卒業論文

題目

ヘテロジニアスマルチコア対応のキャッシュシステム自動設計ツールの改良

指導教員

佐々木 敬泰助教

2018 年

三重大学 工学部 情報工学科
コンピュータアーキテクチャ研究室

柴田 昂輝 (414836)

内容梗概

近年，高性能かつ低消費電力のマルチコアプロセッサが求められている．これに対し，異なる構成のコアを複数搭載する，ヘテロジニアスマルチコアプロセッサの開発が進められている．しかしヘテロジニアスマルチコアプロセッサは，異なる構成のコアそれぞれに設計と検証が必要である．そのため，設計労力が大きいことが問題として挙げられている．そこで当研究室では，ヘテロジニアスマルチコアプロセッサの自動設計ツールである FabHetero を提案している．FabHetero はプロセッサの構成情報を表すパラメータを基に，コア，キャッシュ，バスを自動設計するツールである．このようなツールを用いることにより，ヘテロジニアスマルチコアプロセッサの設計労力を下げることを目的としている．しかし，FabHetero 内のキャッシュを自動設計するツールである FabCache は，キャッシュの設計データがそれぞれの階層や仕様に応じた専用の設計となっているため，仕様や階層を追加するにつれて追加が必要な設計データが爆発的に増えてしまう問題がある．

そこで本研究では，キャッシュを構成する要素を細分化しそれぞれ組み合わせることで，1 つの設計データから任意のキャッシュを生成可能な汎用的なキャッシュ生成モジュールを提案し，設計労力の低減を目指す．命令キャッシュとデータキャッシュを対象に提案手法を実装し，論理合成ツールを用いて従来の FabCache と面積評価を行った結果，僅か 0.4%程度のハードウェア面積の増加で，命令キャッシュとデータキャッシュを統一的に扱える，汎用的なキャッシュ生成モジュールを実現できることがわかった．

Abstract

Recent years, high performance and low energy multi-core processors are required. For this rquirement, many researches of heterogeneous multi-core processors (HMP) are proposed. However, a HMP has divers processor cores, therefore design effort of HMP is seriously heavy. That is known as the main problem when we design HMPs. To solve this problem, FabHetero is proposed as automated design tool for HMP. FabHetero is tool-set that automatic design core, cache and bus by using configuration information parameter of generating cache. The objective of this tool is to reduce design effort of HMP. However, FabHetero have problems in cache design part. FabCache is cache automated design tool in FabHetero, but it has dedicated design data in terms of hierarchy or specification. This causes a problem that the design scale and complexity increase explisively when increasing cache level or specification. This paper proposes universal cache generating module to reduce design effort. In our proposed approach, we divide a cache system into some components which is simple and offers individual function, and the cache system is composed by these individual components.

This paper also implements proposed module capable of generating instruction cache and data cache, evaluate by comparing the area of the proposed FabCache and the conventional FabCache using logic synthesis tool. According to evaluation results, proposed module generates caches extra circuit only 0.4%. This investigate achieved to implement proposed module that can unify instruction caches and data caches.

目 次

図 目 次

表 目 次

1　はじめに

近年，マルチメディアコンテンツの高精細化やスマートフォンの普及に伴い，高性能かつ低消費電力のマルチコアプロセッサが求められている．現在主流となっているホモジニアスマルチコアプロセッサは，同じ構成のコアのみを複数搭載しているマルチコアプロセッサである．しかし，1種類のコアのみを持つホモジニアスマルチコアプロセッサでは，要求されるタスクに対して最適なリソースを割り当てることができない．そのため，タスクに対してコアの処理能力が不足，または過剰になってしまい，性能面・電力面で高いパフォーマンスを発揮することができないという問題がある．そこで，異なる構成のコアを複数搭載するヘテロジニアスマルチコアプロセッサが注目されている．ヘテロジニアスマルチコアプロセッサは，要求されるタスクに対して最適なコアを割り当てることができるため，性能と消費電力に対する要求を両立することができる．しかしヘテロジニアスマルチコアプロセッサは，それぞれのコアについて設計・検証を行う必要があるため，設計・検証の労力が大きいという問題がある．

そこで当研究室では，ヘテロジニアスマルチコアプロセッサの設計・検証の労力を低減するために，ヘテロジニアスマルチコアプロセッサを自

動設計するツールとして FabHetero[1] を提案している．FabHetero はコアを自動設計する FabScalar[2]，キャッシュを自動設計する FabCache[3]，バスを自動設計する FabBus[4] から成り立っている．本研究では，キャッシュシステムの自動設計ツールである FabCache を対象とする．FabCache は，キャッシュ容量，ラインサイズ，連想度などをパラメータで指定し，任意な構成のキャッシュシステムを自動設計するツールである．ヘテロジニアスマルチコアプロセッサでは，コアの構成が異なることから，コアに対するキャッシュシステムの構成も異なる．このため，キャッシュシステムの構成をパラメータで指定可能な FabCache を設計時に用いることで，キャッシュ設計に費やされる時間を削減し，ヘテロジニアスマルチコアプロセッサの設計労力を下げることが可能である．しかし現在の FabCache は，キャッシュの設計データが専用設計になっており，キャッシュ階層や仕様の拡張をすると設計データが爆発的に増えるため，破綻するという問題がある．これは，ヘテロジニアスマルチコアプロセッサの設計コストを下げるという FabHetero の目的を満たしていない．

そこで本研究では，設計データの専用設計を解消するために，汎用的なキャッシュ生成モジュールを提案する．汎用的なキャッシュ生成モジュールは，キャッシュの構造を要素ごとに分けて持っており，各要素をパラメー

タを用いて組み合わせることで任意のキャッシュを生成する．この提案モジュールを用いることによって，1 つの設計データから任意のキャッシュシステムを生成することが可能となり，設計データの増大を防ぐことが可能になる．

本論文は次のように構成されている．まず，次章でマルチコアプロセッサ，キャッシュシステムについての概要を述べる．次に第 3 章で関連研究として，キャッシュシステムの自動生成手法に関する研究について述べ，第 4 章で先行研究として従来の FabCache について述べる．そして，第 5 章で提案手法の汎用的なキャッシュ生成モジュールについて述べ，第 6 章で提案モジュールの性能評価を行う．

2　研究背景

本章では，FabHetero が自動設計の対象としているマルチコアプロセッサとキャッシュシステムについて述べる．また，FabHetero は高性能向けのプロセッサを自動設計することを考慮しているため，高性能プロセッサ向けのキャッシュシステムについて述べる．

2.1　マルチコアプロセッサ

マルチコアプロセッサとは 1 つのチップ上に複数のコアを搭載しているプロセッサである．複数のコアで実行プログラムの処理を分担し，並列に動作させることにより高性能化を実現している．図 2.1 の左に現在広く使われているホモジニアスマルチコアプロセッサの構成を示す．ホモジニアスマルチコアプロセッサは，同じ構成のコアのみを複数搭載するプロセッサである．設計・検証を 1 つのコアに対してするだけでよく，設計の労力が低い．しかし同じ構成のコアのみでは，それぞれのコアに対して最適なタスクを割り当てることができないという問題がある．

そこで，ホモジニアスマルチコアプロセッサに対して，ヘテロジニアスマルチコアプロセッサ (Heterogeneous Multi-core Processor:HMP) の研究が進められている．図 2.1 の右に HMP の構成を示す．HMP は異なる

構成のコアを複数搭載するプロセッサである．消費電力や処理速度などが異なるコアを使い分けることで，実行タスクの処理に応じた最適なコアを割り当てることが可能である．そのためHMPは，ホモジニアスマルチコアプロセッサと比べて，高性能と低消費電力を実現可能である．しかし，複数の異なる構成のコアを搭載する場合，それぞれのコアに対して設計・検証が必要となる．またキャッシュシステムについても，キャッシュの構成がコアの用途や構成に応じて変化するため，設計・検証が複雑化している．

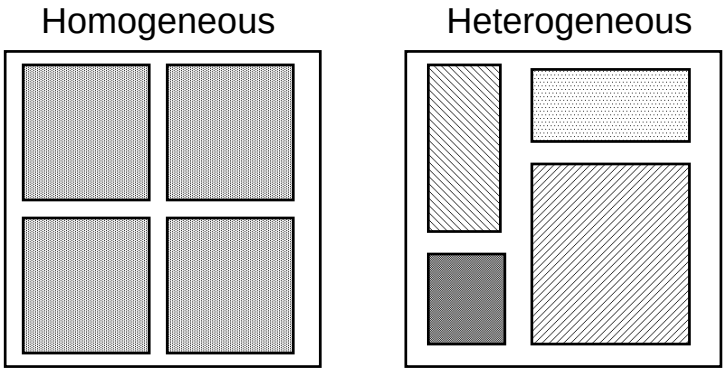

図 2.1: ホモジニアス及びヘテロジニアスマルチコアプロセッサ

2.2 キャッシュシステム

通常，プロセッサは動作に応じて必要な命令やデータを主記憶から読みだして処理を行う．しかし，主記憶からデータを読み出す処理は他の

動作に比べると非常に遅く，プログラム実行時のボトルネックになることが多い．そのためプロセッサと主記憶の間に，小容量で高速な記憶装置であるキャッシュメモリを挿入することによって，プロセッサの処理を高速化している．図 2.2 にキャッシュシステムの例を示す．これは，2 階層のキャッシュからなるプロセッサである．CPU がメモリアクセスの命令を実行する際に必要なデータがキャッシュに格納されていない場合，主記憶からそのデータを読み出すと同時に，周辺のデータも合わせてキャッシュに格納する．この時キャッシュへは，キャッシュラインと呼ばれる単位でデータを格納する．キャッシュは，メモリアクセスの命令が実行されるごとに，要求データを持っているかどうかを調べる．要求データがキャッシュメモリ内に存在していれば (キャッシュヒット)，キャッシュからデータを読み出し，データがキャッシュ内に存在していなければ (キャッシュミス)，メインメモリないし下位メモリへとデータ要求を送る．このようにキャッシュからデータを読み出すことで主記憶へのアクセス回数を減らし，プロセッサの処理を高速化している．

2.2.1　インタリーブキャッシュ

インタリーブキャッシュは主記憶から読みだしたデータを，キャッシュメモリを複数に分割したバンクへ，キャッシュライン単位で分けて格納す

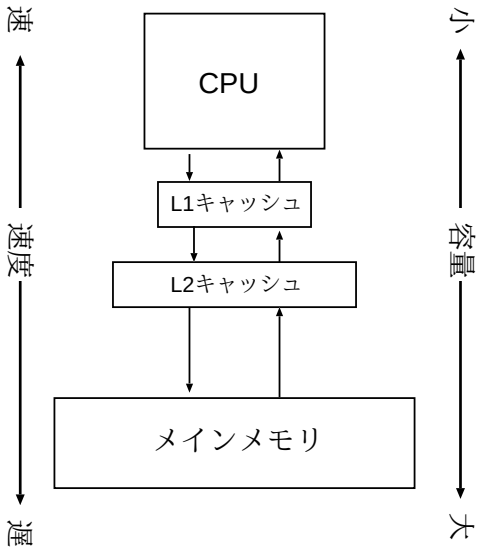

図 2.2: キャッシュシステムの例

る手法である．データを複数のバンクに分けて格納することにより，それぞれのバンクに対して同時にアクセスすることが可能になる．図 2.3 に 2 バンクのインタリーブキャッシュを示す．a から p は 1 つのデータを意味しており 1 ラインにつき 4 つのデータが格納されている．通常のキャッシュメモリでは c,d,e,f のようなキャッシュラインをまたぐ読み出しは 2 サイクルかかってしまう．2 バンクのインタリーブキャッシュは，キャッシュメモリを偶数ラインを格納する偶数バンクと奇数ラインを格納する奇数バンクに分けることで，キャッシュラインをまたぐようなデータアクセス要求が来た場合に，通常のキャッシュでは 2 サイクル必要とするところを 1 サイクルで読み出すことを可能にしている．

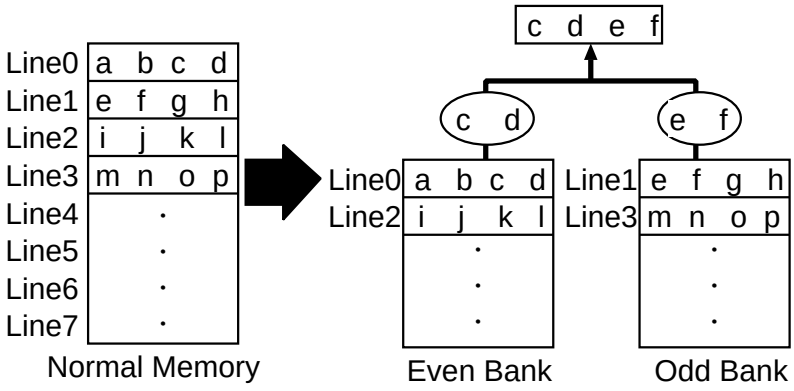

図 2.3: インタリーブキャッシュの例

2.2.2 ノンブロッキングキャッシュ

ノンブロッキングキャッシュはキャッシュミスが発生した場合の，ミスペナルティを少なくする機構を持つキャッシュである．図2.4にノンブロッキングキャッシュの例を示す．リクエストバッファから受け取ったロード・ストア命令を，タグ制御部，データ制御部に渡してキャッシュへ処理を行ない，結果を CPU へと渡している．キャッシュへのアクセス時にキャッシュミスが発生した場合，ミス情報保持レジスタ (Miss Status Holding Register:MSHR) にミス時の情報を保持し，MSHR のエントリに空きが無くなるまで後続命令によるキャッシュアクセスを許す．これにより，キャッシュミスが発生するごとにプロセッサの動作を止める (ストールする) 必要がなくなる．よって，先行命令がメインメモリにデータを読み出して

いる間に，後続命令がキャッシュを利用できるため，キャッシュミスによるペナルティが隠ぺい可能になり，スループットを上げることが可能である．

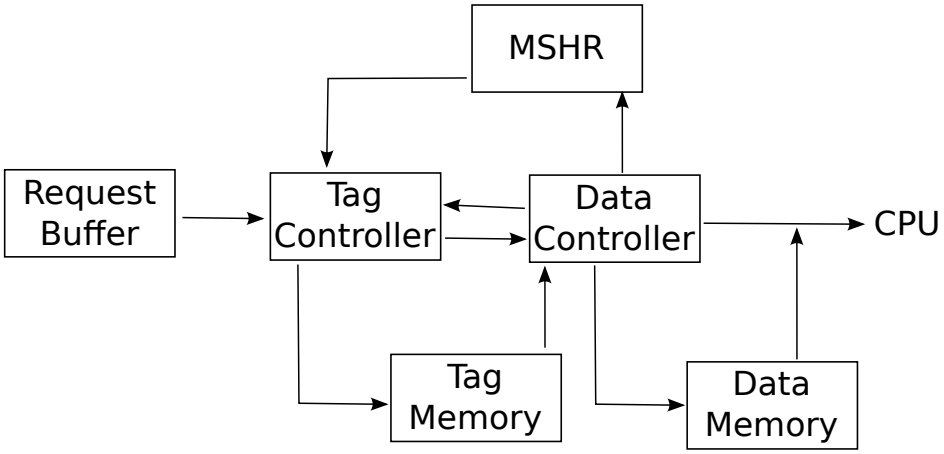

図 2.4: ノンブロッキングキャッシュの例

3　関連研究

最適なキャッシュシステムは，プロセッサで実行されるアプリケーションに大きく依存する．そのため，キャッシュシステムはプロセッサに依存した専用設計であることが多い．最適なキャッシュシステムを設計する場合，キャッシュの階層や容量に応じた，面積，消費電力など，複数の重要な項目の変更を繰り返しながら設計を行う必要がある．そのため，これらの項目は可変に設定可能である方が設計効率が高くなる．このようなパラメータによる可変構成のキャッシュ設計ツールとして，FPGA 用のパラメータ化されたキャッシュジェネレータ [5] や，商用プロセッサである LEON4 に搭載されているキャッシュシステム [6] などがある．しかし前者は，キャッシュ容量やデータ幅などは可変になっているが，生成できる階層や連想度が決まっているという制限がある．後者は，連想度は可変になっているが，対象としているプロセッサが固定となっているという制限がある．加えて，これらのツールは命令フェッチ幅が可変となっておらず，後者は特定のプロセッサにしか対応していない．そのため，異なるプロセッサを持つ HMP には対応できないという問題がある．

4 FabCache

FabCache は，第 3 章で述べた手法とは異なり，HMP に対応したキャッシュシステム自動設計ツールである．本章では FabCache の概要，実装戦略を述べた後，FabCache の問題点について述べる．

4.1 概要

FabCache はキャッシュ設計者が作成したパラメータに応じて，任意のキャッシュシステムを自動設計するツールである．様々な構成のコアに対応するために，キャッシュ容量，命令フェッチ幅，ラインサイズ，連想度，異なるキャッシュ階層間のインターフェース等，様々な構成要素をパラメータで指定できる．また FabCache は，高性能かつ低消費電力の高性能プロセッサに組み込むことを想定しているため，生成するキャッシュシステムには，第 2.2.1 項で述べたインタリーブキャッシュや，第 2.2.2 項で述べたノンブロッキングキャッシュのように，高性能プロセッサ向けのキャッシュシステムも生成する対象に含めている．図 4.5 に，FabCache によるキャッシュ設計の流れを示す．図 4.5 では，1 階層の命令キャッシュとデータキャッシュの設計を行っている．まず，容量や連想度といった設計対象のキャッシュ構造の情報をパラメータファイルとして与える．FabCache は

受け取ったパラメータをもとに，ハードウェア記述言語の System Verilog で記述された，1 階層の命令キャッシュとデータキャッシュの設計データを用いて，キャッシュを自動設計する．このように，パラメータファイルと設計対象の設計データを組み合わせて，様々な構成のキャッシュを自動設計している．

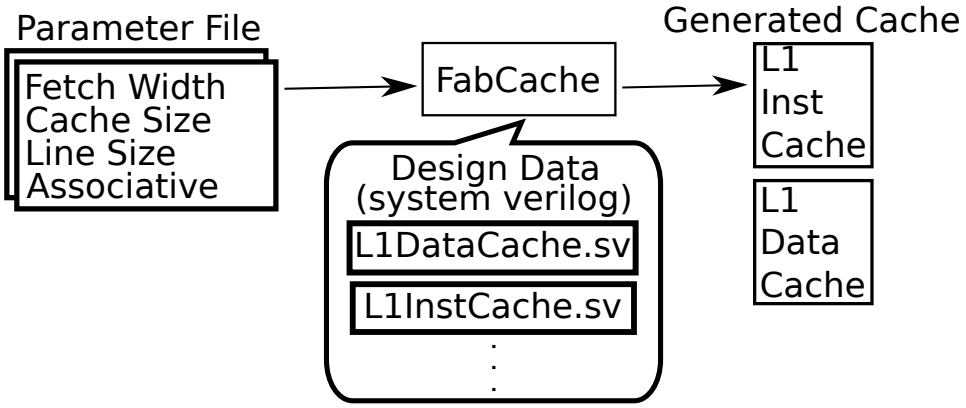

図 4.5: FabCache によるキャッシュ設計の流れ

4.2 スーパーセット戦略

従来の FabCache は，生成可能なキャッシュとキャッシュ構成のスーパーセットを持っており，スーパーセットの構成からパラメータにより，キャッシュの構成を選択して，様々なキャッシュシステムを設計可能にしている．図 4.6 にスーパーセット戦略によって設計したキャッシュシステムの例を示す．Core1〜Core3 では L1 キャッシュが有効化されており，Core4 では

L1 キャッシュが無効化されている．このようにキャッシュシステムのスーパーセットに対して，キャッシュの有効・無効をパラメータで指定することで，キャッシュ構成を変更している．

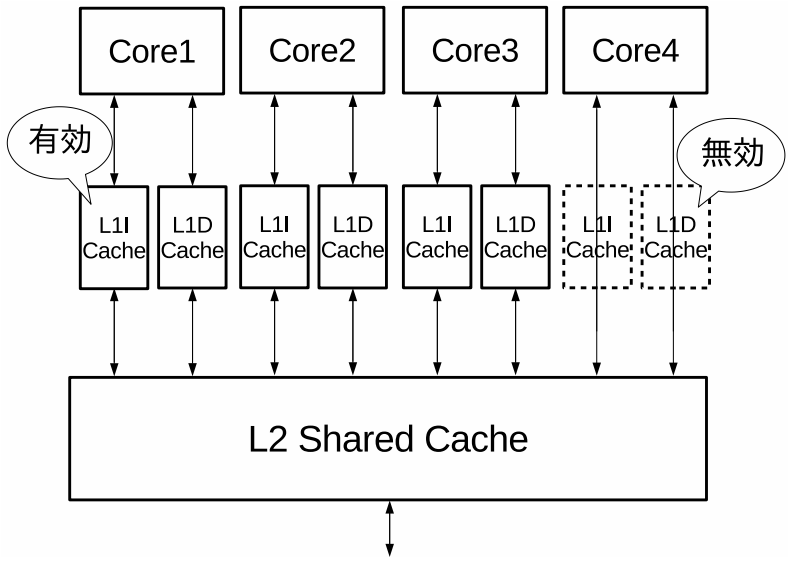

図 4.6: スーパーセット戦略の例

4.3 FabCache の問題点

従来の FabCache は，キャッシュ設計のために複数の設計データを持っており，それぞれ専用設計になっている．そのため，新たな階層の追加や仕様拡張時の設計労力が大きい問題がある．図 4.5 で示した命令，データキャッシュに対して，仕様や階層を拡張する流れを図 4.7 を用いて説明する．キャッシュの設計データは，L1 のデータキャッシュや，L1 の命令

キャッシュのように階層や用途に応じた専用設計になっている．まず，命令，データキャッシュに仕様 A，B をそれぞれ追加する場合，設計データは 4 種増加する．次に，仕様 A，B をどちらも使用する仕様 AB のキャッシュを追加する場合は 2 種増加する．その後，キャッシュ階層を追加する場合は，6 種の設計データが増加してしまう．このように，仕様や階層を追加するごとに，設計データ数は指数関数的に増加する．一般に，似た設計データを何度も記述する場合には，設計データの流用により設計労力を削減可能である．しかし，キャッシュの階層や仕様が異なる場合，キャッシュの振る舞いや構造が大きく変化するため，単純な設計データの流用は困難である．そのため，現状の仕様では任意構成のキャッシュ設計へ対応する場合，非現実的な労力が必要となる．

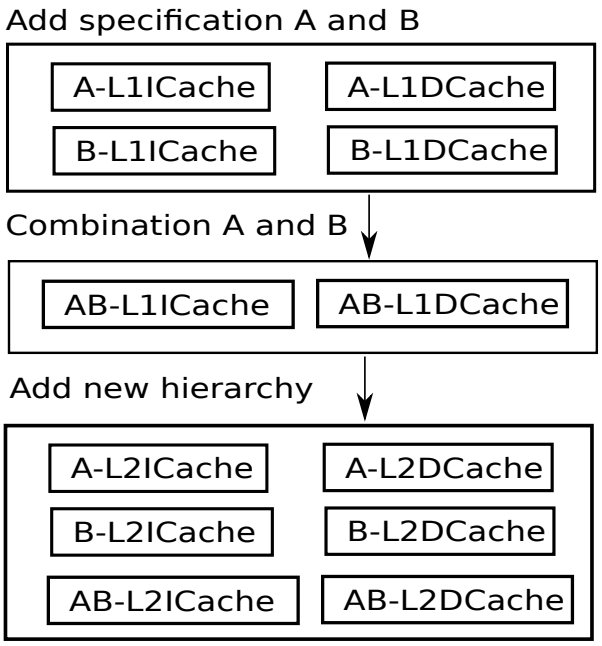

図 4.7: 仕様と階層追加による設計データの増大例

5　汎用的なキャッシュ生成モジュールの提案

本研究では，第 4.3 節で述べた問題点を解決し，設計の労力を下げるために，汎用的なキャッシュ生成モジュールを提案する．提案モジュールは，パラメータに応じて任意のキャッシュを生成するために，タグメモリやデータメモリといった基本的に共通する構造と，メモリインタリーブやノンブロッキングキャッシュのような固有の構造を細分化した上で整理した，キャッシュの構成要素を持っている．その構成要素を実装時に決定するパラメータから動的に生成することで，単一のモジュールから任意のキャッシュを自動生成可能にする．図 5.8 は，提案モジュールを用いたキャッシュの生成例である．キャッシュを構成する要素として，キャッシュに必須であるタグ部とデータ部の他に，データの一貫性を保つコヒーレンシ手法がＡとＢの２種類実装されている．さらに，命令，データ，共有キャッシュそれぞれの独自仕様が記述されている．この汎用キャッシュモジュールに，L1 命令キャッシュ，L1 データキャッシュ，L2 共有キャッシュを生成するためのパラメータを渡すことで，必要な構成要素を有効化し，キャッシュを生成する．このように，提案モジュールは様々なキャッシュの構成要素を組み合わせることで，任意のキャッシュを１つの設計データから自動生成している．また，新しい仕様を追加する場合，新仕様の設

計データを 1 つ追加するだけで，他の設計データと組み合わせることが可能になり，設計データの爆発的な増加を防ぐことが可能である．

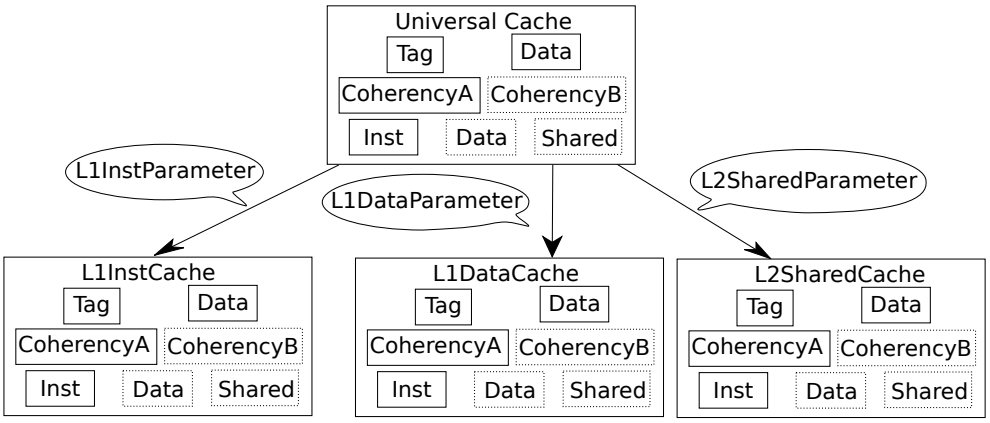

図 5.8: 汎用的なキャッシュ生成モジュールの例

以降は，提案モジュールの実装手法について述べる．提案モジュールはキャッシュの構成要素を組み合わせてキャッシュを生成するため，キャッシュの構成要素を設計する必要がある．まず各構成要素を設計するために，従来の設計データの処理を構造や機能ごとに分割する．このとき，分割した設計データの内，同じ構造や機能に対する設計データは冗長であるため共通化する．共通化することができない部分は，パラメータに応じて処理を制御する．第 5.1 項から第 5.3 項にかけて，キャッシュの構成要素を設計する詳細な手法を述べる．また本研究では，命令キャッシュとデータキャッシュの生成を対象として提案モジュールの実装を行う．

5.1　キャッシュの構成要素への分割による共通化

従来の FabCache の設計データは専用設計となっているため，これをキャッシュの構成要素とすることはできない．また，これらの設計データは専用設計となっているため，タグメモリやデータメモリといった基本的に共通する構造であっても，複数の設計データに分離しており設計効率が悪い．そのため，専用設計となっている従来の FabCache の設計データを分割し，タグメモリ，データメモリのような共通部分を，それぞれ階層に依らない 1 つの設計データに整理する．これにより，設計データの分散を防ぎ設計効率を改善することが可能である．

5.2　非共通部分のパラメータ制御

第 2.2.1 項で述べたインタリーブキャッシュや第 2.2.2 項で述べたノンブロッキングキャッシュのような，仕様の違いにより固有の構造を持つキャッシュは，共通化することができない．この場合，FabCache が受け取るパラメータの他に新たなパラメータを追加し，そのパラメータに応じて生成するかどうか処理を制御する．仕様の種類が増えるほど制御パラメータと設計データが増えてしまうが，第 5.1 節で述べた手法により，機能ごとに分割を行っているため，組み合わせによる指数関数的な設計

データの増大は発生しない．また，今後新たなキャッシュ構成の手法が考案された場合，パラメータによる制御を追加することで，自動生成の選択肢に入れることができるため，拡張性を確保することが可能である．

5.3　非共通部分の仕様差隠ぺいによる共通化

第5.2節の手法では，冗長なハードウェアは生成されないが，設計データが分離したままであり，設計効率は向上しない．そこで，設計効率を向上させるために，非共通部分の仕様差を隠ぺいし，設計データの共通化を可能にする．本研究では，命令キャッシュとデータキャッシュの仕様差の隠ぺいと，キャッシュをバンク化することによる，通常のキャッシュとインタリーブキャッシュの仕様差の隠ぺいを行う．

命令キャッシュとデータキャッシュに対する仕様差の隠ぺいについて述べる．命令キャッシュとデータキャッシュは仕様の違いから，信号線の有無によって構造が異なるため，設計データを共通化できない．命令は静的なデータであり，基本的に書き換えが発生しないため，命令キャッシュには書き込み線が存在していない．しかし，データキャッシュでは，データの書き換えが発生するため，書き込み線が存在し，命令キャッシュとは構造が異なる．そこで，命令キャッシュに書き込み線にあたるダミー線を追

加し，キャッシュ間のインタフェースを揃えることで，設計データを共通化する．追加したダミー線は命令キャッシュで使われることがないため，論理合成時の最適化によって，面積増加を最小限に抑えることが可能である．このように，命令・データキャッシュのインタフェースを揃えることで，仕様を隠ぺいしつつ設計データの共通化を可能にする．

通常のキャッシュとインタリーブキャッシュに対する仕様差の隠ぺいについて述べる．第 2.2.1 項で述べたように，インタリーブキャッシュではキャッシュが複数のバンクに分割されており，通常のキャッシュとは構造が異なるため，設計データを共通化できない．そこで，通常のキャッシュをインタリーブキャッシュの仕様と合わせるために，キャッシュ全てをバンクとみなして設計する．提案モジュールに渡すパラメータにバンクの数を追加し，その数に応じてバンクメモリを生成する．この時，バンクの数が 1 つだった場合，キャッシュを 1 つに分割するバンクとみなして，通常のキャッシュと同様の処理を行う．これにより，バンク数が変化しても共通の設計データを用いてキャッシュを生成することが可能となり，設計データを共通化することが可能になる．

6　性能評価

性能評価では，従来の FabCache と提案モジュールを実装した FabCache を比較する．評価内容は，提案モジュールを実装したことによって変化した FabCache のハードウェア面積と，設計データの共通化によって変化した FabCache 及びその下位モジュールのソースコード行数である．ハードウェア面積については，Synopsys 社の論理合成ツールである Design Compiler を用いて論理合成を行い，ゲートレベルでのハードウェア面積を比較する．表 6.1 に生成したキャッシュの構成を示す．

表 6.1: 生成キャッシュの構成

	キャッシュ容量	連想度	ラインサイズ	インタリーブ
L1 命令キャッシュ	32KByte	4way	16Byte	有効
L1 データキャッシュ	32KByte	4way	16Byte	無効

6.1　評価結果

表 6.2 に評価結果を示す．従来の FabCache のハードウェア面積と提案モジュールを用いた FabCache のハードウェア面積を比較した結果，ハードウェア面積が 0.4%増加した．これは FabCache 全体の面積と比べると無視できるほど少ない増加量であるため，少ないハードウェア量の追加

で提案モジュールの実装に成功したと言える．生成するキャッシュの構成を変えた場合であっても，SRAM で構成されるキャッシュの方が占める面積の割合が大きいため，ハードウェア面積の増加に大きな影響はない．また，設計の効率を上げるという観点から，FabCache 及び FabCache の下位モジュールのコード行数についても評価した．従来の FabCache と同様のコーディング規則に従って，構成要素の共通化などを行った結果，設計データの行数が 2.5%減少した．この削減率は全体のコード行数と比較して多いとは言えない．しかし，提案モジュールの各構成要素は，データ要求部やキャッシュ制御部などを分割，共通化する余地を残している．そのため，共通化部分を更に増やすことで，20〜25%のコード削減率を見込むことができる．

表 6.2: ハードウェア面積およびコード行数の評価結果

	ハードウェア面積 (μm^2)	コード行数
従来手法	1,458,678	7,034
提案手法	1,464,519	6,858

7　結論

本研究では，ヘテロジニアスマルチコアプロセッサ対応のキャッシュシステム自動設計ツールである FabCache の設計効率を向上させるために，汎用的なキャッシュ生成モジュールの提案と，命令キャッシュとデータキャッシュを対象とする提案モジュールの実装を行った．これにより，仕様追加による設計データ数の指数関数的な増大を防ぐことに成功した．提案モジュールを実装した結果，ハードウェア面積の増加を 0.4%に抑えて実装することに成功した．これは，FabCache 全体に対して非常に小さいため，無視できる値となった．コード行数を従来の FabCache と比べて約 2.5%削減できたことから，設計効率の向上が確認でき，提案手法にのっとって設計を進めることで、更なる設計効率の向上が可能であることを示した．これらの結果から，キャッシュを統一的に扱うことができるフレームワークの実装に成功したと言える．今後の展望として，更なるキャッシュ構造共通化手法の考案と生成対象の拡張が挙げられる．

謝辞

本研究を行うにあたり，多数の助言を頂きました近藤利夫教授，深澤研究員，並びにご指導を頂きました佐々木敬泰助教に深く感謝いたします．

コンピュータアーキテクチャ研究室の院生・学生のメンバーには常に刺激的な議論を頂き，精神的にも支えられました．また，本研究はSynopsys社CADツールによる東京大学VDECの支援により実施されたことを並びに感謝します．

参考文献

[1] T. Nakabayashi, et. al.“FabHetero: An Environment for Developing Diverse Heterogeneous Multi-core Processors.” ISSEMU, Nov 2012.

[2] N. K. Choudhary，et．al．”FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template.”ISCA-38, pp.11-22, June 2011.

[3] T. Okamoto et. al. ”FabCache:Cache Design Automation for Heterogeneous Multi-core Processors.” CANDAR, pp.602-606, Dec 2013.

[4] Y. Seto, et. al. ”FabBus: A Bus Framework for Heterogeneous Multi-core processor.” ITC-CSCC2013, pp.254-257, July 2013.

[5] P. Yiannacouras and J. Rose. "A Parameterized Automatic Cache Generator for FPGAs" FPT, pp.324-327, Dec. 2003.

[6] "Leon4 Processor" http://www.gaisler.com/index.php/ products/processors/leon4 (accessed 2018-2-7)