Vol.2019-ARC-235 No.7
Vol.2019-SLDM-187 No.7
Vol.2019-EMB-50 No.7
2019/3/17

マルチプロセッサにおける並列実行プログラムの共有キャッシュによる実行時間への影響度の見積もり手法

婁 宜之[1,a)]　本田 晋也[1]　曾 剛[1]　高田 広章[1]

概要：近年，組込みシステムにおいて，チップマルチプロセッサ（CMP，Chip Multiprocessor）が普及してきている．CMP では，コヒーレンシを維持するため，共有キャッシュを搭載することが一般的である．しかし，共有キャッシュの競合によるキャッシュミスの増加とともに，プロセッサ全体の性能が劣化することも多い．そのため，プロセッサ間の負荷分担の最適化などのために，この性能劣化を数値化して評価することが重要である．PC 向けの数値化評価手法はすでに提案されているが，組込み向けの手法は少ないという現状がある．本研究は，並列実行による性能劣化を予測する Intel プロセッサ向けの既存手法を 64 ビットの ARM プロセッサ（AArch64）を対象として再現，評価を行った．本研究で，再現した手法によって，AArch64 プロセッサを対象として並列実行した時の実行時間の予測誤差は 2 コアで 1.79%，3 コアで 3.22%，4 コアで 6.00%であった．

キーワード：AArch64，チップマルチプロセッサ，共有キャッシュ，見積もり

Estimation Method of Shared Caches' Effects on Co-run Execution Speed in Multi-Core Processors

Yizhi Lou[1,a)]　Shinya Honda[1]　Gang Zeng[1]　Hiroaki Takada[1]

Abstract: In recent years, embedded systems are commonly designed with Chip Multiprocessors (CMP). In CMP, sharing of last-level caches is demanded to provide coherency among processors. The sharing can cause performance degradations due to contentions in the shared caches . Estimations are needed to reduce the degradation. For now, there are few such estimation methods for embedded systems, despite those for PCs. In this research, we implemented existing estimation methods of PC's, specifically of Intel x86-64 proseccors(x64), in 64-bit ARM processors(AArch64), which are widely used in embedded systems. We successfully implemented an estimation method, and estimated performance of co-run programs in 2, 3 and 4 core AArch64 CMP environment, resulting an estimation error of only 1.79%, 3.22% and 6.00%.

Keywords: AArch64, Chip Multiprocessor, Shared Caches, Estimation

1. はじめに

近年，計算機の計算性能を上げるために，マルチコアアーキテクチャが多く採用されている．さらに，組込みプロセッサにおいては，PC よりもマルチプロセッサを利用する傾向が強まっている．

マルチコアプロセッサは，コヒーレンシを維持するため，共有キャッシュを使用することが一般的である．しかし，共有キャッシュへのアクセスによって生じる競合により，マルチコア化しても計算性能が上がらない場合がある．これは，共有キャッシュはコア間で共有され，複数コアからのアクセスによって上書きされることが頻繁に発生する可能性があることから，コアごとに独立してキャッシュを持

[1] 名古屋大学
Nagoya University
[a)] louyz@ertl.jp

つ場合よりも，ミスが起きやすい傾向があるためである．

ここで，スケジューリング，最適化などの手段によって，共有キャッシュでの競合を低減することにより，このような問題を軽減することが可能である．また，共有キャッシュの競合による，システムの性能劣化の数値化およびその解析と評価に関する研究も多くなされている．例として，Jiang ら [2] の手法では，キャッシュに対する動作の見積もり結果を考慮してスケジューリングすることで，プログラムの平均性能劣化を 34%削減することが可能であった．

Xu ら [1] は並列実行時の競合により共有キャッシュのミス率と実行時間の影響度を見積もる手法である CAMP を提案した．ただし，これらの既存手法は，すべて PC 向けであり，Intel 社のプロセッサ（x86 ないし x64）かつ 2 コアの環境を想定している．その上，組込みシステムで広く用いられている ARM プロセッサ（AArch32 ないし AArch64 と呼ぶ）向けの研究は存在しない．

近年，PC（x64）で開発、検証されたプログラムを組込みシステム（AArch64）で使用することが多い．しかし，同じプログラムであっても，プロセッサのアーキテクチャが異なれば，単独実行性能も，並列実行による性能劣化も異なる．このため，x64 でのプログラムの性能評価の結果は，AArch64 で直接適用できない．さらに，x64 向けの評価手法を AArch64 で応用できるかどうかも未知である．これらの手法を利用するため，性能に関わるパラメータが必須である．ただし，プロセッサの差異により，評価時に必要なパラメータが測定できるかどうかも未知である．

本研究では，組込み向けの AArch64 で利用される共有キャッシュによるプログラムの実行時間の劣化を評価する手法を実現する．本研究の評価方法は，異なるプログラムを異なるコアの配置して実行する時のキャッシュの競合によるお互いの実行時間の影響度を実際に並列実行することなく計算することである．また，3 コア，4 コアなど組込みで普及しているマルチプロセッサ環境への適応も行う．

CAMP[1] は x64 プロセッサでは予測精度が非常に高いため，この手法を AArch64 の環境で再現できるかどうかを確かめる．そして，再現可能であれば 2，3，4 コアのプロセッサで実際に評価する．

2. 既存研究

Subramanian ら [3][4][5] は，並列実行プログラムのメモリと共有キャッシュによる性能劣化の評価手法を提案した．これらの手法による性能劣化の予測誤差は 9%程度である．ただし，これらの手法は，CPU で計測用のレジスタを多数追加することが必須であるため適用は難しい．Tam ら [6] が提案した手法では，プログラムが利用できるキャッシュのサイズと性能の関係を評価し，その評価結果を用いてプログラムの共有キャッシュのパーティションを行う．この手法は，キャッシュをリソースとして効率的に利用することを可能とする．ただし，この手法はメモリ，キャッシュの割り当てアルゴリズムを変更しなければならないため適用が困難である．

また，ハードウェアを変更する必要がなく，ソフトウェアで軽量的に再現できる手法として，Jiang ら [2] は CAPS（Cache-Contention Aware Proactive Scheduling）を提案した．CAPS はプログラムの共有キャッシュに対する動作傾向を評価し，その評価を参照して並列実行のスケジューリングを最適化する手法である．CAPS により，PC で実行するプログラムの平均性能劣化を 34%削減することが可能であった．Xu ら [1] が提案した CAMP（Cache Aware Performance Model for Multi-core Processors）では，並列実行プログラムの性能（ミス率，実行時間など）を予測可能である．CAMP による PC で並列実行の実行時間の予測誤差は 1.57%である．

本研究では，CAPS と CAMP において AArch64 上で再現可能かどうかを検討した．CAPS は，見積もりするために細かい粒度のパラメータを測定することが必須である．Intel 社のプロセッサでは，Intel PIN[*1]というツールで測定できるが，AArch64 の実験環境と測定ツールでは再現できなかった．本研究では，残りの CAMP を AArch64 へ適用することについて検討する．

2.1 CAMP

2.1.1 概要

CAMP は，並列実行プログラムの性能に対する予測手法である．同じプログラムであっても，別のプログラムと並列実行される際，その組合せが異なると，並列実行時の性能が変化する．例として，図 1 に示すのは，異なるプログラム A-E の中から 3 つ選んで 12-Way のセットアソシアティブ共有キャッシュを持つハードウェア上で並列実行させる時，各プログラムの平均的に利用できるキャッシュのサイズである．B と D のキャッシュに対する動作傾向が異なるため，それらと並列実行する際，A と C の平均的に利用できるキャッシュの量が変化する．さらに A と C を，新しいプログラム E と並列実行する場合，A，C の平均的に利用できるキャッシュの量は B と D と並列実行した場合と異なっている．このような並列実行する時に利用できるキャッシュの平均量が分かれば，ミス率，実行時間といった，それぞれのプログラムの性能を見積もることができる．

CAMP は，このような他のプログラムの影響と実行環境（キャッシュのサイズ，Way 数など）を統合的に考慮し，プログラムの並列実行の性能を予測する手法である．

CAMP の原理を図 2 に示す．各のプログラムを，ストレスマーク（Stressmark）というプログラムと並列実行することで性能パラメータを測定する．すべてのプログラム

[*1] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

Vol.2019-ARC-235 No.7
Vol.2019-SLDM-187 No.7
Vol.2019-EMB-50 No.7
2019/3/17

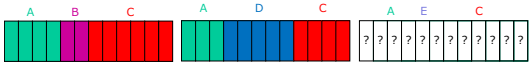

図 1　キャッシュの平均利用量の変化

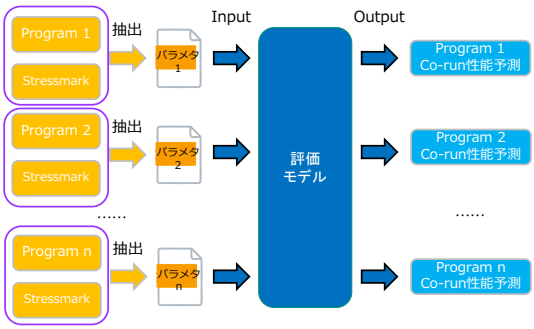

図 2　CAMP の原理

のパラメータの測定が完了した時，それらを使用して各プログラムの並列実行時の性能を計算する．

2.1.2　ストレスマーク

ここでは，インプットとしてのパラメータを測定するために使用するストレスマーク（Stressmark）を説明する．ストレスマークとは，プログラムと並列実行しながらストレス（競合）を与えるプログラムである．CAMP で使用したストレスマークは，指定した範囲内で繰り返しキャッシュにアクセスするプログラムである．つまり，その範囲のキャッシュが，ストレスマークに独占されることになり，ほかのプログラムが実際に利用できるキャッシュは，ストレスマークが使用しない部分だけになる．CAMP はセットアソシアティブ方式のキャッシュ向けであるため，「Way」の粒度でキャッシュのサイズを議論する．CAMP で使用されるストレスマークも，Way を粒度としてアクセスするキャッシュのサイズを指定できる．

本研究で使用した Cortex-A57 プロセッサ（第 3.1 節で説明する）を例として説明する．A57 の共有キャッシュは，16 Way セットアソシアティブである．図 3 に示すように，ストレスマークがその中の 15 Way を繰り返して使用することになると，ストレスマークと並列実行されるプログラムが利用できるのは 1 Way のみである．ストレスマークが 14 Way を利用すると，並列実行されるプログラムが利用できるのは 2 Way である．他の状況もこれより推定できる．つまり，ストレスマークが利用するキャッシュのサイズを調整することで，並列実行されるプログラムが利用できるキャッシュサイズをコントロールすることができる．このような手法により，プログラムが各キャッシュサイズで実行時の性能パラメータが測定できる．

2.1.3　インプット

インプットの測定について 16 Way のキャッシュを例として説明する．0 Way から 16 Way までのキャッシュを利用するストレスマークと各プログラムを並列実行させ，各 Way 数でのプログラムのパラメータを測定する．測定した

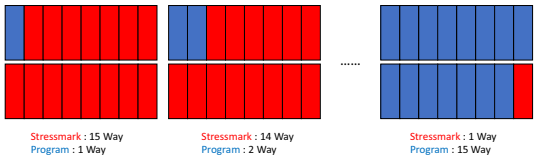

図 3　ストレスマークの機能

パラメータは，以下の 4 つである．

(1) 17 組の MPA（それぞれ，0 Way-16 Way のキャッシュが利用できる場合の値）
(2) 17 組の SPI（それぞれ，0 Way-16 Way のキャッシュが利用できる場合の値）
(3) α と β
(4) API（平均値）

CAMP のインプットのパラメータとしては，まず各 Way 数で実行する時の MPA（Miss per Access，共有キャッシュでのミス率）と，SPI（Second per Instruction，命令ごとの実行時間）が挙げられる．さらに，同じプログラムの MPA と SPI の間には，式（1）のような線形関係がある．この線形関係は，各プログラムに対して固定される．つまり，同じプログラムであれば，実行するハードウェアが変わらない限り，α と β は変化しない．よって，この線形関係を表すパラメータ α と β を求める必要がある．最後に，API（Access per Instruction，命令ごとのキャッシュをアクセスする回数）も必要である．

$$SPI = \alpha * MPA + \beta \tag{1}$$

2.1.4　アウトプット

CAMP で計算する性能指標は，以下の順序で計算される．

(1) 有効キャッシュサイズ（Effective Cache Size，ECS）
(2) 共有キャッシュでのミス率（Miss per Access，MPA）
(3) 命令ごとのかかる時間（Second per Instruction，SPI）

CAMP では，まず ECS が計算できる．ECS は，プログラムが実行中，平均的に利用できるキャッシュのサイズである．つまり，図 1 のように，並列実行時の各プログラムの利用できるキャッシュのサイズである．ECS がわかれば，プログラムのキャッシュミス率（MPA）を計算できる．MPA がわかれば，SPI は式（1）で計算できる．

3.　共有キャッシュの影響調査

本章は，x64 と AArch64 での共有キャッシュの影響度の違いについて調査した．

3.1　実験環境

3.1.1　ハードウェア

本研究で使用したハードウェアプラットフォームを説明する．コア数，OS などのソフトウェア環境，利用可能な計測ツールなどを考慮した上，本研究で利用したのは NVIDIA 社の GPU 開発ボード Jetson TX2 である．

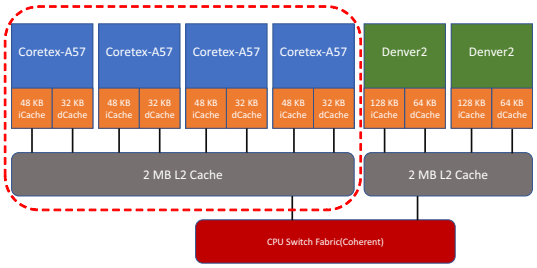

図 4　Jetson TX2 のプロセッサ構成

表 1　本研究のハードウェアのスペック

項目	Jetson TX2	Diginnos VF-HE10
CPU	Quad ARM® A57	Core i7 6500U(2 cores)
共有 LLC キャッシュ	2MB	4MB
メモリ	8GB LPDDR4	8GB PC3-12800
ディスク	32GB eMMC	500GB SATA
OS	Ubuntu 16.04.5 LTS	Ubuntu 16.04 LTS
カーネル	4.4.15-tegra	4.4

Jetson TX2 には，6 コアの AArch64 を搭載している．その中の 4 コアは ARM Cortex-A57，2 コアは Denver2 である．Jetson TX2 のブロック図を図 4[*2]の示す．

本研究では，後述する測定ツールの関係や 4 コアでの評価を実施するため，Cortex-A57 を対象に評価を行った（図 4 の赤い枠の中の部分）．

さらに，比較として PC でも比較実験を行った．利用した PC は，Diginnos VF-HE10 である．

Jetson TX2 と Diginnos VF-HE10 のスペックを表 1 に示す．

3.1.2　測定ツール

本研究では，性能プロファイリングツール perf[7] を利用した．perf は，Linux の性能プロファイリングツールである．perf を通じてプロセッサのパフォーマンスカウンタ（Performance Counter）レジスタから，主に性能と関わるパラメータが利用できる．本研究で perf を使用して取得したのは，プログラムの実行時間，サイクル数，命令数，L2 キャッシュのアクセス回数とミス回数である．

パフォーマンスカウンタとは，計算機の実行時性能を測定するためのハードウェアで，実装するかどうか，いくつ実装するかというのはプロセッサに依存する．Denver2 プロセッサは一部のパフォーマンスカウンタレジスタ（共有キャッシュのアクセス，ミス回数）を搭載していないため，perf で共有キャッシュのアクセス回数，ミス回数などのパラメータを測定することができない．Cortex-A57 プロセッサでは本研究で必要なパフォーマンスカウンタレジスタをすべて搭載しており，perf で測定することができる．このため，本研究で利用したのは Cortex-A57 のみである．

*2 https://devblogs.nvidia.com/jetson -tx2-delivers-twice-intelligence-edge/

表 2　使用した SPEC CPU 2017 ベンチマーク

番号	名前	ディスクリプション
505	mcf	停車場スケジューリング問題
520	omnetpp	ネットワークシミュレーター
531	deepsjeng	チェスの策略計算
541	leela	碁の策略計算
548	exchange2	数独の計算
507	cactuBSSN	アインシュタイン方程式の計算
508	namd	生体分子シミュレーション
510	parest	生物医学の 2D/3D 画像変換
519	lbm	無圧縮流体の 3D シミュレーション
521	wrf	天気予測システム
527	cam4	大気モデル
544	nab	分子動力学のシミュレーション
549	fotonik3d	光子の導波路の計算

3.1.3　評価用プログラム

ここでは，評価対象のプログラムを説明する．本研究で並列実行の性能劣化を評価するために SPEC CPU 2017[8] ベンチマークを用いた．これは，SPEC（Standard Performance Evaluation Corporation）が開発した CPU 性能を評価するベンチマークである．SPEC CPU 2017 には，43 種類のベンチマークがある．本研究は，コンパイル可能であったベンチマークの中で実行時間を考慮した結果，13 種類（整数計算 5 種類，浮動小数計算 8 種類）をランダムに選んで評価することにした．選んだ 13 種類のベンチマークの詳細は表 2 の示すようである．本論文では以降，5 からはじまる 3 桁の数字（505 など）でベンチマークを表示する．

3.2　評価手法

本研究では，ベンチマークを単独実行/並列実行し，それぞれの実行時間を比較した．

ここで，単独実行とは，他のコアが空いている状態での実行である．並列実行とは，他のコアにでもベンチマークを実行させた状態での実行である．プログラムのコアへの配置は，taskset 命令で実現する．

Diginnos VF-HE10 では，ベンチマーク 549 が動作できないため，ここでは他の 12 種類のベンチマークを測定した．さらに，各並列実行の組合せに対してそれぞれ 25 回測定して平均値を求めた．

3.3　Cortex-A57 と i7-6500U 上での評価結果

評価結果をそれぞれ図 5 と図 6 に示す．性能劣化は，単独実行時と並列実行時に測定した実行時間を用いて式（2）により計算する．

$$Degradation[\%] = \frac{(time_{co} - time_{single})}{time_{single}} * 100 \quad (2)$$

結論として，プロセッサのアーキテクチャが異なれば，同じ並列実行の組合せであっても，プログラムの性能劣化

Vol.2019-ARC-235 No.7
Vol.2019-SLDM-187 No.7
Vol.2019-EMB-50 No.7
2019/3/17

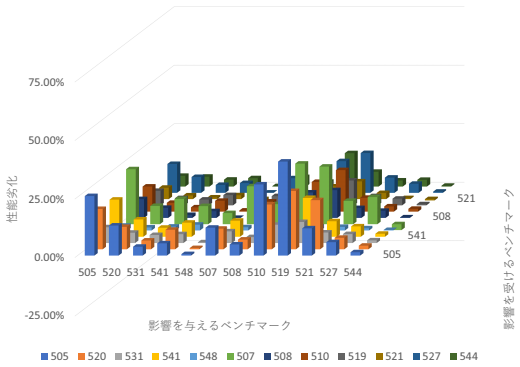

図 5　Cortex-A57 の性能劣化

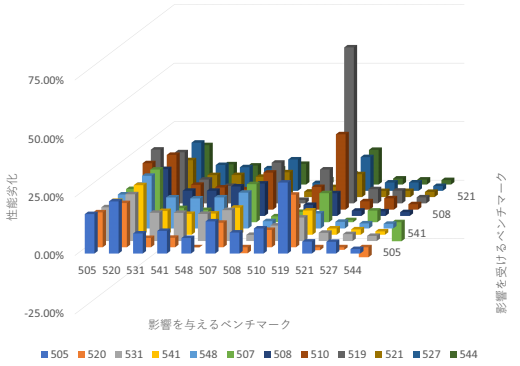

図 6　i7-6500U の性能劣化

の度合いが異なる．

4. AArch64 における CAMP の実現

本章では，AArch64 における CAMP の実現と，2 コアの環境での評価結果を説明する．

4.1 実装

4.1.1 インプット

第 2.1.3 節において，インプットのパラメータが MPA，SPI，API と α と β であると説明した．ここでは，これらのパラメータを実際に測定する方法を説明する．

実際に測定したパラメータは，以下の 4 つである．ストレスマークを使用し，プログラムによって利用されるキャッシュが 0 Way から 16 Way までの状況において，perf でこの 4 つのデータを各 17 組測定する．（以降の説明は，全部 16 Way のキャッシュを想定しているため，Way 数は 0 Way から 16 Way までとする）

(1) 実行時間（TIME[0,1,...,16]）
(2) 実行された命令数（INSTRUCTION[0,1,...,16]）
(3) 共有キャッシュでのアクセス回数（ACCESS[0,1,...,16]）
(4) 共有キャッシュでのミス回数（MISS[0,1,...,16]）

これらのパラメータによって，CAMP の計算モデルのインプットとして利用するために MPA，SPI，API と α と β を計算する．MPA，SPI と API の計算方法は，下記の式

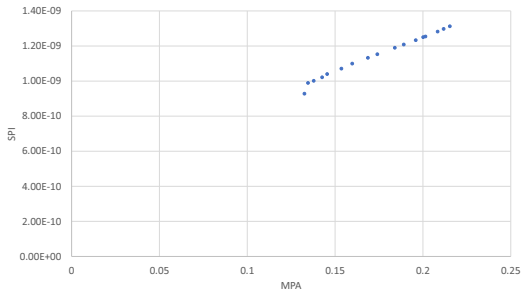

図 7　ベンチマーク 505 の SPI-MPA

(3)，(4)，(5) である．MPA と SPI は，それぞれ 17 組が必要である，API は，17 組の平均値を求める．

$$MPA_i = \frac{MISS_i}{ACCESS_i}, i = 0, 1, ..., 16 \quad (3)$$

$$SPI_i = \frac{TIME_i}{INSTRUCTION_i}, i = 0, 1, ..., 16 \quad (4)$$

$$API = \frac{\sum_{i=0}^{16} \frac{ACCESS_i}{INSTRUCTION_i}}{17} \quad (5)$$

さらに，17 組の MPA と SPI を用いて線形回帰により式 (1) を満たす α と β を求めることができる．

ここまでは，1 つのプログラムに対して CAMP を実現するために必要となるインプットの測定方法である．測定できた MPA と SPI は，図 7 の示すようである．これは，ベンチマーク 505 の実際に測定した SPI-MPA である．

4.1.2 アウトプット

アウトプットとしての性能予測結果の中で，特に SPI に着目する．SPI は，命令ごとに要する時間という意味である．SPI により，本研究の目標の「プログラムの性能劣化を数値化に表す」を実現することができる．つまり，SPI は並列実行による（共有キャッシュの競合による），プログラムの実行時間がどの程度長くなるということを表す．

4.1.3 予測モデル

本研究では，C 言語で，任意のコア数，任意の並列実行プログラム数の並列実行時の性能が計算できる CAMP の予測モデルを実装した．

4.2 評価手法

ここから，実装した CAMP の予測モデルの評価手法について説明する．評価手法を図 8 に示す．図に示すのは 2 コアの状況であるが，より多くのコア数の場合でも同様に評価できる．

予測値の測定は，本章で説明した通り，実現した CAMP で得られた予測値を用いる．一方，実測値の測定は，それらのプログラムを実際に並列実行させ，性能パラメータを測定して計算した値を用いる．実測値はそれぞれ 25 回測定して計算した平均値である．

今回実装した CAMP の予測誤差は，式 (6) と式 (7) によって計算できる．

Vol.2019-ARC-235 No.7
Vol.2019-SLDM-187 No.7
Vol.2019-EMB-50 No.7
2019/3/17

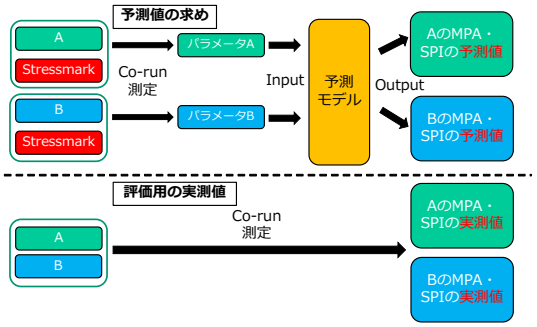

図 8　評価手法

表 3　ベンチマーク別の MPA と SPI 平均誤差

ベンチマーク	MPA 平均誤差	SPI 平均誤差
505	1.96%	3.14%
520	2.60%	1.85%
531	9.49%	1.17%
541	9.01%	1.31%
548	161.08%	0.30%
507	2.91%	7.30%
508	13.27%	2.57%
510	1.09%	1.21%
519	0.24%	0.89%
521	1.30%	0.59%
527	7.15%	1.51%
544	147.23%	0.72%
549	4.77%	0.99%

$$MPA_{Error}[\%] = \frac{(MPA_{Predict} - MPA_{Real})}{MPA_{Real}} * 100 \quad (6)$$

$$SPI_{Error}[\%] = \frac{(SPI_{Predict} - SPI_{Real})}{SPI_{Real}} * 100 \quad (7)$$

4.3　2 コアでの評価結果

2 コアの場合では，13 種類のベンチマークを 2 つずつ組合わせて評価する．組合せの数は 91 である．組合せごとに 2 つのベンチマークに対して，それぞれの SPI と MPA の結果を評価した．つまり，2 コアの場合では，MPA と SPI の結果の数はそれぞれ 182 である．

評価の結果，MPA の平均予測誤差は 31.32%，SPI の平均予測誤差が 1.79%であった．SPI においては非常に精度が高かったものの，MPA においては高い精度は得られなかった．ベンチマーク別の結果は，表 3 に示す．ベンチマーク別では，それぞれ MPA と SPI が 13 サンプルあり，表 3 に示したのはその平均値である．MPA の誤差が大きい原因は，ベンチマーク 544 と 548 にあるが，544 と 548 の SPI の平均誤差は非常に低い．式（1）により MPA と SPI には線形関係がある．ここで 544 と 548 の MPA の誤差が大きく，SPI の誤差が小さい理由を分析する．

SPI は，$(\alpha * MPA)$ と (β) の 2 項の和である．これは，プログラムの実行時間が項ごとに分割できるという意味である．$(\alpha * MPA)$ が表すのは，メモリ，キャッシュへのアクセスにかかった時間で，(β) が表すのはそれらの動作と関わらなく，計算などにかかった時間である．544 と 548 は，(β) と比べると $(\alpha * MPA)$ が非常に小さいため，プログラムの全体の実行時間に MPA の影響が非常に小さい．

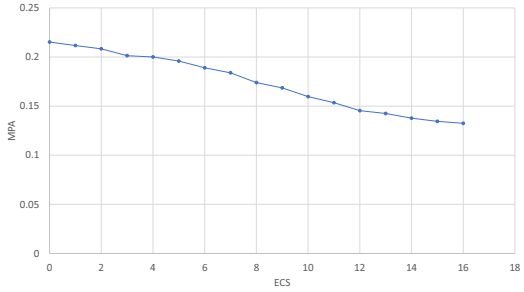

図 9　ベンチマーク 505 の MPA-ECS 特性

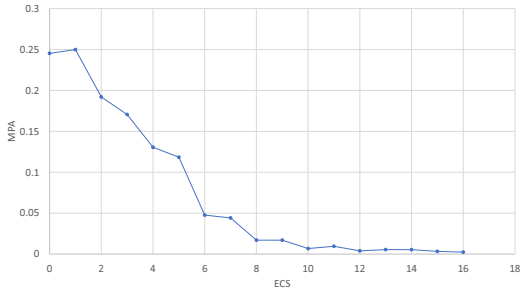

図 10　ベンチマーク 548 の MPA-ECS 特性

SPI を 100%とした時に，544 と 548 の $(\alpha * MPA)$ が占める割合は，0.77%と 0.34%である．比較として，他の 11 のベンチマークのこの割合の平均値は，44.53%であり，その中で最もこの割合が低いベンチマークでも 8.55%である．この割合は，総実行時間のうち，メモリ，キャッシュアクセスにかかった実行時間が占める割合を表している．この割合が低いプログラムは，より CPU 集約的（CPU-intensive）である．一方，この割合が高いプログラムは，よりメモリ集約的（memory-intensive）である．

つまり，544 と 548 のような CPU 集約のプログラムでは，MPA の予測誤差が非常に大きい値になっても，SPI の予測には影響しない．さらに，並列実行，共有キャッシュの競合による性能劣化の度合いも低い．548 を例とすると，548 の $(\alpha * MPA)$ の割合は 0.34%である．これは，このプログラムのミス率（MPA）が十倍増大にしても，実行時間（SPI）が 3.4%しか増加しないという意味である．このため，このようなプログラムは，並列実行による性能劣化は無視できる程度であるため，本研究の評価対象外と言える．

さらに，測定したデータから，544 と 548 のもう 1 つの共通点が見える．その共通点はプログラムの MPA-ECS 特性で説明できる．図 9 と図 10 に示すのは，505 と 548 のこの特性である．ECS は，利用できるキャッシュのサイズで，粒度は Way である．これらは，ストレスマークと並列実行する際に測定した結果によって得られたグラフであ

Vol.2019-ARC-235 No.7
Vol.2019-SLDM-187 No.7
Vol.2019-EMB-50 No.7
2019/3/17

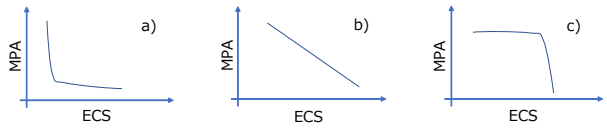

図 11 MPA-ECS 特性の 3 タイプ

る．この特性が表すのは利用できるキャッシュサイズの変化に従って，プログラムのミス率が変化する傾向である．この特性によって 544 と 548 には下記の 2 つの共通点があることがわかる．

共通点 1 ECS のとりうる範囲で，非常に小さい MPA となる

共通点 2 ECS が一定のサイズ以上になると，MPA が線形に減らない（飽和する）

利用できるキャッシュサイズが一定の値以上（図 10 では 8 Way）になると，利用できるキャッシュのサイズの増加とともにミス率が減少しなくなる．この時これらのプログラムのミス率の絶対値は 0 に非常に近い．他の 11 のプログラムの中で，同じような特徴が見えるのは（544 と 548 のように明らかではないが），508，531 および 541 である．他に，507 と 527 が共通点 1）を持っている．表 3 を見ると，544 と 548 以外で，MPA の平均予測誤差が大きいのは，508，531，541 および 527 である．この 2 つの共通点は，MPA の平均予測誤差とは関係があると考えられる．

従って，ストレスマークを評価対象として同じ実験を行った．つまり，ストレスマークを CAMP の測定ツールとしてだけ利用するではなく，0 Way から 16 Way まで利用するストレスマークを，17 個のプログラムとして扱い，他のプログラムと同様に CAMP によって見積もりを行った．その結果と前述のベンチマークの結果により，ハードウェアが一定であれば，プログラムの MPA-ECS 特性は図 11 の示す 3 種類に分けることができることが分かった．

タイプ a) のプログラムは，544 と 548 のように CPU 集約的で，キャッシュとメモリの使用が少なく，並列実行でキャッシュの競合による性能劣化も小さい．タイプ b) は，CPU 集約とメモリ集約の中間であり，タイプ c) は，メモリ集約的で，キャッシュを大量に使うプログラムである．タイプ b) と c) は，今回実装した CAMP により，高い精度で予測することができる．

本研究の主要な評価目標は SPI であるため，評価結果によって本研究で実現した CAMP は，高精度な性能予測が可能であることがわかった．CPU 集約のプログラムの特性上，SPI の予測誤差が必然的に小さいため，評価する必要性が低いものの，評価結果の精度は高い．メモリ集約のプログラムも，その中間のプログラムも，高い精度を得られた．

表 4 2，3，4 コアの平均予測誤差

コア数	2 コア	3 コア	4 コア
MPA	5.43%	6.39%	7.90%
SPI	2.02%	3.22%	6.00%

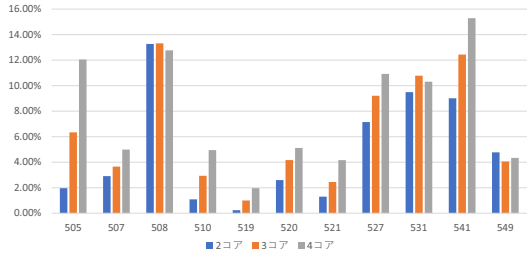

図 12 2，3，4 コアでベンチマーク別の MPA 平均予測誤差

5. 3/4 コアへの拡張

5.1 3 コア/4 コアでの評価結果

2 コアの結果から，544 と 548 が評価する必要性が低いとわかったため，今回は 3 コアと 4 コアでは 544 と 548 を除外し，他の 11 のベンチマークの組合せを 3 コアと 4 コア上で評価した．このため，今回 3 コアの評価では，286 の組合せ数であり，得られた MPA と SPI の結果のサンプル数はそれぞれ 858 個である．4 コアの評価では，1001 の組合せ数であり，得られた MPA と SPI の結果のサンプル数はそれぞれ 4004 個である．これらの結果を，表 4 に示している．ここでの 2 コアの結果についても，544 と 548 は除外している．コア数の増加とともに，予測誤差も大きくなることが分かった．

5.2 補正手法の提案

5.2.1 新たな課題と分析

ベンチマーク別の結果（図 12 と図 13）を見ると，すべてのベンチマークの増加量は同様ではない．

図のように，2 コアでは予測精度の高いベンチマークであっても，3 コアと 4 コアでは予測精度が下がっている．今回使用したベンチマークの中で最もメモリ集約的である 505，510，519 は，3 コアと 4 コアでの予測誤差が非常に大きくなっている．ただし，508，531，541 など CPU 集約的に近いプログラムはコア数の増加による影響が低い．メモリ集約なプログラムは，MPA による SPI の影響が強いので，MPA だけではなく，本研究の目標の SPI の予測精度も下がった．

この問題が起きる原因を説明する．予測用のインプットは，CAMP の特性上，対象のプログラムと 1 つのストレスマークの 2 つのプログラムの並列実行で得られた．このため，プログラムの最も良い性能（プログラム + 0 Way ストレスマーク）と最も悪い性能（プログラム + 16 Way ストレスマーク）は，2 コアの時の測定値である．そのため，

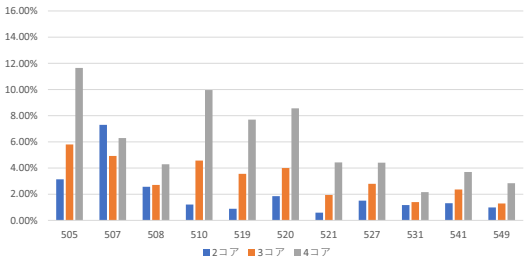

図 13 2，3，4 コアでベンチマーク別の SPI 平均予測誤差

表 5 補正すべきプログラムと補正すべきではないプログラム

コア数	補正すべき	補正すべきではない
3 コア	505 510 519	507 508 520 521 527 531 541 549
4 コア	505 507 510 519 520 531	508 521 527 541 549

このようなインプットによる予測範囲では，3 コアと 4 コアの最悪の性能が予測できない．

5.2.2 補正手法の提案

この問題の対策として，本研究では，予測範囲を拡大する補正手法を提案する．さらに，3 コアと 4 コアの環境でその手法を実施して評価した．

補正の手法は，予測のインプットデータに，3 コアと 4 コアの時の最悪状況を考慮することである．3 コアを例として説明する．2 コアの並列実行（プログラム + ストレスマーク）で測定したパラメータは，3 コアの予測に対応できないため，3 コアの時の並列実行の最悪状況（プログラム + 16 Way ストレスマーク　+ 16 Way ストレスマーク）の性能を測定し，これにより 3 コアの時の予測範囲を広げる．4 コアも同じように，4 コアの並列実行によりパラメータを測定する．

5.3 補正手法の評価

この補正手法を実際に実施すると，誤差を抑えることが可能となるが，実施上の新しい課題も存在する．あるプログラムの補正によって，並列実行の別のプログラムの予測精度が悪化することである．総合的に見て他のプログラムへの影響の大きいプログラムは，もとの 2 コアの状況で測定したパラメータを利用する方が予測精度が高い．

つまり，補正すべきプログラムと補正すべきではないプログラムがある．プログラムごとに補正すべきかどうかについて実験した結果，3 コアと 4 コアの場合，補正すべきプログラムの数が異なっていた（表 5）．表 5 に示すのは，全体的に予測誤差が最も小さい補正する/補正しないの組合せである．表 5 の組合せで補正することで，表 6 で示すように誤差を抑えることができる．

これにより，この補正手法は予測誤差を抑える効果があるとわかるが，コア数がいくつの状況で，どのようなプログラムを補正すべきかは現在完全に未知である．実験によって 3 コアと 4 コアでの最も誤差が小さい補正の組合せがわかったが，それは実験によって得られた結果からまとめた情報である．それらの補正すべきプログラムには実際にはどのような共通点があるかはわからない．

表 6 補正による SPI の予測誤差の変化

コア数	補正無し	補正あり
3 コア	3.22%	2.85%
4 コア	6.00%	4.48%

6. おわりに

本研究は，x64 プロセッサ向けの共有キャッシュによる性能劣化の影響を予測する手法である CAMP を，AArch64 で再現し，2 コア，3 コア，4 コアの環境で評価した．予測誤差はそれぞれ 1.79%，3.22%，6.00%であった．

さらに，3 コアと 4 コアの予測範囲を補正する手法を提案して評価した．この手法を使い，3 コアと 4 コアの予測誤差は 2.85%と 4.48%まで下がった．

参考文献

[1] Xu, Chi and Chen, Xi and Dick, Robert P and Mao, Zhuoqing Morley.: *Cache Contention and Application Performance Prediction for Multi-core Systems*. In Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on, pp.76-86. IEEE, 2010.

[2] Jiang, Yunlian and Tian, Kai and Shen, Xipeng.: *Combining Locality Analysis with Online Proactive Job Coscheduling in Chip Multiprocessors*. In International Conference on High-Performance Embedded Architectures and Compilers, pp.201-215. Springer, 2010.

[3] Subramanian, Lavanya and Seshadri, Vivek and Kim, Yoongu and Jaiyen, Ben and Mutlu, Onur.: *MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems*. In High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on, pp.639-650. IEEE, 2013.

[4] Subramanian, Lavanya and Seshadri, Vivek and Ghosh, Arnab and Khan, Samira and Mutlu, Onur.: *The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory*. In Proceedings of the 48th International Symposium on Microarchitecture, pp.62-75. ACM, 2015.

[5] Subramanian, Lavanya and Seshadri, Vivek and Kim, Yoongu and Jaiyen, Ben and Mutlu, Onur.: *Predictable Performance and Fairness Through Accurate Slowdown Estimation in Shared Main Memory Systems*. In arXiv preprint arXiv:1805.05926. 2018.

[6] Tam, David and Azimi, Reza and Soares, Livio and Stumm, Michael.: *Managing Shared L2 Caches on Multicore Systems in Software*. In Workshop on the Interaction between Operating Systems and Computer Architecture, pp.26-33. Citeseer, 2007.

[7] Perf Wiki，入手先 ⟨https://perf.wiki.kernel.org/index.php/Main_Page⟩（参照 2018-01-14）.

[8] SPEC CPU 2017，入手先 ⟨https://www.spec.org/cpu2017/⟩（参照 2018-01-14）.