組込みシステムにおける
消費エネルギー最適化のための
スクラッチパッドメモリ管理技術

髙瀬　英希

組込みシステムにおける消費エネルギー最適化のための スクラッチパッドメモリ管理技術

要旨

組込みリアルタイムシステムにおいて，性能や計算精度を保証しながら消費エネルギーを最適化することは，非常に重要な課題となっている．消費エネルギーを削減することによって，製造・運用コストの節減や信頼性の向上などの様々な利益が得られる．近年の組込み向けプロセッサでは，キャッシュに代表されるメモリシステムの消費エネルギーが，プロセッサ全体の約半分を占めている．このことは，メモリシステムで消費されるエネルギーを削減することは，組込みシステム全体の消費エネルギー削減に繋がることを示している．そこで本研究では，小容量かつ高速なオンチップ SRAM であるスクラッチパッドメモリ（以下，SPM）の活用に着目する．SPM は，回路面積，実時間性，および，消費エネルギーの面で組込みシステムに適したオンチップメモリである．ただし，SPM は，保持する内容を更新するハードウェア機構を内部に持たない．このため，SPM の配置内容はソフトウェアによって明示的に管理する必要がある．

本研究の目的は，組込みシステムのプロセッサおよびメモリシステムの消費エネルギーを最小化することである．キャッシュと SPM を組み合わせて一次メモリが構成されるシステムにおいて，SPM を効率良く活用する技術を確立し，システム全体の消費エネルギー最適化を目指す．さらに，提案する SPM 管理技術を，実用システムに適用可能なソフトウェア・ツールチェーンとして開発する．

本論文では，以下に示す 5 つの研究課題の成果を報告する．まず，組込みシステムに SPM を導入することの有効性を，リーク電力も含めて評価した．2 つめの研究では，組込みシステムのマルチタスク環境における SPM 領域活用手法を提案した．3 つめの研究では，プリエンプティブな固定優先度ベースの組込みリアルタイムシステムに適用可能な SPM 活用戦略および SPM 管理のためのワークフローを提案した．4 つめの研究では，特別なハードウェアを使用することなく SPM の配置内容を実行時に効率良く管理できるリアルタイム OS を開発した．最後に，5 つめの研究では，多階層の協調による組込みリアルタイムシステムの消費エネルギー協調最適化フレームワークを構築した．

リーク電力とは，メモリの動作とは無関係に定常的に漏れ流れる電流に起因する電力のことを指す．半導体設計技術の微細化が進む最先端のメモリデバイスでは，リーク電力の消費量は増大の一途を辿っているといわれている．そこで 1 つめ

の研究では，評価実験によって，リーク電力を考慮に含めてもなお，キャッシュとSPMを組み合わせて組込みシステムの一次メモリを構成することの有効性の高さを明らかにした．また，リーク電力の消費量が顕著となる次世代，次々世代の製造技術のメモリシステムにおいても，組込みシステムにおけるSPMの優位性は変わらず高いという傾向も示した．本研究は，従来研究と比較して，リーク電力もSPMの有効性の議論に考慮しているところに新規性があり，組込みシステムにおけるSPMの有効性をより妥当に検証できている．

高いリアルタイム応答性が要求される組込みマルチタスクシステムでは，固定優先度ベースのスケジューリング方式が一般的に採用される．2つめの研究では，非プリエンプティブなマルチタスク環境に対応した，効率的なSPM領域分割法を示した．提案手法は，空間分割法（使用可能なSPMの容量を各タスクに分配して与える手法），時間分割法（実行状態のタスクがSPMの全領域を占有する手法），および，混合分割法（上記2方針の組み合わせ）の3種類である．これらのSPM領域分割法は，組込みシステムの命令メモリの消費エネルギー最小化に貢献する．それぞれの領域分割法は，各タスクに割当てるSPM容量とSPM領域へのコード配置とを同時に探索可能な整数計画問題として定式化した．

次に，3つめの研究では，上記の手法を発展させ，プリエンプティブな固定優先度ベースのリアルタイム・タスクスケジューリング方式に従う環境に適用可能なSPM活用戦略を提案した．これに伴い，前述の混合分割法を発展させたSPM活用戦略である混合活用法を提案した．本研究における混合活用法は，高優先度タスクは，より優先度が低いタスクのSPM領域を横取りして使用できる．これにより，SPMの配置内容の入れ替えにかかるオーバヘッドを抑制しつつ，より柔軟にSPMを活用できる．それぞれのSPM活用戦略は，定式化の際にタスクの優先度や起動周期といったスケジューリングの性質を考慮することにより，消費エネルギー最適となるSPMの管理方法を決定しているのが特徴である．それぞれのSPM活用戦略は，整数線形計画問題として形式的に定式化した．さらに，SPMの保持する内容を実行時に管理するために求められるシステム要件を明らかにした．本要件は，リアルタイムOSとハードウェアの協調動作によってSPM管理を実現するワークフローとして構築した．提案したワークフローを実験環境に実装して評価を行い，SPM活用戦略の有効性を確認した．

マルチタスク環境においては，限られたメモリ容量を有効に活用できるよう，タスク間でSPM領域の配置内容を時間的に再配置するのが望ましい．そこで4つめの研究では，複数のタスクで共有して活用されるSPM領域を実行時に管理する機

能を有するリアルタイムOSを開発した．提案手法は，全てソフトウェアの機能のみで動作し，小メモリサイズおよび低オーバヘッドで実現できるよう設計した．実行時のSPM管理機能は，SPM管理情報および実行時SPM管理機構から構成される．SPM管理情報とは，リアルタイムOSがSPM管理の実現のために持つ静的な情報のことである．実行時SPM管理機構は，リアルタイムOSの内部動作として実現できるように設計した．タスクのスケジューリング時にSPM管理情報を参照することで，SPM領域のうち再配置が必要な部分のみを選択できる．これにより，SPM領域の配置内容の効率的な管理を実現する．提案手法をTOPPERS/ASPカーネル上に実装し，SPM管理機能のメモリサイズと性能のオーバヘッド，および，アプリケーション実行時の消費エネルギーを評価した．これらの評価を通して，本研究の有効性を明らかにした．

5つめの研究では，多階層に跨る消費エネルギー協調最適化フレームワークを構築した．本フレームワークは，複数の要素技術を統合することで，組込みリアルタイムシステム全体の協調最適化を実現する．統合した要素技術としては，タスクの振る舞いや特性を実行トレース群から自動抽出するプロファイラ技術，コードおよびデータのメモリ領域配置先を最適に決定するコンパイラ技術，および，ハードウェア資源におけるエネルギーと性能のトレードオフを実行時に最適に制御するリアルタイムOS技術がある．複数の要素技術を統合するため，本フレームワークは設計時のタスク毎およびタスク間，実行時の3段階にて対象システムの最適化を行う形態をとる．段階的なアプローチを取ることで，統合した各要素技術の最適化効果を効率的に活かすことができる．さらに，本フレームワークは，組込みシステムのアプリケーション開発支援環境となるソフトウェア・ツールチェーンとして開発した．ツールチェーンによる消費エネルギー最適化処理は，自動的に適用されるよう設計されている．テレビ会議システムを例題とした評価を行うことにより，開発成果が実用的な組込みアプリケーションに適用可能であり，かつ，組込みリアルタイムシステムの消費エネルギー最適化を実現することを示した．

以上の研究は，次に挙げる特色および独創的な点がある．まず，実用製品に則ったシステム環境を強く考慮して研究に取り組んできている．1つめの研究では，消費量が年々増大しているリーク電力にも着目し，SPMの有効性をより妥当に評価した．2つめおよび3つめの研究では，組込みシステムにおいてひろく一般に採用される固定優先度ベースのマルチタスク環境において，SPMの有効な活用手法を示した世界初の成果である．さらに，本研究は，単なる理論研究にとどまらず，提案手法を実用的なものとして実装に取り組んでいる．4つめの研究で提案するSPM管

理機能は，μITRON 仕様に準拠したリアルタイム OS である TOPPERS/ASP カーネル上に実装されることを想定して設計している．このため，実用的な組込みアプリケーションにおいてひろく提案手法を適用できる．さらに，5 つめの研究のような，複数の設計階層が統合された革新的な最適化フレームワークを提示した例は，私の知る限り存在しない．開発したソフトウェア・ツールチェーンは，テレビ会議システムという実用システムに適用できている．このことは，開発成果物が産業界で利用できる実用化に近いレベルまで完成度が高められていることを証明している．これらの研究成果が，組込みシステム製品の消費エネルギー最適化を促進することにつながり，社会全体のエネルギー問題に貢献することが期待できる．

Scrath-pad Memory Management Techniques for Energy Optimization of Embedded Systems

Abstract

Energy optimization has become one of the primary goals in the design of modern embedded systems. In the embedded systems, especially battery-powered ones, it is important to reduce energy consumption. Reducing energy consumption can extend battery lifetime of portable systems, decrease chip cooling costs, and increase system reliability, *etc.* Recently memory subsystems have consumed a large amount of total energy in the embedded processors. These days, cache memory is used not only in general-purpose processors but also in embedded processors in order to improve their performance. Cache also contributes to energy reduction because of decreased accesses to off-chip memory. However, cache is one of the most energy-hungry components in embedded processors. Thus, a large amount of studies have addressed energy reduction in the memory subsystems.

More recently, scratch-pad memory (SPM) has attracted attention as an alternative to cache memory. SPM is a small and fast on-chip SRAM. Since no tag comparison is necessary, SPM has advantages in terms of circuit area, real-time predictablity, and energy efficiency compared with cache memory. Thus, SPM is suitable for embedded real-time systems. However, the contents of SPM should be managed by the application designer or the software because there is no hardware mechanism to manage the contents of SPM. The software approach is needed to manage the contents of SPM.

This thesis proposes software-centric techniques for the management of SPM. The main purpose of this research is the energy minimization of embedded real-time systems. Proposed techniques are implemented on the software toolchain to be applicable for the practical systems.

This thesis includes five topics on the efficient management of SPM. First, evaluation of SPM to embedded systems considering leakage energy is investigated. Second, SPM division policies corresponding to the non-preemptive multi-task systems are proposed. Third, this thesis proposes three methods of partitioning and allocation of SPM for fixed-priority-based preemptive multi-task systems. Fourth,

design and implementation of an SPM management mechanism within the real-time operating system are conducted. Fifth, this thesis proposes an integrated optimization framework for minimizing the energy consumption of embedded real-time applications.

A number of approaches have been proposed so far for reducing the energy consumption of embedded systems by using SPM. However, most of them focused on dynamic energy reduction, and neglected the leakage energy in their evaluations. As technology scales down to the deep sub-micron domain, the leakage energy in memory devices has contributed to a significant portion of the total energy consumption. Thus, evaluating total energy consumption including leakage energy is especially important. In this research, the effect of SPM on energy reduction considering both dynamic and leakage energy is investigated. The experiments were performed for 65 nm, 45 nm, and 32 nm memory devices. The results demonstrated the effectiveness of SPM in deep sub-micron technology. It is also observed that the leakage energy becomes less significant along with the technology scaling.

Recently, the scale and the complexity in real-time systems have been increasing. Processors are typically required to execute two or more tasks concurrently. The task scheduling algorithm under the priority is generally employed since fast response is important in real-time systems. However, almost all of previous SPM allocation techniques have focused on single-task environments. Applying these techniques to multi-task systems will result in non-optimal energy savings. This research proposes partitioning and allocation approaches of SPM in non-preemptive fixed-priority multi-task systems. This reseach proposes three approaches (i.e. spatial, temporal, and hybrid approaches) which enable energy efficient usage of the SPM region. These approaches achieve the energy minimization of instruction memory. Each approach is formulated as an integer programming problem that simultaneously determines (1) partitioning of the SPM space for the tasks, and (2) allocation of functions to the SPM space for each task. These formulations take into account on task periods for the purpose of energy minimization. The evaluation experiments were conducted, and the effectiveness of proposed techniques was confirmed.

Next, these above approaches are extended to be applicable to preemptive multi-task systems in the third research. With the spatial method, each task occupies its

exclusive space in SPM. With the temporal method, the running task uses entire space of SPM. The content of SPM is swapped out as a task executes or gets preempted. The hybrid method is based on the spatial one but a higher priority task can temporarily use the space of lower priority task. The amount of SPM space is prioritized for higher priority tasks. These methods not only support the real-time task scheduling but also consider aggressively the periods and priorities of tasks for the energy minimization. Additionally, an RTOS-hardware cooperative support framework is proposed for runtime code allocation to the SPM space. The experimental results have demonstrated the effectiveness of these techniques. Up to 73 % energy reduction compared to a conventional method was achieved.

The fourth research proposes a design of the software mechanism for the efficient management of the SPM. This mechanism consists of the SPM management information and the runtime SPM function on the real-time operating system. The former is the table information for managing of the SPM. These pieces of information are produced at system design time according to the use of SPM among multiple tasks. The space of SPM is dynamically reallocated by the function of real-time operating systems according to these pieces of information. The proposed mechanism is implemented within the TOPPERS/ASP kernel. Additionally, an API for the management of SPM is proposed. This reseach discusses the applicable range and effectiveness of the management API. The evaluation about memory size and performance overhead shows the effectiveness of proposed mechanism.

The last research presents a framework for the purpose of energy optimization of embedded real-time systems. The presented framework is implemented as an optimization toolchain and an energy-aware real-time operating system. This framework is the integration of multiple techniques which optimize the target application together. The main idea of this approach is to utilize a trade-off between energy and performance of the processor configuration. The optimal processor configuration is selected at each appropriate point in the task. Additionally, an optimization technique about the memory allocation is employed in this framework. This framework is also gradual, that is, the target application is optimized in a step-by-step manner. The characteristic and the behavior of target applications are analyzed and optimized for both intra-task and inter-task levels by the toolchain at the static time. Based on the results of static time optimization,

the runtime energy optimization is performed by a real-time operating system according to the behavior of the application. A case study with the video-conference system shows that the minimization of the average energy consumption is achieved while keeping the real-time performance.

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究背景

組込みシステムとは，携帯電話端末やデジタルテレビをはじめとした情報家電製品などに内蔵されている，特定の機能を実現するためのコンピュータシステムのことを指す．近年では，身の回りにあるほとんどの機器には，何らかの組込みシステムが搭載されているといっても過言ではない．組込みシステムには，大型計算機やパーソナルコンピュータなどの汎用システムとは異なるいくつかの特徴がある．まず，システムに要求される機能や性能が明確であることが挙げられる．次に，システムに組込まれているプロセッサは，ある程度特定の条件下での動作が想定されている．さらに，組込みシステム上で実行されるプログラムは，定められた一定時間内に処理を完了しなければならないというデッドライン制約を満たし，安定したリアルタイム性（実時間性）を保証することが要求される．ここで，特に，高いリアルタイム性が要求されるシステムのことを，組込みリアルタイムシステムという．

組込みリアルタイムシステムにおいては，消費エネルギーの最適化が非常に重要な課題となっている．システムの性能や動作，計算精度，さらにはリアルタイム性を保証しながら消費電力ないしは消費エネルギー[1] を削減することによって，数多くの意義深い利点が得られる．例としては，製造コストおよび運用コストが低減されること，信頼性が確保されること，環境および生活が保全されることなどが挙げられる．電力線から電力を直接供給される組込みシステムでは，動作時だけでなく待機中にも消費エネルギーが発生するため，消費エネルギーが小さいことは，電力料金が抑えられると同時に，システムの信頼性が高まる．また，バッテリ駆動の携帯型組込みシステムでは，性能や機能が高いことに加え，屋外で利

[1] ここで，消費電力と消費エネルギーの違いについて簡単に補足する．
消費電力は仕事率（瞬間発熱量）を表し，スイッチング時にトランジスタ中を流れる電流と電源電圧との積により求められる．これに対して，消費エネルギーは仕事量を表し，消費電力の時間積分（近似的には消費電力と実行時間の積）により求められる．消費電力はシステムの信頼性に関わり，消費エネルギーはバッテリ寿命に影響を与える．

用することを想定すると連続駆動時間が長いことが望まれる．つまり，組込みシステムの消費エネルギーの大小は，システムの商品価値を大きく左右する要素になっている．さらに，組込みシステムは身の回りに数多くあるため，システム単体の消費エネルギーを少しずつでも削減すれば，社会全体で大きな削減効果が得られる．しかしながら，システムが大規模・複雑化する中で，消費エネルギー削減のために十分な開発工数が割けない状況が叫ばれている．

近年では，組込み向けプロセッサの性能向上を目的として，チップ上にキャッシュが組み込まれている．しかし，キャッシュはチップ上に大量のトランジスタを要するため，プロセッサの面積の大部分を占めることになる．そして，キャッシュで消費されるエネルギーは，プロセッサ全体の約半分をを占めるほど大きいという報告がなされている [27]．つまり，メモリで消費されるエネルギーを削減することは，組込みリアルタイムシステム全体の消費エネルギーを削減することにつながるといえる．

以上で述べたことから，組込みリアルタイムシステムの性能や計算精度を保証しながら消費エネルギーを最適化することは，解決すべき重要な課題となっている．システムの高性能化および高機能化に従う消費エネルギーの増大もまた，近年の大きな問題となっている．これらの諸問題に対応するため，現在までに組込みリアルタイムシステムの消費エネルギーを削減することを目的とした研究が盛んに行われている．

本論文では，組込みリアルタイムシステムにおける**スクラッチパッドメモリ**（以下，SPM）に着目する．

SPM とは，小容量で高速なオンチップの SRAM のことを指す．SPM に配置するプログラムコードやデータは，コンパイラなどのソフトウェアによる制御で明示的に管理される．SPM は，キャッシュにおけるタグ領域部やタグ比較回路を持たないため，メモリを構成する面積およびトランジスタ数が少なくなる．また，スイッチングを行うトランジスタが少なくなるため，1 回のメモリアクセスで消費される動的なエネルギーが小さくなると同時に，静的なリーク電力も小さく抑えられる．さらには，キャッシュミスのような状況が避けられるため，プログラム実行時のシステムの振る舞いが容易に解析でき，実行時間の予測性が高まるという利点もある．以上に挙げたように，SPM は，消費エネルギー，回路面積および時間予測性の面から，組込みリアルタイムシステムに向いたオンチップメモリであるといえる．

組込みシステムにSPMを活用して消費エネルギーないしは性能の最適化を目指す研究は，これまでに数多く行われている．また，産業界においてもSPMを搭載した製品が発表されている．しかしながら，以下に挙げる問題点から，実際の組込みシステム製品では，SPMにある利点を最大限有効に活用できているとはいえない．

1. 実際の組込みシステム製品における適用状況を想定した上でSPMの有効性を評価したときに，組込みシステムにSPMを活用することが真に消費エネルギー削減へと貢献するかの議論は解消されていない．

2. これまでに提案されてきた従来手法は，実用的な製品のシステム構成や環境を強く意識していない．このため，従来手法は，高性能・高機能化が進む昨今の組込みシステム製品に適用できるものではない．

3. SPMは，その回路構成中に配置内容を更新するための機構を持たない．このため，SPMに配置するプログラムコードやデータは，設計者が明示的に指定する必要があり，プログラミングが困難となる．現状では，アプリケーション開発工程におけるこの作業を自動化し，設計負担を軽減できうるソフトウェア・ツールチェーンは十分に整備されていない．

SPMの有する利点や貢献を組込みリアルタイムシステムにおいて発揮するためには，このような状況を打破できる管理技術が求められると考える．

1.2　研究目的と概要

本研究の目的は，組込みリアルタイムシステムにおける消費エネルギーを最小化することである．実際の組込み製品に即したシステム環境下においてSPMを有効に活用し，要求される動作やリアルタイム性を保証しつつ，消費エネルギー最適化を実現するSPM管理技術を確立する．提案する技術はソフトウェアとして実装し，提案手法と開発成果物の実用性の高さを立証する．

本論文では，以下に示す5つの課題に関する研究を行う．まず，組込みシステムにSPMを導入することの有効性を，リーク電力も含めて評価する．次に，非プリエンプティブなマルチタスク環境に対応したSPM領域分割手法を提案する．さらに，3つめの研究では，プリエンプティブな固定優先度ベースのリアルタイム・タ

スクスケジューリング方式に従うシステムに適用可能な SPM 活用戦略，および，SPM 管理のためのワークフローを提案する．4 つめの研究では，SPM の配置内容を実行時に効率良く管理できる機能を有するソフトウェア機構を開発する．最後に，多階層の協調による組込みリアルタイムシステムの消費エネルギー協調最適化フレームワークを構築する．

1 つめの研究では，キャッシュと SPM 双方のメモリを用いて組込みシステムの一次メモリを構築することの有効性を，消費エネルギーの観点から議論する．評価実験には，メモリアクセスの動作に依らず，静的に消費されるリーク電力も評価対象とする．リーク電力とは，メモリの動作とは無関係に定常的に漏れ流れる電流に起因する電力のことを指す．リーク電力は，近年進むデバイス技術の微細化によって全体の消費電力に占めるその割合が顕著になってきており，もはや無視できるほど小さい量ではない．リーク電力の増大を招くデバイスの微細化は，半導体製造技術の進展に伴い今後も次々と進んでいく．このことから，リーク電力を含めて消費エネルギーを見積もることの価値は，非常に高くなってきている．

本研究は，従来研究と比較して，リーク電力も考慮した上で SPM の有効性を議論しているところに新規性がある．過去にも，メモリアクセス時の動的な消費エネルギーのみを評価して，SPM の有効性を主張する研究はこれまでにも数多く行われてきている．しかしながら，リーク電力も含めてメモリの消費エネルギーを評価したものはほとんど見られない．

近年の組込みシステムでは，高性能・高機能化の要求が高く，マルチタスクでの処理が必須となっている．2 つめの研究では，非プリエンプティブな固定優先度ベースのマルチタスク環境に対応した，3 種類の効率的な SPM 領域分割手法を提案する．各手法について，タスクごとに割当てる SPM 容量と SPM 領域へのコード配置とを同時に探索可能な整数計画問題として定式化する．定式化の際にタスクの起動周期を考慮することにより，消費エネルギー最適な SPM の分割方針を決定できるのが特徴である．評価実験によって，提案する 3 つの手法の有効性を確認する．

3 つめの研究では，プリエンプティブな固定優先度ベースのマルチタスク環境に適用可能で，かつ，実用性の高い SPM 管理技術の確立を目指す．高いリアルタイム性が要求される組込みリアルタイムシステムでは，プリエンプティブな固定優

先度ベースのスケジューリング方式が一般的に採用される．

本研究では，まず，上記のスケジューリング方式に従うマルチタスク環境に対応した 3 種類の SPM 活用戦略を示す．提案する戦略は，空間活用法（使用可能な SPM の容量を各タスクに分配して与える戦略），時間活用法（実行状態のタスクが SPM の全領域を占有する戦略），および，混合活用法（上記 2 方針の組み合わせ）の 3 種類である．混合活用法は，基本は空間活用法が適用されるが，高優先度タスクは，低優先度タスクの使用する SPM 領域を時間活用法により横取りして活用することができる．各戦略について，2 つめの研究における整数計画問題をベースとして，SPM 領域の分割方針および SPM へのコード配置とを同時に探索可能な整数計画問題として定式化する．

さらに，本研究では，SPM の保持する内容を管理するためのワークフローを提案する．本ワークフローは，システム設計時の静的解析の段階で SPM 活用戦略を実現するプロファイラおよびコンパイラ，および，実行時最適化として SPM への動的なコード配置を実現するリアルタイム OS およびハードウェアからなる．実行環境に本ワークフローを構築した上で評価実験を行い，提案する SPM 活用戦略の有効性を示す．

マルチタスク環境において限られたメモリ容量を有効に活用するためには，タスク間で SPM 領域を時間的に共有できるのが望ましい．4 つめの研究では，複数のタスクで共有して活用される SPM 領域を実行時に管理できる機能を有するソフトウェアを開発する．本研究では，実行時の SPM 管理を担うソフトウェアとして，リアルタイム OS の活用に着目する．

提案するソフトウェア機構は，SPM 管理情報および実行時 SPM 管理機構からなる．SPM 管理情報とは，リアルタイム OS が SPM 管理の実現のために持つ静的な情報のことである．実行時 SPM 管理機構は，タスクのスケジューリング時に SPM 管理情報を参照することで，SPM の配置内容を効率的に管理する．提案手法は，小メモリサイズおよび低オーバヘッドで要求仕様を実現できるように設計している．さらに，実行時 SPM 管理機構は，リアルタイム OS の内部機能として設計しており，システム設計者は SPM 管理のために対象とするアプリケーションのソースコードに修正を加える必要はない．また，主にソフトウェアの機能のみで要求される動作を実現するため，対象システムに新たなハードウェアを追加する必要はない．さらに本研究では，リアルタイム OS 上に提案手法を実装し，実用システムに適用可能なソフトウェアとして提供することを目指す．開発成果は，アプ

リケーション実行中に SPM の内容を再配置することが求められる管理手法の実現を支援するものである．これらの SPM 管理手法とは，例えば第 5 章における時間活用法や混合活用法が該当する．

評価として，SPM 管理機能のメモリサイズ，性能のオーバヘッド，および，評価アプリケーション実行時の消費エネルギーを示す．これらの評価結果を通し，本研究の有用性を明らかにする．

最後に，多階層に跨る消費エネルギー協調最適化フレームワークを構築する．本フレームワークには，コンパイラ，リアルタイム OS およびプロセッサといった複数の設計階層における要素技術が統合されている．これまでの研究では SPM を活用したメモリ配置最適化にのみ着目していたが，それらを発展させ拡張することにより，システム全体の消費エネルギー最適化を実現するフレームワークを開発する．本研究における協調最適化フレームワークには，SPM の活用によりマルチタスクシステムの消費エネルギーを最小化する技術に加え，著者が共同研究などで開発してきた複数の要素技術が統合的に含まれている．消費エネルギー最適化のための複数の要素技術を統合的に適用することで，組込みリアルタイムシステム全体の協調最適化を実現する．本フレームワークは，設計時のタスク毎およびタスク間，実行時の 3 段階にて消費エネルギー最適化を狙う形態をとる．段階的なアプローチを取ることで，統合したそれぞれの要素技術の最適化効果を効率的に活かすことができる．

本フレームワークは，組込みリアルタイムシステムのアプリケーション開発支援環境となるソフトウェア・ツールチェーンとして開発する．ツールチェーンによる消費エネルギー最適化処理は，自動的に適用されるよう設計する．本研究の開発成果物により，ソフトウェア開発者に負担を強いることなく，対象システムの消費エネルギー最小化の実現が期待できる．テレビ会議システムを例題とした評価を行うことにより，開発成果が実用的な組込みアプリケーションに適用可能であり，かつ，組込みリアルタイムシステムの消費エネルギー最適化を実現することを示す．

SPM を活用する研究は世界中で多々行われているが，本論文で取り組む研究は，以下の特色や貢献がある．

- 先に挙げた 1 つめの問題点に対処するため，実用製品に則った環境を強く考慮した．1 つめの研究では，消費量が年々増大しているリーク電力にも着目

し，SPM の有効性をより妥当に評価する．

- 組込みシステムにおいては，固定優先度ベースのスケジューリング方式が一般的に採用される．2 つめの研究は，このようなシステムを対象として SPM の有効な活用手法を示した世界初の成果である．さらに，3 つめの研究は，高性能・高機能化が進み，高いリアルタイム応答性が求められるプリエンプティブな組込みリアルタイムシステムにおいても SPM を有効かつ柔軟に活用できることを示す．

- 4 つめの研究は，単なる理論研究にとどまらず，提案する SPM 活用戦略をリアルタイム OS 上に実装し，産業界で利用できる実用化に近いレベルまで完成度を高めている．さらに，5 つめの研究のような，複数の設計階層が統合された革新的な消費エネルギー協調最適化フレームを提示した例は，私の知る限り存在しない．さらに，提案手法を実装したソフトウェア・ツールチェーンは，TV 会議システムという実用システムに適用できており，3 つめの問題点への解となることを実証する．

1.3　論文の構成

まず第 1 章で研究背景と目的を述べたのちに，第 2 章にて，本研究で着目する SPM の特徴や性質を，キャッシュと比較しながら詳説する．さらに，第 2 章では，関連研究として現在までに提案されている組込みシステムの消費エネルギー最適化手法についても触れる．関連研究は，SPM へのデータ配置の決定が静的なものと動的なもの，シングルタスク環境を想定しているものとマルチタスク環境を想定しているものとに分類される．

第 3 章では，リークエネルギーを考慮した上で組込みシステムに SPM を導入することの妥当性を評価した研究の成果を報告する．第 4 章では，非プリエンプティブなマルチタスク環境に対する SPM 領域分割手法の詳細を解説する．また，評価実験の結果と考察について述べる．次に，第 5 章にて，プリエンプティブな固定優先度ベースのマルチタスク環境に適用可能な SPM 活用戦略の詳細を解説する．さらに，SPM の配置内容を更新するためのワークフローについても述べる．第 6 章では，SPM 領域の配置内容を実行時に効率良く管理する機能を有するリアルタイム OS の設計および実装を解説する．提案するソフトウェア機能は，SPM 管理情報および実行時 SPM 管理機構によって構成される．第 7 章では，多階層に跨る

消費エネルギー協調最適化フレームワークに関する研究の成果を報告する．本フレームワークの設計方針および含まれる要素技術を述べたのち，技術統合と最適化処理の流れについて解説する．ソフトウェア・ツールチェーンとして実装したフレームワークをテレビ会議システムに適用した事例を通して，本研究成果に対する評価を述べる．

最後に，第 8 章で本論文のまとめと今後の展望を述べる．

第2章　スクラッチパッドメモリ（SPM）とその管理技術

SPM とは，小容量かつ高速なオンチップの SRAM のことを指す．産業界において実際に発表および販売されている SPM が搭載されたプロセッサの例としては，ARM 社の ARM966E-S [50]，東芝社の MeP [56]，ルネサス・エレクトロニクス社の SH7201 [60] といった製品が挙げられる．

本章では，まず，SPM の詳細を，キャッシュとの比較を交えながら述べる．さらに，本論文執筆時点までに提案されている関連研究として，SPM の管理技術に関する調査結果を述べる．

2.1　SPM の特徴と組込みシステムにおける利点

本節では，SPM のもつ固有な特徴・性質を，キャッシュと比較しながら述べる．

SPM とは，小容量で高速なオンチップの SRAM のことをさす．キャッシュと同様に，アクセス時間および消費エネルギーが大きい外部主記憶へのアクセス回数を小さく抑えることを目的として利用される．これにより，システム全体の実行時間および消費エネルギーの改善が期待される．プログラム中の命令またはデータの配置に関する基本的な方針として，キャッシュは，時間的および空間的局所性の双方が高いものが優先的に保持される．いっぽうで，SPM は，基本的には実行頻度・再使用性の高いものを保持する．メモリアクセスの空間的局所性には関係なく，アクセス頻度および時間的局所性の大きさに重点が置かれる．

SPM は，保持するデータの決定が空間的局所性に左右されないため，メモリ中のブロックサイズを対象プロセッサの語長に合わせて設計できるという利点がある．キャッシュの場合は，メモリアクセスの空間的局所性を活かすことを目的として，通常は語長よりも数倍大きいブロックサイズで構成される．命令またはデータがキャッシュから読出されるときは，該当ブロックに保持されている情報が一度

に読み込まれる．つまり，キャッシュからはプロセッサが必要としない情報も読込まれ，余分なエネルギーが消費される可能性がある．

SPM に配置するデータを決定する時期や方法にも，キャッシュとは異なる特徴がある．キャッシュの場合，ブロック内に保持するデータの更新は，タグ比較器などの内部のハードウェア機構によってプログラム実行中に動的に行われる．いっぽう，SPM は，保持するデータを自動的に更新するハードウェア機構を内部構造に持たない．SPM に配置するデータは，プログラムの実行前に静的に決定され，コンパイラや OS などのソフトウェアによる明示的な制御が必要となる．また，SPM の配置内容を実行時に更新するためには，一般に，SPM の内容を更新するための新たなハードウェア機構の追加や，プログラムのソースコードの修正が要求される．

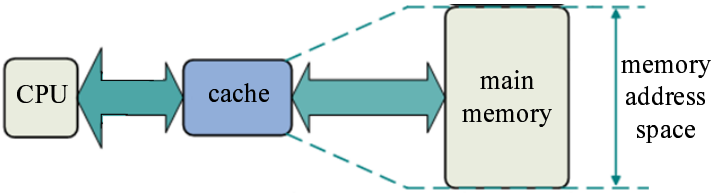
キャッシュおよび SPM と外部主記憶との関係を，メモリアドレス空間の観点から述べる．図 2.1 は，キャッシュおよび SPM と外部主記憶とのメモリアドレス空間における関係をあらわしたものである．

一次メモリがキャッシュのみの場合には，図 2.1(a) に示すように，メモリアドレス空間は外部主記憶の領域だけに割り振られる．キャッシュは，メモリアドレスの上位ビットから得られるタグ情報を用いて，外部主記憶中のキャッシング可能領域[1]に配置されている命令またはデータの一部を保持する．

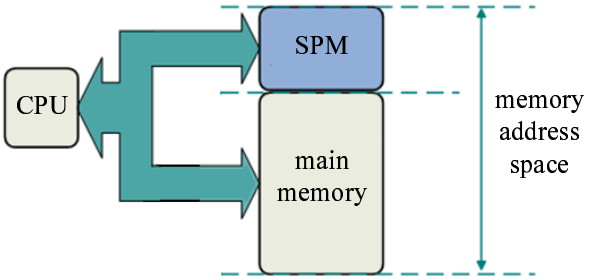
アクセス時にキャッシュブロック上に要求される情報が保持されていないときには，キャッシュミスと呼ばれる処理が発生する．キャッシュミス時には，外部主記憶からキャッシュブロックのワード数分のデータをバーストアクセスにより読み出し，キャッシュ内容の置換といった処理を行う必要がある．また，主にデータキャッシュの場合には，キャッシュの保持する情報への書込み処理が発生すると，キャッシュの内容と対応する外部主記憶の内容が異なる状況になるため，該当する外部主記憶の領域へも新しいデータを書込む処理が必要となる．以上の処理には，それぞれ大きな実行時間および消費エネルギーが発生する．

いっぽう，図 2.1(b) に示すように，一次メモリが SPM のみの場合には，SPM にメモリアドレス空間の一部が静的に割り振られる．ある命令またはデータのアドレスは，SPM の占有するアドレス空間または外部主記憶のメモリ領域のどちらかに一意に属することになる．SPM の内容が動的に更新される場合には，SPM と外部主記憶との間でのデータ転送の処理が必要となる．ただし，この場合にも，外部

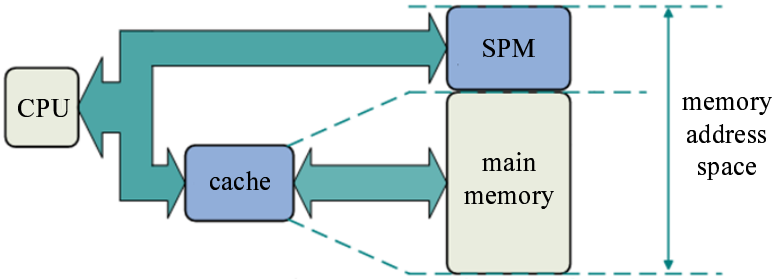
[1] キャッシュを介してアクセスされる主記憶上のメモリアドレス空間の領域．



(a) キャッシュと主記憶



(b) SPM と主記憶

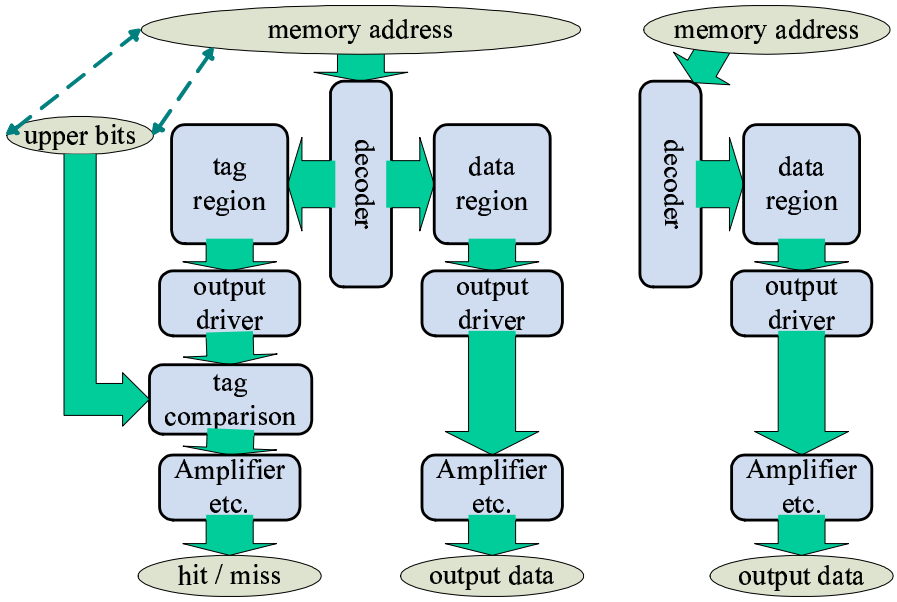


(c) キャッシュおよび SPM と主記憶

図 2.1: キャッシュおよび SPM と外部主記憶とのアドレス空間における関係

主記憶とのアドレス空間上の関係は変わらない．また，一次メモリに SPM のみを用いたシステムでは，キャッシュミスのようなメモリアクセス時に要求される情報が一次メモリに存在しない状況は発生しない．しかし，一次メモリが SPM のみの構成においては，SPM に要求されるデータがない場合（いわば SPM ミス時）には，外部主記憶への 1 ワードずつの直接アクセスが必要となり，実行時間が増大することがある．

図 2.1(c) は，キャッシュと SPM とを組みあわせてシステムの一次メモリを構成した場合を示している．本構成においては，プログラム中のデータの一部は SPM 領域に配置され，残りのデータは外部主記憶のキャッシング可能領域に配置される．図 2.1(c) のような構成であれば，SPM ミス時でもキャッシュに対する 1 サイクルでのアクセスが可能性が残されるため，実行時間および消費エネルギーをより抑えることができる．



(a) キャッシュ (b) SPM

図 2.2: キャッシュおよび SPM の内部構造

次に，SPM がキャッシュよりも消費エネルギーが小さい理由を，ハードウェア構成の違いから解説する．図 2.2(a) がキャッシュのハードウェアの内部構造を，図 2.2(b) が SPM の内部構造を示している．

はじめに，キャッシュのアクセス時において処理される動作を述べる．キャッシュへの読出しアクセス要求が発生すると，まず，キャッシュは，対象メモリアドレスの上位ビットに保持されているインデックス情報から，アクセスすべきキャッシュラインを決定する．次に，そのキャッシュラインのタグ領域部に保持されているタグを読み出す．これとメモリアドレスのタグ情報とを比較して，キャッシュ中に対象メモリアドレスのデータが存在していることかどうかを検出し，キャッシュヒット／ミスを出力する．キャッシュヒットの場合，データ領域内のキャッシュブロックに保持されている所望の情報を読み出し，信号の増幅などの処理を経て，読出し対象のデータを出力する．書込みアクセス時には，タグ領域部へのタグおよびデータ領域への書込みまたは置換の処理などの動作が行われる．

これに対して，SPM へのアクセス時には，データ領域の該当ブロックに保持されている命令またはデータを読み出し，信号の増幅などの処理を経て出力するだけである．キャッシュにおけるタグ領域部の読出しや，メモリアドレスの上位ビットと読出しタグとの比較，および，キャッシュヒット／ミスの出力といった各処理

は不要である．書込み時にも，タグ領域部へのの書込み・置換の処理などの動作は行われない．結果として，1 回のメモリアクセスに要する処理が少なくなり，また，スイッチングを行うトランジスタが少なくなる．以上のことから，SPM は，キャッシュと比較して読出しおよび書込みアクセス時に発生する動的な消費エネルギーは小さくなる．

さらに，図 2.2 をみると，SPM はキャッシュに対して構成要素が少なく済むことが分かる．メモリを構成するモジュールおよびトランジスタ数がキャッシュよりも減少するため，SPM の回路面積は比較して小さくなる．今日の組込み向けプロセッサでは，面積の大半がオンチップメモリで占められている．SPM を用いることによって，メモリの面積が減少されることが期待でき，これもまた SPM の重要な長所であるといえる．そして，SPM は回路素子が少ないため，トランジスタ中で静的に消費されるリーク電力は，キャッシュに比較して小さく抑えられる．

回路の内部構成の違いと前述した主記憶とのアドレス空間の関係から，SPM の利点がもう 1 つ挙げられる．図 2.1(b) や図 2.1(c) のように一次メモリに SPM を用いる構成の場合，あるデータがどのメモリアドレス空間に配置されているかは，メモリアドレスの主に上位ビットを判定することによって判断できる．つまり，それぞれのメモリにアクセスするサイクル数は，システム設計時でも容易に見積もることが可能である．このことは，一次メモリに SPM を含む構成の場合は，プログラムの実行時間が把握しやすくなることを示している．いっぽう，キャッシュへのアクセスは，キャッシュミス時にかかる実行時間のオーバヘッドが不安定でこれを予測することは難しい．また，静的解析時からキャッシュミスがどの程度発生するかを調べることは容易ではない．通常，組込みシステム上で実行されるプログラムは，定められた時間内に処理を完了し，安定したリアルタイム性を保証しなければならない．以上で述べたことより，SPM は組込みシステムにより適したオンチップメモリであることを示している．

本節では，主記憶とのアドレス空間の関係や消費エネルギーの観点から，SPM の有効性を解説した．SPM は，消費エネルギーのみならず，回路面積および実時間性の観点からも，組込みリアルタイムシステムに適したオンチップメモリであるといえる．SPM は，使用目的はキャッシュと同じであるが，その特徴や性質でキャッシュとは異なる．組込みリアルタイムシステムのオンチップメモリを構築する際には，それぞれのメモリの特徴を活かしてキャッシュと SPM とを組み合わせ

ることが重要であると考えられる．

2.2 既存の SPM 管理技術

本論文執筆時点までに，SPM を用いることによって組込みリアルタイムシステムの消費エネルギーまたは実行時間の最適化を実現する既存研究は，数多く行われている．本節では，既存の SPM 管理技術に関する調査結果を述べる．

SPM に関する研究は，まず，シングルタスク環境に適用するものと，マルチタスク環境に適用可能なものとに分類される．さらに，SPM 領域を含めたメモリ領域へのデータおよびコード配置が，静的に行われるものと動的に行われるものとがある．ここで，“配置が静的” とは，プログラム内のデータおよびコードのメモリ領域への配置はシステム動作前に実行され，その配置はシステム実行中に変更されることがないという意味をあらわしている．いっぽう，“配置が動的” とは，SPM に配置されるコードおよびデータの決定はシステム実行前に行われるが，メモリ領域への配置はシステム動作中に行われるという意味である．配置が動的な手法では，SPM の保持する内容がシステム動作中に変更されることがある．

2.2.1 シングルタスク環境における手法

組込みシステムのシングルタスク環境に対して SPM を適用した研究として，文献 [2, 3, 5, 6, 9, 12, 17, 18, 19, 25, 29, 30, 38, 39] が挙げられる．本節では，データおよびコードの配置が静的なものと動的なものに分類して，シングルタスク環境における関連研究の概説を述べる．

静的な手法

文献 [3] では，データに対するメモリ領域割当てのためのコンパイル手法を提案し，プログラムの処理に要する実行時間を最小化することを目標としている．キャッシュのようなハードウェア機構により制御されるメモリは持たない，SPM のようなソフトウェア制御の書込み可能メモリを少なくとも 2 つ持つ組込み向けプロセッサが対象である．各メモリへのデータ配置の決定は，それぞれの書込み可能メモリへの読出しおよび書込みにかかるアクセス時間と，プログラム内のデータのアクセス回数を指標とした 0-1 整数計画問題によって行われる．

文献 [5, 6] は，消費エネルギー最小化を目的として，上記の 0-1 整数計画問題を適用した手法である．SPM のメモリ領域への配置対象は，データ，命令の基本ブロックおよび関数となる．0-1 整数計画問題の指標は，命令またはデータのフェッチに掛かる消費エネルギーとメモリ量になる．

文献 [29] は，文献 [5, 6] を発展させた研究で，いくつか新しい点が挙げられる．まず，複数の連続した命令の基本ブロックまたは関数が SPM に配置される場合には，その間にジャンプ命令を挿入しないようにしている．そして，ある関数が SPM のメモリ領域に配置された場合には，その関数内にある基本ブロックは配置対象として考慮しないという制約を 0-1 整数計画問題に加えている．これらの工夫により，冗長な消費エネルギーの発生要因を除外している．

文献 [5, 6, 29] の研究では，オンチップメモリとして SPM のみを用いた場合のほうが，プログラムの処理に要する実行サイクル数および消費エネルギーを大きく削減できるという結果が得られている．しかし，用いたベンチマークプログラムがメモリの容量よりも小規模なものであったり，主記憶への直接アクセスに掛かるサイクル数がかなり小さく仮定されているなど，疑問な点が多い．

文献 [17] において Ishitobi らは，組込みプロセッサの消費エネルギー最小化を目的としたコード配置の決定を，整数線形計画問題としてモデル化した手法を示している．プログラム中のコード配置対象として，SPM 領域，キャッシング可能領域，および，非キャッシング領域を候補として考慮している．モデル中にキャッシュ・ミス回数の変化が含まれているため，キャッシュ・シミュレーションを繰り返す必要は無い．モデル化された整数線形計画問題の最適解を求めることにより，消費エネルギー最小となるコード配置が一意に定まる．

メモリ領域への命令またはデータ配置に整数計画問題を用いない手法として，文献 [2] や文献 [25]，文献 [38, 39] がある．0-1 整数計画問題は NP 困難な問題であり，最適解を導くための計算時間は指数関数のオーダとなる．このため，問題に対する入力の個数が増えるに従って，データ配置の決定に要する処理時間が爆発的に増加する可能性がある．

文献 [2] は，配置決定のための処理時間を，動的計画法の適用によって短縮することを目指している．具体的には，行方向を配置候補である命令の基本ブロック，列方向を SPM の残り容量としたテーブルを定義し，動的計画法の適用によってコード配置の決定が一方向に再計算することなく行われるようにしている．動的計画法によって，コード配置の決定に掛かる計算時間が多項式関数のオーダに

抑えられている．

文献 [25] では，データ配置の決定に TCF という指標を用いる手法を提案している．本手法によりデータ配置を決定することで，プログラムの実行に要するサイクル数の削減が達成できる．TCF とは *Total Conflict Factor* の略であり，キャッシュ中の競合性ミス[2]の発生頻度を示す値として定義されている．TCF 値が大きいデータは，データキャッシュのメモリ領域に配置されたときに，競合性ミスが多く起きやすいとみなせる．本手法では，TCF 値の特徴に着目し，TCF 値が大きいデータを優先的に SPM 領域に配置している．

文献 [38, 39] において著者らは，命令キャッシュの競合性ミス回数の削減と消費エネルギーの最小化を同時に達成することを目的とした，コード配置決定アルゴリズムを提案している．プログラムの制御フローグラフ（CFG）の枝には，両基本ブロック間で発生しうるキャッシュの競合性ミス回数が重み付けされる．SPM 領域へのコード配置探索時には，枝に付与された重みの削減量が最大となるように基本ブロックを選択していく．

文献 [10] では，対象システムにメモリ変換ユニット（MMU）を用いることによって，SPM のオーバレイを実現している．データアクセス時にメモリ変換を行うことにより，SPM に配置されたデータへのアクセスを実現する．ただし，対象システムにハードウェアの追加が必要となり，アドレス変換にかかるオーバヘッドが発生する．

文献 [14] では，静的データおよびスタックをデータ SPM に配置することにより，組込みプロセッサの消費エネルギー最小化を達成する手法を提案している．特にスタックについては，関数呼び出しまたは終了時にスタックポインタの指すメモリ領域を変更するための命令列を用意し，SPM 領域の効率的な活用を実現している．

動的な手法

SPM の保持する命令またはデータをシステム動作中に動的に更新する操作を，オーバレイという．オーバレイの実現のためには，一般的には，ソフトウェアのソースコードの修正や専用ハードウェアの追加が求められる．

文献 [9] では，第 2.2.1 節の文献 [29] などで提案されている 0-1 整数計画問題に

[2]複数のデータが同じブロックを巡って競合するために発生するキャッシュミス．

対して拡張を加え，オーバレイを考慮した整数計画問題を示している．オーバレイの対象となりうるコードは，プログラムの構造と基本ブロックの実行順序の解析を経て決定される．また，SPM と主記憶間のデータ転送が効率良く行えるよう，転送粒度は複数ワードからなる一定の page 単位で決定するにしている．

文献 [30] は，第 2.2.1 節で紹介した文献 [29, 39] などと同じ研究グループによる研究成果にあたる．この研究では，ARM の LORD/STORE 命令をソースコード中に挿入することによって，SPM 領域への動的なコード配置を実現している．消費エネルギーおよび実行時間の改善を目的として，SPM の保持する命令の基本ブロックまたは関数がシステム実行中に変更される．命令を特別に挿入するのは，オーバレイのタイミングをプロセッサに明示的に通知するためである．ARM の LORD/STORE 命令が実行された際には，オーバレイにはコード転送のためのオーバヘッドが発生する．配置対象となるコードの選択は，このオーバヘッドを考慮したうえで行われる．

SPM と外部主記憶との間のデータ転送には，外部主記憶からの読込み，SPM への書出し，SPM からのデータ読込みといった一連の動作を含む．文献 [12, 18, 19] では，これらの処理を効率良く行う，専用のハードウェアおよび API（Aplication Program Interface）を提供している．

文献 [12] は，データ転送のためのハードウェアとして DMAC（Direct Memory Access Controller）を採用する手法を提案している．DMA とは，プロセッサの定常的な制御なしにデータ転送をメモリ間で直接処理できる転送方式のことをさす．また，DMA を使用すると，プロセッサ上の命令実行とデータ転送とを並列に処理することが可能となる．文献 [12] では，DMA によって主記憶と SPM 間でデータ転送を処理し，プログラム実行時の消費エネルギーおよび実行時間を削減することを目指している．また，DMA の制御を目的とした独自の API を用意している．API の実現する処理には，SPM の初期化および領域確保や，DMAC の処理するジョブの作成，および，プロセッサのストール制御などが定義されている．設計者はこれらの API をソフトウェアに挿入することで，消費エネルギーと実行時間の効率的な削減が実現できるとしている．

文献 [19] および文献 [18] では，独自のハードウェアである SPM controller を用意して，SPM 領域への動的なコード配置を実現している．SPM contoroller とは，SPM と外部主記憶とのコード転送処理を制御するためのハードウェア機構である．また，機構の内部に，BBT（Basic Block Table）と呼ばれる基本ブロックの主記

憶，SPM 双方の開始アドレス，および，基本ブロックのメモリ量の情報を保持するテーブルを持つ．そして，専用 API として，BBT 中の開始アドレスをオペランドにとる SMI（Scratchpad Managing Instruction）命令を用意している．ソースコード中に挿入された SMI 命令が実行されると，プログラムコードが基本ブロック単位で SPM に動的に配置される．

文献 [19]，文献 [18] ともに，コード配置にかかる静的解析の処理時間を削減する工夫を施している．プログラムの基本ブロックをグラフ問題として捉えたアルゴリズムを定義し，本アルゴリズムを実行することで SPM に配置する命令を決定している．文献 [19] では，頂点を基本ブロック，重みをコード量，有向枝を実行順とみなしたグラフを考案している．いっぽう，文献 [18] では，処理時間のさらなる削減のため，頂点に concomitance という重みを付与している．concomitance とは，ある基本ブロックが実行されてから，次にまた実行されるまでに処理される基本ブロックのコード量と定義されている．

2.2.2 マルチタスク環境における手法

マルチタスク環境に適用可能な SPM 活用手法に関する研究として，文献 [11, 20, 26, 31, 32, 37] が挙げられる．

著者の知りうる限り，文献 [37] における研究が，初めてマルチタスク（マルチプロセス）環境に適用可能な SPM 活用手法を示したものである．文献 [37] の対象とする環境は，複数のプロセスがラウンドロビン方式[3] で並列に処理されるものを想定している．文献 [37] において Verma らは，マルチプロセス処理中の消費エネルギー最適化の観点から，処理の効率化の向上を目的として各タスクにいかに SPM の容量を分配して与えるかという問題を提起している．

文献 [37] では，SPM のメモリ領域の分割方針として，(a) Non-Saving, (b) Saving, (c) Hybrid の 3 種類が提案されている．(a) は，SPM 領域を，存在するプロセスごとに完全に分割して与えるものである．割当てられたどの SPM 領域の内容も，システム動作中に変更されることはない．いっぽう，(b) では，SPM 領域はプロセスごとに分割されない．プロセッサ上で処理中のプロセスが，SPM の領域全てを占有して使用できる．プロセスの切替え時間が到来すると，SPM と主記憶の間

[3] 各プロセスの処理を一定時間（タイムスライズ）ずつに順番に行う方式．ラウンドロビン方式では，すべてのプロセスが平等に扱われる．

でデータ転送を行う必要がある．(c) は (a) と (b) を組み合わせた方針である．最も頻繁に実行されるコードおよびデータは (a) の方針により，その次に実行頻度の高いものは (b) の方針により，SPM 領域を各プロセスへ割当てる．

しかし，文献 [37] の研究では，ラウンドロビン・スケジューリング規則に従って，複数のタスクを並行して処理する時分割システムが前提となっている．また，この手法では，消費エネルギー最小となる SPM 容量の分配は，全数探索を用いるアルゴリズムによって決定される．このため，アルゴリズムへの入力数によっては，決定のための探索時間が膨大になる可能性がある．

文献 [11, 26] では，文献 [37] と同様の環境において SPM の保持する内容を実行時に管理することができる機構を提案している．本機構は，スケジューラとディスパッチャのみからなる小規模なリアルタイムカーネル上に実装される．SPM への配置対象となるデータは，システム実行前のプロファイル処理によってある程度選択される．ただし，メモリ領域の配置は，本機構の制御によりシステム実行中に決定される．また，実行時の配置決定の支援のため，リアルタイム OS が操作するデータ構造も提案している．プロファイル処理を選択発見的なアルゴリズムとして定義することにより，設計時の前処理時間の削減も達成している．

文献 [32] における研究では，複数のタスクで構成されるアプリケーションに SPM を導入し，システム実行時の最悪応答時間（WCRT: Worst Case Response Time）の改善を実現する手法を示している．タスク間の同期・通信やタスクごとの存在期間を考慮し，ワークフローの繰返しによる学習解析で目標を達成している．

文献 [20] や文献 [31] のように，組込みシステムのマルチプロセッサ環境においても効率的な SPM の活用手法を提案した研究もいくつか発表されている．

文献 [20] は，組込みシステムのマルチプロセッサ環境において SPM を適用した手法を示している[4]．対象とするアーキテクチャでは，プロセッサごとに SPM をそれぞれ 1 つずつ持つ．プロセッサは，自身の SPM に加え，他のプロセッサの SPM へ複数サイクルを掛けてアクセスすることができる．SPM 中のデータの再利用性を高めることで，システム全体の消費エネルギー最小化を図る．

Suhendra らは，文献 [31] において，マルチプロセッサ環境でのタスク割付けお

[4]文献 [20] は，大規模な配列計算からなる単一タスクのアプリケーションを対象としている．しかしながら，本手法はマルチプロセッサ環境を対象としており，さらに，マルチタスク環境にもそのまま適用可能な手法であることから，第 2.2.2 節に分類している．

よびSPMへのデータ配置の双方の最適化によって，システム全体の実行時間を削減する手法を提案している．手法の手順としては，まず，タスクグラフを基にして各プロセッサへのタスク割付けを決定する．タスクのスケジューリングとパイプライン化も，これと同時に定める．次に，プロセッサごとにSPMの容量を分配し，さらに実行時間最適なデータ配置も同時に決定される．

以上で述べたように，SPMを用いることによって組込みリアルタイムシステムの消費エネルギーおよび実行時間を最小化する手法は，これまでに数多く提案されている．しかし，著者の知りうる限り，リーク電力を含めてSPMの有効性を議論した研究や，高いリアルタイム応答性が要求されるリアルタイムシステムにおいて，消費エネルギー最適であり，かつ，実用性の高いSPM管理技術を確立したものは存在しない．

第3章　リーク電力を考慮したSPMの有効性の評価

3.1　概要

第 2.2 節で取り上げたように，SPM を組込みシステムに適用することによって消費エネルギーを削減する手法は，これまでに数多く提案されている．しかし，それらの研究の多くは，メモリの動的な消費エネルギーのみを評価しており，静的な消費エネルギーであるリークエネルギーは評価対象として考慮されていない．メモリシステムにおける消費エネルギーを見積もる際には，メモリアクセス時の動的な消費エネルギーだけでなく，様々な要因や諸条件によって変動する静的な消費エネルギーであるリークエネルギーも含めて評価を行うことが重要である．

ここで，メモリアクセスに要する動的な消費エネルギーと，プログラム実行中に静的に消費されるリークエネルギーの違いについて説明する．

動的な消費エネルギーとは，組込みシステム上のプログラム実行中に，メモリのゲート出力が 0 から 1 に，あるいは，1 から 0 に変化するときに消費されるエネルギーである．動的な消費エネルギーは，トランジスタのスイッチング（負荷容量の充放電）によって発生するものが大部分を占めている．これに対して，リークエネルギーは，スイッチングの動作に関わらずトランジスタ中に漏れ流れる電流に起因して発生する消費エネルギーである．リークエネルギーは，プログラムの実行とは関係なく，システム電源の投入中にトランジスタで定常的に消費される．半導体設計技術の微細化が進む昨今のメモリデバイスでは，リーク電力の消費量は増大の一途を辿っているといわれている．

CMOS デバイスにおけるリーク電流をモデル化した式として有名な HotLeakage [43] によれば，リーク電流 $I_{leakage}$ は以下の式で定義される．

$$I_{leakage} = \mu_0 \cdot C_{OX} \cdot \frac{W}{L} \cdot e^{b(V_{dd}-Vdd0)} \cdot v_t^2 \cdot \left(1 - e^{\frac{-V_{dd}}{v_t}}\right) \cdot e^{\frac{-|V_{th}|-V_{off}}{n \cdot v_t}} \tag{3.1}$$

ここで，V_{th} は閾値電圧をあらわす．

上式の通り，リーク電流は，様々な要因や諸条件の変動から影響を受ける．とくに，リーク電流は $e^{-|V_{th}|}$（e は自然対数の底）に比例する．CMOS デバイスの微細化に応じて電源電圧を低下させることができるため，動的な消費エネルギーは小さくなる．いっぽうで，メモリの性能の確保と CMOS テクノロジの微細化によるデバイス間の電子的な干渉を防ぐために，閾値電圧は低く設定する必要がある．しかしながら，(3.1) 式に示したとおり，CMOS テクノロジの微細化に伴う閾値電圧の低下は，リーク電流の指数関数的な増加を招くことにつながる．近年，プロセッサの集積度と性能の向上を大きな目的として，組込み向けプロセッサのますますの微細化が進んでいる．

リーク電力は，近年進むデバイス技術の微細化によって全体の消費電力に占めるその割合が顕著になってきており，もはや，無視できるほど小さい量ではないといわれている．リーク電力の爆発的な増大を引き起こすデバイスの微細化は，性能向上の要求から今後も次々に進んでいくとみられる．このことから，リーク電力を含めて消費エネルギーを見積もることの価値は，非常に高くなってくる．

本章では，キャッシュと SPM の双方の利点を活かし，これらを組み合わせて組込みシステムの一次メモリを構築することの有効性を，消費エネルギーの観点から評価する．評価実験の際には，メモリアクセスの動作に依らず静的に消費されるリーク電力も評価対象とする．実験では，命令メモリのみを評価対象とする．プログラムコードのアドレス領域への配置粒度は，プログラムの関数単位とする．過去にも，メモリアクセス時の動的な消費エネルギーのみを評価して，SPM の有効性を主張する研究は数多く行われてきたが，リーク電力も含めた評価を行ったものはほとんど見られない．研究着手時における最先端，次世代，および，次々世代の半導体設計技術によって製造されたメモリシステムを想定して評価実験を行い，組込みシステムにおける SPM の有効性を議論する．

3.2　評価手順

本節では，組込みシステムの命令メモリで消費されるエネルギーを導出するための手順を示す．システムの一次メモリは，キャッシュと SPM を組み合わせて構築する．評価実験では，デバイスの微細化によって変化する消費エネルギーの傾向を把握することを目的としている．このため，動的な消費エネルギーだけでなく，静的に消費されるリークエネルギーも含めた評価を行う．

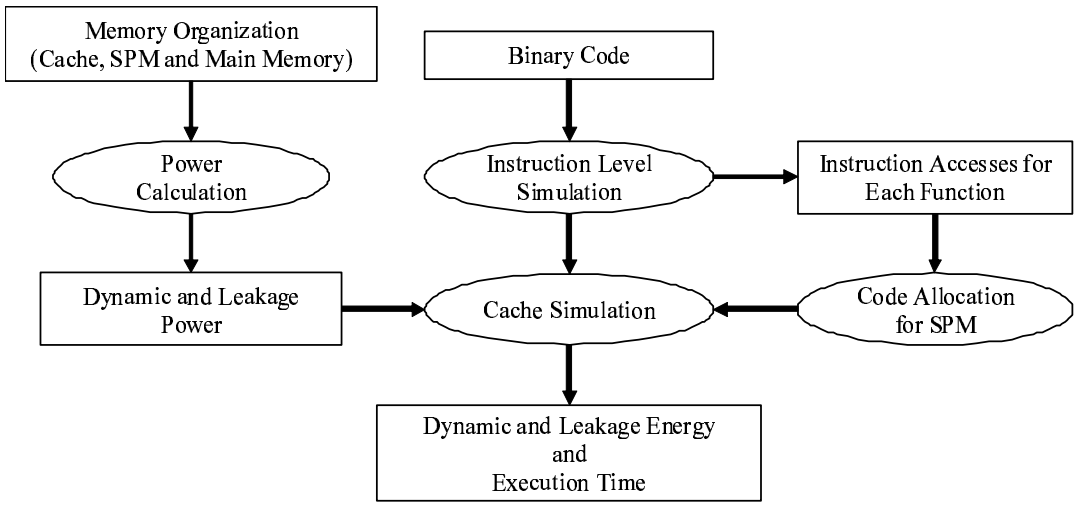


図 3.1: リーク電力を考慮した評価実験のワークフロー

3.2.1　評価環境

図 3.1 に，本研究において行った評価実験の流れを示す．

キャッシュ，SPM および主記憶のそれぞれのメモリで消費されるエネルギーは，CACTI 5.0 [36, 40] の出力値を基にして導出した．CACTI とは，メモリの容量やデバイスサイズ，動作温度などを入力値として与えることによって，各メモリのアクセス 1 回当たりの動的な消費エネルギーやシステム動作中のリーク電力，回路面積といった数値を出力するシミュレーションツールである．

ベンチマークプログラムは，MiBench [15] から表 3.1 に示す 6 種類のプログラムを選んだ．実験には，C 言語で記述されているそれぞれのプログラムを，マイクロプロセッサシミュレータ SimpleScalar/ARM [61] 上で実行可能なバイナリコードにコンパイルしたものを用いた．SimpleScalar/ARM は，幅広い用途で採用されている組込みプロセッサのひとつである ARM7TDMI [48] の命令セットに準拠した ISS（命令セットシミュレータ）である．

ベンチマークプログラムの各バイナリコードに対して，命令セットに準拠したアセンブリ言語への逆アセンブル，および，SimpleScalar/ARM によるトレース実行の各処理を行った．これらの処理により，それぞれの対象プログラムに含まれる各関数のメモリ量と実行命令数を取得した．

各ベンチマークプログラムに含まれる関数の個数および総メモリ量を，表 3.1 中に示す．表 3.1 中に記載してある数値は，プログラムの実行中に関数呼出しがあり，実行処理が行われたものである．ソースコード中に宣言・定義されているが，プロ

表 3.1: リーク電力を考慮した評価実験で用いたベンチマークプログラム

プログラム	概要	関数の個数	関数の総メモリ量（Bytes）
bitcnts	5 種のビットカウンタ	123	44476
cjpeg	JPEG 画像の符号化	248	52008
djpeg	JPEG 画像の復号化	225	64412
dijkstra	グラフの最短経路探索	115	50344
patricia	パトリシア木の生成	137	76492
ispell	Web 文書の綴り検査	163	60196

グラム実行中に一度も実行処理されることのない関数は，アドレス領域の配置対象に含まれないことが明らかである．

3.2.2　消費エネルギーの評価式

本節で記す評価式を用いて，組込みシステムのメモリ全体で要する消費エネルギーを導出した．

まず，メモリ全体の消費エネルギー E_{total} は，次に示す式であらわされる．

$$E_{total} = E_{Cache} + E_{SPM} + E_{MM} \tag{3.2}$$

ここで，E_{Cache}，E_{SPM} および E_{MM} は，キャッシュ，SPM および主記憶で消費される全体のエネルギーをあらわす．それぞれのメモリにおける消費エネルギー値は，

$$E_{Cache} = E_{C\_dyn} + E_{C\_lkg} \tag{3.3}$$

$$E_{SPM} = E_{S\_dyn} + E_{S\_lkg} \tag{3.4}$$

$$E_{MM} = E_{MM\_dyn} + E_{MM\_stby} \tag{3.5}$$

となる．ここで，E_{C_dyn}，E_{S_dyn} および E_{MM_dyn} は，キャッシュ，SPM および主記憶の動的なエネルギーをあらわす．E_{C_lkg} および E_{S_lkg} はキャッシュおよび SPM のリークエネルギーであり，E_{MM_stby} は主記憶のスタンバイエネルギーをあらわす．

E_{C_dyn} は，キャッシュヒット時の消費エネルギーの合計 E_{C_hit} と，キャッシュミ

ス時の消費エネルギーの合計 E_{C_miss} との和として求める．つまり，

$$E_{C\_dyn} = E_{C\_hit} + E_{C\_miss} \tag{3.6}$$

$$E_{C\_hit} = E_{C\_read} \times N_{C\_hit} \tag{3.7}$$

$$E_{C\_miss} = (E_{C\_read} + E_{C\_write}) \times N_{C\_miss} \tag{3.8}$$

となる．ここで，E_{C_read} および E_{C_write} は，キャッシュにおいて，読出しおよび書込みアクセスを 1 回行うときに消費される動的なエネルギーである．N_{C_hit} および N_{C_miss} は，プログラム実行中に発生するキャッシュヒットおよびキャッシュミスの回数をあらわす．

いっぽう，SPM の消費エネルギーは，

$$E_{S\_dyn} = E_{S\_read} \times N_{S\_hit} \tag{3.9}$$

となる．E_{S_read} は，SPM における読出しアクセス 1 回当たりの動的な消費エネルギーであり，N_{S_hit} は，SPM のアドレス領域にアクセスする回数をあらわす．

主記憶の動的な消費エネルギーは，バーストアクセスの有無によって場合分けされる．一次メモリにキャッシュを用いるかどうかによって，

$$E_{MM\_dyn} = \begin{cases} E_{MM\_read\_burst} \times N_{C\_miss} \\ \qquad \text{(キャッシュを用いるとき)} \\ E_{MM\_read\_random} \times N_{S\_miss} \\ \qquad \text{(キャッシュを用いないとき)} \end{cases} \tag{3.10}$$

となる．ここで，$E_{MM\_read\_burst}$ は，主記憶にバーストアクセスを 1 回行って複数命令分を一度に読み出すときに消費されるエネルギーをあらわす．$E_{MM\_read\_random}$ は，主記憶へのランダムアクセスにより 1 命令分だけを読出すときの消費エネルギーである．N_{S_miss} は，一次メモリが SPM のみで構成されるときに，主記憶のアドレス領域へ直接アクセスする回数をあらわす．

リークエネルギーは，リーク電力と実行時間の積により導出される．すなわち，

$$E_{C\_lkg} = P_{C\_lkg} \times CC_{total} \times \frac{1}{f} \tag{3.11}$$

$$E_{S\_lkg} = P_{S\_lkg} \times CC_{total} \times \frac{1}{f} \tag{3.12}$$

として求める．ここで，CC_{total} はプログラムの実行サイクル数を，f はプロセッサの周波数をあらわす．これは，CACTI 5.0 は，リークエネルギーの出力をサポー

トしておらず，代わりにキャッシュおよび SPM のリーク電力値をあらわす P_{C_lkg} および P_{S_lkg} が出力されるためである．

E_{MM_stby} も，CACTI 5.0 から得られるスタンバイ電力 P_{MM_stby} によって，

$$E_{MM\_stby} = P_{MM\_stby} \times CC_{total} \times \frac{1}{f} \tag{3.13}$$

として求められる．

3.2.3　SPM への関数配置

本研究では，プログラムコードのアドレス領域への配置は静的であり，コードの配置粒度は関数単位とする．すなわち，SPM のアドレス領域に配置される関数は，システム動作前に静的に決定され，プログラム実行中に SPM の内容は更新されない．文献 [3] を参考にして，SPM のアドレス領域に配置される関数の決定方法は，以下に示すナップサック問題として定義した．

［定義］

プログラムに含まれる i 番目の関数を $func_i$，そのメモリ量を $C(func_i)$，$func_i$ の実行に要するメモリアクセス回数を $N(func_i)$ とする．x_i は次式で定義される 0-1 変数とする．

$$x_i = \begin{cases} 1 & (func_i \text{が SPM のアドレス領域に配置されるとき}) \\ 0 & (\text{それ以外}) \end{cases} \tag{3.14}$$

このとき，次式における変数 $gain$ の値が最大となるような x_i の値の組を最適解として求める．

$$gain = \sum_i N(func_i) \times x_i \tag{3.15}$$

ただし，SPM の容量が S であるとき，x_i は

$$\sum_i C(func_i) \times x_i \leq S \tag{3.16}$$

を満たす値の組である．

本研究では，上で定義したナップサック問題の最適解を得るために，ILP ソルバ lp-solve 5.6 [8] を用いた．これは，NP 困難である問題の最適解を導出するアルゴリズムのうちの分枝限定法を実装したソルバである．これを用いることにより，想定するメモリ構成の，SPM の容量ごとに最適となる関数のアドレス領域配置を決定した．

表 3.2 に，それぞれの SPM 容量に対応した関数配置の結果を示す．4 列目の “全命令に占める割合 (%)” は，SPM に配置される関数の実行命令数の，プログラム実行中の総命令数に占める割合を示している．SPM の各容量は，第 3.2.4 節で述べる手順によって定めている．

3.2.4　メモリの構成

本実験において想定したメモリの構成を述べる．

まず，一次メモリの容量について述べる．一次メモリとして用いるキャッシュと SPM の合計の容量は，対象プログラムで実行されるコードの総メモリ量の 20 % とした．つまり，第 3.2.1 節の表 3.1 より，それぞれのベンチマークプログラムに対する一次メモリの総容量は 8192 Bytes と定められる．

キャッシュは，その容量と連想度の変更が可能であるコンフィギュラブル・キャッシュを想定した．1 ウェイの容量が 1024 Bytes，最大で 8 ウェイ（すなわち 8192Bytes）まで選択可能なキャッシュである．キャッシュのラインサイズは 32 Bytes で一定とした．SPM の容量は，キャッシュの容量に応じて一次メモリの総容量が 8192 Bytes となるように定めた．例えば，キャッシュを 2048 Bytes（2 ウェイ）とすれば，SPM の容量は 6144 Bytes と定まる．主記憶は DRAM を想定した．DRAM の容量は 2 MBytes，バンク数は 64 個の構成とした．

以上で述べたメモリの構成を，CACTI への入力値として与えた．これによって，それぞれのメモリのアクセス 1 回当たりの動的な消費エネルギーおよびリーク電力の値を得た．

次に，メモリのアクセスに掛かるサイクル数について説明する．キャッシュおよび SPM にアクセスし，そこから 1 命令を読み出すためには 1 サイクルかかると仮定した．実行対象のプログラムコードが SRAM 中に保持されておらず，DRAM にアクセスするときのサイクル数は，バーストアクセスの有無によって異なる．まず，キャッシュミスが発生し，DRAM からバースト転送を行ってキャッシュ中の 1

表 3.2: SPM のアドレス領域への命令配置

プログラム名	SPM の容量 (Bytes)	SPM に配置される関数の実行命令総数	全命令に占める割合（%）
bitcnts	4096	743659874	99.997
	6144	743663834	99.998
	7168	743665191	99.998
	8192	743666243	99.998
cjpeg	4096	86285808	82.484
	6144	101246910	96.786
	7168	103014934	98.477
	8192	104035305	99.452
djpeg	4096	22418556	95.837
	6144	23226581	99.292
	7168	23285508	99.543
	8192	23327182	99.722
dijkstra	4096	262681045	96.342
	6144	266221680	97.641
	7168	266608405	97.783
	8192	267738061	98.197
patricia	4096	331366865	51.742
	6144	381453289	59.563
	7168	394905959	61.664
	8192	403917705	63.071
ispell	4096	725282707	88.695
	6144	767737084	93.887
	7168	781021951	95.512
	8192	788352865	96.408

ラインの内容を更新する場合は，18 サイクルかかると仮定した．いっぽう，SPM のみで一次メモリを構築する場合は，DRAM へのランダムアクセスによって，プログラムコードを 1 命令ずつ読み出すことになる．このときのサイクル数は，1 回のアクセスにつき 4 サイクルであるとした．以上の値は，CACTI 5.0 から得られる出力のひとつであるメモリのアクセス時間にしたがって定めた．

3.3　評価

3.3.1　45 nm デバイスでの評価

まず，45 nm デバイスのメモリシステムで評価を行った．本節では，45 nm デバイスでの評価実験の結果と考察を述べる．

図 3.2 に，メモリのデバイスサイズを 45 nm とした場合の実験結果を示す．プロセッサの周波数を 1.00 GHz，システムの動作温度を 27 ℃とし，この設定のもとでの CACTI から動的な消費エネルギーおよびリーク電力を導出した．図 3.2 に示した消費エネルギー値および実行時間は，キャッシュ8192 Bytes のみを一次メモリとして用いた場合を基準に正規化した値である．グラフの縦軸の第 1 軸（左側）が消費エネルギー値に，第 2 軸（右側）が実行時間にそれぞれ対応している．横軸の ‘C xKB’ および ‘S xKB’ は，それぞれ，キャッシュおよび SPM の容量をあらわしている．図 3.2 中の棒グラフの ‘SRAM Dynamic’ および ‘SRAM Leakage’ は，キャッシュと SPM の動的なエネルギーおよびリークエネルギーの消費量の合計をあらわす．‘Main Memory Dynamic’ および ‘Main Memory Standby’ は，主記憶の動的なエネルギーおよびスタンバイエネルギーである．DRAM ではリフレッシュ電力が消費されるが，これは ‘Main Memory Standby’ に含めて結果を導出している．

本実験に対する考察を述べる．

まず，一次メモリがキャッシュのみの場合と，キャッシュと SPM の両方を用いる場合とを比較する．図 3.2 に示す各プログラムの平均値をみると，キャッシュのみで一次メモリを構成するよりも，一次メモリに SPM を組み合わせることにより，メモリ全体の消費エネルギーを 28.8〜31.5 %削減することができている．これは，同じ SRAM ベースのキャッシュよりも，動的なエネルギー消費量を小さく抑えら

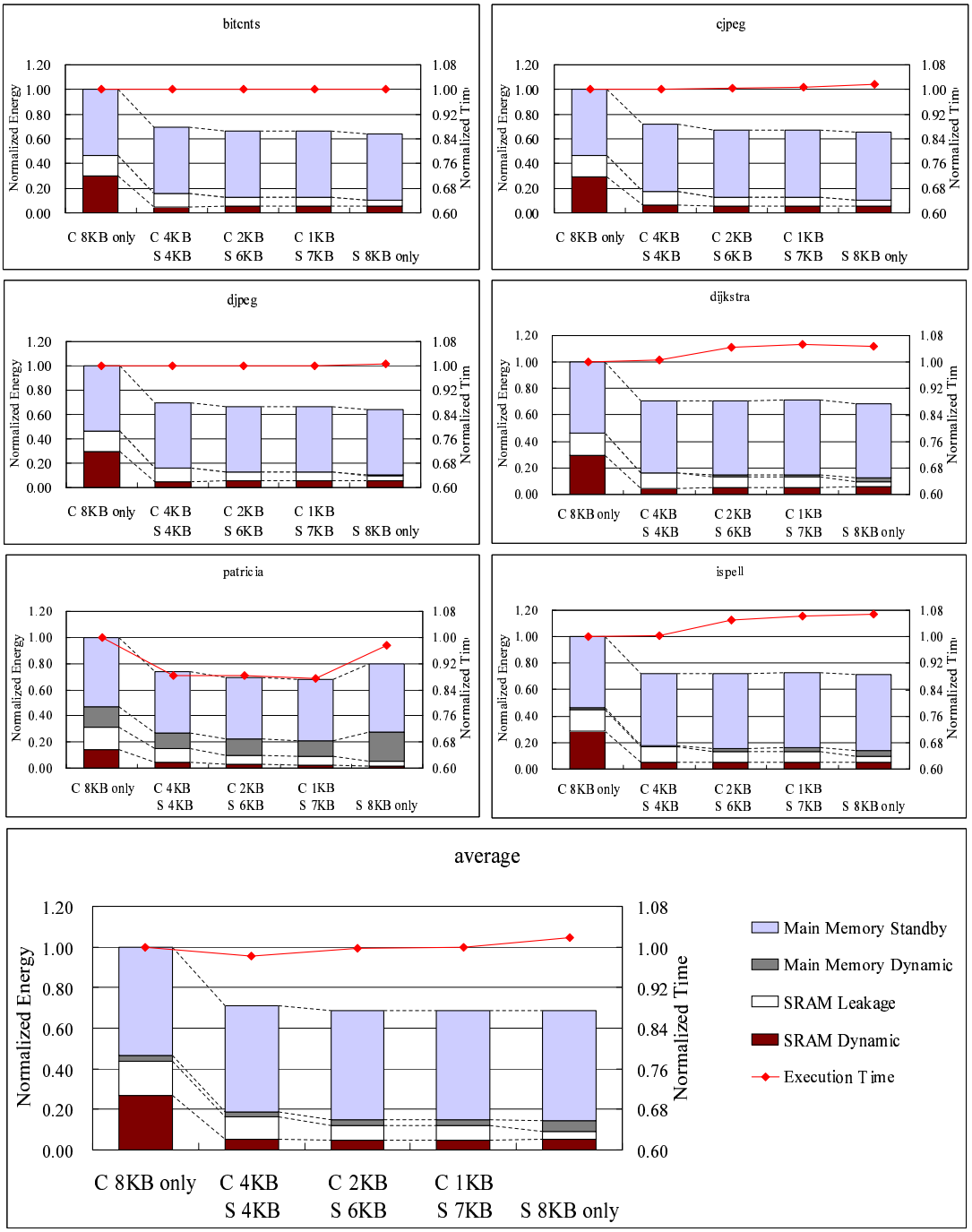


図 3.2: リーク電力を考慮した 45 nm メモリデバイスでの実験結果

れるSPMへのアクセスが多くなったためである．また，SPMを用いることによって，アクセス時の動的な消費エネルギーが大きい主記憶へのアクセス回数を最小化できることも理由として挙げられる．

次に，SPMのみの場合と，キャッシュをSPMと組み合わせて用いる場合をみる．プログラム “patricia” では，キャッシュとSPMを組み合わせたメモリ構成のほうが，SPMのみの場合よりも，消費エネルギーを7.5〜15.5 %削減することができている．プログラム “patricia” においては，表3.2から分かるとおり，SPMに配置される命令数の総実行命令数に占める割合は小さい．このような状況で，かつ，一次メモリがSPMのみであれば，動的な消費エネルギーの大きい主記憶から1命令ずつ読み出す回数が多くなってしまう．また，実行時間が長くなり，その結果，リークエネルギーも増大することになる．以上より，第3.2.3節において定義したナップサック問題の最適解における $gain$ が小さく，SPMに配置される実行命令数の割合が総実行命令数に対して小さい状況では，キャッシュとSPMを組み合わせて一次メモリを構成することが，消費エネルギーの最小化につながるといえる．

“patricia” 以外のプログラムでの実験結果をみる．これらのプログラムは，SPMに配置される実行命令数の割合が総実行命令数に対して大きい状況であるととらえられる．これらのプログラムでは，SPMのみで一次メモリを構成する場合のほうが，メモリ全体で消費されるエネルギーは小さくなくなる．しかし，グラフ中の折れ線で表現している実行時間 ‘Execution Time’ は，SPMのみの場合よりも，キャッシュとSPMを組み合わせた場合のほうが小さくなっている．ただし，消費エネルギーと実行時間のどちらも，その差はかなり小さいものである．

3.3.2　異なるデバイスサイズでの比較

本節における実験では，前節の45 nmデバイスに加えて，65 nmおよび32 nmのものについてもメモリ全体の消費エネルギー値を導出した．メモリデバイスの差異を反映させるため，動作周波数はCMOSサイズごとに正規化して定めることとする．具体的には，CACTI 5.0から得られる8ウェイ 8192 bytesのキャッシュのアクセス時間を指標とした．45 nmデバイスのキャッシュのアクセス時間およびプロセッサの動作周波数（1.00 GHz）を基準にして，デバイスのサイズごとにプロセッサの動作周波数を定めた．これによって，メモリのデバイスサイズの違いを結果に正しく影響させることができる．システムの動作温度は，デバイスのサイズに依らず，第3.3.1節と同じく27 ℃とした．

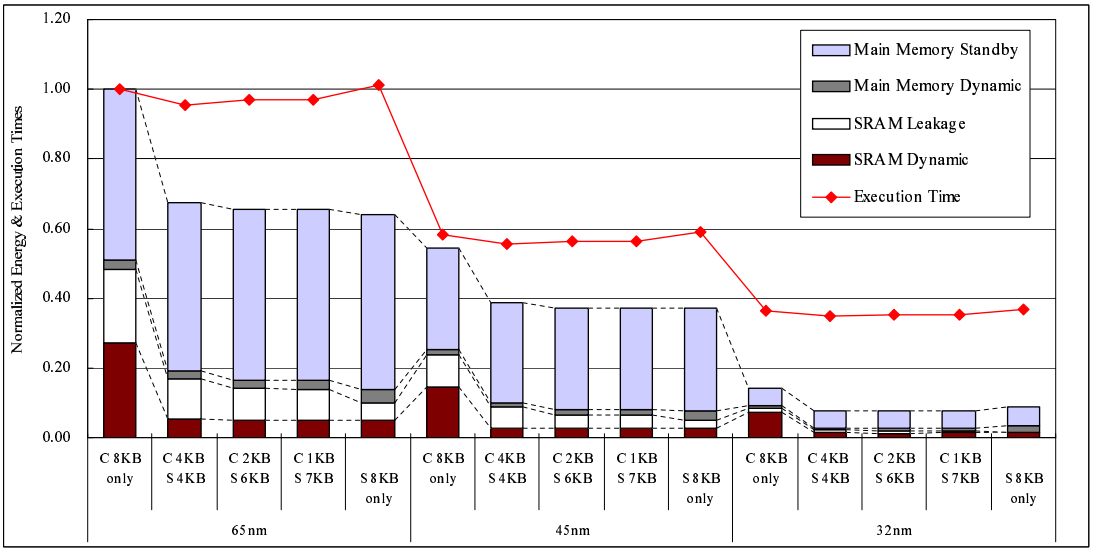


図 3.3: リーク電力を考慮したプログラム “ispell” の実験結果

本評価実験の結果と考察を述べる．

図 3.3 に，プログラム “ispell” による実験結果を示す．65 nm，45 nm，および，32 nm デバイスにおいて，キャッシュ単独の場合での一次メモリ中のリークエネルギーの割合は，それぞれ，42.0 %，36.8 %，および，12.8 %である．このことから，微細化が進んだデバイスサイズの小さいメモリにおいては，消費量が大きいリーク電力を見積もり，その影響を把握することの価値は高いといえる．なお，32 nm デバイスにおいてリークエネルギーの割合が小さくなるのは，アクセス時間およびプロセッサの動作周波数の高速化によって，プログラムの実行時間が短くなったことが関係していると考えられる．

図 3.3 をみると，どの世代のデバイスのメモリでも，第 3.3.1 節で述べたものと同じような考察がいえる．図 3.3 ではプログラム “ispell” のみを取り上げているが，他の 5 種類のプログラムでも同様の傾向が得られている．このことから，組込みシステムに SPM を用いることは，デバイスのサイズに依らず効果が高いものであるといえる．また，デバイスの微細化が将来さらに進んでいったとしても，組込みシステムにおける SPM の有効性は，依然として変わらないであろうと予測できる．

最後に，主記憶の消費エネルギーについて触れる．

今回の実験では，主記憶に DRAM を想定し，その動的な消費エネルギーとスタンバイエネルギーを評価対象とした．このとき，どのデバイスサイズでも，DRAM

のスタンバイエネルギーが全体の消費エネルギーに対して支配的になるという結果が得られた．一次メモリとしてキャッシュおよびSPMを適用したため，DRAMへアクセスする回数は最小限となっている．このため，DRAMの動的な消費エネルギーはかなり小さく抑えられている．以上のことより，組込みシステムにSPMを適用して，かつ，主記憶における静的な消費電力の管理を行う技術の導入が必要となってくるといえる．

3.4　まとめ

本章では，組込みシステムの一次メモリにSPMを適用することの有用性を，消費エネルギーの観点から評価した．評価実験では，デバイスの微細化によって消費量が顕著になっているリークエネルギーも評価対象として考慮した．

まず，研究着手時における最先端の半導体製造技術のCMOSテクノロジにより設計された45 nmデバイスのメモリを想定して実験を行った．その結果，SPMを用いることによって，キャッシュ単独で一次メモリを構成するよりも，メモリ全体の消費エネルギーを平均で28.80〜31.51 %削減することができた．次に，同様の評価実験を，90 nm，65 nmおよび32 nmデバイスのメモリに対しても行った．その結果，デバイスのサイズが異なっても，メモリの構成ごとの消費エネルギー量にあまり変化はみられないという傾向が得られた．

以上のことから，リーク電力を考慮しても，組込みシステムにSPMを用いることの有効性は高いと結論付けられた．さらには，将来，デバイスの微細化がさらに進んでいっても，組込みシステムにSPMを用いることの優位性は，変わらず高いものであるという予測が得られた．本研究において得られたこれらの知見は，組込みシステムにおけるオンチップメモリの半導体製造技術がCMOSによる限り，有効であると考える．

第4章　非プリエンプティブなマルチタスク環境におけるSPM領域分割法

4.1　概要

大規模・複雑化が進む近年の組込みシステムでは，複数のタスクを並行して処理することが要求される．このようなシステムでは，固定優先度ベースのタスクスケジューリング方式が採用されるのが一般的である．しかしながら，第 2.2.2 節で紹介した文献 [26, 37] のように，時分割システム向けの消費エネルギーの削減を目的とした SPM 活用技術の研究はなされているものの，固定優先度に基づくスケジューリング方式に対応できる SPM 管理技術に関する研究は，現在までにほとんど行われていない．

本章では，非プリエンプティブなマルチタスク環境における SPM 管理技術に関する研究の成果を報告する．組込みシステムの命令メモリに SPM を活用し，限られた SPM 領域をタスク間で効率良く分割利用することで，消費エネルギーの最小化を目指す．本研究では，固定優先度付きの周期タスクが複数存在する組込みシステムに対応した SPM 領域分割手法を提案する．本章において提案する SPM 領域活用方針は，空間分割法（第 4.3.1 節），時間分割法（第 4.3.2 節），および混合分割法（第 4.3.3 節）の 3 種類である．各方針における SPM 領域分割の決定は，システムの命令メモリの消費エネルギー削減量の最大化を目的関数とした整数計画問題に帰着させる．タスクの起動周期を考慮して定式化した整数計画問題の最適解を導出することにより，タスクごとに使用する SPM 領域のメモリ量および SPM 領域に配置するプログラムコードが同時に決定できる．評価実験を行い，提案手法の有効性を示す．

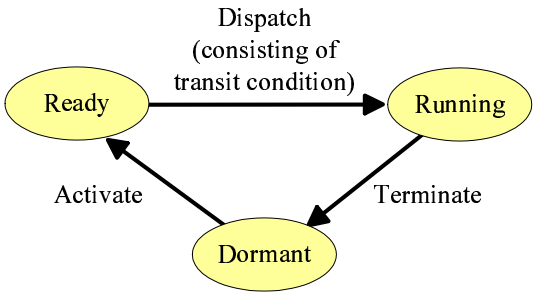


図 4.1: 非プリエンプティブなマルチタスク環境におけるタスクの状態遷移

4.2　準備

4.2.1　対象とするシステム構成

本研究の対象とする環境は，処理すべきタスクが複数存在するマルチタスクシステムである．タスクは，図 4.1 に示すように，休止，実行可能，および実行の 3 状態をとる．プリエンプションは発生しないため，実行状態から実行可能状態に遷移することはない．つまり，タスクがいったん実行状態に遷移したら，その処理が終了するまで実行中断することはない．タスク間の通信・同期処理はなく，各タスクは独立して動作する．また，全てのタスクは周期的に実行され，起動周期はシステム動作前に静的に決定される．

タスクの実行順序は，非プリエンプティブな固定優先度ベースのスケジューリング方式に従って決定されるとする．CPU 使用権が解放された時点で，実行可能状態にあるうちで最高優先度のタスクが CPU 使用権を得て実行状態に遷移する．

4.2.2　変数の定義

表 4.1 にて，本章における整数計画問題で用いる変数を定義する．表 4.1 に示した変数の値は，静的な解析によってすべて取得可能である．

4.3　SPM 領域分割法

本節では，非プリエンプティブなマルチタスク環境において，SPM の有効な活用を実現する領域分割手法の詳細を述べる．提案する手法は，空間分割法，時間分割法，および混合分割法の 3 種類である．各方針におけるタスクごとの SPM 領

表 4.1: 非プリエンプティブなマルチタスク環境における変数定義

変数名	定義
$task_i$	i 番目のタスク．$1 \leq i \leq M$（M はタスク数）．
$period_i$	$task_i$ の起動周期
$func_{i,j}$	$task_i$ の j 番目の関数．$1 \leq j \leq N$（N は関数の個数）．
$fetch_{i,j}$	$func_{i,j}$ の命令フェッチ総数
$size_{i,j}$	$func_{i,j}$ のコードサイズ
E_{Cache_read}	キャッシュへの 1 回の読出しアクセスに要する消費エネルギー
E_{SPM_read}	SPM への 1 回の読出しアクセスに要する消費エネルギー
E_{SPM_write}	SPM への 1 回の書込みアクセスに要する消費エネルギー
E_{MM_read}	主記憶への 1 回の読出しアクセスに要する消費エネルギー
$Esaving_{i,j}$	$func_{i,j}$ を SPM 領域に配置することにより削減できる消費エネルギー
$Eoverhead_{i,j}$	$func_{i,j}$ を主記憶から SPM にコピーするための消費エネルギー
$SPMsize$	SPM の総容量
$hyperperiod$	全タスクの起動周期の最小公倍数

域分割およびコード配置の決定は，消費エネルギー削減量の最大化を目的関数とした整数計画問題に定式化する．なお，本研究では，プログラムコードのみをメモリ領域への配置対象とし，配置粒度は関数単位とする．

4.3.1　空間分割法

図 4.2 に示すように，空間分割法は，SPM の領域を各タスクに排他的に分割して与える手法である．タスクは与えられた SPM 領域を，タスクセット実行中に占有して使用する．タスク数 3 の図 4.2 においては，SPM のアドレス空間を 3 領域に分割して各タスクに与えている．タスクごとに使用する SPM 容量および SPM 領域への関数配置は，すべて静的に決定される．SPM の保持する内容は，システム動作中に変更されない．

空間分割法では，タスクの起動周期を情報として用いることにより，実行頻度の高いコードが SPM 領域により多く配置されるようにする．これにより，SPM のさらなる有効活用を実現する．起動周期の短いタスクの関数は実行頻度が高くなるため，与えられる SPM 領域は大きくなる．例えば，図 4.2 では，周期最小の ‘Task1’ は，SPM 領域を多く占有している．

それぞれのタスクに割当てる SPM の容量と，タスクごとの SPM 領域への関数

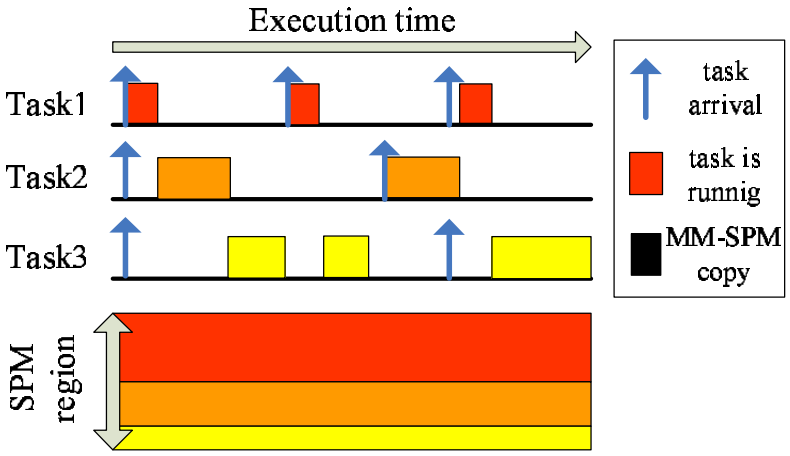


図 4.2: 非プリエンプティブなマルチタスク環境における空間分割法

配置の決定は，次に示す整数計画問題として定式化した．関数の命令フェッチ総数にタスクの起動周期を付与することにより，SPM を用いるによって削減できる消費エネルギーの総量 $Esaving$ の最大化を目指している．

$$\textbf{Maximize} : Esaving = \sum_i \sum_j Esaving_{i,j} \times x_{i,j} \tag{4.1}$$

$$Esaving_{i,j} = fetch_{i,j} \times \frac{hyperperiod}{period_i} \times E_{SPMgain} \tag{4.2}$$

$$E_{SPMgain} = E_{Cache\_read} - E_{SPM\_read} \tag{4.3}$$

$$\textbf{s.t.} : \sum_i \sum_j size_{i,j} \times x_{i,j} \leq SPMsize \tag{4.4}$$

ここで，式中の決定変数 $x_{i,j}$ は，$func_{i,j}$ が SPM 領域に配置されるときに 1 となる 0-1 変数である．この解集合を求めることにより，タスクごとのメモリ量およびコード配置を，同時に決定できる．

4.3.2　時間分割法

時間分割法では，実行状態にあるタスクが，SPM の全容量を独占して使用する (図 4.3)．タスクが実行状態に遷移する際には，SPM 領域に配置するプログラムコードが，主記憶から SPM へ転送される．図 4.3 では，‘MM-SPM copy’ の部分がこのコード転送を行っている時点にあたる．SPM 領域に配置される関数は，この

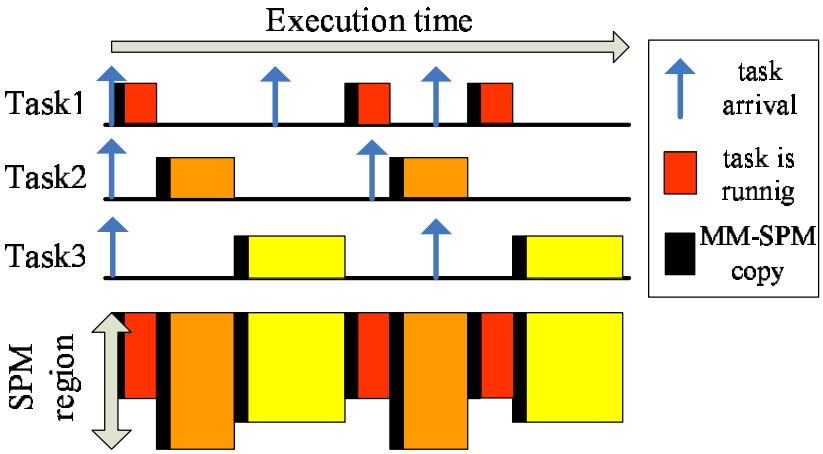


図 4.3: 非プリエンプティブなマルチタスク環境における時間分割法

転送にかかる消費エネルギーをオーバヘッドとして加味して決定される．このため，オーバヘッドが大きい関数が多く含まれるタスクは，SPM の全容量を使いきるとは限らない．

時間分割法における関数配置の決定を定式化した整数計画問題は，以下の通りとなる．主記憶－ SPM 間転送にかかる消費エネルギーのオーバヘッド $Eoverhead_{i,j}$ を考慮した上で，$Esaving$ の最大化を目指している．

$$\textbf{Maximize} : Esaving = \sum_{j} Esaving_{i,j} \times y_{i,j} \tag{4.5}$$

$$Esaving_{i,j} = fetch_{i,j} \times E_{SPMgain} - Eoverhead_{i,j} \tag{4.6}$$

$$E_{SPMgain} = E_{Cache\_read} - E_{SPM\_read} \tag{4.7}$$

$$Eoverhead_{i,j} = size_{i,j} \times (E_{SPM\_write} + E_{MM\_read}) \tag{4.8}$$

$$\textbf{s.t.} : \ \forall i. \ \sum_{j} size_{i,j} \times y_{i,j} \leq SPMsize \tag{4.9}$$

決定変数である 0-1 変数 $y_{i,j}$ の解を求めることにより，タスクごとの関数配置を決定できる．

4.3.3　混合分割法

混合分割法は，上記 2 つの SPM 空間分割法を併用した手法である．SPM 領域のより柔軟な活用を実現し，消費エネルギーのさらなる削減を目指す．図 4.4 に示す

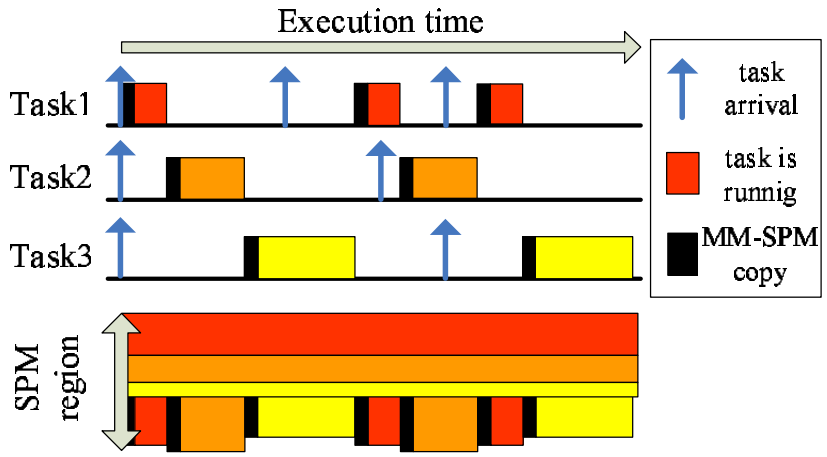


図 4.4: 非プリエンプティブなマルチタスク環境における混合分割法

ように，空間および時間分割法によって使用する SPM 領域のメモリ量は，消費エネルギー削減量 $Esaving$ が最大となるように，それぞれ分割して与えられる．混合分割法において定式化した整数計画問題を，次に示す．

$$\textbf{Maximize} : Esaving = Esaving\_spt + Esaving\_tmp \tag{4.10}$$

$$Esaving\_spt = \sum_i \sum_j Esaving\_spt_{i,j} \times x_{i,j} \tag{4.11}$$

$$Esaving\_tmp = \sum_i \sum_j Esaving\_tmp_{i,j} \times y_{i,j} \tag{4.12}$$

$$Esaving\_spt_{i,j} = fetch_{i,j} \times \frac{hyperperiod}{period_i} \times E_{SPMgain} \tag{4.13}$$

$$E_{SPMgain} = E_{Cache\_read} - E_{SPM\_read} \tag{4.14}$$

$$Esaving\_tmp_{i,j} = (fetch_{i,j} \times E_{SPMgain} \tag{4.15}$$

$$- Eoverhead_{i,j}) \times \frac{hyperperiod}{period_i} \tag{4.16}$$

$$Eoverhead_{i,j} = size_{i,j} \times (E_{SPM\_write} + E_{MM\_read}) \tag{4.17}$$

$$\textbf{s.t.} : \ SPMsize\_spt + SPMsize\_tmp \leq SPMsize \tag{4.18}$$

$$\textbf{s.t.} : \ \sum_i \sum_j size_{i,j} \times x_{i,j} \leq SPMsize\_spt \tag{4.19}$$

$$\textbf{s.t.} : \ \forall i. \ \sum_j size_{i,j} \times y_{i,j} \leq SPMsize\_tmp \tag{4.20}$$

$$\textbf{s.t.} : \ \forall i, \ \forall j. \ x_{i,j} + y_{i,j} \leq 1 \tag{4.21}$$

本問題の決定変数は，$SPMsize\_spt$，$SPMsize\_tmp$，$x_{i,j}$ および $y_{i,j}$ である．$SPMsize\_spt$ および $SPMsize\_tmp$ は，それぞれ，空間分割法および時間分割法として用いる SPM の容量をあらわしている．第 1 の制約式によって，各手法における SPM の容量が決定される．あるタスクが使用できる SPM の容量は，空間分割法による領域からさらに各タスクに分配される SPM 領域と，時間分割法の方針として用いられる領域の合計になる．0-1 変数 $x_{i,j}$ は，$func_{i,j}$ が空間分割法による SPM 領域に配置されるときに 1 となる．$y_{i,j}$ は時間分割法のそれに対応する 0-1 変数である．これらの解を求めることにより，空間および時間分割法によって用いられる SPM の各容量，空間分割法による SPM 領域中で各タスクに割り付けられる SPM の容量，および，タスクごとに SPM に配置される関数を，全て同時に決定することが可能である．

4.4　評価

提案する 3 種類の SPM 領域分割方針の有効性を確認するため，評価実験を行った．本節では，評価実験の手順，結果および考察を記す．

4.4.1　手順

実験には，ベンチマークスイート MiBench [15] からプログラムを選定し，表 4.2 に示す 5 種類のタスクセットを構成して用いた．それぞれのタスクには，50 個から 100 個程度の関数が含まれる．実験には，C 言語記述である各タスクのコードを，マイクロプロセッサシミュレータ SimpleScalar/ARM [61] 上で実行可能なバイナリコードにコンパイルしたものを用いた．

図 4.5 に，実験のワークフローを示す．各タスクのバイナリコードに対して，命令セットに準拠したアセンブリ言語への逆アセンブル，および，SimpleScalar/ARM 上でのトレース実行の各処理を行った．これにより，タスクセットに含まれるタスクの各関数のメモリ量と実行命令数の結果を取得した．タスクの起動周期は，命令レベルシミュレーションの結果を基にし，タスクセット実行時の CPU 使用率が 60 %程度となるように，命令フェッチの総数に比例して設定した．

これらの情報に対して，提案手法を適用した．SPM の領域分割などを決定する整数計画問題の最適解の導出には，シンプレックス法を実装した ILP ソルバであ

表 4.2: 非プリエンプティブな環境における提案手法の評価実験のために用いたタスクセット

セット名	含まれるタスク	タスク数
tasksetA	bf, tiff2rgba	2
tasksetB	cjpeg, crc, qsort, tiff2rgba	4
tasksetC	bitcnts, cjpeg, ispell, rawcaudio, sha	6
tasksetD	bitcnts, bf, crc, dijkstra, ispell, qsort, rawcaudio, sha	8
tasksetE	bitcnts, bf, cjpeg, crc, dijkstra, ispell, qsort, rawcaudio, sha, tiff2rgba	10

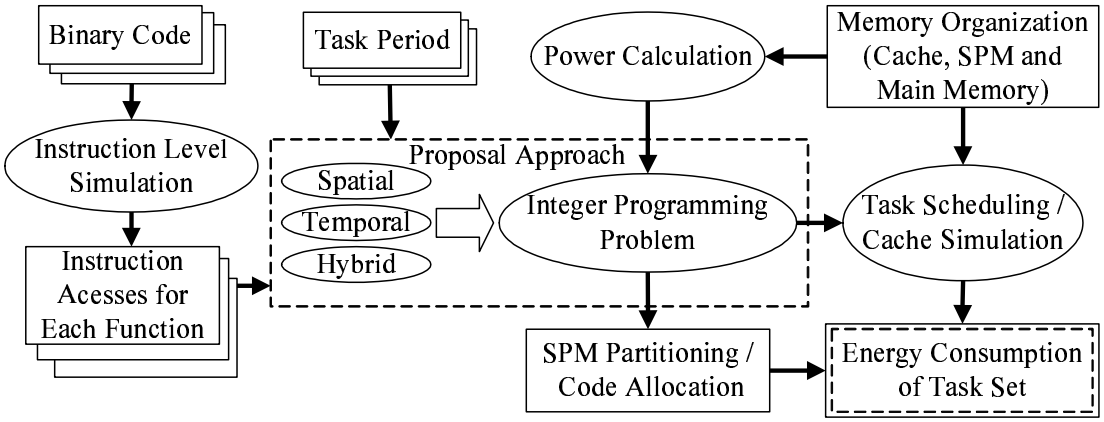


図 4.5: 非プリエンプティブなマルチタスク環境における評価実験のワークフロー

る glpsol 4.23 [55] を使用した．本研究における実験環境においては，高々10 秒以内で全ての整数計画問題の最適解が導出できた．

一次メモリは，16 KBytes 4-way のキャッシュに，4 K / 8 K / 12 K / 16 KBytes の SPM をそれぞれ組み合わせて構成した．外部主記憶は，Mobile DDR SDRAM とした．主記憶にはバースト・リードアクセスを行い，アクセス 1 回当たりに 4 ワード分のコードが読み出される．キャッシング可能領域に配置されるプログラムコードについては，タスクスケジューリングおよびキャッシュシミュレーションを行い，キャッシュヒットおよびミス回数を計測した．

以上の情報を基にして，タスクセット実行時の命令メモリにおける消費エネルギーの総量を計算した．なお，本研究では，システム動作中にメモリで静的に消費されるリークエネルギーは考慮しない．メモリの消費エネルギーモデルは，一次メモリには CACTI 4.2 [34, 40] を，外部主記憶には Micron System Power Calculator [57]

をそれぞれ用いた．

これまでに，固定優先度付きのマルチタスク環境下でSPMを適用する研究は，ほとんど行われていない．このため，提案手法の有効性をみるための評価基準とする次のような手法を導入した．まず，それぞれのタスクにSPMの容量を均等に割り付け，そののちに，各タスクごとに，割り付けられたSPM容量を制約としたナップサック問題によってSPM領域へのコード配置を関数単位で決定する方法である．

4.4.2　実験結果

図4.6，図4.7，図4.8，図4.9，および，図4.10に，実験結果を示す．グラフの縦軸は，タスクセット実行中にシステムの命令メモリで消費されるエネルギー値（単位はmJ）を示している．各タスクの起動周期は異なるため，周期の最小公倍数時間における消費エネルギーを計算した．

図中の横軸の“Smp”は評価基準（各タスクにSPM容量を均等に割り付けてからコード配置を決定する方法）を適用したときの消費エネルギー量を示している．“Spt”，“Tmp”，および“Hyb”は，本研究の提案手法である空間分割法，時間分割法，および混合分割法にそれぞれ対応している．‘xK’は，一次メモリ中のSPMの容量をあらわしている．“Cache hit”，“Cache miss”，および“SPM hit”は，キャッシュヒット，キャッシュミス，およびSPMアクセスで消費されるエネルギーの総量を示している．“Cache miss”には，キャッシュラインの内容更新に掛かる主記憶アクセスのための消費エネルギーも含まれる．“Overhead”は，主記憶－SPM間コード転送における消費エネルギーの総計である．

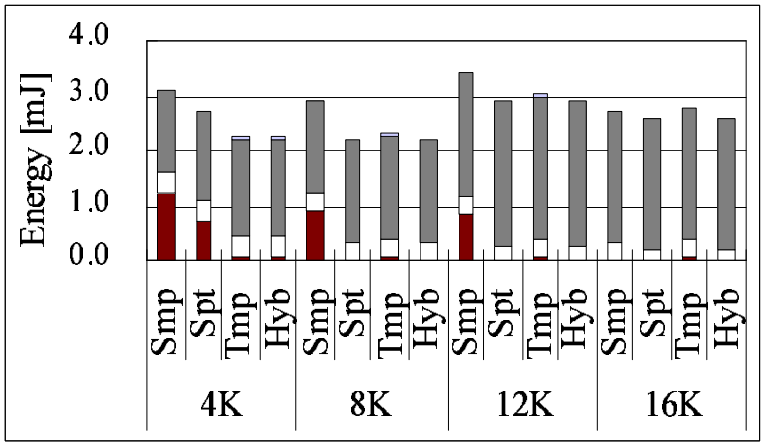


図 4.6: 非プリエンプティブなマルチタスク環境における ‘tasksetA’ の結果

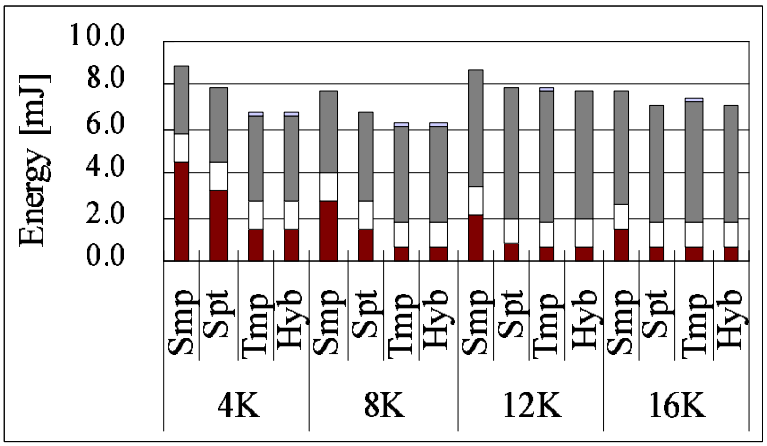


図 4.7: 非プリエンプティブなマルチタスク環境における ‘tasksetB’ の結果

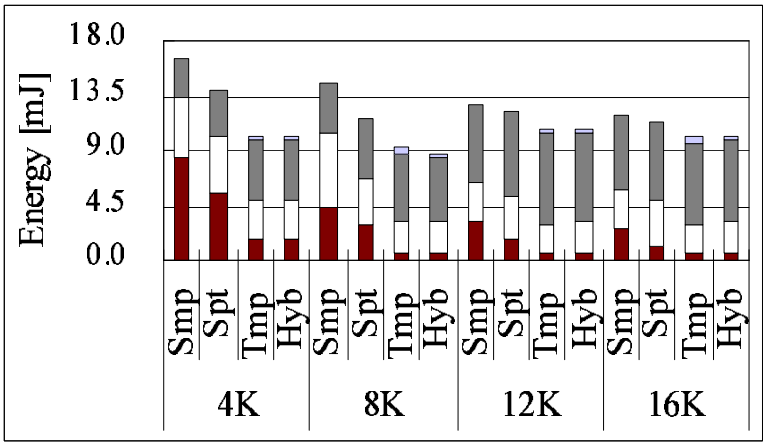


図 4.8: 非プリエンプティブなマルチタスク環境における ‘tasksetC’ の結果

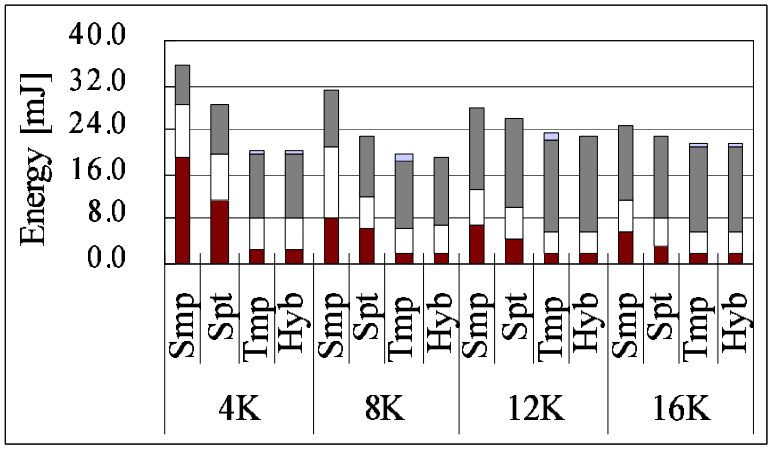


図 4.9: 非プリエンプティブなマルチタスク環境における ‘tasksetD’ の結果

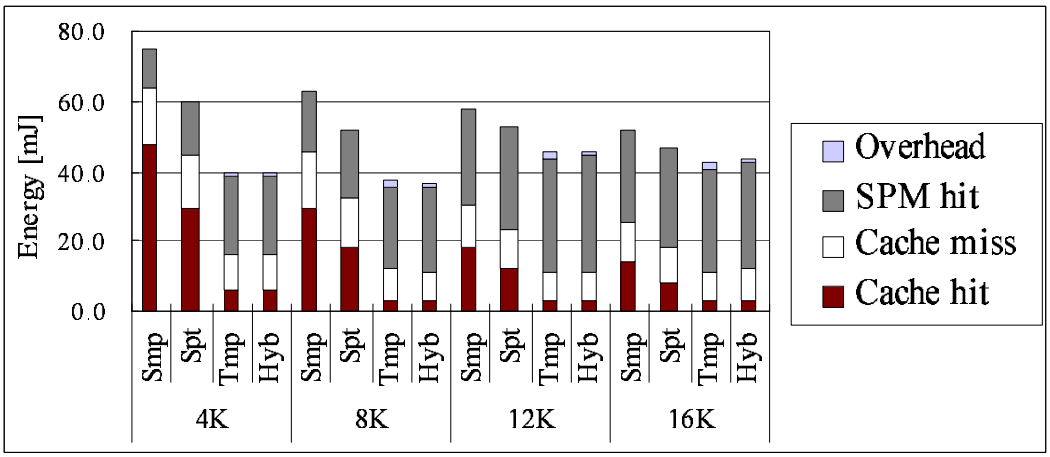


図 4.10: 非プリエンプティブなマルチタスク環境における ‘tasksetE’ の結果

4.4.3　考察

実験結果に対する考察を述べる．

まず，ほとんど全ての状況において，提案手法によって消費エネルギーが小さくできていることがわかる．最大で，空間分割法では 27.3 %，時間および混合分割法では 47.2 %の消費エネルギー削減が達成できた．タスクセット，SPM 容量ごとの平均をとると，領域，時間，および混合分割法でそれぞれ，12.4 %，21.6 %，および 23.0 %の消費エネルギー削減となった．このことから，マルチタスクシステムの命令メモリにおける提案手法の有効性が確認できた．

次に，タスクの個数と提案手法との関係をみる．タスク数 10 の図 4.10 では，SPM 容量が小さく限られるとき，これらのタスクセットでは，実行状態のタスクが SPM の全領域を占有する時間および混合分割法の有効性は高くなる．さらに，SPM 容量が大きいときに，空間および時間分割法を同時に利用可能な混合分割法の消費エネルギーが小さくなる．タスク数 6 の図 4.8 およびタスク数 8 の図 4.9 でも，ほぼ同様の傾向が得られている．このことから，タスク数が大きいとき，混合分割法は消費エネルギーの削減に最も適していることがわかる．

タスク数が少ない図 4.6 および図 4.7 では，SPM の容量が小さいときは空間分割法および混合分割法が有効に作用する．‘tasksetA’ のうちでは，8K SPM の空間分割法が，実験したメモリ構成および適用した提案手法のうちで，最も消費エネルギーが小さくなる．いっぽう，SPM の容量が大きくなると，これらのタスクセッ

ト実行時の消費エネルギー量は大きく増加してしまう．これは，タスク数 2 および 4 である ‘tasksetA’ および ‘tasksetB’ の総コードサイズが小さかったことによる．SPM が過容量となり，SPM からの命令フェッチや主記憶－ SPM 間データ転送にかかる消費エネルギーが余分に掛かっている．

SPM の総容量と消費エネルギー量の関係に注目する．実験結果をみると，SPM の総容量を整数計画問題への入力である定数値として設定した本研究の提案手法では，どのタスクセットでも，SPM の容量が 8KBytes のときに消費エネルギー最小になっていることがわかる．これは，本章で提案した SPM 領域分割方針を適用すると，SPM の容量を増やすことが必ずしも消費エネルギー削減に繋がらないことを示している．今回確認されたこの傾向は，消費エネルギー最小にできる SPM 容量を，変数として決定できる可能性を示唆している．

最後に，混合分割法に触れる．混合分割法は，空間および時間分割法の双方を活かした手法である．このため，提案する 3 種類の SPM 領域分割方針のうちで，混合分割法の消費エネルギー削減量が最も大きくなるはずである．しかし，図 4.9 における 4K SPM など，必ずしも全ての状況において混合分割法が消費エネルギー最小となっているわけではない．これは，整数計画問題の定式化の際に，キャッシュの動的な振る舞いを考慮しなかったためである．整数計画問題の定式化において用いた変数は，システム動作前の静的な解析によってすべて取得可能な値である．いっぽう，キャッシュ・ミス回数の計測には，システム実行中の動的な解析が必要である．しかし，注目すべきは，全タスクセットの全ての SPM 容量で，混合分割法によって消費エネルギーの安定的な削減を達成できていることである．

以上のことより，提案手法の有効性が確認できた．

4.5　まとめ

本章では，マルチタスク環境下での命令メモリにおける消費エネルギー削減を目的とした，3 種類の SPM 領域分割方針を提案した．提案手法は，非プリエンプティブな固定優先度ベースのスケジューリング方式に従うシステムに適用可能である．各方針について，タスクごとに使用する SPM 容量およびプログラムコード割当てを同時に決定可能な，線形計画問題に定式化した．

評価実験により，提案手法の有効性を確認した．提案手法を適用することにより，

タスクセット実行時の消費エネルギーを最大で 47.8 %削減することができた．タスク数が少ないタスクセットでは領域分割法が有効であり，多い場合には，SPM–主記憶間のデータ転送を行う時間分割法および混合分割法の有効性が高くなった．特に，混合分割法は，実験した全ての状況において安定的に消費エネルギーを削減できた．

第5章　プリエンプティブなマルチタスク環境におけるSPM管理技術

5.1　概要

高いリアルタイム応答性の確保が求められるハードリアルタイムな組込みシステムでは，複数のタスクを優先度に基づいて並行して処理することが要求される．タスクには，その処理の重要度に応じた実行優先順位が設定される．そして，デッドライン制約を厳密に守ることを目的として，高優先度タスクからの低優先度タスクのプリエンプションを行う．プリエンプションとは，あるタスクが実行中に，その処理を中断させて異なるタスクが先に実行されるようにタスク切替えを発生させることをいう．特にリアルタイムシステムにおいては，低優先度タスクの実行中に高優先度タスクが実行可能状態になった際に，高優先度タスクによるプリエンプションが実行される．ハードリアルタイムな組込みシステムでは，このプリエンプティブな優先度ベースのタスクスケジューリング方式が一般的に採用される．

本章では，プリエンプティブな固定優先度ベースのマルチタスク環境に対応したSPM管理技術に関する研究の報告を行う．

まず，リアルタイムシステムに適用可能なSPM領域の活用戦略を提案する．本章におけるSPM活用戦略は，第4章で提案した非プリエンプティブなマルチタスク環境におけるSPM領域分割手法を基にし，プリエンプティブな環境に対応できるよう拡張したものにあたる．提案する戦略は，空間活用法，時間活用法，および混合活用法の3種類である．空間活用法は，SPM領域を各タスクに排他的に分割して与える手法である．時間活用法は，実行状態のタスクがSPMの全領域を占有し，タスク切替え時にSPM内容を再配置する手法である．混合活用法は，上記2手法の組合せである．基本は空間活用法が適用されるが，高優先度タスクは，低優

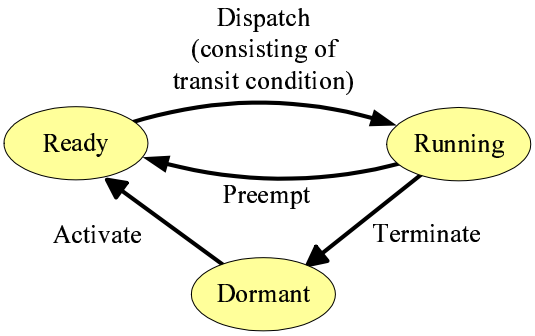


図 5.1: プリエンプティブなマルチタスク環境におけるタスクの状態遷移

先度タスクの使用する SPM 領域を時間活用法により横取り活用することができる．

各戦略における SPM 領域の活用方法は，システムの命令メモリの消費エネルギー削減量を目的関数とした整数計画問題として形式的に定式化する．整数計画問題の最適解の導出により，タスクごとに割当てる SPM の容量，および，SPM に配置するプログラムコードを同時に決定することができる．

さらに，SPM 領域の配置内容を実行時に更新することを目的としたワークフローを提案する．本ワークフローを採用したシステムにおけるプロファイラおよびコンパイラは，タスクの静的解析を担い，SPM 管理のための情報を生成する．システムの実行時における SPM 再配置は，リアルタイム OS およびハードウェアが担う．これらのモジュールが協調動作しながら，静的に生成された情報を基にして SPM 活用戦略を実現する．

最後に，評価実験の結果および考察を示し，提案手法の有効性を明らかにする．

5.2　準備

5.2.1　対象とするシステム構成

本研究の対象とする環境は，処理すべきタスクが複数存在し，優先度に基づいた順序で処理されるマルチタスクシステムである．タスクは，図 5.1 に示すように，休止（Dormant），実行可能（Ready），および実行（Running）の 3 状態をとる．タスク間に通信・同期処理はなく，各タスクは独立に動作する．全てのタスクは周期的に起動され，その起動周期はシステム動作前に静的に決定される．

タスクのスケジューリング規則は，レートモノトニック・スケジューリング方

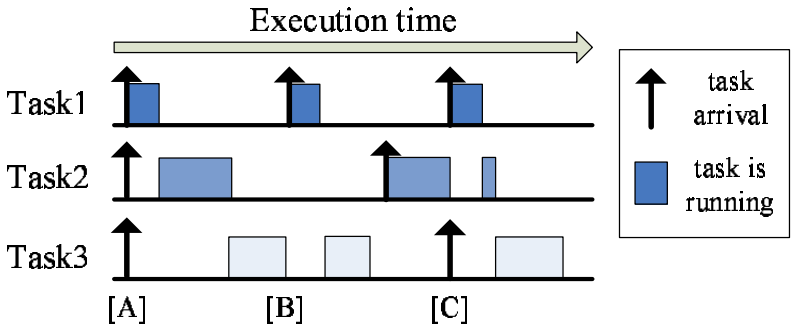


図 5.2: プリエンプティブな固定優先度ベースの方式によるスケジューリング例

式 [22] に従う．これは，プリエンプティブな固定優先度ベースのマルチタスク環境において，最適なスケジューリング規則となることが知られている方式である．プリエンプティブな環境では，優先度の高いタスクが実行可能状態になったときに，より優先度の低いタスクが実行状態にあってもタスク切替えが発生する．あるタスクが実行状態であるときに，より高優先度のタスクが実行可能状態となった場合には，図 5.1 に示すように，現在実行中のタスクの処理を中断して実行可能状態へと遷移する．低優先度タスクは，高優先度タスクの処理が終了したのちに，再び実行状態へと遷移して中断時点から処理を再開する．

5.2.2　タスクのスケジューリング方式

レートモノトニック・スケジューリング方式に従ったスケジューリングの例を，図 5.2 に示す．図中の矢印は，タスクの起動時刻を示している．文献 [22] にしたがい，タスクの優先度は起動周期の短い順に高くなるとする．すなわち，‘Task1’ の優先度が最も高く，‘Task3’ が最低優先度のタスクとなる．

図中の [A] の時点では，全てのタスクが同時に起動されて実行可能状態に遷移する．リアルタイム OS のスケジューラは，優先度の高い順（つまり，‘Task1’ → ‘Task2’ → ‘Task3’ の順）にディスパッチ対象となるタスクを選択して，実行状態に遷移していく．

図中の [B] のように，‘Task2’ の実行中により優先度の高い ‘Task1’ が起動されたときには，プリエンプションによるコンテキスト・スイッチが発生する．スケジューラは，‘Task2’ の実行を中断して実行可能状態に遷移させ，ディスパッチ対象として ‘Task1’ を選択して処理を先に行う．‘Task2’ の実行は，‘Task1’ の実行が

表 5.1: プリエンプティブなマルチタスク環境における変数定義

変数名	定義
$task_i$	i 番目のタスク．$1 \leq i \leq M$（M はタスク数）．
$period_i$	$task_i$ の起動周期
$func_{i,j}$	$task_i$ の j 番目の関数．$1 \leq j \leq N$（N は関数の個数）．
$fetch_{i,j}$	$func_{i,j}$ の命令フェッチ総数
$size_{i,j}$	$func_{i,j}$ のコードサイズ
E_{gain}	SPM とキャッシュとの読出しアクセスの消費エネルギー差
E_{C_read}	キャッシュへの 1 回当たりの読出しアクセスの消費エネルギー
E_{S_read}	SPM への 1 回当たりの読出しアクセスの消費エネルギー
E_{S_write}	SPM への 1 回当たりの書込みアクセスの消費エネルギー
E_{MM_read}	主記憶への 1 回当たりの読出しアクセスの消費エネルギー
$Esaving_{i,j}$	$func_{i,j}$ を SPM 領域に配置することにより削減できるエネルギー
$Eoverhead_{i,j}$	$func_{i,j}$ を主記憶から SPM に転送するための消費エネルギー
$SPMsize$	SPM の総容量
$SPMsize\_spt_i$	$task_i$ が空間活用法で割当てられる SPM 容量
$SPMsize\_tmp_i$	$task_i$ が時間活用法で割当てられる SPM 容量
$hyperperiod$	全タスクの起動周期の最小公倍数

終了してから処理が再開される．

[C] は，‘Task2’ の実行中に，優先度のより高い ‘Task1’ とより低い ‘Task3’ が同時に起動周期が訪れた時点をあらわしている．このとき，[B] と同様に，スケジューラはまずプリエンプションによるコンテキスト・スイッチを発生させ，‘Task1’ の処理を先に行う．低優先度の ‘Task3’ は，‘Task1’ の処理と，実行再開後の ‘Task2’ の処理が終了するまで実行状態に遷移されない．

5.2.3　変数の定義

第 5.3 節にて定式化する整数計画問題で用いる変数の定義を，表 5.1 に示す．本表中の変数の値は，システム動作前のタスクごとのプロファイル処理によって静的な解析によってすべて取得可能である．

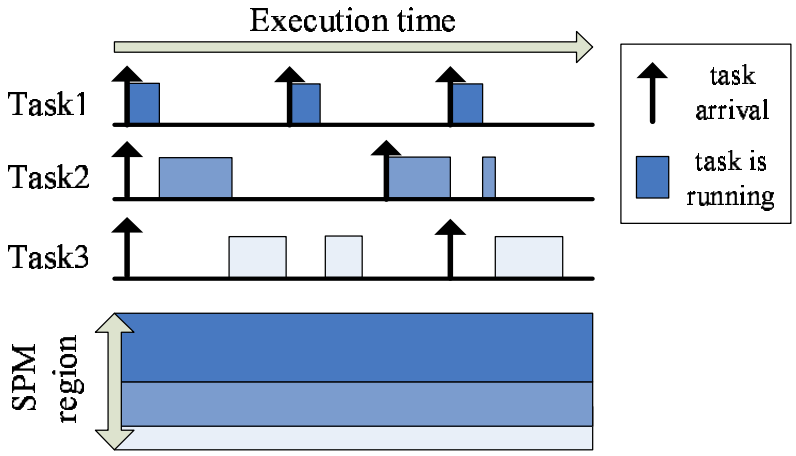


図 5.3: プリエンプティブなマルチタスク環境における空間活用法

5.3　SPM 活用戦略

本節では，プリエンプティブな固定優先度ベースのマルチタスク環境において，SPM の有効活用を実現する活用戦略の詳細を解説する．提案する戦略は，空間活用法，時間活用法，および混合活用法の 3 種類である．これらの戦略は，第 4 章で提案した SPM 領域分割手法を基にして，プリエンプティブな環境に適用可能となるように拡張を加えた手法にあたる．消費エネルギー最小化のの観点から，各方針におけるタスクごとの SPM 領域分割およびコード配置の決定は，組込みシステムのメモリにおける消費エネルギー削減量の最大化を目的関数とした整数計画問題に定式化する．

なお，本研究では，タスクのユーザコードおよびユーザコードに付随する標準ライブラリ関数のみをメモリ領域への配置対象とし，リアルタイム OS のカーネルコードは一意にキャッシング可能領域へ配置するとする．また，プログラムコードの配置粒度は関数単位とする．

5.3.1　空間活用法

空間活用法は，第 4.3.1 節で述べた非プリエンプティブな環境における空間分割法と，すべて同様の戦略となる．これは，タスクへの SPM 容量の割当ては動的に変更されることがないため，プリエンプションの有無の影響は受けないからである．

図 5.3 に示すように，空間活用法は，SPM の容量を固定的に分配して各タスク

に与える手法である．各タスクは，与えられた SPM 領域を占有して使用する．空間活用法では，タスクの起動周期を情報として用いることにより，実行頻度の高いコードが SPM 領域により多く割当てられるようにする．

各タスクに与える SPM の容量と，タスクごとの SPM 領域への関数配置の決定は，次に示す整数計画問題として定式化した．

$$\textbf{Maximize} : Esaving = \sum_i \sum_j Esaving_{i,j} \times x_{i,j} \tag{5.1}$$

$$Esaving_{i,j} = fetch_{i,j} \times \frac{hyperperiod}{period_i} \times E_{gain} \tag{5.2}$$

$$E_{gain} = E_{C\_read} - E_{S\_read} \tag{5.3}$$

$$\textbf{s.t.} : \sum_i SPMsize\_spt_i \leq SPMsize \tag{5.4}$$

$$SPMsize\_spt_i = \sum_j size_{i,j} \times x_{i,j} \tag{5.5}$$

ここで，$SPMsize\_spt_i$ は，$task_i$ に空間活用法で割当てられる SPM 容量をあらわす変数である．$x_{i,j}$ は，次式で定義される 0-1 決定変数をあらわしている．

$$x_{i,j} = \begin{cases} 1 & (func_{i,j} \text{が SPM の空間活用領域に配置されるとき}) \\ 0 & (\text{それ以外}) \end{cases} \tag{5.6}$$

これらの解集合を求めることにより，タスクごとに割当てられる SPM 容量および SPM 領域へのコード配置を同時に決定する．

5.3.2　時間活用法

図 5.4 に示すように，時間活用法では，実行状態にあるタスクが SPM の全容量を独占して使用する．タスクがある起動周期内で初めて実行状態に遷移するときには，タスク中の関数のうちから，アクセス頻度の高いものを主記憶から SPM へと転送する．タスクの実行終了時には，主記憶–SPM 間のコード転送を再度行うことによって SPM の保持する内容を開始前の状態に戻す．図 5.4 では，‘MM-SPM copy’ が主記憶–SPM 間転送を行っている部分にあたる．

プリエンプティブな環境での時間活用法と，第 4.3.2 節で述べた時間分割法とが大きく異なるのは，タスクの実行終了時（休止状態に遷移した時）にも，主記

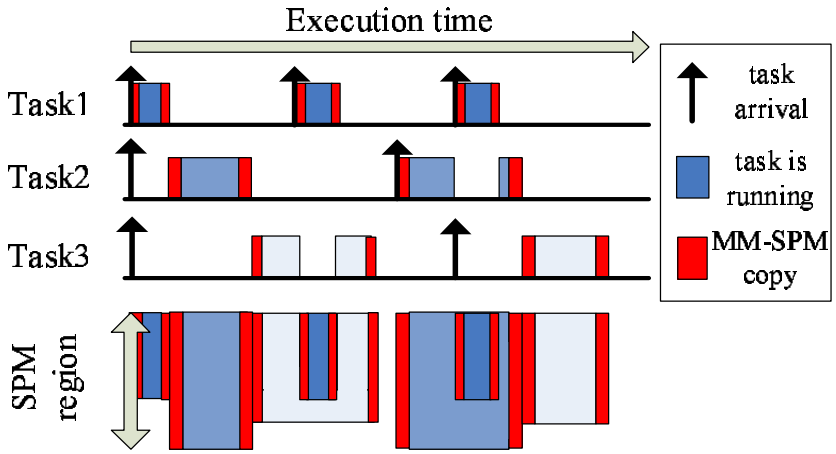


図 5.4: プリエンプティブなマルチタスク環境における時間活用法

憶–SPM 間のプログラムコード転送を行う必要がある点である．SPM 領域に割当てられる関数の決定には，計 2 回分のコード転送にかかる消費エネルギーのオーバヘッドを考慮する必要がある．このため，オーバヘッドが大きい関数を多く含むタスクは，SPM の全容量を使いきるとは限らない．

時間活用法における SPM 領域への関数配置を定式化した整数計画問題を，次に示す．主記憶–SPM 間コード転送のオーバヘッド $Eoverhead_{i,j}$ を計 2 回分加味したうえで，$Esaving$ の最大化を目指している．

$$\textbf{Maximize}: Esaving = \sum_{j} Esaving_{i,j} \times y_{i,j} \tag{5.7}$$

$$Esaving_{i,j} = fetch_{i,j} \times E_{gain} - Eoverhead_{i,j} \times 2 \tag{5.8}$$

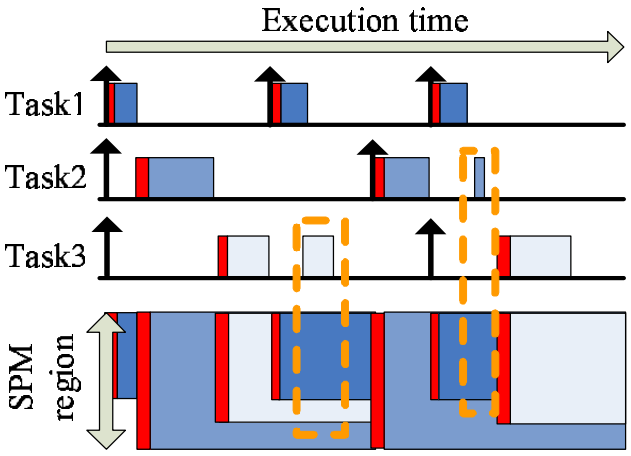
$$E_{gain} = E_{C\_read} - E_{S\_read} \tag{5.9}$$

$$Eoverhead_{i,j} = size_{i,j} \times (E_{S\_write} + E_{MM\_read}) \tag{5.10}$$

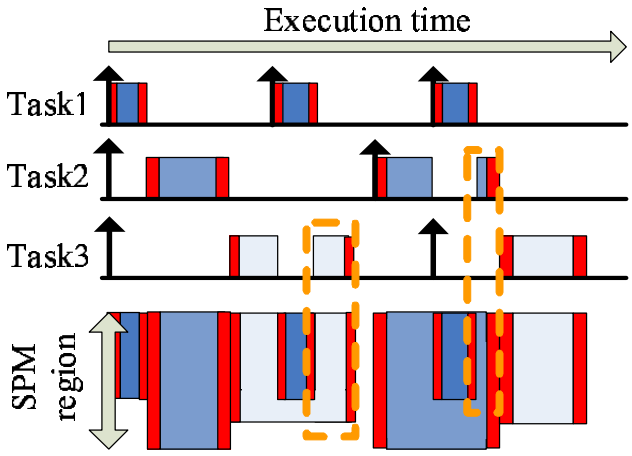
$$\textbf{s.t.}: \ \forall i. \ \ SPMsize\_tmp_i \leq SPMsize \tag{5.11}$$

$$SPMsize\_tmp_i = \sum_{j} size_{i,j} \times y_{i,j} \tag{5.12}$$

式中の $y_{i,j}$ は，以下で定義される 0-1 決定変数である．



(a) コード転送を実行開始のみ 1 回だけ行う場合



(b) コード転送を実行開始と終了の 2 回行う場合

図 5.5: 時間活用法のプリエンプティブな環境への拡張方針

$$y_{i,j} = \begin{cases} 1 & (func_{i,j} \text{が SPM の時間活用領域に配置されるとき}) \\ 0 & (\text{それ以外}) \end{cases} \tag{5.13}$$

これらの解集合を求めることによって，各タスクの関数配置が決定される．

ここで，図 5.5 を例にして，プリエンプティブな環境における時間活用法では．なぜオーバヘッドを計 2 回分加味する必要があるかを解説する．

図 5.5 の破線長方形で囲った部分のように，プリエンプションによるタスク切替えで低優先度タスクの実行が中断した場合を考える．もし第 4.3.2 節の方針と同様

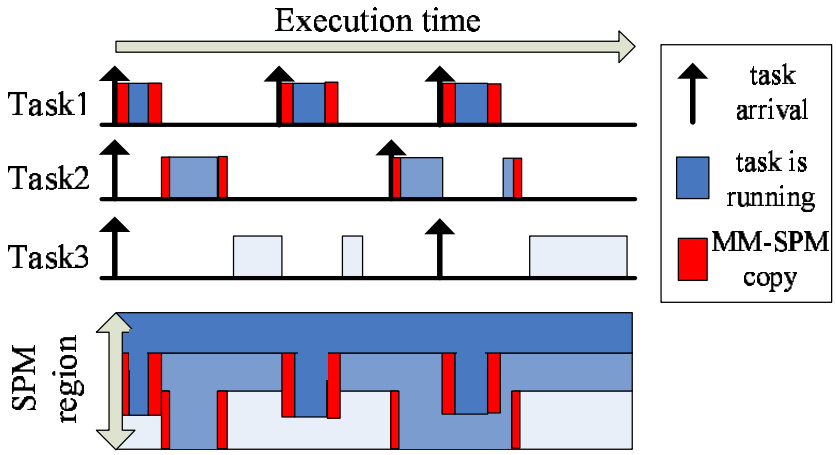


図 5.6: プリエンプティブなマルチタスク環境における混合活用法

に，タスクの実行開始時のみにコード転送を行うとき（図 5.5(a)），低優先度タスクが実行を再開する時には，SPM の内容が書換えられた状態のままとなる．つまり，低優先度タスクが実行再開後にアクセスしようとするコードが，SPM 領域に配置されていない状況が発生しうる．図 5.5(b) のように，タスク終了時にも SPM 内容を更新して高優先度タスクが SPM の変更内容を書き戻す拡張を加えることによって，低優先度タスクは実行再開後も SPM に配置されたコードを利用できるようになる．

5.3.3　混合活用法

混合活用法は，上記 2 つの SPM 活用法を併用する手法である．図 5.6 に示すように，空間活用と時間活用の両手法を組み合わせ，SPM の柔軟な活用を目指す．

混合活用法における高優先度タスクは，空間活用法によって自らに与えられる領域に加え，より優先度の低いタスクの使用する領域を時間活用領域として使用することができる．ただし，“横取り” して使用した領域は，その低優先度タスクの実行再開に備え，高優先度タスクの実行終了後に内容を元に戻す必要がある．つまり，高優先度タスクが SPM 領域を横取りする場合には，第 5.3.2 節で解説した理由と同様に，主記憶–SPM 間の消費エネルギーのオーバヘッドを計 2 回分考慮する必要がある．最低優先度のタスクは，空間活用法による領域しか与えられない．例えば，図 5.6 の例では，‘Task1’ は ‘Task2’ および ‘Task3’ の領域を，‘Task2’

は ‘Task3’ の領域を横取りして使用している．最低優先度の ‘Task3’ は，横取りできる対象がないため，みずからの空間活用領域しか使用しない．

タスクごとに割当てられる空間活用領域，および，高優先度タスクが横取りして時間活用領域として使用する SPM のメモリ量は，消費エネルギー削減量 $Esaving$ が最大となるように，それぞれ決定される．あるタスクが使用できる SPM の容量は，空間活用法によって分配される領域と，低優先度タスクの空間活用領域から時間活用法として横取りできる領域の合計となる．

混合活用法において定式化した整数計画問題を，次に示す．

$$\textbf{Maximize} : Esaving = Esaving\_spt + Esaving\_tmp \tag{5.14}$$

$$Esaving\_spt = \sum_i \sum_j Esaving\_spt_{i,j} \times x_{i,j} \tag{5.15}$$

$$Esaving\_spt_{i,j} = fetch_{i,j} \times \frac{hyperperiod}{period_i} \times E_{gain} \tag{5.16}$$

$$Esaving\_tmp = \sum_i \sum_j Esaving\_tmp_{i,j} \times y_{i,j} \tag{5.17}$$

$$Esaving\_tmp_{i,j} = (fetch_{i,j} \times E_{gain} - Eoverhead_{i,j} \times 2) \times \frac{hyperperiod}{period_i} \tag{5.18}$$

$$E_{gain} = E_{C\_read} - E_{S\_read} \tag{5.19}$$

$$Eoverhead_{i,j} = size_{i,j} \times (E_{S\_write} + E_{MM\_read}) \tag{5.20}$$

$$\textbf{s.t.} : \sum_i SPMsize\_spt_i \leq SPMsize \tag{5.21}$$

$$SPMsize\_spt_i = \sum_j size_{i,j} \times x_{i,j} \tag{5.22}$$

$$\textbf{s.t.} : \forall i. \quad \sum_j size_{i,j} \times y_{i,j} \leq SPMsize\_tmp_i \tag{5.23}$$

$$\exists k,\ period_k > period_i\ .$$

$$SPMsize\_tmp_i = SPMsize - \sum_k SPMsize\_spt_k \tag{5.24}$$

$$\textbf{s.t.} : \forall i,\ \forall j.\ \ x_{i,j} + y_{i,j} \leq 1 \tag{5.25}$$

本問題の決定変数は，$SPMsize\_spt_i$，$SPMsize\_tmp_i$，$x_{i,j}$ および $y_{i,j}$ となる．

$SPMsize\_spt_i$ は，各タスクに与えられる空間活用領域の SPM 容量をあらわしている．この変数値は，第一の制約である式 (5.21) によって決定される．制約式 (5.23) は，高優先度タスクが低優先度タスクから横取りして時間活用領域として使

用する SPM 容量 $SPMsize\_tmp_i$ を設定し，その上で時間活用領域に配置する関数を決定している．0-1 変数 $x_{i,j}$ および $y_{i,j}$ は，式 (5.6) および式 (5.13) によって定義される 0-1 変数である．それぞれの 0-1 変数値の組は，式 (5.25) の制約も満たす．

これらの決定変数の解集合を求めることにより，SPM 領域中で空間活用法によってタスクごとに割付けられる SPM の容量，高優先度タスクが時間活用領域として用いる SPM の容量，および，各タスクについて SPM へ配置される関数を，全て同時に決定することが可能である．

5.4　SPM 管理のためのワークフロー

本節では，時間および混合活用法において SPM の配置内容を実行時に管理するためのワークフローの詳細を述べる．

本研究で提案するワークフローを採用したシステム設計および実行環境の全体的な流れを，図 5.7 および図 5.8 に示す．図 5.7 は，実行時に用いる情報を，プロファイラおよびコンパイラによる静的解析で生成する流れをあらわしている．システムの実行時には，図 5.8 に示すように，リアルタイム OS とハードウェアの協調支援による処理によって，第 5.3 節にて提案した SPM 活用戦略が実現される．

5.4.1　設計時の処理

第 5.3 節で述べた SPM 活用戦略を適用するためには，入力値として，表 5.1 に示したタスクコードに関する各変数の値を取得する必要がある．タスクに含まれる各関数のコードサイズは，プロファイラによるバイナリファイルの逆アセンブルによって取得される．

各関数の実行命令総数は，図 5.7 の上半分に示すように，プロファイラによる命令レベルシミュレーションによって解析できる．命令レベルシミュレーションは，タスク毎に単体で実行するだけでよい．これらの静的解析の結果と，キャッシュ，SPM および主記憶の消費エネルギーモデルが，コンパイル時に実行される SPM 活用戦略への入力値となる．

コンパイル時には，まず，コンパイラの基本機能として，カーネルコードとタスクコードとをリンクした実行可能ファイルを出力する．そして，プロファイラの静的解析によって得られた情報を基にし，SPM 活用戦略を適用し，SPM 領域の分

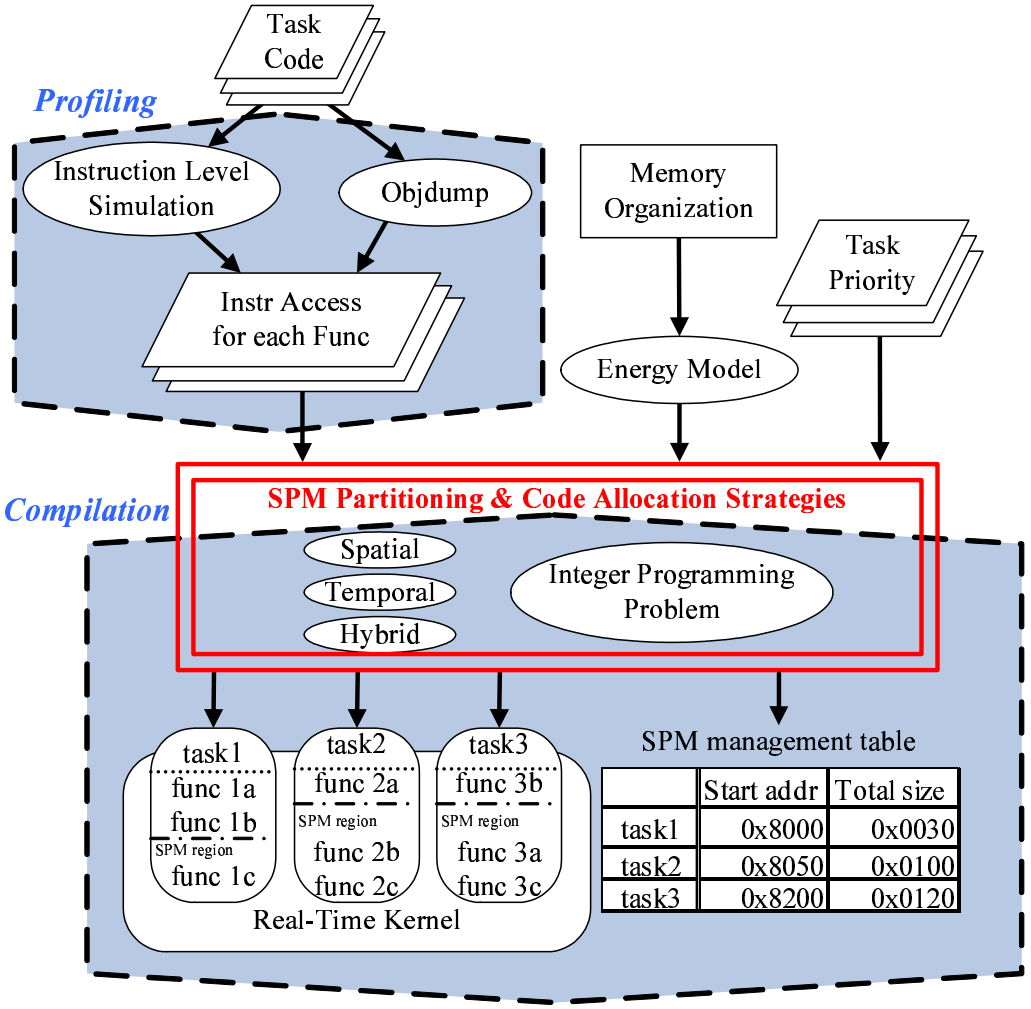


図 5.7: ワークフローにおける SPM 管理のための静的解析とテーブル情報の生成

割方法とコード配置を決定する．図 5.7 の下半分に示すように，SPM 活用戦略によって選択された関数のプログラムコードは，SPM 領域への配置対象であることを示すリンク属性が付与される．

さらに，コンパイラは，SPM 領域への配置対象となるコード群の情報を持つテーブル情報を生成する．本テーブルのエントリは，タスクの ID 番号である．メンバ情報は，各タスクの SPM 領域の開始アドレスと SPM 領域に配置された関数の総メモリ量のみである．

ここまでで解説したプロファイルおよびコンパイルまでの処理は，すべて静的に実行される．すなわち，SPM への配置対象となる関数はすべて，システム実行

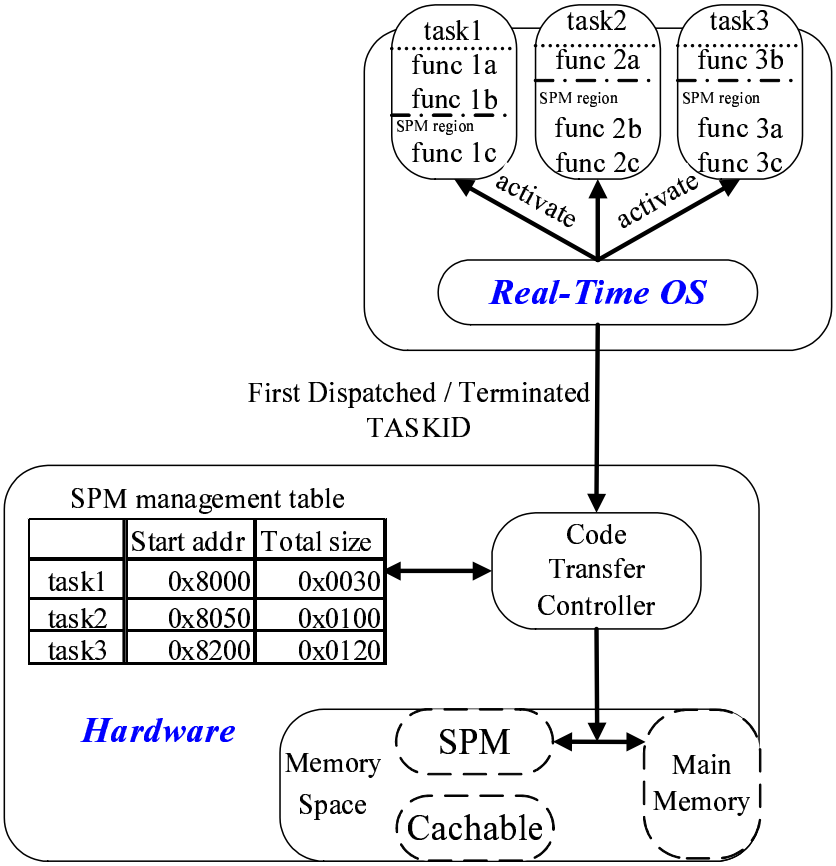


図 5.8: ワークフローにおける実行時最適化の処理

前の設計時に決定される．実行時に変更されるのは，混合活用法において各タスクが占める SPM 領域，および，時間および混合活用法でのコード配置となる．

5.4.2　実行時の SPM 管理動作

ワークフローにおけるシステム実行時の SPM 管理は，リアルタイム OS とハードウェアの協調動作により実現される．

まず，リアルタイム OS のディスパッチャは，図 5.8 に示すようにタスクが起動してからその周期で初めてディスパッチされた時と，タスクの実行が終了して休止状態に遷移した時の計 2 回の時点で，対象タスクの ID 番号をハードウェア中のコード転送コントローラ（図中の “Code Transfer Controller”）の専用レジスタに通知する．これは，ハードウェア側からは，現在どのタスクがシステムで実行さ

れているかを知る手段を持たないためである．

図 5.8 に示すように，SPM の管理を支援するためのハードウェアは，SPM 管理のためのテーブル情報およびコード転送モジュールからなる．コンパイラが生成したテーブル情報は，（図中の “SPM management table”）は，ハードウェア上のモジュールとして実装される．コード転送モジュールは，時間活用領域における主記憶–SPM 間のコード転送を実現するための専用モジュールである．

コード転送モジュールは，SPM 活用戦略が時間および混合分割法であるときに動作する．リアルタイム OS がコントローラの専用レジスタへと対象タスク ID 番号を書込むと，コード転送モジュールはその動作を開始する．コード転送コントローラは，ハードウェア上のテーブル情報を参照し，対応するタスクのプログラムコード群を主記憶から SPM 領域へと転送する．

5.5　評価

第 5.3 節にて提案した 3 種類の SPM 活用戦略の有効性を確認するため，評価実験を行った．本節では，評価実験の手順，結果および考察を記す．

5.5.1　手順

ハードウェアのシミュレーション環境は，SkyEye-1.2.6_rc1 [21, 62] を採用した．これは，ARM 社製の複数種類のプロセッサ・コアをシミュレーションすることができる，オープンソースの ISS（命令セットシミュレータ）である．SkyEye は，ARM アーキテクチャ用の実行可能ファイルを一命令ごとに読込んでいき，パイプライン動作でデコードしながらシミュレーションを行っていく．さらに，SkyEye は高速なシミュレーションのために，バイナリトランスレーションをサポートしている．

第 5.4.1 節で述べた設計時のプロファイル処理を行うため，文献 [44] および文献 [47] における成果を参考にして，SkyEye のアーキテクチャ記述に機能拡張を行った．具体的には，命令メモリのアクセスログを出力できるようにした．また，シミュレーションのサイクル精度の確保のため，SkyEye のソースコード中にいくつかの記述変更を施した．タイマの進度にパイプラインハザードやストールの影響を含めることにより，正確なティック処理の精度を実現した．

シミュレーションのターゲット・コアは ARM920T [49] を選択し，クロス開発環境（コンパイラやリンカなど）は GNU arm-elf [54] を用いた．コンパイラ arm-elf-gcc

表 5.2: プリエンプティブなマルチタスク環境における SPM 管理技術の評価実験で採用したプログラム

タスク名	ファミリ名	概要	サイクル数 [$\times 10^3$]
aifftr01	automotive	FFT 変換フィルタ	2959
basefp01	automotive	整数／浮動小数点演算	39
bitmnp01	automotive	ビット操作処理	59
cacheb01	automotive	高負荷キャッシュシミュレーション	99
idctrn01	automotive	逆離散コサイン変換	337
rgbcmy01	consumer	RGB–CMYK 画素変換	2970
rgbyiq01	consumer	RGB–YIQ 画素変換	4844
ospf	networking	Dijkstra グラフ最短経路探索	197
pktflow	networking	TCP/IP ルータによるパケット送受信	587
routelookup	networking	IP パケットルータの送受信	581
bezier01	office	Bezier 曲線の計算	3111
dither01	office	Floyd-Steinberg 法による誤差拡散法	7026
rotate01	office	bitmap 画像の直角転置	1295
text01	office	テキスト探索	3597
conven00	telecom	畳み込み行列演算	201
viterb00	telecom	viterbi 行列演算	424

のバージョンは 4.1.1 である．

リアルタイム OS として，異なる優先度のタスク同士ではプリエンプティブな優先度ベーススケジューリング方式を採用している，TOPPERS/ASP カーネル（Release 1.3.2）[63] を用いた[1]．本カーネルのディスパッチャ周りの記述に対して，第 5.4.2 節にて提案した動作が実現できるよう，ソースコードに機能拡張を施した．このためのコードの追加量は，オリジナルのものに対してアセンブリ言語で 35 命令相当であった．

TOPPER/ASP および μITRON 4.0 仕様には，タスクの周期的起動を実現する機能は提供されていない．このため，タスクに与えられた実行周期をパラメータ

[1]異なる優先度の周期タスク同士では，レートモノトニック・スケジューリング方式と同等のタスクスケジューリングが行われる．なお，TOPPERS カーネルでは，優先度が等しいタスクの間は，FCFS 方式（First Come First Service．実行可能状態となった順からタスクを実行していく方式）のスケジューリング規則が採用されている．

表 5.3: プリエンプティブなマルチタスク環境における SPM 管理技術の評価実験で用いたタスクセット

	#	含まれるタスク
SetA	5	aifftr, basefp, bitmnp, cacheb, idctrn
SetB	7	bezier, dither, ospf, pktflow, rotate, routelookup, text
SetC	11	conven, rgbcmy, rgbyiq, viterb, and SetB
SetD	16	the combination of SetA and SetC

表 5.4: メモリシステムのパラメータ

Memory		Read access [pJ]	Write access [pJ]
Cache	8k	8.335	6.981
	16k	10.94	9.521
SPM	1k	1.280	1.225
	2k	1.763	1.754
	4k	2.615	2.218
	8k	3.973	3.576
SDRAM		174.1	—

とする周期ハンドラを用意し，周期ハンドラから対応するタスクを起動するプログラムを作成した．

実験のため，ベンチマークスィート EEMBC [52] から表 5.2 に示す 16 個のプログラムを選定した．各プログラムを周期タスク化し，表 5.3 に示す 4 種類のタスクセットを構成して用いた．それぞれのタスクには，20 個から 30 個程度の関数が含まれる．それぞれのタスクの起動周期は，全体の CPU 使用率が 60 %程度となるように，各実行時間に比例して設定した．なお，全体の CPU 使用率の大小は，消費エネルギーの評価結果の傾向には影響を与えないことに注意されたい．

第 5.3 節において定式化した整数計画問題の最適解の導出には，シンプレックス法を実装した ILP ソルバである glpsol 4.23 [55] を使用した．本研究における実験環境においては，高々1 秒以内で全ての整数計画問題の最適解が導出できた．

一次メモリは，8 KBytes 4-way のキャッシュに，1K / 2K / 4K / 16KBytes の

SPM をそれぞれ組み合わせて構成した．さらに，比較対象として，8 KBytes および 16 KBytes のみの 4-way のキャッシュのみで一次メモリを構成するものも評価した．外部主記憶は SDRAM とし，Micron 社製 Mobile DDR SDRAM [58] よりメモリ構成のパラメータを採用した．SDRAM にはバースト・リードアクセスを行い，アクセス 1 回当たりに 4 ワード分のコードが読み出される．メモリの消費エネルギーモデルには，CACTI 5.3 [36, 40, 51] を用いた．表 5.4 に，CACTI 5.3 の出力として得られるメモリシステムの消費エネルギー値を示す．‘Read Energy’ および ‘Write Energy’ の行が，各メモリからの 1 回当たりの動的な読込みおよび書出しアクセスの消費エネルギー（単位は pJ）をあらわしている．なお，CMOS テクノロジのデバイスサイズは 65 nm，動作温度は 360 度と仮定した．SDRAM は，4 ワード・バーストアクセス 1 回当たりの値を記載している．なお，本研究では，リークエネルギーは考慮しない．

これまでに，レートモノトニック・スケジューリング方式に従う環境において SPM を適用する研究はほとんど行われていない．このため，提案手法の有効性をみるための評価基準とする次のような手法を導入した．まず，それぞれのタスクに SPM の容量を均等に割り付け，そののちに，各タスクごとに，割り付けられた SPM 容量を制約としたナップサック問題によって SPM 領域へのコード配置を決定する方法 [29] である．

5.5.2　実験結果

図 5.9，図 5.10，図 5.11，および，図 5.12 に，評価実験の結果を示す．各タスクの起動周期は異なるため，周期の最小公倍数時間における実験結果を導出した．実験結果には，リアルタイム OS のカーネルコードのフェッチで消費されたエネルギーも含めて算出している．ただし，システム動作開始直後に実行されるスタートアップルーチンおよびアイドル状態（カーネルの処理すべきタスクが何もない状態）でのメモリアクセスは，結果から除外している．

グラフの縦軸は，該当タスクセットの実行中にメモリシステムで消費されるエネルギー値（単位は μJ）を示している．横軸は，一次メモリの構成とその容量をあらわしている．‘I$ only’ は，キャッシュのみで一次メモリを構成したシステムにおける評価結果をあらわしている．‘I$ 8K & SPM αk’ は，8 KBytes のキャッシュに組み

合わせた SPM の容量をあらわしている．“Std” は評価基準（各タスクに SPM 容量を均等に分配してからコード配置を決定する方法）を適用したものであり，“Spt”，“Tmp”，および “Hyb” は，本研究で提案した SPM 活用戦略である空間活用法，時間活用法，および混合活用法にそれぞれ対応している．

グラフ中の凡例である ‘cache hit’，‘cache miss’，および ‘SPM hit’ は，キャッシュ・ヒット，キャッシュ・ミス，および SPM アクセスで消費されるエネルギー量の合計をあらわす．‘cache miss’ には，キャッシュ・ラインの内容を更新するための主記憶アクセスで消費されるエネルギーも含まれている．‘overhead’ は，時間および混合活用法でのタスク切替え時に発生する，主記憶–SPM 間コード転送時のメモリアクセスに掛かる消費エネルギー量をあらわしている．

5.5.3　考察

実験結果に対する考察を述べる．

まず，全ての SPM 容量およびタスクセットで，タスクセット実行中の消費エネルギーが，評価基準よりも小さくできていることがわかる．同容量の評価基準と比較して，最大で，空間活用法では 33 %，時間活用法では 69 %，混合活用法では 73 %の消費エネルギーをそれぞれ削減している．平均では，空間活用法では 18 %，時間活用法では 38 %，混合活用法では 41 %の消費エネルギーをそれぞれ削減している．このことから，本研究で提案した 3 種類の SPM 活用戦略および SPM 管理ワークフローの適用によって，プリエンプティブなマルチタスクシステムで消費されるエネルギーを削減できることが確認できた．

次に，キャッシュのみの構成と提案手法を比較する．どのタスクセットにおいても，8 KB の命令キャッシュのみで構成した場合の消費エネルギーは，8 KB のキャッシュに 1 KB の SPM を組み合わせて構成し，SPM には比較手法（‘Std’）を適用した場合とほぼ同量であった．さらに，キャッシュの容量を 16 KB とした場合は，全てのタスクセットで消費エネルギーが大きくなった．つまり，単純にキャッシュの容量を増やすことは，消費エネルギー削減に貢献できない．そして，SPM に対して本研究の提案手法を適用した場合には，キャッシュのみの構成より消費エネルギーを削減できることができた．このことは，キャッシュと SPM を組み合わせて組込みリアルタイムシステムの一次メモリを構成することの有効性が高いことを示している．

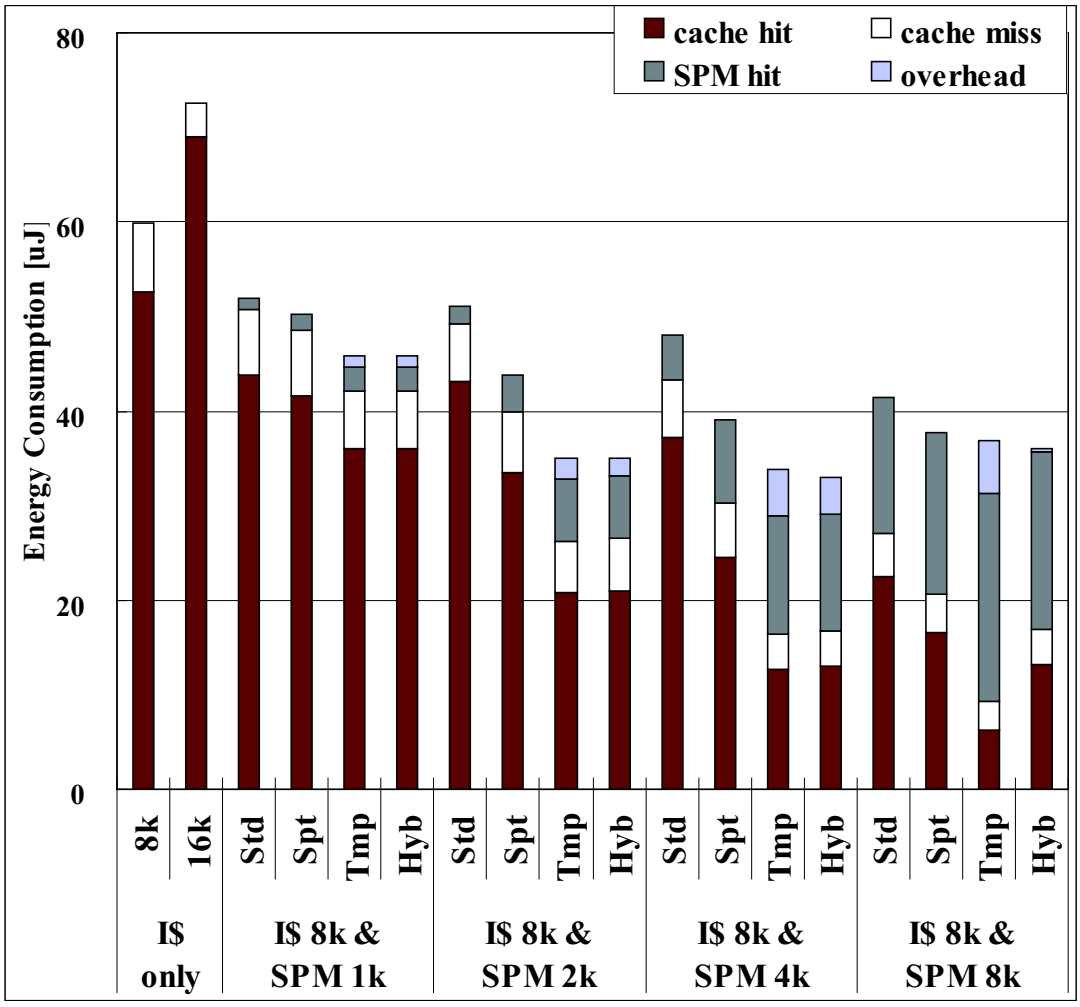


図 5.9: プリエンプティブなマルチタスク環境における ‘setA’ の結果

提案戦略の有効性とタスク数の関係に着目する．タスク数が 11 および 16 である図 5.11 および図 5.12 では，SPM 領域の内容を動的に管理する時間および混合活用法の有効性が高くなっている．このことから，タスク数が大きい場合にとくに，提案手法が有効となることがわかる．

提案する SPM 活用戦略ごとの比較を述べる．実験結果で注目すべきは，全タスクセットの全ての SPM 容量において，3 種類の戦略のうちで混合活用法が最も消費エネルギー最小化を達成できている．混合活用法は，最大で，空間活用法と比較して 64 %，時間活用法と比較して 28 %の消費エネルギーを削減できている．このことから，空間活用法と時間活用法を組みあわせる混合活用法が SPM を最適に活用できる戦略であることがいえる．

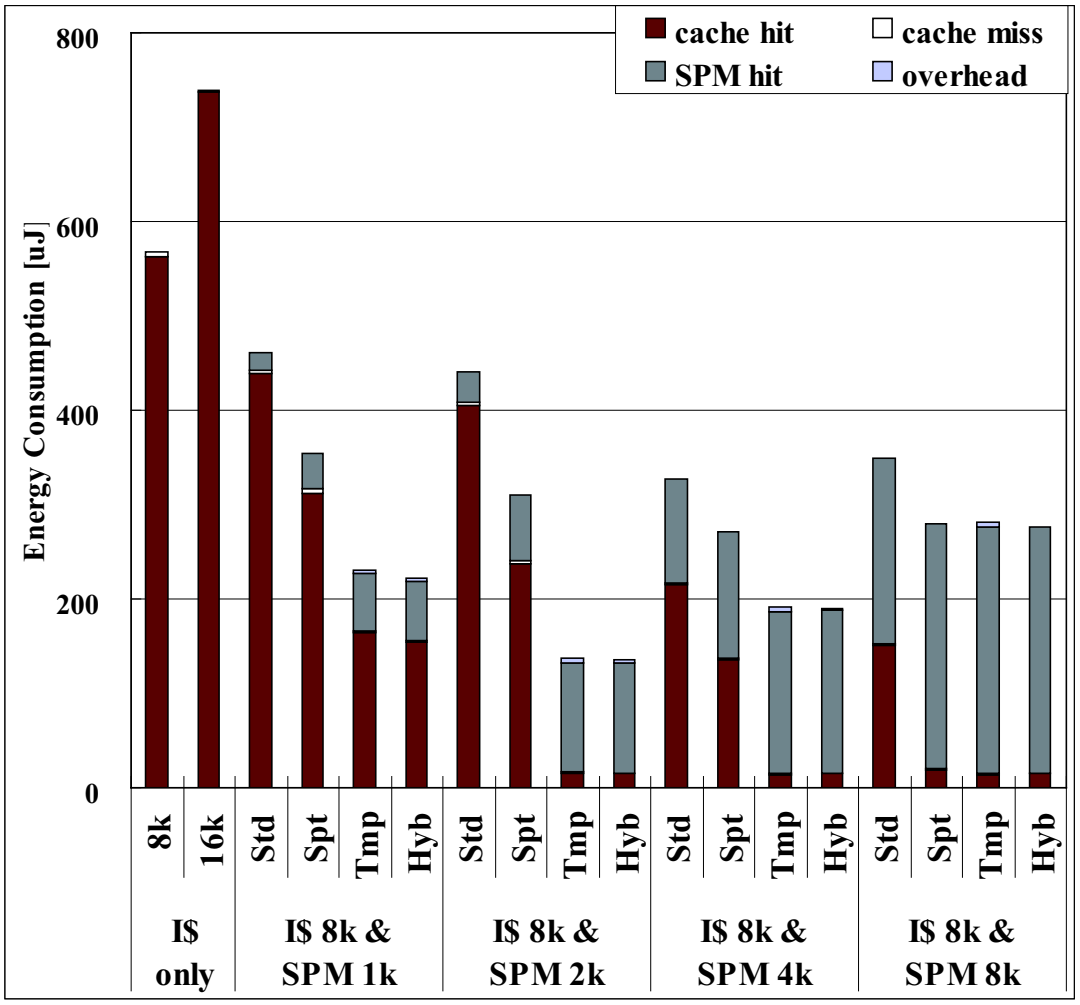


図 5.10: プリエンプティブなマルチタスク環境における ‘setB’ の結果

主記憶–SPM コード転送に掛かるオーバヘッドをみる．SPM 内容の動的な管理が必要になりうる時間活用法および混合活用法の図および表の ‘overhead’ の数値を見ると，コード転送に掛かる消費エネルギーはそれほど大きくなっていないことがわかる．コード転送に掛かる消費エネルギーは，最大でも全体の 5 %にとどまるのみであった．時間活用法は，タスクの起動時と終了時の計 2 回のコード転送が常に必要であり，終了時のコード転送は状況によっては冗長であるように思われる．しかし，SPM 内容の適切な制御管理によって SPM 領域へと配置できるコードが多くできるという利益面が大きくなり，この点が大きな問題となっていない．オーバヘッドの削減手法に工夫の余地は残るものの，本研究で提案した SPM 活用

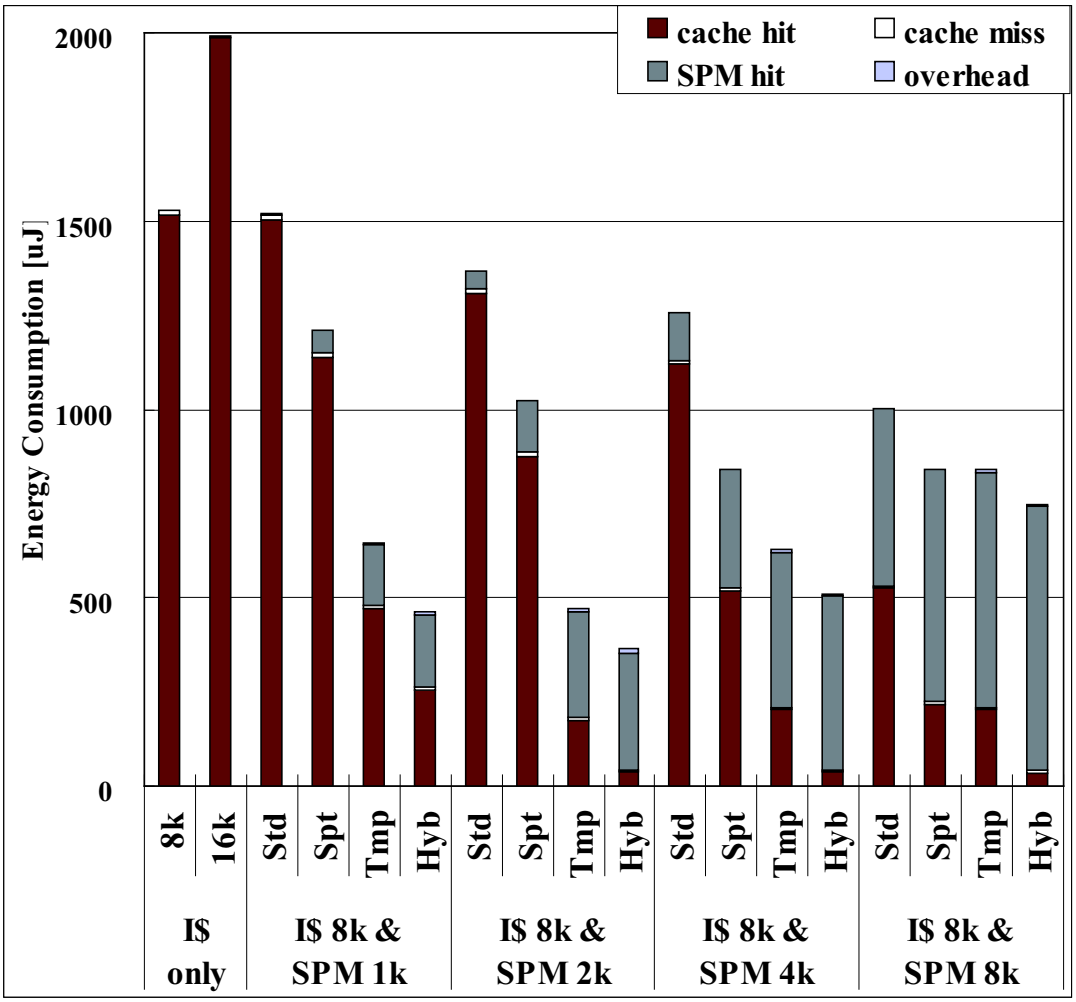


図 5.11: プリエンプティブなマルチタスク環境における ‘setC’ の結果

戦略は，プリエンプティブなマルチタスク環境において有効に消費エネルギーを削減できている．

カーネルコードのアクセスに要する消費エネルギーに触れる．カーネルコードのフェッチは，第 5.3 節で触れたように全てキャッシュを介して行われると想定したが，その消費エネルギーは，最大でも全体の 3 %程度しか発生しなかった．これは，本実験ではタスク間の同期・通信処理がない独立タスクを対象とし，カーネルの行う処理は，タスクの起動と終了処理，および，周期ハンドラの制御のためのタイマ割込み程度であったためである．しかしながら，カーネルのサービスコールを用いてタスク間通信を頻繁に行うようなアプリケーションを想定した場合には，カーネルコードも SPM 領域への配置対象として考慮する必要が出てくると考えら

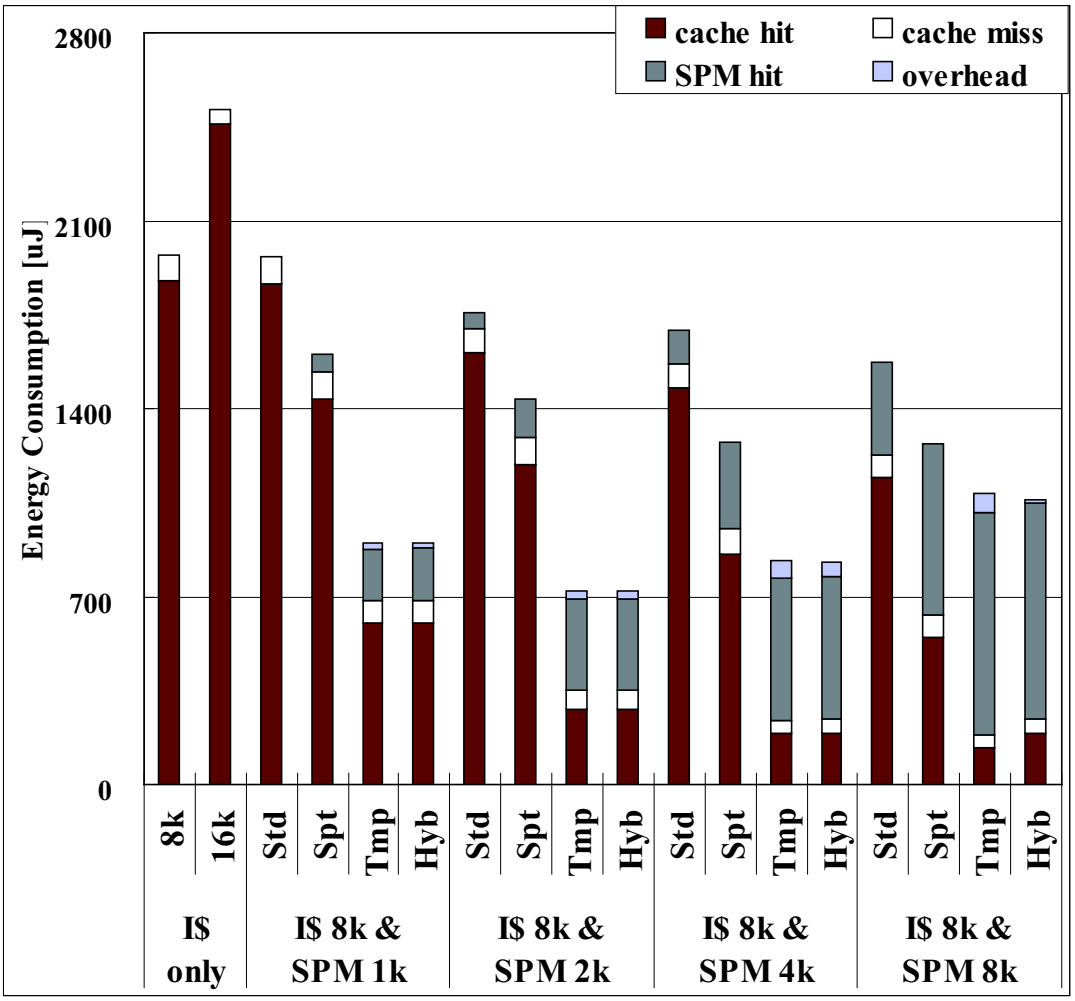


図 5.12: プリエンプティブなマルチタスク環境における ‘setD’ の結果

れる．

最後に，SPM の総容量と消費エネルギー量の関係に注目する．実験結果に着目すると，SPM の総容量を整数計画問題への入力である定数値として設定した本提案手法では，どのタスクセットでも，SPM の容量が 2 KBytes または 4 KBytes の混合活用法のときに消費エネルギー最小となることがわかる．これは，本章で提案した SPM 活用戦略を適用すると，SPM の容量を増やすことが必ずしも消費エネルギー削減に繋がらないことを示している．今回確認されたこの傾向は，消費エネルギー最小にできる SPM 容量を，変数として決定できる可能性を示唆している．

表 5.5: プリエンプティブなマルチタスク環境における CPU 使用率の評価結果 [%]

SPM size	Method	SetA	SetB	SetC	SetD
1k	Std	63.05	64.15	67.41	66.17
	Spt	63.06	64.16	67.41	66.18
	Tmp	63.16	64.19	67.44	66.22
	Hyb	63.15	64.19	67.45	66.23
2k	Std	62.90	64.15	67.40	66.14
	Spt	62.94	64.15	67.40	66.19
	Tmp	63.21	64.21	67.43	66.24
	Hyb	63.19	64.20	67.45	66.23
4k	Std	62.88	64.14	67.39	66.10
	Spt	62.84	64.13	67.39	66.18
	Tmp	63.50	64.21	67.43	66.33
	Hyb	63.28	64.15	67.39	66.28
8k	Std	62.63	64.11	67.35	66.06
	Spt	62.54	64.11	67.36	66.09
	Tmp	63.44	64.21	67.43	66.33
	Hyb	62.58	64.11	67.35	65.98

5.5.4　議論

本節では，提案した SPM 活用戦略について，消費エネルギー以外の観点からその有効性を議論する．

性能に関する評価　提案手法の性能に関する議論のため，CPU 使用率を評価した．CPU 使用率は，全体の実行時間（各タスクの起動周期の最小公倍数）に占めるタスクおよびカーネルの実行時間の割合で算出した．表 5.5 に，CPU 使用率の評価結果を示す．

表 5.5 をみると，時間活用法および混合活用法においては，その CPU 使用率が増加していることがわかる．これは，タスク切替え時の SPM 領域の管理時に，CPU の実行を中断しているためである．しかしながら，その増加量は最大でも 0.81 % とわずかである．今回は CPU の実行を中断して主記憶–SPM 間のコード転送を処理しているが，この処理が CPU 実行と並列して行えるならば，わずかながらの性能の悪化も防げると考えられる．

スケジューラビリティ　リアルタイムシステムにおいて求められる最も重要な要

件は，全てのタスクがデッドラインミスを発生させないことである．先に評価した CPU 使用率は，あまり重点を置かれない．

本研究における貢献として，SPM を持つシステムにおけるスケジューラビリティの向上がある．本研究にて提案した SPM 活用戦略を用いることによって，SPM を持つ環境でも，消費エネルギー最適化を実現しつつプリエンプティブな固定優先度ベースのリアルタイム・タスクスケジューリング方式が適用可能となった．実際に，評価実験では，全てのタスクセットでデッドラインミスを発生させることなく実行することができている．さらに，SetB，setC および setD においては，優先度および起動周期の設定が同じ非プリエンプティブなスケジューリング方式では，デッドラインミスが発生した．

入力データと実行パスの多様性　評価実験では，整数計画問題における $fetch_{i,j}$ の値は命令レベルシミュレーションにより取得した．しかし，$fetch_{i,j}$ の値は，入力データに依存して変化する可能性がある．タスクへの入力データは，その実行パスと相関性がある．このような場合は，タスクに対して複数の入力データを与え，それぞれの $fetch_{i,j}$ の値の平均を取ることで最適化すればよい．

5.6　まとめ

本章では，プリエンプティブな固定優先度ベースのリアルタイム・タスクスケジューリング方式を採用したマルチタスク環境における 3 種類の SPM 活用戦略を提案した．提案手法は，組込みリアルタイムシステムの命令メモリにおける消費エネルギー最小化に貢献する．SPM 活用戦略は，タスクの性質およびスケジューリングの特徴を使用することで，SPM 領域をタスク間で有効活用することができる．それぞれの戦略は，タスク毎およびタスク間の同時最適化を実現する整数線形計画問題として形式的に定式化した．さらに，SPM の配置内容を実行時に管理するためのワークフローを構築した．本ワークフローを採用したシステムにおけるプロファイラおよびコンパイラは，タスクの静的解析を担い，SPM 管理のための情報を生成する．システムの実行時における SPM 再配置は，リアルタイム OS およびハードウェアが担う．これらのモジュールが協調動作しながら，静的に生成された情報を基にして SPM 活用戦略を実現する．

実験により，提案手法の有効性を評価した．その結果，メモリシステムの消費エネルギーを最大で 73 %の削減を達成した．特に，高優先度タスクがより低優先度

のタスクの持つSPM領域を横取りして活用する混合活用法が，最も有効であるという結果を得た．

第6章　SPMの実行時管理機能を有するリアルタイムOS

6.1　概要

組込みシステムにおける課題として，限られた資源の管理とリアルタイム性の確保がある．前者としては，まず，効率的なメモリ資源の活用が挙げられる．組込みシステムには価格やチップ面積といった制約が厳しく，搭載されるオンチップメモリの容量は限られる．後者の課題については，システムの高いリアルタイム性が求められ，タスクのデッドライン制約を保証しなければならない．組込みシステムでは，タスクの処理の正確さに加え，その処理時間の正確さが要求される．

大規模・複雑化の一途を辿る昨今の組込みシステムにおいては，システムの処理すべきタスクが複数存在するのが一般的である．このようなマルチタスク環境では，限られたメモリ容量を有効に活用するため，タスク間でSPM領域を時間的に共有できるようにデータを再配置するのが望ましい．このため，実行されるタスクに応じて，メモリの配置内容を効率良く管理する必要がある．

本章では，主にソフトウェアの機能のみによって実現できるSPMの実行時管理機能を提案する．オンチップメモリとしてSPMを搭載するシステムを対象とし，アプリケーション実行中にSPMの内容の再配置が必要な管理手法の実現を支援するソフトウェアを設計する．これらのSPM管理手法とは，例えば第5章における時間活用法や混合活用法が該当する．複数のタスク間で共有して活用されるSPM領域の管理には，リアルタイムOSを用いたアプローチを採用する．

提案手法は，SPM管理情報および実行時SPM管理機構からなる．SPM管理情報とは，リアルタイムOSがSPM管理の実現のために持つ静的な情報のことである．この管理情報は，タスク毎に持つものとシステム全体で持つものの2種類に分類して規定する．実行時SPM管理機構は，リアルタイムOSの内部機能として設計する．タスクのスケジューリング時にSPM管理情報を参照することで，SPM

の配置内容の効率的な管理が可能となる．

さらに，本研究では，SPM 管理を制御するための API を定義する．本 API の仕様を設計し，活用が想定される適用事例やその有用性について議論する．

組込み向けカーネルの内部機能として SPM の実行時管理を実現する手法としては，文献 [11, 26] がある．これらの手法は，リアルタイム OS を用いたアプローチにより，複数のタスク間で共有して活用される SPM 領域を管理する．ただし，これらの研究における管理機構は，ラウンドロビン・スケジューリング方式だけをサポートするスケジューラとディスパッチャのみからなる小規模なカーネル上への実装を対象としている．

対象システムに専用のハードウェアを追加することで SPM の配置内容を管理する手法としては，メモリ管理ユニット（MMU）を用いる文献 [10]，独自に設計した SPM controller を用いる文献 [18] が提案されている．また，第 5.4 節において解説した SPM の管理のためのワークフローも，ハードウェアの追加によって SPM の配置内容の実行時管理を実現する手法である．

提案する SPM 管理機能は，リアルタイム OS の内部機能で実現されるため，アプリケーションのソースコードへの修正は不要である．さらには，特別なハードウェアを必要としない設計となっているため，対象システムへの新たなハードウェア資源の追加も不要である．本研究のように，実用的なリアルタイム OS 上で実装可能であり，かつ，ソフトウェアの機能のみで SPM の実行時管理を実現しつつ汎用的なタスク動作を保証できる機構を提案する例は，著者らの知る限りでは存在しない．

評価実験では，SPM 管理機能のメモリサイズと性能のオーバヘッド，および，アプリケーション実行時の消費エネルギーを示す．これらの評価を通して，本研究の有効性を明らかにする．

6.2　全体像

本研究は，オンチップメモリとして SPM を搭載し，リアルタイム OS を採用するシステムを対象としている．提案手法は，μITRON 4.0 仕様 [59] における方式である，プリエンプティブな優先度ベースのタスク・スケジューリング方式を採用したアプリケーションへ適用できる．なお，本章においては，SPM 領域へのプログ

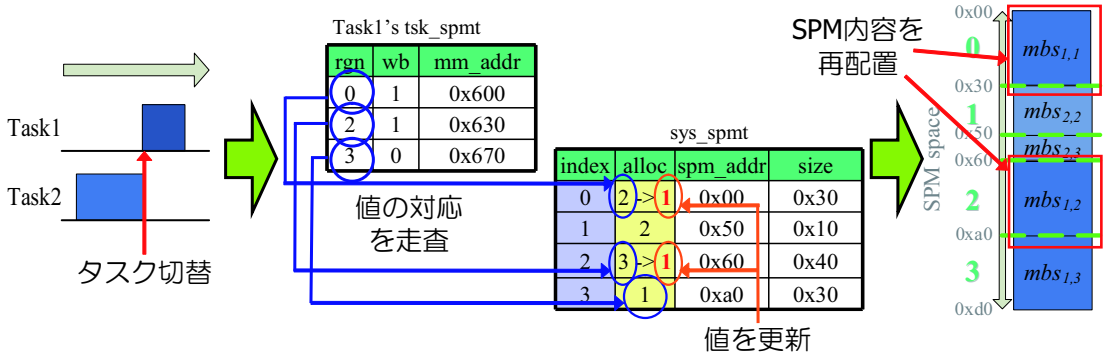


図 6.1: リアルタイム OS による実行時の SPM 管理動作の例

ラムコードやデータの配置を決定する手法は対象外としている．これらの配置問題は，第 5 章において提案した手法などを適用することによって決定されることを想定している．本章の開発成果は，第 5 章における時間活用法や混合活用法のような，実行時に SPM の内容を再配置する SPM 管理手法の実現を支援するものである．

提案する SPM 管理機能は，SPM 管理情報および実行時 SPM 管理機構から構成される．これらのモジュールは，リアルタイム OS の内部機能として設計されている．図 6.1 に，リアルタイム OS による実行時の SPM 領域の管理動作の概要を示す．

SPM 領域の管理は，リアルタイム OS 処理内のタスク切替え時に実行される．リアルタイム OS は，設計時に用意された SPM 管理情報を参照しながら，必要な SPM 領域の内容のみを適切に再配置する．図 6.1 の例では，まず，実行すべきタスクが Task1 に切り替わるタイミングにおいて，SPM 管理情報から必要な箇所を参照する．そして，再配置が必要である SPM 領域のみについて，その配置内容を更新する．このように，提案手法は，設計時に生成された SPM 管理情報を有効に活用することで，実行時の SPM 再配置におけるオーバヘッドの低減を図る．

SPM 管理機能は，タスク・スケジューリング中において新しいタスクがディスパッチされる直前に自動的に実行されるため，アプリケーションコードへの修正は不要となる．また，本機能は，日本国内の組込み分野におけるデファクト OS である TOPPERS/ASP カーネル [63] 上に実装されることを想定して設計している．

ここで，TOPPERS カーネルとは，μITRON 4.0 仕様の発展形である TOPPERS 新世代カーネル統合仕様書 [63] に準拠したオープンソースのリアルタイム OS のこ

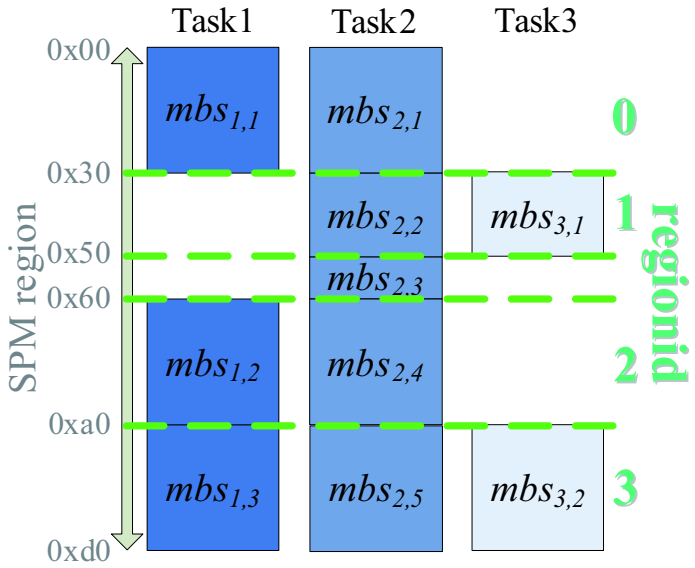


図 6.2: SPM 領域のタスク間での分配

とを指す．これは，オープンソース・ソフトウェアプラットフォームの開発および普及活動を行っている TOPPERS プロジェクト [63] からリリースされた開発成果のひとつである．μITRON 4.0 [33, 59] という組込み業界で最も多く使用されている OS 仕様に準拠したリアルタイム OS である TOPPERS/JSP を拡張・改良する形で，TOPPERS 新世代カーネルの基盤となるものとして開発された．

本研究では，実行時 SPM 管理機構はシングルプロセッサ向けの TOPPERS/ASP カーネル上に実装しているが，本統合仕様書に準拠した他のカーネルにも実装可能であると考える．また，本機構は，μITRON 4.0 仕様のタスク・スケジューリングの動作に制約を与えず，アプリケーションの汎用的な動作が保証できるという利点もある．このため，提案する SPM 管理機能は，実用的な組込みアプリケーションにおいてひろく適用することができる．

6.3 SPM 管理情報

本節では，SPM の実行時管理を支援するために設計時に用意する SPM 管理情報について解説する．SPM 管理情報は，タスク毎に持つものとシステム全体で持つものの 2 種類を規定する．資源制約が厳しい組込みシステムに適合するために，SPM 管理情報のメモリサイズが小さく抑えられるよう設計している．

SPM 管理情報は，複数のタスクに共有して使用される SPM 領域についてのみ用

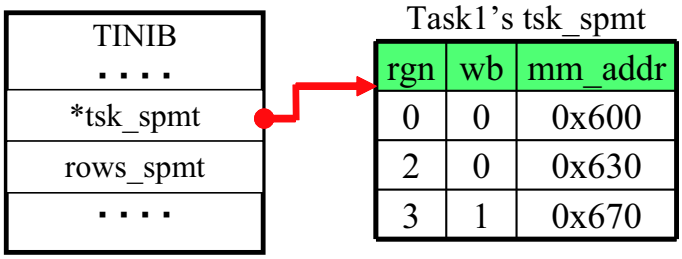


図 6.3: タスク毎に持つ SPM 管理情報

意する．図 6.2 に示すように，SPM の共有領域には，固有の ID 番号 `regionid` が付与される．図 6.2 の例では，Task1 および Task2 が共有する 0x00〜0x30 の SPM 領域に 0 が，Task2 および Task3 が共有する 0x30〜0x50 に 1 が，同様に，0x60〜0xa0 に 2 が，0xa0〜0xd0 に 3 がそれぞれ割り振られる．Task2 のみが使用する 0x50〜0x60 の SPM 領域には，`regionid` および SPM 管理情報は不要である．

SPM 管理手法に応じた SPM 管理情報は，決定された SPM の配置に応じて，システム設計時の解析ツール等で静的に生成されることを想定している．一般に，組込みアプリケーションは，その性質や実行時の振る舞いが設計時に十分に解析可能であると言われている．このため，上記の仮定は妥当であると考える．

6.3.1　タスク毎に持つ情報

図 6.3 は，図 6.2 における Task1 を例としたタスク毎の SPM 管理情報を示している．タスク毎に持つ SPM 管理情報には，タスク管理ブロック（TCB）に対して追加する情報，および，タスク毎テーブルがある．

まず，図 6.3 の左側に示す TCB に対して追加する情報を説明する．全てのタスクの TCB 内に追加する SPM 管理情報として，タスク毎テーブルへのポインタ `tsk_spmt`，および，そのテーブルの行数 `rows_spmt` を用意する．該当タスクが SPM を使用しない場合は，`rows_spmt` の値を 0 とする．このとき，実行時 SPM 管理機構は `tsk_spmt` を参照することはないため，ポインタの値は不定でよい．ただし，第 6.5 節において提案する API を用いてアプリケーション設計者が SPM 管理を明示的に実行する場合は，この限りではない．

なお，TCB に対して追加する情報は，実行時には参照されるのみで値が書き換わることはない．このため，これらの情報は，タスク初期化ブロック（TINIB）内

sys_spmt

index	alloc	spm_addr	size
0	0	0x00	0x30
1	0	0x50	0x10
2	0	0x60	0x40
3	0	0xa0	0x30

図 6.4: システム全体で持つ SPM 管理情報

および ROM 領域上に静的に配置できる．

次に，図 6.3 の右側に示すタスク毎テーブル `tsk_spmt` とは，SPM に配置するデータの情報をタスク毎にまとめたテーブルのことである．タスク毎テーブルは，‘rgn’ 列，‘wb’ 列，および，‘mm_addr’ 列の情報から構成される．

‘rgn’ 列は，対応するメモリブロックの SPM 領域上の配置先を表す．‘rgn’ 列の値は，SPM 領域の `regionid` の値に対応している．‘wb’ 列は，対応するメモリブロックが変数データか否かを表すブール変数である．値が 1 である場合には，タスクの実行中断または終了の際に SPM に配置された対応する行のメモリブロックを主記憶に書き戻す必要があることを表す．対応する行のメモリブロックがテキストや書き戻しの不要な定数データであるときは，値を 0 とする．‘mm_addr’ 列は，対応するメモリブロックの主記憶上の配置先先頭アドレスを表す．

タスク毎テーブルに含まれるこれらの情報も，読込み専用の静的なデータであり，ROM 領域に配置できる．

6.3.2　システム全体で持つ情報

システム全体で持つ SPM 管理情報としては，SPM 状態テーブル `sys_spmt` を用意する．図 6.4 は，図 6.2 の SPM 配置を例とした SPM 状態テーブルを示している．本テーブルは，実行時に値が書き換わる動的な情報である ‘alloc’ 列，および，静的な情報である ‘spm_addr’ 列および ‘size’ 列からなる．なお，図 6.4 中の ‘index’ 列は，`regionid` に対応する SPM 状態テーブルの行番号であり，実際のデータとして用意する必要はない．

‘alloc’ 列は，対応する `regionid` の SPM 領域において，現在配置されているデー

タの属するタスクの ID 番号を保持する．リアルタイム OS は，タスクのスケジューリング状況および SPM の配置状況に応じて，この列の値を適切に書き換える．なお，SPM 管理情報のうちで ‘alloc’ の列の情報のみ，そのデータを RAM 領域に配置する必要がある．

‘spm_addr’ 列は，対応する `regionid`，つまり，対応する SPM 領域の先頭アドレスを表す．‘size’ 列は対応する `regionid` のメモリサイズを表す．SPM 状態テーブルに含まれるこれらの情報は，静的な定数データであるため，ROM 領域に配置できる．

6.4　実行時 SPM 管理機構

本節では，リアルタイム OS の内部機能として設計した実行時 SPM 管理機構の仕様と動作について解説する．

図 6.5 は，実行時 SPM 管理機構の処理の流れを示している．ここで，`p_runtsk` とは，実行状態になるタスクの TCB を指すポインタである．なお，SPM 管理の効率化のため，直近に実行されたタスクの TCB を指すポインタ `p_pretsk` を追加している．

本機構の処理は，直近に実行されたタスクに属するデータのうち必要なものを SPM から主記憶に退避する処理（2〜8 行目），および，次に実行されるタスクのデータを SPM に配置する処理（9〜16 行目）から構成される．

直近に実行されたタスクについては，タスク毎テーブルの ‘wb’ 列の値を参照（4 行目）し，SPM に配置されていて主記憶に退避すべきデータを選択する．次に実行状態となるタスクについては，タスク毎テーブルの ‘rgn’ 列の値から対応する SPM 状態テーブルの行番号を走査し，その ‘alloc’ 列の値とタスク ID を比較する（10〜12 行目）ことで，再配置すべき SPM 領域とデータを選択する．‘alloc’ 列の値とタスク ID が一致している場合は，その領域はすでに必要なデータが配置されていることを表しており，SPM 管理の処理は不要となる．なお，対象タスクがタスク毎の SPM 管理情報を持たない場合には，for all 文内の処理は発生しない（3,10 行目）．以上のように，必要な領域を適切に選択していき，リアルタイム OS は該当する SPM 領域のみを再配置していく（6,13 行目）．

```
1: if p_runtsk ≠ p_pretsk then
2:   /* 直近に実行されたタスクの SPM 内容を退避 */
3:   for all i such that 0 ≤ i < p_pretsk->p_tinib->rows_spmt do
4:     if p_pretsk->p_tinib->tsk_spmt[i].wb==1 then
5:       /* SPM から主記憶にデータ転送 */
         idx = p_pretsk->p_tinib->tsk_spmt[i].rgn
6:       copy_memory(p_pretsk->p_tinib->tsk_spmt[i].mm_addr,
         sys_spmt[idx].spm_addr, sys_spmt[idx].size);
7:     end if
8:   end for
9:   /* 次に実行されるタスクの SPM 内容を配置 */
10:  for all i such that 0 ≤ i < p_runtsk->p_tinib->rows_spmt do
11:    idx = p_runtsk->p_tinib->tsk_spmt[i].rgn
12:    if TSKID(&p_runtsk) ≠ sys_spmt[idx].alloc then
13:      /* 他のタスクが使用していた領域のみ，主記憶から SPM にデータ転送 */
         copy_memory(sys_spmt[idx].spm_addr,
         p_runtsk->p_tinib->tsk_spmt[i].mm_addr, sys_spmt[idx].size);
14:      /* alloc 列の値を更新 */
         sys_spmt[idx].alloc = TSKID(&p_runtsk);
15:    end if
16:  end for
17:  /* 直近に実行されたタスクとして保存 */
     p_pretsk = p_runtsk;
18: end if
```

図 6.5: タスクの切替え時における実行時 SPM 管理機構の処理

6.5　SPM 管理のための API

本節では，SPM 配置の管理を目的とした API について解説する．図 6.5 に，提案する API である `spm_mng` の仕様を示す．第 1 の引数 `tskid` は，SPM 管理の対象とするタスクの ID 番号である．第 2 の引数 `regionid` は，管理対象とする SPM 領域の ID 番号である．これらの引数は，第 6.3 節で解説した SPM 管理情報を参照するために与える．第 3 の引数 `dst` は，データ転送の方向を定めるブール変数である．

本 API は，配置内容を自動的に処理する実行時 SPM 管理機構とは別に提供される．アプリケーション開発者は，ソースコード中に本 API を記述することで，SPM の配置内容を明示的に管理することができる．アプリケーション実行中にリアル

【API】
　ER ercd = spm_mng(ID tskid, ID regionid, bool_t dst);
【パラメータ】
　ID tskid: タスクの ID 番号
　ID regionid: SPM 領域の ID 番号
　bool_t dst: データの転送方向
【リターンパラメータ】
　ER ercd: 正常終了（E_OK）またはエラーコード
【エラーコード】
　E_ID: 不正 ID 番号（tskid が不正）
　E_PAR: パラメータエラー（regionid が不正）
　E_SYS: システムエラー（データ転送が失敗）

図 6.6: SPM 管理のための API

タイム OS は，本 API が呼び出されたタイミングで，引数で指定されたデータを適切なメモリ領域へ配置する．

本 API の有用性について議論する．まず，本 API が活用されうる適用事例として，タスク内で SPM の内容を再配置する状況が考えられる．第 6.4 節において解説した実行時 SPM 管理機構は，タスクの切替え時のみに動作するものであり，タスクの実行中には SPM の配置内容を変更することはない．文献 [3] のような手法は，メモリアクセスパターンに応じてタスクの実行中に SPM を再配置することで，システムの消費エネルギー最小化を狙っている．さらに，データのコヒーレントを取るために本 API を活用することが考えられる．タスク間またはコア間で共有するデータを SPM に配置する場合には，該当するメモリ領域へデータを明示的に書き戻すことで，他のタスクまたはコアが計算後のデータを参照できるようにする．以上のような手法を実現するために，本 API の提供が必要であると考える．

6.6　評価

提案する SPM 管理機能を，メモリサイズ，性能に関するオーバヘッド，および，アプリケーションの実行にかかる消費エネルギーの観点から評価した．

表 6.1: SPM 管理機能のメモリサイズ

		text	data	bss	total
sample1		23258	60	85048	108358
rows	0	23598	60	85052	108710
	1	23618	64	85052	108734
	2	23638	68	85052	108758
	3	23658	72	85052	108782

6.6.1　評価環境

提案機構は，TOPPERS/ASP カーネル（Release 1.4.0）[63] 上に実装した．ターゲットは東芝社製 MeP [24] の命令セットシミュレータ mepsim とし，コンパイラは GNU mep-elf-gcc 3.4.6 [54] を用いた．なお，本評価で用いたコンフィギュレーションの mepsim はそれぞれ 8 KB の命令 SPM およびデータ SPM を持つ．主記憶と SPM の間のデータ転送は，DMA によるバースト転送方式を実装した．

なお，実装の動作確認として，TOPPERS/ASP カーネルの配布パッケージに含まれる test/ ディレクトリ以下のプログラムを用いた．テストプログラムを実行した結果，実行時 SPM 管理機構を実装したカーネル上においても，全て正常に動作することを確認した．このことは，提案する実行時 SPM 管理機構を追加実装しても，アプリケーションプログラムの仕様やカーネル本体の動作にはなんら制約を与えないことを証明している．

6.6.2　メモリサイズ

SPM 管理情報および実行時 SPM 管理機構のメモリサイズを評価した．本評価には，TOPPERS/ASP カーネルの配布パッケージに含まれるサンプルプログラムである sample1 を用いた．なお，sample1 は 5 個のタスクから構成される．

表 6.1 に評価結果を示す．表 6.1 の ‘sample1’ の行は，SPM 管理機構が未実装であるカーネルにおける sample1 のメモリサイズを示している．‘rows’ 行は，SPM 管理機能を追加したカーネルにおける sample1 のメモリサイズを示している．なお，メモリサイズは，セクションごとに bytes で示してる．‘rows’ 行のそれぞれの値は，`MAIN_TASK` のタスク毎テーブルに含まれる行数，すなわち `rows_spmt` の値

表 6.2: SPM 管理機能を有するリアルタイム OS のサービスコール発行に掛かる実行サイクル数

	act_tsk	iact_tsk	wup_tsk	rot_rdq	ext_tsk
w/o spmm	344	555	485	378	472
with spmm	408	619	549	442	536

に対応している．

評価結果をみると，表 6.1 において，メモリサイズ全体の増加量は 0.3〜0.4 %程度とごく僅かである．.text セクションの増加分は，プログラムコード（依存部 64 bytes, 非依存部 244 bytes），TCB への追加情報（タスク 1 つにつき 8 bytes），タスク毎テーブル，および，SPM 状態テーブルの ROM 部分（‘spm_addr’ 列および ‘size’ 列）である．さらに，.data セクションの増加分は SPM 状態テーブルの RAM 部分（‘alloc’ 列），.bss セクションの増加分は p_pretsk の追加によるものである．

以上のことから，SPM 管理情報および実行時 SPM 管理機構のメモリサイズは，非常に小さく抑えられていることがわかる．

6.6.3　性能に関するオーバヘッド

実行時 SPM 管理機構の性能に関するオーバヘッドを，実行サイクル数により評価した．

まず，サービスコールの発行によってタスク切替えを発生させた際の処理に要した実行サイクル数を計測した．タスク切替えを伴うサービスコールとして，act_tsk, iact_tsk，wup_tsk，rot_rdq および ext_tsk を取り上げた．なお，本評価においては，全タスクが SPM 管理情報を持たないとしている．

表 6.2 の ‘w/o spmm’ 行は SPM 管理機能を持たないカーネル，‘with spmm’ 行は SPM 管理機能を持つカーネルにおける結果を表している．表 6.2 をみると，サービスコール発行に伴う実行サイクル数の増加量は 11.5〜18.6 %と小さいとはいえない．ただし，この評価結果はサービスコール発行に関する実行サイクル数だけに着目したものであることに注意されたい．一般の実用アプリケーションにおいては，サービスコールの発行回数およびタスク切替えに掛かる実行サイクル数の

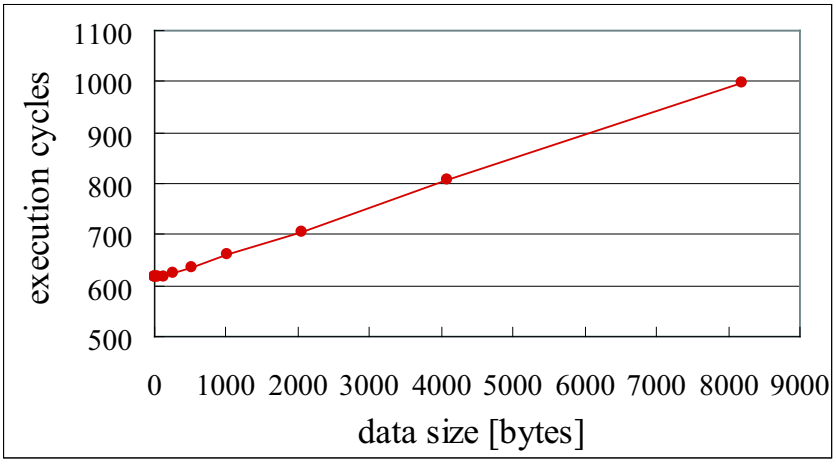


図 6.7: SPM 管理機能を有するリアルタイム OS の実行時 SPM 再配置に掛かる実行サイクル数

割合は小さい．つまり，アプリケーション全体の実行サイクル数からみれば，実行時 SPM 管理機構のオーバヘッドは許容できると考えられる．

次に，SPM の再配置に掛かるオーバヘッドを評価した．第 6.5 節において提案した API を実装し，主記憶からデータ SPM へ指定サイズ分の転送および SPM 再配置を実行した際のサイクル数を計測した．

図 6.7 の横軸がデータ転送サイズ，縦軸が API の処理に要した実行サイクル数である．この結果，SPM の全領域に対して再配置を行ったとしても，実行サイクル数は高々1000 サイクル程度に収まっていることがわかる．さらに，データ転送のサイズと実行サイクル数には線形性があることが確認できる．このことは，SPM へ再配置するをデータを決定する際にそのサイズと実行サイクル数を線形式として考慮した探索が可能であることを示唆している．

6.6.4 消費エネルギーに関する評価

SPM 管理機能を有するリアルタイム OS の有効性を，消費エネルギーによって評価した．

評価には，EEMBC [52] の AutoBench，OABench および TeleBench，および，MiBench [15] からプログラムを選定し，合成タスクセットを作成した．各プログラムを周期タスクとし，表 6.3 に示す 13 個のタスクを含む評価アプリケーション

表 6.3: SPM 管理機能の消費エネルギーに関する評価のためのタスクセット

benchmark	task name
AutoBench	a2time01, basefp01, canrdr01, pntrch01, puwmod01, tblook01
OABench	bezier01fixed
TeleBench	fft00, viterb00
MiBench	basicmath, patricia, qsort, stringsearch

表 6.4: SPM 管理機能を用いて実行した際の評価アプリケーションの消費エネルギー

	CPU	on-chip	off-chip	total
spt	188.12	184.19	26.88	399.19
tmp	153.61	128.22	20.15	301.98

を構成した．消費エネルギーは，文献 [13] において開発された見積もりツールを用いて測定した．各タスクの起動周期は平均実行時間の 25 倍程度となるように設定し，スケジューリング方式はレイトモノトニック方式を採用した．

評価では，各タスクのプログラムコードを命令 SPM への配置対象とし，配置の決定には第 5 章の第 5.3.1 節および第 5.3.2 節において提案した空間活用法および時間活用法を用いた．前者は，各タスクが SPM 領域を静的かつ排他的に活用する手法であり，SPM 管理機能を使用しない．後者は，タスクの切替え時に，実行時 SPM 管理機構が適切なプログラムコードを命令 SPM に再配置する．

表 6.4 に，評価結果を示す．‘spt’ 行が空間活用法の ‘tmp’ 行がリアルタイム OS 上の SPM 管理機能を使用して時間活用法を適用したときの消費エネルギー（単位は mJ）を表している．消費エネルギーは，CPU，オンチップメモリおよびオフチップメモリに分類して示した．

評価の結果，SPM 管理機能を用いた手法により，全体の 24.4 %の消費エネルギーを削減できた．オンチップメモリのみについて着目すると，30.39 %の消費エネルギー削減を達成している．このことから，本研究で提案した SPM 管理機能を有効活用することにより，組込みシステムの消費エネルギー最適化を実現できることがわかる．

6.7　まとめ

本研究では，SPM の配置内容を管理するための SPM 管理情報および実行時 SPM 管理機構を提案した．SPM 管理情報は，タスク毎に持つものとシステム全体で持つものに分類され，小さなメモリサイズで静的に用意することができる．リアルタイム OS の内部機能として設計された実行時 SPM 管理機構は，適切な SPM 領域のみを小さなオーバヘッドで効率よく管理する．

提案した SPM 管理機能を TOPPERS/ASP カーネル上に実装し，その有用性を評価した．評価の結果，提案手法は，小メモリサイズかつ低オーバヘッドで実装できることが確認できた．さらに，これまでに提案してきた SPM 活用手法と組み合わせて活用することで，組込みシステムの消費エネルギー最適化が実現できた．

第7章　多階層の協調による消費エネルギー最適化フレームワーク

7.1　概要

組込みシステムの消費エネルギーを最小化するためには，コンパイラやリアルタイム OS，プロセッサといった複数の設計階層における要素技術を総動員することが望ましい．設計階層を跨いだ横断的な技術を適用できれば，最適化の可能性が広がる．しかし，個別の要素技術を単に組み合わせるだけでは，それぞれの最適化を打ち消しあうことも考えられる．個々の技術の最適化効果を効率的に活かすためには，各設計階層における要素技術を注意深く統合する必要がある．

本章では，組込みリアルタイムシステムのための協調最適化フレームワークを提案する．多階層における要素技術を統合的に適用することで，対象とするシステムに求められる QoS を保証しつつ，実行時の消費エネルギーを最小化することを目指す．ここで，QoS とは，システムの計算精度や信頼性をあらわす指標のことをいう．特に，組込みシステムにおいては，リアルタイム性が求められることが多い．本研究では，特にリアルタイム性（具体的には最大応答時間）を保証するという制約下で，平均消費エネルギーを最小化することを目標とする．

協調最適化フレームワークは，組込みソフトウェアの開発環境として提供する．ソフトウェア開発環境は，システム設計時における解析・最適化ツールチェーン，および，実行時の最適化を担うリアルタイム OS からなる．ソフトウェア設計者は，本研究における成果物を開発工程において活用することで，組込みシステムの消費エネルギー最適化を自動的に実現できる．

本研究における協調最適化フレームワークには，本論文においてこれまで提案してきた SPM を活用したマルチタスク環境におけるメモリ配置最適化技術に加え，著者が共同研究などで開発してきた複数の要素技術が統合的に含まれている．具

体的には，ハードウェア構成を変更するポイント（チェックポイント）を決定するプログラム解析技術，チェックポイントにおいて最適なハードウェア構成を決定する技術，ハードウェア構成を実行時に変更可能なプロセッサ技術，データ SPM の活用によりスタックデータのメモリアクセスにおける消費エネルギーを最小化する技術，および，命令セットシミュレーションを元に消費エネルギーを正確に見積もる技術である．

さらに，提案するフレームワークは，タスク毎，タスク間，実行時の 3 段階による最適化の形態をとる．設計時のタスク毎およびタスク間のフェーズにより，アプリケーションの振る舞いや性質を解析していく．組込みシステムにおいては，どのようなアプリケーションソフトウェアが実行されるかはわかっているのが一般である．このため，アプリケーションの性質を最大限に解析し活用して，消費エネルギーの削減を行う方針をとる．設計時の解析結果は，リアルタイム OS が実行時最適化のために活用する．対象とするソフトウェアを段階的に最適化していくことにより，各要素技術の効果を打ち消しあうことなく，効率的に活かすことができる．

プロセッサや OS，コンパイラといった複数の設計階層の協調による消費エネルギー最適化技術は，これまでにもいくつか研究されている．

文献 [1] では，コンパイラと OS の協調による動的電圧・周波数制御（DVFS: Dynamic Voltage and Frequency Scaling）手法が提案されている．コンパイラによって抽出された情報を基にして，OS 上の周期的な割込みルーチンが DVFS を実行する．文献 [28] では，タスク毎およびタスク間の最適化を統合した DVFS 手法が提案されている．文献 [4] では，タスク毎 DVFS のための効率的なプロファイル手法が提案されている．以上で述べた DVFS に関する既存研究は，コンパイラと OS による協調最適化を実現するが，メモリ配置最適化は考慮されていない．

文献 [41] では，複数階層の協調による最適化フレームワークが提案されている．ただし，文献 [41] における手法は，メディア系アプリケーションを対象としている．文献 [31] では，マルチコア SoC における性能向上のための統合最適化手法が提案されている．この研究において定式化された整数計画問題は，コア間のタスク配置，コア毎のタスクスケジューリング，および，タスク間の SPM 配置を同時に決定できる．しかしながら，これらの研究の対象とするシステムは，リアルタイム性が要求されないソフトリアルタイムな環境である．さらに，これまでに述べた既存手法は，ハードウェアの持つスケーラビリティの活用を考慮していない．

本研究は，コンパイラ，リアルタイム OS，および，プロセッサという 3 つの設計階層に跨る統合的な消費エネルギー協調最適化フレームワークを提供する世界初の研究である．提案手法は，組込みリアルタイムシステムのアプリケーション開発に適用できるツールチェーンおよびリアルタイム OS として実装する．本フレームワークにおける最適化は自動的に処理されるため，組込みソフトウェアの設計者の開発負荷低減にも貢献する．開発成果の適用事例として，テレビ会議システムを例題として取り上げる．これを通して，開発したツールチェーンおよびリアルタイム OS が消費エネルギー最適化に貢献し，かつ，実用アプリケーションに適用可能であることを示す．

7.2　消費エネルギー協調最適化フレームワークの全体像

7.2.1　対象とするシステム

本研究の目的は，組込みアプリケーションの平均消費エネルギーを最小化することである．対象は，正しい計算結果とともに高いリアルタイム応答性が求められるハードリアルタイムな組込みシステムである．組込みリアルタイムシステムにおいて最も重要視されることは，全てのタスクがデッドライン制約を守る（そのデッドライン時間までに正常に処理を終える）ことである．このため，本研究における協調最適化フレームワークにおいても，システムに求められるリアルタイム性（デッドライン制約）を制約条件として保証した上で，目的とする平均消費エネルギーの最適化を達成する．

本フレームワークの適用対象は，シングルプロセッサシステムである．アプリケーションは，それぞれ独立に動作する周期タスクまたは最小起動間隔が既知の非周期タスクによって構成されることを仮定している．全てのタスクは静的な優先度を持ち，プリエンプティブな固定優先度ベースのスケジューリング方式によってタスクの実行順が決定される．

7.2.2　ワークフロー

本研究では，各タスクの実行トレース群を解析対象として扱う．図 7.1 に示すように, 協調最適化フレームワークに与える入力情報は，各タスクのソースコードおよびテストデータである．これらの情報を入力としてサイクルレベルシミュレー

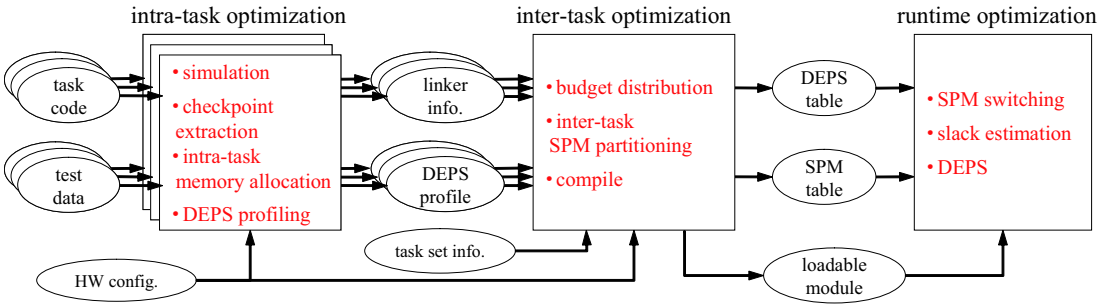


図 7.1: 消費エネルギー協調最適化フレームワークの全体像

ションを実行し，解析対象とする実行トレース群を取得する．実行トレース群を解析対象として扱うのは，それらの情報には，アプリケーション実行時の実際に動作した振る舞いや性質が反映されている [7] からである．さらに，高信頼ソフトウェアの開発にあたっては，プログラムの様々な実行パスを通るようなテストケースを用意するのが通常で，そのテストケースを入力としてプログラムを動作させると，網羅性の高い実行トレースが取得できる．すなわち，実行トレース群は，システム開発時に容易に取得することができる．実行トレース群を基にして協調最適化フレームワークが活用できるため，本手法は実用アプリケーションにひろく適用可能である．

図 7.1 は，本研究で提案する消費エネルギー協調最適化フレームワークの最適化の流れを示している．本フレームワークは，タスク毎，タスク間および実行時最適化の 3 段階の形態をとる．

第 1 のタスク毎最適化フェーズでは，まず，大量の実行トレース群から，それぞれのタスクの振る舞いや性質を解析する．第 2 のタスク間最適化フェーズでは，前段の解析結果を基にして，アプリケーションレベル（マルチタスクレベル）での振る舞いや性質を解析する．また，設計時の解析結果として，実行時最適化を支援するためのテーブル情報を生成する．第 3 の実行時最適化フェーズでは，設計時に生成された 2 種類の管理テーブルを参照することで，対象アプリケーションの消費エネルギーを最適化しながらその処理を実行する．

設計時における最適化フェーズは，さらに複数のステップに分かれた形態をとる．各ステップにおける要素技術の適用順は，互いの最適化効果を打ち消すことなく，効率的に活かすことができるように設計している．さらに，最適化のフローは，そのステップを後戻りする必要のない一方向に流れる形態となっている．

表 7.1: 消費エネルギー協調最適化フレームワークの実装対象とする開発環境

コンパイラ	GNU GCC 3.4.6
アセンブラ，リンカ	GNU Binutils 2.19
C 標準ライブラリ	GNU Newlib 1.17.0
リアルタイム OS	TOPPERS/ASP カーネル Release 1.4.0

消費エネルギー協調最適化フレームワークを上記のようなワークフローとした理由は，以下の通りである．

まず，設計時にできる限りのアプリケーション解析を行うことで，リアルタイム OS の実行時におけるオーバヘッドを小さく抑えることが期待できる．段階的な形態をとり，組込みシステムの性質を活用することで，リアルタイム OS の担う処理を限定しつつ消費エネルギー最適化を達成できる．さらに，最適化フレームワークに対して他の要素技術を追加できる形態となっている．組込みシステムのさらなる消費エネルギー最適化を実現するには，より多くの最適化技術を適用できることが望ましい．本研究で取り扱わなかった要素技術についても，本フレームワークには後から容易に拡張可能である．

7.2.3　実装方針

本研究では，提案する消費エネルギー協調最適化フレームワークを実用システムに適用可能なソフトウェア・ツールチェーンとして整備する．ツールチェーンとして，静的解析を担うプロファイラ，設計時の最適化を担うコンパイラ，および，実行時の最適化を担うリアルタイム OS を実装する．

実装のターゲット・プロセッサとして，東芝社製 MeP [24] を採用した．MeP は，メディア処理用途に特化した，コンフィギュレーションと機能拡張を選択可能なカスタムプロセッサである．クロス開発環境には，GNU mep-elf 環境 [54] を用いた．設計時の最適化ツールチェーンは，このクロス開発環境に沿って実行できるように実装する．リアルタイム OS は，オープンソースのシングルプロセッサ向けカーネルである TOPPERS/ASP カーネル [63] をベースとし，この機能拡張により実装する．表 7.1 に，消費エネルギー協調最適化フレームワークの実装のベースとするツールのバージョンを示す．

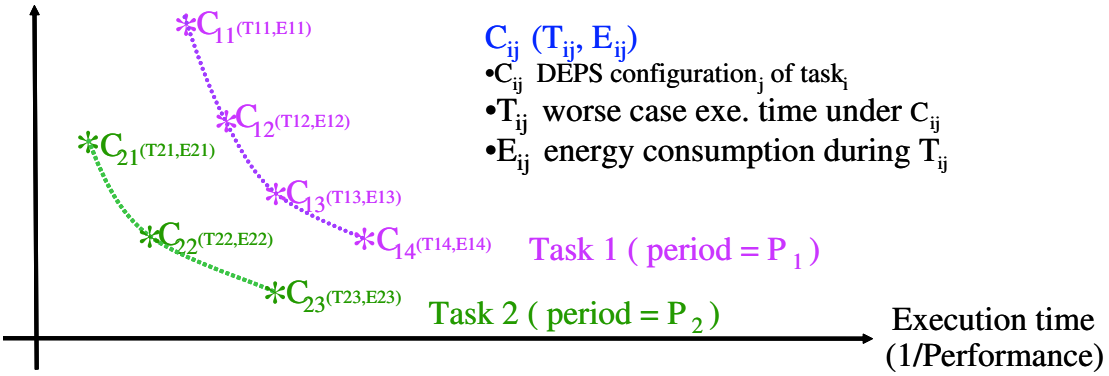


図 7.2: ハードウェアの構成による消費エネルギーと性能の関係

7.3　要素技術

本節では，消費エネルギー協調最適化フレームに組み込まれているそれぞれの要素技術について解説する．なお，本節で概説する技術の一部は，著者が共同研究などを通して開発してきたものが含まれる．

7.3.1　動的エネルギー・性能制御技術

本フレームワークの中心的な要素技術として，動的エネルギー・性能制御（DEPS: Dynamic Energy and Performance Scaling）技術 [42] がある．DEPS は，システム実行時に動作周波数と供給電圧を適切に制御する DVFS 技術 [23] を一般化した手法である．つまり，DEPS では，アプリケーションの振る舞いに応じて，ハードウェアの変更可能な構成要素を動的に制御する．ここで行うハードウェア構成の変更には，動作周波数と電圧を変化させることはもちろん，キャッシュメモリの容量や構成の変更，さらには，異なるアーキテクチャを持つ（例えば，パイプライン段数の異なる）プロセッサを用意しておき，適切なものを選んで動作させることも含まれる．

図 7.2 に示すように，一般にハードウェアの構成には，消費エネルギーと性能に関するトレードオフ関係がある．すなわち，要求される処理量の多い時には高性能・大エネルギー消費のハードウェア構成で動作させ，そうでない時は低性能・小エネルギー消費の構成で動作させるようにする．DEPS は，ハードウェアの各変更可能要素を適切に制御することで消費エネルギー最適化を実現する．

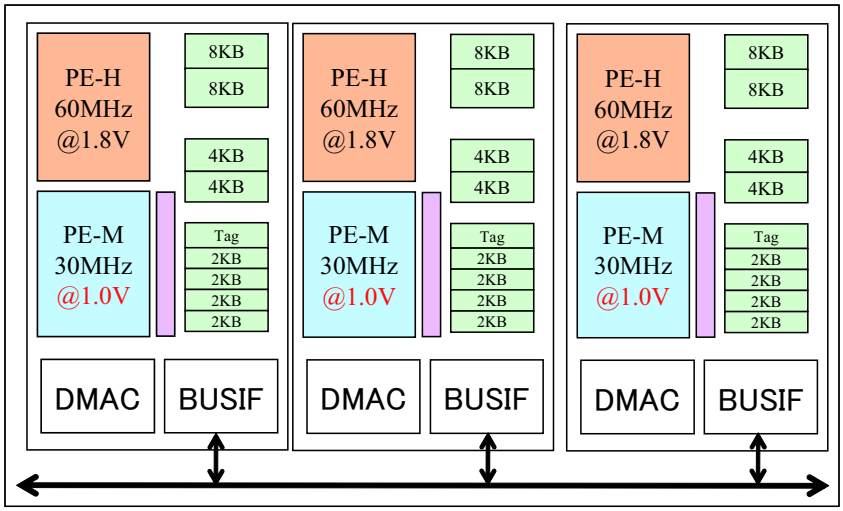


図 7.3: マルチパフォーマンスプロセッサの構成

DEPS 技術を適用する際の課題としては，消費エネルギーと性能の関係を容易に定式化できないことが挙げられる．これは，DEPS の取り扱うハードウェア構成要素の種類によって，それらのトレードオフ関係が変動するためである．この課題を解決する方策として，文献 [45, 46] において，DEPS プロファイリング技術を提案している．本手法は，実行トレース群を基にして DEPS が適用された際のプログラムの実行時間および消費エネルギーを解析するプロファイラ技術である．また，DEPS プロファイリング技術を支援するために，実行トレースから消費エネルギーの情報を取得できる見積りツール [13] を開発した．

7.3.2　マルチパフォーマンスプロセッサ

前述の DEPS を実現するためには，プロセッサが，消費エネルギーと性能をトレードオフさせる機能を持っていることが必要である．前節の DEPS 技術の有効性を検証するためのプロセッサ技術として，マルチパフォーマンスプロセッサ [16] を開発した．図 7.3 に示すように，本プロセッサは，2 つのプロセッシング・エレメントおよび容量とウェイ数を動的に変更できる命令キャッシュを持つ．

それぞれのプロセッシング・エレメントは，異なる動作周波数および供給電圧に最適化して設計されている．コアの命令セットは，東芝社製 MeP [24] に準拠している．命令キャッシュはシステム動作中にウェイ数を変更でき，1-way/2KB から 4-way/8KB までの構成をとることができる．これらのプロセッサ構成の変更は，

リアルタイム OS から制御できるレジスタの値変更で実現される．マルチパフォーマンスプロセッサの大きな特徴としては，構成の変更にかかるオーバヘッドが商用の DVFS 可能プロセッサよりも非常に小さく抑えられることが挙げられる．

7.3.3 実行トレースマイニング

DEPS の制御を効果的に行うためには，アプリケーションプログラムの性質がわかっていることが望ましい．前述の通り，本研究ではアプリケーションの性質を最大限に活用する方針としているが，ソフトウェア開発者の負担を軽減するためには，アプリケーションの性質をソフトウェア設計者が与えるのではなく，できる限り機械的に調べるのが望ましい．

プログラムの性質を機械的に調べる解析方法として，実行トレースマイニング [35] 技術を開発した．これは，命令セットシミュレータを用いて取得したプログラムの大量の実行トレースを解析することにより，実行時の振る舞いや性質を自動的に抽出するプロファイル技術のことをいう．本研究では，実行トレースマイニングにより，プログラムの振る舞いが大きく変動する箇所をチェックポイントとして探索することに利用した．探索されたチェックポイントは DEPS の有効性を向上することができる箇所とみなすことができ，この箇所においてハードウェア構成を変更することで，実行時の消費エネルギー最適化を実現できる．

7.3.4 メモリ配置最適化

メモリ配置最適化の技術としては，SPM の活用に着目する．頻繁にアクセスされるプログラムコードおよびデータを SPM 領域に集中配置することにより，消費エネルギー最適化を実現する．コードおよび静的データについては，第 5 章にて提案したマルチタスク環境における SPM 活用戦略を適用する．実行時の SPM の管理は，第 6 章にて開発したソフトウェア機能を活用する．

本研究ではさらに，文献 [14] において提案されているスタックデータに対するメモリ配置最適化手法を適用する．本手法は，図 7.4 に示すように，頻繁にアクセスされる一部の関数のスタックフレームを SPM 領域に配置することで消費エネルギーの削減を図る．スタックポインタの移動は，warp/unwarp 命令をソースコードに挿入することで実現される．

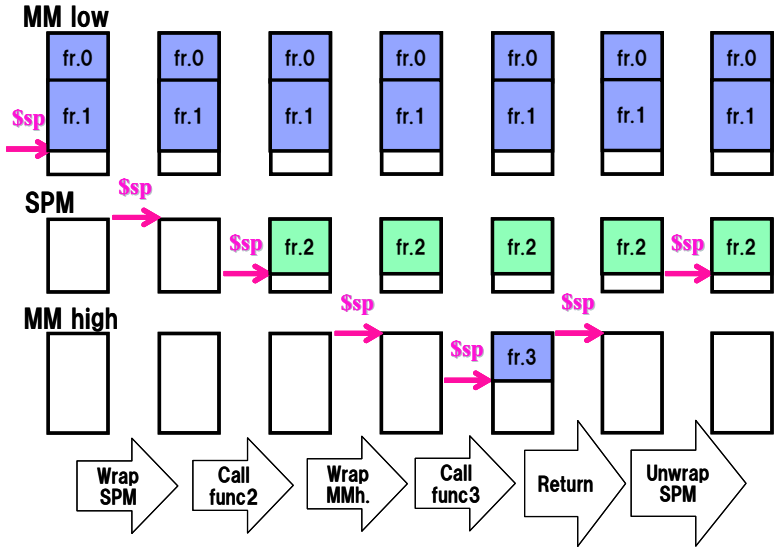


図 7.4: スタックデータに対するメモリ配置最適化

7.4　タスク毎最適化

本節では，消費エネルギー協調最適化フレームワークの 1 段階目であるタスク毎最適化フェーズの詳細を，処理のステップごとに解説する．

図 7.5 は，タスク毎最適化フェーズの流れをあらわしている．タスク毎最適化フェーズに含まれるステップは，シミュレーションによる実行トレース取得，実行トレースマイニングによる DEPS に有効なチェックポイントの抽出，タスク毎のメモリ配置最適化，および，DEPS プロファイル生成である．タスク毎最適化における入力情報は，タスクのソースコードおよびテストデータとなる．出力情報は，修正ソースコード，タスク毎のリンカ情報，および，DEPS プロファイル（第 7.4.4 節で解説）となる．

7.4.1　シミュレーション

タスク毎最適化フェーズの最初のステップとして，入力情報であるソースコードおよびテストデータから，大量の実行トレース群を取得する．本研究では，MeP プロセッサの命令セットシミュレータを用いて実行トレース群を取得した．

テストデータは，プログラムの通過する実行パスに相関性がある．本フレームワークでは，プログラムの実行パスを十分に網羅できる実行トレース群が取得できることが望ましい．さらに，それぞれのテストデータは，その出現頻度に応じ

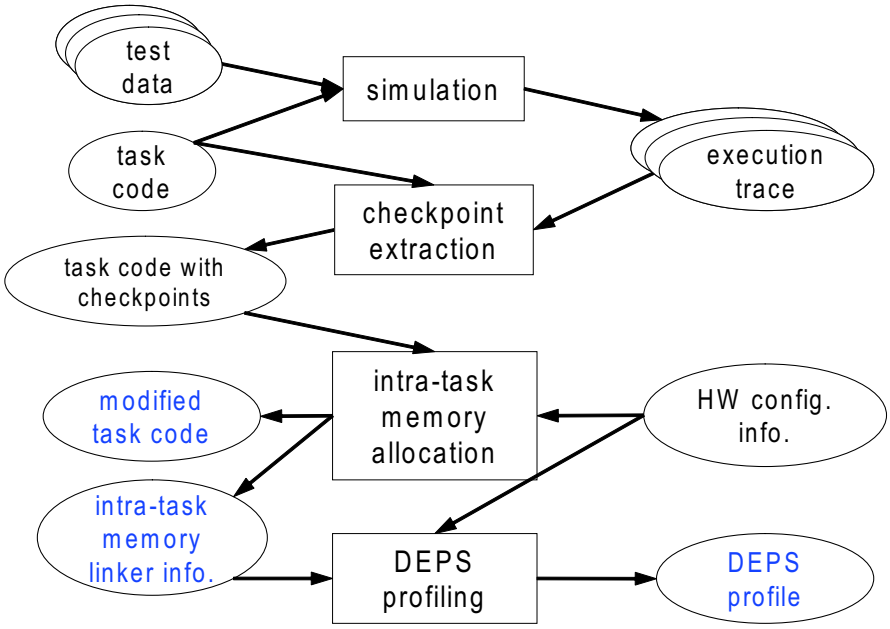


図 7.5: タスク毎最適化フェーズの流れ

て重み付けされているとなおよい．これらの条件が満たされることは，フレームワークの最適化効果をより高められることに繋がる．

7.4.2 チェックポイントの抽出

本フレームワークでは，DEPS のためのチェックポイント抽出のステップを，他の処理に先がけて行う．これは，チェックポイント抽出処理は，他のステップの最適化処理への影響が小さく，タスク毎の大まかな解析のみで目的を達成できるためである．

本ステップの処理の目的は，タスク中の有効なチェックポイントを抽出し，ソースコードに対してチェックポイントを挿入することである．チェックポイントとは，文献 [4] における定義と同様に，ハードウェアの構成を変更すべきプログラム箇所の候補のことを指す．この目的を達成するため，本フレームワークでは，大量の実行トレース群からプログラムの振る舞いや性質を抽出する技術である実行トレースマイニング [35] を適用する．

本研究では，実行トレースマイニングによるチェックポイントの抽出処理，および，ソースコードのチェックポイントの挿入処理を自動的に行うソフトウェアツールを開発した．チェックポイントは，DEPS 処理を行う関数の呼び出し口となる API として挿入される．

チェックポイントは，DEPS の有効性がより向上できる箇所に挿入できることが求められる．つまり，タスクの最大残り実行時間が大きく変化する箇所，または，タスクの性質が大きく変化しエネルギー効率の良いハードウェア構成が変化する箇所である．前者について具体的には，条件分岐で実行時間の短い方のパスへ分岐した直後や，ループ回数が変動するループを抜けた直後がこれに該当する．後者は，例えば，プログラムを効率的に実行するために必要なキャッシュの容量が大きく変化する場所では，プロセッサのキャッシュ容量を変更することで，消費エネルギーが削減できる場合がある．このように，プログラムの性質が大きく変化する場所に，チェックポイントを挿入する．

ただし，チェックポイントを多く挿入し過ぎると，チェックポイントの処理オーバヘッドにより消費エネルギーが増加してしまうおそれがある．このため，有効なチェックポイントを選定する手法も文献 [35] および本手法において適用している．

7.4.3　タスク毎メモリ配置最適化

タスク毎最適化フェーズにおける次のステップは，SPM を考慮したタスク毎でのメモリ配置最適化である．本ステップにおける処理は，スタックデータの配置最適化，および，プログラムコードと静的データの配置最適化の 2 種類がある．

スタックデータについては，文献 [14] による手法を用いて割当て先を決定する．本手法では，実行トレース群より得られるメモリアクセス履歴の情報を基にして，頻繁にメモリアクセスのある関数のスタックフレームを選定する．そして，それらを SPM 領域に配置できるよう，プログラムコードにスタックポインタの値を制御する専用の命令列（warp/unwarp 命令 [14]）を挿入する．本処理によって，タスクのソースコードに対する修正は完了となる．

プログラムコードおよび静的データについては，メモリアクセス履歴から頻繁にアクセスされる実行トレース群より得られるメモリアクセス履歴の情報を基にして，メモリ領域の割当て先を決定する．割当て先の決定には，第 5 章における時間活用法を，タスク毎に適用する．本ステップにおける処理では，タスク毎では SPM の使用量のみが決定されることに注意されたい．SPM 領域内でのそれぞれのデータの配置先アドレスは，タスク間最適化フェーズによって決定される．

CP0	CP1	CP2	CP3	WCET	AEC
config2	config2	config1	config4	23.7	433.2
config3	config3	config1	config4	28.3	345.1
config3	config4	config2	config6	32.1	301.5
config4	config4	config2	config6	35.2	273.8
config6	config6	config4	config6	45.1	205.2

図 7.6: DEPS プロファイルの例

スタックデータ配置最適化のためのソースコード修正処理および，プログラムコードおよびデータの配置先を決定する処理についても，それぞれソフトウェアツールとして実装した．第 7.4.2 節におけるツールも自動化されているため，開発者は対象とするアプリケーションに対して最適化のためのソースコード修正作業を負う必要はない．本ステップの処理によって，最適化のために修正されたタスクのソースコード，および，タスク毎のリンカ情報が出力される．

7.4.4　DEPS プロファイル生成

タスク毎最適化フェーズにおける最後のステップは，文献 [46] において開発した DEPS プロファイル生成処理である．本ステップで出力される DEPS プロファイルとは，図 7.6 に示すように，各チェックポイントで選択するハードウェア構成の組み合わせ毎に，最大実行時間（WCET: Worst-Case Execution Time）および平均消費エネルギー（AEC: Average Energy Consumption）を記録したテーブル情報のことをさす．

本ステップでは，まず，決定済みのメモリ配置下で，サイクル精度シミュレーションによるタスクの実行トレース群を取得する．この実行トレース群を基にして，タスク毎に最大実行時間および平均消費エネルギーを算出する．実行時間および消費エネルギーの算出には，文献 [13] において開発された見積もりツールを用いた．この研究では，まず，ターゲットプロセッサのポストレイアウト設計に対してゲートレベルシミュレーションを適用し各モジュールのパラメータ値を取得する．さらに，実行トレース群を用いて多重線形回帰分析を試行し，そのパラメータ見積もり精度を向上させている．

DEPS プロファイル生成処理は，各チェックポイントで選択するハードウェア構

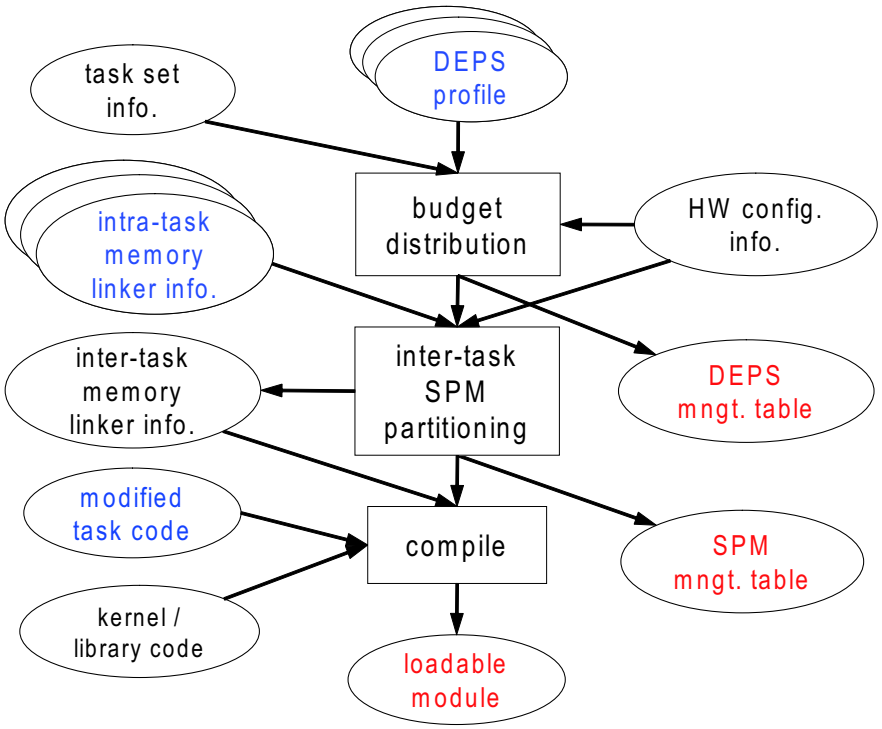


図 7.7: タスク間最適化フェーズの流れ

成の組み合わせ毎に，最大実行時間および平均消費エネルギーを算出する必要がある．しかし，DEPS 技術において考えられるハードウェア構成の組み合わせは膨大であり，全ての組み合わせでこれらの値を算出することは，解析時間の面から現実的ではない．この問題を解決するため，文献 [46] による手法では，パレート最適となる最大実行時間および平均消費エネルギーをとるハードウェア構成の組み合わせのみを高速に選定するアルゴリズムを提案している．パレート最適でない値を取る組み合わせについては，実行時にその構成をとることはないため，DEPS プロファイルから削除される．DEPS プロファイル生成処理は，タスク毎最適化フェーズの最後段で実施するため，より正確な解析精度を実現することができる．

7.5　タスク間最適化

本節では，消費エネルギー協調最適化フレームワークの 2 段階目であるタスク間最適化フェーズの詳細を，処理のステップごとに解説する．図 7.7 は，タスク間最適化フェーズの流れをあらわしている．本フェーズでは，各タスクのデッドラインや優先度，起動間隔といったタスクセットの情報を用いて，アプリケーションレベルでの振る舞いや性質を解析する．さらに，前段のタスク毎最適化フェーズに

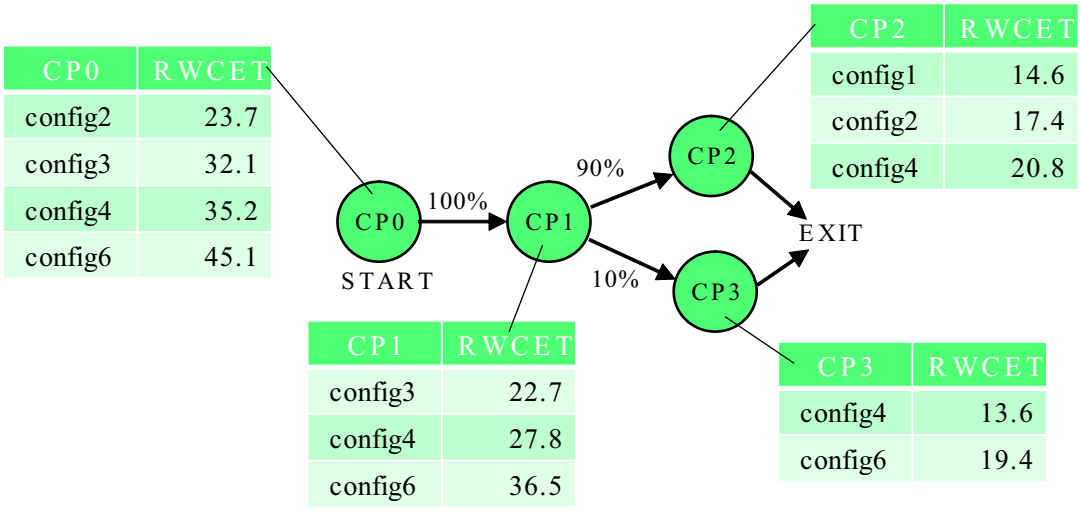


図 7.8: DEPS 管理テーブルの例

おける出力情報を基にして，タスク間での静的最適化を行う．本フェーズにおける出力情報は，DEPS と SPM 管理に関する 2 種類のテーブル情報である．後者については，第 6 章の第 6.3 節で解説した SPM 管理情報に対応する．なお，本フェーズでは，実行時の消費エネルギー最適化を実現できる対象アプリケーションの実行可能モジュールも同時に生成する．

7.5.1　DEPS 管理テーブル生成

本ステップでは，まず，マルチタスク環境でのシミュレーションを試行し，アプリケーションレベルのプログラムの振る舞いや性質を解析する．そして，各タスクがデッドライン制約を満たしつつ平均消費エネルギーを最小化できるよう，スケジューリング中に発生するバジェット時間をそれぞれのタスクに分配する処理を行う．タスクのスケジューリング時にプロセッサの負荷に余裕が生じた場合，その余裕分を各タスクに分配し，タスクのデッドラインが満たされる範囲で，なるべく低性能・小エネルギー消費のハードウェア構成でタスクを動作させるようにする．この処理を実現するため，文献 [42] において，タスクセット情報と DEPS プロファイルを用いた整数線形計画問題を提案している．

本ステップにおける出力情報は，各チェックポイントにおける DEPS 管理テーブルである．DEPS 管理テーブルとは，図 7.8 に示すように，各チェックポイントにおいて選択可能なハードウェア構成の組み合わせに応じた最大残り実行時間を記した情報のことを指す．テーブル中の要素は，最大残り実行時間の昇順にソート

して記録されている．本管理テーブルは，リアルタイム OS が DEPS に関する実行時最適化のために参照する情報として用意される．

本研究では，各タスクの DEPS プロファイルから DEPS 管理テーブルを生成できるソフトウェアツールを開発した．

7.5.2　タスク間メモリ配置最適化

タスク間メモリ配置最適化のステップでは，各タスクが使用する SPM の合計容量が，プロセッサの持つ SPM の容量よりも大きい場合に，SPM をどのように活用するかを決定する．SPM 活用方法の決定には，本稿の第 5 章で提案したプリエンプティブな固定優先度ベースのマルチタスク環境における SPM 活用戦略を適用する．また，本ステップでも，DEPS 管理テーブルと同様に，リアルタイム OS が参照すべき SPM 管理情報を生成する．SPM 管理情報の詳細は，第 6 章の第 6.3 節で解説している．

7.6　実行時最適化

消費エネルギー協調最適化フレームワークにおける実行時最適化フェーズは，リアルタイム OS 上の機能により，次に挙げる処理のみが実行される．

1. タスク切替え時に SPM 領域の内容を再配置する処理
2. チェックポイント毎に各タスクのスラック時間を算出する処理
3. チェックポイント毎にプロセッサの構成を変更する処理

すなわち，本フレームワークにおけるリアルタイム OS は，第 6 章において開発した実行時 SPM 管理機能に加え，DEPS に関する管理機能を有する．

リアルタイム OS の担う 1 つめの処理は，第 6.4 節で解説した実行時 SPM 管理機構が担う．新たなタスクが実行状態に遷移した際に，SPM 管理情報を基にして，そのタスクのプログラムコードおよび静的データを適切な SPM 領域へ再配置する．スタックデータに関しては，切替え対象となるタスクのスタックポインタの値の退避／復帰処理およびデータ SPM の更新処理が実行される．

2 つめの処理では，タスクのスケジューリング時に発生するスラック時間を算出する．3 つめの処理は，算出されたスラック時間を基にした DEPS を適用する．タスク間最適化フェーズで生成された DEPS 管理テーブルの情報を参照しながら，プロセッサのハードウェア構成を，タスクのデッドライン制約を保証した上での消費エネルギー最適なものへと変更する．DEPS によるハードウェア構成の変更は，各タスクのチェックポイント毎に実行される．

以上の処理は，リアルタイム OS 上の機能のひとつとして実装する．SPM 管理処理は，ディスパッチャの機能拡張により実現される．DEPS に関する処理は，チェックポイント API の提供と DEPS 処理ルーチンによって実現される．消費エネルギー協調最適化フレームワークにおける実行時最適化のオーバヘッドは，設計時最適化の出力情報を有効活用するため，小さく抑えることができる．

7.7　適用事例による評価

多階層に跨る消費エネルギー協調最適化フレームワークの有効性を評価した．フレームワークを実装したツールチェーンおよびリアルタイム OS を実用システムに適用すること，評価実験を行った．

適用事例として，テレビ会議システムのアプリケーションを取り上げた．本システムは，Xvid MPEG-4 ビデオコーデック [64] および FFmpeg ライブラリ [53] によって構成されており，マルチパフォーマンスプロセッサのサイクル精度シミュレータ上で動作できるように移植実装した．対象とするアプリケーションは，図 7.9 に示すように，映像のエンコード，デコードおよび入出力制御の処理により構成されている．映像のエンコードおよびデコードの処理は，複数のタスクによって実現される．消費エネルギーの評価は，テレビ会議アプリケーションの実行ログを取得し，文献 [13] による見積もりツールに入力することで行った．

表 7.2 は，本評価実験において使用したマルチパフォーマンスプロセッサのハードウェア構成を示している．本プロセッサは，動作周波数および供給電圧の異なる 2 種類のプロセッシング・エレメントとメモリ容量およびウェイ数が 4 段階で切替え可能な命令キャッシュを持つ．すなわち，合計で DEPS 技術によって制御可能な 8 種類のハードウェア構成の組み合わせを持つ．SPM は，命令とデータそれぞれ個別に持つ．データ SPM は，8 KB ずつを静的データおよびスタックデータそれ

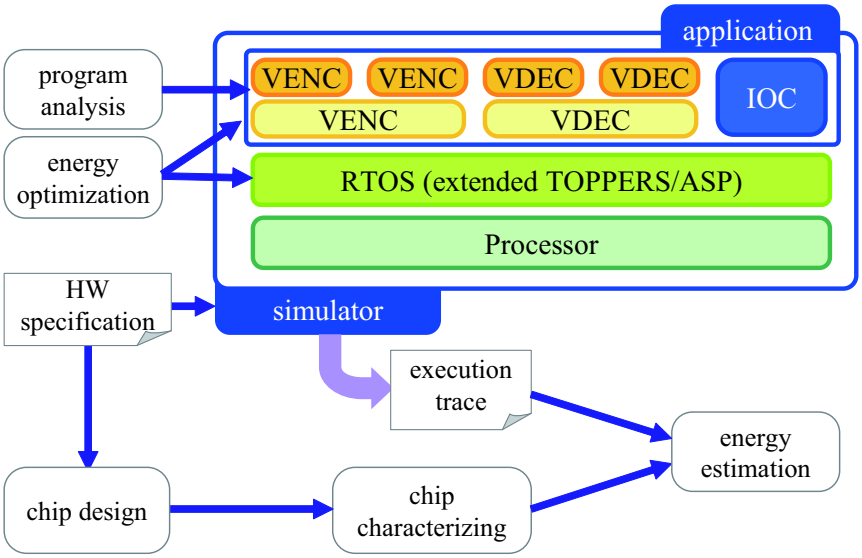


図 7.9: テレビ会議システムの構成と評価の流れ

表 7.2: マルチパフォーマンスプロセッサのハードウェア構成

CPU		PE-H: 60 MHz / 1.8 V PE-L: 30 MHz / 1.0 V
On-chip memory	I-cache	8 KB / 4-way (resizable by changing its associativity)
	I-SPM	8 KB
	D-SPM	16 KB
Off-chip	SDRAM	256 MB

ぞれの配置先候補として扱った．

テレビ会議システムのアプリケーションに対して，本研究にて開発したタスク毎およびタスク間の最適化ツールチェーンを適用した．さらに，静的最適化の結果を用いて，実行時最適化機能を有するリアルタイム OS 上で評価アプリケーションを実行した．

図 7.10 は，マルチパフォーマンスプロセッサのシミュレータ上で実際に動作しているテレビ会議システムのスクリーンショットである．本研究において開発した消費エネルギー協調最適化フレームワークが，実用アプリケーションに適用可能であることを証明するものである．なお，図 7.10 は，2 つのシステムが互いに双方向通信をしながら，ネットワークを介して送受信される映像を処理している様

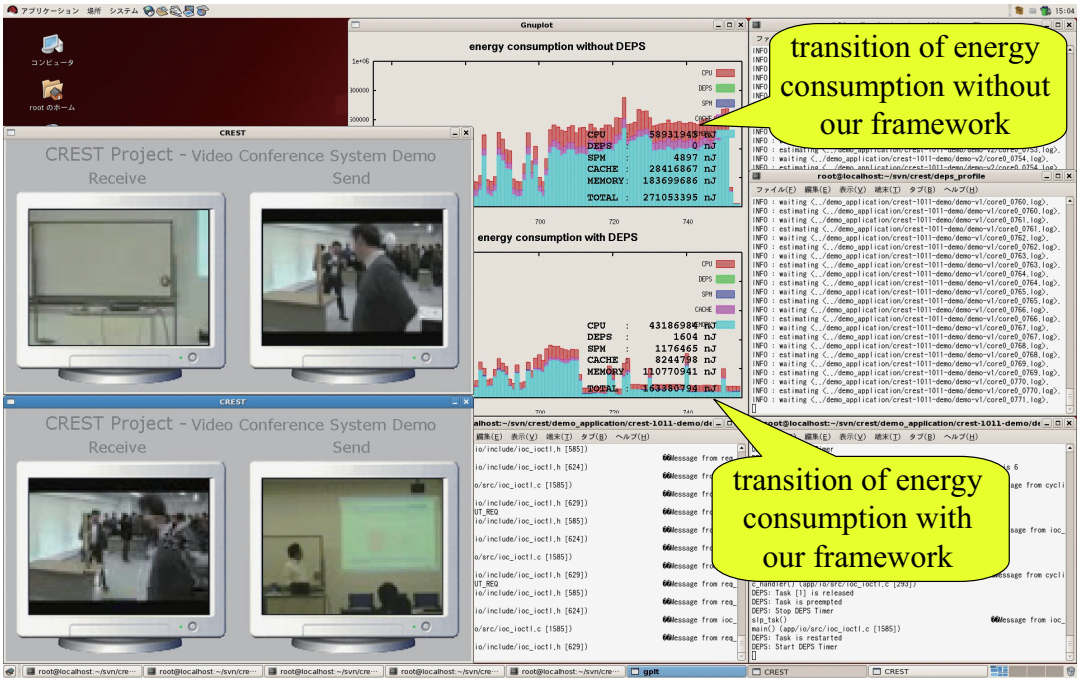


図 7.10: 消費エネルギー協調最適化フレームワークの適用事例：テレビ会議システム

子である．図 7.10 の上側の画面は，消費エネルギー最適化技術を適用していないシステムの様子である．下側が，本研究において開発した消費エネルギー協調最適化フレームワークを適用したシステムの様子をあらわしている．中央に表示されているグラフは，それぞれのシステムにおける消費エネルギー量の推移を実時間に沿ってあらわしたものである．

適用事例における消費エネルギーの評価結果を，図 7.11 に示す．提案手法を適用したものの評価値（“opt”）は，最適化を適用しないもの（“nonopt”）によって正規化して示している．消費エネルギーは，CPU，オンチップの命令メモリ（キャッシュと SPM），オンチップのデータ SPM，オフチップメモリにそれぞれ分類した．横軸の “data$X$” は，入力データの番号をあらわしている．

評価結果より，提案する消費エネルギー協調最適化フレームワークの有効性が立証できた．図 7.11 より，システムに求められるリアルタイム性能を満たした上で，平均して 44 %の消費エネルギーを削減することができた．さらに，消費エネルギーの削減効果は，試行した全ての入力データで安定して確認できた．以上のことは，本フレームワークが消費エネルギー最適化に関する複数の要素技術をう

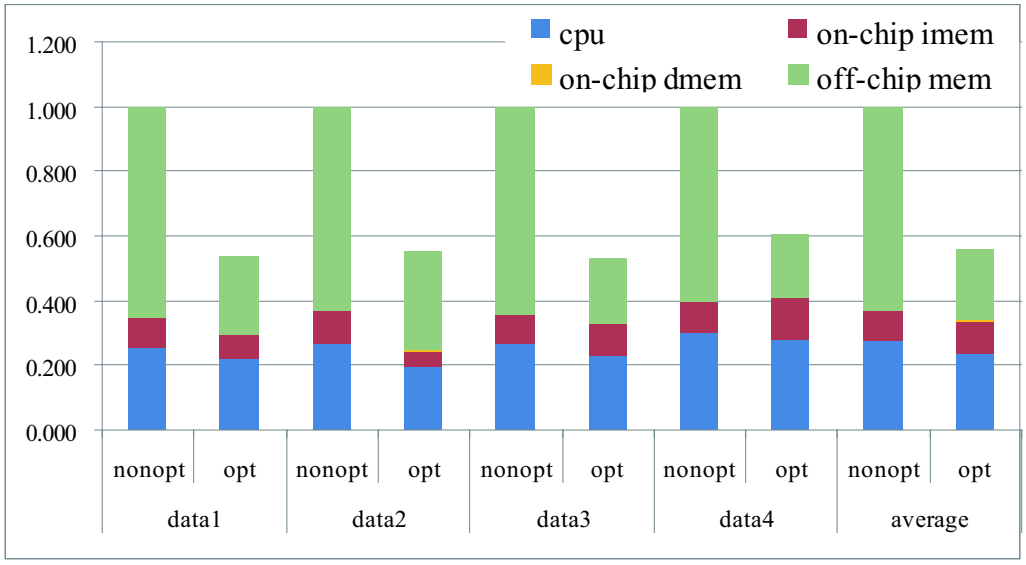


図 7.11: 消費エネルギー協調最適化フレームワークの評価結果

まく統合でき，それらの最適化効果を効率的に活かすことができたことを示しているといえる．さらに，アプリケーション実行時の DEPS 制御および SPM 管理にかかるオーバヘッドはごくわずかであった．これらの値は，実行時間と消費エネルギーともに，全体の 0.1 %以下に抑えられた．このことは，設計時に出来る限りの最適化を施す段階的なフレームワークの形態が，有効に作用したことを示している．

7.8　まとめ

本章では，組込みリアルタイムシステムのための多階層に跨る消費エネルギー協調最適化フレームワークに関する研究成果の報告を行った．提案したフレームワークは，コンパイラ技術，リアルタイム OS 技術，および，プロセッサ技術に関する各要素技術を統合的に適用する．タスクのリアルタイム性能を制約条件として保証しつつ，システム全体の平均消費エネルギー最小化に貢献する．設計時のタスク毎およびタスク間，さらに実行時の 3 段階による最適化の形態をとっており，それぞれの要素技術における最適化効果を効率的に活かすことができる．

消費エネルギー協調最適化フレームワークは，静的最適化を担うソフトウェア・ツールチェーンおよび実行時最適化を担うリアルタイム OS として実装した．最適化の処理は自動的に適用されるため，アプリケーション設計者は対象とするソー

スコードに変更を加える必要はない．さらに，各タスクの実行トレース群を静的最適化の入力情報として扱っているため，本ツールチェーンは一般的な組込みシステムにひろく適用できる．開発成果をテレビ会議システムへと適用した評価実験を通し，提案するフレームワークが消費エネルギー最適化に貢献し，かつ，実用アプリケーションに適用可能であることを示した．

第8章　結論

8.1　まとめ

本論文では，組込みリアルタイムシステムにSPMを用いることの新たな視点からの有効性の評価，マルチタスク環境に対応したSPM管理技術，さらには，種々の要素技術を統合した消費エネルギー協調最適化フレームワークに関する研究の成果について報告を行った．

第3章では，組込みシステムにSPMを用いることの有効性を，リーク電力も評価対象に加えて検証した．リーク電力とは，デバイスの微細化によって消費量が顕著になっている静的な電力のことを指す．本研究では，まず，組込みシステムの命令メモリにおける，リーク電力を含めた消費エネルギーを導出するための手順の詳細を解説した．そして，組込みシステムにおいて一次メモリをキャッシュとSPMを組み合わせて構築することの有効性を議論するための評価実験を行った．

評価実験では，まず，研究着手時における次世代の半導体設計技術にあたる45 nmデバイスのメモリを想定した評価実験を行った．その結果，SPMを用いることによって，キャッシュ単独で一次メモリを構成するよりも，メモリ全体の消費エネルギーを平均で28.8〜31.5 %削減することができた．多くのベンチマークプログラムでは，一次メモリの構成を，SPMのみとした場合とキャッシュとSPMを組み合わせた場合とでは，その消費エネルギーの差はごくわずかであった．しかし，SPMに割当てられる関数の実行命令数の割合が総実行命令数に対して小さい場合には，キャッシュとSPMを組み合わせて一次メモリを構築したほうが，SPMのみの場合よりも，メモリ全体で消費されるエネルギーを小さくできるという結果も得られた．

次に，同様の評価実験を，65 nmデバイスおよび32 nmデバイスのメモリに対しても行った．その結果，半導体製造技術によるデバイスサイズによって，メモリで消費されるエネルギー量は大きく異なった．しかし，デバイスのサイズが異なっても，メモリの構成ごとの消費エネルギー量の傾向に，あまり変化はみられ

なかった．

以上のことから，リーク電力を考慮しても，組込みシステムに SPM を用いることの有効性は高いと結論付けられた．さらには，将来，デバイスの微細化がさらに進んでいっても，組込みシステムに SPM を用いることの優位性は，変わらず高いものであるという予測が得られた．本研究において得られたこれらの知見は，組込みシステムにおけるオンチップメモリの半導体製造技術が CMOS による限り，有効であると考える．

第 4 章では，非プリエンプティブな固定優先度ベースのマルチタスク環境において，命令メモリの消費エネルギー削減を目標とした SPM 領域分割方針を提案した．SPM 領域分割方針は，空間分割法，時間分割法，および，混合分割法の 3 種類である．空間分割法は，それぞれのタスクに使用できる SPM 領域を分割して与える手法である．タスクの起動周期を基にすることにより，実行頻度の高いタスクのプログラムコードが SPM 領域により多く配置される．時間分割法は，実行状態のタスクが SPM の全領域を独占して用いる．混合分割法は，空間および時間分割法の双方を同時に活用する手法である．これにより，SPM 領域のより柔軟な活用が実現できる．さらに，それぞれの方針について，各タスクに割当てられる SPM 容量およびプログラムコード割当てを同時に決定可能な，整数計画問題に定式化した．

実験により，提案手法の有効性を確認した．提案手法を適用することにより，タスクセット実行時の消費エネルギーを最大で 47 %削減することができた．空間分割法は，タスク数が少ない場合に有効であるという傾向がみられた．タスク数が多い場合には，タスク切替え時に SPM の配置内容を管理する時間および混合分割法の有効性が高くなった．さらに，混合分割法は，実験した全ての状況において安定的に消費エネルギーを削減できることが確認できた．本研究における提案手法は，シングルプロセッサシステムの非プリエンプティブなマルチタスク環境において有効な消費エネルギー最適化技術である．

デッドライン制約の厳しいリアルタイムシステムにおいては，プリエンプティブな固定優先度ベースのスケジューリング方式が採用される．第 5 章では，このリアルタイム・タスクスケジューリング方式に従う組込みシステムに適用可能な，3 種類の SPM 活用戦略を提案した．さらに，SPM の配置内容を実行時に更新するためのワークフローを示した．これらの技術は，組込みリアルタイムシステムの命令メモリにおける消費エネルギーの最小化に貢献する．

プリエンプティブな固定優先度ベースのマルチタスク環境でのSPM活用戦略は，第4章で提案した方針を基にした拡張を加える形で開発した．空間活用法は，SPM領域の分割がシステム実行中に変更されないため，拡張の必要はなかった．時間活用法は，実行状態のタスクがSPMの全領域を独占して用いる．タスクの初回の実行開始時および終了時の計2回について主記憶–SPM間コード転送を行い，SPMの保持する内容を管理する．混合活用法は，空間および時間活用法の双方を柔軟に活用する手法である．高優先度タスクは，空間活用法によって与えられるSPM領域に加え，より優先度の低いタスクに割当てられているSPM領域を，時間活用領域として横取りして使用することができる．各方針について，タスクごとに使用できるSPM容量，および，プログラムコード配置の同時決定が実現できる整数計画問題として形式的に定式化した．

さらに，SPM領域の配置内容を実行時に更新することを目的としたワークフローを提案した．本ワークフローを採用したシステムにおけるプロファイラおよびコンパイラは，タスクの静的解析を担い，SPM管理のための情報を生成する．システムの実行時におけるSPM再配置は，リアルタイムOSおよびハードウェアが担う．これらのモジュールが協調動作しながら，静的に生成された情報を基にしてSPM活用戦略を実現する．

実験により，提案するSPM活用戦略の有効性を確認した．実行環境には，本研究で提案したワークフローを実装したうえで，評価実験を行った．提案手法を適用することにより，タスクセット実行時の消費エネルギーを最大で73 %削減することができた．とくに，タスク数が多いセットでは，主記憶–SPM間転送を行う時間活用法および混合活用法の有効性が高くなった．さらに，混合活用法は，評価実験を行ったうちのどの状況においても，全体の消費エネルギーを最小にすることができた．本研究における提案手法は，リアルタイム・スケジューリング方式を採用するシングルプロセッサシステムにおいて有効な消費エネルギー最適化技術である．

マルチタスク環境下では，限られたSPM容量を有効に活用するため，タスク間でSPM領域を時間的に共有できるように，タスクの実行に応じてデータを再配置するのが望ましい．第6章では，主にソフトウェアのみによって実現されるSPMの実行時管理機能を提案した．提案する機能は，リアルタイムOS上で動作するよう設計した．リアルタイムOSの管理により，複数のタスク間で共有して活用されるSPM領域は，実行時に効率良く再配置される．提案手法は，第5章における時

間活用法や混合活用法のような，アプリケーション実行中に SPM の内容を再配置することが求められる管理手法の実現を支援するものである．

提案手法は，SPM 管理情報および実行時 SPM 管理機構からなる．SPM 管理情報は，SPM の配置内容を管理するために用意されるシステム設計時の静的なテーブル情報を含み，タスク毎に持つものとシステム全体で持つものに分類される．また，実行時に値が更新される情報が極力少ない設計となっており，小さなメモリサイズで用意することができる．実行時 SPM 管理機構は，リアルタイム OS の内部機能として設計した．タスク切替え時に管理対象とする SPM 領域を適切に選択することで，実行時のリアルタイム OS の処理にかかるオーバヘッドを小さく抑えることができる．

提案する SPM 管理機能は，リアルタイム OS の内部動作で実現するため，アプリケーションのソースコードへの修正は不要である．さらには，特別なハードウェアを使用することなく動作するよう設計しているため，対象システムへの新たなハードウェア資源の追加も不要となる．さらに，第 6 章では，SPM 管理を制御するための API を定義した．本 API の活用が想定される適用事例や，その有用性について議論した．

提案した SPM 管理機能を TOPPERS/ASP カーネル上に実装し，メモリサイズ，性能のオーバヘッド，アプリケーション実行時の消費エネルギーについて評価した．その結果，提案手法は，小メモリかつ低オーバヘッドで実装できることが確認できた．さらに，これまでに提案してきた SPM 活用手法と組み合わせることで，開発成果は組込みシステムの消費エネルギー削減にも寄与することが確認できた．以上のことから，提案手法の有用性が立証できた．

組込みシステムの消費エネルギーの最小化のためには，設計階層を跨いだ複数の要素技術を総動員することが望ましい．第 7 章では，組込みリアルタイムシステムのための消費エネルギー協調最適化フレームワークを提案した．コンパイラ，リアルタイム OS，および，プロセッサといった多階層における要素技術を統合的に適用することで，システムのリアルタイム性能を保証しつつ，実行時の平均消費エネルギーの最小化を達成する．

消費エネルギー協調最適化フレームワークには，本論文においてこれまで提案してきた SPM を活用したマルチタスク環境におけるメモリ配置最適化技術に加え，著者が共同研究などで開発してきた複数の要素技術が統合的に含まれている．具体的には，ハードウェア構成を変更するプログラム箇所を決定する実行トレース解

析技術，チェックポイントにおいて最適なハードウェア構成を決定する技術，ハードウェア構成を実行時に変更可能なプロセッサ技術，データ SPM の活用によりスタックデータのメモリアクセスにおける消費エネルギーを最小化する技術，および，命令セットシミュレーションを元に消費エネルギーを正確に見積もる技術である．

さらに，提案手法は，タスク毎，タスク間，および，実行時の 3 段階による最適化の形態をとる．設計時のタスク毎およびタスク間のフェーズでは，実行トレース群を基にして，アプリケーションの振る舞いや性質を解析する．設計時の解析結果は，リアルタイム OS による実行時最適化のために活用する．対象とするソフトウェアを段階的に最適化していくことにより，それぞれの要素技術における最適化効果を効率的に活かすことができる．また，段階最適化の形態は，リアルタイム OS による実行時最適化処理にかかるオーバヘッドの低減にも繋がる．

さらに本研究においては，協調最適化フレームワークを，組込みリアルタイムシステムのソフトウェア開発環境として実装した．ソフトウェア開発環境は，システム設計時における解析・最適化ツールチェーン，および，実行時の最適化を担うリアルタイム OS からなる．開発成果物において，対象アプリケーションへの最適化の処理は自動的に適用される．このため，アプリケーション設計者は，本研究における成果物を開発工程において活用することで，組込みリアルタイムシステムの消費エネルギー最適化を自動的に実現できる．さらに，各タスクの実行トレース群を静的最適化の入力情報として扱っているため，本ツールチェーンは一般的な組込みシステムにひろく適用できる．

提案手法の評価として，開発成果をテレビ会議システムへと適用した．この適用事例を通し，提案するフレームワークが消費エネルギー最適化に貢献し，かつ，実用アプリケーションに適用可能であることを示した．

本論文において取り組んできた研究課題は，次に挙げる特色や独創的な点がある．まず，実用製品に則ったシステム環境を強く考慮して研究に取り組んできている．1 つめの研究では，消費量が年々増大しているリーク電力にも着目し，SPM の有効性をより妥当に評価した．この研究において得られた知見は，組込みシステムにおけるオンチップメモリの半導体製造技術が CMOS による限り，有効であると考える．2 つめおよび 3 つめの研究では，組込みシステムにおいてひろく一般に採用される固定優先度ベースのマルチタスク環境において，SPM の有効な活用手法を示した世界初の成果である．さらに，本研究は，単なる理論研究にとどま

らず，提案手法を実用的なものとして実装に取り組んだ．4 つめの研究で提案した SPM 管理機能は，実用製品への採用実績が豊富である TOPPERS/ASP カーネル上に実装されることを想定して設計した．このため，実用的な組込みアプリケーションへの提案手法の適用が期待できる．さらに，5 つめの研究のような，複数の設計階層が統合された革新的な最適化フレームワークを提示した例は，私の知る限り存在しない．開発したソフトウェア・ツールチェーンは，テレビ会議システムという実用システムに適用できることを示した．このことは，開発成果物が産業界で利用できる実用化に近いレベルまで完成度が高められていることを証明している．本論文におけるそれぞれの研究成果は，シングルプロセッサによる組込みリアルタイムシステムにおいて，消費エネルギーの観点から高い有効性があることを示せた．以上の成果が，組込みシステム製品の消費エネルギー最適化を促進することにつながり，社会全体のエネルギー問題に貢献することが期待できる．

8.2　今後の展望

今後の展望としては，まず，カーネル中のプログラムコードやデータに対する SPM 管理技術の確立が挙げられる．本論文で提案した SPM 管理手法は，リアルタイム OS のカーネルオブジェクトは SPM 領域への配置対象として考慮していない．また，本論文の対象としたアプリケーションは，各タスクは独立して動作すると仮定しており，セマフォなどのカーネルオブジェクトは利用しない．より複雑な処理が要求されるシステムにおいては，複数のタスク間で同期や通信を行いながら処理が実現される．このようなアプリケーションでは，カーネルオブジェクトや，サービスコールが頻繁に活用されることが予想される．今後，カーネルオブジェクトにも適用可能な，SPM 活用戦略および管理技術の確立を目指していく予定である．

さらには，第 7 章で構築した消費エネルギー協調最適化フレームワークへの新たな要素技術の統合が挙げられる．段階的な形態を取る本フレームワークは，第 7.2.2 節で述べた通り，消費エネルギー最適化に関する要素技術を後から容易に追加できるという利点がある．今後，SPM 管理技術に限らず新たな要素技術を開発し，本フレームワークに技術統合することで組込みリアルタイムシステムのさらなる消費エネルギー最適化を目指す．

発展的な課題としては，本論文で提案した技術のマルチプロセッサ環境への拡張がある．本論文における研究は，シングルプロセッサ環境のみを対象としてきた．

しかしながら，近年の組込みシステム分野では，チップの発熱量による制約から，マルチプロセッサシステムの採用が一般的となっている．このようなシステムでは，プロセッサ間の通信やデータコヒーレントといった，マルチプロセッサシステムに固有の課題にも対処する必要がある．本論文において開発してきたシングルプロセッサ向けの技術では，これらの課題を考慮していない．今後，これらの課題を克服した上での提案技術の拡張を目指していく．さらには，実用システムに適用可能であるという本研究の貢献を踏襲しつつ，新たなソフトウェア技術の開発や実装を進めていく予定である．

謝辞

本博士論文に関する研究を進めるにあたり，多くのご指導とご助言を戴きました，名古屋大学大学院情報科学研究科の高田広章教授に深く感謝致します．また，博士課程後期課程１年次までは名古屋大学大学院情報科学研究科の准教授として，立命館大学理工学部に転任後も，研究活動において熱心なご指導を戴きました冨山宏之教授に心から感謝致します．さらに，本論文の執筆にあたり，ご指導とご助言を戴きました，名古屋大学大学院情報科学研究科の坂部俊樹教授に深く感謝致します．

本論文に関する研究の一部においては，独立行政法人科学技術振興事業団（JST）戦略的創造研究推進事業（CREST）「情報システムの超低消費電力化を目指した技術革新と統合化技術」の支援を頂きました．研究課題『ソフトウェアとハードウェアの協調による組込みシステムの消費エネルギー最適化』における共同研究を通し，研究成果への多くのご助言により，新たな知見を獲得する機会を与えて頂きました，京都大学大学院情報学研究科の石原亨准教授，名古屋大学大学院工学研究科の曾剛講師，および，南山大学情報理工学部の横山哲郎准教授に深く感謝致します．また，本研究課題に共に取り組んできた高田チームの関係者各位ならびに名古屋大学高田研究室 低消費エネルギーグループの諸氏に深く感謝致します．なお，本研究を進めるにあたっては，独立行政法人日本学術振興会 特別研究員制度の支援を頂きました．関係者各位に深く感謝致します．

本研究の評価実験環境の整備のために技術的なご協力を頂きました，名古屋大学大学院情報科学研究科附属組込みシステム研究センターの本田晋也准教授，ならびに，本研究室 OB である図子純平氏，古川貴士氏，安積卓也氏，相庭裕史氏に深く感謝いたします．さらに，本研究を進めるにあたって研究生活に活力と潤いを与えて頂きました，名古屋大学高田研究室の各位に，感謝いたします．

研究活動の傍ら，博士課程前期課程学生生活の間に，組込みシステム技術に関するサマースクール（SSEST）実行委員会への参画という貴重な経験をさせていただきました．約２年携わった実行委員会の運営活動を通して，数多くの経験や能力，さらには，多くの仲間を得ることが出来ました．本活動への参画に理解を示

して頂いた先生方ならびに研究室関係者各位に重ねて感謝を申し上げるとともに，実行委員会メンバ，ならびに参加者・関係者各位に，深く感謝致します．

最後に，長い間，研究活動に理解と支援をしてくれた家族と，充実した時間を共に過ごした仲間達に心から感謝致します．

参考文献

[1] N. AbouGhazaleh, D. Mossé, B. R. Childers, and R. Melhem, "Collaborative Operating System and Compiler Power Management for Real-Time Applications," *ACM Transaction on Embedded Computing Systems (TECS)*, vol. 5, no. 1, pp. 82–115, Feb 2006.

[2] F. Angiolini, L. Benini, and A. Caprara, "An Efficient Profile-Based Algorithm for Scratchpad Memory Partitioning," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 11, pp. 1660–1676, Nov 2005.

[3] O. Avissar, R. Barua, and D. Stewart, "An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems," *ACM Transaction on Embedded Computing Systems (TECS)*, vol. 1, no. 1, pp. 6–26, 2002.

[4] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau, "Profile-Based Dynamic Voltage Scheduling Using Program Checkpoints," in *Proceedings of the conference on Design, automation and test in Europe*, Paris, France, pp. 168–175, Aug 2002.

[5] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Comparison of Cache- and Scratch-Pad based Memory Systems with respect to Performance, Area and Energy Consumption," Technical Report 762, TU Dortmund, Faculty of Computer Science 12, 2001.

[6] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems," in *Proceedings of the International Symposium on Hardware/Software Codesign (CODES)*, Estes Park, Colorado, pp. 73–78, May 2002.

[7] P. Bjuréus and A. Jantsch, "Performance Analysis with Confidence Intervals for Embedded Software Processes," in *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS '01)*, Montréal, P.Q., Canada, pp. 45–50, May 2001.

[8] S. E. Buttrey, "Calling the lp_solve Linear Program Software from R, S-PLUS and Excel," *Journal of Statistical Software*, vol. 14, no. i04, 2005.

[9] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min, "A Dynamic Code Placement Technique for Scratchpad Memory using Postpass Optimization," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*, Seoul, Korea, pp. 223–233, Oct 2006.

[10] B. Egger, J. Lee, and H. Shin, "Dynamic Scratchpad Memory Management for Code in Portable Systems with an MMU," *ACM Transactions on Embedded Computer Systems (TECS)*, vol. 7, no. 2, pp. 11:1–11:38, Jan 2008.

[11] B. Egger, J. Lee, and H. Shin, "Scratchpad Memory Management in a Multitasking Environment," in *Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT '08)*, Atlanta, USA, pp. 265–274, Dec 2008.

[12] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias, "An Integrated Hardware/Software Approach For Run-Time Scratchpad Management," in *Proceedings of Design Automation Conference (DAC)*, Los Alamitos, CA, USA, pp. 238–243, Jun 2004.

[13] L. Gauthier and T. Ishihara, "Compiler Assisted Energy Reduction Techniques for Embedded Multimedia Processors," in *Proceedings of the 2nd APSIPA Annual Summit and Conference*, Biopolis, Singapore, pp. 27–36, Dec 2010.

[14] L. Gauthier, T. Ishihara, H. Takase, H. Tomiyama, and H. Takada, "Placing Static and Stack Data into a Scratch-Pad Memory for Reducing the Energy Consumption of Multi-task Applications," in *Proceedings of The 16th Work-*

shop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2010), Taipei, Taiwan, pp. 7–12, Oct 2010.

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free Commercially Representative Embedded Benchmark Suite," in *Proceedings of IEEE International Workshop on the Workload Characterization (WWC)*, Los Alamitos, USA, pp. 3–14, Dec. 2001.

[16] T. Ishihara, "A Multi-Performance Processor for Reducing the Energy Consumption of Real-Time Embedded Systems," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E93-A, pp. 2533–2541, Oct 2010.

[17] Y. Ishitobi, T. Ishihara, and H. Yasuura, "Code and Data Placement for Embedded Processors with Scratchpad and Cache Memories," *Journal of Signal Processing Systems*, vol. 60, no. 2, pp. 211–224, Dec 2008.

[18] A. Janapsatya, A. Ignjatovic, and S. Parameswaran, "Exploiting Statistical Information for Implementation of Instruction Scratchpad Memory in Embedded System," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, pp. 816–829, Aug 2006.

[19] A. Janapsatya, S. Parameswaran, and A. Ignjatovic, "Hardware/Software Managed Scratchpad Memory for Embedded System," in *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '04)*, New Jersey, USA, pp. 370–377, Nov 2004.

[20] M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu, "Compiler-Directed Scratch Pad Memory Optimization for Embedded Multiprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, pp. 281–287, Mar 2004.

[21] S. Kang, H. Wang, Y. Chen, X. Wang, and Y. Dai, "Skyeye: An Instruction Simulator with Energy Awareness," in *Proceedings of the International Conference on Embedded Software and Systems (ICESS)*, Hangzhou, China, pp. 456–461, Dec 2004.

[22] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[23] W. Kim, D. Shin, H.-S. Yun, J. Kim and S.-L. Min, "Performance Evaluation of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems," *Journal on Low Power Electronics*, vol. 1, no. 3, pp. 207–216, Dec 2005.

[24] A. Mizuno, H. Uetani, and H. Eichel, "Design Methodology and System for a Configurable Media Embedded Processor Extensible to VLIW Architecture," in *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, Freiburg, Germany, pp. 2–7, Sep 2002.

[25] P. R. Panda, A. Nicolau, and N. Dutt, *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration.* Norwell, MA, USA: Kluwer Academic Publishers, 1998.

[26] R. Pyka, C. Faßbach, M. Verma, H. Falk, and P. Marwedel, "Operating System Integrated Energy Aware Scratchpad Allocation Strategies for Multiprocess Applications," in *Proceedings of 10th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, Nice, France, pp. 41–50, Apr 2007.

[27] S. Segars, "Low Power Design Techniques for Microprocessors,", IEEE International Solid-State Circuits Conference (Tutorial), Feb 2001.

[28] J. Seo, T. Kim, and N. D. Dutt, "Optimal Integration of Inter-task and Intra-task Dynamic Voltage Scaling Techniques for Hard Real-Time Applications," in *Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '05)*, San Jose, CA, pp. 450–455, Nov 2005.

[29] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning Program and Data Objects to Scratchpad for Energy Reduction," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '02)*, Paris, France, pp. 409–415, Mar 2002.

[30] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel, "Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory," in *Proceedings of the 15th International Symposium on System Synthesis (ISSS)*, Kyoto, Japan, pp. 213–218, Oct. 2002.

[31] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*, Seoul, Korea, pp. 401–410, Oct 2006.

[32] V. Suhendra, A. Roychoudhury, and T. Mitra, "Scratchpad Allocation for Concurrent Embedded Software," in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '08)*, Atlanta, GA, USA, pp. 37–42, Oct 2008.

[33] H. Takada and K. Sakamura, "μITRON for Small-Scale Embedded Systems," *IEEE Micro*, vol. 15, no. 6, pp. 46–54, Dec 1995.

[34] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "CACTI 4.0," Technical Report, HP Laboratories, 2006.

[35] T. Tatematsu, H. Takase, G. Zeng, H. Tomiyama, and H. Takada, "Checkpoint Extraction Using Execution Traces for Intra-Task DVFS in Embedded Systems," in *Proceedings of The 6th International Symposium on Electronic Design, Test and Application (DELTA2011)*, Queenstown, New Zealand, pp. 19–24, Jan 2011.

[36] S. Thoziyoor, N. Muralimanohar, and N. P. Jouppi, "CACTI 5.0," Technical Report, HP Laboratories, 2006.

[37] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel, "Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A First Approach," in *Proceedings of IEEE 3rd Workshop on Embedded System for Real-Time Multimedia (ESTIMedia)*, Jersey City, USA, pp. 115–200, Sep 2005.

[38] M. Verma, L. Wehmeyer, and P. Marwedel, “Cache Aware Scratchpad Allocation,” in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Paris, France, Feb 2004.

[39] M. Verma, L. Wehmeyer, and P. Marwedel, “Cache Aware Scratchpad Allocation for Energy-Constrained Embedded Systems,” *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, pp. 2035–2051, Oct 2006.

[40] S. J. E. Wilton and N. P. Jouppi, “CACTI: An Enhanced Cache Access and Cycle Time Model,” *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[41] W. Yuan, K. Nahrstedt, S. V. Adve, D. L. Jones, and R. H. Kravets, “GRACE-1: Cross-Layer Adaptation for Multimedia Quality and Battery Energy,” *IEEE Transactions on Mobile Computing*, vol. 5, pp. 799–815, Jul 2006.

[42] G. Zeng, H. Tomiyama, and H. Takada, “A Generalized Framework for Energy Savings in Hard Real-Time Embedded Systems,” *IPSJ Transactions on System LSI Design Methodology*, vol. 2, pp. 180–188, Aug 2009.

[43] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, “HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects,” Technical Report, University of Virginia, Department of Computer Science Technology, 2003.

[44] 古川貴士, 柴田誠也, 本田晋也, 冨山宏之, 高田広章, “マルチプロセッサ RTOS 対応コシミュレータ,” **情報処理学会研究報告組込みシステム**, vol. 2008, no. 7, 神奈川, pp. 17–22, 2008 年 1 月.

[45] 川島裕崇, 曾剛, 渥美紀寿, 立松知紘, 高瀬英希, 高田広章, “DEPS プロファイルの評価法とそれを利用したチェックポイント選定,” **情報処理学会研究報告**, vol. 2011-EMB-20, no. 43, 宮古島, 2011 年 3 月.

[46] 川島裕崇, 曾剛, 渥美紀寿, 立松知紘, 高田広章, “DEPS フレームワークにおける最悪実行時間と平均消費エネルギーのタスク内解析手法,” **電子情報通信学会技術研究報告**, vol. 110, no. 432, 沖縄, pp. 19–24, 2011 年 3 月.

[47] 相庭裕史, 柴田誠也, 古川貴士, 本田晋也, 冨山宏之, 高田広章, “マルチプロセッサ RTOS 対応シミュレーション環境の機能拡張と効率化,” **電子情報通信学会技術研究報告**, vol. 107, no. 558, 屋久島, pp. 13–18, 2008 年 3 月.

[48] ARM7TDMI - ARM Processor, `http://www.arm.com/products/CPUs/ARM7TDMI.html` (accessed 2009-01-27).

[49] ARM920T - ARM Processor, `http://www.arm.com/products/CPUs/ARM920T.html` (accessed 2009-01-27).

[50] ARM966E-S Technical Reference Manual, `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0213e/ARM966E-S_TRM.pdf` (accessed 2010-10-21).

[51] CACTI 5.3, `http://www.hpl.hp.com/research/cacti/cacti.5.3.rev.174.tar.gz` (accessed 2009-01-27).

[52] EEMBC – The Embedded Microprocessor Benchmark Consortium, `http://www.eembc.org/` (accessed 2009-01-27).

[53] FFmpeg multimedia system, `http://www.ffmpeg.org/` (accessed 2011-04-10).

[54] GCC, the GNU Compiler Collection, `http://gcc.gnu.org/` (accessed 2009-01-27).

[55] GLPK (GNU Linear Programming Kit), `http://www.gnu.org/software/glpk/` (accessed 2009-01-27).

[56] MeP (Media embedded Processor), `http://www.semicon.toshiba.co.jp/product/micro/selection/mep/index.html` (accessed 2010-10-21).

[57] The Micron System Power Calculator, `http://www.micron.com/support/designsupport/tools/powercalc/powercalc` (accessed 2009-01-27).

[58] Micron Technology, Inc., `http://www.micron.com/` (accessed 2009-01-27).

[59] μITRON4.0 仕様, `http://www.ertl.jp/ITRON/SPEC/mitron4-j.html` (accessed 2009-01-27).

[60] SH7201 グループ, http://japan.renesas.com/products/mpumcu/superh/sh7200/sh7201/sh7201_root.jsp (accessed 2010-10-21).

[61] SimpleScalar LLC, http://www.simplescalar.com/ (accessed 2009-01-27).

[62] SkyEye - Open Source Simulator, http://www.skyeye.org/ (accessed 2009-01-27).

[63] TOPPERS プロジェクト, http://toppers.jp/ (accessed 2009-01-27).

[64] Xvid, MPEG-4 Compliant Video Codec, http://www.xvid.org/ (accessed 2011-04-10).

研究業績

学術雑誌論文

- Hideki Takase, Gang Zeng, Hiroyuki Tomiyama, Hiroaki Takada, "Energy efficiency of scratch-pad memory in deep submicron domains: an empirical study," *IEICE Electronics Express*, Vol. 5, No. 23, pp. 1010–1016, Dec 2008.
- Hideki Takase, Hiroyuki Tomiyama, Hiroaki Takada, "Partitioning and Allocation of Scratch-Pad Memory in Priority-Based Multi-Task Systems," *IPSJ Transactions on System LSI Design Methodology*, Vol. 2 (2009), pp. 180–188, Aug 2009.
- Hideki Takase, Hiroyuki Tomiyama, Hiroaki Takada, "Partitioning and Allocation of Scratch-Pad Memory for Energy Minimization of Priority-Based Preemptive Multi-Task Systems," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E94-A, No. 10, pp. 1954–1964, Oct 2011.
- 谷口一徹, 安藤友樹, 高瀬英希, 安積卓也, 松原豊, 村上靖明, 菅谷みどり, "学生および若手技術者による組込みシステム技術に関するサマースクールの実践," **情報処理学会論文誌**, Vol. 52, No. 12, pp. 3221–3237, 2011 年 12 月.
- 立松知紘, 高瀬英希, 曾剛, 川島裕崇, 冨山宏之, 高田広章, "実行トレースマイニングを用いた組込みシステムにおけるタスク内 DVFS のためのチェックポイント抽出," **情報処理学会論文誌**, Vol. 52, No. 12, pp. 3729–3744, 2011 年 12 月.
- 一場利幸, 森孝夫, 高瀬英希, 鴫原一人, 本田晋也, 高田広章, "命令セットシミュレータの実行制御機構を用いたマルチプロセッサ RTOS のテスト効率化手法," **電子情報通信学会論文誌 D**, Vol. J95-D, No. 3, 2012 年 3 月. (採録決定)

国際会議論文（査読あり）

- Hideki Takase, Hiroyuki Tomiyama, Gang Zeng and Hiroaki Takada, "Energy Efficiency of Scratch-Pad Memory at 65 nm and Below: An Empirical

Study," in *Proceedings of International Conference on Embedded Software and Systems (ICESS)*, pp. 93–97, Chengdu, China, Jul 2008.

- Hideki Takase, Hiroyuki Tomiyama and Hiroaki Takada, "Allocation of Scratch-Pad Memory in Priority-Based Multi-Task Systems," in *Proceedings of 2009 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pp. 68–71, Hsinchu, Taiwan, Apr 2009.

- Hideki Takase, Takuya Azumi, Ittetsu Taniguchi, Yutaka Matsubara, Hayato Kanai, Shintaro Hosoai and Midori Sugaya, "History of Summer School on Embedded System Technologies Organized by Students and Young Engineers," in *Proceedings of 2009 Workshop on Embedded Systems Education (WESE'09)*, pp. 19–26, Grenoble, France, Oct 2009.

- Hideki Takase, Hiroyuki Tomiyama and Hiroaki Takada, "Partitioning and Allocation of Scratch-Pad Memory for Priority-Based Preemptive Multi-Task Systems," in *Proceedings of 2010 Design, Automation and Test in Europe (DATE2010)*, pp. 1124–1129, Dresden, Germany, Mar 2010.

- Lovic Gauthier, Tohru Ishihara, Hideki Takase, Hiroyuki Tomiyama and Hiroaki Takada, "Placing Static and Stack Data into a Scratch-Pad Memory for Reducing the Energy Consumption of Multi-task Applications," in *Proceedings of The 16th Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2010)*, pp. 7–12, Taipei, Taiwan, Oct 2010.

- Lovic Gauthier, Tohru Ishihara, Hideki Takase, Hiroyuki Tomiyama and Hiroaki Takada, "Minimizing Inter-Task Interferences in Scratch-Pad Memory Usage for Reducing the Energy Consumption of Multi-Task Systems," in *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2010)*, pp. 157–166, Scottsdale, AZ, USA, Oct 2010.

- Tomohiro Tatematsu, Hideki Takase, Gang Zeng, Hiroyuki Tomiyama and Hiroaki Takada, "Checkpoint Extraction Using Execution Traces for Intra-Task DVFS in Embedded Systems," in *Proceedings of The 6th Interna-*

tional Symposium on Electronic Design, Test and Application (DELTA2011), pp. 19–24, Queenstown, New Zealand, Jan 2011.

- Hideki Takase, Gang Zeng, Lovic Gauthier, Hirotaka Kawashima, Noritoshi Atsumi, Tomohiro Tatematsu, Yoshitake Kobayashi, Shunitsu Kohara, Takenori Koshiro, Tohru Ishihara, Hiroyuki Tomiyama and Hiroaki Takada, "An Integrated Optimization Framework for Reducing the Energy Consumption of Embedded Real-Time Applications," in *Proceedings of International Symposium on Low Power Electronics and Design 2011 (ISLPED 2011)*, pp. 271–276, Fukuoka, Japan, Aug 2011.

- Hirotaka Kawashima, Gang Zeng, Hideki Takase and Hiroaki Takada, "Checkpoint Selection for DEPS Framework Based on Quantitative Evaluation of DEPS Profile," in *Proceedings of The 17th Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, Beppu, Japan, Mar 2012. （発表予定）

国内会議発表論文（査読あり）

- 高瀬英希, 曾剛, 冨山宏之, 高田広章, "リーク電力を考慮したスクラッチパッドメモリの有効性の評価," **組込みシステムシンポジウム 2007 論文集**, Vol. 2007, No. 8, pp. 82–89, 東京, 2007 年 10 月.

国内研究会発表論文（査読なし）

- 高瀬英希, 冨山宏之, 高田広章, "マルチタスク環境におけるスクラッチパッドメモリ領域活用法," **電子情報通信学会技術研究報告**, Vol. 107, No. 558, pp. 109–114, 屋久島, 2008 年 3 月.

- 高瀬英希, 冨山宏之, 高田広章, "プリエンプティブなマルチタスク環境におけるスクラッチパッドメモリ領域分割法," **情報処理学会研究報告**, Vol. 2008, No. 55(SE-160 EMB-9), pp. 57–64, 東京, 2008 年 6 月.

- 高瀬英希, 矢頭岳人, 及川達裕, 岩田英一郎, 池田健太郎, 畑尚志, 川口麻美, 飯野敦資, 高井日淑, 中島広茂, 松葉俊信, 吉村悠, "第 4 回組込みシステム技

術に関するサマースクール（SSEST4）実施報告," **電子情報通信学会技術研究報告**, Vol. 108, No. 463, pp. 7–12, 佐渡島, 2009 年 3 月.

- Hideki Takase, Hiroyuki Tomiyama and Hiroaki Takada, "A Scratch-Pad Memory Management Framework for Embedded Real-Time Systems," **情報処理学会研究報告**, Vol. 2009-SLDM-140, No. 8, 小倉, 2009 年 5 月.

- 立松知紘, 高瀬英希, 曾剛, 冨山宏之, 高田広章, "実行トレースマイニングを用いたタスク内 DVFS に有効なチェックポイント抽出手法," **情報処理学会研究報告**, Vol. 2010-EMB-18, No. 10, 函館, 2010 年 8 月.

- 三輪遼平, 高瀬英希, 曾剛, 冨山宏之, 高田広章, "組込みシステムにおける低消費エネルギー志向の効率的なスラック時間の導出," **情報処理学会研究報告**, Vol. 2010-EMB-18, No. 11, 函館, 2010 年 8 月.

- 一場利幸, 高瀬英希, 鴫原一人, 本田晋也, 高田広章, "マルチプロセッサ環境におけるタイミング依存のシナリオを実行可能なシミュレーション機構," **情報処理学会研究報告**, Vol. 2011-EMB-20, No. 4, 宮古島, 2011 年 3 月.

- 川島裕崇, 曾剛, 渥美紀寿, 立松知紘, 高瀬英希, 高田広章, "DEPS プロファイルの評価法とそれを利用したチェックポイント選定," **情報処理学会研究報告**, Vol. 2011-EMB-20, No. 43, 宮古島, 2011 年 3 月.

- Hideki Takase, Gang Zeng, Hirotaka Kawashima, Noritoshi Atsumi, Tomohiro Tatematsu, Lovic Gauthier, Tohru Ishihara, Yoshitake Kobayashi, Shunitsu Kohara, Takenori Koshiro, Hiroyuki Tomiyama and Hiroaki Takada, "An Energy Optimization Framework for Embedded Applications," **情報処理学会研究報告**, Vol. 2011-EMB-20, No. 3, 宮古島, 2011 年 3 月.

- 高瀬英希, 高田広章, "スクラッチパッドメモリの実行時管理機能を有するリアルタイム OS の実装および評価," **電子情報通信学会技術研究報告**, Vol. 111, No. 324, pp. 97–102, 宮崎, 2011 年 11 月.

国内研究会ポスター発表（査読なし）

- 安積卓也, 及川達裕, 小田哲也, 金川太俊, 白石崇, 瀬戸敏喜, 高瀬英希, 細合晋太郎, 村上靖明, “第 3 回組込みシステム技術に関するサマースクール（SSEST3）,” **第 9 回組込みシステム技術に関するサマーワークショップ（SWEST9）予稿集**, pp. 43–46, 2007 年 8 月.

- 高瀬英希, 及川達裕, 矢頭岳人, 飯野敦資, 池田健太郎, 岩田英一郎, 川口麻美, 高井日淑, 畑尚志, 中島広茂, 松葉俊信, 吉村悠, “第 4 回組込みシステム技術に関するサマースクール,” **第 10 回組込みシステム技術に関するサマーワークショップ（SWEST10）予稿集**, pp. 100–113, 2008 年 9 月.

- 高瀬英希, 小原俊逸, 深谷哲司, Lovic Gauthier, 石原亨, 冨山宏之, 高田広章, “ソフトウェアとハードウェアの協調による組込みシステムの消費エネルギー最適化,” **第 12 回 組込みシステム技術に関するサマーワークショップ（SWEST12）予稿集**, pp. 60–63, 豊橋市, 2010 年 9 月.

- 高瀬英希, 高田広章, “スクラッチパッドメモリの実行時管理機能を有するリアルタイム OS,” **第 13 回 組込みシステム技術に関するサマーワークショップ（SWEST13）予稿集**, pp. 21–24, 下呂, 2011 年 9 月.

その他

- ［技術交流会 話題提供］高瀬英希, 松原豊, “SSEST: 若手がやってみた組込み教育,” 組込みシステム開発技術研究会（CEST）第 114 回技術交流会, 名古屋, 2009 年 5 月.

- ［雑誌記事］佐藤洋介, 山郷成仁, 高瀬英希, 鈴木里沙, “ライン・トレース・カーで学ぶ組み込みシステム開発の基礎知識（第 1 回）モデリング言語を使って仕様書を書く,” CQ 出版社 Interface2009 年 08 月号, pp. 130–142, 2009 年 7 月.

- ［セッションコーディネート］高瀬英希, “若手研究者によるお悩み相談室 〜この先端研究ってホントに産で使えるの？この最新技術ってホントに学で扱えるの？〜,” 第 12 回組込みシステム技術に関するサマーワークショップ（SWEST12）, 2010 年 9 月.

- ［講演］ 高瀬英希, “ソフトウェアとハードウェアの協調による組込みシステムの消費エネルギー最適化,” 第 2 回名古屋大学組込みシステム研究センターシンポジウム, 2010 年 9 月.

受賞等

- 情報処理学会 2007 年度コンピュータサイエンス領域 奨励賞, 2008 年 7 月.
- 第 134 回 SLDM 研究会 優秀発表学生賞, 2008 年 8 月.
- 平成 20 年度 情報処理学会 SLDM 研究会 優秀論文賞, 2008 年 8 月.
- 第 140 回 SLDM 研究会 優秀発表学生賞, 2009 年 8 月.
- 2009 年度 IPSJ 論文船井若手奨励賞, 2010 年 3 月.
- Outstanding Paper Award, Lovic Gauthier, Tohru Ishihara, Hideki Takase, Hiroyuki Tomiyama, Hiroaki Takada, Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI), 2010 年 10 月.
- 平成 23 年 名古屋大学学術奨励賞, 2011 年 7 月.