解　説

新しいアーキテクチャとコンパイラ技術†

久　野　　靖††

1. はじめに

近年，計算機アーキテクチャの分野においてさまざまな新しいアイデアが提案され，そのようなアイデアを取り入れた計算機システムが市販されることも多くなっている．そのようなシステムのうちには，Lisp マシンや Prolog マシンに代表される，専用の言語を効率よく実行することに重点を置いたシステム☆ もあるが，一方で Fortran, Pascal, Cなど旧来の手続き型言語をコンパイルして高速に実行させることに重点を置いたものも多く見られる．本解説ではこの後者の種類のシステムを中心に取り上げる．

後者の種類のシステムでは，手続き型言語を新しいアーキテクチャで効率よく実行できるように翻訳する技術が重要な位置を占めている．言い換えれば，そのようなコンパイラ技術の進歩なくしてはこれらのシステムは日の目を見なかったであろう．また，これまでのアーキテクチャは「それ自身としてのよしあし」を基準として開発されてきたのに対し，現在は「コンパイラと組み合わせたトータル性能」を基準とするように変わりつつある時期である，ともみられる．このように，コンパイラ技術の動向が実用機として市販されるアーキテクチャの動向を左右する，というのはこれまでに例をみなかったことである．

本稿は以上のような見解のもとに，コンパイラ技術の側からみて新しいアーキテクチャに対応していくためにどのような進歩があったかを中心に解説を試みる．一方，具体的な個別のアーキテクチャとコンパイラの関係についての細部は，本特集各編[1)~4)]を参照されたい．以下，2.では新しいアーキテクチャの諸側面としてどのようなものがあり，それがコンパイラとどのように関連しているかを概観する．続いて3.で特定のアーキテクチャからは離れて，主として新しいアーキテクチャのためのコンパイラにみられるコンパイラ技術全般について取り上げる．4.では2., 3.で取り上げた点が具体的なコンパイラでどのように実現されているかを，実例をもとに概説する．最後に5.でまとめを行う．

2. 新しいアーキテクチャと新しいコンパイラ技術

本章では，前章で述べたような新しいアーキテクチャにみられる諸側面としてどのようなものが存在するか，またそれらがコンパイラ技術とどのように関連しているかについて概観する．なお，ここで述べる諸側面のいくつかは既存の汎用大型機などにもみられ，その意味で全てが目新しいというわけではないが，その活用のしかたという面でコンパイラ技術の助力が今後一層重要になると思われたのでここに含めた．

(1) パイプライン

パイプラインとは，CPU が一つの命令の実行完了を待たずに次の命令，その次の命令，...の処理を開始することにより，命令の実行間隔を短縮し，実質的な CPU の命令実行速度を高める方式を指す．パイプラインそのものは現在の計算機システムに広くみられ特に新しいものではないが，CPU の高速化にともない，先取りされる命令の個数(パイプラインのステージ数)の多いシステムが実用化されるようになった．この結果，実行を開始した命令が使用する資源への干渉が後から分かって，その命令を一時待機させたり（インタロック），条件分岐などによりパイプライン中の命令を全て取り消したり（フラッシュ）するための性能低下が問題となるようになった．このためパイプラインの特性を考慮して命令をスケジュールする機能をコンパイラにもたせることが行われるようになった[5),13)]．

また，従来の命令セットアーキテクチャの意味付けを変更しないようにインタロックやフラッシュを正し

† Recent Progress on Compiler Technology and its Relation to New Computer Architectures by Yasushi KUNO (Graduate School of Systems Management, Univ. of Tsukuba).

†† 筑波大学経営システム科学専攻

☆ アーキテクチャの性能を引き出すために専用の言語を使用する，という場合もこの一種であろう．

く処理してゆくためのハードウェアの負担と性能への影響も無視できない．このためアーキテクチャ定義の一部としてパイプラインの性質の一部を取り込み，遅延分岐*や遅延ロード**をもつアーキテクチャが作られている[7),23)]．このようなアーキテクチャでは分岐スロットやロードスロットに有効な命令を埋め込むことではじめて所期の性能が発揮されるが，そのような命令配置をハンドコーディングに期待することは非現実的であるので，このようなアーキテクチャは最適化コンパイラによる命令スケジューリングを前提としたものといえる．

（2） キャッシュメモリ

キャッシュメモリとは，大容量だがアクセスに時間のかかる主記憶と CPU の間に小容量だが高速のメモリを介在させ，一度主記憶から転送された値をここに記憶させておき，再度同じ値が必要とされたときにはここからデータを供給することにより実質的な主記憶アクセス速度を高め CPU の高速動作を可能にする機構である．キャッシュそのものは従来のシステムでも多く採用されているが，最近は参考文献 10)などにキャッシュの存在を前提とし，ループなどで高密度に使用されるコード群が互いに衝突せずキャッシュに収まるよう配置するコンパイラ機能の研究がみられる．特に近年はキャッシュを2レベル以上に構成し，容量は限られているが非常に高速なオンチップキャッシュ（CPU と同じ VLSI チップ上にキャッシュを実装したもの）を搭載するシステムもあるので，このような技術は重要であると考えられる．また，文献 10)には命令に「キャッシュへの格納回避ビット」を含めることの提案もあり，現在のようにハードウェアにより動的に内容を制御するキャッシュ技術から完全にソフトウェアにより内容や一貫性を制御するキャッシュ技術への変転も考えられる．

（3） 多数のレジスタ

数個ないし十数個程度の汎用レジスタをもつアーキテクチャは特に珍しくないが，近年においてはさらに多数（数十個以上）のレジスタをもつものが実用化されている．レジスタの役割は CPU に近接して高速のデータ記憶をもたせることで時間のかかるメモリアクセスを回避することだが，数十個以上のレジスタとなると，それを有機的に活用するよう手でコーディングすることは考えられず，したがってコンパイラによるレジスタの活用が前提となるという質的変化をもたらした．

また，従来から最適化コンパイラはレジスタ上に変数値を置くことでメモリとレジスタ間の転送を少なくする機能を提供してきたが，数十以上のレジスタとなると一つの手続き内での変数をすべてレジスタに割り当ててもまだ剰余があることになり，複数の手続き間を通じたレジスタ割り当てが重要となっている．この方向付けは手続き呼出が頻繁に行われるという近年のプログラミングスタイルと関連しても重要である．というのは，一つの手続き内でレジスタを活用したとしても，多数のレジスタを手続き呼出のたびに退避回復するのではその効果が相殺されてしまうからである．これに対し手続き間レジスタ割り付けでは，ある手続きが使用するレジスタがそこから呼び出される手続きの使用するレジスタと干渉しないよう割り当てることで退避回復を省略できる．一方，同じ問題へのハードウェア面からのアプローチとして，多数のレジスタをスタックフレームに対応する「窓」を通じてアクセスし，手続き呼出に際しては窓の位置を変更する機構をハード的に用意することで退避回復を不要とするものもある[7)]が，コンパイラによる手続き間レジスタ割り付けで同等以上の効果があるため，ハードウェアの負担を増すやり方は得策でないとの論もある[8)]．

（4） RISC

RISC とは Reduced Instruction Set Computers の略であり，1970年代ころまでのアーキテクチャがマイクロコードを多用し，多数のアドレッシングモードを含む豊富な命令体系をもっていた（RISC に対して CISC——Complex Instruction Set Computers と呼ばれる）ことに対するアンチテーゼとして近年提唱されるようになった概念である[7)]．

RISC を支持する論拠としては，CISC でマイクロコードにより各命令を解釈実行するよりは単純な命令体系をマイクロ命令と同速度で直接ハードウェアにより実行したほうが高速であること，CISC の豊富な命令体系やアドレッシングモードは実際にはコンパイラの生成コード中ではあまり活用されず無意味であること，それよりはハードウェアを単純化しシリコンチップ上の資源を他の目的（オンチップキャッシュ，多数のレジスタ，多数の演算器など）に利用するほうが得策であること，などがあげられている．

---

* 分岐命令があっても，それに続くいくつかの命令は分岐に先立ち常に実行されるアーキテクチャの性質．それらの命令は分岐スロット内にある，という．

** メモリからレジスタへの転送命令があってもその次のいくつかの命令では転送された値は利用可能でないようなアーキテクチャの性質．それらの命令はロードスロット内にある，という．

RISC の定義には諸説があるが，たとえば文献7)では初期の RISC にみられる共通した特性として次のようなものをあげている．

a）メモリをアクセスする命令は load と store に限定され，そのほかの演算命令はレジスタ間命令である．

b）命令の種類，アドレッシングモードが少ない．また命令の実行は1マシンサイクル1命令が基本．

c）命令形式が単純で，語境界にまたがらない．

d）パイプラインの特性を反映した命令の意味づけ．

これからも分かるように，RISC の思想自体は，画期的な新しい技術というより，これまでに知られてきた技術を見直し，うまく再構成することで高速なシステムを作ろうとする思想であると言えよう．またハードウェアで単純化した部分についてはソフトウェア（つまりコンパイラ）でうまく補完する，という思想を併せもっていることも特記しておきたい．

（5） ベクトルプロセッサ

ベクトルプロセッサの基本的な考え方は，ベクトルや行列など多数のデータが規則的に配列されているもの同士の演算を，ベクトル命令と呼ばれる一つの命令で指定することで演算器に連続的にデータを供給し，結果的に短時間に多数の演算を可能にすることにある．

このようなアーキテクチャが発表された当初は，ベクトル命令がプログラマによって実際にどの程度有効に活用されるかを危ぶむ向きもあったが，現実には通常の手続き型言語コードのループ部分を自動的に抽出してベクトル命令に変換するベクトル化コンパイラ[1),5)]の普及により多くのプログラマが容易にベクトル命令の恩恵に浴せるようになり，それとともにベクトルプロセッサも大きな普及をみている．

（6） マルチプロセッサ

一つの計算機システム内部に複数の CPU をもつものを総称してマルチプロセッサと呼ぶ．マルチプロセッサそのものも特に新しいものではないが，従来の大型汎用機や TSS マシンにおけるマルチプロセッサの目的は主として複数のタスク/プロセスを複数の CPU で並行して処理することによりシステム全体のスループットを向上させることにあった．タスク／プロセスは互いに比較的独立しているからこのような適用はやさしいが，しかしこのようなシステムでは各プログラムの実行時間が短くなるわけではない．

これに対し，一つのプログラムの内部で複数の CPU を活用すればその分だけ高速化することは可能であるが，手による並列化は労力もかかり虫取りもむずかしくなるなどの問題がある．そこで通常のコードからコンパイラが並列性を抽出し，複数の CPU に振り分ける，（ベクトル化コンパイラと同じような意味での）並列化コンパイラに対する研究も盛んである[2),9),10)]．

また現状では主としてプログラミングのしやすさから共有メモリをもつマルチプロセッサが主流であるが，今後 CPU 数が増えてきた場合共有メモリの実現には限界があるので非共有メモリマルチプロセッサが取って代わる可能性があり，そのような場合にはデータの交換，通信路の制御，より多くの CPU を活用する並列性の抽出などの要求が出てくる．これらをハードウェアや実行時サポートソフトウェアを通じて動的に処理することも考えられるが，コンパイラにより翻訳時に静的に割り当てを行うというのも有力なアプローチである．

（7） データフロープロセッサ

データフロープロセッサでは命令の実行順序は通常のプロセッサのように命令の置かれている順序ではなく，命令が扱うデータの依存関係によって定まる．言い換えれば，ある演算はそれが必要とするデータが揃った時点でただちに起動されるので，全体として高速実行が可能であることが期待される．また，パイプライン，ベクトルプロセッサ，並列化などでは命令の順序実行の高速化を計るあまりデータ依存関係を壊さないよう注意することが大きな問題となるのに対し，データフローの場合には原理的にそのような問題が起きないという利点がある．

一方，プログラマがいきなりデータフローグラフを作成することによりプログラミングを行うというのは非現実的であるので，なんらかの高級言語からデータフローマシンが直接実行できる形に翻訳して実行する，という過程はどうしても不可欠である．この高級言語として普通の手続き型言語が使用できれば理想的であるが，現状では手続き型言語が命令の順次実行を前提としていることからデータフローとのギャップは大きく，関数型言語や単一代入言語などデータフローとなじみのよい言語を設定してこれによるプログラミングを要求するものが多い．しかしながらこれらの専用言語をできるだけ普通の手続き型言語に近づける努力も着実に進められている．またデータ依存関係に対応した同期についても，コンパイラによる解析情報を利用することでハードウェアの負担を減らす試みがある．

(8) VLIW

VLIW (Very Long Instruction Words) とは多数の演算ユニットを独立に制御できる機構のアーキテクチャ*を総称したものである．VLIW の場合にはベクトルプロセッサやパイプラインなどと異なり，多数の演算機をどのように並行して利用するかは完全にソフトウェアにゆだねられており，しかもそのようなコーディングをプログラマが手で行うことはきわめて困難とされているので，コンパイラによる命令スケジューリングに頼らざるをえない[11),12)]．

見かたを変えれば，VLIW とはパイプラインアーキテクチャにおいては主としてハードウェアが行っていた演算器間の並列動作の管理を全面的にソフトウェアによる静的スケジューリングに置き換えたものと考えられ，これまでにあげてきたどれにも増してコンパイラに依存したアーキテクチャであるといえる．

3. 新しいコンパイラ技術の諸側面

前章では新しいアーキテクチャの諸側面とそれに対応するコンパイラ技術との関連について述べたが，本章ではコンパイラ側からみたこれら諸技術の関連と構成について系統的にまとめる．ただしコンパイラの諸部分のうち，フロントエンド部分（プリプロセッサ，字句解析，文解析，構記号表，意味解析）については，言語仕様が同じである以上従来のコンパイラと変わるところはないので特に触れない．

(1) データフロー解析

データフロー解析はコード中のどの値がどこで定義され，どの経路を通ってどこで参照されるかを調べるものであり，その情報は後に列挙する各種の最適化に不可欠である．原理的にはデータフロー解析は，連続して実行され分岐を含まないコード列である基本ブロックを節とし，分岐を有向アークとするグラフを考え，その上で各基本ブロックにおいて定義される値の集合，参照される値の集合を計算してゆく形で行える[14),15)]．

ただし，古典的な最適化コンパイラではそのような形で十分であったとしても，たとえばループのベクトル化などを行う場合にはどの部分がループであるかなどの情報を残しておく必要がある．また別の行き方としていったん節とアークのみのグラフの形に落としたものからもとの if, while などの制御構造に相当するものを再び構築して（インタバル解析）これに基づいて参照関係の計算を進める方法もある[15)]．特に手続き間データフロー解析を行う場合には，一時に扱うコードの量が大きいので，実用的な速度で解析を行うためには計算量のオーダが小さいアルゴリズムや効率よく計算を進めるための工夫が重要となる．

(2) 手続き内広域最適化

定数式の伝播，共通式の削除，ループ外への不変式の移動，乗除算から加減算への変換，ループの展開，など古典的な最適化がここに含まれる[26)]．このような最適化を「広域」と呼ぶのはプログラムの基本ブロックをまたがってコードの移動などを行うことからきているが，手続き間にまたがって最適化を行うことが一般化しつつある現在では不適切な命名であると言える．

広域最適化そのものは確立した技術であるが，新しいアーキテクチャの導入にともなってその適用法も変更が必要となる．たとえばループ展開を例に取ると，ベクトル化を行う場合にはループはもとの形で残しておく必要があり，また展開によりループ本体のサイズが大きくなりキャッシュに入らずかえって遅くなる場合もあるなど，アーキテクチャの特性を考慮して各最適化の適用を判断する必要が生じている．

(3) ベクトル化

ベクトル化は多くの場合ループ構文をベクトル命令に置き換える形で行われるが，無条件に置き換えを行うと，プログラムの意味が変化してしまう場合がある．新しい言語仕様では (Fortran 90 のように) ベクトル命令への置き換えを明示的に許す構文をもつ場合もあるが，ほとんどの場合にはコンパイラが各種の情報を利用してプログラムの意味を変化させない範囲で自動的にベクトル化を行うことが重要となる．具体的なベクトル化手法については文献 1),9),14) などを参照されたい．

また，これまで多くのコンパイラはループが手続き呼出を含むときはベクトル化を断念していたが，この弱点を克服するため手続きにまたがった解析を行うことも実用化されつつある[14)]．さらに，これらの手法では再内側のループのみを対象とされることが多かったが，これをさらに外側のループについてまで拡張する手法も研究されている．

(4) 並 列 化

手続き型言語で最も明確に並列性が見いだせるのはループ内部なので，並列化もループの並列化（一つの

☆ 各ユニットを制御するフィールドを並行して含んだ命令語は長くならざるをえないので必然的に Very Long Instruction Words となるわけである．

ループの周回を N 個のプロセッサでそれぞれ $1/N$ ずつ分担して実行する）の形で行われることが多かった[2]．さらに各 CPU がベクトルプロセッサならベクトル化も併用できる[5]．いずれにせよ，この種の並列化ではループ内のデータ依存関係を解析する必要があるのはベクトル化と同様である．また，ループのほかにも任意のコードセグメントに対して並列性を抽出する可能性があり，たとえば手続き呼出レベルでの並列化（呼び出す側と呼び出される側を並列に実行する）なども試みられている[6]が，この場合には両者の間の手続き間依存関係の解析が必要となる．一方，VLIW などではより小さい式レベル，文レベルの並列性抽出が行われる[11),12)]．いずれにせよ自動並列化についてはまだ負荷分散の方式，並列の細かさ（粒度）の選定，同期オーバヘッドの低減など多くの課題が残されている状態である．

（5） インライン展開

手続き呼出をまたがって最適化を行うやり方の一つに，呼び出される手続きを呼出の場所にその場展開する方法がある．これは単純で実現も容易であり，実用化されている例もある．ただし，その場展開には，展開にともないコードサイズが大きくなる，コードが一体の巨大な手続きに変換されてしまうので翻訳時間が長くなるなどの問題点がある．

これに対処する一つの方法としてソースレベルで展開するのではなく，中間コードレベルで展開を行いその後最適化フェーズを実行するという方式もある[16]．このやり方を取ることにより同じ字面を繰り返し字句解析から実行するという無駄をなくすことができる．またオブジェクトコードの大きさの問題については，手続き間情報の流通により展開の起きる回数と展開される手続きの大きさを考慮して選択的に展開を行うという方法があり得る．

（6） 手続き間最適化

近年では，プログラムコードが小さい，多数の手続きからなることが多くなってきた．一方，最適化技術の多くはデータや制御の流れを克明に追跡することを前提としているので，手続き呼出内部をブラックボックスとして扱うかぎりその近辺のコードに十分な最適化を施せない．このため必然的に，複数手続き間にまたがって最適化情報を流通することが必要となる．ただしプログラムを構成する全手続きを一つにまとめてデータフローグラフなどを作成するのでは作業量が大きすぎるので，各手続き単位でサマリを作成して必要な箇所で参照する，という形で「局所」部分と「手続き間」部分を分けることが一般的である[27]．

また，手続き間情報を流通するためには各モジュールを独立に翻訳してオブジェクトコードに落としライブラリと併せて結合する，という旧態依然のフェーズ構成では対応できないことはもちろんであり，このため全ソースコードをソースデータベースに入れてコンパイラの管理下におき必要に応じてここから各種情報を抽出するようなシステムなどもある[28]．

（7） レジスタ割付け

レジスタは貴重な資源であるので，与えられたデータ群のアクセスにできるだけ少ないレジスタで対応することがレジスタ割付けの目的となる．最小限のレジスタに値を割当てる問題は各値をグラフの節とし，それらの間で同時に生きているもの同士には辺があるとして，そのグラフを最小限の色で塗る彩色問題として定式化できる．この問題自体は NP 完全だが，実際のコンパイラではヒューリスティックなどを使用して実用的な時間で処理を行う[21]．それでもグラフが大きくなる手続き間レジスタ割付けでは効率よく計算するのはやさしくない．また，レジスタの数が足りない場合，どの値を一時的にメモリに退避する（スピル）かも問題である．その判断基準として単なるコード上の参照回数では必ずしも正確でなく，プログラムのサンプル実行の結果を適用するのが有効であるとの研究もある．

（8） コード生成

コード生成の役割は各ターゲットアーキテクチャに対応した機械語を生成することだが，近年ではテーブルドリブンのコード生成機構により，コンパイラ自体は機械独立でターゲットアーキテクチャの機械記述を書くことで複数のアーキテクチャに対応するコンパイラも多い[17]．また，コード生成後にピープホール最適化などによりコンパクトなコードへの変換を行うことを前提として「素直な」コード生成を行うことが多いのでひと頃のように機械依存のケース分析を行うことは少なくなりつつある．

（9） ピープホール最適化

ピープホール最適化とはコード列を局所的にみてゆきながら隣接する命令列をより良い命令列に取り替えてゆくものであるが，現在では上述のコード生成に対する位置づけと相応して，最終的な命令フォーマットに合わせて単純な命令群を詰め合わせて組み立てる，という意味合いももつようになっている[18]．その実行

は形式的には条件を満たす命令列を別の命令列に置き換え，その結果の中で再び置き換えられる命令列を探すというパターンマッチと置き換えの反復であるが，これを時間の掛かるパターンマッチを避けて高速に行う手法もある[29]．

(10) 命令スケジューリング

命令スケジューリングは基本的にはターゲットアーキテクチャのパイプラインなどの特性を考慮して命令列ができるだけ短時間で実行されるように配置することである．このイメージに近いものとして生成済みの機械語コードをパイプライン特性に合わせて並べ替えたり遅延分岐，遅延ロードのスロットを有効利用するように命令を移動して不要な NOP 命令を削除するなどの作業がある．しかし，さらに進んでパイプラインや演算器をできるだけ有効活用するように演算の起こる順序そのもののレベルからスケジュールを行ってゆくことも広い意味での命令スケジューリングに相当するといえよう．具体的には軌跡スケジューリングやソフトウェアパイプライニングなどがこれに相当する[11),12)]．

(11) 分岐予測

パイプラインをもつ計算機において分岐命令は（たとえ遅延分岐を取り入れても）これまでの命令列と異なる場所から新しい命令列を取り出すためパイプラインの流れを乱し，実効性能を低下させる．特に条件分岐の場合にはそれが実行されるまでは分岐する側の命令列としない側の命令列のどちらを先行取得してよいか分からない．しかし実際のコードでは分岐する場合としない場合のどちらかが多く起こるのが普通なので，的確にそちらの枝を選択して先行取得を行えば分岐のペナルティを低く押さえることができる．そのために，各分岐命令ごとにどちらへの分岐が起きたかを記録しておく，というハードウェア的アプローチもあるが，コンパイラの側でその確率を予測して命令中の「分岐するらしいビット」を適切に設定する，という方式も存在する[16)]．現状ではたとえばループ脱出の条件分岐はループ反復側に予測するなどの単純な方策が実用化されているが，記号実行などによるより精密な判定も将来的には可能かも知れない．またサンプル実行のデータをもとに決める方式も有力である．

(12) 記憶配置

機械語の命令列が定まったとしても，そのメモリ上の配置については依然として大きな自由度が存在する．一方，仮想記憶などを考えた場合，関連して実行されるコードができるだけ近接して配置されていることが有利なのでコンパイラによってはそのような配置を実際に行おうとするものもある．また仮想記憶ではなくキャッシュメモリの特性を考慮し，関連して実行される命令列がうまくキャッシュ内に共存するよう配置することも提案されている[6)]．このような技術は高速なオンチップキャッシュの有効利用のために重要である．

(13) プロファイルの利用

コンパイラが行う最適化のうちには，コードが実際にどのようにふるまうかを知ることでより効果的に行えるものが多い．たとえば値に対する参照の動的頻度が分かっていれば，レジスタからどの値をメモリにスピルするのが効果的かを的確に予測できる．また各条件分岐命令ごとに分岐が起こる確率があらかじめ分かっていれば（分岐予測ビットのような最小限のハードと組み合わせて）実際に進む確率が高い側をフェッチすることでパイプラインのフラッシュを減らすことができる．

このようなことを実現するには，典型的な入力データでコードを実際に走行させそこから必要な情報を採取すれば良い．プログラムを実行させその挙動のサマリを採取するのはプロファイル機能として広く行われていることであるが，これまではその情報はプログラマが参照してコードの手による調整を行うためにしか使われないことが多かった．しかし，このような手法で高価なハードウェアによる工夫と比較して遜色ない全体性能の向上が得られる例も報告されているので，今後はコンパイラフェーズの一部としてプロファイルの採取とその結果による最適化を組込むものが増えることが予想される[6),16)]．

4. 実 例

ここまでで新しいアーキテクチャの諸側面，およびそれらを活かすために必要とされるコンパイラ技術についてまとめてきたが，本章では具体的なイメージを読者にもっていただくことを目的として，一つのコンパイラの例を取り上げて簡潔に解説する．ここで取り上げるのは MIPS 社の RISC プロセッサ用コンパイラであるが，RISC という側面にとらわれず本解説で述べたような技術をバランスよく取り入れていること，市販システムとして使用されていて完成度も高いこと，実用のコンパイラとしては異例に入手可能な文献が多く詳細が分かること[19)~23)]から選択した．以下

ではこのコンパイラの各フェーズについて，前章までに述べてきたコンパイラ技術がどのように組込まれているかを中心に概観する．

（1） フロントエンド

このコンパイラのフロントエンド部はC，Pascal，Fortran 77，PL/I，COBOL，Ada の各言語にそれぞれ専用のものが用意され，いずれもが構文解析，意味解析，および共通の中間コード生成までを行う．中間コードは U-Code と呼ばれ，スタック仮想マシンふうの言語となっている．これ以降のフェーズはどの言語についても共通であり，これにより多用な最適化を必要とし開発コストの高いバックエンド部を一つで済ませるとともに，どの言語に対しても同じレベルでの最適化を提供し，複数の言語間でのリンケージも問題なく実現でき，それら複数の言語で書かれた手続き間の広域最適化も結果として可能になる．

（2） 中間コードリンカ (uload)

このフェーズではフロントエンドが生成した複数の中間コードファイルを連携編集して，一つの中間コードファイルにまとめることができる．手続き間最適化には対象となる全手続きの情報が必要だが，このフェーズで中間コードファイルをまとめることで，後段の最適化フェーズでは一つの中間コードファイルに含まれる複数モジュール相互の手続き間最適化が行える．また手続き間最適化が必要ない場合にはこのフェーズを省略して従来どおりのリンカにゆだねることもできる．

このフェーズでは単に複数のファイルを「束ねる」だけではなく連携編集に相当する作業も行われる．たとえば統合される中間ファイル内のみで参照される広域記号は局所記号に変換される．これにより広域変数が局所変数，広域手続きが局所手続きに変更されるので，後段の最適化フェーズにおいて外からの干渉の恐れなくこれらの変数や手続きの最適化が行える．この作業を正しく行うには外部から記号への参照の有無を知る必要があるため，統合される中間ファイルだけでなく後段で結合される中間ファイル／オブジェクトファイルやライブラリも正しく指定する必要がある*．

（3） 手続き展開 (umerge)

このフェーズは手続きのその場展開を行うが，もしその必要がなければ uload と同様省略できる．手続き展開も中間コードレベルで実行されるため，異なる言語で書かれた手続きもその場展開できる．1回しか呼ばれない手続きは常に展開したほうが有利であるが，複数回呼ばれる場合にはコードサイズの問題のため手続き本体の大きさと呼ばれる回数の兼ね合いを考慮する．

また，中間コードファイルに含まれる手続き群を呼出関係でみて下から上の順に並べ替えることも行う．これは後段のフェーズでコード中の手続き呼出を見たときに，その手続きが処理済みでそれに関する情報が参照できることが望まれ，下から上の順に並べてあればそれが自然に達成されるからである．

（4） 最適化 (uopt)

このフェーズではデータフロー解析を行い，共通式の削除，乗除算の加減算への置き換え，不変式の移動，不要なコードの削除など通常の手続き内最適化を行うが，その際おなじ中間コードファイルに含まれている手続きの情報も活用する．このコンパイラはベクトル／並列プロセッサのためのものではないのでベクトル化や並列化は行わないが，そのような機能の必要があればこのフェーズに組み込むことは原理的に可能である．

また，このフェーズではデータフロー解析の結果を利用してレジスタ割り付けも行う．レジスタ割り付けは呼出関係において下に位置する手続きから順に行い，上の手続きでは下の手続きに割り当てたレジスタをよけて配置することで，呼出におけるレジスタ退避回復をできるだけ行わないように配慮する．

（5） コード生成 (ugen)

このフェーズでは中間コードレベルでの最終的な局所最適化を行った後，コード生成を行い，アセンブリ言語形式のファイルを出力する．

（6） アセンブリ (asl)

多くのシステムではアセンブラは入力の各命令をそのまま1対1にオブジェクトコードに変換するが，このコンパイラではこのフェーズでまず機械語レベルでのピープホール最適化を行い，続いて分岐スロットやロードスロットのための命令の再構成とスケジューリングを行う．アセンブリ言語レベルでの命令列と機械語での命令列の順序が対応していないとアセンブリ言語のプログラマにとっては不安かも知れないが，逆にアセンブラのプログラムに対しても可能な最適化は適用されることになる．また，命令スケジューリングがこの段階で行われることは，アセンブリ言語レベルでの命令セットアーキテクチャは同一だが実行時の命令

* ライブラリは中間ファイル形式のものとオブジェクト形式のものの両者が用意される．中間ファイルライブラリとして取り込まれた手続きに対してはその場展開，手続き間最適化が行われる．

のタイミングや分岐スロットの扱いなどが異なるファミリ機の可能性を示唆しているようにも思える．これは従来のファミリ機開発に際しての，命令タイミングなどを既存機種と互換にするための労力をソフトウェアに移すものといえる．

（7） 実行時規約の設計

一般に実行時規約はできるだけ手続きの呼びと戻りのオーバヘッドを少なくするよう設計されることが望ましいが，このコンパイラではそのためのサポートを多くコンパイラ側で提供している点に特徴がある．

まず，多くのシステムと異なり，スタックフレーム上にある変数をアクセスする起点となる環境ポインタレジスタを原則として使用しない．代わりに，環境ポインタとスタックポインタの値の差はほとんどの場合翻訳時に決定できることを利用し，スタック上の変数はスタックポインタからのオフセットを用いてアクセスする*．スタックポインタ自身も，手続きの入り口で必要な最大値ぶん動かし，出口で戻す以外には値を変更しない．

次に，呼出側と呼ばれる側で合意があれば引数はいくらでも多くレジスタ渡しとできる．また，手続き呼出において値を保存するレジスタの組は一応約束されているが，手続き間レジスタ割り当てで互いに使用するレジスタをよけて配置した場合には退避回復を行わない．さらに局所変数がすべてレジスタにのれば，戻り番地の格納を除きスタックをアクセスする必要はなくなる．そして葉の（手続き呼出を含まない）手続きでは戻り番地もレジスタに置いたままにでき，スタックポインタを動かす必要もなくなる．

このように最大限不要な命令を排除する結果，最短で手続きの呼びに1命令，戻りに1命令で済むようになるが，その代わり変数のスタックポインタからの変位（または置かれているレジスタ）はコード上のどこを実行しているかに依存することになる．そこでコンパイラはこれらの情報をファイルに出力し，デバッガはこれを参照して各値のありかを正しく見つける．

5. ま と め

全体として，新しいアーキテクチャとしては多様なバリエーションがあるものの，それら共通にみられる性質として，アーキテクチャ自体はハードウェアによってできるだけ効率よく実現できることを目指し，そのアーキテクチャ／ハードウェアにより効率よく動くコードを出すという仕事はますます多くをコンパイラの機能に依存するという傾向がある．より具体的には，これまではハードウェアによって行われてきたさまざまな資源の動的スケジューリングをコンパイラによる静的なスケジューリングに置き代える，というのがその基本的方向であるように思える．それを実現するための基礎的な研究も多く行われているが，最終的にはコンパイラは日々のプログラミングに欠かすことのできない道具であり，その道具としての使いごこちを損なわずにいかにしてこれらの新しい機能を取り込んでゆくか，という点で機能とシステム全体としての完成度とのバランスが求められ，それだけに難しくもあり面白くもある分野であるといえる．

* スタック上に動的配列を取る手続きでは例外的に環境ポインタを用いる．

参 考 文 献

1) 田中，安村：ベクトル計算機のためのコンパイラ技術，情報処理，Vol. 31, No. 6 (1990).
2) 本多：並列アーキテクチャのためのコンパイラ技術，情報処理，Vol. 31, No. 6 (1990).
3) 関口，山口：データ駆動計算機のためのコンパイラ技術，情報処理，Vol. 31, No. 6 (1990).
4) 中谷：VLIW 計算機のためのコンパイラ技術，情報処理，Vol. 31, No. 6 (1990).
5) Gibbons, P. B. and Muchnick, S. S.: Efficient Instruction Scheduling for a Pipelined Architecture, Proc. ACM SIGPLAN 1986 Symp. on Compiler Construction, pp. 11-16 (1986).
6) McFarling, S.: Program Optimization for Instruction Caches, Proc. 3rd Inter. Conf. on Architectual Support for Prog. Lang. and Oper. Syst. (ASPLOS III), pp. 183-191 (1989).
7) Patterson, D. A.: Reduced Instruction Set Computers, CACM, Vol. 28, No. 1, pp. 8-21 (1985).
8) Wall, D. W.: Register Windows vs. Register Allocation, Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation (PLDI '88), pp. 67-78 (1988).
9) Padua, D. A. and Wolfe, M. J.: Advanced Compiler Optimizations for Supercomputers, CACM, Vol. 29, No. 12, pp. 1184-1201 (1986).
10) Triolet, F., Irigoin, F. and Featurier, P.: Direct Parallelization of Call Statements, Proc. ASM SIGPLAN 1986 Symp. on Compiler Construction, pp. 176-185 (1986).
11) Fisher, J. A.: Very Long Instruction Word Architectures and the ELI-512, Proc. 10th Ann. Symp. on Compt. Arch., pp. 140-150 (1983).
12) Lam, M.: Software Pipelining: An Effective

Scheduling Technique for VLWI Machines, Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation (PLDI '88), pp. 318-328 (1988).
13) Hennessy, J. and Gross. T.: Postpass Code Optimization of Pipeline Constraints, TOPLAS, Vol. 5, No. 3, pp. 422-448 (1983).
14) Li, Z. and Yew, P-C.: Efficient Interprocedual Analysis for Program Parallelization and Restructuring, Proc. ACM SIGPLAN PPEALS pp. 85-97 (1988).
15) Jain, S. and Thompson, C.: An Efficient Approach to Dataflow Analysis in a Multiple Pass Global Optimizer, Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation (PLDI '88), pp. 154-163 (1988).
16) Hwu, W. W., Conte, T. M. and Chang, P. P.: Comparing Software and Hardware Schemes for Reducing the Cost of Branches, Proc. 16th Ann. Symp. on Comp. Arch., pp. 224-238 (1989).
17) Glanville, R. S. and Graham, S. L.: A New Method for Compiler Code Generation, Fifth ACM Symp. on Principles of Programming Languages, pp. 509-514 (1978).
18) Fraser, C. W. and Wendt, A. L.: Integrating Code Generation and Optimization, Proc. ACM SIGPLAN 1986 Symp. on Compiler Construction, pp. 242-248 (1986).
19) Chow, F., Himelstein, M., Killian, E. and Weber, L.: Engineering a RISC Compiler System, Proc. 1986 COMPCON, pp. 132-137 (1986).
20) Himelstein, M. I., Chow, F. C. and Enderby, K.: Cross-Module Optimizations: Its Implementation and Benefits, Proc. Summer 1987 USENIX Conf. pp. 344-356 (1987).
21) Chow, F. and Hennessy, J.: Register Allocation by Priority-based Coloring, Proc. ACM SIGPLAN 1984 Symp. on Compiler Construction, pp. 222-232 (1984).
22) Chow, F. C.: Minimizing Register Usage Penalty at Procedure Calls, Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation (PLDI '88), pp. 85-94 (1988).
23) Chow, F., Correl, S., Himelstein, M., Killian, E. and Weber, L.: How Many Adressing Modes are Enough?, Proc. 2nd Int. Conf. on Architectural Support for Prog. Lang. and Oper. Syst. (ASPLOSII), pp. 117-121 (1987).
24) Kane, G.: MIPS RISC Architecture, Prentice-Hall (1987).
25) Hennessy, J. L., Jouppi, N., Baskett, F., Gross, T. R. and Gill, J.: Hardware/Software Tradeoffs for Increased Performance, Proc. Architectural Support for Prog. Lang. and Oper. Syst. (ASPLOS), pp. 2-11 (1982).
26) Aho, A. V., Sethi, R. and Ullman, J. D.: Compilers: Principles, Techniques, and Tools, Addison-Wesley (1986).
27) Callahan, D.: The Program Summary Graph and Flow-Sensitive Interprocedual Flow Analysis, Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation (PLDI '88), pp. 47-56 (1988).
28) Cooper, K. D., Kennedy, K. and Torezon, L.: The Impact of Interprocedual Analysis and. Optimization in the Rn Programming Environment, TOPLAS, Vol. 8, No. 4, pp. 491-523 (1986).
29) Fraser, C. W. and Wendt, A. L.: Automatic Generation of Fast Optimizing Code Generator, Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation (PLDI '88), pp. 79-84 (1988).

(平成2年2月5日受付)