情報処理学会論文誌　プログラミング　Vol. 2　No. 2　54–69 (Mar. 2009)

動的電圧制御システムにおける評価戦略選択に基づく高効率消費エネルギー関数型プログラミング

横山哲郎[†1]　今井敬吾[†1]　曾剛[†1]
冨山宏之[†1]　高田広章[†1]　結縁祥治[†1]

動的電圧制御（DVS: dynamic voltage scaling）は，プロセッサへの供給電圧とその動作周波数をプログラム実行時に変化させる技術であり，現在，多くの商用プロセッサに実装されている．本稿では，デッドラインなどの制約のあるリアルタイムシステムを対象に，DVS システムにおいて動的エネルギー消費のより少ないプログラムを開発する方法論の 1 つを提示する．DVS システムで実行されるプログラムに動的エネルギー消費の最適化が有効であるためには，残り予測実行時間の実行早期の正確な見積りを容易にすることが重要である．そのため，プログラムは何を計算するかに加えどう計算するかをうまく指定する必要がある．しかし，プログラム改変によるエネルギー消費の最適化はプログラムのモジュール性を著しく損なう．本稿では，プログラムから独立し，エネルギー消費の最適化戦略を開発する手法を提案する．提案手法により，エネルギー消費の最適化を行うときに元のプログラムの部分正当性が容易に保存され，元のプログラムおよびエネルギー消費の最適化を行うための評価戦略が独立してそれぞれモジュール性を有するようになった．遅延評価などのプログラミング言語の特徴や構成的アルゴリズム論における組化などを活用することにより半自動化が実現できたことが，本開発法の特徴の 1 つである．本稿では，整列，選択，文字列検索などの基本的なアルゴリズムに提案手法を適用する．また，電力モデルを備えた命令セットシミュレータ（ISS: instruction set simulator）において実験を行い，エネルギー消費がどれだけ最適化されたかを評価する．基本的なアルゴリズムにおいて，本稿のアプローチが有効であることから，複雑なアルゴリズムに対しても本手法が効果的であることが期待される．

Energy Efficient Functional Programing on DVS Systems by Varying Evaluation Strategies

Tetsuo Yokoyama,[†1] Keigo Imai,[†1] Gang Zeng,[†1]
Hiroyuki Tomiyama,[†1] Hiroaki Takada[†1]
and Shoji Yuen[†1]

DVS (dynamic voltage scaling) is a technique for scaling the processor's supply voltages and working frequencies. Several commercially available processors provide voltage/frequency controls. We propose a development method for deriving dynamically energy efficient programs on DVS-enabled real-time systems, which have several constraints such as deadline. In order to improve energy efficiency of programs on DVS systems, it is necessary to accurately estimate remaining predicted execution time in the early phase of the execution of programs. Thus, how to execute codes is as important as what codes to execute. However, the revision of programs for energy efficiency seriously harms their modularity. We separate concerns of the development of energy optimization strategy from the development of programs. As the result, partial correctness of original programs is preserved, and each of original programs and energy optimization strategy has modularity, independently. Lazy evaluation of functional programming languages and tupling in constructive algorithmics are employed for realizing the semi-automation of our development method. Our techniques are applied to basic algorithms such as sorting, selection, and string matching. Their energy efficiency is evaluated by using an instruction set simulator (ISS) with a power model.

1. はじめに

携帯電話，携帯情報端末など電源駆動の組込みシステムにおいて，VLSI の計算能力の向上および小型化の進展にともない，熱密度およびエネルギー消費の問題が顕在化してきた[13]．電力消費の最適化は VLSI の製造コストを低減させて信頼性を向上させ，エネルギー消費の最適化は運用コストを低減させさらには天然資源使用量の削減につながる．環境と相互作用のあるシステムではハードリアルタイム制約の下で資源制御を行うことが必要であ

†1 名古屋大学大学院情報科学研究科
Graduate School of Information Science, Nagoya University

© 2009 Information Processing Society of Japan

る．このとき，プロセッサは典型的な制御対象の 1 つであり，本稿ではプロセッサのエネルギー消費の最適化に注目する．

タスクに必要とされる性能が VLSI システムの最大性能以下である場合，デッドライン制約を満たす範囲内で，クロック周波数および対応する供給電力を削減することができる．これは動的電圧制御（DVS: dynamic voltage scaling）の背後にある考え方である．DVS システムにおいて，エネルギー消費と性能はトレードオフである．プログラムの高速実行には多量の，しかし低速実行には少量のみのエネルギーが必要である．このトレードオフをプログラマがどのように簡易に記述するかは重要な問題である．本稿では，プログラム本体と DVS によるエネルギー消費の最適化のための戦略を分離して開発を行う方法論を提示する．両者は独立して，開発および再利用を行うことができる．

以降の構成を述べる．2 章では，関連研究をまとめ，本稿のようなアプローチをとった背景について述べる．3 章では，以降の章で用いる，プログラミングおよび DVS システムの基礎を説明する．4 章では，エネルギー消費の最適化が有効なプログラムの開発手法を提案し，例を通しその機構を詳説する．5 章では，提案開発法手法により開発されたプログラムの効率性を実験により評価する．6 章では，まとめと今後の課題について述べる．

2. 関 連 研 究

プログラムのモジュール性および可読性の向上に対してエネルギー消費の最適化はトレードオフ関係にある．既存のタスク内 DVS（IntraDVS: intra-task DVS）[5),16),24),33)] のアプローチでは，実装言語は命令型であり，またプログラムの実行順序は静的であると想定されてきた．たとえば，文献 33) では電圧制御を行うポイント（VSP: voltage scaling point）をソースコードに埋め込んでいる．VSP の埋め込み時には元のソースコードに改変が加えられず，命令型言語を用いているために計算順序はソースコードの記述に依存してしまっている．したがって，プログラム開発時に DVS を効果的に行える実行順序になるようプログラマが明示的に記述しなければならなかった．また，実行順序をどう制御すればエネルギー消費が最適化できるかという戦略の記述がソースコード内に散見し，その改良や再利用なども難しかった．

特定のアーキテクチャに依存せずにソフトウェアとエネルギー消費の関係を研究する取り組みがある．埋込み Turing 機械（ATM: augmented Turing machine）[22)] は，各状態遷移に消失エネルギー量を埋め込んだモデルである．このモデルは，エネルギー消費の漸近的振舞いなどのエネルギー計算量研究の礎になるかもしれない．しかし，現在の計算機は Turing 機械をモデルとして作成されていないため，実際のチップの振舞いの傾向を調べるのに適していない．

命令セットシミュレータ（ISS: instruction set simulator）[4),38)] は，同一命令の繰返し実行時における実チップの電流測定を行いて作成されたモデルを用いることで，精度の高いエネルギー消費予測を実現している．たとえば，エネルギー消費に大きく影響する命令・データキャッシュの効果やキャッシュ・メモリ間のデータ転送の遅延時間（latency）の差などが考慮されている．また，ISS と電力モデルを組み合わせる試みがある[8),35),37)]．しかし，アルゴリズムが与えられたとき，これらのシミュレータは各チップに特有の特徴から影響を受けすぎてしまい，一般的な傾向や特性を把握しにくいという弱点がある．本稿では，特定のアーキテクチャを離れアルゴリズムのエネルギー消費の一般的な傾向について議論する．

ソースコードを変換することでエネルギー消費を最適化する試みがある．文献 11) では，部分計算により性能の最適化とエネルギー消費の最適化を行った．彼らの結果によると，エネルギー消費と性能の最適化がほぼ同じ傾向を示している．こういったアプローチは，たとえば，文献 29) が詳しい．しかし，性能の最適化とエネルギー消費の最適化は必ずしも一致しない．本稿では，この 2 者の原理が異なる例に焦点を当てて最適化を行う．理想的な仮定の下では，性能は変化しないのにエネルギー消費が最適化される例をみる．

DVS はハードウェア的には成熟された技術であり，研究用途などの試作プロセッサだけでなく，多くの商用プロセッサにおいて実装されている．たとえば，Intel XScale[20)]，Intel StrongARM[19)]，Transmeta[39)] Crusoe，AMD[2)] K6-II+，IBM PowerPC 405LP[28)] などが知られている．このように DVS はハードウェア的にはすでに確立された技術である．また，ハードおよびソフトリアルタイム制約下でエネルギー消費最適化[6),10),16),25),30),33)]，エネルギー制約下で性能最適化[6),10),30)]，および時間とエネルギーを媒介変数とする評価関数の最小化問題[6),26)] が広く研究されてきた．

DVS は，OS からみた処理の実行単位であるタスクをどう電圧制御するかにより，分類される．電圧制御を，タスク変更時に行うものをタスク間 DVS（InterDVS: intertask DVS），タスク実行時に行うものをタスク内 DVS（IntraDVS: Intratask DVS）とそれぞれ呼ぶ．InterDVS では，各タスクの電圧を決定するために OS に変更を加えなければならない．また，多くの組込みモバイルアプリケーションで，シングルタスク環境が用いられている．マルチタスク環境でも特定のタスクの実行時間が支配的であることがある[34)]．一方，IntraDVS では，あるタスクの電圧制御に必ずしも他のタスクの詳細な知識を必要としない．また，システムレベルで伝統的なスケジューリング技術を用いたときにも IntraDVS を用いること

© 2009 Information Processing Society of Japan

ができる．このため，本稿では，プログラム実行時に実行順序を制御し，IntraDVS によりエネルギー消費の最適化を行うことに焦点を当てる．

IntraDVS には，大きく分けて時分割法[24] とチェックポイント法[5),16),33)] がある．時分割法は，ある定められた時刻に，電圧を変更させる方法である．プログラムの構造を考慮していないため，問題固有のエネルギー消費の最適化が行えないことが，この手法の限界の 1 つである．チェックポイント法は，ソースコード中に実行時に電圧制御を行うポイント VSP を挿入する手法である．既存のチェックポイント法には，VSP をソースコード中に早期に出現させるために，条件式を変更し場所を移動させる手法がある[33]．しかし，プログラムの実行パスは元のプログラムと完全に同一であるという強い制限があった．

本稿では，チェックポイント法の IntraDVS に焦点を当て，プログラムの実行順序を評価戦略[40] を指定することで，DVS システムにおけるエネルギー消費の最適化に有効なプログラムを開発する．評価戦略をエネルギー消費の最適化に活用することは，本研究における新しい視点である．

3. 準　　備

本稿では，DVS システムで実行されるプログラムを関数型言語を用いて記述する．以下に，本稿で必要とされる，関数プログラミングおよび DVS システムの周辺知識を示す．

3.1 表　記　法

本稿では，関数型言語 Haskell[7] にならった記法を用いる．

関数と演算子　関数適用は括弧を省略した形で表記し，一般的表記法の `f(x)` ではなく，`f x` と書く．関数はカリー化（currying）する．カリー化された関数適用は左結合的である．たとえば，`f a b` = `(f a) b` である．関数適用は 2 項演算子よりも結合の順序が高いものとする．たとえば，`f a + b` は括弧を明示的に表すと，`(f a) + b` である．関数定義は等式で表す．たとえば，与えられた数を 2 乗する関数は

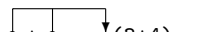
```
sq x = x * x
```

と記す．本稿では，与えられた式の中において，ある関数定義の左辺の出現を，その関数定義の右辺に書き換えることを，1 ステップと数えることにする．

リスト　リストは同一の型の要素を 1 次元的に並べたデータである．n 個の要素 $\mathtt{a}_1$，$\mathtt{a}_2$，…，$\mathtt{a}_n$ からなるリストは `[a1,a2,...,an]` と記す．2 つのリスト `xs`，`ys` の連接は演算子 `++` を用いて `xs ++ ys` と記す．たとえば

```
[1] ++ [3,4] ++ [2] = [1,3,4,2]
```

```
sq (3+4)
⇒ { definition of sq }
```




```
⇒ { definition of + }
```


```
⇒ { definition of * }
49
```

図 1　式 `sq (3+4)` の遅延評価
Fig. 1　Lazy evaluation of expression `sq (3+4)`.

である．

型　各関数定義の前に型宣言を行う．型宣言は，関数名，2 重のコロン，型からなる．たとえば，整数型 `Int` を引数にとり，`Int` を返す関数 `sq` は

```
sq :: Int -> Int
```

と記す．もし，`Int` の代わりに，順序関係演算子を含む型クラス `Ord` に属する多相型 `a` の上に `sq` が定義される場合は

```
sq :: (Ord a) => a -> a
```

と記す．

3.2 評　価　器

本稿では，遅延評価（lazy evaluation）を用いる．遅延評価は，最外グラフ簡約（outermost graph reduction）とも呼ばれ，共通する部分式はポインタで共有され式の外側から内側へ順に評価が行われていく．たとえば，`sq (3+4)` は**図 1** の順番で簡約化される．部分式 `3+4` はポインタの参照先であるため，1 度しか評価されてない．一般に，遅延評価は，式の内側から外側へ順に評価が行われていく先行評価（eager evaluation）よりステップ数が多いことがないという意味で，効率的である．

3.3 エネルギーモデルと DVS

DVS における周波数のスケーリング因子を s（$0 \leq s \leq 1$）で表す．本稿では，実行時間の逆数を性能と呼ぶ[15]．理想的な仮定[⋆1]の下では，性能は s に，動的エネルギー消費は s^2

⋆1 回路の閾値電圧を 0 に近似する．周波数は電圧に比例するものとする．

© 2009 Information Processing Society of Japan

に比例する[14]．たとえば，半分の性能で実行すると動的エネルギー消費は 1/4 になる．本稿では，このモデルに従ったエネルギー消費を最適化する．

3.4　ステップ数解析

本稿では評価時間の傾向を表す指標の 1 つである評価ステップ数[7),31)] を用いて解析を行う．関数 f の定義からその評価ステップ数を計算する関数 T_f に翻訳する規則は

$$\{\ f\ a_1\ a_2\ \cdots\ a_n\ =\ e\ \}\ =\ \{\ T_f\ a_1\ a_2\ \cdots\ a_n\ =\ 1+T(e)\ \} \qquad (1)$$

と定義される．ここで式の評価ステップ数 $T(e)$ は**図 2** のように表される．$T(e)$ の値は正格評価によるものであり，遅延評価による評価ステップ数はこれより小さくなりうることに注意されたい．

可変長データを入力とする多くの関数は，そのサイズに依存して評価ステップ数が決まる．簡単のため，入力データサイズ n のときの関数 f の評価ステップ数を $T_f(n)$ という記法を用いて表す．たとえば，リストの長さを返す関数

```
length       :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

が与えられたとき，リスト `xs` の長さが n とすると，

$$T_{\texttt{length}}(n)\ =\ T_{\texttt{length}}\ \texttt{xs}\ =\ n+1$$

と表される．条件文が関数定義 f に含まれる場合は，その関数の最悪および平均の評価ステップ数を考えることができる．それらを上付きのラベルによりそれぞれ T_f^{wc} および T_f^{ac} と表す．

$$\begin{aligned}
T(c) &\Rightarrow 0\\
T(v) &\Rightarrow 0\\
T(\texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c) &\Rightarrow T(a) + \mathit{if}\ [\![a]\!]\ \mathit{then}\ T(b)\ \mathit{else}\ T(c)\\
T(b\ \texttt{where}\ \mathit{pat} = e) &\Rightarrow T(b) + T(e)\\
T(p\ a_1\ a_2\ \cdots\ a_n) &\Rightarrow T(a_1) + T(a_2) + \cdots + T(a_n)\\
T(f\ a_1\ a_2\ \cdots\ a_n) &\Rightarrow T(a_1) + T(a_2) + \cdots + T(a_n) + T_f\ a_1\ a_2\ \cdots\ a_n
\end{aligned}$$

図 2　式の評価ステップ数
Fig. 2　Step-counting of expressions.

4. エネルギー消費の最適化が有効なプログラムの開発

リアルタイムシステムにおいて，タスクの実行には時間制約がある．タスクが終了していなければならない時刻をデッドラインと呼ぶ．デッドラインとタスクの終了時刻の差をスラック（slack）時間と呼ぶ．スラック時間の範囲内で，タスク実行時に性能および電圧の低下をすることで，システムのエネルギー消費の最適化を行うことができる．

図 3 にエネルギー消費の最適化が有効なプログラムの開発の流れを示す．与えられた入力プログラムの再帰構造から，計算の情報を表現するデータ型，およびそのデータ型を走査する関数を生成する．これと入力プログラムを組化し，中間プログラムを生成する．中間プログラムとエネルギー消費を考慮した評価戦略を統合することで目的のプログラムが得られる．本提案手法の特徴の 1 つは，入力プログラムと評価戦略を独立に扱うことである．

以下，基本的なアルゴリズムである整列，選択，および文字列検索に適用することを通じ，本手法の詳細を示す．こういった基本的なアルゴリズムに適用できることから，本手法はより複雑なアルゴリズム群にも適用できることが期待できる．

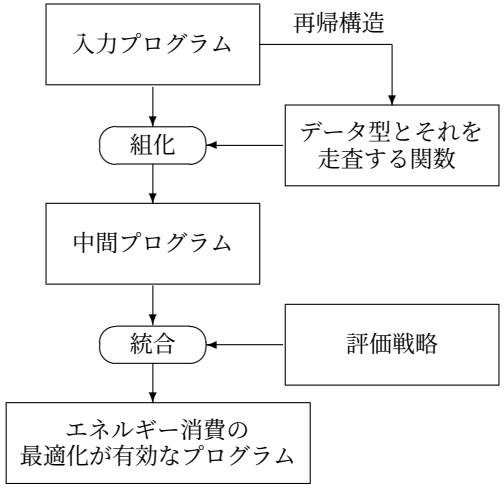


図 3　エネルギー消費の最適化が有効なプログラムの開発の流れ
Fig. 3　Our development flow of energy efficient programs.

© 2009 Information Processing Society of Japan

4.1　クイックソートと評価戦略

プロセッサコアの電圧・周波数の状態は破壊的に更新されるため，プログラミングにおいては副作用として扱うことが適切であろう．本稿では，電圧制御を IO モナドを通して行い，それ以外の方法で電圧制御を行わない．また，与えられた引数の値にコアの供給電圧を設定する関数

```
setV :: Double -> IO ()
```

をラッパ

```
set :: Int -> IO ()
```

を通して使用する．関数 set は残り評価ステップ数を引数にとり，環境よりデッドラインまでの時間を取得し，デッドラインを満たす範囲で最も低いコア供給電圧を setV を用いて設定する．

順序 Ord の定義された型 Elm のリストを [Elm] とする．昇順に整列されたリストを返すクイックソートは以下のように与えられる．

```
qsort :: [Elm] -> [Elm]
qsort []     = []
qsort (x:xs) = qsort ls ++ [x] ++ qsort us
  where (ls,us) = splitby x xs

splitby :: Elm -> [Elm] -> ([Elm],[Elm])
splitby x []     = ([],[])
splitby x (y:ys) = if x < y then (x:ls,us) else (ls,x:us)
  where (ls,us) = splitby x ys
```

関数 qsort は，空リスト [] を受け取ると空リストを返す．また，空でないリストを受け取ったときは，先頭 x をピボットに選択し splitby x xs により残りのリスト xs を x より小さい要素からなるリスト ls および，x より大きいか等しい要素からなるリスト us に分割する．分割された 2 つのリストは再帰的に整列が行われ，その結果を連接することで全体の結果が得られている．簡単のため，本稿では，リストの連接 (++) の計算量は $\Theta(1)$ と仮定する[⋆1]．

関数 splitby および qsort の評価ステップ数は

$$T_{\texttt{splitby}}(n) = n + 1 \tag{2}$$

$$T^{\mathrm{wc}}_{\texttt{qsort}}(n) = \frac{1}{2}n^2 + \frac{5}{2}n + 1 \tag{3}$$

$$T^{\mathrm{ac}}_{\texttt{qsort}}(n) \approx n \lg n + 2n \tag{4}$$

と表される．ただし，$T_{\texttt{splitby}}$ の引数は第 2 引数のリストの大きさを表しているものとする．関数 qsort の最悪および平均評価ステップ数はそれぞれ $\Theta(n^2)$ および $\Theta(n \lg n)$ である．したがって，クイックソートのみのタスクを考えた場合，平均的には $\Theta(n \lg n)$ の時間で終了するにもかかわらず，少なくとも $\Theta(n^2)$ 以降の時刻にデッドラインを設定しなければならない．このとき，最悪および平均評価ステップ数の差は平均的なスラック時間を表している．これは入力サイズが大きくなるにつれて大きくなる．このスラック時間の実行早期における見積りがエネルギー消費最適化の鍵である．

一般に，クイックソートにおける中間結果は

$$\texttt{qsort xs}_1 \texttt{ ++ [x}_1\texttt{] ++ qsort xs}_2 \texttt{ ++ [x}_2\texttt{] ++ } \cdots \texttt{ ++ [x}_{m-1}\texttt{] ++ qsort xs}_m \tag{5}$$

と表される．メモリ使用量を削減するために，通常のライブラリでは qsort の再帰ケースにおいて ls と us の長さが小さい方から計算を始める．このとき，スタックフレームのサイズは $\mathrm{O}(\lg n)$ となる．

評価順序を入れ替えることで，評価時における，残り予測評価ステップ数の減少量は大幅に変化しうる．たとえば，qsort の再帰ケースにおいて ls と us の長さが大きい方から計算を始めたり，クイックソートの中間結果 (5) において $\texttt{xs}_i$（$1 \leq i \leq m-1$）が一番長い方から計算を始めたりすると残り予測評価ステップ数は早期に大幅に減少する．ただし，このとき必要な空間計算量は増す．このようにエネルギー消費が効果的に最適化される評価戦略に沿って，エネルギー消費の最適化が有効なプログラムを新たに開発することができる．

しかし，エネルギー消費の最適化を既存プログラム中に記述することには短所が 2 つある．第 1 に，意味が明白な，もしくはすでに正当性がある程度確かめられたプログラム（上の例では，qsort）へ拡張を加えたときに，良い性質を壊してしまう恐れがある．たとえば，新たなバグを埋め込んだり，プログラムのモジュール性を損なったりすることが考えられる．それぞれ局所的にエネルギー最適化が行われた複数のプログラムを組み合わせて大きなプログラムを作ったり，複数タスクを OS で制御したりするときに全体最適につながるとは限らないのである．第 2 に，エネルギー消費を効果的に最適化する評価戦略は，単一のプログラム（上の例では，qsort）だけではなく共通の構造がある複数のプログラムに適用でき

⋆1 これは，リストの表現を変更することで実現できる[17]．

© 2009 Information Processing Society of Japan

る．しかし，各プログラムをそのつど変更する方法では，こういった評価戦略の明示的な再利用が行えていない．

4.2　評価戦略の開発とその統合

以降では，再利用可能な評価戦略の開発と既存プログラムとの統合を用いることで，元のプログラムに対する部分正当性を保証し，プログラムおよび評価戦略の双方にモジュール性を持たせる．評価中に分割のされ方の情報にアクセスし，また残り評価ステップ数を計算するために中間データ構造

```
data BTree = Leaf | Bin [Elm] BTree BTree
```

を導入する．この木は qsort の関数呼び出しの依存関係を表すのに使われる．ここで qsort はベースケースと再帰ケースの 2 通りの定義があったが，これが上記二分木の葉とノードにそれぞれ対応している．各葉および各ノードの情報で残り評価ステップ数を予測できるようにする．葉に対応する残り評価ステップ数は 0 で，ノードにおいては splitby により分割されたリストが格納されていれば予測可能である．この木を生成する関数は

```
mkBTree :: [Elm] -> BTree
mkBTree []     = Leaf
mkBTree (x:xs) = Bin (x:xs) (mkBTree ls) (mkBTree us)
  where (ls,us) = splitby x xs
```

と定義することができる．

関数 mkBTree により生成された木の各ノードの情報を用いることで，残り評価ステップ数を計算することができる．ここで 3 つの問題がある．第 1 に，生成された二分木の情報をもとにどのように qsort の評価順序を指定するのか．第 2 に，qsort と mkBTree の両方でまったく同じ入力リストに対し splitby を 2 度評価するため，効率が悪い．第 3 に，mkBTree を先にすべて評価してしまっては，splitby を評価するときに電圧を効果的に低下させることができない．以降では，組化と遅延評価によりこれらの問題を解決する．

関数 qsort と mkBTree を組化することで，1 度の走査でリストの整列と上記の木を返す以下の関数が得られる．

```
tupling :: [Elm] -> ([Elm],BTree)
tupling []     = ([],Leaf)
tupling (x:xs) = (ls' ++ [x] ++ us', Bin (x:xs) t1 t2)
  where (ls, us) = splitby x xs
        (ls',t1) = tupling ls
        (us',t2) = tupling us
```

なお，この変換は，プログラム変換システム（たとえば，文献 36)）を用いることで自動化できる．

評価戦略 strategy の適用は

```
apply :: ([BTree] -> IO ()) -> [Elm] -> IO [Elm]
apply strategy xs = strategy [t] >> return ys
  where (ys,t) = tupling xs
```

と表せる．これを用いると，評価順序を指定しないときの単純な戦略を適応したときのクイックソートは

```
qsort2 :: [Elm] -> IO [Elm]
qsort2 = apply (\_ -> return ())
```

と表される．

関数 qsort において分割された部分リストのうち，リスト長が大きい方から計算する評価戦略は以下のように定義される．

```
eval1 :: [BTree] -> IO ()
eval1 []                  = return ()
eval1 (Leaf         : ts) = eval1 ts
eval1 (Bin xs t1 t2 : ts) = eval1 (insert t1 (insert t2 ts))

insert :: (Ord a) => a -> [a] -> [a]
insert Leaf ts     = ts
insert u    []     = [u]
insert u    (t:ts) = if u > t then u : t : ts else t : insert u ts
```

関数 eval1 の引数中の BTree の順序は，対応するリストの長さによって決定される．関数 insert は BTree を，リスト長の大きいものから評価されるように，降順に並べる．入力された木 BTree が葉 Leaf のときは挿入されない．関数 eval1 の右辺の部分式 insert t1 (insert t2 ts) の評価時に，分割されたリストの長さが比較される．このとき，組化された関数 tupling により生成された木が深さ 1 だけ走査され，遅延されていた関数 splitby の評価が行われる．このように，関数 tupling により生成された木を走査する順序によりクイックソートの評価順序が決まる．

上記評価戦略を用いた関数

© 2009 Information Processing Society of Japan

```
qsort3 :: [Elm] -> IO [Elm]
qsort3 = apply eval1
```

が実現できる．

ここで，元のプログラム qsort が，ソースコード自身としてはまったく変更されていないことに注意してほしい．関数 qsort は mkBTree と組化され評価戦略 eval1 と組み合わされたのである．一般にこの方法で開発されたプログラムは，評価が停止すれば元のプログラムと同じ結果を返す．すなわち，変更後のプログラムにおいて，元のプログラムに対し，部分正当性が成立している．評価戦略にかかったオーバヘッドを除けば，プログラムを変更する前後で，評価ステップ数はまったく変化していない．

関数 qsort3 を例として，遅延された計算がどのように駆動されるのかを見てみる．直観的に一言でいうと，評価戦略 eval1 により先行評価が行われている．たとえば，式 qsort3 [3,2,5,...] は次のような中間式に評価される．

```
eval1 [t] >> return ys
  where (ys,t) = tupling [3,2,5,...]
```

ここで値に影響があるのは return ys のみであることに注意してほしい．式 eval1 [t] の評価は，その過程で効果的電圧制御に必要な情報を求め，return ys の評価を途中まで進めることが目的である．次に，式 eval1 [t] が評価され eval1 (insert t1 (insert t2 [])) が得られる．ここで eval1 を評価するには t1，t2 の頭正規形が必要である．さらに頭が Bin のときはその第 1 引数の頭正規形も必要であり，

```
t1 = Bin [2,...] _ _
t2 = Bin [5,...] _ _
```

というところまで評価される．したがって tupling の where 句の中の splitby が駆動される．この splitby は元々のクイックソート qsort で問題分割に用いられていたものであり，return ys が部分的に評価されたと見なすことができる．一方，tupling の返値の組の第 1 要素はこの時点では評価されない．この最悪実行時間が変化しない連接にかかる評価は return ys の評価によって駆動されるのである．以降は eval1 の引数のリストの先頭から再帰的に評価が行われていき，頭正規形が返され評価が停止したらその値は破棄され return ys の評価に移りその結果が値として返される．

残り評価ステップ数を考慮に入れて電圧を設定する評価戦略を考える．

```
eval2 :: Int -> [BTtree] -> IO ()
eval2 s []                  = return ()
eval2 s (Leaf         : ts) = eval2 s ts
eval2 s (Bin xs t1 t2 : ts) = set s' >>
                              eval2 s' (insert t1 (insert t2 ts))
  where s' = s - wc_qsort xs + wc_qsort (list t1) + wc_qsort (list t2)
```

ここで，wc_qsort は $T^{\mathrm{wc}}_{\mathtt{qsort}}$ かその実行時間に相当する値を返す関数である．評価戦略 eval2 は残り評価ステップ数と qsort の計算途中を表した中間データ構造のリストを受け取り，問題の分割を進めていく．ここで

```
list :: BTtree -> [Elm]
list (Bin xs _ _) = xs
```

は木のルートノードのリストを返す．

この評価戦略を用いたクイックソートは

```
qsort4 :: [Elm] -> IO [Elm]
qsort4 xs = apply (eval2 (wc_qsort xs)) xs
```

と表される．このようにクイックソート qsort へまったく変更を加えることなく，変換戦略を独立に開発することができた．

4.3　評価戦略の改良

前節の評価戦略 eval2 は，直観的に理解しやすいが，実用上いくつかの問題点がある．たとえば，分割数が大きい場合の insert や，分割するごとに残り最悪ステップ数の更新は全体の評価時間に影響を与える．これらの問題点は，評価戦略をいくつかの視点から改良することで解決できる．評価戦略を変更することで，電圧スケジューリングや評価順序が変化する．

前節の評価戦略 eval2 の中では $T_{\mathtt{qsort}}$ で表されていた，残り予測実行時間を変更すると，set により設定される電圧が変化する．このように残り予測実行時間の選択が電圧スケジューリングに影響する．したがって，既存の有効な評価戦略[32] の中から，問題に適したものを選択・拡張することが重要である．評価戦略のオーバヘッドが無視できないとき，残り予測評価ステップ数が大幅に減る初期のみ評価戦略を用いることが考えられる．このためには，たとえば，分割されたリストの大きさがある一定以下であることをチェックするための引数を eval2 に渡してやるなどするだけでよい．また，プロセッサコアの電圧が最低まで下がっていたときには評価戦略を停止する方法も考えられる．これは s の値とデッドラインまでの時間をチェックすることで eval2 の再帰を打ち切ってやればよい．

前節で定義された eval2 は，分割された大きい方の問題から解くため，最悪評価時間が

© 2009 Information Processing Society of Japan

急激に減少していた．しかし，この方法では追加的に必要とされる空間消費量は入力サイズに対して線形となる．また計算が進み問題の分割数が増えてくると，残り評価ステップ数の変化は少なくなる．保持する部分問題数の上限を定めることで追加的に必要とされる空間消費量の上限を定めることができる．また，問題の分割回数を定めることで，残り評価ステップ数の変化が少なくなってきたとき，予測のための空間および時間のオーバヘッドをなくすことができる．

4.4　イントロソートへの応用

DVS システムにおけるクイックソートの問題点は，最悪および平均実行ステップ数の比が大きすぎることである．たとえば，StrongARM SA-1100 の周波数のスケーラビリティは約 4 倍である．したがって，実行時間の 4 倍を超える大量のスラック時間は有効に活用することができない．そのため，DVS システム上でエネルギー消費の最適化が行いやすいアルゴリズムを選択するには，最悪および平均実行ステップ数が大きくなりすぎないようにすることが重要である．

クイックソートを通してみた開発法は，他プログラムの開発にも同様に適用することが可能である．ここでは，最悪および平均の評価ステップのオーダが同じで，評価時における仕事量の変化（work load variation）によるスラック時間が DVS に効果的に用いられる量を生成するイントロソート[27]（introspective sort）について考える．順序 `Ord` の定義された型 `Elm` からなるリスト型 `[Elm]` を受け取り，整列されたリストを返すイントロソートは以下のように表される．

```
isort :: [Elm] -> [Elm]
isort xs = isort' lim xs
  where lim = 4 * log2 (length xs)

isort' :: Int -> [Elm] -> [Elm]
isort' 0   xs     = hsort xs
isort' lim []     = []
isort' lim (x:xs) = isort' lim' us ++ [x] ++ isort' lim' ls
  where (us,ls) = splitby x xs
        lim' = lim - 1
```

ここで，`log2 n` は底を 2 とし `n` の対数を超えない最大の整数を返し，`hsort` はヒープソートを表すものとする．ヒープソートは平均および最悪の実行時間差が少ないアルゴリズムで，平均および最悪ステップ数は

$$T^{\mathrm{ac}}_{\mathtt{hsort}}(n) \approx T^{\mathrm{wc}}_{\mathtt{hsort}}(n) \approx 3n \lg n \tag{6}$$

と表される[31]．イントロソートの最悪実行ステップはヒープソートが用いられるときである．したがってイントロソートの最悪実行ステップも式 (6) と同じく

$$T^{\mathrm{wc}}_{\mathtt{isort}}(n) \approx 3n \lg n \tag{7}$$

と表される．

イントロソートの部分関数 `isort'` では 3 つの場合分けにより評価がされている．したがって，実行過程を表すための木データ構造中のデータ構築子数も 3 つになり，

```
data BTtree2 = Leaf2 | Leaf2h | Bin2 [Elm] BTree2 Btree2
```

と表される．クイックソートに用いた `eval3` に対応する評価戦略は以下のように表される．

```
eval3 :: Int -> [BTtree2] -> IO ()
eval3 s []                    = return ()
eval3 s (Leaf2  : ts)         = eval3 s ts
eval3 s (Leaf2h : ts)         = eval3 s ts
eval3 s (Bin2 xs t1 t2 : ts) = set s' >>
                                 eval2 s' (insert t1) (insert t2 ts)
  where ls = list t1
        us = list t2
        s' = s - wc_isort xs + wc_isort ls + wc_isort us
```

ただし，`wc_isort` は $T^{\mathrm{wc}}_{\mathtt{isort}}$ かその実行時間に相当する値を返す関数である．

ここで，3 番目の場合分けにおいて，データ構成子 `Leaf2h` を無視し，ヒープソート `hsort` の評価結果を要求しないことで評価を遅延させ，さらに評価ステップ数を残り予測ステップ数 `s` から引いていないことに注意してほしい．前述の評価ステップ数 (6) ように，ヒープソートは平均および最悪実行ステップ数がほぼ同じであった．こういったスラック時間を生み出さない式はなるべく後で評価したほうが，全体のエネルギー消費が最適化される．このように，本稿のアプローチでは遅延評価器を用いているので，元プログラムの書き換えをすることなく，エネルギー消費の最適化が行われている．なお，評価戦略の適応法はクイックソートのときとまったく同じである．

4.5　他のアルゴリズムへの応用

前節までの開発手法を，他の分割統治のアルゴリズムに適用できることと，平均および最悪実行ステップ数に差があるアルゴリズムに適応できることを示す．

© 2009 Information Processing Society of Japan

4.5.1 Hoare の選択アルゴリズム

与えられた整列されていないリストから与えられた自然数番目に小さい要素を返す問題を考える．Hoare の選択アルゴリズム[12] は，クイックソートにおける分割に相当することを行う．ただし，クイックソートと異なり，与えられた自然数番目に小さい要素が入っている可能性がある，分割された問題の片方のみを再帰的に処理するだけでよい．最悪実行ステップ数が $O(n^2)$ ではあるが，平均実行ステップ数が $\Theta(n)$ でありほとんどの入力に対し良い性能を示す．Hoare の選択アルゴリズムは，以下のように表される．

```
qselect :: Int -> [Elm] -> Elm
qselect i (x:xs) = if m + 1 > i then qselect i ls
                                else if m + 1 == i
                                       then x
                                       else qselect (i-m-1) us
  where (ls,us) = splitby x xs
        m = length ls
```

途中結果を返すために，前節まで同様中間データ構造を導入する．ただし，今回は `qselect` 中の再帰は線形であるので，新たなデータ型を導入しなくても，リストを中間データ構造として用いることができる．`qselect` とこのリストを走査する関数を組化し，評価戦略

```
eval4 :: Int -> Int -> [Elm] -> IO ()
eval4 s i (x:xs) =
  if m + 1 > i then
    let s' = s - wc_qselect (x:xs) + wc_qselect ls
    in set s' >> eval4 s' i ls
  else if m + 1 == i then return ()
    else let s' = s - wc_qselect (x:xs) + wc_qselect us
         in set s' >> eval4 s' (i-m-1) us
  where (ls,us) = splitby x xs
        m = length ls
```

と組み合わせることで，プログラムは DVS システム上でエネルギー消費の最適化がより有効になる．ただし，`wc_qselect` は $T^{\mathrm{wc}}_{\mathtt{qselect}}$ かその実行時間に相当する値を返す関数である．

4.5.2 内省的選択アルゴリズム

前節の整列アルゴリズムの内省化と同様に，選択アルゴリズムを内省化する．これにより，DVS システム上でエネルギー消費の最適化がより有効になる．以下で考える内省的選択アルゴリズムは，入力リストの長さを n としたとき，実行ステップ数が $\Theta(n)$ である[27]．平均的には Hoare の選択アルゴリズムで高速に終了する．しかし，`k` 回の分割で長さが半分にならないときには線形時間選択アルゴリズムに切り替えられる．この内省的選択アルゴリズムは以下のように表される．

```
iselect :: Int -> [Elm] -> Elm
iselect i xs = iselect' (k,length xs) i xs

iselect' :: (Int, Int) -> Int -> [Elm] -> Elm
iselect' (-1,len) i (x:xs) = linear_select i (x:xs)
iselect' (th,len) i (x:xs) =
  if u + 1 > i then
     iselect' (th',len' `div` 2) i us
  else if u + 1 == i then x
       else iselect' (th',len' `div` 2) (i-u-1) vs
  where (us,vs) = splitby x xs
        (len',th') = if length (x:xs) <= len then (length (x:xs),k)
                     else (len,th)
```

```
eval5 :: (Int, Int) -> Int -> [Elm] -> [Int]
eval5 (-1,len) i (x:xs) = []
eval5 (th,len) i (x:xs) =
  if u + 1 > i then ret us : eval5 (th-1,len' `div` 2) i us
  else if u + 1 == i then []
       else ret vs : eval5 (th-1,len' `div` 2) (i-u-1) vs
  where (us,vs) = splitby x xs
        u = length us
        len' = if length (x:xs) <= len then length (x:xs) else len
```

図 4　内省的選択アルゴリズムの評価戦略
Fig. 4　Evaluation strategy for introspective selection algorithm.

© 2009 Information Processing Society of Japan

これに対応する評価戦略は**図 4** のように表される．残りの開発は前節までと同様である．ここで，バッククオートで囲まれた関数‘·‘は中値演算子を表している．

4.5.3 Rabin-Karp 文字列検索アルゴリズム

与えられたテキストの中に出現するパターンの位置をすべて返す問題を考える．Rabin-Karp 文字列検索アルゴリズム[12),23)] は，パターンとテキストを直接照合するのではなく，パターンとテキストそれぞれから作り出したハッシュどうしを照合することで高速化されている．このアルゴリズムは，パターンの長さが m，テキストの長さが n のとき，平均 $\Theta(m)$，最悪 $\Theta((n-m+1)m)$ の実行時間がかかる．この差が多くの実行においてスラック時間になる．

```
rkmatch :: (Pat,Text) -> Int -> Int -> [Int]
rkmatch (ps,ts) d q = match 0 h0 (ps,ts)
  where n = length ts
        m = length ps
        h0 = pow(d,m-1) `mod` q
        (p,h0) = mkPattern (0,0) (ps,ts)

        mkPattern :: (Int,Int) -> (Pat,Text) -> (Int,Int)
        mkPattern (p,h) ([], _) = (p,h)
        mkPattern (p,h) (ps,ts) = mkPattern ((d*p0+(hd ps)) `mod` q,
                                             (d*h+(hd ts)) `mod` q)
                                            (tail ps,tail ts)

        match :: Int -> Int -> (Pat,Text) -> [Int]
        match s h (ps,ts)
          | s > n-m = []
        match s h (ps,ts) = if p == h then if take m ps == take m ts
                                           then s:rs
                                           else rs
                            else rs
          where h' = (d*(h-(hd ts)*h) + ts!!m) `mod` q
                rs = match (s+1) h' (tail ps) (tail ts)
```

図 5　Rabin-Karp 文字列検索アルゴリズム
Fig. 5　Rabin-Karp string match algorithm.

パターンとテキストは

```
type Pat  = String
type Text = String
```

のように文字列型と宣言する．Rabin-Karp 文字列検索アルゴリズムの実装を**図 5** に示す．前節までと同様の方法で，エネルギー消費の最適化が有効なプログラムが得られる．

パターンとテキストのハッシュが一致してしまった場合，図 5 のプログラムではパターンとテキストを直接照合しなければならない．また，この部分のパターンとテキストの照合の最悪実行時間は大幅に変化することはない．遅延評価を用いることで，このような最悪実行時間が減少する可能性が薄い部分の計算を後回しにすることができていることに注意してほしい．このことは，実行初期にスラック時間を増加させることに貢献しており，エネルギー消費の最適化に役立っている．

5. 評 価 実 験

前章までは，特定のアーキテクチャから離れて議論を行ってきた．このような細かなチューニングをしない一般的な提案手法が，現実的かつ具体的なアーキテクチャを想定した場合，どれほど効果的なのかを実験を行い評価する．本章では，例として，ARM ベースのアーキテクチャを考える．この場合，実行時間の計量には，ステップ数ではなく，より真の実行時間に近いサイクル数を用いれるので，これを用いる．また，現実に近づけるため電圧は離散値をとり，上下限が存在するものとする．実験対象プログラムは実行時に最悪実行時間の減少幅の大きい例であるイントロソート，小さい例である Rabin-Karp 文字列検索アルゴリズムを対象とした．

5.1 手　　順

実験には，C 言語記述である各ソースコードを，SimpleScalar/ARM[1),4)] を拡張して作られた，電力モデル付きのマイクロプロセッサシミュレータである Sim-Panalyzer 2.0.3[37)] の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 1 つである ARM7TDMI の命令セットに準拠した ISS である．

© 2009 Information Processing Society of Japan

表 1　プロセッサの構成
Table 1　Configuration of processor.

Fetch queue size	2
Branch predictor	Not taken
Fetch, decode width	1
Issue	In-order
Functional units	1 integer ALU, 1 integer multiplier/divider, 1 floating point ALU, 1 floating point multiplier/divider
Instruction L1 cache	8 KB; 64 set; 4 way; block size 32 B; LRU
Data L1 cache	8 KB; 64 set; 4 way; block size 32 B; LRU
L2 cache	None
Memory bus width	4 B

表 2　コアの電圧，周波数，および SDRAM へのアクセスのレイテンシ
Table 2　Voltage, frequency, and SDRAM access latency.

Voltage (V)	Frequency (MHz)	Latency (cycle)
0.80	59.0	4
0.87	73.7	5
0.94	88.5	6
1.01	103.2	6
1.08	118.0	6
1.15	132.7	8
1.22	147.5	9
1.29	162.2	10
1.36	176.9	11
1.43	191.7	12
1.50	206.4	12

表 1 に，実験で用いた，Sim-Panalyzer に入力するプロセッサの構成を示す．Sim-Panalyzer はこれらの項目のそれぞれについて，動的および静的のエネルギー消費を計算する．**表 2** に，コアの電圧，周波数，および SDRAM へのアクセスのレイテンシを示す．これらの値は StrongARM SA-1100 [19] を参考に選択した．最大スケーラビリティは，電力が約 12 倍（$\approx (1.50/0.80)^2 \times (206.4/59.0)$），性能が約 3.6 倍（周波数より 206.4/59.0 倍とメモリアクセスのレイテンシを考慮）である．

実験のワークフローは以下のとおりである．まず，Haskell を用いて，通常どおりプログラムを記述する．これとは独立に評価戦略を開発し，図 3 の流れに沿って，エネルギー消費の最適化が有効なプログラムを得る．ここまでは前章までの説明どおりである．Haskell から Sim-Panalyzer 上で動作するバイナリコードへのコンパイラが手に入らなかったので，Haskell から C へ，意味を保ったまま手動で書き換えを行った．このソースコードから，gcc によるコンパイルにより，Sim-Panalyzer 上で動作するバイナリコードを得た．このバイナリコードおよび表 1 と表 2 の構成を入力とし，消費エネルギーや実行サイクル数をはじめとする統計情報を得た．

4 章で作成したプログラムに 4.3 節の改良を施し，手動で書き換えられたイントロソートを**図 6** に示す．イントロソートを行う関数 `introsort` は配列とその長さを引数にとる．まず，スタック変数などの局所定義があり，値が代入される．ここで分割された配列の左端の要素を `leftstack` が，右端の要素を `rightstack` が，クイックソートからヒープソートに移る再帰回数を `limstack` が保持している．初期のみ評価戦略を使うというのは，スタック内の順序を制御する最大回数 `max` を用いることで実現されている．ヒープソートの計算は遅延され最後に計算される．電圧・周波数は残りの問題サイズにより設定される．

イントロソートの入力配列はランダムなものを選び，100 から 10000 まで長さを変化させた．デッドラインは，同一の配列長の入力の中で問題分割のフェーズで必ず 0 個の要素の問題と残りすべての要素の問題に分割され，ヒープソートのフェーズでランダム入力を整列するのにかかる実行時間と仮定した．デッドラインはイントロソートの最悪実行時間であると期待される．n を問題の入力サイズとした式 (7) をデッドラインに正規化し評価戦略中で予測残り実行時間を見積もった．

一方，Rabin-Karp 文字列検索アルゴリズムのパターンの配列長は 50，テキストの配列長は 20000 とした．パターンおよびテキストから作成したキャッシュどうしの照合が成功する確率を 1/2 とし，キャッシュどうしの照合とパターン対テキストの照合の実行時間比を 10 倍となるように入力データを作成した．計算時間が最大となる入力は，パターンおよびテキストのキャッシュがつねにヒットするものと仮定した．これは図 5 において，関数 `match` の右辺で `p==h` がつねに成り立つときである．デッドラインは，上で述べた計算時間が最大となる入力が与えられたときに最高電圧（1.50 V）で実行したときの終了時刻とした．

評価戦略は，DVS オフ，実行時に各 VSP 通過時に最悪実行時間に間に合うような最小の電圧を選択する単純な評価戦略[33]（以降，RWET 戦略），平均実行時間にちょうど間に合うような評価戦略[33]（以降，ACET 戦略），遅延評価を用いた RWET 戦略（以降，L-RWET 戦略）を用いた．評価戦略はアーキテクチャの詳細は意識せずに汎用的書かれたものであり，イントロソートと Rabin-Karp 文字列検索アルゴリズムの双方で同一の評価戦略を用いていることに注意してほしい．各 VSP における電圧選択は大域的なエネルギー最適化を考えずに行った．

© 2009 Information Processing Society of Japan

```
void introsort(keytype a[], int n)
{
    int leftstack[STACKSIZE], rightstack[STACKSIZE], limstack[STACKSIZE];
    ... /* 局所変数定義 */

    leftstack[0] = 0; rightstack[0] = n - 1;
    limstack[0] = 4 * floor(log(n)/log(2)); p = 1;
    max = floor(log(n)/log(2));

    for ( ; ; ) {
        if (p == 0) break;
        p--;
        left = leftstack[p]; right = rightstack[p]; lim = limstack[p];
        if(right - left <= 0) continue;
        if(lim==0) {
            push_heapstack(left,right); /* ヒープソートの計算を遅延 */
            continue;
        }
        ... /* 通常のクイックソートと同様に
               a[i] より小さいものを a[left..i-1] に,
               大きいものを a[j+1..right] に集める... */
        set_voltage_and_frequency(remaining_size(...)); /* 電圧・周波数を設定 */

        leftstack[p] = left; rightstack[p] = i - 1; limstack[p] = lim - 1;
        max--;
        if(max > 0)
            insert(leftstack,rightstack,limstack,p);
        p++;
        ... /* 直前の a[left..i-1] の場合と同様に
               a[j+1..right] の結果をスタックで記憶しておく */
        if(max == 0)
             swap(leftstack,rightstack,limstack,p);
        p++;
    }
    ... /* 遅延された heapsort の実行 */
}
```

図 6　C に書き換えられた変換後のイントロソートプログラム
Fig. 6　Manually translated introsort program.

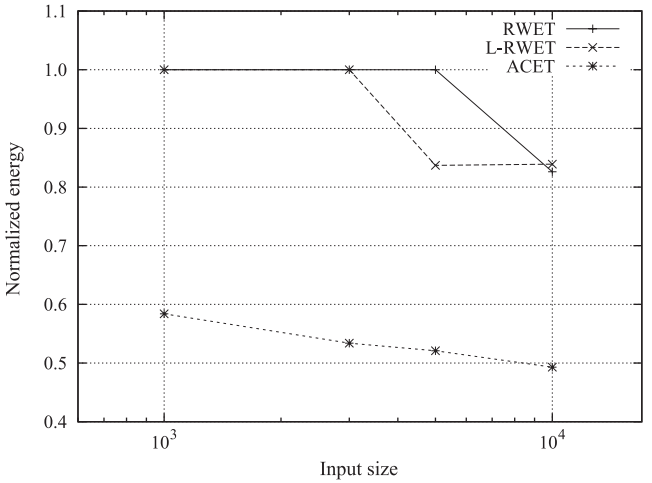


図 7　DVS システムにおける整列プログラムの正規化エネルギー
Fig. 7　Normalized energy of sorting programs on DVS system.

動的電圧周波数制御の切替オーバヘッドを**表 3** に示す．文献が明示されていないデータは文献 32) から引用した．上半分が商用の，下半分が研究段階の DVS プロセッサである．エネルギーオーバヘッドはどれも約数 μJ 以上であると思われる．最後の Multi-Performance Processor（MPP）は各電圧・周波数に特化した設計の行われたコアを切り替えて計算を行うプロセッサであり，オーバヘッドが非常に小さくなっている．エネルギーオーバヘッドは約 10 nJ である．本実験では，変更前後の電圧によらず，時間オーバヘッドは約 150 μs，エネルギーオーバヘッドは約 3 μJ であると仮定する．

5.2　実 験 結 果

イントロソートの実験結果を**図 7** に示す．すべての戦略は DVS オフの場合で正規化されている．RWET 戦略と L-RWET 戦略は入力配列長が小さいときは DVS オーバヘッドが相対的に大きいため電圧変更がされなかった．入力配列長が大きくなると実行早期にスラック時間を生み出しやすい L-RWET 戦略の消費エネルギーが削減され，続いて RWET 戦略も L-RWET 戦略と同様に消費エネルギーが削減されていた．一方，ACET 戦略は入力配列長が小さいときも実行開始時から低い電力が設定され，結果として消費エネルギーが大幅に削減されていた．

Rabin-Karp 文字列検索アルゴリズム実行時の消費電力の推移を**図 8** に示す．消費電力

© 2009 Information Processing Society of Japan

表 3　動的電圧周波数制御の切替えオーバヘッド
Table 3　Transition overhead of dynamic voltage/frequency scaling.

	Processor	Clock range (MHz)	Voltage range (V)	Transition time (μs)
Commercial	Transmeta Crusoe	200 – 700	1.1 – 1.65	300
	AMD Mobile K6	192 – 588	0.9 – 2.0	200
	Intel PXA250	100 – 400	0.85 – 1.3	500
	Compaq Itsy	59.0 – 206.4	1.1 – 1.6	189
	TI TMS320C55x	6 – 200		3300 (1.6 → 1.1 V)
				300 (1.1 → 1.6 V)
	IBM PowerPC 405LP [3]	33 – 266	1.0 – 1.7	117 – 162
Research	Burd *et al.*	5 – 80	1.2 – 3.8	520
	LART	59 – 251	0.79 – 1.65	5500 (1.65 → 0.79 V)
				40 (1.65 → 0.79 V)
	SH-4A [18]	75 – 600	1.0 – 1.4	92 (SMP Linux 600 → 300 MHz)
				94 (SMP Linux 300 → 600 MHz)
				34 (UP Linux 600 → 300 MHz)
				38 (UP Linux 300 → 600 MHz)
	MPP [21]	67 – 200	0.68 – 1.0	1.5

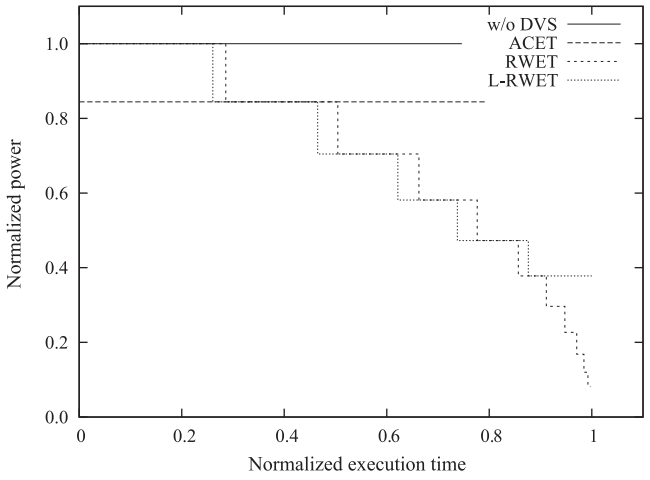


図 8　Rabin-Karp 文字列検索アルゴリズム実行時の消費電力の推移
Fig. 8　Power consumption transition of Rabin-Karp string search algorithm.

は DVS オフの場合で，実行時間は最悪実行時間で正規化されている．Rabin-Karp 文字列検索アルゴリズムはスラック時間が大幅に生成されないので，ACET 戦略はイントロソートのときのように大幅なエネルギー削減は達成できていなかった．また，RWET 戦略では終了間際に電力を低下させることができているにすぎなかった．一方，L-RWET 戦略では単純な RWET 戦略よりも早期に電力を低下させられており，使用されたコンフィギュレーションも RWET 戦略より少なかった．それぞれの線の下側の面積がそれぞれの消費エネルギーに対応している．ACET，RWET，L-RWET 戦略はそれぞれ 8.8%，11.6%，13.4%の消費エネルギー削減が確認された．

5.3 考　　察

イントロソートおよび Rabin-Karp 文字列検索アルゴリズムの双方に同一の評価戦略を用いることに成功し，評価戦略のモジュール性が示された．また，特定のアーキテクチャに特化した最適化を用いなくても現実的な仮定の下で本手法の有効性が示された．イントロソートは，実行初期に問題の分割が行われるたびに，大幅に残り最悪実行時間が減る．しかも，ある程度の大きさまでの入力およびデッドラインに対しては，クイックソートのように膨大なスラック時間が生成され，ほとんど使いきれないことは避けられる．このため L-RWET 戦略による，実行早期の最悪実行時間の正確な見積りが行われ，さらにエネルギー消費削減を実現できた．L-RWET 戦略には評価順序を制御するためのコードを実行するオーバヘッ

© 2009 Information Processing Society of Japan

ドがあるにもかかわらず，たとえばイントロソートの実験結果では単純な RWET 戦略よりも 20%近く良い消費エネルギーを示した．ただし，問題サイズが小さい場合はエネルギー削減はまったく行われなかった．これは DVS のオーバヘッドのためである．ACET 戦略は大幅に消費エネルギーを削減できていたが，デッドラインを守る保証がない．我々のアプローチをとれば，目的に応じてこういった評価戦略を簡易に変更することが可能である．

Rabin-Karp 文字列検索アルゴリズムでは L-RWET 戦略が最良の結果であった．我々のアプローチをとれば，最悪実行時間が変化しない計算の遅延は機械的に行われる．また，L-RWET 戦略は RWET 戦略よりも使用する電力数が少なかった．本稿では比較的大きい問題サイズで実験を行ったが，このオーバヘッドの影響は問題サイズが小さいほど増すと考えられる．したがって問題サイズが小さくなると L-RWET 戦略を用いる利点が増すと考えられる．

なお，電圧が連続値に近くなれば，RWET 戦略において各 VSP で最適な電圧が選択されるため，この意味で今回の結果が最適化されると思われる．したがって，将来のプロセッサでは提案アプローチはさらに効果的であることが期待される．

6. おわりに

本稿では，デッドライン制約のある DVS システムにおいてエネルギー消費の最適化が有効なプログラムの開発法を示した．従来，動的電圧制御や評価順序の制御は VSP の埋め込みや制御構造を変更し元のプログラムを改変することで実現されてきた．このため部分正当性の保証や元のプログラムおよび評価戦略にモジュール性を与えることはプログラマの責任であった．我々は，アルゴリズムと評価戦略を分離し，この二者を独立に開発する手法を提案した．この二者の組合せは組化や遅延評価により半自動的で実現された．この結果，プログラムと評価戦略はそれぞれモジュール性を有するようになった．すなわち，元のプログラムの可読性は損なわれず，目的プログラムは元のプログラムに対し部分正当性を保ち，汎用的評価戦略プログラムを元のプログラムから独立して開発可能になったのである．

筆者らの知る範囲では，本稿は，遅延評価を応用しエネルギー消費の最適化を実現した初めての報告である．遅延評価は，目的プログラムの導出を容易にするなど，重要な役割を果たしている．基本的な考え方は，残り最悪実行時間が減らない（スラック時間が増えない）計算は遅延しなるべく後に実行することにある．また，参照透過性や遅延評価のおかげで，構成的アルゴリズム論における組化などを活用しても，評価戦略と元のプログラムの変換の前後で意味が変わらないことを保証しやすくなっていた．

本稿ではこういったプログラミング方法論が，整列や文字列検索などの基本的なアルゴリズムに対し，効果があることを実験を通し確認できた．したがって，他の複雑なアルゴリズムに対しても，本アプローチの効果があると予想される．また，本稿ではエネルギー消費を指標としこの最適化を行ったが，目的によって指標を変更し別の問題に適用されることが期待される．

提案手法によって開発されたプログラムに対し，現実的かつ具体的なアーキテクチャを想定し電力モデル付きの ISS により評価実験を行った．

我々は，組込みシステムにおけるエネルギー消費の最適化のため，ソフトウェアとハードウェアの協調が効果的だと考え，研究を行っている．既存ハードウェア技術のソフトウェアでの効率的制御だけでなく，エネルギー消費の最適化のためにソフトウェア主導でハードウェア構成への示唆を行うことが今後の課題である．

謝辞　本稿の草稿にコメントをいただいた安積卓也氏，西田直樹氏に感謝いたします．本研究の一部は，科学技術振興事業団（JST）戦略的創造研究推進事業（CREST）の支援による「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」の研究プロジェクトの一環として行われた．本研究の一部は，科学研究費補助金基盤研究（C）19500026 の助成による．

参考文献

1) SimpleScalar LLC. http://www.simplescalar.com/
2) Advanced Micro Devices, Inc. http://www.amd.com/
3) Anantaraman, A.V., Mahmoud, A.E.-H., Venkatesan, R.K., Zhu, Y. and Mueller, F.: EDF-DVS Scheduling on the IBM Embedded PowerPC 405LP, *IBM Thomas J. Watson $P = ac^2$ Conference* (2004).
4) Austin, T., Larson, E. and Ernst, D.: SimpleScalar: An Infrastructure for Computer System Modeling, *Computer*, Vol.35, No.2, pp.59–67 (2002).
5) Azevedo, A., Issenin, I., Cornea, R., Gupta, R., Dutt, N., Veidenbaum, A. and Nicolau, A.: Profile-Based Dynamic Voltage Scheduling Using Program Checkpoints, *Design, Automation and Test in Europe*, pp.168–175, IEEE Computer Society (2002).
6) Barnett, J.A.: Dynamic Task-Level Voltage Scheduling Optimizations, *IEEE Trans. Comput.*, Vol.54, No.5, pp.508–520 (2005).
7) Bird, R.: *Introduction to Functional Programming Using Haskell*, 2nd edition, Prentice Hall (1998).
8) Brooks, D., Tiwari, V. and Martonosi, M.: Wattch: A Framework for Architectural-

© 2009 Information Processing Society of Japan

Level Power Analysis and Optimizations, *Computer Architecture*, pp.83–94, ACM Press (2000).
9) Burstall, R. and Darlington, J.: A Transformation System for Developing Recursive Programs, *J. ACM*, Vol.24, No.1, pp.44–67 (1977).
10) Cao, G.: Proactive Power-Aware Cache Management for Mobile Computing Systems, *IEEE Trans. Comput.*, Vol.51, No.6, pp.608–621 (2002).
11) Chung, E.-Y., Benini, L., Micheli, G.D., Luculli, G. and Carilli, M.: Value-Sensitive Automatic Code Specialization for Embedded Software, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.21, No.9, pp.1051–1067 (2002).
12) Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C.: *Introduction to Algorithms*, 2nd edition, MIT Press (2001).
13) Gelsinger, P.P.: Microprocessors for the New Millennium: Challenges, Opportunities, and New Frontiers, *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pp.22–25 (2001).
14) Henkel, J. and Parameswaran, S. (Eds.): *Designing Embedded Processors: A Low Power Perspective*, Springer (2007).
15) Hennessy, J.L. and Patterson, D.A.: *Computer Architecture: A Quantitative Approach*, 4th edition, Morgan Kauffman (2007).
16) Hsu, C.-H. and Kremer, U.: The Design, Implementation and Evaluation of a Compiler Algorithm for CPU Energy Reduction, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.38–48 (2003).
17) Hughes, R.J.M.: A Novel Representation of Lists and Its Application to the Function Reverse, *Information Processing Letters*, Vol.22, No.3, pp.141–144 (1986).
18) 出原章雄，田原康宏，山本　整，菅井尚人，飯塚　剛：組込みマルチコアプロセッサ向け SMP Linux における省電力機能の実装と評価，組込みシステムシンポジウム 2008, pp.115–123.
19) Intel Corp.: *Intel StrongARM SA—1100 Microprocessor Developer's Manual* (1999).
20) Intel Corp.: XScale. http://developer.intel.com/design/intelxscale/
21) Ishihara, T., Yamaguchi, S., Ishitobi, Y., Matsumura, T., Kunitake, Y., Oyama, Y., Kaneda, Y., Muroyama, M. and Sato, T.: AMPLE: An Adaptive Multi-Performance Processor for Low-Energy Embedded Applications, *Symposium on Application Specific Processors*, pp.83–88 (2008).
22) Jain, R., Molnar, D. and Ramzan, Z.: Towards a Model of Energy Complexity for Algorithms, *Wireless Communications and Networking Conference*, Vol.3, pp.1884–1890 (2005).
23) Karp, R.M. and Rabin, M.O.: Efficient Randomized Pattern-Matching Algorithms, *IBM Journal of Research and Development*, Vol.31, No.2, pp.249–260 (1987).
24) Lee, S. and Sakurai, T.: Run-Time Voltage Hopping for Low-Power Real-Time Systems, *Design Automation Conference*, pp.806–809 (2000).
25) Lorch, J.R. and Smith, A.J.: Improving Dynamic Voltage Scaling Algorithms with PACE, *ACM SIGMETRICS Performance Evaluation Review*, Vol.29, No.1, pp.50–61 (2001).
26) Martin, A.J.: Towards an Energy Complexity of Computation, *Infomation Processing Letters*, Vol.77, No.2-4, pp.181–187 (2001).
27) Musser, D.R.: Introspective Sorting and Selection Algorithms, *Software — Practice and Experience*, Vol.27, No.8, pp.983–993 (1997).
28) Nowka, K.J., Carpenter, G.D. and Brock, B.C.: The Design and Application of the PowerPC 405LP Energy-Efficient System-on-a-Chip, *IBM Journal of Research and Development*, Vol.47, No.5/6, pp.631–639 (2003).
29) Pedram, M. and Rabaey, J.M. (Eds.): *Power Aware Design Methodologies*, Kluwer Academic (2002).
30) Qiu, Q. and Pedram, M.: Dynamic Power Management Based on Continuous-Time Markov Decision Processes, *Design Automation Conference*, ACM, pp.555–561 (1999).
31) Rabhi, F. and Lapalme, G.: *Algorithms: A Functional Programming Approach*, Addison-Wesley (1999).
32) Shin, D. and Kim, J.: Optimizing Intra-Task Voltage Scheduling Using Data Flow Analysis, *Asia and South Pacific Design Automation Conference*, New York, NY, USA, pp.703–708, ACM (2005).
33) Shin, D. and Kim, J.: Optimizing Intratask Voltage Scheduling Using Profile and Data-Flow Information, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.26, No.2, pp.369–385 (2007).
34) Shin, D., Kim, J. and Lee, S.: Intra-Task Voltage Scheduling for Low-Energy, Hard Real-Time Applications, *IEEE Design and Test of Computers*, Vol.18, No.2, pp.20–30 (2001).
35) Sinha, A. and Chandrakasan, A.P.: JouleTrack: A Web Based Tool for Software Energy Profiling, *Design Automation Conference*, pp.220–225, ACM Press (2001).
36) Sittampalam, G. and de Moor, O.: *MAG version 2.11: User Manual*, University of Oxford (2003).
37) The SimpleScalar-Arm Power Modeling Project. http://www.eecs.umich.edu/~panalyzer/
38) Tiwari, V., Malik, S. and Wolfe, A.: Power Analysis of Embedded Software: A First Step towards Software Power Minimization, *International Conference on Computer-Aided Design*, pp.384–390 (1994).
39) Transmeta Corp. http://www.transmeta.com/

© 2009 Information Processing Society of Japan

40) Trinder, P.W., Hammond, K., Loidl, H.-W. and Jones, S.L.P.: Algorithms + Strategy = Parallelism, *Journal of Functional Programming*, Vol.8, No.1, pp.23–60 (1998).

(平成 20 年 9 月 28 日受付)
(平成 20 年 12 月 22 日採録)

横山　哲郎

2006 年東京大学大学院情報理工学系研究科博士課程修了．同年日本学術振興会特別研究員．2007 年より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．エネルギー最適化，可逆計算，プログラミング言語等の研究に従事．博士（情報理工学）．日本ソフトウェア科学会，ACM 各会員．

今井　敬吾

2004 年名古屋大学工学部物理工学科卒業．2006 年同大学大学院情報科学研究科情報システム学専攻博士前期課程修了．現在，同大学院情報科学研究科情報システム学専攻博士後期課程在学中．プロセス計算とその型システム，プログラミング言語の基礎理論，関数型言語を用いたシステム開発に興味を持つ．

曾　　剛

2006 年千葉大学大学院自然科学研究科博士課程修了．同年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研究員，2008 年より特任助教．組込みシステム設計，エネルギー最適化等の研究に従事．博士（工学）．IEEE 会員．

冨山　宏之（正会員）

1999 年 3 月九州大学大学院システム情報科学研究科博士後期課程修了．同年米国カリフォルニア大学アーバイン校客員研究員．2001 年（財）九州システム情報技術研究所研究員．2003 年名古屋大学大学院情報科学研究科講師．現在，同准教授．SOC や組込みシステムの設計技術に関する研究に従事．情報処理学会 TSLDM 編集幹事，ACM TODAES 編集委員．電子情報通信学会，ACM，IEEE 各会員．博士（工学）．

高田　広章（正会員）

名古屋大学大学院情報科学研究科情報システム学専攻教授．1988 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同専攻助手，豊橋技術科学大学情報工学系助教授等を経て，2003 年より現職．2006 年より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長を兼務．リアルタイム OS，リアルタイムスケジューリング理論，組込みシステム開発技術等の研究に従事．オープンソースの ITRON 仕様 OS 等を開発する TOPPERS プロジェクトを主宰．博士（理学）．IEEE，ACM，電子情報通信学会，日本ソフトウェア科学会各会員．

結縁　祥治（正会員）

1990 年名古屋大学大学院博士課程満了．名古屋大学大学院工学研究科助手，1998 年同情報メディア教育センター助教授，同大学院情報科学研究科准教授を経て，現在，同大学院同研究科教授．博士（工学）．

© 2009 Information Processing Society of Japan