一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2021-51,CPSY2021-20,RECONF2021-59(2022-01)

RTOS 利用システムの汎用高位合成系を用いたフルハードウェア化

安堂　拓也[†]　石井　雄吾[†]　石浦菜岐佐[†]　冨山　宏之[††]　神原　弘之[†††]

† 関西学院大学　〒669–1337 兵庫県三田市学園 2–1
†† 立命館大学　〒 525–8577 滋賀県草津市野路東 1 丁目 1–1
††† 京都高度技術研究所　〒 600–8813 京都市下京区中堂寺南町 134 番地

あらまし　本稿では, RTOS を用いたシステムのフルハードウェア実装を汎用的な高位合成システムによって行う手法を提案する. 六車らは, タスク/ハンドラおよび RTOS のカーネル機能を全てハードウェア化することによりリアルタイムシステムの応答性能を飛躍的に向上させる手法を提案しているが, 独自のバイナリ合成システムに依存しており, 汎用的な高位合成システムではタスクの実行制御や共有変数へのアクセスをそのまま合成することが困難であった. 本稿では, タスクの実行を実行/停止信号ではなく, タスクからのサービス要求の実行/保留により制御する方式と, メモリアクセスのラッパークラスを定義して最小限の書き換えで共有変数へのアクセスを可能にする方法により, 一般的な高位合成システムで RTOS 利用システムのフルハードウェア実装を可能にする. 本手法を TOPPERS/ASP3 カーネル付属サンプル “sample1” を縮小したプログラムに適用した結果, Xilinx Vitis HLS を用いてハードウェアを合成することができた. また, これにより従来手法に比べて回路規模を大幅に削減することができた.

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

Full Hardware Implementation of RTOS-Based Systems Using General-Purpose High-Level Synthesizer

Takuya ANDO[†], Yugo ISHII[†], Nagisa ISHIURA[†], Hiroyuki TOMIYAMA[††], and Hiroyuki KANBARA[†††]

† Kwansei Gakuin University,　Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan
†† Ritsumeikan University, 1–1–1 Noji–Higashi, Kusatsu, Shiga, 525–8577, Japan
††† ASTEM RI/KYOTO, 134 Chudoji Minamimachi, Shimogyo–ku, Kyoto, 600–8813, Japan

Abstract　This article proposes a method for implementing a whole RTOS-based system as hardware using general-purpose high-level synthesizer. Muguruma has proposed a scheme where both all the tasks/handlers and all the RTOS functions are implemented as hardware. However, it assumes the use of a dedicated binary synthesizer, ACAP, where generated task modules have *stall* ports for suspending their execution and accesses to globally shared variables are realized as loads/stores using automatically generated addresses, which are not necessarily possible by general high-level synthesizers. This paper proposes a method where execution of tasks is controlled by allowing/disabling execution of service calls from the tasks, and code transformation using a wrapper class for shared variable accesses and functions within a function, to make general high-level synthesizers applicable to the full-hardware scheme. Based on the proposed methods, a hardware module for a reduced version of “sample1” bundled with TOPPERS/ASP has been successfully implemented as hardware using Xilinx Vitis HLS, where the size of the resulting circuit was substantially smaller than that by the previous method.

Key words　Real-Time Systems, RTOS, System Synthesis, Hardware Accelerator, TOPPERS/ASP3, High-Level Synthesis

1. は じ め に

情報通信技術の発展に伴い, 組み込みシステムには新しいサービスを提供するために益々高い機能が要求されるようになっている. 特に, 車載機器やロボットの制御には高い機能とともに高い応答性能が要求される. このようなシステムの開発はリアルタイム OS (RTOS) を用いて行われる. RTOS は, 予め定められた時間内にタスクの実行が完了できるようなシステムを設計す

This article is a technical report without peer review, and its polished and/or extended version may be published elsewhere.
Copyright ©2022 by IEICE

るための機能を提供する. しかし, システムの高機能化が進むにつれてその応答性能の確保は難しくなってきている.

RTOS を用いたシステムの応答性能を向上させる手法として, RTOS 機能の一部または全てをハードウェア実装する方法がある. 文献 [1] [2] [3] では RTOS のスケジューラのハードウェア化による高速化を行っており, 文献 [4] [5] では RTOS のほとんどの機能をハードウェア実装している. しかし, これらの手法ではタスクやハンドラはソフトウェアで実装されており, CPU 待ちやコンテキストスイッチによるオーバーヘッドが発生する. 一方, タスクやハンドラを高位合成技術 [6] を利用してハードウェアに合成することにより応答性能の向上を図る手法 [7] [8] が提案されているが, これらの手法では, RTOS および一部のタスクはソフトウェアとして実行される. また, 文献 [9] [10] は割込みハンドラを含むシステム全体のハードウェア化を行っているが, 対象はベアメタルシステムに限られる.

この課題を解決する一手法として, 文献 [11] はタスク/ハンドラおよび RTOS のサービス機能全てをハードウェア化する手法を提案している. タスクはそれぞれ独立に動作するハードウェアに合成され, 実行可能状態のタスクは全て並列に実行できるため, CPU 待ちやタスク切り替えのオーバーヘッドが無い. TOPPERS/ASP3 および FreeRTOS を対象とした実装 [12] [13] [14] では高い応答性能を実現しているが, 回路規模に課題があった. 文献 [15] では, 文献 [11] のハードウェア構成において, 各タスク内部に重複して実装されている RTOS のサービス処理機能をタスク管理ハードウェア側に集約することによって, 回路規模の削減を試みている.

しかし, これらの手法では独自のバイナリ合成システム ACAP [16] に依存した実装を行っているため, 汎用的な高位合成システムを用いてタスクの実行制御や共有変数へのアクセスをそのまま合成することが困難である.

本稿ではこの課題を解決する手法として, 汎用的な高位合成システムを用いて RTOS 利用システムをフルハードウェア化する手法を提案する. 本手法では, タスクの実行制御を実行/停止信号を用いて行うのではなく, タスクからのサービス要求の実行/保留により行うようにする. また, 共有メモリアクセスのラッパークラスを定義することにより最小限の書き換えで一般的な高位合成システムを用いてタスクをハードウェア化できるようにする.

本手法を TOPPERS/ASP3 カーネル付属サンプル “sample1” を縮小したプログラムに適用した結果, Xilinx Vitis HLS を用いてハードウェアを合成することができた. また, これにより従来手法に比べて回路規模を大幅に削減することができた.

2. RTOS を用いたシステムのフルハードウェア実装

2.1 概　　念

文献 [11] [15] では, RTOS 機能を利用したプログラムを入力とし, これを実行するプロセッサと機能等価なハードウェアを合成する手法を提案している. この手法の概念を 図 1 に示す. TSKi は入力となるタスクプログラムであり, 高位合成によって独立したハードウェアモジュールとして実装される. また, マネージャ (manager) は RTOS 機能をハードウェア化したものである.

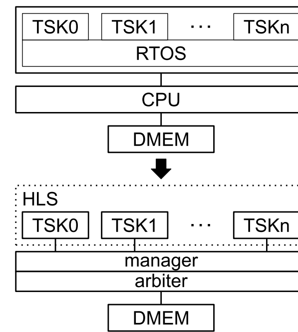


図 1: RTOS を用いたシステムのフルハードウェア実装

実行可能状態になったタスクは全て並列に動作する. マネージャは各タスクの状態変数の値を用いて実行/停止を制御する信号を生成し, これらの実行制御を行う. 複数のタスクがデータメモリ (DMEM) へ同時にアクセスした場合は, 優先度を用いてアービタ (arbiter) モジュールが調停を行う.

この手法では, タスクはハードウェア化されている上, 独立に並列実行されるため, CPU 待ちやコンテキストスイッチによるオーバーヘッドがなく, 従来手法に比べてシステムの応答性能を大幅に向上させることができる.

2.2 文献 [15] のアーキテクチャ

文献 [15] で合成されるハードウェアの構成を 図 2 に示す. TSKi はタスクをハードウェア化したものである. タスクの実行制御はマネージャが各タスクの状態に基づいて実行を停止する制御信号 stall を出すことにより行う. 即ち, タスクが実行状態にあれば stall 信号を 0 にし, それ以外の場合には stall を 1 にしてタスクの実行を一時停止する. なお, タスクが実行可能状態になるとマネージャは次のサイクルにこれを実行状態に更新する. Sj はサービスモジュールでありタスクの状態変更, ミューテックスロック, データキュー等 RTOS が提供するサービスを実行する.

タスクは TiF にサービス番号を, TiA に引数を書き込むことにより, マネージャにサービスを要求する. これらのレジスタはアドレス空間に配置されており, addr と data のポートを使ってタスクからアクセスできる. サービス間の競合を避けるため, サービスは一度に一つずつ実行される. 複数のタスクがサービスを要求している場合には, 調停回路 (Request Arbiter) がタスクの優先度に基づいて調停を行い, 実行するサービスを決定する. サービスの返り値等の結果値は TiA に書き込まれ, タスクはこの値を受信して実行を再開する.

2.3 ACAP と汎用高位合成系

文献 [11] [15] が用いているバイナリ合成ツール ACAP は, MIPS の機械語プログラムを入力として, これを高位合成と同様の手法によってハードウェア記述言語に変換するものである. C や C++ で書かれたプログラムは MIPS 用コンパイラでコンパイルして得られる機械語を経て合成できる他, アセンブリやイン

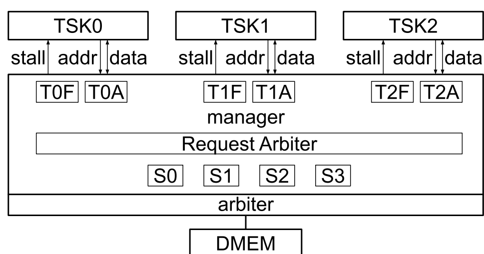


図 2: 文献 [15] のハードウェア構成

ラインアセンブリによるプログラムも合成可能である. また, 機械語を入力とするため, プログラム中のメモリアクセスをそのままハードウェア化することが可能であり, グローバル変数やポインタを介したデータの読み書きを含むプログラムをソースコードの修正なしに合成できる.

文献 [11] [15] の実装は ACAP に依存しているが, タスクやハンドラはほとんどの場合 C や C++ 言語で書かれるため, 必ずしも機械語からの合成の必要はなく, 一般的な高位合成系を利用できることが望ましい.

文献 [15] の実装を汎用の高位合成系で行うためには, 二つの課題を解決する必要がある.

一つは, ACAP の合成する回路は stall ポートを有しており, これによって回路の実行停止を制御している点である. 汎用の高位合成系は必ずしもこのようなポートを生成しない. また, 外部からの信号によって任意の時点で実行を停止するような動作を C 言語レベルで記述することはできない.

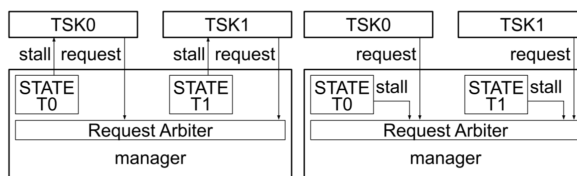
もう一つは複数のタスクが共有する変数 (グローバル変数) へのアクセスである. ACAP では機械語にある lw/sw のようなメモリアクセス命令の動作がそのままハードウェア化されるため, 多くの共有変数があってもそれらをアドレス空間に配置して自然な方法でアクセスできる. 汎用の高位合成系でもアドレスポートを生成して同様のアクセスを実現することはできるが, そのためにはユーザプログラム (タスク) の書き換えが必要になる.

3. 汎用高位合成系を用いたフルハードウェア化

3.1 概　　要

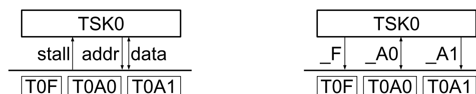
本稿では, リアルタイムシステムのフルハードウェア実装を汎用高位合成系を用いて行う手法を提案する. 従来手法と同様, RTOS 機能を利用したプログラムを入力とし, これを実行する CPU と機能等価なハードウェアを生成する. RTOS の機能を提供するマネージャは RTL で設計し, タスクモジュールを高位合成により生成する.

従来手法では実行を一時停止するための信号 (stall) を使ってタスクの実行制御を行っていたのに対し, 本手法ではタスクのサービス待ちを利用することによって, stall 信号を使わずにタスクの実行制御を行う. 制御レジスタへのアクセスはメモリマップド I/O ではなく, ポートを介して直接行うようにする. また, メモリアクセス用のラッパークラスや関数内関数を用いることにより, ソースコードの書き換え量を最小限にする.



(a) 文献 [15]　　(b) 本手法

図 3: stall 信号によるタスクの実行制御



(a) 文献 [15]　　(b) 本手法

図 4: タスク/マネージャ間のインタフェース

3.2 タスクの実行制御

本手法では, タスク T がサービスを要求した際に,

> T が実行状態でなければサービスを実行せず
> タスクを待たせる

ことにより, stall 信号によるタスクの実行停止と等価な機能を実現する. ここでマネージャの提供するサービスには共有メモリへのアクセスも含める. この方法では, 実行状態でないタスクもローカルには動作することになる. しかし, 実行状態でないタスクからのサービスコールと共有メモリアクセスが保留されるのであれば, システムの全体としての動作の意味は変わらない.

従来手法では 図 3 (a) のように各タスクの状態から生成した stall 信号をタスクに送信していたが, 本手法では同じ stall 信号を 図 3 (b) のようにタスクからのサービスをブロックするのに用いる. 即ち, タスクがサービス要求をして, そのタスクが実行状態でなければ RA が要求を受けつけないようにする. その結果, タスクは再び実行可能状態になるまで実行されなくなる.

3.3 制御レジスタへのアクセスとサービスコールのスタブ

本手法では, 制御レジスタをメモリ空間に配置してメモリアクセスにより読み書きを行うのではなく, 各制御レジスタに対して直接アクセスするポートを作成して制御レジスタの読み書きを行う. これにより, 複数の制御レジスタアクセスが必要な場合のサイクル数が削減できる.

従来手法と本手法のタスク/マネージャ間のインタフェースをそれぞれ 図 4 (a) (b) に示す. _F は T*i*F レジスタに要求するサービスの ID を書き込むためのポートであり, _A*j* は T*i*A*j* レジスタに引数を書き込んだり, サービスの結果値を T*i*A*j* レジスタから読み出すためのポートである. _A*j* ポートはシステムで使用しているサービスに必要な最大数に合わせて用意する. 前節で述べた通り本手法では stall 信号を廃している.

本手法のインタフェースに基づくサービス呼び出しのためのコード記述例を 図 5 に示す. (a) はタスクからのサービス呼び出しであり, この例の chg_pri(tskid, tskpri) は ID が tskid であるタスクの優先度を tskpri に変更するものである. (b) はサービスコールの本体, 即ちマネージャの制御レジスタを読み書きするた

```
{
  ...
  chg_pri(TSK1, LOW_PRI);
  ...
}
```

(a) タスクからのサービスコール

```
#define chg_pri(tskid, tskpri) \
       _chg_pri(tskid, tskpri, _F, _A0, _A1)

ER _chg_pri(ID tskid,
            PRI tskpri,
            volatile int* const _F,
            volatile int* const _A0,
            volatile int* const _A1){
  *_A0 = tskid;
  *_A1 = tskpri;
  ap_wait(); // 次サイクルに以下を実行
  *_F = SERV_CTRL_TSK | METHOD_CHG_PRI;
  ap_wait(); // 次サイクルに以下を実行
  return *_A0;
}
```

(b) サービスコールの本体 (スタブ)

図 5: タスクおよびサービスコールの記述例

```
ER M_READ_int (int index,
               volatile int* const _F,
               volatile int* const _A0){
  *_A0 = index;
  ap_wait(); // 次サイクルに以下を実行
  *_F = SERV_GRW | METHOD_READ;
  ap_wait(); // 次サイクルに以下を実行
  return *_A0;
}
```

(a) M_READ_int

```
ER M_WRITE_int (int index,
                int value,
                volatile int* const _F,
                volatile int* const _A0,
                volatile int* const _A1){
  *_A0 = index;
  *_A1 = value;
  ap_wait(); // 次サイクルに以下を実行
  *_F = SERV_GRW | METHOD_WRITE;
  ap_wait(); // 次サイクルに以下を実行
  return *_A0; // 書き込み処理の完了通知
}
```

(b) M_WRITE_int

図 6: 共有変数アクセスサービスの記述例

めのスタブである. 1–2 行目の define 文によって制御レジスタにアクセスするポートへのポインタを追加している. 9–10 行目で 2 つの引数を _A0, _A1 に出力している. 12 行目で _F にサービス要求 (サービスモジュールを指定する SERV_CTRL_TSK と処理を指定する METHOD_CHG_PRI を連結したもの) を書き込むとマネージャがその処理を開始する. 11 行目の ap_wait 文は, 引数を書き込む前に実行が始まるのを防ぐために _F への書き込みを 1 クロック後に行うことを指定するものである. マネージャは実行結果 (返り値) を _A0 に書き込むので, それを待ってその値を 14 行目でタスクに返す.

本手法では共有メモリの読み出しおよび書き込みもサービスコールと同様の枠組で行う. 例えば, int 型の共有データアクセスに対しては M_READ_int, M_WRITE_int というアクセス関数を用意する. M_READ_int(addr) は共有メモリから addr に対応するデータを読み出して返すものであり, M_WRITE_int(addr, data) は共有メモリの addr に対応する場所に data を書き込むものである.

これらの関数本体の記述例をそれぞれ 図 6 (a) (b) に示す. いずれも引数を出力した後にサービス番号を出力している. M_WRITE_int 関数の 11 行目にある return 文は実行状態でないタスクのサービス要求を待たせるためのものである.

これらの共有メモリアクセス関数を用いると, 図 7 (a) のタスク中のグローバル変数アクセスは 図 7 (b) のように書き換えられる. 上方の X_ADDRESS, Y_ADDRESS は, 各共有変数のアドレス値である. 元プログラムの `x = 1;` は下方の 3 行目のようになり, `y = x + 2;` は 4 行目のようになる.

このような書き換えを軽減するため, 本手法では, 共有変数アクセスのラッパークラスを定義する. ラッパークラスを用いた場合のタスクの記述例を 図 7 (c) に示す. 2–3 行目は x と y を G_int クラスのインスタンスとして宣言したものであり, これによって 5–6 行目のようにグローバル変数のアクセスは元のプログラムから書き換えることなく扱うことができる. G_int は 図

```
int x;
int y;

{
  ...
  x = 1;
  y = x + 2;
  ...
}
```

(a) タスクの元プログラム

```
#define X_ADDRESS 0x80000000
#define Y_ADDRESS 0x80000004
```

```
{
  ...
  M_WRITE_int(X_ADDRESS, 1);
  M_WRITE_int(Y_ADDRESS, M_READ_int(X_ADDRESS) + 2);
  ...
}
```

(b) 書き換え後のプログラム

```
{
  G_int x(_F, _A0, _A1);
  G_int y(_F, _A0, _A1);
  ...
  x = 1;
  y = x + 2;
  ...
}
```

(c) ラッパークラスを利用したプログラム

図 7: タスクの共有変数アクセスの記述例

8 のように定義できる.

3.4 合成するタスクプログラムの構成

一般に 1 つのタスクプログラムは複数の関数から構成される. そのため, 制御レジスタにアクセスするための外部ポートや共有変数アクセス用のオブジェクトをこれらの関数間で共有する必要がある. 本手法では, タスクを構成する各関数をタスクモジュールに合成される関数の関数内関数にすることにより, これらの共有を可能にする.

例えば, 図 9 (a) に示すようなタスクプログラムが与えられ

```
static int addr = 0x80000000;

class G_int{
  const int address;
  volatile int* const _F;
  volatile int* const _A0;
  volatile int* const _A1;

public:
  G_int(volatile int* const f, volatile int* const a0,
        volatile int* const a1)
         : address(addr), _F(f), _A0(a0), _A1(a1) {addr += 4;}

  operator int () { return M_READ_int(address, _F, _A0); }

  G_int& operator = (int value) {
    M_WRITE_int(address, value, _F, _A0, _A1);
    return *this;
  }

  G_int& operator = (G_int& x) {
    *this = (int) x;
    return *this;
  }
};
```

図 8: 共有変数アクセス用ラッパークラス

```






int x;
int y;

void sub(){
  x = x + 3;
  y = y + x;
}

void tsk(){
  x = 1;
  y = x + 2;
  chg_tsk(TSK1, LOW_PRI);
  sub();
}


//
```

(a) タスクの元プログラム

```
void tsk_main(
  volatile int* const _F,
  volatile int* const _A0,
  volatile int* const _A1
  ){

  G_int x(_F, _A0, _A1);
  G_int y(_F, _A0, _A1);

  auto sub = [=]() mutable {
    x = x + 3;
    y = y + x;
  };

  auto tsk = [=]() mutable {
    x = 1;
    y = x + 2;
    chg_tsk(TSK1, LOW_PRI);
    sub();
  };

  tsk();
}
```

(b) 高位合成の入力となる記述

図 9: 複数の関数から成るタスクの変換例

たとする. グローバル変数 x, y と関数 sub, tsk を宣言しており, 関数 tsk がこのタスクの本体であるとする. このプログラムは 図 9 (b) のようなコードに変換して高位合成系の入力とする. 1–23 行目でタスクモジュールに対応する関数 tsk_main を新たに宣言し, 元プログラム全体を覆う. 2–4 行目では作成した関数の引数に外部ポート用の変数を記述し, 7–8 行目でクラスインスタンス x, y を宣言している. そして, 10–20 行目で sub 関数と tsk 関数を関数内関数として記述している.

これにより, _F, _A0, _A1, x, および y は sub および tsk からグローバル変数のように扱うことができる. 関数 sub および tsk の本体はそのまま用いることができるので, この変換の自動化は容易に行える.

4. 実装と実験

本手法に基づき 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 つの並行タスク TASK1, TASK2, TASK3 からなる. MAIN_TASK はシリアル通信から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,
loc_cpu, unl_cpu

ただし, アラームハンドラ, サイクルハンドラ, 割り込みハンドラは未実装であり, 関連するサービスコールの呼び出しは削除している.

合成した各モジュールの回路規模を表 1 (a) に示す. マネージャおよびサービスモジュールは Verilog HDL で設計し, Xilinx Vivado 2020.2 で Xilinx FPGA Artix-7 (xc7a100tcsg324-3) をターゲットに論理合成した. タスクモジュールは, 本手法では高位合成系 Xilinx Vitis HLS 2020.2 により合成し, 文献 [15] のシステムはバイナリ合成システム ACAP [16] により合成した. #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.

arbiter モジュールは文献 [15] では外部メモリアクセスの調停に用いていたが, 本手法ではメモリアクセスをサービスに統合したことにより arbiter モジュールを廃している. serv_grw モジュールは本手法で新たに設計した共有変数アクセスのサービスモジュールである. manager モジュールの回路規模減少は, タスクモジュールとのインタフェースの変更や arbiter モジュールとのインタフェースを廃したことによるものと考えられる.

高位合成により生成する TASK1, TASK2, TASK3, MAIN_TASK, EXC_TASK の回路規模は文献 [15] に比べて大幅に削減できている. これは主に高位合成系のツール性能の差によるものと考えられる.

合成したシステムのクリティカルパス遅延を表 1 (b) に示す. 回路規模と同様に, 文献 [15] に比べて遅延も削減できている.

合成したシステムの応答性能を表 2 に示す. #cycle は, タスクが各サービスコールの要求処理を開始してから返り値を受信するまでの実行サイクル数を表す. 例えば, 1 行目の act_tsk は, タスクが TA レジスタに引数を書き込み, 対象タスクを休止状態から実行可能状態に遷移させ, サービス完了通知を受信するまでのサイクル数であり, 文献 [15] に比べて 1 サイクル削減できている. サービス処理の内容は変更していないので, 制御レジスタへのアクセス方法を変更した影響であると考えられる. また, latency は実行サイクル数とクリティカルパス遅延の積であり. いずれのサービスコールも 150 ns 以内に実行することができている.

5. むすび

本稿では, 汎用的な高位合成系を用いて, RTOS を利用したシステムをフルハードウェア実装する手法を提案した. 新たなタスクの実行制御方式, およびメモリアクセス用ラッパークラスや関数内関数定義を利用したプログラム変換方法を提案した. 本手法に基づき, テストプログラムが高位合成システム Xilinx Vitis HLS を用いてハードウェア実装できることを確認した. また, これにより文献 [15] に比べてシステム全体の回路規模を大幅に削減することができた.

表 1: `sample1` の合成結果

(a) 回路規模

	文献 [15] (ACAP)		本研究 (Vitis HLS)	
module	#LUT	#FF	#LUT	#FF
top	0	5	0	0
arbiter	323	5	-	-
serv_grw	-	-	300	1,025
serv_ctrl_tsk	815	141	990	146
manager	3,592	3,871	2,394	3,041
TASK1	5,327	973	103	218
TASK2	5,988	901	104	219
TASK3	5,729	937	104	219
MAIN_TASK	5,331	963	323	559
EXC_TASK	5,883	967	8	8
total	32,988	8,763	4,326	5,435

(b) クリティカルパス遅延 [ns]

文献 [15]	本研究
13.407	9.783

High-level synthesizer: ACAP (2016.10), Xilinx Vitis HLS (2020.2)
Logic synthesizer: Xilinx Vivado (2020.2)
Target: Xilinx Artix-7 (xc7a100tcsg324-3)

表 2: サービスコールの実行サイクル数と遅延

	文献 [15]		本研究	
service call	#cycle	latency [ns]	#cycle	latency [ns]
act_tsk	11	147.477	10	97.830
ext_tsk	11	147.477	10	97.830
slp_tsk	16	214.512	15	146.745
dly_tsk	14	187.698	12	117.396
chg_pri	11	147.477	9	88.047
get_tim	11	147.477	9	88.047

現在, TOPPERS/ASP3 を主なターゲットとして RTOS のサービス機能のハードウェア設計を進めているが, FreeRTOS にも対応していくことが課題として挙げられる. また, 現在は手動でタスクのプログラムの変換とマネージャ等のハードウェアモジュールの設計を行っているため, これらの自動化も今後の課題である.

謝　辞

本研究に関して有益な御助言を頂い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, およびご支援を頂いた関西学院大学石浦研究室の諸氏に感謝致します. 本研究は一部 JSPS 科研費 19H04081, 20H00590, および 21K19776 の助成による.

文　献

[1] Y. Cho, S. Yoo, K. Choi, N-E Zergainoh, and A. A. Jerraya: "Scheduler implementation in MPSoC design," in *Proc. ASP-DAC 2005*, pp. 151–156 (Jan. 2005)

[2] M. Vetromille, L. Ost, C. A. M. Marcon, C. Reif, and F. Hessel: "RTOS scheduler implementation in hardware and software for real time applications," in *Proc. RSP '06* pp. 163–168 (June 2006).

[3] P. Kohout, B. Ganesh, and B. Jacob: "Hardware support for real-time operating systems," in *Proc. CODES+ISSS '03*, pp. 45–51 (Oct. 2003). DOI:`http//doi.org/10.1145/944645.944656`

[4] T. Nakano, Y. Komatsudaira, A. Shiomi, and M. 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).

[5] N. Maruyama, T. Ishihara, and H. Yasuura: "An RTOS in hardware for energy efficient software-based TCP/IP processing," in *Proc. SASP 2010*, pp. 58–63 (June 2010).

[6] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[7] S. Shibata, S. Honda, H. Tomiyama, and H. Takada: "Advanced system-builder: A tool set for multiprocessor design space exploration," in *Proc. ISOCC 2010*, pp. 79–82 (Nov. 2010).

[8] Y. Ando, S. Honda, H. Takada, M. 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).

[9] N. Ito, N. Ishiura, H. Tomiyama, and H. Kanbara: "High-level synthesis from programs with external interrupt handling," in *Proc. SASIMI 2015*, pp. 10–15 (March 2015).

[10] N. Ito, Y. Oosako, N. Ishiura, and H. Tomiyama, and H. Kanbara: "Binary synthesis implementing external interrupt handler as independent module," in *Proc. RSP 2017*, pp. 92–98 (Oct. 2017).

[11] Y. Oosako, N. Ishiura, H. Tomiyama, and H. Kanbara: "Synthesis of full hardware implementation of RTOS-based systems," in *Proc. RSP 2018*, pp. 1–7 (Oct. 2018).

[12] 大迫 裕樹, 石浦 菜岐佐, 冨山 宏之, 神原 弘之: "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," 信学技報, VLD2018-122, (Mar. 2019).

[13] 中野 和香子, 石浦 菜岐佐, 冨山 宏之, 神原 弘之: "FreeRTOS を用いたシステムのフルハードウェア合成," 信学技報, VLD2019-70 (Jan. 2020).

[14] W. Nakano, Y. Shinohara, and N. Ishiura: "Full Hardware Implementation of FreeRTOS-Based Real-Time Systems," in *Proc. IEEE Region 10 Conference* (Dec. 2021).

[15] 六車 伊織, 石浦 菜岐佐, 安堂 拓也, 冨山 宏之, 神原 弘之: "RTOS 利用システムのフルハードウェア化におけるサービス処理機能の集約," 信学技報, VLD2020-75 (Mar. 2021).

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