東京工業大学工学部

学士論文

FPGAで実現された計算機の高性能キャッシュを用いた高速化

指導教員 吉瀬 謙二 准教授

平成27年2月

提出者

学科　　情報工学科
学籍番号 11_05340
氏名　　小川 愛理

指導教員認定印	
学科長認定印	

平成26年度　学士論文内容梗概

FPGAで実現された計算機の高性能キャッシュを用いた高速化

指導教員　　吉瀬 謙二　准教授

情報工学科
11_05340　　小川 愛理

ムーアの法則が提唱されて以来，集積回路上のトランジスタ数は指数オーダで増加を続けてきた．集積回路の回路規模が増大したことで，従来集積できない規模の回路も1チップに実現可能になっている．この高集積化によりFPGA (Field Programmable Gate Array)の性能は飛躍的に向上しており，これまでASIC (Application Specific Integrated Circuit) が採用されてきた分野においてもFPGAが用いられるようになっている．今後，オペレーティングシステムの動くコンピュータシステムにおいてもFPGAが広く用いられるようになると考え，オペレーティングシステムの動作するFPGAで実現された計算機を開発している．

この計算機は既存のFPGAシステムに改良を加えたものである．現在はオペレーティングシステムとしてLinuxとFreeDOSが起動し，その上でさまざまなアプリケーションを動かすことができる．しかし，この計算機は近代的な汎用計算機と比較して非常に低速であり，実用的なシステムとは言いがたい．計算機の実行時間を解析したところ，全体の実行時間のうち，メモリアクセス時間の占める割合が大きいことが性能を下げる一因であることがわかった．

メモリアクセス時間を低減する方法として，キャッシュメモリの利用が挙げられる．一般的に知られているダイレクトマップ方式やセットアソシアティブ方式のキャッシュ構成に加えて，簡単な構成で実現できる高性能キャッシュ方式として，skewedキャッシュが提案されている．このキャッシュは，way毎に異なるハッシュ関数をインデックスに使用することで，書き換え時のデータの衝突を軽減することができる．そこで，より実用的なシステムにするために，すでに搭載されているL1キャッシュに加え，FPGAで実現された計算機に高性能なskewedキャッシュをL2キャッシュとして実装し，高速化を図った．

前述のFPGAで実現された計算機にskewedキャッシュを含む様々な方式のキャッシュを実装し，キャッシュのハードウェア使用量の評価を行った．また，計算機上でLinuxを動作させ，アプリケーションの実行時間とキャッシュミス率の評価を行った．その結果，オペレーティングシステムの動く計算機におけるskewedキャッシュの有用性が明らかになった．また，開発している計算機の大幅な速度向上を達成した．

目　次

第1章

緒論

1.1 研究の背景と目的

ムーアの法則が提唱されて以来，集積回路上のトランジスタ数は指数オーダで増加を続けてきた．集積回路の回路規模が増大したことで，従来集積できない規模の回路も1チップに実現可能になっている．この高集積化によりFPGA (Field Programmable Gate Array) の性能は飛躍的に向上しており，これまでASIC (Application Specific Integrated Circuit) が採用されてきた分野においてもFPGAが用いられるようになっている．今後，オペレーティングシステムの動くコンピュータシステムにおいてもFPGAが広く用いられるようになると考え，オペレーティングシステムの動作するFPGAで実現された計算機を開発している．

この計算機は既存のFPGAシステムに改良を加えたものである．現在はオペレーティングシステムとしてLinuxとFreeDOSが起動し，その上でさまざまなアプリケーションを動かすことができる．しかし，この計算機は近代的な汎用計算機と比較して非常に低速であり，実用的なシステムとは言いがたい．計算機の実行時間を解析したところ，全体の実行時間のうち，メモリアクセス時間の占める割合が大きいことが性能を下げる一因であることがわかった．

メモリアクセス時間を低減する方法として，キャッシュメモリの利用が挙げられる．一般的に知られているダイレクトマップ方式やセットアソシアティブ方式のキャッシュ構成に加えて，簡単な構成で実現できる高性能キャッシュ方式として，skewedキャッシュが提案されている．このキャッシュは，way毎に異なるハッシュ関数をインデックスに使用することで，書き換え時のデータの衝突を軽減することができる．

そこで本論文では，より実用的なシステムにするために，すでに搭載されているL1キャッシュに加え，FPGAで実現された計算機に高性能なskewedキャッシュをL2キャッシュとして実装し，高速化を図った．また，本論文ではキャッシュを実装した計算機上でLinuxを動作させて，キャッシュのハードウェア使用量の評価や，アプリケーションの実行時間とキャッシュミス率の評価を行う．

1.2 本論文の構成

本論文の構成は以下の通りである．2章では一般的なキャッシュメモリの概要と，高性能キャッシュの一例としてskewedキャッシュの構成について述べる．また，評価環境であるFPGAで実現された計算機の構成を述べる．3章では高性能キャッシュを用いた高速化手法と実装について述べ，4章では高速化した計算機の評価を行う．最後に，5章で本論文の結論をまとめる．

第2章

関連研究

2.1　プロセッサ

プロセッサは，コンピュータの中で，ソフトウェアプログラムに記述された命令セットを実行するためのハードウェアである． 演算装置，レジスタ，周辺回路などから構成される．

2.1.1　プロセッサの発展

プロセッサが開発される前までは，用途ごとに異なるICを製作する必要があった．プロセッサを使うことで，用途に合わせたプログラムを用意すればよく，様々な処理に柔軟に対応できるようになった．Intel社が世界初のプロセッサであるi4004[1]を発売してから，プロセッサの性能は急速に発展を遂げた．表2.1に，Intel社のマイクロプロセッサの系譜[2]を示す．

回路規模の増大に伴い処理ビットやメモリ空間も大きく進歩し，近年ではプロセッサの中核機能であるプロセッサコアを複数搭載したマルチコアとよばれるプロセッサも登場している．例えば， Gulftownのコードネームで呼ばれたCore i7-980Xは6コアである．

表2.1: Intel社のマイクロプロセッサの系譜

年代	1971	1974	1978	1985	2000	2010
型番	4004	8080	8086	80386	Pentium 4	core i7-980X (6コア)
処理ビット	4	8	16	32	32	64
素子数(個)	2300	8500	3万	28万	4200万	11億7000万
クロック	100kHz	1MHz	10MHz	20MHz	1GHz	3GHz
アドレス空間	4.5K	64K	16M	4G	4G	24G

2.1.2　プロセッサとメインメモリ

プロセッサの性能向上に伴ってメインメモリへの要求の頻度は増えていくため，メモリの性能は重要になっていく．しかし，プロセッサの性能は飛躍的に述びているのに対し，メモリの性能向上はやや劣る．前述の Core i7 のような高性能プロセッサは，コア当たり 1 クロックにつきデータメモリ参照を 2 つ出すことができる上にマルチコアであるため，メインメモリのアクセスレイテンシの改善はより重要な課題である．これに対処するため，プロセッサの内部にはキャッシュが搭載されている．

2.2　キャッシュメモリ

キャッシュメモリとは，プロセッサとメインメモリとの間に挿入される高速なメモリのことである． 参照の局所性を利用することでプロセッサのアクセス頻度の高いデータや命令を保存し，メインメモリの代わりに入出力を行う．プロセッサはデータや命令を読み出す時に，よりプロセッサに近い場所にあるキャッシュから読み出すことによりメモリアクセスの時間を大幅に短縮することが可能である．最近のプロセッサとメモリの性能差の拡大，マルチスレッドなどアクセス範囲の拡大に対応するため，キャッシュも多段構造とする例が増えている．この場合プロセッサに近い側から L1 (レベル 1) キャッシュ，L2 (レベル 2) キャッシュと呼ばれる [3].

Core i7-980X の場合には，L1 命令キャッシュと L1 データキャッシュがそれぞれ 32KB，L2 キャッシュが 256KB で 6 つのコアのひとつずつに搭載されている．また，6 つのコアに共通の 12MB の L3 キャッシュも内蔵する．これら大容量のキャッシュがプロセッサの内部に搭載されるようになったことは，メインメモリとの性能の差をカバーし，プロセッサの性能向上に貢献してきた．

2.2.1　キャッシュメモリの動作

キャッシュとは，プロセッサとメインメモリとの間でメインメモリから読み出した内容を一時的に保持するメモリのことで，メインメモリより小容量だがはるかに高速である． 同じデータへの 2 回目以降のアクセス時に，メインメモリから読み出す代わりにキャッシュから読み出すことよって，ヒット時にはアクセス時間を大幅に高速化できる．

キャッシュはデータをブロックと呼ばれるある程度まとまった単位で管理する．あるアドレスのデータにアクセスする際に，キャッシュのどのブロックにデータが存在するかを選ぶのがキャッシュのインデックスである．また，該当ブロックのデータが，要求されたデータに対応するものかどうかを判別するた

めの情報がキャッシュのタグである．キャッシュはブロックの総数と等しい数だけのエントリを持ち，インデックスによりどのエントリのブロックにキャッシュを格納するかを選択する．通常は，データのアドレスからバイトオフセットを除いた下位ビットをキャッシュのインデックスに使用し，残りの上位ビットをタグに使用する．各ブロックは，ブロック内のデータが有効であるかどうかを示す valid ビットを持つ．キャッシュに新しくデータが入力されると，データが格納されたブロックの valid ビットを 1 にする．

図 2.1 に最も簡単なキャッシュの構成である，ブロックサイズが 1 ワードのダイレクトマップ方式のキャッシュの構成を示す．キャッシュは，要求があったデータのアドレスからキャッシュのインデックスとタグを取り出し，インデックスに対応するデータが要求されたデータかどうかをキャッシュ内に保存されているタグと比較することで判断する． タグが一致した場合にはキャッシュヒットとなり，ヒットしたデータへのアクセスを行う．タグが一致しなかった場合にはキャッシュミスとなり，メインメモリにアクセスする必要が生じる．これには多くのサイクル数を要するため，キャッシュミスの割合を減らすことはキャッシュの性能に大きく貢献する．

2.2.2 キャッシュの書き込みの取り扱い

ストア命令を実行する際には，メモリへの書き込みが発生する． このとき，キャッシュのヒットやミスにかかわらず，キャッシュに新しいデータが保存されることになる． しかし，キャッシュからデータが取り除かれた後にも書き込み後の正しいデータがどこかに存在している必要があるため，メインメモリにデータを書き込む必要が生じる． メインメモリへのデータの書き込みのタイミングには，主に以下の 2 つの方式がある．

ライトスルー方式
: キャッシュにデータを書き込むのと同じタイミングでメインメモリにも書き込むという方式． キャッシュとメインメモリとの間のデータの一貫性が常に保たれるため，簡単な構成で実現できる． しかし，書き込みの際には常にメインメモリにもアクセスするため，性能はあまり高くない．

ライトバック方式
: データを書き込む際にはキャッシュのみを更新して，ブロックの置き換えが生じたときに変更されたデータをメインメモリに書き戻すという方式．キャッシュのデータがメインメモリから変更されていることを表す dirty ビットを利用して実装される． ライトスルー方式と比べると実現にはやや複雑な処理が必要になるが，書き込みの回数が少なくて済むため性能が良い．

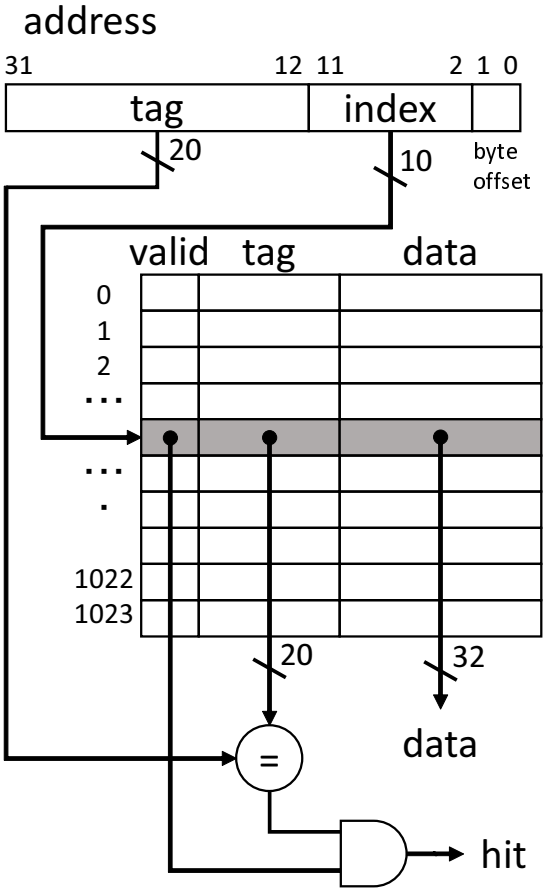

図 2.1: ダイレクトマップキャッシュの構成例

2.2.3 キャッシュの構造とブロックの置き換え方式

キャッシュにデータを格納する際，キャッシュのどのブロックにデータを格納するかを決めるのがキャッシュのインデックスである．キャッシュメモリは読み出しや書き込み時の検索速度を高めるため，アクセスされたアドレスから割り出されたインデックスにより，検索されるブロックの数を変更している． このときのアクセスするブロックの数を連想度といい，連想度によってデータ構造に違いがある．

ダイレクトマップ方式

図 2.2 にダイレクトマップ方式のデータの格納構造を示す． データをキャッシュに格納する際，格納場所が特定の 1 箇所に決まるという方式．連想度は 1 であり，同じインデックスを持つデータが 2 つ以上キャッシュに入力された場合には，キャッシュブロックのデータに置き換えが生じる．

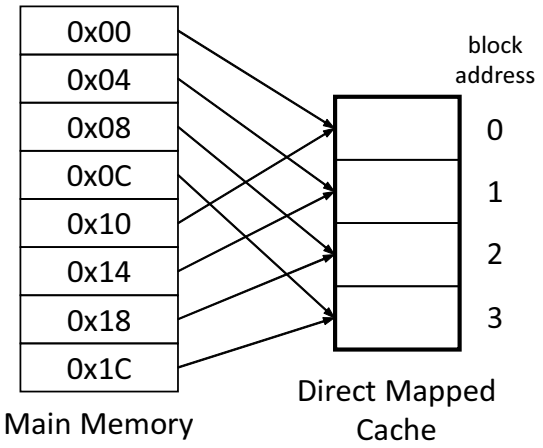

図 2.2: ダイレクトマップキャッシュ方式

フルアソシアティブ方式

図 2.3 にフルアソシアティブ方式のデータの格納構造を示す．データをキャッシュに格納する際，キャッシュ内の任意の場所に格納することができる方式．連想度はキャッシュ内のブロックの総数と等しい．この方式のキャッシュにおいて求めるブロックを検索するには，キャッシュ内の全てのエントリを検索しなくてはならないため，適用できるのはブロック数が少ないキャッシュに限定される．

セットアソシアティブ方式

図 2.4 にセットアソシアティブ方式のデータの格納構造を示す．ダイレクトマップ方式とフルアソシアティブ方式の中間的な設計で，データをキャッシュに格納する際，格納場所が連想度の数だけ存在するという方式．連想度は 2 以上の定数である．同じインデックスを持つデータが連想度の数より多くキャッシュに入力されるとブロックのデータに置き換えが生じる．連想度が n のセットアソシアティブ方式のキャッシュを n-way セットアソシアティブキャッシュと呼ぶ．

連想度が 2 以上のキャッシュにおいてキャッシュミスが発生したとき，同じインデックスを持つ全てのブロックにデータがすでに格納されていた場合には，いずれかのブロックを置き換える必要がある．どのブロックを置き換えるかを決める方法のうち，最も一般的な方法が LRU (Least Recently Used) 法である．LRU では，ブロックの使用履歴を保存することにより，使用されずにいた時間が最も長いブロックを置き換える．LRU は，連想度を上げるにつれて実現が難しくなるが，高い性能を発揮する．

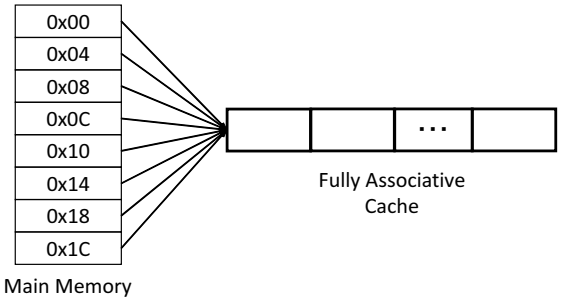

図 2.3: フルアソシアティブ方式

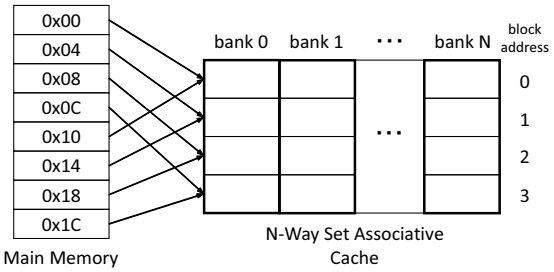

図 2.4: n-way セットアソシアティブ方式

2.3 skewed キャッシュ

skewed キャッシュ[4][5] とは，セットアソシアティブ方式のキャッシュにおいて，キャッシュのインデックスにハッシュ関数を使用することにより，メモリバンク毎に別のエントリを選ぶキャッシュのことである．これにより，ブロックの置き換え時のデータの衝突を軽減し，ヒット率を向上させることができる．

2.3.1 skewed キャッシュの概要

2-way セットアソシアティブにおける例を考える．図 2.5 に，2-way セットアソシアティブのキャッシュのインデックスの取り方の例を示す．通常は，データのアドレスからバイトオフセットを除いた下位ビットをインデックスとし，キャッシュエントリを選ぶ． 同じインデックスを持つアドレスAとアドレスBのデータがキャッシュに入力された場合，2つのデータは同じエントリに格納されることになる． つまり，2-way の場合には，1つのインデックスに対してキャッシュに格納できるデータの数は2つである．これに対して，2-way skewed アソシアティブのキャッシュのインデックスの取り方の例を示したのが図 2.6 で

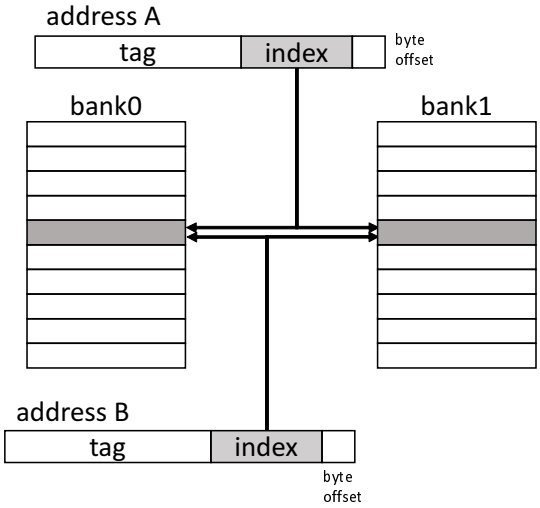

図 2.5: 2-way セットアソシアティブにおけるキャッシュエントリの取り方

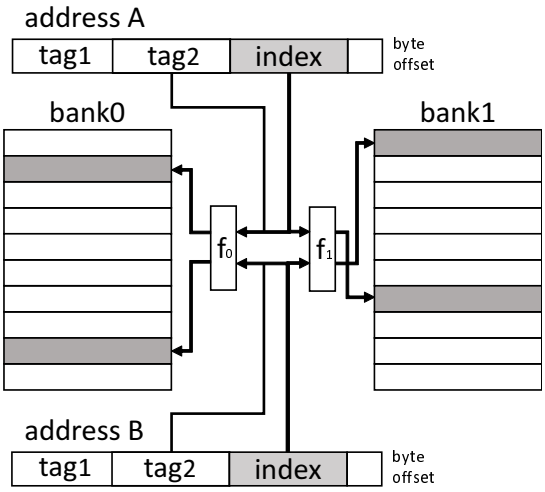

図 2.6: 2-way skewed アソシアティブにおけるキャッシュエントリの取り方

ある．skewed キャッシュでは，メモリバンク毎に異なるハッシュ関数を用いる．アドレスから生成される元のインデックスとタグの一部にハッシュ関数を使用し，得られたハッシュ値により新たなインデックスを選ぶ．そのため，アドレス A とアドレス B の下位ビットが同じであっても，ハッシュ関数を適切に選ぶことが出来れば，way 毎に全て異なるエントリに格納されることになる．つまり，2-way skewed アソシアティブでは，1 つのアドレスに対して，最大で 4 つの場所にデータを格納することができる．

2.3.2 ハッシュ関数

skewed キャッシュに用いるハッシュ関数には様々な取り方が考えられるが，主な例を取り上げる．キャッシュのエントリの数が 2^n，ブロックンサイズが 1 ワードの 4-way セットアソシアティブキャッシュにおいて，アドレス A を持つデータが入力されたと仮定する．アドレス A が，バイナリ表示において A_0 から A_3 の 4 つに分解されるとする． つまり，$A = \{A_3, A_2, A_1, A_0\}$ と表される．A_0 は 2 ビットのバイトオフセットである． A_1 と A_2 は共に n ビットとし，A_3 は残りのビットである． 通常は，A_1 をインデックス，$\{A_3, A_2\}$ をタグとして使用するが，skewed キャッシュにおいては，A_1 と A_2 を用いて新たなインデックスを生成する．関数 σ，ϕ，ϕ' を，ビット順列の任意の並べ替えとすると，キャッシュのメモリバンク 0 からメモリバンク 3 に対応する $index_0$，$index_1$，$index_2$，$index_3$ は，次式のように取ることができる．

$$
\begin{aligned}
index_0 &= \{\phi(A_1) \oplus \phi'(A_2)\} \\
index_1 &= \{\sigma(\phi(A_1)) \oplus \phi'(A_2)\} \\
index_2 &= \{\sigma^2(\phi(A_1)) \oplus \phi'(A_2)\} \\
index_3 &= \{\sigma^3(\phi(A_1)) \oplus \phi'(A_2)\}
\end{aligned}
$$

ここで，$\oplus$はビット毎の排他的論理和である．σは，任意のビット数のシフト演算であり，ϕとϕ'は，nビットの任意の並べ替えである．ϕとϕ'は全てのインデックスに共通する関数であり，場合によってはなくてもかまわない．σをnビットのシフト演算であるとすると，σ^2は，$n*2$ビットのシフト演算，σ^3は$n*3$ビットのシフト演算となる．これらの計算は比較的簡単な実装で行え，かつアドレスの上位ビットが同じ場合にも違うキャッシュエントリに格納されるため，バランスがよい設計である．

簡単な例を考える． キャッシュエントリの数が2^4，ラインサイズが1ワードの4-wayセットアソシアティブキャッシュにおいて，アドレス$A = 1010101101100$を持つデータが入力されたとする．このとき，前述の設定におけるA_0からA_3は，$A_0 = \{00\}$，$A_1 = \{1011\}$，$A_2 = \{0101\}$，$A_3 = \{101\}$と表せる．σが1ビットの左巡回シフトであるとすると，σ^2は2ビットの左巡回シフト，σ^3は3ビットの左巡回シフトとなる．ϕとϕ'はないものとすれば，$index_0$，$index_1$，$index_2$，$index_3$は次のように表せる．

$$
\begin{aligned}
index_0 &= \{A_1 \oplus A_2\} = \{1110\} \\
index_1 &= \{\sigma(A_1) \oplus A_2\} = \{0010\} \\
index_2 &= \{\sigma^2(A_1) \oplus A_2\} = \{1011\} \\
index_3 &= \{\sigma^3(A_1) \oplus A_2\} = \{1000\}
\end{aligned}
$$

この例では，ひとつのアドレスから4つの異なるインデックスを生成している．このような場合，下位ビットが同じアドレスが再び現れ時にデータの衝突を低減できる．ひとつのアドレスからそれぞれのwayに入るインデックスをなるべく異なるものにすることで，skewedキャッシュの性能はより高まる．

2.4 FPGAで実現された計算機

高速化を行う計算機[6]は，計算機で最も一般的なISAであるx86をサポートし，OSが動作するFPGAシステムである．このシステムは既存のプロジェクトに改良を加えたものであり，現在はAltera DE2-115 FPGA ボード上でLinux

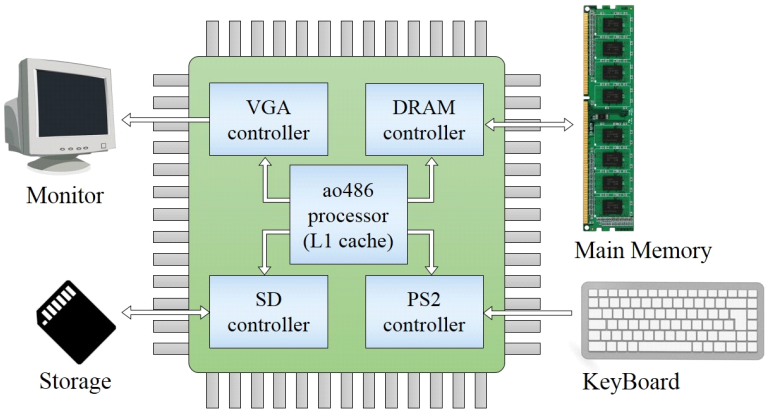

図 2.7: ao486 SoC の概略図

(Tiny Core 5.3) と FreeDOS 1.1 が起動する． 以下にこの計算機の構成について述べる．

2.4.1 ao486 SoC

高速化を行う計算機には，x86 をサポートし，かつ OS が動く既存の FPGA システムとして ao486[7] と呼ばれるオープンソースのプロジェクトをベースとして用いている．ao486 とは，ハードウェア記述言語である Verilog HDL で記述されており，Intel 80486SX の全ての機能を実装したソフトプロセッサである．これに加えて，ao486 プロジェクトには VGA コントローラー，PS2 コントローラーなどの様々なデバイスコントローラーが含まれており，これらを合わせて ao486 SoC と呼んでいる．図 2.7 に ao486 SoC の概略図を示す．ao486 SoC は Terasic 社の Altera DE2-115 FPGA ボード上で動作し，開発者のドキュメントによれば，Linux カーネル 3.13 と Windows95 を起動することができる．

2.4.2 開発中の計算機の構成

性能評価を行う計算機は，ao486 SoC に一部改良を加えたものである．ao486 SoC は，実装の簡単化のために Altera 社のソフトコアプロセッサである Nios Ⅱ を使用している．Nios Ⅱ は計算機が起動する際の BIOS の転送等を行っており，多くのデバイスコントローラーと接続している．性能評価を行う計算機では，この Nios Ⅱ の代わりに BIOS の転送等を行うモジュールを Verilog HDL で記述し，さらに評価に不必要なサウンドモジュールなどの削除を行っている．改良後の計算機の FPGA 内部のブロック図を図 2.8 に示す．

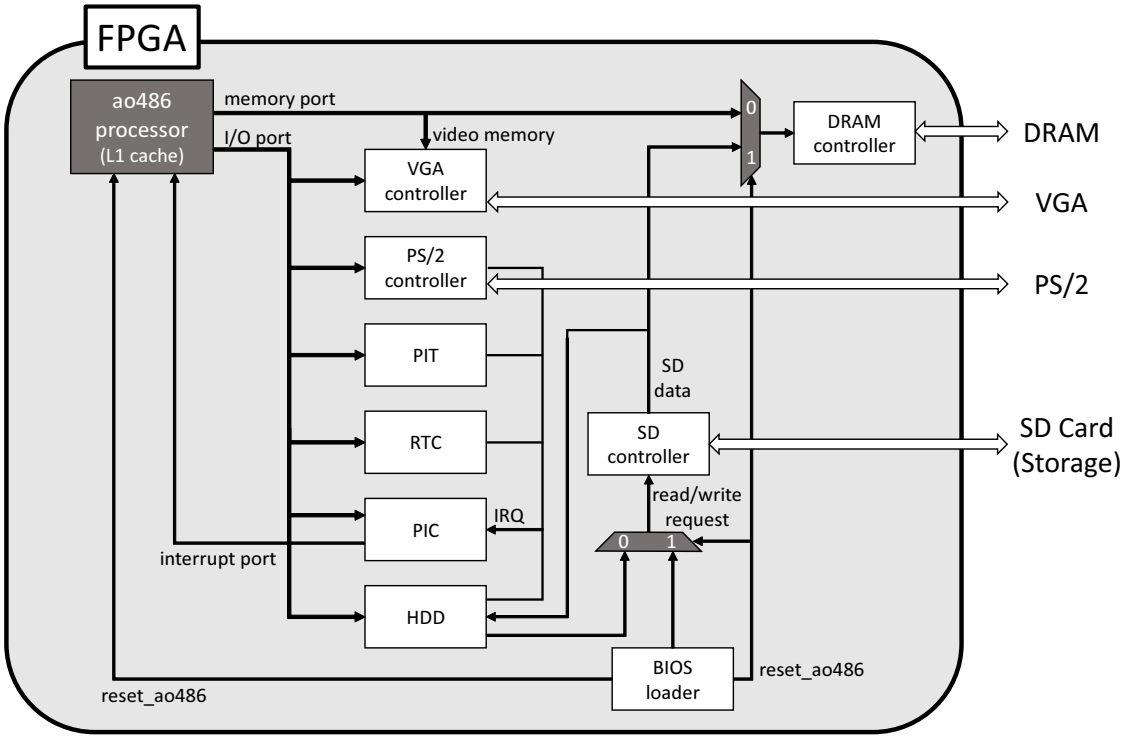

図 2.8: FPGA で実現された計算機のブロック図

ao486 プロセッサーは，内部にそれぞれ 16KB の命令キャッシュとデータキャッシュを持っている．I/O ポートは，DRAM，VGA，PS/2，SD Card の 4 つがあり，それぞれコントローラーで制御されている．この他に，PIT (Programmable Interval Timer)，RTC (Real Time Clock)，PIC (Programmable Interrupt Controller)，HDD，BIOS loader といったモジュールが FPGA に実装されている． HDD モジュールは，内部では SD contoller と接続しており，実際のデータの保存領域は SD カードである．SD カードは，OS 等を格納するストレージの役割と，BIOS 等のファームウェアを格納するメモリの役割を持つ．

計算機の起動時には，BIOS loader が SD controller に要求を出し，SD カードに格納されている BIOS のデータを DRAM へ転送する．この間，BIOS loader は ao486 プロセッサーへの reset 信号を high にする．BIOS の転送が終わると BIOS loader から ao486 プロセッサーへの reset 信号は low となり，プロセッサーが動作を開始する．SD カードのストレージ領域に適切に OS を格納することで，この計算機上での汎用 OS の起動を確認している．図 2.9 はこの計算機上で Tiny Core 5.3 が起動した際のスクリーンショットであり，図 2.10 は FreeDOS 1.1 を起動し，DOOM[8] をプレイしているスクリーンショットである．

図 2.9: FPGA で実現された計算機での Linux (Tiny Core 5.3) の起動の様子

図 2.10: FPGA で実現された計算機で FreeDOS 1.1 を起動し，DOOM をプレイする様子

第3章

高性能キャッシュを用いた高速化

3.1 性能低下原因の検討

2.4節で述べたFPGAで実現された計算機は，現代の計算機と比べて極めて低速であり，実用的とは言いがたい．そこで本論文では，この計算機の高速化を目指す．高速化を目指すにあたり，計算機の性能を下げる要因としてメモリアクセス時間が占める割合が大きいのではないかと考え，予備評価を行った．

この計算機環境を実装したCyclone IV EP4CE115F29C7を搭載するAltera DE2-115 FPGAボード[10]上でLinux (Tiny Core 5.3)を起動し，アプリケーションを実行した際の実行サイクル数等をホストコンピューターで計測することにより評価を行う．使用したアプリケーションはクイックソート，行列積計算，SPEC CPU92ベンチマーク[9]に含まれる整数演算の碁を解くプログラムの3つである．クイックソートは1024×1024要素のランダムな数のソートを行うプログラムであり，行列積計算は256×256の行列積の計算を行うプログラムである．碁のプログラムは，コンピュータが自動で対戦を行い，白黒両者ともにパスするまで対局を行う．プレイヤーレベルと版面の大きさを引数で指定することができる．プレイヤーレベルは探索の深さに影響する．このプログラムにプレイヤーレベル，版面ともに12の引数を与えて測定を行った．これらのアプリケーションは全てC言語で記述されており，これをホストコンピュータでIntel 80486向けにGCCを用いてコンパイルし，実行ファイルとしている．

プロセッサがDRAMへread要求を行ってからデータが得られるまでの待ち時間，プロセッサのwrite要求が受理されるまでの待ち時間及びプロセッサの計算等の時間をサイクル数から計測し，アプリケーションの実行時間における割合を調べたところ，図3.1に示す通りになった．sortはクイックソートプログラム，mmは行列積計算，goは碁を解くプログラムを表す．このプロセッサは内部にL1キャッシュを持つため，DRAMへのread/write要求はL1キャッシュにミスした場合に行われる．

この計測によると，行列積計算と碁を解くプログラムにおいては，read待ち時間とwrite待ち時間を合わせたメモリアクセスによるプロセッサの待ち時間

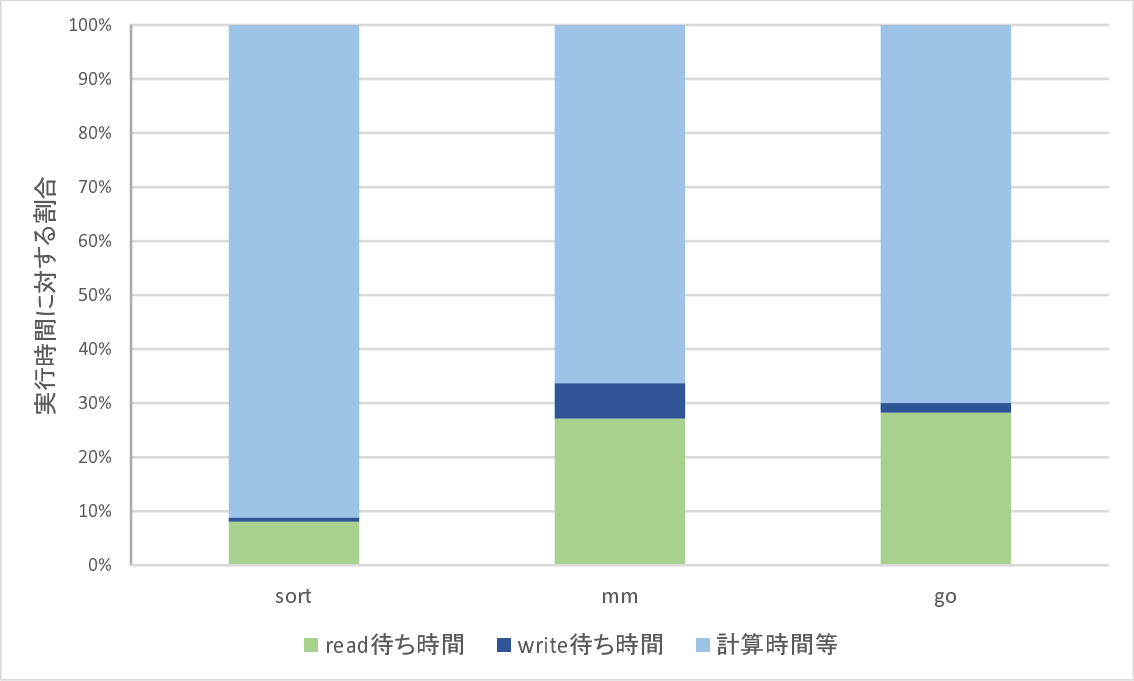

図 3.1: アプリケーションの実行時間の内訳

が実行時間全体の約30%を占める．評価に用いたFPGAで実現された計算機のプロセッサはインオーダー実行であり，メモリアクセスによる待ち時間の間プロセッサはストールする．この結果から，メモリアクセスの多いアプリケーションを動かす際に，高性能なL2キャッシュを実装することによりメモリアクセスによるストール時間を減らすことが出来れば，計算機の性能向上が達成できると予想される．本論文では，高性能なL2キャッシュの実装による計算機の高速化を目指す．次節より，作成した高性能キャッシュの構成について述べる．

3.2 キャッシュの構成

3.1節の予備評価の結果を踏まえ，FPGAで実現された計算機にL2キャッシュを実装した．表3.1に，実装した8つのキャッシュの構成を示す．キャッシュへの書き込みの制御にはライトバック方式を採用した．

実装したキャッシュは，Verilog HDLで記述しており，キャッシュコントローラーモジュール，メモリバンクモジュール，RAMモジュールの3つで構成されている．メモリバンクモジュールはRAMモジュールを内包し，RAMへの書き込みの制御やタグの管理等を行う．キャッシュコントローラーモジュールはメモリバンクモジュールを内包し，ステートマシンを用いることで，メモリバン

表 3.1: 実装するキャッシュの構成

構成名	連想度	ラインサイズ	エントリ数	skewed
1word Direct mapped	Direct mapped	1 word	2^{14}	no
4word Direct mapped	Direct mapped	4 word	2^{12}	no
1word 2-way set associative	2-way set associative	1 word	2^{13}	no
4word 2-way set associative	2-way set associative	4 word	2^{11}	no
4word 2-way skewed associative	2-way set associative	4 word	2^{11}	yes
1word 4-way set associative	4-way set associative	1 word	2^{12}	no
4word 4-way set associative	4-way set associative	4 word	2^{10}	no
4word 4-way skewed associative	4-way set associative	4 word	2^{10}	yes

クへの書き込みの制御やプロセッサ，DRAMの入出力を管理する．

3.2.1 キャッシュの状態遷移

作成したキャッシュの状態遷移図を図 3.2に示す．キャッシュはIDLE状態でのみread/write要求を受け付ける．IDLE状態においてread/write要求を確認すると，キャッシュはCOMP状態に遷移する．COMP状態では，要求があったアドレスのデータがキャッシュ内に存在するかを確認する．各メモリバンクは，要求があったアドレスからキャッシュのインデックスを生成し，そのインデックスが保持していたタグの値と要求のあったアドレスのタグとを比較することで，hit/modify/miss の 3つのうちいずれかのステータスをキャッシュコントローラーへ返す．各ステータスにおけるキャッシュの状態遷移は以下の通りである．

hit

hitは要求のあったデータがキャッシュ内に存在していたことを示す．hitの場合にはread/write共に1サイクルでIDLE状態に戻ることができる．

modify

modifyは要求のあったデータがキャッシュ内に存在せず，かつキャッシュとメインメモリの間に一貫性が保たれていないことを示す．この場合，キャッシュに新たなデータを書き込む為にはキャッシュ内のデータをライトバックする必要が生じる．そのためキャッシュはWB状態へ遷移し，メインメモリへの書き戻しを行い，その後DRAMから必要なデータを得るためのFETCH状態へと遷移する．

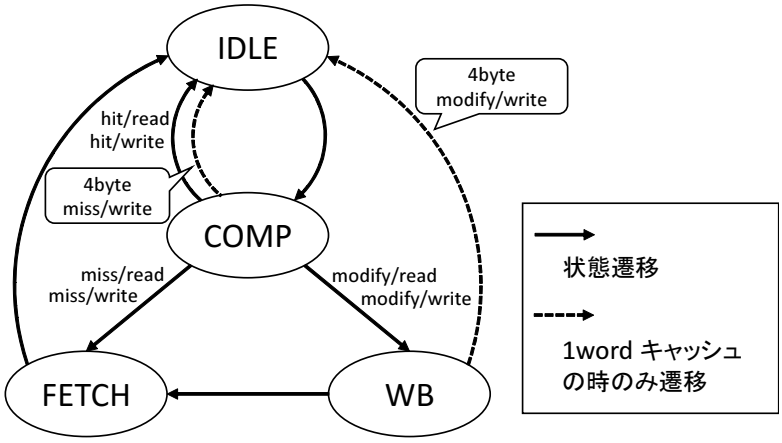

図 3.2: L2 キャッシュの状態遷移図

miss

miss は要求のあったデータがキャッシュ内に存在せず，かつキャッシュとメインメモリの間に一貫性が保たれている場合を示す．キャッシュに有効なデータが無い場合も miss となる．miss の場合には，キャッシュデータの書き戻しを行う必要は無いため，DRAM から必要なデータを得るための FETCH 状態へと遷移する．

なお，ラインサイズが 1word のキャッシュのみ，状態遷移が異なる場合がある．プロセッサからの write 要求があった場合，32bit の writedata と共に 4bit の byte_enable 信号が送られる．byte_enable 信号が立っているバイトのみを書き換える処理はキャッシュの中で行っている．キャッシュはバイト毎の書き込みが可能であるが，キャッシュラインは常に 1word 分最新のデータを保持する必要がある．そのため，1 バイトや 2 バイトの半端な書き込みに関しては，キャッシュにヒットしなかった場合にメインメモリから該当するデータをフェッチし，その上で write 要求のあったバイトの上書きをする必要がある．しかし，4 バイトの write 要求であれば，要求した writedata がキャッシュに保持するべき最新の値であるため，メインメモリから該当データをフェッチする必要はない．そのため，modify や miss の時に FETCH 状態を経由せずに IDLE 状態に戻ることができる．4word キャッシュの場合には，4 バイト書き込みに関しても FETCH 状態を経由する必要がある．これは，キャッシュラインが 4word であるので，4word の最新のデータをキャッシュに保持する必要があり，キャッシュミスの際には必ず DRAM からデータをフェッチするためである．

これに加え，ラインサイズが 4word のキャッシュの場合には，DRAM からのデータのフェッチ及び DRAM へのデータの書き戻しは 4word 分行う必要があり，キャッシュミスの際のレイテンシは 1word のキャッシュよりかなり大きい．

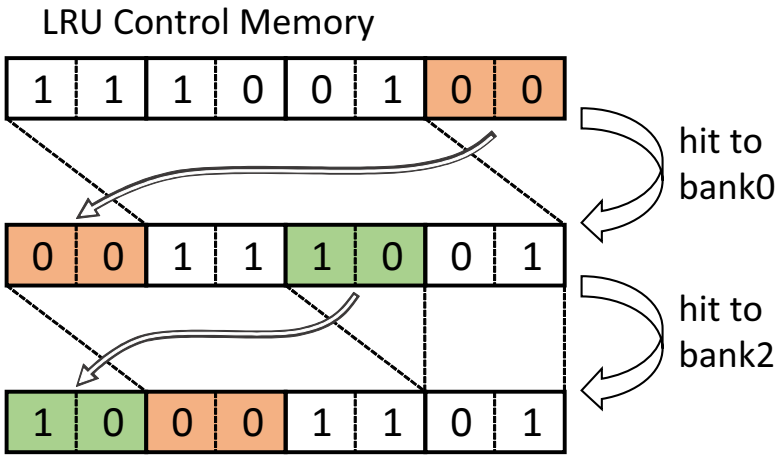

図 3.3: LRU Control Memory の遷移

3.2.2 LRU の実装

2-way セットアソシアティブ，4-way セットアソシアティブのキャッシュの置き換えアルゴリズムには LRU を採用した．LRU の実現のためには，キャッシュバンクが使用された履歴を保存する必要がある．2-way セットアソシアティブキャッシュの場合には各キャッシュエントリ毎にどちらのキャッシュバンクが最近使われたかを示す 1bit のフラグを用意する．4-way セットアソシアティブキャッシュには，少し実装コストが高いが簡単な方法を用いた．各キャッシュエントリ毎に 8bit の LRU Control Memory を用意し，4 つのキャッシュバンクの使用履歴を保存する．図 3.3 に，LRU Control Memory による LRU の実現の様子を示す．LRU Control Memory には，2bit 毎に 0，1，2，3 いずれかの数値が保存される．上位 bit ほど最近使用されたことを示し，下位 bit ほど使用された履歴がもっとも古いことを示す．例えばメモリバンク 0 にキャッシュヒットした場合には，LRU Control Memory の中の {00} を最上位に移動することで，メモリバンク 0 が最も最近使用されたことを表す．キャッシュデータの置き換えが必要になった時には，LRU Control Memory の最下位 2bit を見ることで，置き換えるべきキャッシュバンクを知ることができる．

3.3 skewed キャッシュの実装

ラインサイズが 4word の 2-way セットアソシアティブと 4-way セットアソシアティブのキャッシュに関して，skewed キャッシュを実装した．実装は，2.3 節の手法に倣った．アドレス A が $A = \{A_3, A_2, A_1, A_0\}$ と表される時に，$index_0$ から

$index_3$ は

$$\begin{aligned} index_0 &= \{\phi(A_1) \oplus \phi'(A_2)\} \\ index_1 &= \{\sigma(\phi(A_1)) \oplus \phi'(A_2)\} \\ index_2 &= \{\sigma^2(\phi(A_1)) \oplus \phi'(A_2)\} \\ index_3 &= \{\sigma^3(\phi(A_1)) \oplus \phi'(A_2)\} \end{aligned}$$

と表される．今回の実装では，σ は1bitの右巡回シフトとし，ϕ と ϕ' は1とする．σ^2 は2bitの右巡回シフト，σ^3 は3bitの右巡回シフトとなる．

実装するキャッシュはライトバックであるため，キャッシュからメインメモリへの書き戻しの際にはインデックスとタグから元のアドレスを復元する必要がある．例えば，$index_1$ からアドレスを復元するとき，

$$index_1 = \{\sigma(A_1) \oplus A_2\}$$

であるから，A_1 は

$$A_1 = \{\sigma^{-1}(index_1 \oplus A_2)\}$$

と表される．$\{A_3, A_2\}$ はタグとして保持されているため，これにより元のアドレスを復元することができる．これらの処理は1サイクルで行うことができ，元のキャッシュと比べてミス時のレイテンシが伸びることはない．

3.4 FPGAで実現された計算機への実装

作成したL2キャッシュは，2.4節で述べたFPGAで実現された計算機におけるDRAMコントローラーとプロセッサの間に実装した．L2キャッシュの挿入位置を記したブロック図を図3.4に示す．計算機の起動時には，SDカードに格納されているBIOSのデータをDRAMへ転送するため，SD controllerからのwrite要求を受け付けるが，転送が終わりプロセッサのリセットが解除されるとプロセッサからのread/write要求を受け付けるようになる．挿入したL2キャッシュの周辺モジュールのブロック図を図3.5に示す．プロセッサから要求するreaddata及びwritedataは基本的に32bitである．

プロセッサはバーストread/writeを行う．バーストアクセスは，1から4の整数値を取るバーストカウントにより制御され，バーストカウントの回数分だけアドレスをインクリメントして要求を出す．例えば0x00のアドレスでバーストカウント4のread要求がプロセッサから出された場合には，0x00，0x04，0x08，0x0Cの4つのアドレスにread要求を出したことと等しい．バーストアクセスはキャッシュに入る前にBurst Controllerが処理を行い，32bitごとの要求に変換している．また，バイト毎の書き込みは3.2.1節で述べたようにキャッシュの中で処理を行う．

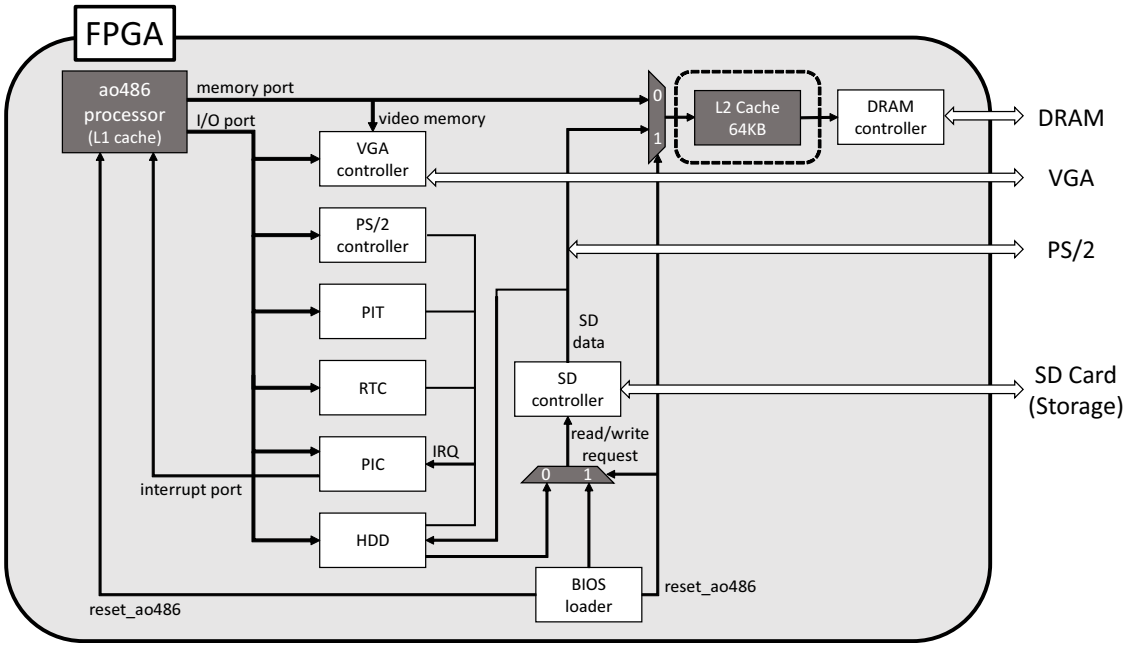

図 3.4: L2 キャッシュを実装した FPGA で実現された計算機のブロック図

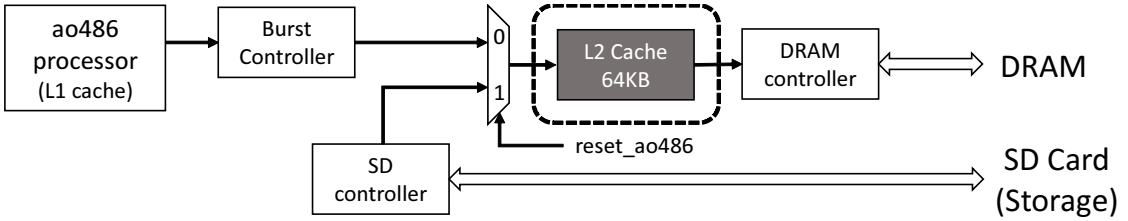

図 3.5: L2 キャッシュの周辺モジュールのブロック図

L2 キャッシュのサイズは 64KB である．これは，Cyclone IV EP4CE115F29C7 を搭載する Altera DE2-115 FPGA ボードに実装することを考慮し，Cyclone IV EP4CE115F29C7 のメモリビットリソースを最大限使うことができる大きさに設定したためである．

第4章

評価

4.1 評価環境

4.1.1 ハードウェアの評価環境

評価には，Cyclone IV EP4CE115F29C7 を搭載する Altera DE2-115 FPGA ボードを用いる．3 章で述べた各キャッシュを実装した，FPGA で実現された計算機環境で Linux (Tiny Core 5.3) を動作し，アプリケーションを実行した際の実行サイクル数等をホストコンピューターで計測することにより評価を行う．

図 4.1 に，実際の測定の様子を示す．Altera DE2-115 FPGA ボードの GPIO の 34 番ピンを通じて UART 転送用のモジュールからホスト PC へシリアル通信を行う．ホスト PC では，TeraTerm などの端末エミュレータを用いてシリアル通信のデータを表示する．FPGA 側では，総サイクル数，キャッシュへの read/write 数，キャッシュヒット数などをキャッシュモジュール等でカウントしており，ディスプレイに”#” 記号が出力されたタイミングでこれらのデータをシリアルデータに変換してホスト PC に転送するように設定している．# 記号が出力されたことは，画面出力を制御している VGA コントローラーモジュールに入る writedata を監視することで確認できる．

アプリケーション側では，計算を開始する前と計算が終了した直後に# 記号を出力するようプログラムする．これにより，アプリケーションの正確な実行時間やキャッシュのミス率をカウントできる．

4.1.2 使用するアプリケーション

評価に用いたアプリケーションには，3.1 節で述べたクイックソート，行列積計算，碁を解くプログラムの 3 つのプログラムである．

クイックソートは，1024×1024 要素をソーティングする．行列積と同じく Xorshift を用いて要素を作成し，クイックソートアルゴリズムを用いてソーティングを行う．

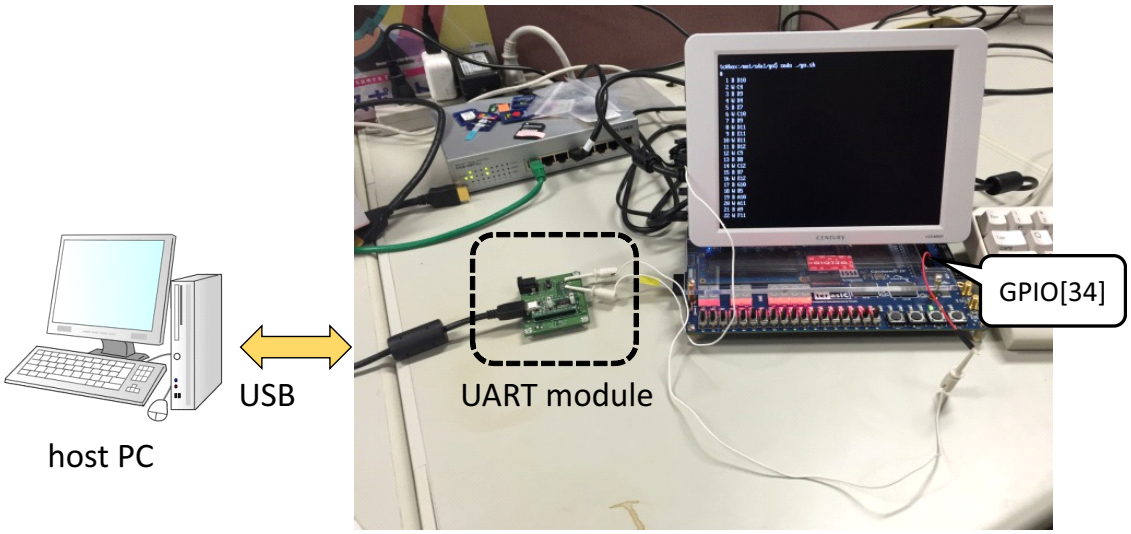

図 4.1: 評価環境での測定の様子

行列積計算は，256 × 256 の行列の積を計算する．擬似乱数生成法の一つである Xorshift を用いて 2 つの行列を生成し，その積を計算する．

碁のプログラムはプレイヤーレベル，版面のサイズともに 12 の引数を与える．プログラムは，プレイヤーレベルに応じた探索を行う．白と黒でそれぞれ探索できる限りでの最適解を選び，両プレイヤーがパスするとゲームオーバーとなり，プログラムが終了する．

また，OS 上での評価である利点を生かし，これらの 3 つのアプリケーションの並列実行による評価も行った．これらのアプリケーションは全て C 言語で記述されており，これをホストコンピュータで Intel 80486 向けに GCC を用いてコンパイルし，実行ファイルにしたものを Tiny Core のイメージファイルの中にあらかじめ入れておく．これにより，評価に用いる FPGA で実現された計算機において Tiny Core 起動後にアプリケーションを実行できる．

4.2 ハードウェアリソースの評価

表 4.1 に Cyclone IV EP4CE115F29C7 に実装した際の各キャッシュのハードウェアリソース使用量を示す．回路規模の参考のため，評価環境のプロセッサである ao486 processor のハードウェアリソース使用量も示している．LC Util per CPU は，ao486 processor と比較したときの各キャッシュの Logic Cells の使用率を示す．

各キャッシュは全てキャッシュサイズを 64KB に設定している．そのため，各

表 4.1: 各キャッシュのハードウェアリソース使用量

cache	Logic Cells	Memory Bits	LC Util per CPU
1word Direct mapped	510	737,280	1.4%
4word Direct mapped	632	577,536	1.6%
1word 2-way set associative	589	761,865	1.6%
4word 2-way set associative	815	583,680	2.3%
4word 2-way skewed associative	846	583,680	2.3%
1word 4-way set associative	841	802,816	2.3%
4word 4-way set associative	1,183	593,920	3.3%
4word 4-way skewed associative	1,284	593,920	3.6%
ao486 processor	36,159	310,784	100%

キャッシュはキャッシュエントリの大きさが異なり，キャッシュエントリの大きさが大きい程タグの保持に用いるメモリのエントリ数も増えるため，リソースを多く消費する．また，2-way セットアソシアティブキャッシュと 4-way セットアソシアティブキャッシュは LRU を実現するための LRU Control Memory を使用しており，その分リソースを多く消費している．

skewed キャッシュは，同じブロックサイズ，連想度の通常のキャッシュと比較してリソース消費量に大きな差はない．skewed キャッシュの実現には新たなメモリを使用せず，簡単なロジック回路で実現できるためである．

4.3 キャッシュミス率の評価

実装した各キャッシュを用いた計算機環境で，前述の 3 つのアプリケーションの実行及びその並列実行を行った際のキャッシュのミス率を評価した．この場合のキャッシュミスは，キャッシュが受け付けた全ての read/write 要求のうち，キャッシュヒットしなかった割合のことを示す．

図 4.2 に，各 L2 キャッシュ毎のアプリケーション実行時のミス率を示す．グラフの中の mm は行列積計算，sort はクイックソート，go は碁を解くプログラムをそれぞれ表す．また，sort&mm はクイックソートと行列積計算の並列実行を示し，mm&go は行列積計算と碁を解くプログラムの並列実行，go&sort は碁を解くプログラムとクイックソートの並列実行を示す．sort&mm&go は 3 つのアプリケーションの並列実行を表す．

グラフよりどのアプリケーションに関しても，ダイレクトマップ，2-way セットアソシアティブ，4-way セットアソシアティブと連想度が上がるにつれキャッシュのミス率が下がることが確認できた．また，同様にブロックサイズが 1word

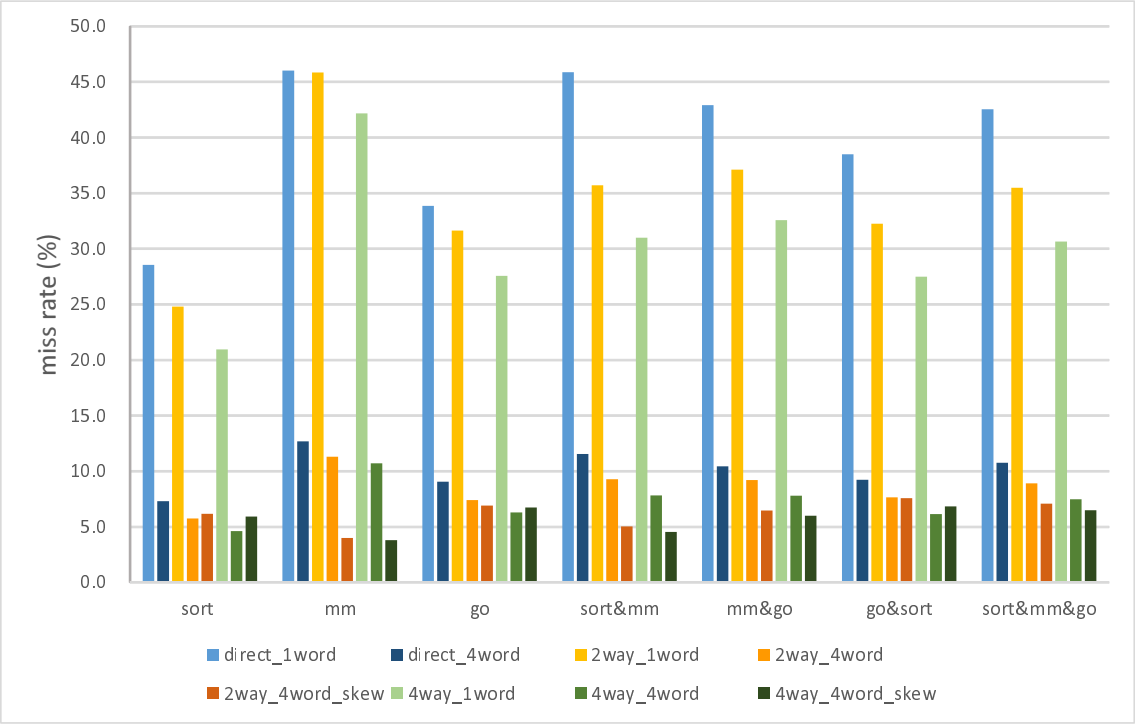

図 4.2: 各 L2 キャッシュ毎のアプリケーション実行時のミス率

から 4word になることでキャッシュミス率が大幅に下がった．

skewed キャッシュは，同じ構成の skewed でないキャッシュに比べ，クイックソートに関しては効果がなく，むしろミス率の悪化が見られた．しかし，碁を解くプログラムでは若干の改善が見られ，行列積計算では大幅にミス率を改善できることがわかった．2 つ以上のアプリケーションの並列実行に関しては，行列積計算が含まれる場合にはミス率の大幅な改善が見られるが，行列積計算を含まないクイックソートと碁を解くプログラムの並列実行では効果が得られなかった．

3 つのアプリケーションの並列実行で比較すれば，最も高かったブロックサイズ 1word のダイレクトマップキャッシュのミス率が 42.5%なのに対し，最も低かったブロックサイズ 4word の 4-way skewed アソシアティブキャッシュのミス率は 6.51%であり，大幅に改善された．

4.4 性能評価

実装した各 L2 キャッシュを用いた計算機環境で，前述の 3 つのアプリケーションの実行及びその並列実行を行った際のアプリケーションの実行時間を評

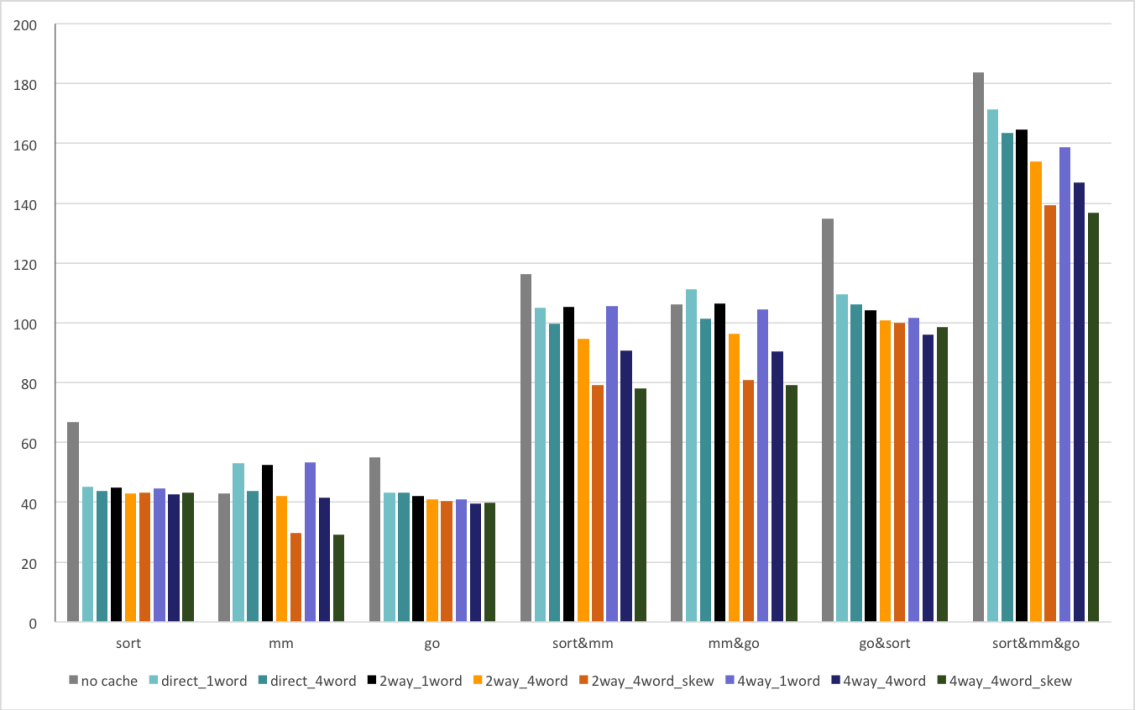

図 4.3: オリジナルと評価対象の各 L2 キャッシュを実装した計算機上でのアプリケーション実行時間

価した．図 4.3 に，L2 キャッシュを実装していないオリジナルの計算機と，評価対象の各 L2 キャッシュを実装した計算機におけるアプリケーションの実行時間を示す．

L2 キャッシュのないオリジナルの環境と最も性能の良いブロックサイズ 4word の 4-way skewed アソシアティブキャッシュを比較すると，sort プログラムで 1.55 倍，行列積計算で 1.46 倍，碁を解くプログラムでは 1.37 倍の高速化が得られた。3 つのプログラムの並列実行で比較すると，1.34 倍の高速化が得られた．しかし，行列積計算においてはブロックサイズ 1word のキャッシュを実装した環境がオリジナルの環境よりも実行時間が延びてしまう結果となった．

次に，L2 キャッシュ毎の比較を行う．クイックソートでは，各キャッシュによる実行時間の差はほとんど見られず，ミス率の時と同様に，skewed キャッシュの導入により同じ構成の skewed でないキャッシュと比べ実行時間が若干伸びるという結果になった．これに対し，行列積計算では，キャッシュによる実行時間の差が大きく現れた．ダイレクトマップ，2-way セットアソシアティブ，4-way セットアソシアティブと連想度を上げた時の差はあまり見られないが，ブロックサイズを 1word から 4word にあげた時は実行時間が大きく削減されている．

また，skewed キャッシュにすることで大きく実行時間の削減が達成されている．碁を解くプログラムでは，ブロックサイズの差よりも連想度を上げた時の性能向上が大きく現れている．また，skewed キャッシュに関しては，2-way セットアソシアティブキャッシュから 2-way skewed アソシアティブキャッシュにすることで性能向上が得られた一方で，4-way セットアソシアティブキャッシュから 4-way skewed アソシアティブキャッシュでは性能向上は得られなかった．

アプリケーションの並列実行においては，行列積計算を含む場合には行列積計算の単体実行の場合と同様の傾向が見られた．行列積計算を含まない，碁を解くプログラムとクイックソートの並列実行では，碁を解くプログラムの単体実行の時と同様の傾向が見られた．

3 つのアプリケーションの並列実行で比較すると，最も実行時間の長いブロックサイズ 1word のダイレクトマップキャッシュの実行時間が 171.3(sec) なのに対し，最も短かったブロックサイズ 4word の 4-way skewed アソシアティブキャッシュの実行時間は 136.9(sec) であり，1.25 倍程度の高速化が得られた．

4.5 考察

4.5.1 キャッシュミス率の考察

キャッシュのブロックサイズが 1word から 4word に増えると，キャッシュミス率は大幅に削減されたのに対し，実際のアプリケーションの実行時間は短縮されたものの，わずかな差である．

キャッシュのブロックサイズが増えることで，キャッシュから DRAM へのフェッチ及び書き戻しにかかるデータ量が増えるため，キャッシュミス時のレイテンシが増えるということが一つの要因である．

もう一つの要因は，キャッシュミス率の測定法にある．キャッシュミス率は，キャッシュへの read/write 要求の回数のうちキャッシュにヒットしなかった回数の割合を示す．ブロックサイズが 4word のキャッシュは，キャッシュミスを起こすと，要求があったデータを含む 4word 分のブロックのデータを DRAM からフェッチする．プロセッサからの read/write 要求はバーストアクセスをしている場合が多く，この場合同じブロックのデータに連続してアクセスすることになる．バースト要求は，キャッシュに入る前に個別の read/write 要求に変換されるため，例えば 4word のバースト read 要求はキャッシュからは 4 回の read 要求に見える．ブロックサイズが 4word のキャッシュでは，この 4 回の read 要求のうち，最初の要求を除く 3 回はキャッシュのヒットが約束されてしまうので，キャッシュのヒット率は 1word のキャッシュに比べ跳ね上がる．

表 4.2: アプリケーション毎のキャッシュへの要求回数

app	read request (million)	write request (million)
sort	54.6	16.4
mm	101.8	70.7
go	156.4	25.3

4.5.2 各アプリケーションとキャッシュの性能

実行時間の評価を見ると，アプリケーション毎に異なる傾向が見られる．特にクイックソートはキャッシュによる性能の差がほとんど見られない．

表 4.2 に，各アプリケーション実行時のキャッシュへの read/write 要求の回数を示す．これは，ブロックサイズ 1word のダイレクトマップを実装した際のデータである．要求回数の十万回以下は四捨五入している．アプリケーションは OS 上で実行しており，同じアプリケーションの実行でも毎回要求回数が同じになるわけではない．また，評価環境のプロセッサはプリフェッチを行っており，キャッシュによりミス時のレイテンシが異なることから，実装するキャッシュによって read/write 要求に若干の差が生じる．

しかし，どのキャッシュでも表 4.2 と同様の傾向は示しており，クイックソートの read/write 要求の回数は他のアプリケーションと比べて少ないことがわかる．このため，キャッシュによる性能の差異があまり出なかったと考えられる．

行列積計算は，キャッシュのよる性能の差異が一番大きく現れており，特に skewed キャッシュによる効果が大きい．これは，本来キャッシュのインデックスとして使われるアドレスの下位ビット部分が等しいデータに頻繁にアクセスしているためであると考えられる．このプログラムでは，int 型の要素を行列の要素数分だけ確保し，xorshift で生成した乱数を代入した配列 a と配列 b を用いて計算を行う．このとき，配列 b は行列の列ごとのアクセスを行う．図 4.4 に 256×256 の行列積の計算における，配列の位置づけと割り当てられるメモリアドレスの例を示す．例えば，b[0] がメモリアドレスの 0x000 から始まるとき，b[1]，b[2] という順にメモリアドレスが連続する．ところが，配列 b は列ごとにアクセスするため，b[0] の次は b[256] にアクセスすることになり，メモリアドレスは 0x400 になる．このように，列ごとのアクセスではメモリアドレスが一定間隔空くため，アドレスの下位ビットが等しいデータに頻繁にアクセスすることになる．通常のキャッシュではアドレスの下位ビットがキャッシュのインデックスであるため，このような場合にキャッシュデータの置き換えが頻繁におこり，性能の低下につながる．skewed キャッシュを用いることで，アドレスの下位ビットが同じでもデータの衝突を少なくすることができる．

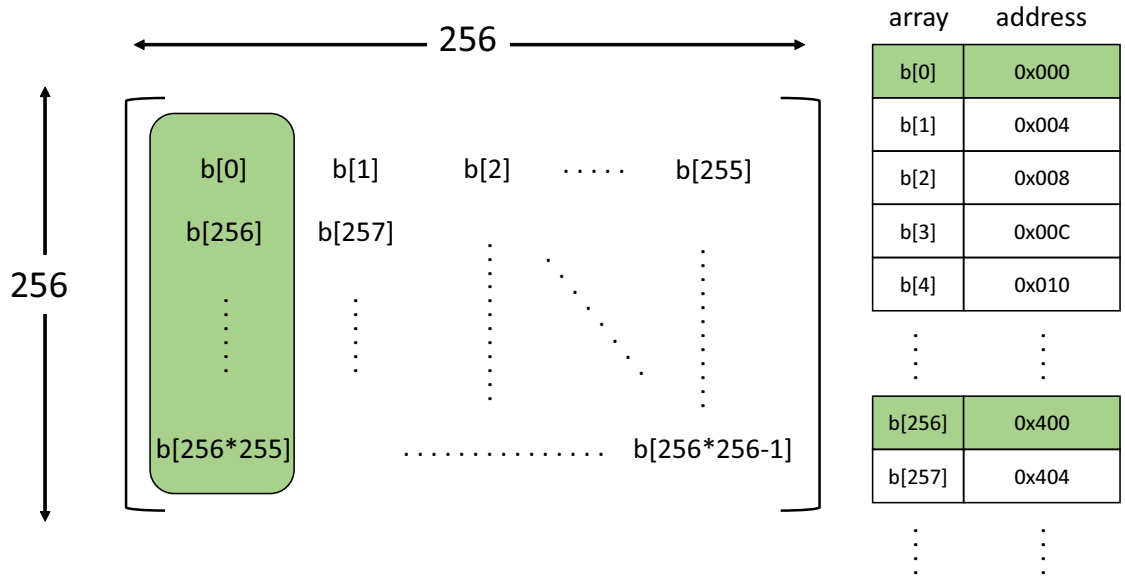

図 4.4: 行列積計算における配列の位置づけと割り当てられるメモリアドレスの例

碁のプログラムは，行列積計算と同程度のメモリアクセスをしている．しかしキャッシュのブロックサイズを大きくしたことによる性能の差異は少なく，連想度を大きくすることで速度向上が得られた．これは，アドレスが連続したメモリアクセスが少ないためであると考えられる．キャッシュの連想度を大きくすると連続したメモリアクセスでなくともキャッシュミス率を減らすことができるため，速度向上が得られた．

3つのアプリケーションを同時に実行した場合には，各アプリケーションを個別に実行した場合の合計よりも実行時間が長くなる．図 4.5は，3つのアプリケーションの並列実行時間と，各アプリケーションの実行時間の合計とのキャッシュ毎の比較を行ったグラフである．評価を行った計算機環境はシングルコアであり，優先度の等しいアプリケーションの並列実行は一定間隔でタスク切り替えを行うことで達成される．別のアプリケーションに切り替わる際にキャッシュミスが頻発することで性能が悪化すると考えられる．

4.5.3 skewed キャッシュの性能

skewed キャッシュの性能を考察する．図 4.6と図 4.7は，前節までの評価のうち，2-way skewed アソシアティブキャッシュと 4-way セットアソシアティブキャッシュの実行時間とミス率を切り出したグラフである．

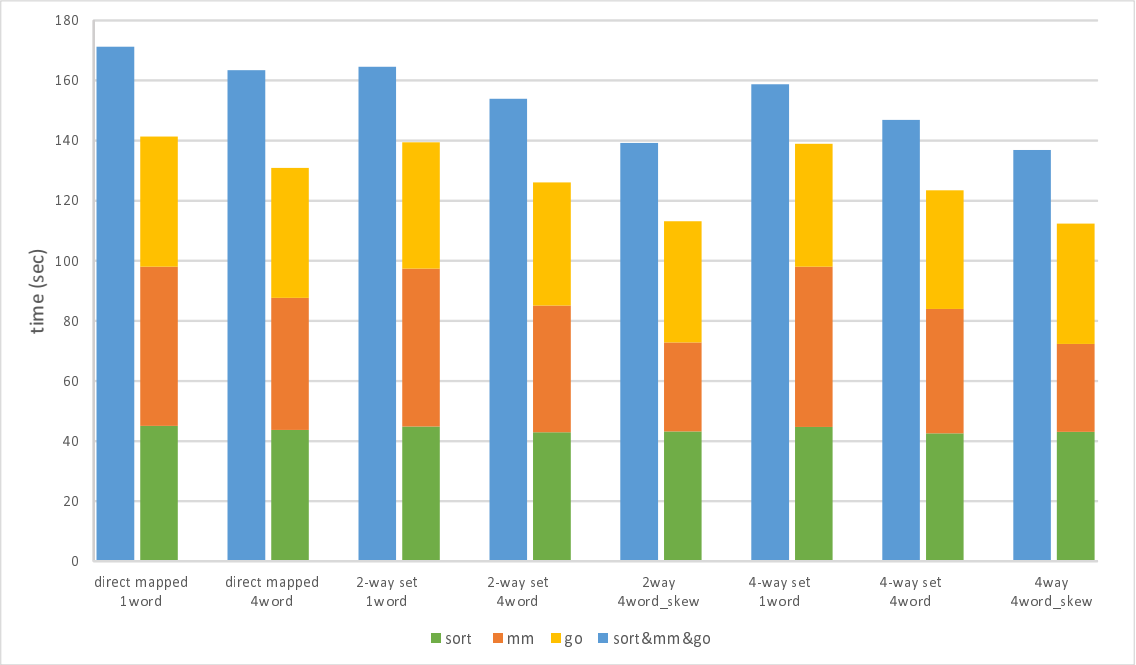

図 4.5: 3つのアプリケーションの並列実行と個別に実行した場合の合計との実行時間の比較

前述までの考察の通り，行列積計算に関してはskewedキャッシュの効果が大きく，4-wayセットアソシアティブキャッシュと比べてもキャッシュミス率は半分以下であり，実行時間も大幅に削減されている．クイックソートと碁を解くプログラムに関しては若干2-way skewedキャッシュの方がミス率で劣るが，実行時間には大きな差はない．

これらのことから，2-way skewedアソシアティブキャッシュは，連想度が低いにもかかわらず4-wayセットアソシアティブキャッシュに匹敵する性能を持つと言える．また，2-way skewedアソシアティブキャッシュは，連想度が低い分LRU Control Memoryに必要なメモリリソースが4-wayセットアソシアティブキャッシュよりかなり少なくて済むという利点もある．加えて，2-wayセットアソシアティブキャッシュと比べても，リソース使用量に大きな増加はなく，ミスレイテンシが増えることもない．簡単な構成で実現できるという意味でも，すぐれた性能を有するキャッシュであることがわかった．

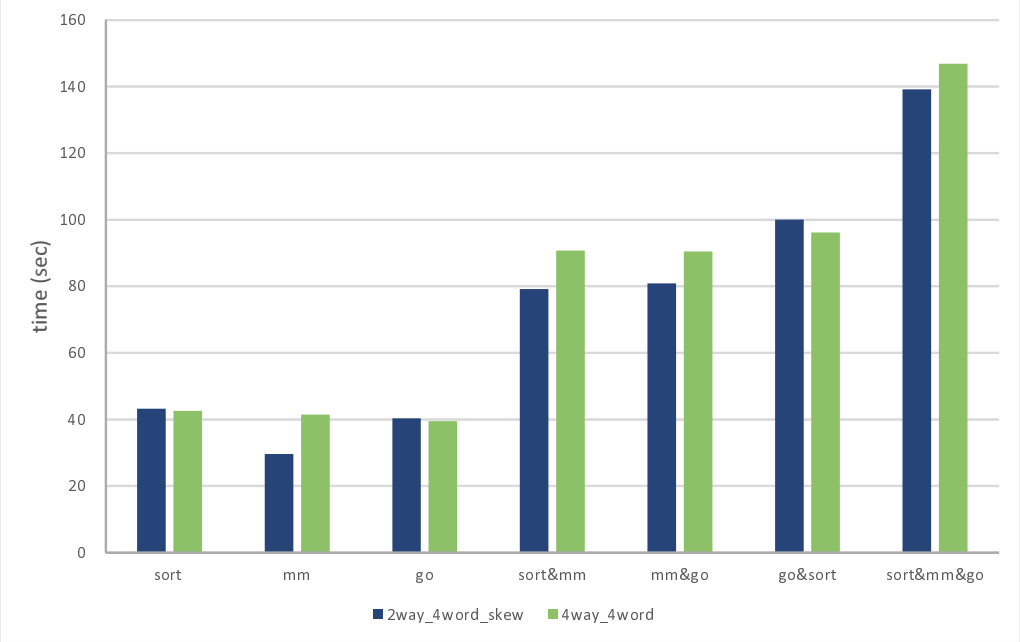

図 4.6: 2-way skewed キャッシュと 4-way キャッシュのアプリケーションの実行時間

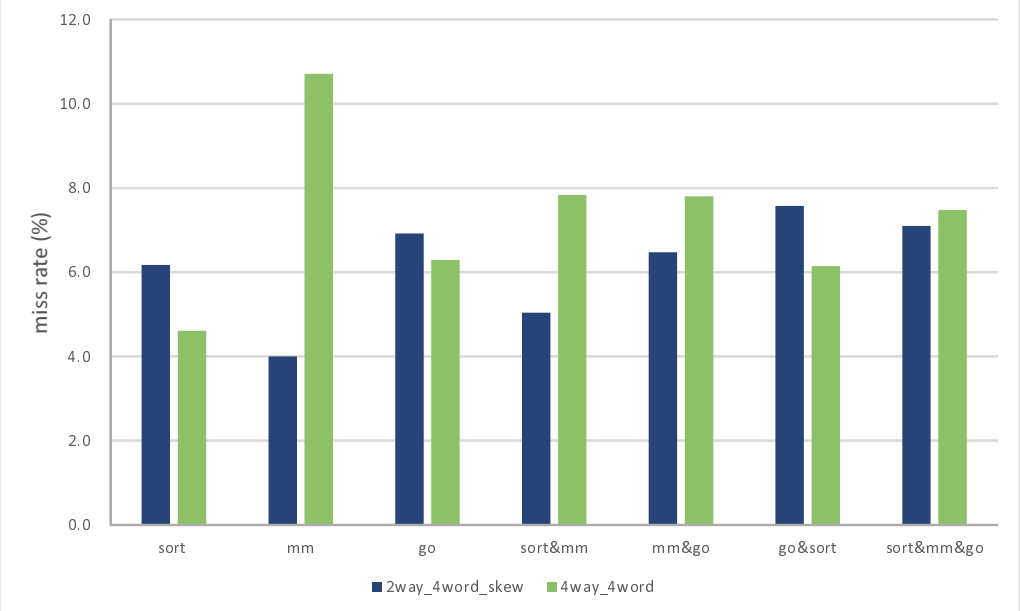

図 4.7: 2-way skewed キャッシュと 4-way キャッシュのミス率

第5章

結論

OSの動くFPGAで実現された計算機に，高性能なキャッシュを実装することでシステムの高速化を図った．また，FPGAで実現された計算機でLinuxを起動し，その上でアプリケーションを実行することで，skewedキャッシュを含む様々なキャッシュの性能評価を行った．その結果，ハードウェアリソースとキャッシュの性能を評価し，特に行列積計算においてskewedキャッシュが有効であることを示した．3つのアプリケーションの並列実行では，最も性能の高かったブロックサイズ4wordの4-way skewedアソシアティブキャッシュは，もっとも性能の低いブロックサイズ1wordのダイレクトマップキャッシュに比べて1.25倍程度，オリジナルのL2キャッシュが実装されていない環境と比べて1.34倍程度の速度向上を達成した．

今回実装したskewedキャッシュは，最も簡単なハッシュ関数で構成されているが，ハッシュ関数の最適化などの改良により，更なる性能向上が期待できる．また，評価に用いたFPGAで実現された計算機は，本論文で実装したskewedキャッシュを用いてもまだ近代の計算機と比較すると低速である．実用的なシステムを目指すためには，メモリシステム以外でも，プロセッサやバスなどに更なる改良を加える必要があり，これらを今後の課題としたい．

謝辞

本研究を進めるにあたり，適切かつ熱心な指導をしていただいた指導教員の吉瀬謙二准教授に深く感謝いたします.

本研究の起点となった計算機の開発メンバーである松田裕貴先輩と味曽野智礼君には，研究全般において多くのアドバイスを賜り，常に支えていただきました.

キャッシュに関する知識をご教授いただき，実装やデバッグを手伝っていただいた森 悠先輩，ご助言を頂くばかりでなく校閲にも協力してくださった小林諒平先輩に心から感謝いたします.

また，様々な相談に乗ってくださったThiem Van Chu 先輩や，版管理等のノウハウをご教授くださった奥村開里先輩，研究室に明るい色をもたらしてくれた臼井琢真君と吉瀬研究室の皆様にお礼を申し上げます.

参考文献

[1] Intel. ”マイクロプロセッサーの歴史”. インテルミュージアム. http://japan.intel.com/contents/museum/hof/index.html

[2] 堀 桂太郎. 図解コンピュータアーキテクチャ入門. 第2版, 森北出版株式会社, 2011, 13p

[3] David A.Patterson/John L.Hennessy. パターソン&ヘネシー コンピュータの構成と設計. 成田光彰訳. 第4版, 日経BP社, 2011

[4] André Seznec. *A Case for Two-Way Skewed-Associative Caches*. ISCA ’93 Proceedings of the 20th annual international symposium on computer architecture, pp.169–178. 1993

[5] André Seznec. *A New Case for Skewed-Associativity*. IRISA-INRIA, Campus de Beaulieu 35042 Rennes Cedex, FRANCE , 1997

[6] 松田裕貴, 小川愛理, 味曽野智礼, 藤枝直輝, 市川周一, 吉瀬謙二. MieruSysプロジェクト：複数のFPGAを用いた先進的な計算機システムの開発. RECONF CPSY VLD IPSJ-SLDM, 信学技報, vol. 114, no. 428, RECONF2014-79, pp. 211-216, 2015

[7] ao486. https://github.com/alfikpl/ao486

[8] idsoftware. http://www.idsoftware.com/en-gb/

[9] spec. ”SPEC CPU92 Benchmarks”. Standard Performance Evaluation Corporation. https://www.spec.org/cpu92/

[10] ALTERA. ”DE2-115 Development and Education Board”. http://www.altera.co.jp/education/univ/materials/boards/de2-115/unv-de2-115-board.html