いた LET を対象とする．

4.　実際の車載制御システムへの LET の適用

本章では，実際のパワトレアプリに LET を適用するための要件について説明する．また，説明する要件を満たすような LET の実装方法として Deadline Miss Tolerant LET (DMT-LET) を提案する．

4.1　適用要件

タスクはリアルタイム性の要件に応じて，ソフトリアルタイムタスクとハードリアルタイムタスクに分類される．エンジンが高回転であるとき，すべてのタスクをスケジュール可能とするのは困難であるため，比較的安全性に対する重要度の低いソフトリアルタイムタスクのデッドラインミスは許容される．一方，ハードリアルタイムタスクは安全性の要求のために必ずデッドライン内に完了できなければならない．

ここで，ソフトリアルタイムタスクがデッドラインミスした場合の要件を以下に示す．

(R1):　デッドラインミスしたタスクが終了するまでそのタスクの起動は行われない．

(R2):　タスクがデッドラインミスした場合であってもタスク内のサブレイヤの実行タイミングは変化しない．

(R3):　デッドラインミスしたソフトリアルタイムタスクが書き込み側であれば，対応する読み込み側のタスクは書き込み側のタスクが前回の周期で生成した共有データを用いて計算を行う．

ハードリアルタイムタスクがデッドラインミスした場合は OS の機能によるシャットダウンなどで対応するため，本論文でそれらのタスクのデッドラインミスに対応する必要はない．

現状の LET はソフトリアルタイムタスクのデッドラインミスを想定していないため，上記の要件を満たすような LET の実装を検討する必要がある．R1 はイベントによるタスク起動時にタスクの状態を確認し，休止状態でなければタスク起動をスキップするようにすることで満たすことが可能である．R2 はサブレイヤの実行タイミングとタス

クの実行が独立するように実装すれば満たすことができる．例えばサブレイヤの実行タイミングを管理するためのカウンタ変数を，そのサブレイヤを含むタスクではなくタスクを起動するイベントによってインクリメントするように実装すればよい．

一方，R3 はダブルバッファを用いた場合，既存の LET の方実現方法では満たすことができない．図 6 では， 時刻 10，12 で起動したサブレイヤ SLR 内のランナブルは本来時刻 8 で起動したサブレイヤ SLW 内のランナブルが生成したデータを使用するべきである．しかし，サブレイヤ SLW を含むタスクがデッドラインミスしたため，R3 により時刻 4 で起動した SLW が生成したデータを使用すべきであるが，それよりも前の時刻 0 で起動した SLW 内のランナブルが生成したデータを使用することになる．本論文では R3 を満たすことができるようなダブルバッファを用いた LET 通信の実装方法について提案する．

4.2 Deadline Miss Tolerant LET (DMT-LET)

DMT-LET のコンセプトは SDG を書き込むサブレイヤを含むタスクがデッドラインミスしたときに LET 処理によるバッファポインタのスワップを行わないようにすることである．図 7 に DMT-LET を用いた場合の LET を示す．DMT-LET では，各 SDG に対し対応するデータ更新フラグを用意する．DMT-LET における LET 処理の処理について説明する．まず，データ更新フラグをチェックする．データ更新フラグが True のときはスワップを行い，False の場合はスワップを行わない．次にデータ更新フラグを False にする．最後にコピーインを行う．書き込み側のランナブルはダブルバッファリングの適用によって WB に直接書き込みを行うが，WB に対するすべての共有データの更新が終了した際に対応する SDG のデータ更新フラグを True とする．

このようにすることで，LET 処理実行時にバッファポインタをスワップするタイミングで SDG のデータ更新フラグが False であれば，その SDG に対応する WB に書き込むランナブルがデッドラインミスしたことがわかる．そして，デッドラインミスした場合はその SDG に関してバッファポインタのスワップを行わず，次の LET 処理実行時に SDG が更新されていれば（つまり，データ更新フラグが True になっていれば）バッファポインタのスワップが行われる．このようにすることで，図 7 の例では，SLR は少なくとも SLW がデッドラインミスしたときの 1 つ前に SLW によって更新された SDG の値を読み込むことができ，R3 を満たすことができる．具体的には，図 7 では，DMT-LET によって，時刻 10 で実行された LET 処理は書き込み側ランナブルが更新する WB に対応する SDG のデータ更新フラグをチェックし，データ更新フラグが False になっているため SDG のバッファポインタはスワップされない．そ

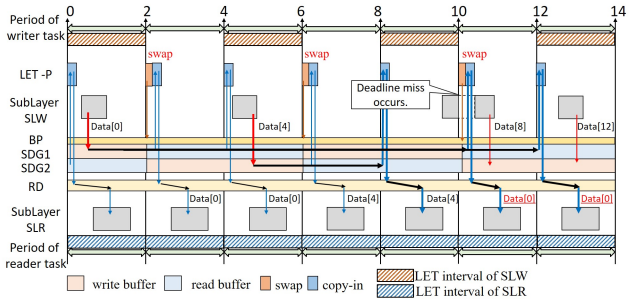

図 6　デッドラインミスが生じたときに起こる問題

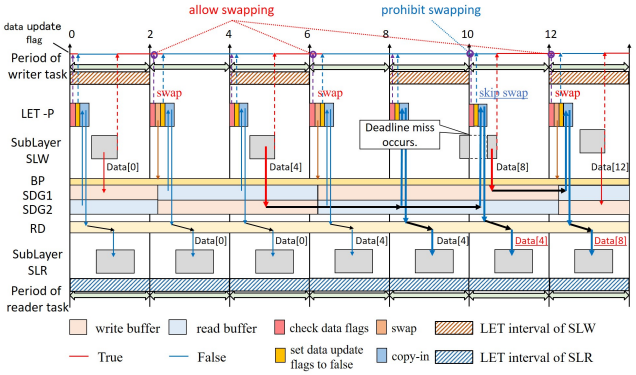

図 7　Deadline Miss Tolerant LET (DMT-LET)

の結果，時刻 10 で起動した SLR 内のランナブルは時刻 4 で起動した SLW 内のランナブルが更新したデータを使用することが可能になっている．

このように DMT-LET によってタスクのデッドラインミスに対応することが可能となるが，データ更新フラグのチェックや False への更新によって LET 処理の実行時間が増加していることに注意する必要がある．

5. LET 分散手法

LET 処理はダブルバッファリングを適用し，コピーアウトによる実行オーバヘッドを削減したとしても，非常に実行負荷が大きく，LET 処理の処理が完了するまで元々のタスクが実行できないことから無駄も大きい．各 CPU あたりの LET 処理による実行負荷を軽減するための方法として，LET 処理を複数 CPU に分散することが考えられる．LET 処理の分散の基本的な考えは，各 CPU に対して LET 処理を用意し，各 CPU に配置されているランナブルに対する SDG への操作をその CPU に用意された LET 処理が実施するというものである．しかし，3.1 節でも述べたように，データの整合性をとるためにコピーアウト（ダブルバッファリング適用下ではバッファポインタのスワップ）とコピーインの実行順序を守らなければならないという制約がある．本節では，DMT-LET によってデッドラインミスの監視および対応しつつ，上記の制約を満たして LET 処理を複数 CPU に分散する手法を提案する．まず，すべてのタスクがハードリアルタイムタスクである場合に適用できる手法である，ADLP(Asynchronized Distributed

LET POrocess) について述べる．次に，すべてのタスクがソフトリアルタイムタスクである場合を対象とした，SDLP(Synchronized Distributed LET Process) について述べる．最後に，実際のパワトレアプリのタスク構成である，ハードリアルタイムタスクおよびソフトリアルタイムタスクが混在するシステムを対象とした HDLP(Hybrid Distributed LET Process) について述べる．

5.1 ADLP

ADLP は LET 区間によってバッファポインタのスワップのタイミングが静的に決まっているというコンセプトに基づいている．そのため，SDG にアクセスするランナブルや LET 処理は，現在時刻を確認し，その時刻に応じて自分がアクセスすべきバッファを選択する．図 8 に ADLP による LET 処理の分散を示す．ある時刻において SDG を読み込むべきランナブルが実行される CPU に配置された LET 処理は現在時刻からどちらのバッファが RB であるかを判断し (図 8(b) の場合は SDG2)，RB から値を読み込み，ローカルデータにコピーする．現在時刻が 0 から 1 までの値であれば、LET 処理は RB として SDG2 を選択し、RB の値を各ランナブルに対応するローカルデータにコピーする．また，書き込みランナブルもまたどちらのバッファが WB であるかを判断し (図 8(b) の場合は SDG1)，直接そのバッファの値を更新する．現在時刻が 0 から 1 までの値であれば、SDG を更新するランナブルは WB として SDG1 を選択し，WB の値を直接更新する．次の区間 (時刻 1 から 2) では，各ランナブルや LET 処理はもう一方のバッファを選択する．

LET によって RB と WB の切り替えのタイミングは静的に決まっている．具体的には，対象の WB を更新するランナブルを含んでいるサブレイヤの LET 区間の末尾において RB と WB が切り替わる．サブレイヤの実行周期はサブレイヤのサブ周期とそのサブレイヤを含むタスクの周期の積であり，その倍の周期がバッファ選択の周期となる．以下に RB の選択の条件式を示す（RB として選択されなかった方が WB となる）．SDG d の 2 つのバッファをそれぞれ d_0，d_1 とする．SDG d を書き込むサブレイヤ s_w を含むタスク τ_w の周期を p_{τ_w}，s_w のサブ周期を p_{s_w}，サブオフセットを of_{s_w} とする．ここで，サブレイヤ s_w の実行周期を $P_w = p_{\tau_w} \cdot p_{s_w}$ とすると，バッファ選択の周期は $2P_w$ である．時刻 $t(0 \leq t < 2P_w)$ のときの SDG d のリードバッファ $d_r(t)$ は下のとおりである．

$$d_r(t) = \begin{cases} d_0 & \text{if } (of_{s_w}+1)p_{\tau_w} \leq t < (of_{s_w}+p_{s_w}+1)p_{\tau_w} \\ d_1 & otherwise \end{cases} \quad (1)$$

この手法では，LET 処理によるバッファポインタのスワップが不要となるため，LET 処理の実行時間を大幅に短縮でき，なおかつスワップとコピーインの順序制約はなくなり，各 LET 処理間で非同期的に実行することが可能である．しかし，ADLP では DMT-LET を適用することができない．なぜならば，タスクのデッドラインミスによりデータを書き込むサブレイヤの実行がスキップされてしまった場合，DMT-LET ではバッファの切り替えを行うべきタイミングが変化するため，式 (1) によって静的に決められたタイミングによってリードバッファを変更することができないからである．したがって本手法の適用対象はハードリアルタイムタスクに限定される．

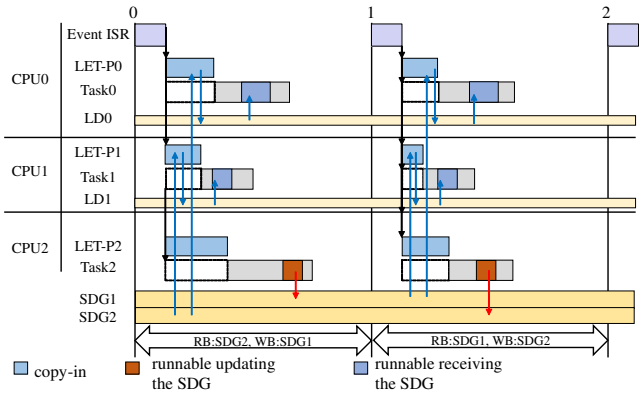

図 8 Asynchronized Distributed LET Process

5.2 SDLP

ソフトリアルタイムタスクが存在する場合に適用する分散手法として，SDLP を提案する．SDLP は DMT-LET を適用しつつ，LET 処理の制約を満たすために LET 処理間で同期を行う手法である．図 9 に SDLP による LET 通信の分散を示す．各 LET 処理では，スワップおよびデータ更新フラグのチェックおよび False への更新を行い，その後他の CPU に配置された LET 処理による同様の操作が完了するまで同期によって待つ．これらの操作がすべて完了した後，各 LET 処理はコピーインを実施する．このようにして，LET 処理を分散しつつ，スワップとコピーインの実行順序の制約を満たすことを可能とする．SDLP はスワップおよびデータ更新フラグのチェックおよび False への更新の操作が必要であり，また同期による待ち時間が発生することから ADLP よりも LET 処理の実行時間は長くなることに注意する必要がある．

5.3 HDLP

ADLP は DMT-LET との両立は困難であるが，バッファポインタのスワップや同期が不要なことから LET 処理の実行時間の削減効果が大きいと考えられるため，ADLP のコンセプトをできるだけ利用したい．対象のパワトレアプリではハードリアルタイムタスクおよびソフトリアルタイムタスクが混在しているため，ハードリアルタイムタスクによって更新される SDG に対しては ADLP を適用し，一

Vol.2019-ARC-235 No.4
Vol.2019-SLDM-187 No.4
Vol.2019-EMB-50 No.4
2019/3/17

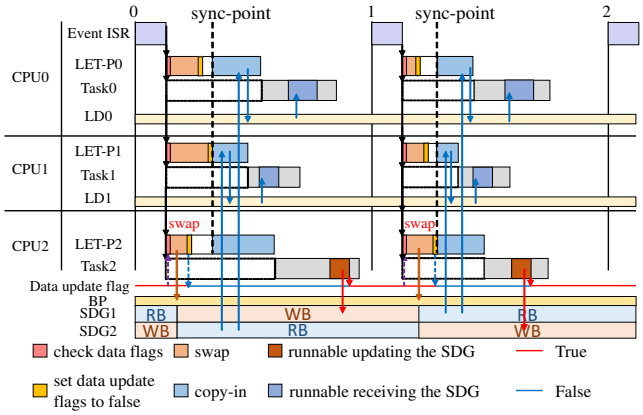

図 9　Synchronized Distributed LET Process

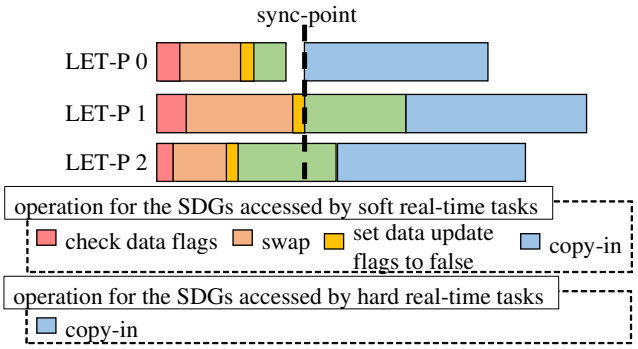

図 10　Hybrid Distributed LET Process (HDLP)

方ソフトリアルタイムタスクによって更新される SDG に対しては SDLP を適用することを考える．この考えに基づき，SDLP と ADLP を組合せた LET 処理の分散方法として，HDLP を提案する．図 10 に HDLP が適用されたときの LET 処理を示す．各 LET 処理は，ソフトリアルタイムタスクによって更新される SDG について，データ更新フラグのチェック，バッファポインタのスワップおよびデータ更新フラグの False への更新を行い，同期フラグを立てる．その後，ハードリアルタイムタスクによって更新される SDG に対するコピーインは LET 処理間で同期される必要がないため，すべての LET 処理の同期フラグが揃うのを待たずに式 (1) で選択した RB に対して行われる．すべての同期フラグが揃った後は，ソフトリアルタイムなタスクによって更新される SDG に対するコピーインを開始する．また，SDG を更新するランナブルがハードリアルなタスクに配置されている場合は式 (1) で選択した WB に対して更新を行い，ソフトリアルタイムタスクに配置されている場合は WBP が指し示す WB に対して更新を行う．

バッファポインタのスワップの回数の削減と同期による待ち時間の短縮により，HDLP は DMT-LET によるデッドラインミスへの対応をしながら SDLP より LET 処理の実行時間を削減することが可能である．

6. 評価

本論文では，SDLP，ADLP，HDLP を適用した場合のメモリ使用量および LET 処理の CPU 利用率を評価する．また，比較対象として 1 つタスクで LET 処理を実装した場合 (単一タスク) についても評価を行う．ここで，ADLP は本来 R3 を満たさないため適用することはできないが，すべてのタスクがハードリアルタイムタスクであると仮定で適用している．

6.1 評価環境

評価環境となるハードウェアは，AURIX アーキテクチャを持つ評価用ボードである TC299B である．AURIX アーキテクチャは，最大 300MHz で動作する CPU を 3 つ持ち，各 CPU は自身のローカル RAM に対して短い遅延でアクセスすることができる一方，他の CPU のローカル RAM やグルーバル RAM に対するアクセスには長い遅延時間が必要となる．評価ソフトウェアは 2.1 節で説明したソフトウェア構成を持ち，その規模は実際のパワトレアプリと同等である．評価ソフトウェアには数十個の時間同期タスクと数個の回転角同期タスクが含まれ，LET 処理は SDLP，ADLP，HDLP のいずれかが適用されている．評価ソフトウェアは AUTOSAR OS（本研究では TOPPERS/ATK2-SC1-MC）上で動作する．ここで，4,000rpm でエンジンが動作しているという想定で回転角イベントの発生頻度が設定されている．各 SDG は最もその SDG に高い頻度でアクセスする CPU が持つローカル RAM に配置されている．

6.2 メモリ使用量の評価

ここでは，SDLP，ADLP，HDLP を用いて実装した場合のメモリ使用量の評価を行う．各手法におけるメモリ使用量を図 11 に示す．結果，それぞれの手法でメモリ使用量に大きな差はなかった．SDLP を用いた実装では，メモリ使用量の観点では単一 LET 処理で実装した場合と差はないため，それらのメモリ使用量は同じである．次に，ADLP を用いた実装では，WBP，RBP およびデータ更新フラグが不要となるため，すべての実装方法で一番メモリ使用量が小さくなる．最後に，HDLP を用いた実装では，一部の SDG について WBP，RBP およびデータ更新フラグが不要となるため，SDLP を用いた場合と ADLP を用いた場合の中間程度のメモリ使用量となる．

6.3 LET 処理の CPU 利用率の評価

ここでは LET 処理に ADLP，SDLP および HDLP を適用した場合の全 LET 処理合計の CPU 利用率の評価を行う．まず，すべてのタスクがハードリアルタイムタスクであるという想定で評価を行う．これは ADLP を適用するこ

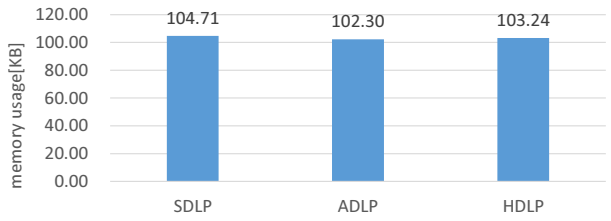

図 11 各通信方法におけるローカルメモリに置かれるデータの合計サイズ

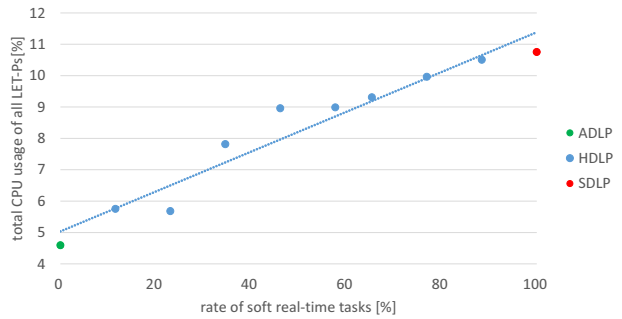

図 12 分散方法ごとの LET 処理の CPU 利用率

と等価である．次に，ソフトリアルタイムタスクとして扱われる可能性の高いタスクから順にソフトリアルタイムタスクに変更していく．このとき，ソフトリアルタイムタスクとハードリアルタイムタスクが混在するため，HDLP を適用する．最終的に，すべてのタスクがソフトリアルタイムタスクとなる．これは SDLP を適用することと等価である．

結果を図 12 に示す．LET 処理の合計実行時間はソフトリアルタイムタスクの比率に比例して増加している．したがって，HDLP は SDLP に比べて LET 処理の実行時間を削減することができている．また，LET 処理の実行時間は ADLP を適用した場合に最も短くなる．しかしながら，ハードリアルタイムタスクのみのシステムは現実的ではない．したがって，LET 処理の実行時間とタスクが本当にハードリアルタイムタスクである必要があるかどうかを考えてソフトリアルタイムタスクおよびハードリアルタイムタスクの割当てを決定していく必要がある．

7. 関連研究

Martinez らは，LET による通信のタイミングを解析し，必要なタイミングでのみデータの送受信を行う手法を提案している [8]．デッドラインミスが生じた場合には通信のタイミングは変化してしまうため，この方法を実際のパワトレアプリに適用する場合にはハードリアルタイムタスク間の通信に限定されると考えられる．

また，Resmerita らは LET によって追加で必要となるローカル領域を削減する手法を適用することで LET の効率化を実現している [6]．この手法では，LET による通信のタイミングを静的に解析し，通信を行うタスクの LET 区間の重なり方に応じて読み込み側もしくは書き込み側のバッファを取り除く．この手法を実際のパワトレアプリに適用する場合にも，デッドラインミスにどのように対応していくかを検討していく必要がある．

8. おわりに

本論文では，タスクがデッドラインミスしたときの影響を軽減する手法として DMT-LET を提案した．そして LET 処理を分散し，各 CPU の負荷を軽減する手法として ADLP，SDLP，HDLP を提案した．評価の結果，各手法でのメモリ使用量にほとんど差はないが，SDLP を適用した場合と比較して，HDLP を適用した場合はメモリ使用量が少なかった．また，HDLP によって SDLP よりも LET 処理の実行時間を削減できることを確認した．

今後の課題としては，LET による通信の遅延を軽減することが挙げられる．

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] Lothar Michel, Torsten Flaemig, Denis Claraz, and Ralph Mader. Shared SW development in multi-core automotive context. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.
[2] AUTOSAR Classic Platform Release 4.4.0 SRS Operating System, 2017.
[3] Thomas A Henzinger, Benjamin Horowitz, and Christoph M Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.
[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*, volume 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
[5] Alessandro Biondi, Paolo Pazzaglia, Alessio Balsini, and Marco Di Natale. Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores. *The WATERS industrial challenge 2017*, 2017.
[6] 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*, pages 1–9. IEEE, 2016.
[7] Masataka Ogawa, Shinya Honda, and Hiroaki Takada. Efficient approach to ensure temporal determinism in automotive control systems. In *8th International Symposium on Embedded Computing and System Design*. IEEE, 2018.
[8] Jorge Martinez, Ignacio Sañudo, Paoloa Burgio, and Marko Bertogna. End-To-End Latency Characterization of Implicit and LET Communication Models. *The WATERS industrial challenge 2017*, 2017.