理 工 学 研 究 科

2019年3月

修 士 論 文

リアルタイム OS を用いたシステムの フルハードウェア実装

47017301 大 迫 裕 樹

(情 報 科 学 専 攻)

修士論文

リアルタイム OS を用いたシステムのフルハードウェア実装

大迫 裕樹

内容梗概

本論文では, リアルタイム OS (RTOS) を利用したシステム全体を, 高位合成を用いて機能等価なハードウェアとして実装する手法を提案する. 本手法では, タスク/ハンドラ等のカーネルオブジェクトおよびRTOS カーネルの機能を, これらを実行する CPU と機能等価なハードウェアに合成する. 全タスク/ハンドラを独立したハードウェアモジュールに合成することにより, 実行可能状態のタスク/ハンドラは並列に実行を行うことが可能になる. これにより, 処理性能を向上させるとともに, コンテキストスイッチの時間を大きく削減する. また, タスク/ハンドラの実行/停止を, その状態に基づいて stall (一時停止) 信号で制御することにより, RTOS のタスクスケジューリング機能を軽量なハードウェアで実現する. さらに, カーネルの API を状態レジスタの参照/更新として実装することにより, 高速化を図る. 本手法に基づく合成システムを実装し, TOPPERS/ASP3 カーネル付属サンプル “sample1” をハードウェア実装した. 実験の結果, タスクの起動を 23 サイクル, 割込みハンドラの起動を 1 サイクルで行えることを確認した.

キーワード

リアルタイムシステム, RTOS, システム合成, ハードウェアアクセラレータ, TOPPERS/ASP3, 高位合成

目 次

第1章　序論

近年, 情報通信技術の発展により, 新しい製品やサービスが次々に提供されるようになっているが, これらに必要な機器を実装するために, 組込みシステムには益々高い機能が要求されるようになっている. さらに, IoT の普及が進む昨今では, ネットワークやセキュリティ等の機能の重要性も増し, システムの複雑化が進んでいる. 特に, 車載機器, 無人飛行機, ロボットの制御には, 高い機能と同時に高い応答性能が要求される.

これらのシステムでは, リアルタイム性が重視される. つまり, システムがある入力を受け取ってから, それに対応する処理を行うまでの時間制約が課される. このようなリアルタイムシステムの開発には, リアルタイム OS (RTOS) が不可欠である. RTOS は, 制約時間内のタスクの実行完了を実現するための機能を提供する. しかし, システムの複雑化が進むにつれ, リアルタイム性の実現は難しい課題となっている.

RTOS を用いたシステムの応答性を向上させる一手法として, RTOS の機能をハードウェアとして実装する手法がある. 文献 [1] [2] [3] では, ハードウェアにより, RTOS のスケジューラの高速化を行っている. また, 文献 [4] [5] [6] では RTOS のほとんどの機能をハードウェア実装している. これらの手法は, カーネル時間 (CPU が動作している時間のうち, カーネルのプログラムを実行している時間) を大きく削減することができる. しかし, これらの手法では, タスク/ハンドラはソフトウェアのままであり, プロセッサのタスク切り替えのオーバーヘッドは改善されない. また, タスクの処理の負荷が大きい場合は応答時間が改善されない場合もある.

一方, 高位合成技術 [7] を用いることにより, タスク/ハンドラを専用のハードウェアモジュールに合成し, 高速化することが可能である. これを用いて, 指定したタスク/ハンドラをハードウェアに合成するためのシステムレベル設計手法 [8] [9] が提案されている. これにより, アプリケーションプログラム (タスク/ハンドラが実行するプログラム) の処理が重い場合にも大きく高速化することが可能になる. しかし, これらの手法では, RTOS および一部のアプリケーションプログラムはソフトウェアとして実行する必要がある.

文献 [10] [11] は割込みハンドラを含むプログラムから, システム全体のハードウェア化を行っている. この手法で生成したシステムは全てハードウェアとして動作するため, より多くの場合で高速化が可能となっている. しかし, 合成対象はベアメタルシステムのみであり, RTOS を使用することはできない.

本研究では, この課題を解決する手法として, RTOS およびタスク/ハンドラの全てをフルハードウェア化する手法を提案する [12][13]. RTOS を用いたプログラムを入力することにより, タスク/ハンドラ, および RTOS のサービスコールの機能全てをハードウェアに合成する. 本手法では, 各タスク/ハンドラを独立したハードウェアモジュールに合成する. 実行可能状態になったタスク/

ハンドラは, 即座に実行を開始することができる. また, 全てのタスク/ハンドラを並列に実行させることを可能にし, 処理性能とレスポンスの向上を目指す. さらに, これにより, 従来の高速化手法と異なり, 複雑な RTOS のタスクスケジューリング機能が不要になる. 本手法では, スケジューリングキューを廃し, タスク/ハンドラの実行/停止を制御する軽量なハードウェアに置き換える.

本手法に基づき, TOPPERS/ASP3 カーネルを用いて C 言語で記述されたリアルタイムシステムから, Verilog HDL で記述されたハードウェア設計を生成するための合成システムを実装した. 実験の結果, タスクの起動が 23 サイクル, 割込みハンドラの起動が 1 サイクルで行えることを確認した.

第2章　リアルタイム OS

RTOS (Real-Time Operating System) はリアルタイムシステムのためのオペレーティングシステムである．ここにおける “リアルタイム” とは, 最悪応答時間が決まっていることを表す．つまり, 特定のイベントが発生してから, それに対応する処理が行われるまでの所要時間の最悪値が保証されることを示す．

RTOS の提供する逐次処理の実行単位にはタスクおよびハンドラが存在する．また, RTOS は, これらの間で同期 · 通信を行うための機能 (カーネルオブジェクトおよび API) も提供する．リアルタイム性を実現するためには, 各処理の順序付け (スケジューリング) が重要である．多くの場合では, 各処理の重要度をベースにスケジューリングを行う．本論文では, この値を “優先度” と呼ぶ．

2.1　タスク管理機能

RTOS は, 複数の逐次処理を並行に動作させるための仕組みを提供する．本論文では, この処理単位を “タスク” と呼ぶ．リアルタイムシステムでは, タスク毎にデッドラインが定義され, 各タスクの処理はこのデッドラインまでに処理を完了することが求められる．

タスクは特定の条件を満たすと実行可能状態となる．実行可能状態のタスクが存在する時, そのうちの 1 つが実行され, その他のタスクは CPU 待ち状態となる．実行タスクの選択は, 各タスクに設定された優先度に基づいて RTOS のスケジューラが行う．RTOS のカーネルによっては, 動的な優先度変更や, 実行中のタスクを中断してより優先度の高いタスクに実行を切り替えるプリエンプションが可能な場合もある．

2.2　ハンドラ管理機能

“ハンドラ” は, イベントを処理するための処理単位である．ハンドラにはタイムイベントハンドラや割込みハンドラ, CPU 例外ハンドラ等がある．タイムイベントハンドラはタイマ割込みがトリガとなるため割込みハンドラに分類されることもあるが, 本論文では, タイマ割込みにより起動されるハンドラを “タイムイベントハンドラ”, それら以外の割込みにより起動されるハンドラを “割込みハンドラ” と呼ぶものとする．

タイムイベントハンドラは, 周期ハンドラやアラームハンドラ, オーバランハンドラ等の総称である．周期ハンドラは, アクティブ状態の間, 特定の周期で繰返し実行される．アラームハンドラ

は, 起動されてから指定された時間が経過した後に 1 度だけ実行される. オーバランハンドラは, タスクが使用したプロセッサ時間が指定値を超えた場合に起動される.

割込みハンドラは, 外部割込みのハンドリングを行う. 外部から割込み要求信号 (IRQ) が送られた時に, 割込みハンドラを経由して対応する処理 (割込みサービスルーチン, ISR) が呼ばれる. 割込みもタスク同様, 優先度を持つ. 複数の割込みが同時に発生した場合, 優先度が高い方が実行される. 場合によっては, 割込み処理中により高優先度の割込み (多重割込み) を許可できることもある.

一般的に, これらのハンドラは, タスクよりも高い優先度で実行される.

2.3 同期・通信機能

RTOS は, タスクやハンドラの管理機能に加えて, タスク間の連携を実現するための排他制御やタスク間通信等の機能を提供する. これは, タスクとは独立したオブジェクトにより実現される.

同期機能は, 複数のタスク間で処理のタイミングに依存関係がある場合にタスク間の同期を取る機能である. 同期機能の例としては, セマフォやミューテックス, イベントフラグ等が挙げられる. セマフォおよびミューテックスはタスク間の排他制御が必要な場合に, イベントフラグは複数のタスク内の処理に依存関係が存在する場合に用いる同期オブジェクトである.

一方, 通信機能は複数のタスク間の処理タイミングだけではなく, データをタスク間で受け渡す等, 処理内容にも依存関係がある場合に用いる. 通信機能の例としては, データキューやメッセージバッファ等が挙げられる. これらは FIFO 順でメッセージを送受信するための通信オブジェクトである.

2.4 TOPPERS

TOPPERS は, 組込みシステム分野で標準的に用いられている μITRON 仕様に準拠したリアルタイムカーネル群で, 用途に応じて様々なカーネル (および拡張パッケージ) が提供されている.

TOPPERS/ASP3[1] は, μITRON4.0 [14] 準拠の第 3 世代リアルタイムカーネルである. TOPPERS プロジェクトの中でも, μITRON の標準的な機能セット (スタンダードプロファイル) を拡張・改良する形で作られており, その他の保護機能付きのカーネルやマルチプロセッサ対応のカーネルは, これをベースに開発されている.

TOPPERS/ASP3 カーネルは, 信頼性・安全性・リアルタイム性を重視しており, カーネル内で動的なメモリ管理が必要とならないように設計されている. これは, 高い信頼性・安全性・リアルタイム性を実現する場合, システム稼働中に発生するメモリ不足への対処が難しいためである. この方針から, TOPPERS/ASP3 カーネルでは, カーネルオブジェクトは, コンパイル時に静的に生成される. つまり, 動的なオブジェクト生成はサポートされておらず, 生成するオブジェクトは全

[1] http://www.toppers.jp

てコンパイル時に決定する必要がある. ただし, アプリケーションプログラムが動的なメモリ管理をするための固定長メモリプール機能は使用することができる.

実行可能タスクは, レディーキューで管理され, 実行中のタスクが, 広義の待ち状態もしくは休止状態に遷移すると, 次の実行タスクがキューから選択され, 実行状態となる. 全タスクは単一のメモリ空間を共有する. スタックのサイズおよび開始アドレスはタスク/ハンドラ毎にコンフィギュレーションファイルで指定することが可能である.

2.5 リアルタイムシステムにおけるオーバーヘッド

前述の通り, RTOS を用いることにより, 演算リソースの分配やタスク間の同期・通信が容易になる等の様々なメリットがある一方で, RTOS 機能自体の処理時間も必要となるというデメリットも存在する. 開発者は, これらの時間を含めてデッドラインを守る必要がある.

リアルタイムシステムにおいて生じるオーバーヘッドとしては, 以下の 3 つが挙げられる.

(1) スケジューリング: 新しいタスクもしくはプリエンプトされたタスクをレディーキューに追加し, 次に実行されるタスクを選択する処理.

(2) コンテキストスイッチ: 実行中のタスク/ハンドラのコンテキスト (レジスタの値) を保存し, 次に実行されるタスク/ハンドラのコンテキストを読込む処理.

(3) CPU 待ち: 実行可能状態のタスクが, 自身より高優先度のタスクの実行完了を待つ時間.

上記 (1) のオーバーヘッドの改善手法として, RTOS のスケジューラをハードウェア化する手法 [1, 2, 3] を用いることができる. タスクスケジューリングをハードウェアアクセラレータで行うことにより, スケジューリングに要する時間を軽減することが可能である. また, RTOS 機能のハードウェア化 [4, 5, 6] についても, RTOS の大部分をハードウェア化することにより, 同様の高速化が可能である.

しかし, これらの手法を用いることによりスケジューリング処理を高速化できる一方で, そのオーバーヘッドを完全に取り除くことはできない. CPU からハードウェアアクセラレータへの切り替え時間や, ある程度のスケジューリングのための時間が必要になるためである.

また, これらの手法では, (2) および (3) のオーバーヘッドの改善はできない. (2) のコンテキストスイッチのオーバーヘッドの改善には CPU 側でのハードウェアによる高速化が必要である. また, (3) の CPU 待ち時間の改善には CPU の並列処理機能の改善が必要となる. つまり, これらについては, CPU とハードウェアアクセラレータでのシステム開発においては改善は難しいといえる.

第3章　RTOS を用いたシステムのフルハードウェア実装

本論文では, リアルタイムシステムを高位合成によりフルハードウェア実装する手法を提案する [12][13]. RTOS 上で動作するアプリケーションプログラムを入力することにより, これを実行する CPU と機能等価なハードウェアを自動生成する. 全タスク/ハンドラを並列実行実行可能にすることにより応答性能を向上させ, また, スケジューラの軽量化を可能にする.

3.1　概要

本手法で行うハードウェア生成の概観を図 3.1 に示す. 入力となるシステムは, タスク (TSKi) および周期/アラーム/割込みハンドラ (CYCi, ALMi, INTi) からなる. これらは RTOS により実行/停止の制御が行われる. 本論文では, タスク/ハンドラの数は合計 10 個程度と想定する. タスク/ハンドラおよび RTOS のプログラムは命令メモリ (IMEM) に格納されている. 本手法では, タスク/ハンドラを含む全てのカーネルオブジェクトはコンパイル時に静的生成されるものとする. すなわち, 実行時に RTOS の API を用いた動的生成は行わないものとする.

本手法は命令メモリおよび CPU を, プログラムを書き換えることなく, 機能等価なハードウェアに自動合成することを可能にする. 生成されるハードウェアモジュールは, 図 3.1 下部の TSKi, CYCi, ALMi, INTi のように, 各タスク/ハンドラ毎に独立している. タスク/ハンドラの管理を行っていた RTOS のプログラムおよび, これを実行する CPU は, 実行制御を行う manager モジュールおよび, メモリアクセス調停を行う arbiter モジュールに置き換える.

本手法により生成される各タスク/ハンドラは, 独立したハードウェアモジュールであり, 個別に実行制御される. これにより, 複数のタスク/ハンドラの並列動作が可能となる. つまり, 実行可能状態になったタスク/ハンドラはその優先度によらず, 即座に実行を開始できる.

manager は, 従来のタスクスケジューラやレディーキューに代わってタスク/ハンドラの実行/停止を管理する制御モジュールである. 通常, 全てのタスク/ハンドラは, その優先度や状態を考慮せず, 記述されたプログラム通りの処理を続ける. タスクが広義の待ち状態 (待ち状態, 強制待ち状態, 二重待ち状態) や休止状態になった場合, ハンドラが実行停止状態になった場合等, 実行を停止すべき時は, manager モジュールから対象となるタスク/ハンドラへ, 実行を一時停止するための信号 (stall) を送信する.

複数のモジュールから同時にメモリ (DMEM) へのアクセス要求があった場合は, arbiter モジュールが調停を行う. つまり, 同時に同時発生した複数のメモリアクセス要求は, 1 つのみが許可され,

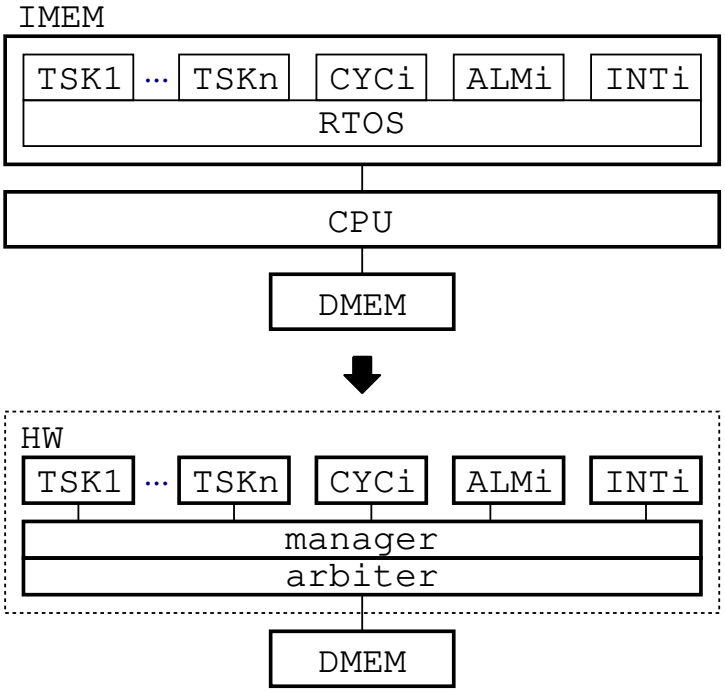

図 3.1: RTOS を用いたシステムのフルハードウェア実装

許可されなかったタスク/ハンドラモジュールには stall 信号が送られる．この調停はタスク/ハンドラの優先度を用いて行う．

これらの仕組みにより, 2.5 節で記述したリアルタイムシステムにおけるオーバーヘッドが解消される．本手法により生成されたシステムはレディーキューを持たないため, スケジューリングのオーバーヘッドが無くなる．また, タスク/ハンドラのモジュールが独立していることにより, ゼロオーバーヘッドでのコンテキストスイッチが可能である．さらに, タスク/ハンドラ単位での並列実行が可能であることから, CPU 待ちのオーバーヘッドも除去される．

なお, 本論文では, 入力となるプログラムは, 複数のタスク/ハンドラが並列に動作可能であるように排他制御が行われていることを前提とする．すなわち, 同時に 2 つ以上のタスク/ハンドラが同時に動作しないことを前提とした排他制御を行っているシステムの合成は本手法の対象外とする．例えば, 高優先度のタスクが実行する処理と低優先度のタスクが実行する処理が競合する場合, 高優先度のタスクのみが実行されるということを前提するのではなく, これらのタスク間で適切に排他制御を行う必要がある．

3.2 ハードウェア構成

本手法で合成するハードウェアの詳細構成を図 3.2 に示す．TSKi, CYCi, ALMi, および INTi は, それぞれ, 図 3.1 のタスク/ハンドラである．mangaer および arbiter も同様に, 図 3.1 で示した mangaer, arbiter である．図 3.2 左の local mem および global mem は, 図 3.1 の DMEM に相当し, 複数のバンクに分割している．mutex は排他制御を実現するためのハードウェアミュー

テックスである.

各タスク/ハンドラモジュールはメモリアクセス用の addr および data の入出力ポートを持つ. addr 信号で読み書き対象のアドレスを指定し, data 信号でメモリデータの送受信を行う. また, タスク/ハンドラは, 制御用の信号として stall, reset の入力ポートを持つ. stall が 1 の時, タスク/ハンドラの動作は停止し, それ以外の場合は実行される. reset が 1 の時, タスク/ハンドラはリセットされ, 初期状態に戻る. また, 各タスク/ハンドラモジュールは実行完了を通知するための end 信号を受け取るポートを持つ.

manager モジュールはタスク/ハンドラの状態を status レジスタとして持つ. status レジスタは, 各タスク/ハンドラモジュールごとの実行状態, 優先度, 待ち要因等を保持する. stall 信号および reset 信号の値は, 各 status レジスタ (task status, cyclic status, alarm status, int status) の値を基に決定する. 実行状態が広義の待ち状態/休止状態の時や, メモリアクセス待ち, ミューテックスロック待ちの場合, stall 信号を 1 にし, タスク/ハンドラの実行を停止させる.

status レジスタはメモリ空間にマッピングする. すなわち, タスク/ハンドラからは特定のアドレスに対するメモリアクセスで読み書きができるようにする. RTOS の API の大部分は status レジスタを参照/更新する処理として実装する. 例えば, タスク 1 が, タスク 2 を実行状態に遷移させる API は, タスク 2 の status ジレスタを更新する処理により実装できる. この処理は, タスク 1 が, タスク 2 の status レジスタに対応するアドレスに対するメモリアクセスを行うことにより実現される.

レジスタにマッピングされたアドレス以外に対するメモリアクセスの場合, manager は arbiter へアクセス要求信号を送り, メモリの読み書きを行う. この時, アクセス要求元のタスク/ハンドラの優先度も pri 信号により送信する.

周期ハンドラ/アラームハンドラについても, タスク同様に制御が可能である. 例えば, ハンドラが停止状態の時, stall の値は 1 であり, 実行状態になると同時に 0 に変化する. タスクの場合との違いは, 周期ハンドラ/割込みハンドラの status レジスタは, タイムイベントの制御用に, 実行開始までの時間の値 (クロックカウンタ) をレジスタ内に保持する点である. カウンタの値はクロック毎に減じられ, 値が 0 になると stall が解除され, ハンドラの実行が開始する.

割込みハンドラは, manager への IRQ (割込み要求) 信号によって起動される. IRQ 信号が送られた時, manager モジュールは割込み処理が可能な状態であれるかどうかの確認を行う. 処理が可能であれば, 割込みハンドラへの stall が解除され, ハンドラが起動する.

また, 全割込みロックフラグ, CPU ロックフラグ, 割込み優先度マスク, ディスパッチ禁止フラグ等のシステム全体の状態は, manager 内の global status レジスタに保存される.

本手法で合成するハードウェアでは, 全タスク/ハンドラが並列に実行可能であるため, 同一のメモリバンクに対して複数のアクセスが同時に発生する可能性がある. arbiter モジュールは, これらのアクセス要求をタスク/ハンドラの優先度を用いて調停する. つまり, 最も高い優先度を持つタスク/ハンドラのアクセスが優先され, その他のモジュールに対しては manager モジュールを介して stall 信号が送られる.

RTOS の提供するミューテックスは, ハードウェアモジュールとして実装する. タスク/ハンド

ラからは API により取得/解放要求が可能である. 複数タスク/ハンドラから取得要求があった場合, ミューテックスを獲得することができるのは 1 モジュールのみである. その他のミューテックス待ちのモジュールには manager モジュールを介して stall 信号が送られる.

また, タスク/ハンドラの並列実行により, 処理内容が競合する API 要求が同時発生する可能性がある. 本論文では, 同時に複数の API 要求が発生した場合, 競合の有無に関わらず, 1 モジュールからの要求のみが処理されるようにする. この排他制御は, タスク/ハンドラ自身がミューテックスを用いることにより実現することができる. その他のモジュールには stall 信号が送られる.

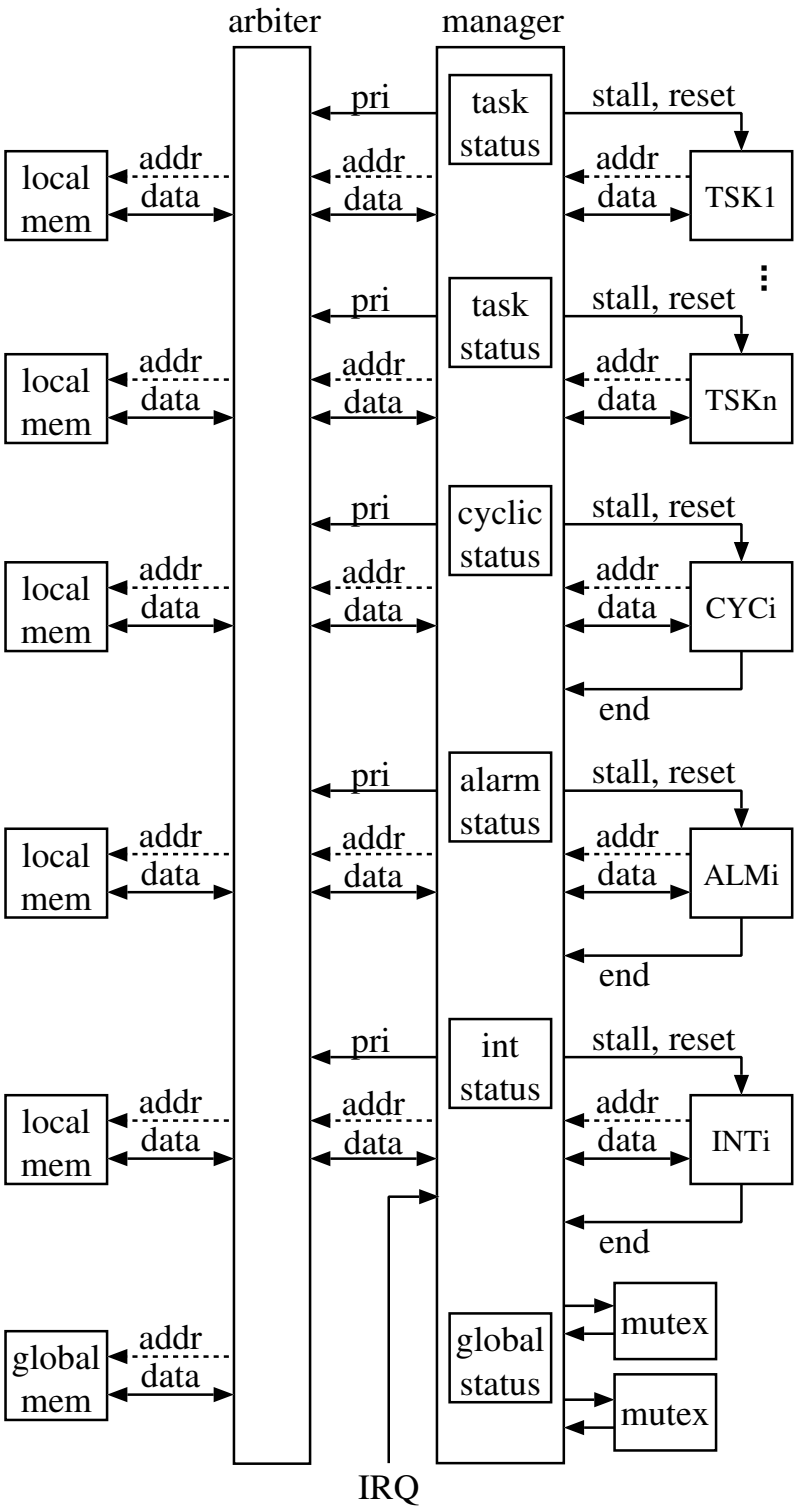

図 3.2: 合成されるハードウェアの構成

第4章　TOPPERS/ASP3 を用いたシステムからの合成

本章では, 提案手法に基づいた, TOPPERS/ASP3 カーネルを用いたプログラムを対象にした合成手法の詳細について述べる.

4.1　タスクの実行制御

TOPPERS/ASP3 では, タスク/ハンドラはコンパイル時に静的生成される. 各タスクは, 表 4.1 の 6 つのうち, どれかの状態をとる. 初期状態はコンパイル時にコンフィギュレーションファイルで指定する. 表 4.2 はタスク/ハンドラの状態制御に関する主要なサービスコールである. これらのサービスコールを用いることにより, 図 4.1 のように, タスクの状態を遷移させることができる. 例えば, act_tsk は指定した ID のタスクを休止状態から実行可能状態に遷移させる.

タスクへの stall 信号は実行状態の時のみ 0 となり, 実行が許可される. その他の状態 (休止, 実行可能, 待ち, 強制待ち, 二重待ち) の時は 1 とし, 一時停止する. また, メモリアクセス要求もしくはミューテックス取得要求が待たされている場合にも stall 信号は 1 となる.

全タスクは並列に動作することが可能であるため, 実行可能状態になったタスクは, ディスパッチ禁止状態でなければ即座に実行状態に遷移する. この状態遷移の管理は manager モジュールが行う. あるタスクが実行可能状態になった時, ディスパッチが禁止されていなければ, manager モジュールは, 次のクロックでタスクの状態を実行状態に更新し, stall 信号を解除する.

起動された状態のタスクがサービスコール等により休止状態に遷移した時, manager は reset 信号と stall 信号を送り, タスクの初期化を行う.

表 4.1: TOPPERS/ASP3 におけるタスクの状態

状態	意味
休止状態	タスクが実行すべき処理がない状態
実行可能状態	タスク自身は実行できる状態にあるが, それよりも優先順位の高いタスクが実行状態にあるために, そのタスクが実行されない状態
実行状態	タスクが実行されている状態. または, そのタスクの実行中に, 割込みまたは CPU 例外により非タスクコンテキストの実行が開始され, かつ, タスクコンテキストに戻った後に, そのタスクの実行を再開するという状態
待ち状態	タスクが何らかの条件が揃うのを待つために, 自ら実行を止めている状態
強制待ち状態	他のタスクによって, 強制的に実行を止められている状態
二重待ち状態	待ち状態と強制待ち状態が重なった状態

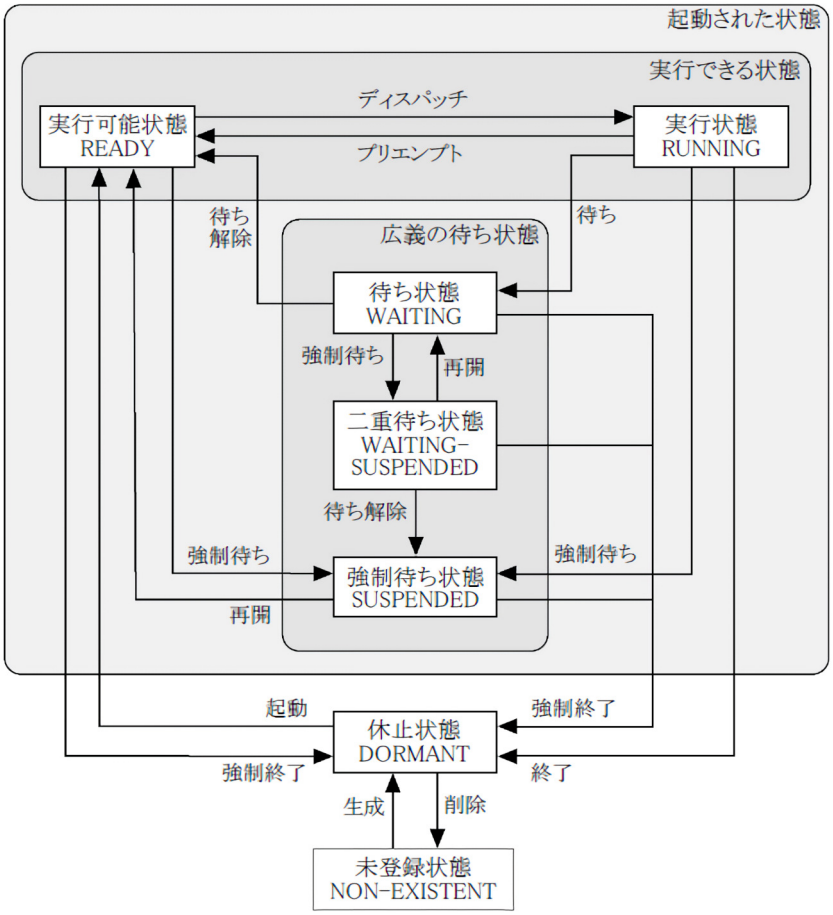

図2-2. タスクの状態遷移

図 4.1: TOPPERS/ASP3 におけるタスクの状態遷移 [15]

各タスクの状態は, manger モジュール内の status レジスタに保存する. ASP3 カーネルでは, タスクの状態変数 (C 言語の構造体) は表 4.3 の 10 個のメンバからなるオブジェクトであり, 本手法では, これらに対応する 10 個のレジスタを用いる. status レジスタはメモリ空間にマッピングされており, タスクモジュールからはメモリのロード/ストア命令でアクセスできる. ASP3 カーネルのほとんどのサービスコールは status レジスタの参照/更新処理として実装できる.

図 4.2 に, タスクを休止状態から実行可能状態に遷移させるサービスコール act_tsk の処理内容を示す. 28 行目が status レジスタを書き換えることにより, タスクを実行可能状態にする処理であり, 3–5, 8–9 行目がサービスコール内部の前処理 7, 15–21 行目はエラーチェックである. このうち, 18–21 行目については, 設定によってエラーとなるか処理を行うかがコンパイル時に決定するため, プリプロセッサディレクティブにより切り替えができるようにしている. 13 行目で actcnt の値を変数に格納しているのは, volatile 変数への複数回アクセスを避け, 実行時間を短縮するためである. これらは, TOPPERS の統合仕様書 [15] に準拠した処理内容であり, 本質的には ASP3 カーネルのソースコードと同様である. 11, 31 行目はサービスコールのシリアライズのためのミューテックスの取得/解放処理であり, これが本手法で新たに追加した部分である.

図 4.3 は, タスクを起床待ち状態から実行可能状態に遷移させるサービスコール wup_tsk の処理内容である. 22 行目が status レジスタを書き換えることにより, タスクを実行可能状態にする処理である. それ以外のメインとなる status の更新処理は 23, 27 行目である. 3–5, 8–9 行目がサービスコール内部の前処理 7, 15–17, 25 行目はエラーチェックである. これらも TOPPERS の統合仕様書に準拠した処理内容であり, 本質的に異なるのは 11, 30 行目の同期処理のみである.

高速化のため, 各サービスコールはインライン展開した後に高位合成する.

表 4.2: 実行制御と排他制御のためのサービスコール

(a) タスク実行制御

サービスコール	機能
act_tsk(ID tskid)	指定したタスク (対象タスク) に対して起動要求を行う
slp_tsk()	自タスクを起床待ちさせる
tslp_tsk(TMO timeout)	自タスクをタイムアウト付きで起床待ちさせる
wup_tsk(ID tskid)	指定したタスク (対象タスク) を起床する
rel_wai(ID tskid)	指定したタスク (対象タスク) を強制的に待ち解除する
sus_tsk(ID tskid)	指定したタスク (対象タスク) を強制待ちにする. 対象が待ち状態であれば, 二重待ち状態にする
rsm_tsk(ID tskid)	指定したタスク (対象タスク) を, 強制待ちから再開する. 対象が二重待ちであれば, 待ち状態にする
dly_tsk(RELTIM time)	指定した時間, 自タスクを待ち状態にし, 遅延させる
ext_tsk()	自タスクを終了させる
ras_ter(ID tskid)	指定したタスク (対象タスク) に終了要求を行う
ter_tsk(ID tskid)	指定したタスク (対象タスク) を終了させる

(b) ハンドラ実行制御

サービスコール	機能
sta_cyc(ID cycid)	指定した周期通知 (対象周期通知) を動作開始する
stp_cyc(ID cycid)	指定した周期通知 (対象周期通知) を動作停止する
sta_alm(ID almid, RELTIM almtim)	指定したアラーム通知 (対象アラーム通知) を動作開始する. 通知時刻は sta_alm を呼び出してから, almtim で指定した相対時間後に設定される
stp_alm(ID almid)	指定したアラーム通知 (対象アラーム通知) を動作停止する

(c) 排他制御

サービスコール	機能
loc_mtx(ID mtxid)	指定したミューテックス (対象ミューテックス) をロックする
unl_mtx(ID mtxid)	指定したミューテックス (対象ミューテックス) をロック解除する

表 4.3: タスクの状態変数

状態変数	意味
tskstat	現在の実行状態
tskpri	現在の優先度
tskbpri	ベース優先度
tskwait	待ち要因
wobjid	待ち対象のオブジェクトの ID
lefttmo	タイムアウトするまでの時間
actcnt	起動要求キューイング数
wupcnt	起床要求キューイング数
raster	タスク終了要求状態
dister	タスク終了禁止状態

```
ER act_tsk(ID tskid) {

  if (IS_TASK_CONTEXT && tskid == TSK_SELF) {
    tskid = TOPPERS_HW_SELF_ID;
  }

  if (tskid <= 0 || TNUM_TSKID < tskid) { return E_ID; }
  volatile T_RTSK *const target =
    &(task_status[tskid - 1].rtsk);

  _loc_service_call();

  const uint_t actcnt = target->actcnt;

  ER rc = E_OK;
  if       (glob_status.f_cpu_locked  ) { rc = E_CTX;  }
  else if (actcnt >= TMAX_ACTCNT      ) { rc = E_QOVR; }
#ifdef SET_TA_NOACTQUE
  else if (target->tskstat != TTS_DMT) {
    rc = E_QOVR;
  }
#else
  else if (target->tskstat != TTS_DMT) {
    target->actcnt = actcnt + 1;
  }
#endif
  else {
    target->tskstat = TTS_RDY;
  }

  _unl_service_call();

  return rc;
}
```

図 4.2: サービスコール act_tsk の C 言語記述

```
ER wup_tsk(ID tskid) {

  if (IS_TASK_CONTEXT && tskid == TSK_SELF) {
    tskid = TOPPERS_HW_SELF_ID;
  }

  if (tskid <= 0 || TNUM_TSKID < tskid) { return E_ID; }
  volatile T_RTSK *const target =
    &(task_status[tskid - 1].rtsk);

  _loc_service_call();

  const STAT tskstat = target->tskstat;

  ER rc = E_OK;
  if       ( glob_status.f_cpu_locked ) { rc = E_CTX; }
  else if  ( tskstat == TTS_DMT       ) { rc = E_OBJ; }
  else if  (
    tskstat == TTS_WAI
    && target->tskwait == TS_WAITING_SLP
  ) {
    target->tskstat = TTS_RDY;
    target->tskwait = 0;
  }
  else if ( target->wupcnt >= TMAX_WUPCNT ) { rc = E_QOVR; }
  else {
    target->wupcnt++;
  }

  _unl_service_call();

  return rc;
}
```

図 4.3: サービスコール wup_tsk の C 言語記述

表 4.4: 周期/アラームハンドラの状態変数

(a) 周期ハンドラ

状態変数	意味
cycstat	周期通知の動作状態
lefttim	次回通知時刻までの相対時間
f_reset	タイマリセットフラグ

(b) アラームハンドラ

状態変数	意味
almstat	アラーム通知の動作状態
lefttim	通知時刻までの相対時間

4.2 周期ハンドラ/アラームハンドラの実行制御

周期ハンドラ/アラームハンドラは, 基本的にタスクと同様の方法で制御する. manager は, 各ハンドラの status レジスタの値を基に stall 信号を送る. タスク制御との相違点は, ハンドラは end 信号を manager に送ることにより, 実行完了を通知する点である.

各ハンドラが status レジスタ内に持つ状態変数を表 4.4 に示す. TOPPERS/ASP3 の周期ハンドラでは, 各ハンドラに対して 2 つの変数を持つ. cycstat はハンドラの停止/動作中の状態を表し, 値が 1 であれば動作中, 0 であれば停止中である. lefttim はタイマ変数であり, 次の実行までの時間をミリ秒単位で保存する. 周期ハンドラが起動するとタイマには周期時間がセットされる. また, タイマに付随する変数として, サブカウンタが存在する. サブカウンタは ASP3 で扱うことのできないミリ秒未満の値を保持し, クロック毎に減算される. サブカウンタが 0 になると, タイマの値も減算される. タイマが 0 になると, ハンドラへの stall が解除され, また, 同時にタイマも再度セットされる.

manager モジュールは, 動作開始のリクエストを保存するためのフラグ変数として f_reset を補助的に用いる.

周期ハンドラの具体的な動作の流れは以下の通りである.

1) (task) sta_cyc() を呼ぶ. このサービスコールは, f_reset, cycstat に 1 をセットする.

2) (manager) f_reset が 1 ならば, lefttim に周期時間をセットし, f_reset を 0 にする.

3) (manager) lefttim をデクリメントする

4) (manager) lefttim が 0 なら, stall を 0 にし, ハンドラの実行を開始する. また, lefttim に周期時間を再セットする.

5) (handler) 実行が完了すれば, end 信号を 1 にする.

6) (manager) end 信号が 1 ならば, stall, reset をともに 1 にし, ハンドラを初期状態に戻す.

7) (manager) 3 に戻る.

アラームハンドラの動作の仕組みは概ね周期ハンドラと同じである. 異なる点は, アラームハンドラは繰り返し処理を行わないため, manager モジュールが lefttim の値を再セットしない点である.

4.3 割込みハンドラの実行制御

割込みハンドラは, 周期ハンドラ/割込みハンドラと同様に制御する. manager は割込み要求信号を受け取ると, 割込みが禁止されていなければ割込みハンドラへの stall 信号を 0 にする.

TOPPERS における割込みは, 複数のデバイスが接続した “要求ライン” と呼ばれる信号線のモデルを用いてグループ化される. あるデバイスから割込みが発生した時, 同じ割込み要求ラインに接続している全てのデバイスの割込みサービスルーチン (ISR) が実行される. 各割込み要求ラインには動的に変更可能な優先度が設定され, この値が割込み優先度マスクより大きい場合は割込み処理は行われない. また, 各割込み要求ラインは割込み禁止フラグを持つ.

本手法では, 各割込み要求ラインに対して 1 つの割込みハンドラのハードウェアモジュールを合成し, このモジュールが割込み要求ラインに接続している全てのデバイスの割込みサービスルーチンを実行する.

割込み要求が発生した時, manager モジュールは以下の条件を満たしているかを確認する.

- 割込み要求ラインに対する割込み要求禁止フラグがクリアされていること
- 割込み要求ラインに設定された割込み優先度が, 割込み優先度マスクの現在値よりも高い (優先度の値としては小さい) こと
- 全割込みロックフラグがクリアされていること
- 割込み要求がカーネル管理の割込みである場合には, CPU ロックフラグがクリアされていること

これらの条件を満たしていた場合, manager モジュールは, ハンドラの実行状態を保存する exec レジスタに 1 をセットし, stall 信号を 0 にし, ハンドラの実行を開始する. ハンドラの実行が完了すると, ハンドラモジュールは, manager モジュールへの end 信号値を 1 にし, 実行完了を通知する. その後, manager モジュールは, exec レジスタを 0 に, stall を 1 にする.

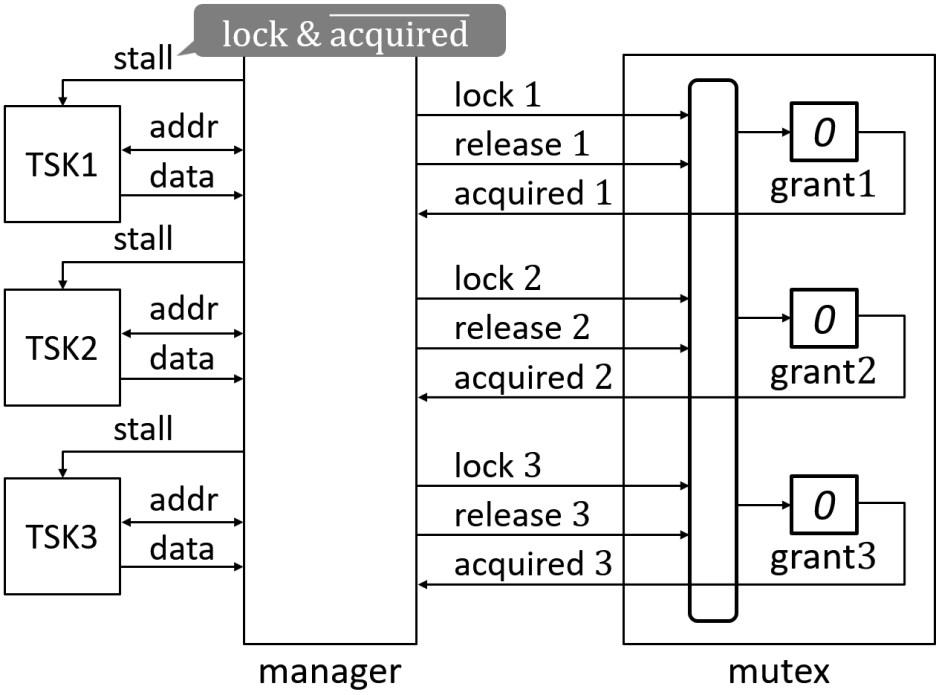

図 4.4: ハードウェアミューテックスの構造

4.4 ミューテックス

本手法では, ミューテックスをハードウェアモジュールとして実装する. 複数のタスク/ハンドラから同時にミューテックスの取得要求があった場合, ミューテックスモジュールはそのうちの 1 つのみを許可し, 他のタスク/ハンドラには stall を送る.

図 4.4 にタスク数 3 の場合のミューテックスのハードウェアモジュールの構造を示す. 右側がミューテックスモジュール, 中央が manager モジュール, 左がタスクモジュールである. タスク/ハンドラの数により, これらのモジュールが持つポートの数は変化する. ミューテックスモジュールは, 各タスク/ハンドラに対応したビット長の grant レジスタを内部に持ち, ミューテックスの取得状態をワンホットコードで保持する.

各ミューテックスは, 各タスク/ハンドラからの取得要求を受け取る lock ポート, 解放要求を受け取る release ポート, 取得状態を送る acquired ポートを持ち, manager を経由してタスクとの通信を行う.

ミューテックスの取得/解放処理の流れは以下の通りである. タスク/ハンドラは, ミューテックスの取得にはサービスコール loc_mtx, 解放には unl_mtx をそれぞれ用いる. ミューテックスを取得した場合, タスク側から, ミューテックスオブジェクト内のメンバ acquired に対応するアドレスに対して load を行うと 1 が返される.

1) (task/handler i) サービスコール loc_mtx を呼ぶ.

2) (manager) ミューテックスモジュールへの i 番目のロック要求信号 LCKi の値を 1 にする.

3) (mutex) i 番目のタスク/ハンドラへの取得成功信号 ACQi を 1 にする. 複数の取得要求が同時発生した場合, i の値が最も小さいものを選択する.

4) (manager) LCKj==1 and ACQj==0 である j 番目のタスク/ハンドラに対して stall の値を 1 にする.

5) (manager) i 番目のタスク/ハンドラからの, ミューテックスオブジェクトのメンバ変数 acquired に対する load 命令の結果として ACQi の値を返す.

6) (task/handler i) acquired==1 ならば, クリティカルセクションの実行を開始する.

7) (task/handler i) サービスコール unl_mtx を呼ぶ.

8) (manager) ミューテックスモジュールへの i 番目のロック解放信号 RELi の値を 1 にする.

9) (mutex) i 番目のタスク/ハンドラへの取得成功信号 ACQi を 0 にする.

第5章　実装と実験

タスク/ハンドラは, 高位合成によりレジスタ転送レベルのハードウェア記述に合成する. 一方, manager や arbiter 等のポートをシステム構成ごとに変化させる必要があるモジュール, および, これらのモジュールが持つレジスタの初期状態を定義したファイルは合成時に生成する. その他, mutex 等のモジュールは手設計する. タスク/ハンドラ, ミューテックスの数等は, 対象システムのコンフィギュレーションファイル中の定義から抽出する.

5.1　実装

本手法に基づく合成システムを, 高位合成システム ACAP[16] を用いて Perl 5 で実装した.

ACAP は, C プログラムもしくは MIPS R3000 の機械語プログラムを入力として, これを 32bit の整数演算命令を並列に実行可能なハードウェアに変換する. ACAP が行う合成処理の過程を, 図 5.1 に示す. まず, 入力となるプログラムは GCC を用いてコンパイル/リンクされる. 得られた MIPS の機械語プログラムを CDFG (control dataflow graph) に変換し, 最適化, スケジューリング, バインディングの処理を施して, Verilog HDL を生成する. これにより ACAP は, C プログラムの修正なしに, タスク/ハンドラを CPU と機能等価なハードウェアに合成できる. Vivado HLS[1] 等の高位合成システムの場合でも, ソースコードへ少しの修正を加えることにより同様に合成することが可能であると考えられる.

本論文で実装したシステムによる合成の流れを 図 5.2 に示す. まず, コンフィギュレーションファイル PROG.cfg から各種カーネルオブジェクトの構成を読み取り, C の定義ファイル kernel_cfg.c, kernel_cfg.h および status レジスタの構成を記述した kernel_cfg.vh を生成する. また, 各タスク/ハンドラのメイン関数を含む C ファイル (TASKi_m.c)[2] を生成する. これらのソースコードおよびアプリケーションプログラム (PROG1.c, ⋯, PROGk.c) から, ACAP を用いて Verilog HDL で記述されたハードウェア設計を生成する. これと同時に, manager/arbiter/mutex の Verilog HDL 記述を, システム構成に適したポート数で生成する.

現時点で, TOPPERS/ASP3 のサービスコール 178 個中 38 個を実装している. プログラムの規模は C 言語 3875 行, Perl 5 6512 行, Verilog HDL (および SystemVerilog) 2873 行, shell 277 行である.

[1] `https://www.xilinx.com/products/design-tools/vivado.html`
[2] ACAP は各ハードウェアモジュールに対して main 関数が必要である

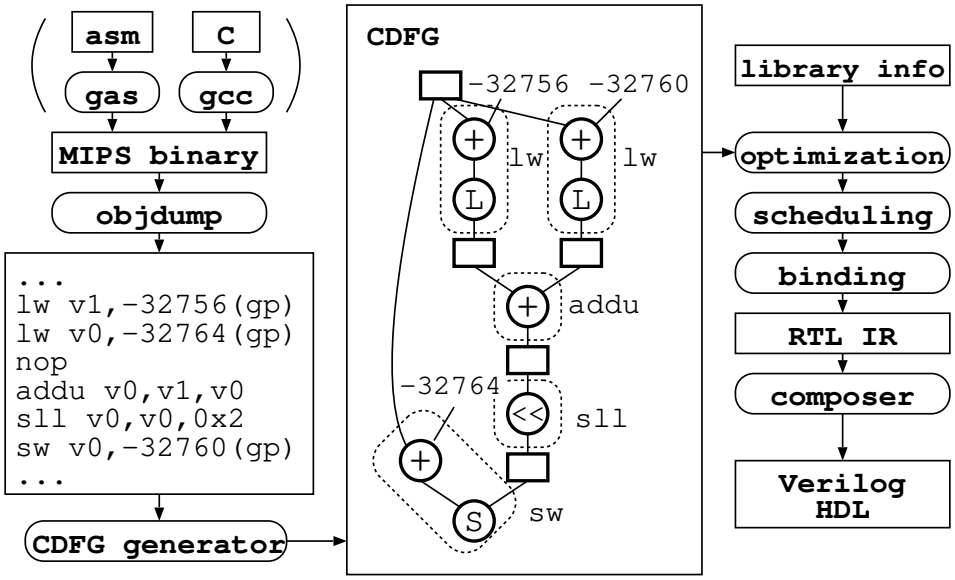

図 5.1: ACAP が行う合成処理過程 [16]

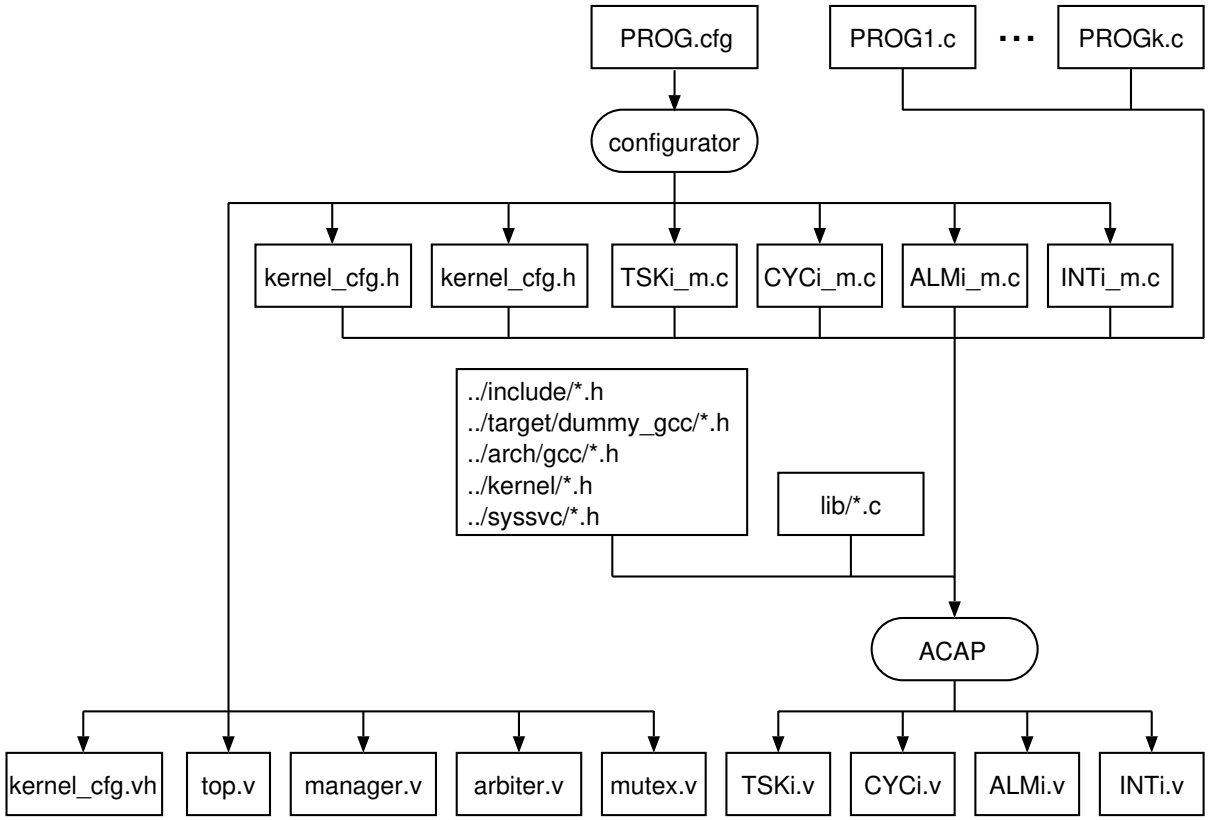

図 5.2: リアルタイムシステムの合成の流れ

表 5.1: `sample1` の合成結果

(a) 実行サイクル数と遅延

service call	#cycle	latency [ns]
act_tsk	23	301.3
wup_tsk	28	366.7
ext_tsk	12	157.2
ras_ter	26	340.5
ter_tsk	23	301.3
slp_tsk	16	209.6
loc_mtx	25	327.5
unl_mtx	10	131.0
sta_cyc	19	248.9
sta_alm	20	262.0
interrupt	1	13.1

(b) 回路規模

module	#LUT	#FF
top	96	1
manager	5,305	3,027
arbiter	573	10
mutex0	29	16
mutex1	29	16
MAIN_TASK	5,382	632
EXC_TASK	4,703	399
TASK1	5,320	950
TASK2	6,620	649
TASK3	6,258	649
ALMHDR1	8,252	850
CLCHDR1	6,714	852
INTNO1	5,669	639
total	54,950	8,689

High-level synthesizer: ACAP (2016.10)
Logic synthesizer: Xilinx Vivado (2016.4)
Target: Xilinx Artix-7 (xc7a100tcsg324-3)

5.2 実験結果

TOPPERS/ASP3 付属のサンプルプログラム “sample1” をハードウェア化した．このプログラムは, 全体を制御するタスク MAIN_TASK, 例外を処理するタスク EXC_TASK および 3 つの並行実行タスク TASK1, TASK2, TASK3 と, アラームハンドラ ALMHDR1, 周期ハンドラ CYCHDR1, 割込みハンドラ INTNO1 からなる．MAIN_TASK はシリアル通信からのメッセージを受けとり, 以下のサービスコールを実行する．

act_tsk, can_act, ter_tsk, chg_pri, get_pri, wup_tsk,

can_wup, rel_wai, sus_tsk, rsm_tsk, ras_ter, rot_rdq,

sta_cyc, stp_cyc, sta_alm, stp_alm, loc_cpu, unl_cpu

各並行実行タスクは, メモリに実行されたことを示すデータを書き込み, 周期ハンドラ, アラームハンドラ, 割込みハンドラはレディーキューを回転させる．EXC_TASK は, ログを残した後にシステムコール ext_ker を呼ぶことによりシステムを終了させる．

C プログラムからの MIPS アセンブリへのコンパイルは GCC 4.8.2 にて -O2 オプションを用いて行った．ACAP で生成された Verilog HDL は, Xilinx Vivado 2016.4 で RTL シミュレーションを行い, レスポンス性能を測定した．また, Xilinx FPGA Artix-7 (xc7a100tcsg324-3) をターゲットに論理合成した．

合成したハードウェアにおけるレスポンス性能を表 5.1 (a) に示す. #cycle, latency はそれぞれ, 各サービスコールが呼ばれてから状態が更新されるまでの実行サイクル数, 時間を表す. 例えば, 1 行目の act_tsk は, タスクを休止状態から実行可能状態に遷移させるまでに 23 サイクル必要である. (次のサイクルで実行可能状態から実行状態に遷移する.) このうち, 状態の更新に要するのは 2 サイクルであり, それ以外の 21 サイクルはエラーチェックやサービスコール実行のシリアライズのための排他制御処理を行っている. また, latency は実行サイクル数とクリティカルパス遅延 13.098 ns の積である. いずれのサービスコールも 400 ns 以内に実行することができた. また, 割込みハンドラの起動は 1 サイクルで行うことができた.

表 5.1 (b) は各ハードウェアモジュールの回路規模を示す. #LUT はルックアップテーブル数, #FF はフリップフロップ数である. 高位合成により生成したタスク/ハンドラの回路規模はやや大きい. これについては, ハードウェアの最適化の余地があると考えられる.

5.3 考察

5.3.1 回路規模

本論文で実装したハードウェアについて, 合計の LUT 数が 54,950, FF 数が 8,689 であるため, 回路規模はやや大きいといえる. 実機の FPGA にコンフィギュレーションする場合, ミドルクラスの FPGA デバイスが必要になる. MIPS R3000 互換のソフトコアプロセッサがおよそ 3200 LUT で実装できるため, 高位合成により生成したタスク/ハンドラの回路規模がその 1.5˜ 2 倍程度とやや大きい. これについては, サービスコールの C ライブラリの改善や, Vivado HLS をはじめとした, 他の汎用高位合成システムの使用等によるハードウェアの最適化の余地があると考えられる.

5.3.2 タスク/ハンドラ数と回路規模, 遅延の関係

今回手設計により作成した top, manager, arbiter, mutex について, タスク/ハンドラ数が変化した場合のモジュール毎の回路規模 (LUT 数, FF 数) および遅延についての考察を表 5.2 に示す. n はタスク/ハンドラの数であり, m はミューテックスの数である.

LUT 数について, オーダーで考えた場合は top の $O(nm)$, arbiter の $O(n^2)$, manager の $O(n^2 + nm)$ が支配的だが, top は全体の回路規模にはほとんど影響を与えず, ミューテックス数の変化による manager への影響も非常に小さい. そのため, 全体としての LUT 数は $O(n^2)$ に近似できると考えられる. なお manager については, 今回のように同時に呼ばれるサービスコールが 1 個であるという条件下においては $O(nm)$ での実装が可能だと考えられる.

フリップフロップ数については top では使用しておらず, その他のモジュールで $O(n)$ であるため, 全体でも $O(n)$ だと考えられる. また, 遅延についても全体で $O(n)$ となるが, 本論文で想定するタスク/ハンドラ数においては, タスク/ハンドラの遅延の方が大きくなり, クリティカルパス遅延は n, m の影響をほぼ受けないと考えられる.

表 5.2: タスク/ハンドラ数と回路規模, 遅延の関係

module	#LUT	#FF	latency
top	O(nm)	0	O(1)
manager	O($n^2 + mn$)	O(n)	O(n)
arbiter	O(n^2)	O(n)	O($\log n$)
mutex	O(n)	O(n)	O(n)

(n: タスク/ハンドラ数, m: ミューテックス数)

5.3.3 今後実装が必要なサービスコールおよび他の OS への対応

本論文では, RTOS の基本となるタスク/ハンドラの管理機能, API およびその排他制御で必要となるミューテックスのハードウェア実装について提案したが, RTOS には, その他にも様々な同期・

通信が存在する. イベントフラグ, データキュー, メッセージバッファ等を含めた, TOPPERS/ASP3 の他のサービスコールの実装は今後の課題である.

今回は, 国内で高いシェアを持つ TOPPERS を対象に実装を行ったが, 本手法は TOPPERS だけでなく, 他の OS への適用も可能な設計としている. 世界的に高い知名度を持つ FreeRTOS を始めとした他のカーネルへの適用も重要な課題である.

5.3.4 排他制御

本論文では, 3.1 節で述べたように, 入力となるプログラムは, 複数のタスク/ハンドラが同時に実行されないことを前提にした排他制御を行っていないことを前提とした. より広範なプログラムを合成可能にするには, このようなプログラムへの対応が必要となる. 解決手法としては, 最も高い優先度を持つタスク/ハンドラのみを並列動作させるモード, 単一のタスク/ハンドラのみが動作するモードを導入することが挙げられる.

第6章　結論

本論文では, RTOS を用いたシステムのフルハードウェア実装を自動生成する手法を提案した. RTOS 上で動作するアプリケーションプログラムを入力することにより, CPU およびソフトウェアに機能等価なハードウェアを自動生成することを可能にした. 全タスク/ハンドラを並列実行可能にすることにより応答性能を向上させ, また, スケジューラの軽量化を可能にした.

実験の結果, 本手法により生成されたハードウェアは, 応答性能が大きく向上していることを確認した. いずれのサービスコールも 400 ns 以下で実行することができ, 割込みハンドラの起動は約 13 ns で行うことができた.

本手法は, 元のソースコードをそのままに, フルハードウェア実装を自動合成することができるため, 今後, ますます複雑化していくリアルタイムシステムの開発において大きな意味があると考えられる.

現在, 生成したハードウェアの回路規模が大きいため, 高位合成の入力となるライブラリの改善, 生成されるハードウェアの更なる最適化が必要である. また, 汎用の高位合成システムを用いることも検討すべきである. 他の課題として, 複数のタスク/ハンドラが同時に実行されないことを前提にした排他制御を行っているプログラムからの合成が挙げられる. 解決手法として, 最も高い優先度を持つタスク/ハンドラのみを並列動作させるモード, 単一のタスク/ハンドラのみを動作させるモードを導入することが挙げられる. また, イベントフラグ, データキュー, メッセージバッファ等に関わる TOPPERS/ASP3 の他のサービスコールの実装も今後の課題である. さらに, 本手法の FreeRTOS[1] 等の他の RTOS への適用も重要な課題である.

[1] https://www.freertos.org

謝辞

本研究に際し, 多くの方々から御指導, 御支援を賜りました. ここに感謝の意を表します.

高位合成の研究に携わる機会を与えていただき, 研究に関する多くの御指導, 御支援を賜りました石浦菜岐佐教授に心より感謝いたします.

本研究に関する有益な議論の場を提供して頂き, 様々な視点から御指導いただきました京都高度技術研究所の神原弘之氏に感謝いたします. 本分野に対する高い見識を持ち, 様々な面で御助力いただきました立命館大学の冨山宏之教授に感謝いたします. 技術的な面で多くの御助言をいただきました元立命館大学の中谷嵩之氏, 元関西学院大学の田村真平氏に感謝いたします.

本研究に関してご協力, ご討議下さいました HLS チームの諸氏に深く感謝いたします.

最後に, 研究のみならず生活面においても多くの御支援をいただいた関西学院大学理工学部石浦研究室の皆様に深く感謝いたします.

関連発表文献

1) 大迫 裕樹, 石浦 菜岐佐, 神原 弘之, 冨山 宏之: “RTOS を用いたシステムのフルハードウェア実装とその自動化,” 電子情報通信学会技術研究報告, (Mar. 2019, to appear).

2) Yuuki Oosako, Nagisa Ishiura, Hiroyuki Tomiyama, and Hiroyuki Kanbara: “Synthesis of Full Hardware Implementation of RTOS-Based Systems,” in *Proc. International Symposium on Rapid System Prototyping (RSP 2018)*, pp. 1–7 (Oct. 2018).

3) Naoya Ito, Yuuki Oosako, Nagisa Ishiura, Hiroyuki Tomiyama, and Hiroyuki Kanbara: “Binary Synthesis Implementing External Interrupt Handler as Independent Module,” in *Proc. International Symposium on Rapid System Prototyping (RSP 2017)*, pp. 92–98 (Oct. 2017).

4) 大迫 裕樹, 石浦 菜岐佐, 神原 弘之, 冨山 宏之: “RTOS を用いたシステムの高位合成によるフルハードウェア化,” 組込みシステム技術に関するサマーワークショップ (SWEST), RS–6, (Aug. 2017).

5) Nagisa Ishiura and Yuuki Oosako: “Introducing Real Constraints in Partitioned ILP-Based Biding in High-Level Synthesis” (short paper), in *Proc. the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2016)*, R4–5, pp. 303–304 (Oct. 2016).

6) 大迫 裕樹, 神原 弘之, 石浦 菜岐佐: “割り込み駆動のモータ制御プログラムからの高位合成,” 情報処理学会関西支部大会, A–01, (Sept. 2016).

7) 大迫 裕樹, 石浦 菜岐佐: “高位合成のバインディングの整数線形計画法による分割解法における実数制約の導入,” 電子情報通信学会ソサイエティ大会, A–6–7, (Sept. 2016).

参考文献

[1] Youngchul Cho, Sungjoo Yoo, Kiyoung Choi, Nacer-Eddine Zergainoh, and Ahmed A. Jerraya: "Scheduler implementation in MPSoC design," in *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC 2005)*, pp. 151–156 (Jan. 2005). DOI:http//doi.org/10.1109/ASPDAC.2005.1466148

[2] Melissa Vetromille, Luciano Ost, Csar A. M. Marcon, Carlos Reif, and Fabiano Hessel: "RTOS scheduler implementation in hardware and software for real time applications," in *Proc. International Workshop on Rapid System Prototyping (RSP '06)* pp. 163–168 (June 2006). DOI:http//doi.org/10.1109/RSP.2006.34

[3] Paul Kohout, Brinda Ganesh, and Bruce Jacob: "Hardware support for real-time operating systems," in *Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*, pp. 45–51 (Oct. 2003). DOI:http//doi.org/10.1145/944645.944656

[4] Takumi Nakano, Yoshiki Komatsudaira, Akichika Shiomi, and Masaharu Imai: "Performance evaluation of STRON: A hardware implementation of a real-time OS," in IEICE Trans. Fundamentals, vol. E82-A, no. 11, pp. 2375–2382 (Nov. 1999). DOI:http//doi.org/

[5] Naotaka Maruyama, Tohru Ishihara, and Hiroto Yasuura: "An RTOS in hardware for energy efficient software-based TCP/IP processing," in *Proc. IEEE Symposium on Application Specific Processors (SASP 2010)*, pp. 58–63 (June 2010).

[6] Renesas Electronics Corp.: "Issues with Real Time Performance in Conventional RTOS and Performance Improvements through HW-RTOS," (Sep. 2018). available at https://www.renesas.com/kr/en/doc/DocumentServer/009/r70wp0002ej0100-rtos.pdf (accessed 2019-01-30).

[7] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[8] Seiya Shibata, Shinya Honda, Hiroyuki Tomiyama, and Hiroaki Takada: "Advanced system-builder: A tool set for multiprocessor design space exploration," in *Proc. International SoC Design Conference (ISOCC 2010)*, pp. 79–82 (Nov. 2010). DOI:http//doi.org/10.1109/SOCDC.2010.5682967

[9] Yuki Ando, Shinya Honda, Hiroaki Takada, and Masato Edahiro: "System-level design method for control systems with hardware-implemented interrupt handler," *IPSJ Journal of Information Processing*, vol. 23, no. 5, pp. 532–541 (Sept. 2015). DOI:`http//doi.org/10.2197/ipsjjip.23.532`

[10] Naoya Ito, Nagisa Ishiura, Hiroyuki Tomiyama, and Hiroyuki Kanbara: "High-level synthesis from programs with external interrupt handling," in *Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2015)*, 10–15 (March 2015).

[11] Naoya Ito, Yuuki Oosako, Nagisa Ishiura, and Hiroyuki Tomiyama, and Hiroyuki Kanbara: "Binary synthesis implementing external interrupt handler as independent module," in *Proc. International Symposium on Rapid System Prototyping (RSP 2017)*, pp. 92–98 (Oct. 2017). DOI:`http//doi.org/10.1145/3130265.3130317`

[12] Yuuki Oosako, Nagisa Ishiura, Hiroyuki Tomiyama, and Hiroyuki Kanbara: "Synthesis of Full Hardware Implementation of RTOS-Based Systems," in *Proc. RSP 2018*, pp. 1–7 (Oct. 2018).

[13] 大迫 裕樹, 石浦 菜岐佐, 神原 弘之, 冨山 宏之: "RTOS を用いたシステムのフルハードウェア実装とその自動化," 電子情報通信学会技術研究報告, (Mar. 2019, to appear).

[14] ITRON Committee, TRON association: *μITRON4.0 Specification* (1999, 2002). Available at `http://www.ertl.jp/ITRON/SPEC/FILE/mitron-400e.pdf` (accessed 2018-06-11).

[15] TOPPERS project: "TOPPERS 第 3 世代カーネル (ITRON 系) 統合仕様書 Release 3.2.1," `https://www.toppers.jp/docs/tech/tgki_spec-321.pdf` (accessed 2019-01-05).

[16] Nagisa Ishiura, Hiroyuki Kanbara, and Hiroyuki Tomiyama: "ACAP: Binary Synthesizer Based on MIPS Object Codes," in *Proc. ITC-CSCC 2014*, pp. 725–728 (July 2014).