社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－ARC－180 (11)
2008／10／21

Feature-Packingのためのソフトウェアによるメモリ管理手法の検討

三好健文[†1] 笹田耕一[†1] 小林良太郎[†2]
植原昂[†3] 吉瀬謙二[†3]

Feature-Packing（FP）プロセッサ・アーキテクチャは多数のシンプルなコアによるメニーコアアーキテクチャである．割込み機構をはじめとするハードウェアによるプログラムの実行支援機能を持たず，可能な限りソフトウェアにより制御するという特徴を持つ．そのため，ソフトウェアの動作の解析による種々の最適化によりハードウェアの持つ性能を有効に利用することが必要となる．一方で，一般のプログラマには，アーキテクチャに対する細かい考慮が必要となるためプログラムの記述は困難である．本論文では，各コアがローカルに持つメモリ（ノードメモリ）を効率良く利用するためのソフトウェアによるメモリ管理手法を検討する．提案する手法は，コンパイル及びリンク時の自動的な命令挿入によって実現することができるため，プログラマに大きな負担をかけない．また，簡単なプログラムを例にとり，提案手法を用いた場合の実行時間を予備実験により評価したところ，ノードメモリのチェックに伴う実行時間の増加は約1%程度であった．

Memory Management for Feature-Packing by Compiler

Takefumi Miyoshi ,[†1] Koichi Sasada ,[†1]
Ryotaro Kobayashi ,[†2] Koh Uehara [†3] and Kenji Kise [†3]

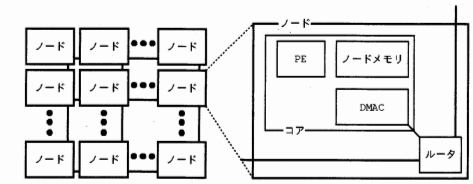
Feature-Packing processor architecture is a many cores architecture which consists of many simple cores. Each processor core does not have software execution support units such as interrupt module so software can control behaviors as possible. Therefore, various optimizations by predicating and estimating behavior of software can exploit the performance of the target architecture. On the other hand, it is difficult to program in general, since it is need to take into account architectural specifications. In this paper, the explicit method for memory management is proposed. The proposed method can manage data assignment onto a local memory of each core in order to execute given programs efficiently. A compiler inserts instructions for memory management so that programming cost is reduced. The proposed method is estimated by some simple examples and the results show that the execution overhead with checking content of local memory is about 1%.

1. はじめに

近年，マルチコア・アーキテクチャが，低消費電力で高い計算処理能力を得ることが可能なプロセッサ・アーキテクチャとして活発に研究開発されている．消費電力や配線遅延の増加などによる大規模な単一コアの性能向上がますます困難になっていく一方で，集積度は着実に向上し続けているため，今後，プロセッサ・アーキテクチャは，数百のコアを搭載するメニーコアの時代となる．そこで，メニーコアの実現における重要な課題である高速化，省電力化，ディペンダビリティの向上，製造コストの低減を解決するためにFeature-Packing（FP）プロセッサ・アーキテクチャが提案されている[1]．FPは，アーキテクチャの技術とソフトウェアの技術を組み合わせることで，多機能なメニーコア・プロセッサを実現するプロセッサアーキテクチャ技術の枠組みである．メニーコアを効率良く活用するためには，その豊富なコア数を活用し，消費電力と配線遅延を抑制しつつプロセッサ全体のスループットを向上させることで，プログラムを効率良く実行する必要がある．また，アプリケーションの多様化に伴い，各アプリケーションの性質に合わせて，プロセッサの性能，消費電力，及びディペンダビリティの間に存在するトレードオフを解決するための，高い柔軟性が必要となる．これらを実現するために，FPでは，キャッシュや割り込み，高機能なメモリアクセス機構といった機構を除いたシンプルなコアを多数用いる手法を提案している．

コアがシンプルであることは，開発コストを抑制し，多数のコアをチップに配置するメニーコアの実現を助ける．また，キャッシュや割り込みがないシンプルな

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†2 豊橋科学技術大学 情報工学系
Dept. of Information and Computer Science, Toyohashi University of Technology
†3 東京工業大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology



(a) 均質な多数のノードにより構成される Feature-Packingプロセッサアーキテクチャ　(b) 各ノード内の構成

図 1 Feature-Pcaking プロセッサアーキテクチャ概要

プロセッサコアはプログラムの動作を解析，予測することが比較的容易であり，各種最適化やコア間の柔軟な連携を実現しやすい．しかしその一方で，プログラマにコア間の通信やデータ管理などといったアーキテクチャ上の制約を考慮したプログラム記述を強いる．

そこで，本論文では，FP を対象とするプログラミングを容易にするために，コンパイラ及びリンカによってサポートすることが可能なソフトウェアによるメモリ管理手法を検討する．これは，割込み機構などのプログラム実行支援機能を備えていない FP のコア上で，ソフトウェアにより明示的にメモリ管理を行う．提案する手法では，効率良いデータアクセスを実現するために，プログラムの命令領域とデータ領域ををそれぞれセクションに分割し，それらをノードメモリ上にオーバレイによって割当てる．また，必要に応じて，外部メモリを用いたスタック領域の退避及び復帰を実現する命令を挿入する．セクションの分割及びメモリ管理コードの挿入は，コンパイル及びリンク時に自動的に実行することができるため，この手法によりプログラマの開発コストを削減することができる．

第 2 節で対象とする FP について述べ，第 3 節で FP におけるメモリ管理の問題について述べる．第 4 節で提案手法であるメモリ管理手法について述べる．第 5 節で，提案手法のオーバヘッドを予備実験により評価する．

2. FP プロセッサ・アーキテクチャ

図 1 に Feature-Packing プロセッサアーキテクチャ (FP) を示す．FP は，2 次元メッシュ状に配置される多数の均一なノードと，それ以外のモジュールから構成される．各ノードは，(1) コア : 2-Way スーパスカラ程度の RISC プロセッサである演算処理器 (PE)，PE から直接アクセス可能な数百 KB 程度のメモリ (ノードメモリ)，及びコア間のデータ転送を行う Direct Memory Access Controller (DMAC) で構成される，及び (2) ルータ : ノード間の Network-on-Chip 機構を提供する，から成る．モジュールとしては，外部 I/O や割込み処理を行う汎用プロセッサや，メインメモリがある．ノード及びモジュールには，それぞれ一意の ID が与えられ，この ID によって特定のコアあるいはモジュール間の通信を実現する．

各コアから大規模なキャッシュ，複雑な分岐予測機構，大規模な投機処理機構などを排除することで，メニーコアの利点である消費電力と配線遅延の抑制をさらに高め，また，タイル・アーキテクチャ[2] と同様，設計と検証のコストを抑えることができる．また，ノードの多様性，融合性，独立性などを利用し，柔軟に制御する．これによりアプリケーションの性質に合わせて，様々なトレードオフを考慮したプロセッサの規模や機能を動的に決定することができる．

3. メモリ管理手法

FP はメニーコアを実現することを目標にするため，各コアが潤沢なメモリを持つことは現実的ではない．そこで各ノードメモリのサイズを小さくすることが求められる．しかし，プログラム実行時に頻繁に外部メモリへのアクセスが発生することで，高い性能を得ることができない．そのため，メモリ管理が必要となる．

一般に，効率良くプログラムを実行するためのメモリ管理として，ハードウェアによるキャッシュやページ管理といった支援機構が用いられる．しかし，ハードウェアによる実行支援では，必要となる多量のハードウェア資源による面積や消費電力の増加及びコアの複雑化に伴う開発コストの増加によって，メニーコアの実現が難しくなる．一方でソフトウェアによるメモリ管理は，ハードウェア資源の増加を必要としない．しかし，プログラマに余計な負担をかけ，ソフトウェア開発コストが増加する．

そこで，ハードウェア資源の追加を必要としない完全なソフトウェアによるメモリ管理を，プログラマに意識せずに実現することが求められる．

4. 提 案 手 法

本論文では，第 3 節で述べた課題を解決するために，ソフトウェアによる明示的なメモリ管理によって，コア内のノードメモリを有効に利用する手法を提案する．

4.1 ソフトウェアによるメモリ管理手法

提案する手法では，効率良いデータアクセスを実現するために，プログラムの命令領域及びデータ領域をセクションに分割し，それらをノードメモリ上にオーバレイによって割当てる．また，必要に応じて外部メモリを用いた，スタック領域の退避及び復帰を実現する管理コードを挿入する．セクションの分割と，メモリ管理コードの挿入はコンパイル及びリンク時に自動的に実行することができるため，この手法によりプログラマの開発コストを削減することができる．

オーバレイは，外部メモリ上の一部分を高速なローカルメモリ上のアドレス空間に重ねて割当て，実行時に切り替えて使用する手法である．アクセスに時間のかかる外部メモリのデータをアクセス時間の短かいローカルメモリ上に置くことで効率良くプログラムを実行することができる．また，プログラム中では，ローカルメモリ上のアドレスが実効アドレスであり，複雑なアドレス変換機構を必要としない．これは FP のシンプルなコアを多数並べるという設計方針に沿っている．

4.1.1 セクション

提案手法では，グローバルアドレス空間をセクションと呼ぶ単位に分割しノードメモリへ割当てる．ハードウェアに手を入れず効率良く実装するために，セクション識別番号 (セクション ID) とノードメモリアドレスをアドレスビットを分割して表現する．すなわち，アドレス空間 32bit，ノードメモリ 256kB の場

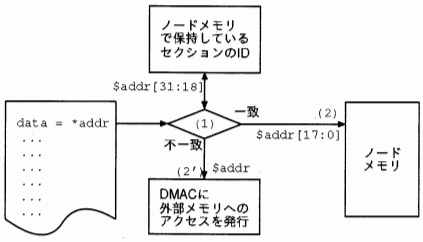


図 2 ソフトウェアによる明示的な管理下でのノードメモリアクセス

```
int *addr; int data;
int id = addr & 0xffc0000;
if(*SECTION_INFO_ADDR == id){
  data = *addr;
}else{
  // ノードメモリにデータを読込 (読込先, 読込元=セクション ID)
  DMA_GET(NODE_DATA_ADDR, id);
  data = *addr;
}
```

図 3 セクション ID チェック付きノードメモリアクセス

合，下位 18bit がノードメモリアドレスに相当し，上位 14bit でセクション ID を示す．

ノードメモリには，命令領域及びデータ領域の各セクションのデータがオーバレイによって割当てられる．プログラム実行中に，ノードメモリ上に保持されているデータのセクション ID をそれぞれレジスタ上に保存し管理する．

4.1.2 ノードメモリへのアクセスと更新手法

ソフトウェアによる明示的な管理下でのノードメモリへのアクセスを**図 2** に示す．ノード内の PE で動作するプログラムコードは，データにアクセスする際，まずノードメモリに現在格納されているセクション ID と，アクセスしようするデータのセクション ID を比較する (図 2(1))．

ここで，ノードメモリに格納されているデータのセクション ID がアクセスしたいデータのセクション ID に等しい場合，図 2 中の矢印 (2) のように，ノードメモリ上のデータに直接アクセスする．しかし，現在ノードメモリ上にあるデータのセクション ID とアクセスしたいデータのセクション ID が異なる場合，図 2 中の矢印 (2)' で示すように，DMAC へメモリ転送要求を発行する．**図 3** は，擬似コードによってこの動作を示す．ここで，`SECTION_INFO_ADDR` は，ノードメモリに保持されるデータのセクション ID を保持する番地である．同様に，命令領域にアクセスする場合は，関数呼び出しや分岐命令時に，該当するアドレスのセクション ID を確認する．

また，一般にプログラムを実行するために十分なスタックを取ることは困難であるが，メモリが少ない場合には，スタックの確保は，より困難となる．そこで，スタック領域の残りサイズを保持しておき，スタックが新たに必要となる際に，必要となるスタックの量と残りサイズを比較する．十分なサイズがない場合にはスタック領域を外部メモリに退避させる．擬似コードを**図 4** に示す．ここで，`REQUEST_SIZE`，`LOCAL_STACK_BASE`，

```
if(stack_size < REQUEST_SIZE){
  // 外部メモリにスタック領域を退避 (書出先, 書出元, サイズ)
  DMA_PUT(stack_addr,
          LOCAL_STACK_BASE - stack_size,
          stack_size);
  stack_addr += stack_size;
  stack_size = LOCAL_STACK_SIZE;
}
stack_size -= REQUEST_SIZE;
func();
```

図 4 スタックの退避の擬似コード

及び `LOCAL_STACK_SIZE` は，それぞれ要求するスタックサイズ，ノードメモリ上のスタックポインタのベース，ノードメモリ上のスタック領域のサイズで，これらは，コンパイル/リンク時に決定される．また，`stack_size` 及び `stack_addr` は，それぞれ現在ノードメモリ上のスタックのサイズ及びスタックの退避先のアドレスである．

4.1.3 コア間データ共有

FP では各ノードのノードメモリに，他のノードからアクセスできる．ここで，他のノードからのアクセスはルータを介し，DMAC に到達する．想定するノードには割込み機構がないため，他のノードからのアクセス時に，ノードメモリ上に適切なセクション ID のデータが配置されているかどうか，ソフトウェアによって確認することができない．そこで，データの一貫性を保持するため，他ノードからアクセスされる可能性のあるデータはその時点において外部メモリに配置する．

しかし，この処理には大量のデータ転送が必要となり，パフォーマンスの低下の原因となる．そこで，データの効果的な配置や複製，及びアクセス時刻の解析による局所化などを利用した最適化が必要となる．

4.2 コンパイラ/リンカによるコード生成

提案手法である明示的なメモリ管理手法をプログラマに実装させることは，ソフトウェアの実装コストを増加させる．そこで，コンパイラ及びリンカによるメモリ管理コードの挿入及び動作時にアクセスすべきアドレスの生成を実現する．コンパイラでは，メモリアクセス時のチェック用のコードの挿入，及びセクションへの分割を行う．また，リンカによって，各セクションのアドレス割り当てを決定する．

5. 予 備 実 験

メモリ管理を実現するためのプログラムコードを挿入した場合のセクション ID のチェックにかかる性能の低下を測定する．これは単にオーバヘッドを評価したものであり，ミスに伴う DMA 転送コストは含まない．評価には，メニーコアプロセッサのソフトウェアシミュレータである SimMc[3] による FP のシミュレータを用いた．サンプルプログラムには，SimMc 上で動作するよう記述されたクイックソート (qsort) 及び行列計算 (matrix) を用いる．それぞれ，2048，32×32 のサイズのデータを扱っている．シミュレータによる実行サイクル数及びオーバヘッドを**表 1** に示す．この結果から，与えられたプログラム中に，提案するメモリ管理のために必要となるチェック部分が少なく，そ

表 1 提案手法によるオーバヘッドの評価

	メモリ管理なし	メモリ管理あり	オーバヘッド
qsort	1358777	1376977	+1.34%
matrix	479136	474484	+0.98%

の実行時間に係るオーバヘッドが小さいことが分かる.

6. 関連研究

マルチコア, メニーコアを対象として, 計算に必要なメモリを近くに配置する手法にはいくつかの研究がなされている. 同じく二次元メッシュ型のメニーコアプロセッサである Raw Machine[4] のメモリ管理手法として Maps[5] が提案されている. この手法では, コンパイル時に ECU とモジュロアンローリングによる静的なデータのコア割り当て及び, 実行時の直列化について述べられている. しかし, ローカルメモリへのアクセスミス時の, 他のコアのメモリあるいは外部メモリへのアクセス手法については述べられていない. FP の各ノードは割込み機構を持たないため, ノードメモリへのアクセスミス時に明示的なメモリ管理手法が必要となる.

また, メモリの局所性解析やメモリ管理の研究には, チップ内のスクラッチパッドメモリを有効に利用するための手法[6)–8)] や, 配列のアクセス範囲の解析手法[9)] が研究されている. これらの最適化手法を, 提案手法であるソフトウェアによるメモリ管理手法にも適用することが可能であり, これは今後の課題である.

7. まとめ

Featcure-Packing プロセッサアーキテクチャを対象としたソフトウェアによる明示的なメモリ管理手法を提案した. 提案手法は, 割込み等のハードウェアによる実行支援機能を用いることなくソフトウェアによるメモリ管理を行う. また, 簡単なプログラムを用い, 提案手法の予備評価を行った. この結果として, 提案手法のオーバヘッドの一つである, ノードメモリ上に保持されているセクション ID のチェックに伴って増加する実行時間が約 1%程度であることを示した. 今後の課題として, ノードメモリ上でのアクセスミス時におけるハードウェア手法も含む各種メモリ管理手法と提案手法を比較し, その実行時間におけるオーバヘッドやハードウェア資源の消費量について評価を行う.

謝辞

東京工業大学吉瀬研究室森谷章さんによるプログラムを評価に用いた.

参考文献

1) 小林良太郎, 吉瀬謙二. 多機能メニーコアを実現するアーキテクチャ技術 feature-packing の構想 (inventive and creative architecture 特別セッション i). 情報処理学会研究報告. 計算機アーキテクチャ研究会報告, Vol. 2007, No. 115, pp. 11–15, 20071121.

2) Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. *SIGARCH Comput. Archit. News*, Vol.32, No.2, p.2, 2004.

3) 植原昂, 佐藤真平, 森谷章, 藤枝直輝, 高前田伸也, 渡邉伸平, 三好健文, 小林良太郎, 吉瀬謙二. シンプルで効率的なメニーコアアーキテクチャの開発. 情報処理学会研究報告 2008-ARC-180, October 2008.

4) M.B. TAYLOR. Evaluation of the raw microprocessor : An exposed-wire-delay architecture for ilp and streams. *Proc. ISCA-31, 2004*, 2004.

5) Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Maps: a compiler-managed memory system for raw machines. *SIGARCH Comput. Archit. News*, Vol. 27, No.2, pp. 4–15, 1999.

6) Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 104–109, New York, NY, USA, 2004. ACM.

7) S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, p. 409, Washington, DC, USA, 2002. IEEE Computer Society.

8) Manish Verma, Stefan Steinke, and Peter Marwedel. Data partitioning for maximal scratchpad usage. In *ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation*, pp. 77–83, New York, NY, USA, 2003. ACM.

9) Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J.Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 1–10, New York, NY, USA, 2008. ACM.