システムＬＳＩ設計技術　107－5
(2002. 11. 28)

符号化とコンパイラ最適化技術によるアドレスバスの低消費エネルギー化

冨山宏之

財団法人九州システム情報技術研究所
〒814-0001 福岡市早良区百道浜 2-1-22 福岡 SRP センタービル７階
E-mail: tomiyama@isit.or.jp

あらまし: プロセッサと命令メモリ間のアドレスバスの消費エネルギーを低減する手法を提案する．現在まで，低消費エネルギー化を目的とした様々なアドレスバスの符号化技法が提案されている．本稿では，プログラムに対していくつかの最適化処理を施すことにより，バスの符号化によるエネルギーの低減効果が飛躍的に向上することを示す．
キーワード: バス符号化，コンパイラ最適化，低消費エネルギー，組込みシステム

Address Bus Energy Optimization via Encoding and Compiler Techniques

Hiroyuki TOMIYAMA

Institute of Systems & Information Technologies/KYUSHU
2-1-22-707 Momochihama, Sawara-ku, Fukuoka 814-0001, Japan
E-mail: tomiyama@isit.or.jp

Abstract: Off-chip or long on-chip buses are major sources of energy consumption in systems-on-chip (SOCs). In the last decade, several techniques to encode address buses for low energy have been proposed. This paper demonstrates that energy of such encoded address buses are further minimized by compiler techniques.
Keyword: Bus Encoding, Compiler Optimization, Low Energy, Embedded Systems

1. はじめに

バッテリー駆動式の組込みシステムを設計する際，消費エネルギーは最も重要な設計制約の一つである．チップ間の，あるいは，チップ上の長いバスは負荷容量が大きいため，多大なエネルギーを消費する．バスのエネルギーを低減するためにはバス上の信号遷移数を削減することが有効であり，それを目的とした様々なバスの符号化技法がこれまでに提案されている．

バス反転符号化（Bus-Invert Coding）[1]は最も代表的な符号化技法である．バス反転符号化は，そのバスを流れる時間的に連続したデータ間の相関が弱い場合に特に適しているが，相関が強いバスに対しては効果が低い．この問題を解決するために，いくつかの改良が考案されている[2]．プロセッサベースのシステムの場合，命令メモリに対するアクセスには特定の規則性がある．即ち，多くのアクセスはある一定のストライドを持った逐次的なアクセスである．現在まで提案されているアドレスバスの符号化技法のほとんどは，この規則性を利用している．文献[3]では，プロセッサのアドレスバスにグレイ符号化（Gray Coding）を使用することを提案している．グレイ符号化を用いると，逐次アクセスに対しては１ビットの信号遷移しか発生しない．Benini らは T0（Transition Zero）と呼ばれる冗長符号化技法を提案している[4][5]．T0 符号化は逐次アクセスに対しては信号遷移が生じないという特徴を持つ．その後，T0-Xor 符号化[6]や T0-C 符号化[7]など，T0 をベースとした様々な符号化技法が提案されている．これらの T0 ベースの符号化技法はプロセッサと命令メモリ間のアドレスバスに対して特に有効である．一方，文献[8]の Working-Zone 符号化や文献[9]の Beach 符号化は，命令メモリのアドレスバスだけでなく，データメモリのアドレスバスに対しても有効である．文献[10]では，アドレスバスとデータバスの両方を対象とした，符号化を行うためのフレームワークを提案している．

本稿では，プロセッサと命令メモリ間のアドレスバスの低消費エネルギー化を目的としたコンパイラ最適化手法を提案する．アドレスバスに対しては，T0 符号化等の低消費電力化のための符号化技法が適用されていることを想定する．本研究の主な貢献は次の３点である．(1)いくつかの既存のコンパイラ最適化手法により，符号化されたアドレスバスの消費エネルギーが更に削減されることを示す．(2)アドレスバスの消費電力

を低減する新しいループ展開手法を提案する．(3)これらの個々のコンパイラ最適化手法を体系的に適用するコンパイルフローを提案する．

本研究で用いる個々のコンパイラ最適化手法のほとんどは既に実用化されているものであり，また，バスの符号化技法も全て既発表のものである．しかし，筆者の知る限り，両者を組み合わせて更なるアドレスバスの低消費エネルギー化を図った研究の発表は過去に無く，本稿が初めてである．

まず第2章で代表的なアドレスバスの符号化技法について説明する．第3章で符号化されたアドレスバスの消費エネルギーを低減するコンパイラ最適化手法を提案する．第4章で実験結果を報告し，第5章でまとめと今後の課題を述べる．

2. アドレスバスの符号化技術

プロセッサから命令メモリへのアクセスには高い規則性がある．即ち，制御命令（分岐命令）が実行されない限り，次にアクセスする命令のアドレスは，現在アクセスした命令のアドレスに一定値を加えた値となる．この時，命令アクセスが逐次的であると言い，アドレスの増加分をストライドと呼ぶ．現在提案されている命令メモリのアドレスバス（以下，単に命令アドレスバスと呼ぶ）の符号化技法のほとんどは，この規則性を利用することにより，バスの信号遷移数を削減している．本章では，3つの代表的な符号化技法T0，T0-C，ならびに，T0-Xor符号化について，その動作原理を簡単に説明する．

本章では次の表記を用いる．

- b(t)：時刻tに送るべき元々のアドレス．
- B(t)：時刻tに実際に送られた符号化後のアドレス．
- S：逐次アクセスの際のストライド値．

T0（Transition Zero）符号化はBeniniらにより提案された符号化技法である[4][5]．T0符号はINCという1ビットの冗長な信号線を持つ．アクセスが逐次的な場合，INCは1に設定され，その他の信号線の値は変化しない．一方，アクセスが逐次的でない場合，INCは0となり，アドレスがそのままバスを介して転送される．まとめると，T0符号器は次のように動作する．

```
If (b(t) == b(t-1) + S) {
        B(t) = B(t-1)
        INC = 1
} else {
        B(t) = b(t)
        INC = 0
}
```

T0-C（T0-Concise）符号化はAghaghiriらによるT0符号化の改良である[7]．T0-C符号化はT0符号化と異なり，冗長な信号線を持たない．T0-C符号器は次のように動作する．

```
If (b(t) == b(t-1) + S) {
        B(t) = B(t-1)
} else if (B(t-1) != b(t)) {
        B(t) = b(t)
} else {
        B(t) = b(t-1) + S
}
```

現在送るべきアドレスの値b(t)が，直前のバスの値B(t-1)と等しい場合の符号化処理が，T0-CとT0とでは異なっている．この場合，もしT0符号化を用いると，逐次アクセスでないにも関わらず，偶然B(t)がB(t-1)が一致する．よって，INCビットが無ければ，T0復号器は時刻tのアクセスが逐次的なのか，逐次的でないのかを判別することが出来ない．一方，T0-C符号化の場合，逐次アクセスでなければ必ず信号遷移が発生するため，両者の判別が可能である．偶然B(t-1)とb(t)が一致することは稀である為，冗長ビットが存在しないT0-C符号化の方がT0よりもエネルギー削減効果が大きい．

文献[6]ではFornaciariらがT0-Xorという非冗長符号化を提案している．T0-Xor符号器は次のように動作する．

$$B(t) = b(t) \oplus (b(t-1) + S) \oplus B(t-1)$$

上記の式において，記号⊕は排他的論理和演算を表す．T0-Xor符号も逐次アクセスに対しては信号遷移が生じない．

文献[7]の実験結果によると，T0，T0-C，および，T0-Xor符号化による信号遷移数の削減率は，それぞれ62.0%，73.1%，および，75.0%である．これら3種類以外にも同程度の削減率を有する符号化技法がいくつか提案されている．本稿で説明するコンパイラ技術は，アクセスの逐次性を活用した符号化技法であれば，先述の3種類以外の技法で符号化されたアドレスバスに対しても有効である．

3. コンパイラ最適化技術

前章で説明したアドレスバスの符号化技法は全て，命令アクセスの逐次性を活用している．このことは，逐次的に実行されるプログラムの方が，分岐を多く含んだプログラムよりもアドレスバスの消費エネルギー

```
for (i=0; i<N; i++) {
    A[i] = B[i] * C[i];
}
```

(a) Original loop

```
for (i=0; i<N; i=i+2) {
    A[i] = B[i] * C[i];
    A[i+1] = B[i+1] * C[i+1];
}
```

(b) Unrolled loop

図 1　ループ展開の例

```
do {
    body;
} while (x < y);
```

(a) Original loop

```
do {
    body;
    if (! (x < y)) break;
    body;
} while (x < y);
```

(b) Unrolled loop

図 2　性能は向上しないがエネルギーが削減されるループ展開の例

が小さくなることを意味する．つまり，消費エネルギーを低減するためには，極力分岐が少なくなるように，コンパイラがプログラムを変形すればよい．ここで注意すべきことは，プログラムテキスト中の分岐命令の出現数ではなく，プログラム実行時における分岐（無条件分岐および成立した条件分岐）の実行回数を最小化することである．よって，消費エネルギーの削減効果を高めるためには，コンパイラが最適化を行う前に典型的な入力データ（の集合）を用いてプログラムを実行し，その実行履歴を用いてコンパイラ最適化を適用すると良い．

本章ではアドレスバスの遷移数を削減するコンパイラ技術を 3 つ示し，これらを体系的に適用するコンパイルフローを提案する．

3.1. ループ展開

ループ展開（Loop Unrolling）は広く用いられているコンパイラ最適化技術の一つである．ループ展開はループのボディのコピーを複数作成することにより，繰返しの回数を削減する．ループ展開において，ループボディのコピーの数をアンロール係数（unroll factor）と呼ぶ．アンロール係数が 2 であるループ展開の例を図 1 に示す．ループを展開すると，条件判定やループインデックスの更新に要する命令の実行回数が削減され，プログラムの実行時間が短縮される．また，プロセッサが同時に複数の命令を発行できる場合には，プログラムの命令レベル並列性が向上されることによる実行時間の短縮も期待される．しかし，ループ展開にはコードサイズが増大するという問題がある．よって，展開すべきループ，および，ループ係数は慎重に決定されなければならない．一般的に，ループボディが小さく，ループ回数が多いループを展開すると良い．

ループ展開を行うと分岐の実行回数が削減され，その結果，アドレスバスの遷移数が削減される．図 1 の例題の場合，ループ展開を行う前の分岐の実行回数は N+1 回であるが[1]，ループ展開を行った後では N/2+1 回となる．

ループ展開を行うと，多くの場合，実行命令数が削減され（よって性能が向上し），同時に，アドレスバスのエネルギーも低減される．しかし，バスのエネルギーは低減されるが，実行命令数は変化しないループ展開も存在する．そのようなループ展開の例を図 2 に示す．図の例題において，ループ展開の前後で実行命令数は変化しない．よって，性能の向上あるいはコードサイズの縮小を追求する通常のコンパイラはこのようなループ展開は行わない．しかし，ループ回数が多い場合，図 2(b)に示すループ展開により分岐命令の成立回数はほぼ半減する．よって，アドレスバスのエネルギーが低減される．アドレスバスの低消費エネルギー化を実現するためには，このようなループ展開も行う価値がある．

3.2. 関数のインライン展開

プログラム中の関数呼出しを関数のボディ（本体）で置き換える処理を関数のインライン展開と呼び，既に多くのコンパイラが実現している最適化技術である．関数のインライン展開には様々な利点がある．例えば，関数呼出し命令が不要になるだけでなく，レジスタやプログラムカウンタの退避と復帰に要する命令や，スタックフレームを操作する命令が不要になる．更には，定数伝播や共通部分式の削除などの最適化をより広域的に適用することが可能となるという利点もある．

関数のインライン展開を行うと，関数呼出しのための分岐命令が不要になるため，命令アクセスが逐次化される．よって，関数のインライン展開は性能の向上だけでなく，アドレスバスの低消費エネルギー化に寄与する．

しかし，ループ展開と同様，コードサイズが増加する恐れがあるため，インラインすべき関数を決定する際には，コードサイズの増加と期待される利点とのトレードオフを考慮しながら決定しなければならない[12][13]．コードサイズの増加を微小に抑えるためには，プログラムコード中の 1 箇所からのみ呼び出される関

[1] ループボディの末尾から先頭に戻るために N 回，および，ループの条件判定が偽になり，ループ文の次の文に制御を移すために 1 回（合計 N+1）．ただし，ループ構造を最適化することにより，分岐の実行は N 回に削減される．

```
int F(int x) {
  static int y = 0;

  if (x > 0)
    y = y + x;
  else
    y = y – x;
  return y;
}
```

(a) Original *if* statement

```
int F(int x) {
  static int y = 0;

  if (x > 0) goto L2;
  y = y – x;
L1:
  return y;
L2:
  y = y + x;
  goto L1;
}
```

(b) *if* statement after trace selection

図3　トレース選択の例

数や，ボディが小さな関数をインライン展開すれば良い．また，当然ながら，呼び出される頻度が少ない関数よりも，頻度が多い関数をインライン展開すべきである．

3.3. トレース選択

トレース選択（Trace Selection）は，命令キャッシュのミス率を低減するために考案されたコンパイラ最適化技術である．トレース選択は文献[14]で提案され，文献[15]で組み合わせ最適化問題として定式化されている．図3にトレース選択の例を示す．図3(a)に示す条件文（if 文）において，もし条件式が偽になる可能性が高いならば，図3(b)のように制御構造を変形する．その結果，当該条件文の前の文，条件判定式，else 節，条件文の後の文が逐次的に実行されることになり[2]，命令アクセスの空間的局所性が高まり，よって，キャッシュミスの削減が期待される．

トレース選択は命令アクセスを逐次化するため，アドレスバスの低消費エネルギー化にも寄与する．図3(a)の条件文では，条件式が真の場合でも偽の場合でも，分岐が1回成立する（真の場合は then 節の最後に1回，偽の場合は条件判定の直後に1回．ただし，関数の return 文に伴う分岐は数えていない）．一方，トレース選択後の図3(b)では，条件式が真の場合には分岐が2回成立するが，条件式が偽の場合には分岐は実行されない．よって，条件式が偽となる可能性が高い場合には，トレース選択によりアドレスバスの遷移数が削減される．トレース選択を効果的に行うためには，典型的な入力データを用いてプログラムを実行し，その実行履歴を使用すると良い．

3.4. 全体のコンパイルフロー

本章のここまでは，アドレスバスの信号遷移数を削減する個々のコンパイラ最適化技術について説明した．しかし，これらの個々の最適化を無秩序に適用しても，アドレスバスの遷移数はあまり削減されずに，コードサイズだけが増大する恐れがある．ここでは，これらのコンパイラ最適化技術を効率よく体系的に適用するフローを提案する．

提案するコンパイルフローは，コンパイルすべきプログラム，そのプログラムに対する入力データ（の集合），および，コードサイズ制約が与えられることを仮定する．そして，コードサイズ制約の下で，分岐が成立する回数を最小化することを目的とする．提案するコンパイルフローは，大きく5つのステップから構成される．以下，各ステップを説明する．

ステップ **1**：プログラムを静的に解析し，プログラムコード中の1箇所からのみ呼び出される関数をインライン展開する．これによって，コードサイズは増加しない．

ステップ **2**：プログラムをコンパイル，実行し，その実行履歴を採取する．ここで，実行履歴はプログラムの各文の実行頻度に関する情報を含む．

ステップ **3**：ループの展開と関数のインライン展開を行う．展開すべきループや関数を選択する際には，展開することによって得られる消費エネルギーの削減とコードサイズの増加との間のトレードオフを定量的に表す評価指標を用いるべきである．そのような指標の一例を以下に示す．

$$\text{ゲイン} = \frac{\text{削減される分岐成立の数}}{\text{増加するコードサイズ}}$$

プログラム中の各関数と各ループについて，上の式で定義されるゲイン値を計算し，最も大きなゲイン値を持つ関数またはループを展開する．コードサイズ制約が違反しない限り，ゲイン値の計算と，関数／ループの展開を繰り返す．

ステップ **4**：再度プログラムをコンパイル，実行し，その実行履歴を採取する．

ステップ **5**：トレース選択を行う．

4. 実験

第3章で提案したコンパイル手法を評価するための環境を，SimpleScalar ツールキット（GNU C コンパイラとシミュレータ）[17]，SUIF コンパイラシステム[18]，GNU プロファイラ（gprof），および，GNU ソースコードカバレッジアナライザ（gcov）を用いて構築した．ただし，一部のコンパイラ最適化（トレース選択やル

[2] 逐次的に実行されるであろう一連の命令（基本ブロック）をトレース（Trace）と呼ぶ．

表 1　アドレスバスの遷移数と削減率

	符号化無し		T0符号化		T0-C符号化		T0-Xor符号化	
	コンパイラ最適化無し	コンパイラ最適化有り	コンパイラ最適化無し	コンパイラ最適化有り	コンパイラ最適化無し	コンパイラ最適化有り	コンパイラ最適化無し	コンパイラ最適化有り
CRC	56,476	43,748	24,338	14,774	14,525	9,557	9,521	6,959
	0.00%	22.54%	56.91%	73.84%	74.28%	83.08%	83.14%	87.68%
FIR	41,664	25,324	11,078	5,216	8,281	3,809	9,483	3,684
	0.00%	39.22%	73.41%	87.48%	80.12%	90.86%	77.24%	91.16%
LMS	514,419	386,660	122,123	49,366	111,878	36,567	123,070	34,018
	0.00%	24.84%	76.26%	90.40%	78.25%	92.89%	76.08%	93.39%
ADPCM	7,394,262	6,693,573	1,782,792	889,517	1,809,531	648,020	1,857,031	546,160
	0.00%	9.48%	75.89%	87.97%	75.53%	91.24%	74.89%	92.61%
Average	0.00%	24.02%	70.62%	84.92%	77.05%	89.52%	77.84%	91.21%

表 2　実行命令数と逐次アクセス数

	実行命令数（総アクセス数）		逐次アクセス数	
	コンパイラ最適化無し	コンパイラ最適化有り	コンパイラ最適化無し	コンパイラ最適化有り
CRC	27,512	21,330	22,130	18,252
	0.00%	22.47%	80.44%	85.57%
FIR	18,793	12,504	16,292	11,352
	0.00%	33.46%	86.69%	90.79%
LMS	225,247	187,236	193,652	177,613
	0.00%	16.88%	85.97%	94.86%
ADPCM	3,746,555	3,224,187	3,230,276	3,071,103
	0.00%	13.94%	86.22%	95.25%
Average	0.00%	21.69%	84.83%	91.62%

ープ回数が静的に解析できないループの展開）は手作業で行った．ベンチマークプログラムは SNU Real-Time Benchmark Suites [19]から 4 つ選択した．各ベンチマークプログラムをコンパイル，実行することにより命令アクセスのアドレスのトレースを採取し，アドレスバスの遷移数を測定した．その際，それぞれのベンチマークプログラムついて，アドレスバスの符号化技法 4 種類（符号化しなかった場合，および，T0，T0-C，T0-Xor で符号化した場合）と本稿で提案したコンパイラ最適化の有無の 2 通りを組み合わせ，合計 8 通りの場合について，アドレスバスの信号遷移数を測定した．

表 1 に実験結果をまとめる．表の 1 列目にベンチマークプログラムの名前を記している．2 列目の上段の値が，アドレスバスを符号化せず，かつ，本稿で提案したコンパイラ最適化も行わなかった場合の遷移数を表している．ただし，この場合でも通常のコンパイラ最適化は行っている（GCC の-O2 オプションを使用）．3 列目以降では，残りの 7 通りの組み合わせについて，アドレスバスの遷移数とその削減率を示している．削減率の基準は 2 列目の値，即ち，符号化もコンパイラ最適化も行わなかった場合の遷移数である．表の最終行に削減率の平均を示した．

アドレスバスの消費エネルギーを考慮したコンパイラ最適化を行わない場合，T0，T0-C，および，T0-Xor 符号化によるアドレスバスの遷移数の削減率は，それぞれ，71%，77%，および，78%である．一方，アドレスバスの消費エネルギーを考慮したコンパイラ最適化を適用した場合，アドレスバスの遷移数の削減率はそれぞれ 85%，90%，および，91%であり，削減率が大幅に改善された．同じアドレスバスの符号化技法を用いた場合で比較すると，多くの場合で，コンパイラ最適化を適用することによりアドレスバスの遷移数が 50%以下に削減されていることが分かる．

コンパイラ最適化によりアドレスバスの遷移数が大幅に削減された理由は大きく 2 つ存在する．理由の 1 つは，コンパイラ最適化により，命令実行数，即ち，命令アクセスの数自体が削減されたことである．第3章で説明したように，個々のコンパイラ最適化技術は元々実行命令数を削減することを目的として開発されたものである．表 2 の 2 列目と 3 列目に，コンパイラ最適化を行った場合と行わなかった場合について，命令アクセスの数を示す．コンパイラ最適化により，平均 22%命令アクセスが削減された．信号遷移数削減のもう 1 つの理由が，コンパイラ最適化により命令アクセスが逐次化されたことである．表 2 の 3 列目と 4 列

表 3　コードサイズ（単位はバイト）

	コンパイラ最適化無し	コンパイラ最適化有り
CRC	24,608	24,704
	0.00%	0.39%
FIR	25,600	26,016
	0.00%	1.63%
LMS	26,128	27,936
	0.00%	6.92%
ADPCM	31,008	34,864
	0.00%	12.44%
Average	0.00%	5.34%

目に，コンパイラ最適化を行った場合と行わなかった場合について，逐次的な命令アクセスの回数，ならびに，総命令アクセスに対する逐次アクセスの割合を示す．コンパイラ最適化を行わない場合，逐次アクセスの割合は約 85%であったのに対し，コンパイラ最適化を行うことにより約 92%に改善された．

一方，コンパイラ最適化によりコードサイズは増大する．本実験においては，表 3 に示す通り，コードサイズの増大は平均 5%であった．

5. おわりに

プロセッサベースの組込みシステムにおいて，プロセッサとメモリの間のバスは多大なエネルギーを消費する．本稿では，プロセッサと命令メモリ間のアドレスバスの消費エネルギーを削減する手法を提案した．提案手法は，アドレスバスの符号化とコンパイラ最適化の両者を組み合わせて用いることにより，アドレスバス上の信号遷移数を削減する．実験の結果，提案手法によりアドレスバスの遷移数が最大 93%削減された．

本稿で提案した手法は，プロセッサと命令キャッシュ間のアドレスバス，あるいは，命令キャッシュが存在しない場合のアドレスバスに対して特に有効である．今後は，命令キャッシュとメモリ間のアドレスバスに対する提案手法の有効性を評価したい．また，プロセッサとデータメモリ間のアドレスバスの低消費エネルギー化を実現するコンパイラ手法についても研究を行う予定である．

文　献

[1] M. R. Stan and W. P. Burleson, "Bus-invert coding for low power I/O," *IEEE Trans. VLSI Systems*, vol. 3, no. 1, 1995.

[2] Y. Shin, S.-I. Chae, and K. Choi, "Partial bus-invert coding for power optimization of application-specific systems," *IEEE Trans. VLSI Systems*, vol. 9, no. 2, 2001.

[3] C. L. Su, C. Y. Tsui, and A. M. Despain, "Saving power in the control path of embedded processors," *IEEE Design & Test of Computers*, vol. 11, 1994.

[4] L. Benini, G. De Micheli, E. Macii, D. Sciuto, and C. Silvano, "Asymptotic zero-transition activity encoding for address buses in low-power microprocessor-based systems," In *Proc. of GLS-VLSI*, 1997.

[5] L. Benini, G. De Micheli, E. Macii, D. Sciuto, and C. Silvano, "Address bus encoding techniques for system-level power optimization," In *Proc. of DATE*, 1998.

[6] W. Fornaciari, M. Polentarutti, D. Sciuto, and C. Silvano, "Power optimization of system level buses based on software profiling," In *Proc. of CODES*, 2000.

[7] Y. Aghaghiri, F. Fallah, and M. Pedram, "Irredundant address bus encoding for low power," In *Proc. of ISLPED*, 2001.

[8] E. Musoll, T. Lang, J. Cortadella, "Working-zone encoding for reducing the energy in microprocessor address buses," *IEEE Trans. VLSI Systems*, vol. 6, no. 4, 1998.

[9] L. Benini, G. De Micheli, E. Macii, M. Poncino, and S. Quer, "Power optimization of core-based systems by address bus encoding," *IEEE Trans. VLSI Systems*, vol. 6, no. 4, 1998.

[10] S. Ramprasad, N. R. Shanbhag, and I. N. Hajj, "A coding framework for low-power address and data busses," *IEEE Trans. VLSI Systems*, vol. 7, no. 2, 1999.

[11] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addition-Wesley, 1986.

[12] R. Leupers and P. Marwedel, "Function inlining under code size constraints for embedded processors," In *Proc. of ICCAD*, 1999.

[13] M. Palkovic, M. Miranda, and F. Catthoor, "Systematic power-performance trade-off in MPEG-4 by means of selective function inlining steered by address optimisation opportunities," In *Proc. of DATE*, 2002.

[14] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," In *Proc. of ISCA*, 1989.

[15] H. Tomiyama and H. Yasuura, "Code placement techniques for cache miss rate reduction," *ACM TODAES*, vol. 2, no. 4, 1997.

[16] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Computers*, vol. C-30, no. 7, 1981.

[17] SimpleScalar Tools, available at http://www.cs.wisc.edu/~mscalar/simplescalar.html.

[18] SUIF Compiler System, available at http://suif.stanford.edu/.

[19] SNU Real-Time Benchmark Suites, available at http://archi.snu.ac.kr/realtime/benchmark/.