分散共有メモリ管理プロセッサ MBP-light の アセンブラおよびコンパイラの実装と評価

阿部 剛・井上浩明・美辺央希・若林正樹・天野英晴
慶應義塾大学 計算機科学専攻
〒 223-8522 横浜市港北区日吉 3-14-1
E-mail: abe@am.ics.keio.ac.jp

概要

超並列計算機 JUMP-1 において分散共有記憶を効率よく管理するために、MBP-light と呼ばれる制御 chip が提案されている。MBP-light は、パケットの高速処理を目的として MMC と RDT Interface という専用ハードウェアと、柔軟性の確保を目的とした MBP Core と呼ばれる 16 bit 幅の RISC プロセッサからなる。MBP Core はプロトコル処理の複雑な仕事を担当し、かつ、ハードウェア量を抑えるため、Packet-Buffer Register アーキテクチャと呼ばれる特殊な構造を持つ。

そこで、本論文では、この特殊なアーキテクチャをうまく活用するような MBP-light 用アセンブラ及びコンパイラの実装及び評価を行う。実装に際しては、MBP Core の特殊性をなるべく吸収し、プログラムの生産性の向上を図ることを目的とする。そして、その結果、簡単なパケット処理時間に関して評価を行ない、十分な性能が得られることを示す。

キーワード: JUMP-1, MBP-light, PBR, アセンブラ, コンパイラ

An Implementation and Evaluation of the Assembler and the Compiler for MBP-light

T.Abe, Inoue. H, O.minabe, M.masaki and H. Amano
Dept. of Computer Science, Keio University
E-mail: abe@am.ics.keio.ac.jp

abstract

A massively parallel processor called JUMP-1 has been developed for building an efficient cache coherent-distributed shared memory on a large system with thousands of processors. MBP(Memory Based Processor)-light is a special purpose controller to manage the DSM (Distributed Shared Memory) on JUMP-1. It consists of a simple RISC core processor to treat a complicated protocol transaction flexibly and hardwired controllers to manage quickly memory systems, bus and network packets.

Since MBP-light is imposed limitations on the small number of gates, It has a specific instruction set architecture called Buffer-Register Architecture.

In this paper, an assembler and a compiler for MBP-light for developing protocol processing programs are introduced. An empirical evaluation shows that these tools are effective for development o f a high speed protocol programs.

Key words : JUMP-1, MBP-light, PBR, Assembler, Compiler

1　はじめに

CC-NUMA(Cache Coherent Non-Uniform Memory Access model) は将来の大規模かつ高性能な並列計算機の構成方式の一つとして注目されている。バス結合型の小規模並列計算機に比べ多数のプロセッサを接続できる一方、これらの小規模並列計算機で開発された並列プログラムを容易に移植できるという利点も持つ。

このため、最近は研究用、商用の CC-NUMA system が各地で開発されている。Stanford 大学の DASH[1]/FLASH[2]、MIT の Alewife[3]、SGI の Origin2000[4]、Sequent 社の NUMA-Q[5] がその代表例である。しかし、従来の CC-NUMA の実現方式は数十、数百プロセッサの規模では効率よく動作するものの、数千、数万プロセッサにも及ぶ規模では分散共有記憶を管理するのに多量の資源を必要とし、効率が悪い。

JUMP-1[8] は、7 大学の協力により開発されている cache coherent な分散共有記憶 (DSM) を持つ超並列計算機の試験機である。この計算機の開発目的の一つは、超並列計算機上に効率の良い DSM を実現するための基本的な技術を確立することにある。このため、多くの新しい技術が JUMP-1 の DSM には導入されているが、これらの技術の実現の鍵となるのは、MBP(Memory Based Processor)-light と呼ばれる分散共有記憶制御用プロセッサである。

本論文では、この MBP-light のためのアセンブラ及びコンパイラを実装し、MBP-light におけるプロトコルプログラミングの生産性の向上を図る。第 2 節では JUMP-1 の構造について説明する。また、MBP-light の構成を第 3 節で、アセンブラを第 4 節、コンパイラを第 5 節、評価を第 6 節で行う。

2　超並列計算機 JUMP-1

図 1に示すように、JUMP-1 は RDT (Recursive Diagonal Torus)[11] と呼ばれる結合網を通じて 256 のクラスタを結合した構成をとる。RDT は図 2のように二次元の torus を対角方向に再帰的に重ねた構成を持ち、torus だけでなく一種の fat tree の構造も合わせ持つ。各々のクラスタからは、高速な I/O ネットワーク [10] を介して DISK や高解像度の VIDEO へつなぐことも可能である。

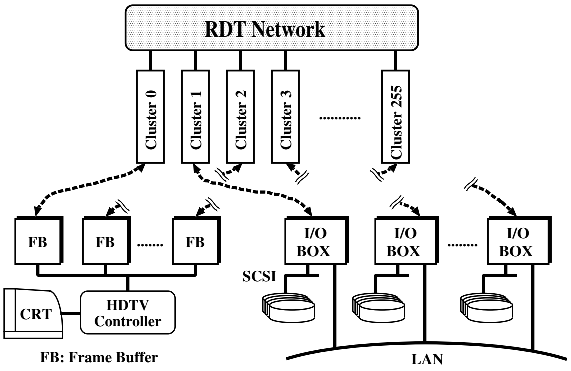

図 1: JUMP-1 の構成

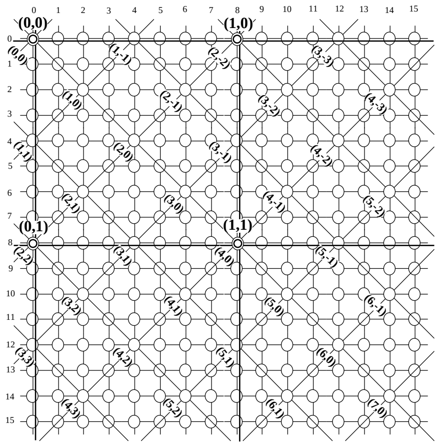

図 2: 結合網 RDT

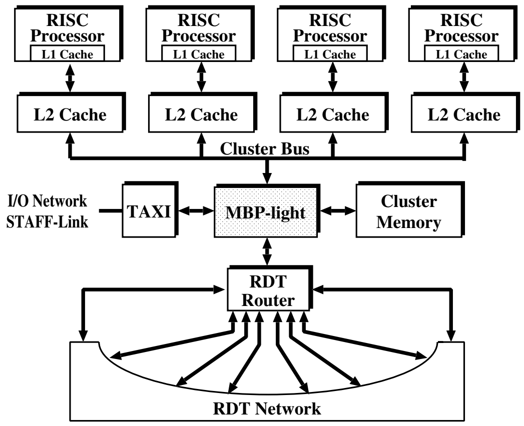

図 3: JUMP-1 の cluster 構成

各々のクラスタは図 3のようなバス結合型の並列計算機であり、L2 cache を持つ 4 台の SuperSPARC+、MBP-light と RDT router[9] からなる。JUMP-1 の心臓部に位置する MBP-light は DSM、同期やパケット処理を管理する専用プロセッサである。

JUMP-1 における DSM 管理の詳細な方式は [12, 13] に記述されている。

3 MBP-light の構成

JUMP-1 において MBP-light は効率の良い分散共有メモリを実現するための核となる。

FLASH の MAGIC や NUMA-Q での SCLIC など従来の DSM 制御用プロセッサでは、まず受信したパケットのヘッダ部とデータ部をハードウェアで分離する。そしてプロトコル処理に関係しないデータ部をバッファ間で直接転送し、プロトコル処理に関係するヘッダ部は強力なコアプロセッサのプログラムにより処理する。最終的にパケットを送信する時に、ヘッダ部とデータ部を再びハードウェアにより高速に組み合わせる。

しかしながら、JUMP-1 でこの方式をとると、以下の問題が発生する。

1. RHBD[7] 方式と言うディレクトリ方式ではメッセージのマルチキャストに基づいているために、応答パケットの高速な生成及び収集が性能に大きく寄与する。これには強力なプロセッサを設けたとしてもその処理能力は十分とは言えない。
2. クラスタメモリ上のデータの状態を示すタグはコアプロセッサよりもむしろハードウェアで検査されるべきである。
3. RHBD 方式は複雑なヘッダ構造を必要とするので、種々のサイズのヘッダ部が JUMP-1 で用いられている。このため、パケットの自動分割は困難である。また、DSM 上での効率のよいメッセージ転送を達成するために、データ部には一種のタグが存在する。これらはプロトコルに関係しているため、コアプロセッサにより扱われなければならない。これらの理由からパケットの自動分割は MBP-light にとって適したものではない。

そのため、以下の設計方針を採用されている。

1. 応答パケットの生成や収集は専用のハードウェア論理により管理される。また、クラスタメモリのタグを検査するためにも別の専用論理を設ける。
2. 上記に述べた処理はハードウェアで行う場合にゲート数の増大を招く。そこで、プロセッサコアを単純な 16bit 幅の RISC とすることで全体のハードウェア量を減少させる。この MBP Core は単純な構成の一方、パケット処理に適合した命令セットである Buffer-Register Architecture[15] により高速にパケットを処理することができる。

図 4に MBP-light の構成を示す。MBP-light は結合網でのパケットを扱う RDT Interface、クラスタメモリおよびバスを制御する MMC (Main Memory Controller)、MBP Core の三つの部分から構成される。RDT Interface と MMC はハードウェア論理で制御され、MBP Core からは独立に動作する。MBP Core の動作が必要な場合は、MBP Core へ割り込みをかけることで処理する。

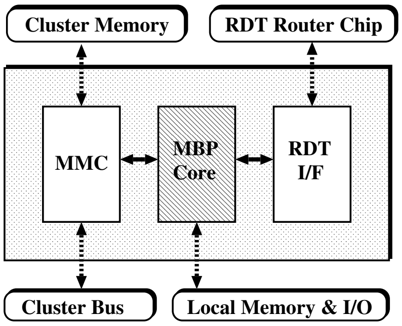

図 4: MBP-light の全体構成

3.1 Buffer-Register Architecture

高速なパケット処理の多くは、RDT Interface と MMC のハードウェア論理により制御されるため、MBP Core は DSM を管理するプロトコルの複雑な部分を受け持つことになる。その仕事としては、受信パケットの解析や GPMT の参照、アドレス変換、送信パケットの生成等があげられる。JUMP-1 のパケットのヘッダ部は複数 flit にわたる複雑なもので、かつデータ部にはプロトコル制御に必要な tag も含まれている。よって、全てのパケットバッファをレジスタとして扱うことが最も効率がよい。しかしながら、パケットバッファの幅が 68bit であるため、もし実現できたとしても、非常に多くのハードウェア量を必要とする。

この問題を解決するために、MBP Core は 16 個 $\times$ 16bit 幅の GPR (General Purpose Register) と 112 個 $\times$ 68bit 幅の PBR(Packet Buffer Register) を持つ。PBR は GPR をポインタとして参照され、あたかも 16bit 幅のレジスタのように振舞う。そのため、PBR–GPR 間と PBR–PBR 間の演算及び転送を行うことができ、また、68bit 幅のパケットとして MMC や RDT Interface へ直接渡すことも可能である。これらの演算が主にパケットバッファとレジスタの間で行われることから、我々はこの構成を Buffer-Register Architecture と呼ぶ。

MBP Core の構成を図 5に示す。MBP Core は 21bit 幅の命令と 16bit 幅のデータを扱う 4 段パイプラインからなる RISC であり、21bit× 64K の専用のローカルメモリを持つ。また、MBP Core は I/O マップ方式をとっており、64K 番地が I/O デバイスや同期操作のために別に与えられている。また、クラスタメモリも PBR を通じて参照される。他にも、16bit× 256 の内部メモリを持ち、table jump や応答パケット用の情報を保持している。

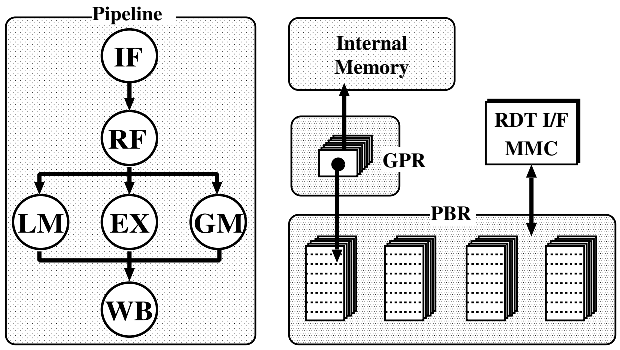

図 5: MBP Core の構成

3.2 命令群

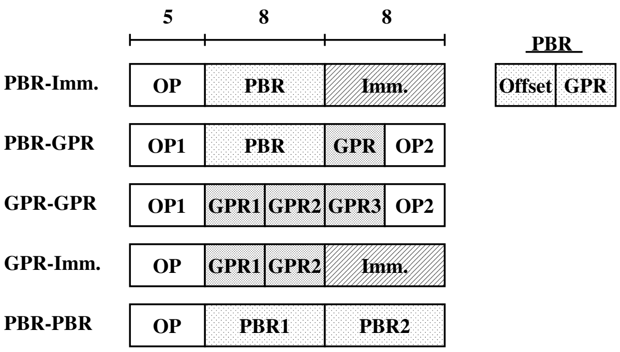

図 6: 代表的な命令形式

図 6は MBP Core の代表的な命令形式を示している。GPR は 3-port memory を用いているので、GPR 間の演算は 3 つのオペランドをとることができる。一方、PBR は 2-port memory であるため、PBR–GPR 間の演算は、2 つのオペランドしかとることができない。図 6中の PBR–GPR 間の演算は GPR の内容と GPR+4bit のディスプレースメントによって指されている PBR 内の 16bit データとの間で行われる。ほとんどのパケットヘッダの field が 8bit 以下であるため、PBR をターゲットとした演算は 8bit か 16bit 幅のものが多い。しかし、データ転送用にそれ以上の幅を持つ命令もある。

表 1: MBP Core の命令の分類

分類名	種類	代表例
WGG 命令	GPR–GPR 間の演算	ADDGG, SUBGG
WGI 命令	GPR–直値間の演算	ADDGI, SUBGI
分岐命令	分岐	BOZ, BNZ
LMA 命令	local memory の参照	LLM, SLM
BPI 命令	PBR–直値間の演算	ADDPI, SUBPI
WPG 命令	PBR–GPR 間の演算	ADDPG, SUBPG
MPP 命令	PBR–PBR 間の転送	MVPP, MVLPP
RDT 命令	RDT Interface 制御	PRDT, GRDT
MMC 命令	MMC 制御	TGM, PBP
TJ 命令	Table Jump	TJ, TJRQ
INT 命令	割り込み関係	RFI, INTE
IMA 命令	内部 memory の参照	LIMG, SIMG
SPE 命令	特殊命令	HSHC, HSHG
NOP 命令	No OPeration	NOP

表 1に MBP Core の持つ命令の分類を示す。今まで述べてきた命令に加え、RDT Interface や MMC を制御する命令や割り込み関係の命令等も存在している。また、プロトコル処理を高速に行う目的で特殊命令を 12 命令程度備えてあり、これにより通常の RISC プロセッサよりも有利な場合がある [15]。

このように、MBP-light は独自の設計を用いており、また後述するが様々な禁止事項がある。これらすべてを常に把握してプロトコルプログラムを組むことは容易ではなく、その負担を少しでも軽くするため本論文では、アセンブラとコンパイラを実装した。

4 MBP-light アセンブラ

MBP-light アセンブラは、入力を lex で字句解析し、yacc により文法チェック及びコードへの変換を行う。出力形式は、VHDL シミュレータ用形式とバイナイリ形式の 2 種類をサポート。アセンブラのコード総行数は約 4500 行。

このアセンブラは、単体での利用と、コンパイラのバックエンドとしての利用に対する二つの機能があり、その動作は多少異なっている。

アセンブラ単独で使用する場合は、一切の最適化を行なわない。文法間違い以外はすべてプログラマが把握しているものとするが、下記にある MBP Core 命令のおける禁止事項に違反した場合は、警告を出力するようにした。また、疑似命令やローカルラベル等を取り入れることにより、生産性の向上を図っている。上の表 1に示した各命令においての禁止事項を以下にまとめておく。

- すべての分岐命令は delay slot 1 を持つ遅延分岐である。分岐で使うレジスタに対してはフォワーディングが行なわれない。
- ポインタとなる GPR については完全にフォワーディングが行なわれるが PBR の値はフォワーディングされない。したがって PBR に書き込んだ値を使うためには 1 命令分の NOP または関連のない命令を 挿入しなければならない。
- MMC 系命令と MMC 系命令の同期命令である MEMBAR 命令との間において、PBR にアクセスする際にはその値は保証されない。ただし、その動作に使用していない種類の PBR へのアクセスは自由に行な うことが出来る。ここで言う PBR の種類区別は (MMC-PBR、From-RDT PBR、To-RDT PBR) の 3 種類を指している。また、MMC 系命令の直前の命 令でも PBR をアクセスしてはならない。
- Table Jump 命令もジャンプ命令なので、1 slot を持つ遅延分岐である。しかも引数である GPR へはフォワーディングができ ない。
- 内部メモリアクセスの直後に Table Jump 命令を行なってはならない。
- 特殊命令の一部の命令 (LUDR、 SUDR) はフォワーディングされないので、直後のストア命令には NOP を入れる必要がある。

コンパイラとともに使用する場合においては、コンパイラで収集された情報を利用し、機械依存の最適化を行なう。現在行なっているのは、メモリアクセスで起きる MBP Core のストールでの最適化だが、これは、メモリアクセス命令とその同期命令である MEMBAR 命令との間に直接このストールと関係ない命令を可能な限り挿入することによって実現する。コンパイラから得られる PBR の種類の情報を利用し、その PBR の種類とは違う命令、もしくはそれとは全く関係ない命令を適時入れていく。

5 MBP-light コンパイラ

このコンパイラは、C 言語という高水準言語の採用により、MBP Core における通信プロトコルプログラムの生産性の向上を図り、PBR における最適化の情報収集、汎用レジスタ最適化等による性能の向上をはかることを目的としている。入力は C 言語のサブセットを基本とし、出力はアセンブリ言語とする。

5.1 MBP-light コンパイラの言語仕様

本コンパイラの対象である MBP Core は、クラスタメモリ間の通信プロトコル処理を専用に行うものであり、その処理で使用される型は整数に限定されている。これを考慮し、基本型には 16bit 整数型 (int) と 8bit 整数型 (char) のみを用意した。派生型とポインタをサポートする。

PBR: プロトコル処理というのは、実際にはパケットの生成、分解が主な処理である。このパケットは P-BR(Packt Buffer Register) という 68bit レジスタに入れられ、加工される。したがって、この PBR を容易に操作できる仕様が求められる。そこで、この PBR 専用の型を作成し、それを配列としてアクセスできるようにした。具体的には、三種ある PBR(汎用 PBR、To RDT PBR、From RDT PBR) をそれぞれ、宣言 pbr、tpbr、fpbr に対応させ、PBR 自体への最小アクセス単位である 8bit(Tag:4bit) でアクセスできるような配列にした。

PBR 配列は、実際の 68bit レジスタそのものなので、グローバル変数扱いとした。PBR 宣言は、汎用の 68bit レジスタは 64 本 (宣言例: pbr p[64][9])、From RDT 用 68bit レジスタは 8 本 (宣言例: fpbr fp[8][9])、To RDT 用 68bit レジスタは 8 本 (宣言例: tpbr tp[8][9]) のようにする。

特殊命令: MBP Core には、C 言語の一般的な文法では表現しにくい特殊命令が数多くある。この解決方法はいくつかあるが、現在はもっとも単純にシステムコールという形で採り入れた。これらはほぼアセンブリ命令列と一対一対応を取り、そのまま変換される。

5.2 コンパイラの実装

実際の処理は、入力を lex で字句解析、yacc で構文木を作成し、構文解析で、中間コードに落す。中間コードに対しレジスタ最適化等を施す。中間コードと記号表を用いアセンブリ言語に変換する。変換は使用する命令の選択と実行時の記憶管理 (静的割り付け、スタック割り付け)、レジスタの割り付けである。今回の実装では MBP Core 特有の最適化及び汎用レジスタ最適化に重点をおいた。コンパイラのコード総行数は約 8 千行。

汎用レジスタ最適化: MBP Core にはキャッシュがついておらず、メモリアクセスに際して多大な遅延が生じる。特に、コンパイラにおいてレジスタの割り付けを効果的に行わないとロード、ストア命令が頻出し、これが

性能に大きく響くと考えられる。そのため、汎用レジスタの最適化を行った。

実装は、コンパイラにおいて、中間コードまで変換したあと、これをフローグラフに変換する。フローグラフは基本ブロックとそれらを結び制御の流れを表すポインタからなる。基本ブロックは連続した中間コードの文からなり、途中で分岐したりしない文の集まりを言う。この基本ブロックからもしくは基本ブロックへのポインタを張ることで、全体の制御の流れを表す。このフローグラフを利用し、各基本ブロックでの変数の定義、使用等の情報を集めることにより、最終的に最適なレジスタ割り付けを決める [16]。

MBP Core 固有の最適化: MBP Core 固有の最適化では、本来、アクセスする PBR の種類 (汎用、From RDT、To RDT) がいずれであるのかは PBR へのポインタであるレジスタ内の値によるので分からないところを、PBR に関する情報をアセンブラに渡すことで、アセンブラによるコード生成の段階で、最適化できるようにする。

レジスタの割り当て: MBP Core には R0 から R15 の計 16 個の 16bit レジスタを持つ。割り当ては表 2に示す。

表 2: レジスタ割り当て

レジスタ	役割
R0	常に 0
R0 ~ R7	一般的な計算に使用
R8、9、10	頻繁する配列計算用に使用
R11	現在実行しているプログラムの静的変数を格納しているメモリの先頭ポインタ (STATIC-TOP)
R12	現在実行している関数の自動変数を格納しているメモリの上端ポインタ (STACK-TOP)
R13	現在実行している関数の自動変数を格納しているメモリの下端ポインタ (STACK-END)
R14	現在実行している関数の返り値 (RET-NUMER)
R15	現在実行している関数終了後に飛ぶ番地 (RET-ADDRESS)

メモリ管理: 実行時のメモリ管理は、先にのべた記憶管理用レジスタ (R11 ~ R15) を利用して行う。現在、静的割り付け、スタック割り付けのみサポートしている。その様子を図 7に示しておく。

一連のコード解析が終ると、中間言語からアセンブリ言語に変換する段階をむかえる。この時考えなければならないことはメモリ管理である。コード解析という静的解析が終了した時点では、コード、静的変数、そして各関数内で宣言される変数の 3 つの大きさが分かっている。割り付けの初期状態時点では、(a) のような状態で R11 及び R13 をセットしておき、この状態でプログラムの実行を開始する。実行開始後、一般的にはまず main 関数に飛ぶことになる。そのときのメモリ割り付け状態が (b) である。最初に現在のシステムレジスタ (R12 ~ R15) を退避したあと、その後ろに main 関数内の変数分の領域を確保するよう再びシステムレジスタを設定しなおす。(c) は main 関数内にあるサブルーチンでさらに飛んだ場合を表しており同様の動作をする。

このような管理をすることにより、静的及び自動変数は相対番地で管理できる。

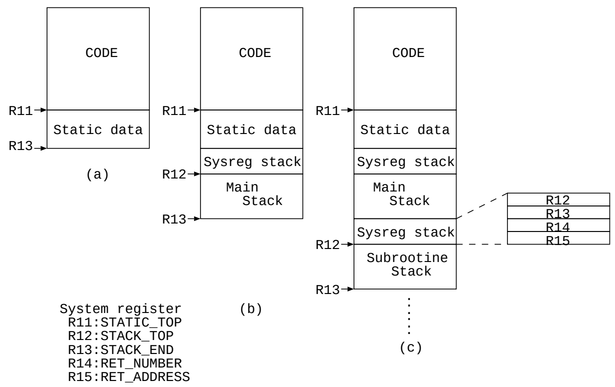

図 7: Memory の割り付け

これらの処理をへて、最終的にコンパイラは MBP Core のアセンブリ言語命令列を出力する。

6 評価

本研究で提案した MBP-light コンパイラを用いて簡単なプロトコルプログラムをコンパイルし、その実行時間を評価する。評価プログラムは、図 8において

1. RDT Network からデータ要求を受信、その要求されたデータをクラスタメモリの中から取り出す
2. 要求を出したクラスタに応答を返す

という動作を行うものである。

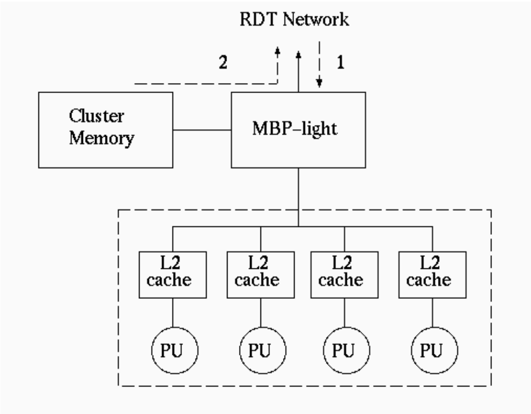

図 8: 評価プログラムモデル

評価は、このプログラムを以下の条件で処理して行った。

- アセンブラレベルで完全に最適化して記述したもの
- C 言語で書き最適化しなかったもの
- 機械依存の最適化のみをしたもの
- 汎用レジスタの最適化までしたもの

それぞれの実行時間を測定した。実行時間の測定はVHDLシミュレータを用いて行い、MBP-light の動作周波数は50MHz として測定した。測定結果を表 3に示す。

表 3: 実行時間

条件	実行時間
最適化なし	11020 ns
機械依存のみ	8860 ns
最適化すべて行う	4652 ns
アセンブラレベル	2200 ns

表 3を見れば分かる通り、アセンブラレベルで完全に最適化した記述に対し、C 言語で記述したプログラムは最小でも約 2 倍の実行時間がかかるという結果がでてしまっている。この原因としては、サブルーチン呼び出しによるシステムレジスタ等のロード、ストアによる遅延時間の増大や、配列の要素計算によるコード量の肥大化などがあげられる。特に後者の影響が大きく、レジスタ値を予測してコードを記述できるアセンブラ記述に対し、コンパイラは添字の変数内容まで追えないために毎回数回の掛け算等を強いられる。そのため、コンパイラのコード及び実行時間が飛躍的に増大してしまった。

最適化に関しては、やはり機械依存の最適化が効果を発揮している。これは、MBP Core が独自のプロセッサであり、MMC 系命令のストールのような特殊な条件で大きなオーバヘッドが出る場合が多いことが関係している。

7 結論

本論文では、MBP-light におけるプロトコルプログラミングの生産性の向上を図るためにコンパイラの実装を行った。また、実際のアセンブラレベルの記述や最適化度合いによる実行時間の測定を行った。その結果、アセンブラレベルの記述に対し、約 2 倍の実行時間となり、配列計算やサブルーチン呼び出しによるオーバヘッドがそのまま実行時間となって現れることがわかった。今後、これらの部分の最適化を行う予定である。

参考文献

[1] D. Lenoski et. al., “The Stanford DASH Multiprocessor,” IEEE Computer, **25**, pp.63–79, 1992.

[2] J. Kuskin et. al., “The Stanford FLASH Multiprocessor,” The 21st ISCA, pp.302–313, 1994.

[3] D. Chaiken and A. Agarwal, “Software-Extended Coherent Shared Memory: Performance and Cost,” The 21st ISCA, pp.314–324, 1994.

[4] J. Laudon and D. Lenoski, “The SGI Origin: A cc-NUMA Highly Scalable Server,” The 24th ISCA, 1997.

[5] T. Lovett and R. Clapp, “STiNG: A CC-NUMA Computer System for the Commercial Marketplace,” The 23rd ISCA, pp.308–317, 1996.

[6] H. Tanaka et. al., “The Massively Parallel Processing System JUMP-1,” オーム社, 1996.

[7] T. Kudoh et. al., “Hierarchical bit-map directory schemes on the RDT interconnection network for a massively parallel processor JUMP-1,” International Conference on Parallel Processing, August, pp.I-186–I-193, 1995.

[8] K. Hiraki et. al., “Overview of the jump-1, an mpp prototype for general-purpose parallel computations,” IEEE International Symposium on Parallel Architectures, Algorithms and Networks, pp.427–434, 1994.

[9] Hiroaki Nishi et. al., “Router Chip: A versatile router for supporting a distributed shared memory,” IEICE-ISPAN, 1997.

[10] H. Nakajo et. al., “An I/O Network Architecture of the Distributed Shared-Memory Massively Parallel Computer JUMP-1,” International Symposium on Supercomputer, 1997.

[11] Y.Yang et. al., “Recursive Diagonal Torus: An interconnection network for massively parallel computers,” IEEE the 5th Symposium on Parallel and Distributed Processing, pp.591-594, Dec. 1993.

[12] T. Matsumoto et. al., “Distributed Shared Memory Architecture for JUMP-1: A General-Purpose MPP Prototype,” International Symposium on Parallel Architectures, Algorithms and Networks, pp.131–137, 1996.

[13] T. Matsumoto and K. Hiraki, “Memory-Based Communication Facilities and Asymmetric Distributed Shared Memory,” Innovative Architecture for Future Generation High-Performance Processors and Systems, pp.30–39, 1997.

[14] K. Li., “A Shared Virtual Memory System for Parallel Computing” In. *Conf. on Parallel Processing, St. Charls, IL*, pp.94–101, August 1988.

[15] 井上浩明他,「超並列計算機 JUMP-1 における MBP Core Architecture の評価,」CPSY, pp.1–8, January, 1998.

[16] A.V. エイホ R. セシィ J.D ウルマン共著 原田 賢一訳 “コンパイラ” 1990.