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
VLD2023-94,RECONF2023-97(2024-01)

RTOS 利用システムのフルハードウェア化における状態レジスタの最適化による回路規模削減

三上　啓[†]　石浦菜岐佐[†]　冨山　宏之[††]　神原　弘之[†††]

† 関西学院大学　〒669–1330 兵庫県三田市学園上ケ原 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 を用いて実装されたシステムをフルハードウェア化する手法において, 状態レジスタの最適化およびサービス要求調停回路の多段化により, タスク数が増えた場合の回路規模およびクリティカルパス遅延を削減する手法を提案する. 大迫・六車らはリアルタイムシステムの応答性能を向上させる手法として, 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手法を提案しているが, タスク数は 8 個程度を想定しており, タスク数がこれよりも増加すると, 回路規模とクリティカルパス遅延が過大になる. 本稿では, タスクやシステムの状態を記憶する状態レジスタのビット幅削減と結合により回路規模増加の抑制を図る. また, サービス要求調停回路を多段化することによりクリティカルパス遅延増加の抑制を図る. Xilinx Vivado を用いて, 本手法に基づくタスク数 64 個の管理ハードウェアを合成した結果, 従来手法に比べて, 回路規模を約 48%, クリティカルパス遅延を約 55% に削減できた. また, タスクがサービス処理を要求してから返り値を受信するまでの実行サイクル数を, 平均約 1 サイクル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

Reduction of Circuit Size by Optimizing Status Registers in Full Hardware RTOS-Based Systems

Kei MIKAMI[†], Nagisa ISHIURA[†], Hiroyuki TOMIYAMA[††], and Hiroyuki KANBARA[†††]

† Kwansei Gakuin University,　1 Uegahara, Gakuen, Sanda, Hyogo, 669–1330, 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 presents a technique for handling increased number of tasks by reducing both circuit size and critical path delay, within the context of full hardware implementation of RTOS-based systems. Oosako and Muguruma previously proposed methods to enhance the response performance of real-time systems by implementing both tasks/handlers and RTOS kernel functions as hardware. However, their methods assume around 8 tasks, and surpassing this count results in impractical circuit size and critical path delay. In our work, we address these scalability challenges by adopting a more efficient circuit design. This involves reducing the bit count of state registers responsible for storing task states and integrating them into a single register per task. Additionally, we control the critical path delay by multi-stage implementation of the service request arbitration circuit. We have designed a management module that incorporates RTOS functions for 64 tasks based on our proposed method. This design has led to a substantial reduction in circuit size, approximately 48%, and a decrease in critical path delay by around 55% when compared to the previous design. Furthermore, we have observed an average reduction of approximately 1 cycle in the number of execution cycles from the initiation of a task requesting service processing to receiving the return value.

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

1. はじめに

近年の情報通信技術の進展により, 新たなサービスやデバイスが次々に開発されている. これに伴い組み込みシステムには益々高い機能が求められるようになっている. 特に車載機器や無人航空機の制御では, 機能性に加えてリアルタイム性が要求される. このようなシステムは, 入力イベントに対する処理を決められた時間内に完了させる機能を提供するリアルタイム OS (RTOS) を用いて設計される. しかし, システムの高機能化が進むにつれてリアルタイム性の実現は困難になりつつある.

RTOS を使用したシステムの応答性能向上の手法として, RTOS 機能のハードウェア化が提案されている. 文献 [1], [2], [3] は RTOS のスケジューリング機能を, 文献 [4], [5] は RTOS のほとんどの機能をハードウェア化することにより, 応答性能の向上を図っている. 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ヘッドが発生する.

この課題を解決する一手法として, 文献 [6] は RTOS の機能およびタスク/ハンドラの全てをハードウェア実装する手法を提案している. 文献 [7] は文献 [6] のアーキテクチャにおいて, RTOS のサービス処理機能を集約することにより回路規模を削減する手法を提案している. また, 文献 [8] は文献 [7] のアーキテクチャにおいて, タスクを汎用的な高位合成ツールで合成できる制御手法を提案している.

しかし, これらの手法ではタスク数があまり多くないシステムを想定しており, タスク数が多くなると回路規模とクリティカルパス遅延が過大になる. 回路規模はタスク数に比例して, クリティカルパス遅延はタスク数の対数に比例して増加し, タスク数 64 の場合には許容範囲を超えてしまう.

本稿では, 文献 [9] のアーキテクチャにおいて, 実用的なタスク数を 64 個まで拡張することを目標に, 文献 [9] の管理ハードウェア内部にある状態レジスタを最適化することにより回路規模の削減を図る. また, サービス要求調停回路を多段化することによりクリティカルパス遅延の抑制を図る.

本手法を用いて, タスク数 4, 8, 16, 32, 64 個のシステムを設計した. タスク数 32 個の際, 回路規模を約 40%, クリティカルパス遅延を約 48% に削減することができた. 同様に, タスク数 64 個の際, 回路規模を約 48%, クリティカルパス遅延を約 55% に削減することができた. また, タスクがサービス処理を要求してから返り値を受信するまでの実行サイクル数は, 平均約 1 サイクル削減することができた.

2. RTOS 利用システムのフルハードウェア実装

2.1 概　　念

文献 [6] は RTOS の機能とタスクおよびハンドラ全てをハードウェア化する手法を提案している. その概念を図 1 に示す. 上図のタスク (TSK*i*) は RTOS の管理下で CPU により実行される. これらのタスクは高位合成技術を用いて独立したモジュールとしてハードウェア化される. RTOS の機能をハードウェア化したものがマネージャ (manager) である.

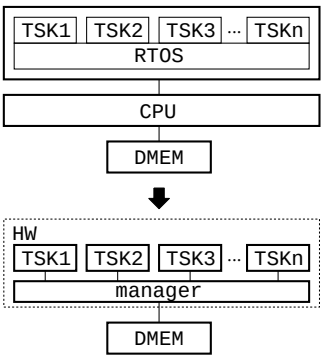


図 1: RTOS 利用システムのフルハードウェア実装 [6]

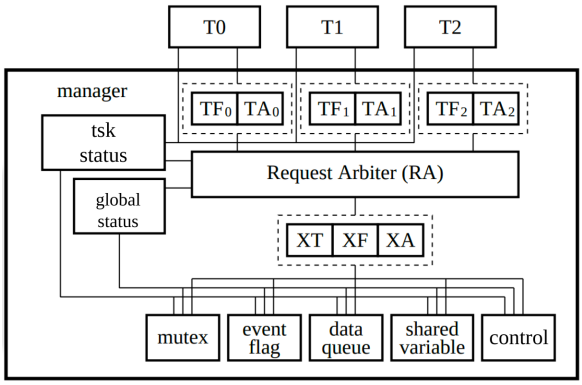


図 2: 文献 [9] のハードウェアの構成

実行可能状態になったタスクは全て並列に実行される. マネージャは各タスクの状態に基づいて実行/停止の信号を出力することによりタスクの制御を行う. この手法では, 各タスクは独立に並列実行されるため, CPU 待ち, スケジューリングおよびコンテキストスイッチのオーバヘッドをなくせる. その上, タスクをハードウェア化して高速実行できるため, システムの応答性能を飛躍的に向上させることができる.

2.2 文献 [9] のアーキテクチャ

本稿で前提とする文献 [9] のアーキテクチャを図 2 に示す. T0, T1, T2 はタスクを, manager は RTOS 機能をハードウェア化したモジュールである.

manager の下部にある mutex, event flag, dataqueue, shared variable, control は, タスクが要求する RTOS のサービス機能を提供するサービスモジュールである. mutex は排他制御, event flag はタスク間の同期, dataqueue はタスク間の通信, shared variable はタスク間で共有される変数へのアクセスや動的メモリ割り当ての機能を提供するモジュールである. また, control はタスクの起動/休止や優先度の変更等のサービスを提供する.

タスクやシステムの状態は manager 内の状態レジスタである tsk status および global status に保持される. tsk status はタスク状態レジスタであり, 各タスクのステータス情報 (タスクの状態, 現在の優先度, ベース優先度, タイマー等) を管理する. global status はグローバル状態レジスタであり, カーネルの情報 (割込みロックフラグ, CPU ロックフラグ, 割込優先度マスク等) を管理する. サービスモジュールは状態レジスタを参照・更新する

表 1: タスク状態レジスタ

項目	内容	削減後の bit 数
tskstat	タスク状態	3
tskpri	タスクの現在優先度	6
tskbpri	タスクのベース優先度	6
tskwait	タスクの待ち要因	4
wobjid	タスクの待ち対象のオブジェクトの ID	3
lefttmo	タスクがタイムアウトするまでの時間	32
actcnt	タスクの起動要求キューイング数	8
wupcnt	タスクの起床要求キューイング数	8
raster	タスク終了要求状態	1
dister	タスク終了禁止状態	1

表 2: グローバル状態レジスタ

項目	内容	削減後の bit 数
f_prohibit_int	全割り込みロックフラグ	1
f_cpu_locked	cpu ロックフラグ	1
intpri_mask	割込優先度マスク	6
f_prohibit_dispatch	ディスパッチ禁止フラグ	1
systim	システム時刻	32
hrtcnt	高分解能タイマのカウント値	32

ことにより処理を実行する.

タスクは並列に実行されるが, サービス間の干渉を避けるためサービスは一つずつ逐次的に実行される. 複数のタスクからサービス要求があった場合には優先度の高い順に要求が処理される. manager 中央にある Request Arbiter (RA) は優先度に基づいてこの調停を行う回路である.

タスク (T*i*) 中のサービスコールは制御レジスタ TF*i*, TA*i* への書き込みに変換される. タスクはレジスタ TF*i* にサービスの ID を, TA*i* に必要な引数を書き込んでサービス処理の完了を待つ. RA は優先度最大のタスク番号を求め, レジスタ XT, XF, XA にそれぞれ優先度最大のタスク番号, TF*i* のサービスの ID, TA*i* の引数を書き込む. サービスモジュールは要求された処理を行い, 返り値を XA レジスタに書き込む. manager が XA の値を TA*i* にコピーしてタスクにサービスの完了を通知すると, タスクは TA*i* から返り値を読み取って自身の処理を再開する.

文献 [9] のアーキテクチャでは, 回路規模はタスク数に比例して, クリティカルパス遅延はタスク数の対数に比例して増加する. 文献 [10] の評価では, タスク数 64 のシステムをハードウェア化した場合の回路規模が 32bit の MIPS プロセッサコアの約 10 倍以上, クリティカルパス遅延が 20 ns に近い値となる. 本稿ではターゲットデバイスの FPGA のデフォルト動作周波数である 100 MHz で動かすことを目標としている. そのため, タスク数が増加した場合でも, クリティカルパス遅延を 10 ns 以下に抑える必要がある.

3. 状態レジスタの最適化による回路規模削減

3.1 状態レジスタ

合成される回路を分析した結果, タスク数が増えた場合の回路規模に最も大きな影響を与えているのは, 状態レジスタおよびそのアクセス回路であることが判明した.

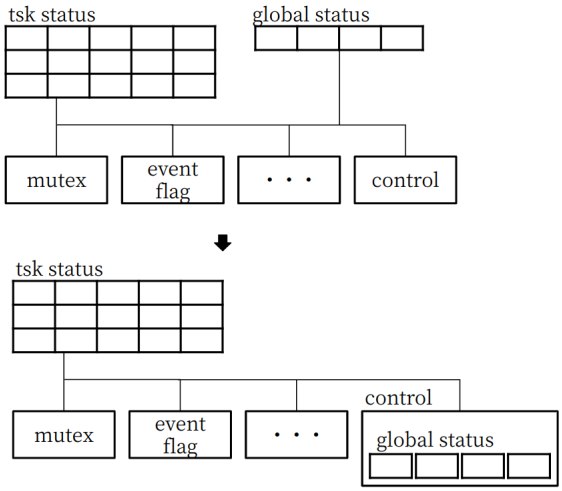


図 3: グローバル状態レジスタの再配置

状態レジスタのうち, タスク状態レジスタはタスク毎に必要になる. 例えば TOPPERS/ASP3 では, 1 つのタスクにつき表 1 に示す 10 項目の記憶が必要になる. 一方, グローバル状態レジスタはシステム全体で一組だけ必要になる. TOPPERS/ASP3 の場合, 表 2 に示す 6 項目の記憶が必要になる.

本稿では, タスク状態レジスタ, グローバル状態レジスタを必要最低限のビット数に削減するとともに, タスク状態レジスタを結合して二次元のレジスタ配列を一次元化することによって, 回路規模の削減を図る.

3.2 ビット数の削減

ソフトウェア実装では状態レジスタはメモリに配置され, 各項目はアクセス効率のため 1 ワード (32bit) に格納される. 文献 [10] のアーキテクチャではこれをそのままレジスタファイルで実装していたため, 回路規模が増加していた. 本稿では, これを個別のレジスタに配置しビット幅を削減する.

TOPPERS/ASP3 の場合には, タスク状態レジスタのビット数は表 1 の最右列に示すように削減できる. 詳細は以下の通りである.

- tskstat : タスクの状態は全部で 6 通りなので 3bit で表現できる.
- tskpri : 優先度の値は 1~32 なので 6bit で表現できる (0 は特殊用途で使用する).
- tskbpri : 優先度の値は tskpri と同様 6bit で表現できる.
- tskwait : 待ち要因の数は 12 個なので 4bit で表現できる.
- wobjid : 実装しているサービスモジュールの数が 8 個なので 3bit で表現できる.
- actcnt : 2^8 回カウントできれば十分なので 8bit で表現できる.
- wupcnt : 2^8 回カウントできれば十分なので 8bit で表現できる.
- raster : 状態は 2 通りなので 1bit で表現できる.
- dister : 状態は 2 通りなので 1bit で表現できる.

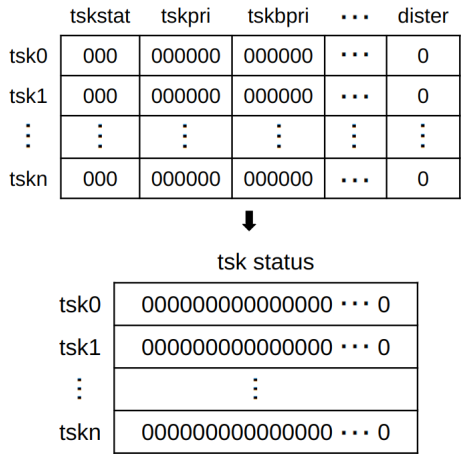


図 4: レジスタの結合

グローバル状態レジスタについてもタスク状態レジスタと同様にビット数削減を行う. グローバル状態レジスタのビット数は表 2 の最右列に示すように削減できる. 詳細は以下の通りである.

- f_prohibit_int : 状態は 2 通りなので 1bit で表現できる.
- f_cpu_locked : 状態は 2 通りなので 1bit で表現できる.
- intpri_mask : 優先度の値は 1~32 なので 6bit で表現できる.
- f_prohibit_dispatch : 状態は 2 通りなので 1bit で表現できる.

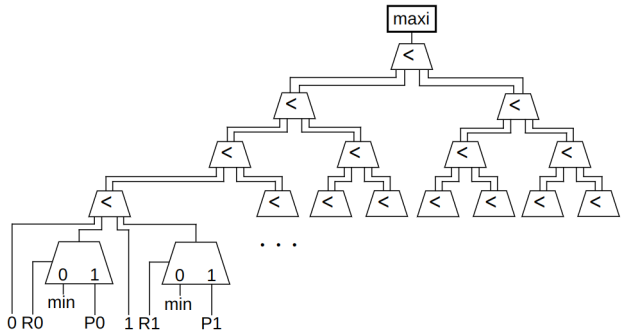
また, サービスモジュールのうちグローバル状態レジスタを参照・更新するのは control のみである. 従来のアーキテクチャ [9] では, タスク状態レジスタもグローバル状態レジスタも全てのサービスモジュールからアクセスできる構成にしていたが, 本手法では図 3 のようにグローバル状態レジスタを control サービスモジュールの内部レジスタに変更する. これにより, タスク状態レジスタと並列にグローバル状態レジスタにアクセスが可能となる. さらに, グローバル状態レジスタの各項目には独立にアクセスが可能となるため, アクセス回路の削減とともに, アクセスの高速化が図れる.

3.3 レジスタの結合

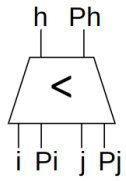
従来 [9] の実装では, タスク状態レジスタはタスク番号とタスク状態レジスタの項目番号を与えてアクセスするレジスタの二次元配列になっており, これがアクセス回路の規模を大きくしていた. 本手法では, 1 つのタスクに関する状態レジスタを結合して, タスク状態レジスタを一次元化することにより, 回路規模の抑制と状態レジスタへのアクセス回数の削減を図る.

レジスタの結合例を図 4 に示す. tskstat や tskpri などの別々のレジスタで保存していた情報を結合して 1 つのレジスタにまとめる. これにより, 読み出しは全ての項目が一度に行える. また, 書き込みはマスクを使用することにより, 指定した項目だけに対して行えるようにする.

いくつかのサービス処理では, 状態レジスタの複数の項目にアクセスする必要がある. 文献 [9] のアーキテクチャでは, その都度状態レジスタへのアクセスが必要であったが, 本稿の回路構成ではこれが効率化できる.



(a) 最大優先度のタスクを求める回路



(b) 比較ユニット

図 5: Request Arbiter (RA) の回路構成

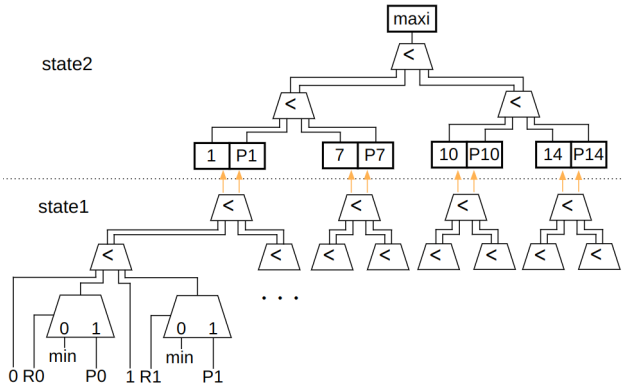


図 6: 優先度比較の多段化

4. サービス要求調停回路の多段化によるクリティカルパス遅延の抑制

文献 [9] のアーキテクチャでクリティカルパス遅延を大きくしている最大の要因は, サービスを要求しているタスクの中から優先度最大のものを求める RA (Request Arbiter) モジュールである.

タスク数 16 の場合の RA の回路構成を図 5 (a) に示す. 回路への入力は, 各タスクについてタスクの番号 (i), サービス要求を出しているかどうか (Ri), およびタスクのその時刻での優先度 (Pi) であり, 出力は, サービス要求を出している最も優先度の高いタスクの番号 (maxi) である. min は仮想的な最低優先度であり, サービス要求を出していないタスクの優先度を表現するのに用いる. タスクの優先度も要求を出すタスクも動的に変化するため, RA の計算は毎サイクル必要になる.

RA は図 5 (b) に示す比較ユニット (2 つの優先度 Pi と Pj を比較し, 大きい方の優先度とタスク番号を出力する) をツリー状

表 3: 合成結果

	文献 [9]				提案手法			
TASK	#LUT	#FF	latency [ns]	RA の段数	#LUT (比率)	#FF	latency [ns]	RA の段数
4	4,067	2,227	7.741	1	1,273 (31.3%)	669	3.772	2
8	6,182	3,996	10.886	1	2,157 (34.8%)	1,239	5.023	2
16	11,667	7,561	13.041	1	5,217 (44.7%)	2,487	5.752	2
32	23,209	14,790	16.226	1	9,456 (40.7%)	4,892	7.817	2
64	48,959	29,462	18.751	1	23,675 (48.3%)	9,761	10.359	2

表 4: 実行サイクル数

	文献 [9]	提案手法
service call	total (起動 + 実行 + 受信)	total (起動 + 実行 + 受信)
loc_mtx	9 (3 + 3 + 3)	9 (4 + 2 + 3)
unl_mtx	8 (3 + 2 + 3)	9 (4 + 2 + 3)
act_tsk	10 (3 + 4 + 3)	9 (4 + 2 + 3)
chg_pri	9 (3 + 3 + 3)	9 (4 + 2 + 3)
wup_tsk	12 (3 + 6 + 3)	9 (4 + 2 + 3)

に接続することにより構成される.

本手法では優先度比較を多段化することによりクリティカルパス遅延を抑制する. RA を構成する比較モジュールの段数はタスク数の対数に比例して増加する. 優先度比較を多段化した例を図 6 に示す. 1 クロック目で求めた結果のタスク番号と優先度をレジスタを用いて次のクロックに渡すことにより, 優先度比較を多段化する.

5. 実装と実験

本手法に基づいて変更を加えた管理ハードウェアを 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. 合成した manager モジュールの詳細は以下である.

- タスク数 4~64
- サービスモジュールは mutex と control の 2 個
- control がサポートするサービスコールは以下の 12 個

  act_tsk, can_act, ter_tsk, chg_pri, get_pri, wup_tsk, can_wup, rel_wai, sus_tsk, rsm_tsk, loc_cpu, unl_cpu

manager モジュールの論理合成結果を表 3 に示す. 「#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り, 括弧の中は文献 [9] を 100% としたときの提案手法の比率である. 本手法では全てのタスク数について RA を 2 段で設計した. 回路規模はタスク数 16 個の場合に 44.7%, 32 個の場合に 40.7%, 64 個の場合に 48.3% に削減できた. クリティカルパス遅延はタスク数 16 個の場合に 44.1%, 32 個の場合に 48.1%, 64 個の場合に 55.2% に削減できた. 一方, タスク数が 64 個の際, 回路規模が約 23,000LUT と依然として大きく, クリティカルパス遅延も目標の 10 ns 以下に抑えられていない.

タスクがサービス要求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 4 に示す. 括弧内の「起動」はタスクがサービス要求を開始してからサービスモジュールが処理を始めるまで,「実行」はサービスモジュールが処理を始めてから終えるまで,「受信」はサービスモジュールが処理を終えてからタスクが返り値を受信するまでの実行サイクル数である. service call の上位 2 つは mutex の処理であり, 下位 3 つは control の処理である. mutex モジュールの処理では, 状態レジスタへのアクセスが少ないため, RA の多段化により 1 サイクル増加している. 一方, control モジュールの処理では, 状態レジスタへのアクセスが多いため, 状態レジスタの結合によりサイクル数が減少している.

6. むすび

本稿では, 状態レジスタのビット数削減とレジスタの結合, 優先度比較を多段化した RA を提案した. タスク数が 32, 64 のように大きい場合でも, 文献 [9] に比べてシステム全体の回路規模とクリティカルパス遅延を大幅に削減することができた.

一方, タスク数が 64 の際, 回路規模とクリティカルパス遅延が目標の値を上回る値になっている. これにより, さらなるボトルネックの分析や回路の最適化が必要である.

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).

[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] 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).

[7] 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).

[8] T. Ando, I. Muguruma, Y. Ishii, N. Ishiura, H. Tomiyama, and H. Kambara: “Full hardware implementation of RTOS-based systems using general high-level synthesizer,” in *Proc. SASIMI 2022*, pp. 2–7 (Oct. 2022).

[9] M. Nakahara and N. Ishiura: “Arrival order processing of service requests in full hardware implementation of RTOS-based systems,” in *Proc. ITC-CSCC 2023*, pp. 467–472 (June 2023).

[10] H. Minamiguchi, N. Ishiura, H. Tomiyama, and H. Kanbara: “Automatic generation of management module for full hardware implementation of RTOS-based systems,“ in *Proc. ITC-CSCC 2023*, pp. 473–478 (June 2023).