計算機アーキテクチャ　104－6
システムソフトウェアと
オペレーティング・システム　62－6
（1994．1．27）

RISC用OS／omicronにおける言語C処理系とシステムソフトウェア開発・評価環境

田中広幸，中村浩之，早川栄一，並木美太郎，高橋延匡

（東京農工大学　工学部　電子情報工学科）

本稿では，RISC用言語C処理系とシステムソフトウェア開発・評価環境について述べる．
我々の研究室で行われている研究から，システムの高速化に対する要求が高まっており，それに対し研究のためのRISCシステムを考えた．まず，そのRISCシステムの中で，すべてのソフトウェアの開発言語である言語C処理系の実現と，そのサポートのためにシミュレータを中心とする開発・評価環境を構築した．この開発・評価環境では，RISCプロセッサの特徴であり，性能に大きく関係するレジスタインターロックやディレイドスロットの測定などを行うことができる．

A Language C Compiler and an Environment for the Development and Evaluation of Systems Software on a RISC-Oriented OS／omicron

Hiroyuki Tanaka, Hiroyuki Nakamura, Eiichi Hayakawa,
Mitarou Namiki and Nobumasa Takahashi

Department of Computer Science,
Faculity of Technology,
Tokyo University of Agriculture and Technology,
2-24-16, Naka-cho, Koganei-shi, Tokyo, 184 Japan

In this paper we describe a language C compiler and an enviroment for the development and evaluation of systems software for a RISC architecture.
In our research there is a need for high performance systems and we have considered a RISC system as a solution to fill this gap. We have implemented a language C compiler as a developmental language for all software in this system, and to support this implementation we have created a development and evaluation environment with a simulater as its nucleus. This environment allows the monitoring of register interlock and delay slot that are characteristics of RISC processors and greatly affect the performance of them.

1．はじめに

現在，計算機システムは様々な場面で利用されており，一大学の研究室レベルで使用する場合でも，大量なデータを扱い処理に非常に多くの時間がかかるものもある．このような状況において，システムの高速化が求められており，それに対しては並列処理や，Super Scaler，VLIW などのプロセッサ技術があるが，単一プロセッサの高速化として RISC プロセッサ［1］は今後も性能向上が期待できる．

我々の研究室では，現在，手書き文字認識，ペン指向ウィンドウシステムなど，様々な研究が行われているが，それらのアプリケーションからシステムの高速化に対する要求が高まっている．そこで我々は，処理を行うプロセッサ自体の高速化が第一に必要であるとして，RISC プロセッサを使用することを考えた．それには，まず研究に適した RISC システムの実現が必要である．そのために，我々の研究室で使われている研究用 OS（Operating System）である OS/omicron［2］の RISC バージョンやその開発言語である言語 C 処理系 CAT［3］，開発・評価系などを考え，RISC システム実現のための環境を提供した．

本稿では，RISC 用言語 C 処理系とシステムソフトウェアの開発・評価環境について述べる．

2．RISC 用 OS/omicron の特徴

RISC 用 OS/omicron は，研究の基盤となるシステムであるという点からも，研究への支援が必要である．RISC 用 OS/omicron は次のような特徴を持つ．

(1) 高速な処理

RISC 用 OS/omicron は高速な処理環境を研究アプリケーションに対し提供する．これには，ただ単に RISC プロセッサの高速性を利用するだけでなく，レジスタのセーブ，リストアが問題となるコンテキストスイッチを軽くする機構［4］の採用などを考えている．

(2) OS を動的に拡張できる機能

研究用アプリケーションごとに必要な機能が RISC 用 OS/omicron には求められる．そのため，基本的な機能だけで OS を構成し，対象アプリケーションごとに OS の機能を動的に拡張できるような機構を提供する．RISC 用 OS/omicron ではその機構として，セグメンテーションを利用し，ダイナミックリンク［5］を実現する．

また，RISC という観点以外にも，従来の OS/omicron の特徴を受け継ぎ，リエンタラブル，リロケータブルなオブジェクトコードやフル 2 バイトのコード体系などの特徴を RISC 用 OS/omicron は持つ．

この RISC 用 OS/omicron の特徴の中で，高速な処理環境を提供することはもちろん重要ではあるが，まずは研究をシステムレベルで支援することを考えて，その環境において高速化を考えることとした．

3．ターゲットプロセッサ

ここまで述べてきた通り RISC 用 OS/omicron はあくまでも研究，開発の基盤となる OS である．そのために，ターゲットプロセッサは RISC プロセッサの中でも研究，開発を支援する機能に対応しているものを選択すべきである．そこで，RISC 用 OS/omicron の特徴の中で OS を動的に拡張できる機能の点から，セグメンテーションをプロセッサレベルでサポートしている PA-RISC［6］を採用することとした．

4．RISC 用システムソフトウェア開発・評価環境

ここでは，RISC 用 OS/omicron を含めた RISC システムを開発・評価していく環境について述べる．

4．1 RISC 用システムソフトウェア開発・評価の問題点

RISC 用システムソフトウェア開発・評価の問

題点を，特に RISC 用という観点から考えた．問題点を次に示す．

(1) ディレイドスロット，レジスタインターロックの評価

RISC プロセッサで性能を上げるには，その特徴であるディレイドスロットを有効に使用し，レジスタインターロックを回避する必要があり，それはコンパイラの命令スケジューリングや変数のレジスタ割当ての最適化で解決される．それらの最適化を有効に行うためには，ディレイドスロット，レジスタインターロックの情報の解析を行い，コンパイラにフィードバックする必要がある．そのために，ディレイドスロット，レジスタインターロックの動的なデータ測定などを行うには，それがハードウェアと密接に結びついたものであるので，特殊な手法が必要である．

(2) パイプライン，キャッシュなどソフトウェアレベルで監視困難なものへの対応

パイプライン遷移に狂いが生じたり，キャッシュのミスヒットが発生すると，それは性能の劣化につながる．パイプラインの問題はコンパイラにおける命令スケジューリングや分岐処理の最適化などに関係し，キャッシュはそのサイズが OS におけるメモリ管理や，コンパイラでのメモリ割付けに関係する．その問題を解決するために，パイプラインやキャッシュを監視することは，それがプロセッサ内のハードウェアに密接した部分であるので非常に困難である．

(3) 実行環境の設計

実行環境は，OS の設計やコンパイラの作成時に考慮をしていく必要がある．RISC ではレジスタ数が多いのでコンテキストスイッチが重くなってしまったり，割込み処理のオーバーヘッドに対しても考慮を行わなければならず，特に実行環境の設計が厳密に性能に影響し，非常に重要となっている．また，実際に実現したシステムに対し，実行環境の妥当性を評価できる機構が必要である．

(4) 割込みなど非同期動作の評価

OS で処理される割込みは，他のプログラムの動作中に発生するので，そのプログラムの環境の保存，復帰などが必要とされる．コンテキストスイッチと同様に RISC においては，環境の保存，復帰時には，用意されている多くのレジスタの保存，復帰を伴い，これは OS における割込み処理の設計やレジスタの使用法に性能が影響される．このような割込みは，処理が非同期で行われるという性格上再現が難しいので，評価などを行うのが困難である．

(5) 高特権レベルで管理している情報の取得，保護

OS などのシステムソフトウェアなどをデバッグする場合，関係する情報の監視やその情報の値の書換えを行いたい場面がある．しかし，それらのソフトウェアは特権レベルで動作をしており，関係する情報も特権レベルにおいて管理されているので，情報の保護や取得を行うのに困難な面がある．

4.2 RISC 用システムソフトウェア開発・評価環境の概要

上記の RISC 用システムソフトウェアの開発・評価での問題点を解決し，RISC 用 OS/omicron の特徴をサポートするためには，独自の開発・評価環境の構築がまず必要である．そこで，開発・評価環境を考える上で次のような方針を立てた．

(1) RISC 用システムソフトウェア開発・評価での問題に対処する
(2) RISC 用 OS/omicron の特徴をサポートする

その開発・評価環境の構築にはいくつかの手法があるが，それぞれの手法の問題点解決に対する対応は表 4.1 のようになる．表 4.1 から，シミュレータによる環境の構築法がすべての問題点を解決しており，最良の方法と言える．仮想マシンなどには速度的なメリットがあるが，ここでは

表 4.1 各手法と要求との対応

	実機上のデバッガ	仮想マシン	シミュレータ
ディレイドスロットのチェック	×	×	○
レジスタインターロックの調査	×	×	○
パイプラインのチェック	×	×	○
割込みの監視	×	○	○
非同期処理の再現	×	△	○
高特権レベルの情報の監視，取得	×	○	○
トレース機能	○	○	○
ブレークポイント機能	○	○	○

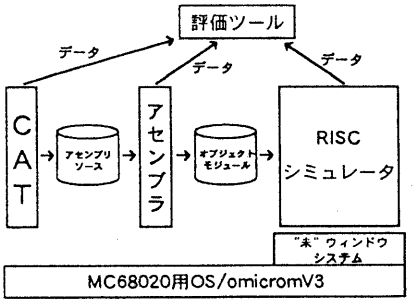

図 4.1 RISC 用システムソフトウェア開発・評価環境

4.1 で述べた問題点を解決することを重要と考え，また従来の OS/omicron の資源を利用できるシミュレータを中心とするクロスの開発・評価環境を採用した．図 4.1 に設計した開発・評価環境を示す．

我々は RISC 用 OS/omicron を実現するためには，まずその開発言語である RISC 用言語 C 処理系 CAT (C Compiler at Tokyo University of Agriculture and Technology) が必要であると考えた．そのために，RISC 用システムソフトウェア開発・評価環境も，中心としては CAT の実現をサポートする部分の構築である点に考慮し，ターゲットプロセッサである PA-RISC のシミュレータから実現を行った．実際には，まず RISC 用 CAT の設計を行い，そこから得られた要求に対し，PA-RISC シミュレータの実現を行って，その環境を用い，CAT の実現を行った．

RISC 用 CAT の実現時にはコンパイラと RISC アーキテクチャとの関係に着目し，また RISC 用システムソフトウェアの開発・評価というソフトウェア工学的視点からも考慮を行った．

5. RISC 用言語 C 処理系 CAT

5.1 RISC 用 CAT の設計

RISC 用 OS/omicron の特徴を考慮し RISC 用 CAT を設計，実現した．CAT 設計時には次の方針を最も重要に考えた．

・実行環境の設計を第一に考える

これは，実行環境が今後，すべてのソフトウェアの性能に非常に影響し，最適化戦略にも関わってくるからである．設計した実行環境の全体構成を図 5.1 に示す．

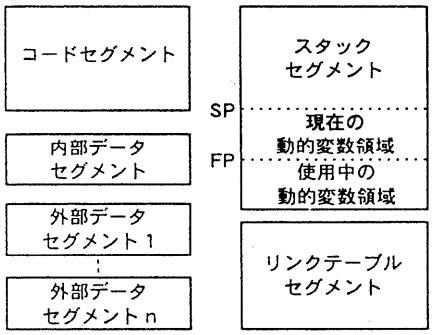

図 5.1 実行環境の全体構成

実行環境はリエンタラビリティを保証するためにセグメンテーションを利用し，プログラムの手続き部とデータ部を分離して管理することとした．コードセグメントにはプログラムの手続き部が置かれ，外部データセグメント，内部データセグメントにはそれぞれ外部変数，内部静的変数が格納される．スタックセグメントには動的変数がスタック構造を用いて格納される．リンクテーブルセグメントには外部関数，外部変数とのリンクのために必要な情報を持つリンクテーブルが格納される．セグメント単位でプログラムを管理することにより，リロケータビリティが得られる．

5.2 RISC 用 CAT の実現

RISC 用 CAT の実現は，信頼性や従来の OS/omicron 資源の利用という点から，現在の

OS/omicron 上でクロス開発を行い，また MC68000 ファミリ用 CAT 第 3 版のソースを再利用して行った．具体的には，プロセッサ非依存である CAT パーザは従来のものを利用し，プロセッサに依存するコードジェネレータの実現を行った．また，コードジェネレータの中でも，再利用できる部分は，可能な限り再利用を行った．実際，従来の OS/omicron の資源を利用できたことは，非常に RISC 用 CAT 実現には有用であった．例えば，CAT 第 3 版のソースを最利用したことにより，プロセッサ非依存部のエラーに悩まされることはなく，また関数コールにおける引数サイズの整合性をチェックするツールを用いたことにより，発生したバスエラーは配列の操作間違いの 1 回だけであった．

RISC 用 CAT コードジェネレータのソース行数に関するデータと CAT 第 3 版からのソースの再利用状況を表 5.1 ，表 5.2 に示す．なお，開発期間は約 8 ヶ月である．CAT 第 3 版はフェーズ分割を行いクロスコンパイラ開発を容易にするという方針を持っていたが，表 5.1 に示す通り今回のコードジェネレータの実現において，全ソースの約 3 分の 1 が再利用したものであることにより，その点が実証された．具体的には表 5.2 に示すように，ファイル入出力や解析木作成部などを再利用した．

表 5.1 RISC 用 CAT コードジェネレータのソース行数

種類	行数
全ソース行数	22721
再利用ソース行数	7624
新規作成ソース行数	15097
再利用割合い	33.55 %

再利用割合い = 再利用ソース行数 / 全ソース行数 × 100

表 5.2 ソースの再利用状況

再利用した部分	新規作成した部分
ファイル入出力 変数，関数管理 文，式解析 中間コード解析 解析木作成	解析木レジスタ数ラベル付け レジスタ管理 コード生成

RISC プロセッサで重要となる最適化は，従来の最適化をダイナミックリンク，セグメンテーションという独特な環境にそのまま適用することができないという考えと，コンパイラの信頼性がまず重要である点から，初版としてはその点に深く関わらない局所的な覗き穴式最適化を中心に行った．

6．PA-RISC シミュレータ

4.2 で述べたように，RISC 用システムソフトウェアの開発・評価環境は，まず RISC 用 CAT の開発・評価をサポートすることを中心として考慮し，PA-RISC シミュレータの実現を行った．

6.1 PA-RISC シミュレータの目的

上でも述べたように，PA-RISC シミュレータは第一段階として，まず RISC 用 CAT の開発・評価を対象とする．PA-RISC シミュレータの目的を次に示す．

(1) RISC 用 CAT が出力したコードの確認
(2) コードの性能評価のためのデータ収集
(3) 最適化開発・評価のためのデータ収集
(4) RISC 用 OS/omicron の開発・評価

6.2 PA-RISC シミュレータの設計

上の目的を受け，PA-RISC シミュレータは次のような機能を持つ．

(1) 性能測定機能

RISC 用 CAT が生成したコードを評価し，そこから最適化を行っていくためにも，まず性能測定が重要と考え，コンパイラにおける最適化に関する次の項目を測定する機能をシミュレータに持たせた．

・レジスタインターロック

命令スケジューリング，変数のレジスタ割当てに影響する．

・ディレイドスロット
命令スケジューリングの最適化を行う場合に対処する必要がある．

・ヌリファイ
PA-RISC 特有の命令の実行をキャンセルできる機能で，これも命令スケジューリングの最適化に利用できる．

具体的には，実行したコードに対し評価を行うために，上記のデータについて実行回数を測定し，また比較評価のために実行時間を測定できる機能を考えた．さらに，最適化へのフィードバックのために，レジスタインターロック，ディレイドスロット非有効使用，ヌリファイのアドレスを実行ログファイルとして出力する機能を持つ．

次の二つの項目は実行制御のために必要である．

(2) トレース・ステップ実行
(3) ブレークポイント設定，解除，有効・無効化

コードや OS の評価などで様々な場面を想定して，データの値を設定し，また，その結果を確認する場合などに，次の二つの機能が使われる．

(4) レジスタ値設定，表示
(5) メモリ値設定，ダンプ，逆アセンブル

6．3　PA-RISC シミュレータの実現

PA-RISC シミュレータは OMICRON V3 未（HITSUJI）ウィンドウシステムを用いて実現を行った．ソースの行数は言語 C で約 9000 行，開発期間は約 3.5 ヶ月である．

PA-RISC シミュレータの画面イメージを図 6.1 に示す．図 6.1 のように PA-RISC シミュレータはすべてのレジスタ，ブレークポイントがビジュアルで表示される．

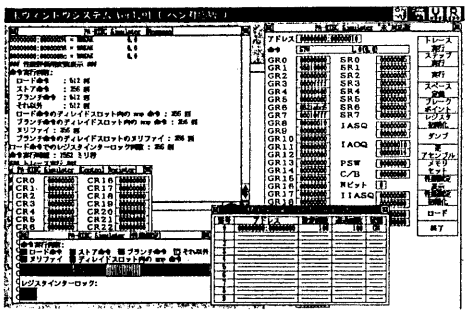

図 6.1　シミュレータの画面

6．4　性能測定機能

6.2 で述べた命令実行回数などのデータに関しては，図 6.2 のような結果が得られるようになっており，命令の実行回数やレジスタインターロックなどの発生回数をグラフ化して見られるようになっている．また，コードの実行ログに関しては，図 6.3 のような結果が得られ，このログデータを統計を取るツールなどで利用することによって，命令スケジューリングなどの最適化の実現，評価にフィードバックすることができる．

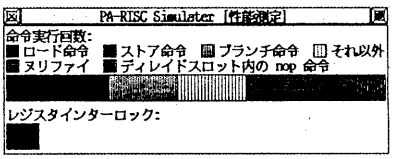

図 6.2　性能測定結果の表示画面

種類	対象アドレス	直前アドレス
ディレイド非有効	00000004:0000002c	00000004:00000028
インターロック	00000004:0000006c	00000004:00000068
ヌリファイ	00000004:0000007c	00000004:00000078
インターロック	00000004:00000088	00000004:00000084
インターロック	00000004:00000090	00000004:0000008c

図 6.3　実行ログファイル

7．RISC 用システムソフトウェアの開発・評価

7．1　最適化実現のための評価

RISC 用 CAT において行われる最適化，ここでは特にローカルな変数のレジスタ割当ての最適化実現のために，現在の RISC 用 CAT が生成したコードから評価を行う．RISC 用 CAT がローカルな最適化に対して持っている戦略を次に示す．

・変数の値がメモリからレジスタにロードされたならば，以後はそのレジスタに変数が割り当てられたとして扱う．その後，関数コールが行われたときに，レジスタに入っている変数の値をメモリに書き戻す

今回はこの変数のレジスタ割当て戦略の妥当性を確かめる．測定には比較的小規模なプログラムを例と考えて，クイックソートと選択ソートの二つに対して行った．その二つのプログラムと上記の戦略に従いハンドコンパイルしたコードとの比較を行った．その測定結果を表 7.1 に示す．

表 7.1　クイックソートと選択ソートの測定結果

種　類		クイックソート		選択ソート	
		①	②	①	②
命令実行回数	ロード命令	1288	702	2326	1015
	ストア命令	607	476	864	635
	ブランチ命令	269	257	499	499
	それ以外の命令	2766	1554	5817	2637
	ロードディレイドでの nop 命令	0	0	0	0
	ブランチディレイドの nop 命令	216	27	368	1
	ヌリファイ	108	0	288	0
	ブランチディレイドのヌリファイ	53	0	131	0
	合計	5307	3016	10293	4787
ロードでのレジスタインターロック		554	0	1264	0
仮想命令実行時間		186.80	105.52	346.03	160.02

①：PA-RISC 用 CAT でコンパイルしたコード
②：ハンドコンパイルによるコード

表 7.1 の結果から，PA-RISC 用 CAT でコンパイルされたプログラムに対し，ハンドコンパイル版はロード命令がクイックソート，選択ソートでそれぞれ約 55% ，約 44% ，ストア命令は約 78% ，約 73% に減少している．また，それ以外の通常命令が約 56% ，約 45% になっているが，これはロード，ストア命令と付随して減少した点が大きい．ハンドコンパイル版において，ロード命令に比べ，ストア命令の減少率が低いのは，ローカルな変数のレジスタ割当てを行っても，関数コール時にその値を保存するストア命令は必要であるので，それ程避けられないからである．

まとめると，ストア命令の減少はそれ程著しくはないが，レジスタインターロックやディレイドスロットに関係するロード命令が半分近く減少する効果は大きいと言える．また，このレジスタ割当ての戦略は比較的簡単であり，最適化を行うコストに比べても効果は十分に得られると考える．

7.2　実行環境の評価

ここでは，ダイナミックリンクを用いた実行環境についての評価・考察を行う．今回の評価は言語処理系に関する点であるので，ダイナミックリンクに対応した実行環境における関数コールのオーバーヘッドを対象とし，リンクが確定している段階での純粋なコーリングシーケンスの測定を行う．

今回は，関数コールに比べて関数本体の処理がそれほど大きくなく，しかも関数コールが頻繁に起きるクイックソートプログラムを対象とした．その測定に使用するプログラムは，再帰呼出しが行われるクイックソートの関数を外部関数として定義するものと，その関数を内部関数として定義するものの二つを用意する．表 7.2 にシミュレータで実行し，測定した結果を示す．

表 7.2　内部関数コールと外部関数コールの測定結果

種　類		内部関数コール	外部関数コール
命令実行回数	ロード命令	1313	1463
	ストア命令	632	682
	ブランチ命令	269	269
	それ以外の命令	2841	3016
	ロードディレイドでの nop 命令	0	0
	ブランチディレイドの nop 命令	216	191
	ヌリファイ	108	108
	ブランチディレイドのヌリファイ	53	53
	合計	5432	5782
ロードでのレジスタインターロック		579	579
仮想命令実行時間		191.39	206.39

表 7.2 の結果から，外部関数コールでは，命令の実行回数でロード命令が 150 回，ストア命令が 50 回増加し，ブランチ命令は変化がないことがわかる．また，その 3 種類以外の演算命令は 175 回実行回数が増加している．全体の命令の割合いを考えても，ダイナミックリンク環境における外部関数コールは，その環境の情報を保存，復帰するロード，ストア命令の回数が多いことが分かる．また，外部関数コールになって，ヌリファイ，レジスタインターロックの回数に変化がなく，ディレイドスロット内の nop 命令の個数は減少しているので，外部関数コールのシーケンス

は命令のスケジューリングが最適に行われていることが確認できる．外部関数コールが内部関数コールより増加している実行時間は 15 μs（CPU クロックを 35MHz として）で，外部関数の場合の実行時間全体に対し 7.28% の増加である．この例の場合，クイックソートの外部関数コールは結局 25 回行われており，その 1 回のコールで内部関数コールに比べて 0.6 μs 増加している．

この結果から通常のプログラム例において，内部関数コールに比べ，10% 未満のオーバーヘッドでダイナミックリンク環境が得られるのであれば，ダイナミックリンクのメリットを考えると，十分有用であると考えられる．

8．おわりに

本稿では，RISC 用言語 C 処理系とシステムソフトウェア開発・評価環境について述べた．今回は，RISC システム全体の開発言語である言語 C 処理系と，その開発・評価のための PA-RISC シミュレータを実現したことにより，RISC 用 OS/omicron の開発・評価の第一段階を行うことができた．今後は，この開発・評価環境を用い，OS/omicron を含めた RISC 用システムソフトウェアや，言語 C 処理系における最適化を開発・評価していく．

参考文献

[1] Patterson, D.A., and Riepho, R.S. : “ RISC Assessment : A High-Level Language Experiment”, Proc. 9th Int'l Symp. on Computer Arch., ACM (1982).

[2] 高橋：“研究プロジェクト総説：OS/omicron の開発”，情報処理学会オペレーティング・システム研究会 39-5 (1988).

[3] 並木，他：“言語 C コンパイラ cat の方式設計”，情報処理学会ソフトウェア工学研究会資料 48-2, pp.1-8 (1986).

[4] 中原，他：“複数の浮動小数点方式を処理する OS/omicron と言語 C 処理系 CAT/N の評価”，情報処理学会オペレーティング・システム研究会 46-1 (1990).

[5] Deley, R.C., Dennis, J.B. : “Virtual Memory, Processes, and Sharing in MULTICS”, Comm. ACM, Vol.11, No.5, pp.306-312 (1968).

[6] D. S. Coutant, C. L. hammond, and J. W. Kelly : “Compilers for the New Genelation of Hewlett-Packard Computers”, Proceedings of COMPCON S'86, pp.48-61 (1986).

[7] 早川，他：“手書きインターフェースを支援する OS OS/omicron 第 4 版の構成”，情報処理学会コンピュータシステムシンポジウム論文集 Vol.92, No.7, pp.35-42 (1992).

[8] 田中，他：“RISC 用 OS/omicron における言語 C 処理系の実行環境の設計”，情報処理学会第 44 回全国大会講演論文集, 2F-11 (1992).

[9] 田中，他：“RISC 用 OS/omicron における言語 C 処理系の開発・評価環境の構築”，情報処理学会第 46 回全国大会講演論文集, 1J-06 (1993).