COINS コンパイラ・インフラストラクチャの開発

中田 育男　渡邊 坦　佐々 政孝　森 公一郎　阿部 正佳

COINS コンパイラ・インフラストラクチャは、コンパイラの研究・開発・教育を容易にする目的で開発したものである。COINS は (1) 高水準中間表現と低水準中間表現の 2 水準の中間表現をもつ、(2) 記述言語は Java で、すべて新規開発した、(3) SSA 最適化など最適化の機能が充実している、(4) 対象機種変更容易なコード生成系をもつ、(5) 並列化の機能を持つ、といった特徴をもっている。開発作業は 10 箇所以上で分散して行い、3 週間に 1 回程度の全体打ち合わせを持ち進めた。途中いくつかの失敗もあったが、ほぼ当初の目標を達成できた。入力言語は C と FORTRAN として、対象機種は SPARC, x86 など、全部で 8 機種のコンパイラが出来ている。C コンパイラの目的コードの性能は、GCC のそれに匹敵するものが得られている。COINS システムは Java で約 26 万行の大きさである。本論文では、このインフラストラクチャについて、技術面と開発作業の観点から述べる。

COINS has been developed as a compiler infrastructure which is used as a base for constructing compilers for various purposes such as research, education, production, and so forth. The salient features of COINS are: (1) two levels of intermediate representation, HIR: High-level Intermediate Representation, and LIR: Low-level Intermediate Representation, (2) entirely written in Java anew, (3) sufficient support for optimization, including an almost full set of SSA optimization, (4) retargetable code generators, (5) basic support for parallelization. It has been developed at more than ten distributed organizations, having developers' meetings once three weeks. The complilers for two languages, C and FORTRAN, and for eight target machines, including SPARC and x86, have been developed in COINS. The performance of their object codes are comparable with those of GCC. COINS is written in about 260KLOC in Java. This paper describes the development aspect as well as technical aspect of this project.

1　はじめに

コンパイラの研究・開発・教育、およびコンパイラ技術の蓄積を容易にすることを目的として、COINS コンパイラ・インフラストラクチャを開発した。コンパイラはコンピュータの核技術の 1 つであり、アーキテクチャとは車の両輪のようなものであるから、その技術がないと新しいアーキテクチャ開発なども制約される。良いコンパイラ（良質なコードを作り出し、信頼性も、使い勝手もよいコンパイラ）を作るには技術の蓄積が必要で少人数ではむずかしい。また、実用的なレベルのコンパイラに関する教育は容易ではない。

これらの問題を解決する 1 つの方法がコンパイラ・インフラストラクチャの利用である。各種の言語、各種のアーキテクチャに対応できる実用性の高いコンパイラの共通インフラストラクチャがあれば、アーキテクチャ開発と同期した早期のコンパイラ開発もコンパイラ技術の蓄積も可能になり、最適化などの新しい技術を開発した場合にそれを組み込んだコンパイラでの実験も容易になり、また、従来のようなトイ（おもちゃ）コンパイラによる教育でなく、実用的なレベル

Developent of COINS compiler infrastructure.

Ikuo Nakata, 法政大学情報科学研究科, Graduate School of Computer and Information Sciences, Hosei University.

Tan Watanabe, COINS コンパイラ･インフラストラクチャ協会, COINS Compiler Infrastructure Association.

Masataka Sassa, 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.

Koichiro Mori, エル･エス･アイ ジャパン (株), LSI Japan Co., Ltd.

Seika Abe, フリーランスプログラマ

コンピュータソフトウェア, Vol.16, No.5 (1999), pp.78–83.

[ソフトウェア論文] 1999 年 8 月 3 日受付.

での教育も可能になる。COINS は、これらを目的として開発した。

本論文では、COINS の設計方針、開発経過、開発したシステムの概要、コンパイラ・インフラストラクチャとしての特徴、性能評価、などを述べる。

2 設計方針

COINS コンパイラ・インフラストラクチャは、各種のコンパイラの研究、開発、教育などに使えるものにすることを目的として開発した。そのための設計方針としては以下を掲げた。

1. 高水準中間表現と低水準中間表現の 2 水準の中間表現を設ける
2. システムの記述言語を Java とする
3. 配布や利用に制約を受けないようにするために、全体を新規開発する
4. 最適化機能を充実させる
5. 対象機種変更容易なコード生成系とする
6. 並列マシンのための並列化の機能を持たせる
7. SIMD 並列化を実現する
8. 実用性を重視する

以下これらの項目について説明する。

2.1 2 水準の中間表現

インフラストラクチャとしては、コンパイラの内部で使われる共通な中間表現を定め、その中間表現を中心としてモジュール群を配置するのが通常考えられる方法である。COINS でも、それにならい、中間表現として、ソースプログラムに近い高水準中間表現（HIR: High-level Intermediate Representation）と、機械語に近い低水準中間表現（LIR: Low-level Intermediate Representation）の 2 水準の中間表現を設けることとした。

2 水準の中間表現を設ける理由は次のとおりである。低水準中間表現は各種の機械語の目的コードを生成するのに適している。高水準中間表現の存在は各種言語のコンパイラのフロントエンドの作成を容易にする。また、各種最適化の中には、並列化のように高水準中間表現を対象とするのが適しているものと、命令スケジュールのように低水準中間表現を対象とするのが適しているものがある。

通常のコンパイル過程では、ソースプログラムを HIR に変換し、HIR 上で最適化などを行った後に HIR から LIR に変換し、LIR 上で最適化などを行った後に機械語の生成を行う。HIR の水準は C 言語に近いので、この通常の過程の他に、HIR から C プログラムなどに変換して出力することも可能になる。

2.2 Java によるシステム記述

COINS の記述言語には Java を選んだ。Java の場合、メモリ管理に気を使わずにすむこと、実行時のエラーチェックの機能などが、信頼性の向上につながると考えたのがその理由である。そのような機能のない C や C++では大きなシステムをいくつかの組織で分散して開発する場合に信頼性を保持するのが難しいと判断した。Java がプラットフォーム独立であることもそれを選んだ l つの理由である。Java で懸念されることは実行速度が C や C++に比べて遅いということであるが、JIT などのコンパイラ技術の進歩とハードウェアの性能向上とにより、あまり問題にならないと判断した。実際にできあがった結果では、GCC にくらべ、7 倍から 20 倍程度のコンパイル時間がかかるが、2.8GHz の Pentium においてたとえば 1500 行の C ソースプログラムのコンパイル時間は 5 秒から 10 秒であり、この程度の効率であれば、上記の利点を考慮すれば十分であると判断した。

2.3 最適化機能の充実

実用的なコンパイラにとっては、効率の良いコードを生成することが重要であるから、コンパイラ・インフラストラクチャとしては最適化の機能を充実させることが必要である。COINS には 2 水準の中間表現があるので、それぞれについて最適化を考えることが出来る。

低水準中間表現 LIR に対する最適化の機能に関しては、SSA 形式（静的単一代入形式）に基づく最適化 [2] [3] （以下，SSA 最適化という）を充実させることにした。SSA 形式は、変数の定義と使用の関係が明確で、いろいろな最適化が見通しよく容易に行えることと、SSA 最適化について多くの研究と実用化が

されていたが、インフラとして充実した機能を備えているものがなかったこと、がその理由である。そのようなインフラとすることによってさらに SSA 最適化自身の研究が進められることを目指した。この SSA 最適化は 3 番地コードのレベルまで分解されたものに適用するのがより効果が出ると考えて LIR に対して実現することとした。

一方、プログラムの構造を変更するような最適化は、ループなどの構造を陽に表現している高水準中間表現 HIR に対して行う方が適している。そこで，HIR においてループ並列化、インライン展開、ループ展開などの最適化を行うこととした。しかし、HIR は一般に機械語レベルの最適化には向かない。たとえば、配列要素 `a[i]` は配列 `a` の `i` 番目の要素であるという形をしている HIR の方が並列化の可否の判定などがしやすいのに対し、`a[i]` が

```
&a[0] + i*8
```

のようなアドレス計算の式に分解されている LIR の方が、機械語レベルでの最適化はしやすい。以上が 2 水準の中間表現を設けた理由の 1 つでもあった。

HIR と LIR の両方に対して、最適化のモジュールを用意すると、両者の機能で重複する部分も生じて一見無駄なようではあるが、いろいろな使われ方をされるインフラストラクチャとしては、必要であると考えた。実際に、HIR を対象としてループ並列化などを行い、LIR を経ずに、その結果の HIR を OpenMP プログラムとして出力する場合には、HIR での最適化が必要である。

一般に最適化には、どのような最適化をどのような順序で適用するのが最適であるかが分からないという問題がある。インフラストラクチャとしては、任意の最適化の適用を任意の順序で指定することができるようにして、最適化の効果を実験し、評価できるようにした。

2.4 対象機種変更容易なコード生成系

インフラストラクチャとしては、コード生成系は、各種対象機種に対応できるように、対象機種変更容易（リターゲッタブル）なコード生成系とするのは当然とも言えるが、当初は 1 つのコンパイラを完成することに追われて、それを十分検討する余裕がなかったので、とりあえず SPARC 用のコード生成系を開発することにし、対象機種変更容易な生成系は後半の重点目標として開発した。これについては 5.2.1 節で後述する。

2.5 並列マシンのための並列化の機能

そもそも COINS プロジェクトの名称は「並列化コンパイラ向け共通インフラストラクチャの研究」（後述の「謝辞」参照）であり、並列化の機能は持たせることにしていたのであるが、並列化コンパイラには多くの技術の積み重ねが必要であり、5 年間のプロジェクトではそれを充実させるのに必要な期間と要員が不足していると判断し、並列化コンパイラに関しては基本的な機能の実現とそれを使った試作にとどめることにした。

基本的な機能としては、ループの do-all 型の並列化と、SMP (Symmetric Multi-Processor) 用の並列化の機能として粗粒度並列化の基本的な機能を持たせることにした。

2.6 SIMD 並列化

最近の多くの機種はマルチメディア用のプログラムを高速に実行するための特殊命令を備えているが、それらは一般に SIMD (Single-Instruction Multiple-Data) 命令と呼ばれる。SIMD 命令には、たとえば、64 ビットのレジスタを 8 ビット × 8 のベクトルレジスタのように使って並列に実行してしまう命令などがある。このような命令をコンパイラがその目的コードの中に生成するのは大変難しく、組込み関数のような形でそれらの命令が使えるようにしているコンパイラはあっても、最適化の一環としてそれを生成しているコンパイラはなかったので、それの実現を目標とした。

2.7 実用性の重視

インフラストラクチャは人に使ってもらえなければ意味がない。使ってもらうためには、機能を充実させ、信頼性を高くするだけでなく、文書化が必要である。

コンパイラのインフラストラクチャとしては、各種の入力言語と対象機種のコンパイラを備える必要がある。C 言語以外の入力言語として FORTRAN、機種としては SPARC の次に x86、その次に ARM [4] をとりあげて、そのコンパイラを作成することにした。また、入力言語として Java 言語についても、検討することにした。

信頼性が高いことを示すためには、少なくとも、広くベンチマークとして使われている SPEC ベンチマーク [5] が正しくコンパイルできて、その実行性能も満足できるものにする必要がある。

文書化は、開発と同時に行うことが望ましいが、開発期間が限られていたので、後回しになった部分もあった。このような部分についてはプロジェクト終了後に早急に文書化を行った。なお、海外でも使ってもらうためには英語のドキュメントを用意する必要があるので、英語のドキュメントやコメントも書きながら開発するのが望ましいが、英語が得意でない人にそれを強制すると開発自身が進まなくなる心配もある。そこで、日本語で書かれたものに対しては、後で英訳を発注するなどの方法を採ることにした。

3 開発経過

3.1 開発体制

開発は表 1 のメンバで、まず 5 年間のプロジェクトとして行った。全体を統括する強力なマネージメントがあるわけではなく、各人、各事業所では独自の方法で開発を進めた。開発環境としては、機種も OS も各人各様であり、ただ一つ共通しているのは Java で開発していることだけであった。プロジェクトとしてのまとまった検討は、ほぼ 3 週間に 1 度集まって数時間行うのが中心で、その他にメーリングリストでのやりとり、年に 1 度の 1 泊 2 日の合宿による集中討議、問題があったときの全員によるデバッグ、などを行った。全員デバッグの例としては、初期のバックエンドがなかなか動かなかったときや、最終段階で SPEC ベンチマークがコンパイルできなかったときに行った。

年に 2 回の研究運営委員会と、年に 1 回程度の外部者からなる研究 WG からのアドバイスも参考にしながら開発した。年額 1 億円の予算では、年額 1 人 2 千万円のソフトウェア会社に発注すれば 5 人分しかないことから、このプロジェクトの大きな課題は、いかに安い費用で良いソフトウェアを開発するかであり、そのために、教員自身による設計やコーディング（これは無料）、優秀な学生のアルバイト、卒業したばかりの学生が経営するソフトウェア会社やコンパイラの技術者のいるソフトウェア会社への発注、などを行った。

表 1　担当者（*印）と協力者（所属はいずれも開発時）

氏名	所属	担当部分
中田育男*	法政大学	代表, Fortran, 文書化
藤瀬哲朗*	三菱総研	幹事, 基本最適化・並列化
渡辺　坦*	電通大	基盤部まとめ, HIR 他
佐々政孝*	東工大	SSA 最適化
弓場敏嗣*	電通大	SMP 並列化
藤波順久*	ソニー	SIMD 並列化
鈴木　貢*	電通大	SIMD 並列化
千葉　滋	東工大	C 構文解析
岩澤京子	拓殖大	ループ並列化
阿部正佳	東大大学院	LIR 仕様, マシン記述仕様
森公一郎	LSI Japan	バックエンド ver. 2
山野紘一	ブナソフト	バックエンド ver. 1
福岡岳穂	管理工学研	SIMD 並列化, SSA 最適化
山本普一郎	愛知県立大	ソフトウェア工学的利用
滝本宗宏	東京理科大	最適化への COINS 利用

C コンパイラのテストプログラムとしては、約 700 個のテストプログラムを作成したが、それだけでは到底不十分である。幸い、SRA 社で開発された約 8000 個の網羅的テストプログラム集[†1]を同社の引地信之氏の好意で COINS 開発用に借用することが出来て、個々の機能のデバッグがほぼ完了した。しかし、それでも SPEC ベンチマークがすぐに正しくコンパイルできたわけではなく、プロジェクトの最終段階でようやくそれが達成された。

COINS のモジュールをバージョンアップする際は

†1　このテストプログラム集は現在は公開されている [6]。

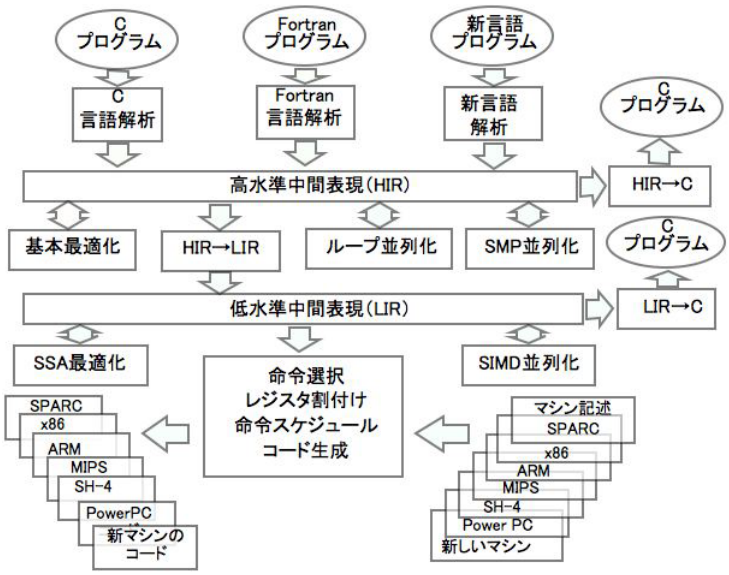

図 1 COINS の構成

上記の約 700 個のテストプログラムでデグレードがないことを確かめることを原則としている。

3.2 開発規模

COINS の構成の概要を図 1 に示す。前章の「設計方針」にしたがって開発した結果、このような構成になった。

最初の 3 年間でプロトタイプを開発したときは Java で約 16 万行であったが、COINS プロジェクト終了時（2005 年 3 月）には Java で約 23 万行で、そのうちマシン記述は 6 機種（SPARC, x86, ARM, MIPS, SH-4, PowerPC）の合計で約 1 万 9 千行であった。その後のプロジェクトや保守で開発したものを合わせて、2006 年 10 月時点で、全部で約 26 万行で、そのうちマシン記述は 8 機種（追加されたものは MicroBlaze と Thumb）の合計で約 2 万 1 千行である。

4 中間表現

ここでは、COINS インフラストラクチャの中核をなしている 2 つの中間表現について、その設計方針、概要、特徴などを述べる。

4.1 高水準中間表現 HIR

4.1.1 HIR の概要

COINS はコンパイラ・インフラストラクチャなので、それを用いて各種の言語に対する様々な方式のコンパイラを作成できなければならない。たとえば、入力プログラムを機械語プログラムに変換するだけでなく、並列化や最適化を行ってその結果を C などのプログラムとして生成するコンパイラも作成できなければならない。これに応えるため、現行の多くの手続き型言語に共通する機能を抽象化して、それらの言語のプログラムの各種の式、文、データ構造および制御構造を表現可能とするように、高水準中間表現 HIR の仕様を定めた。中間表現が複雑であると、それを処理するコンパイラ・モジュールも複雑になるので、その仕様をできる限り単純化するようにした。

結果としては、C のプログラム構造やデータ構造、文、式に対応するものが多くなったが、次のように、種々の点で C とは異なっている。

(a) C の言語仕様によれば添字式付き配列 `a[i]` はポインタによる表現 `(*(a+i))` と等価であるので、配列要素の中間表現としては「ポインタ＋オフセット」のみとすることも考えられるが、HIR では、可能なかぎり添字式つき配列で表現する。その理由は、並列化

や最適化では配列要素の依存解析が重要であり、ポインタを使った表現にするとそれが難しくなるためである。

(b) 配列は要素数だけでなく、添字の下限値も指定可能とし、それらは定数ばかりでなく、式としても表現可能とする。これは FORTRAN など、各種の言語に対応可能とするためである。

(c) C にはソース表現を簡潔にする複合演算子や式の中に代入文を書く機能があるが、これらは基本的な式や文の組合せで表現することにして、HIR を単純化する。

(d) コンピュータの記憶域の番地については、ビット長や、語境界に関する制約、負値を加えることが許されるか否かなどが機種によって異なるので、番地は算術演算用の int, long などとは異なるものとして扱う方がよい。HIR ではポインタはそれが指すデータの型情報を付加した番地 (address) としており、番地が要求されるところではポインタを使う。また、番地の差分は C にはない offset 型データとして表し、番地とそれに対する演算結果をポインタと offset を使って表現する。

int や long, address, offset のビット長などの機種依存情報は MachineParam というクラスに集約し、個別の機種に対してそのサブクラスを作り、文字列定数の処理などの言語依存部分の処理は SourceLanguage というクラスに集約し、個別の言語に対してそのサブクラスを作ることにしている。これによって、HIR の表現自体は機種にも言語にも依存しない形になっている。

4.1.2 記号の表現

コンパイラでは、一般に、変数などの処理対象を表す記号は記号表に記載し、中間表現における処理対象の表現は、記号表の記載項への参照という形をとる。HIR では、変数や型などばかりでなく、定数も記号表に記載している。これは定数の精度がコンパイル時の機種と実行時の機種で異なることがあり得ることから、定数の内部表現を HIR の中に埋め込むと、HIR が機種に依存する可能性が生ずるので、それを避けるためである。

記号の有効範囲を表現する方法には、通常行われる木構造による方法を採用している。すなわち、有効範囲ごとに個別の記号表を作り、有効範囲 A の内側の有効範囲 B で宣言される記号を載せた記号表は A の記号表の子とする木構造によるものである。この木構造の根は大域的に宣言された記号の表であり、大域的に宣言された副プログラムの記号表がその子となる。そうすることにより、プログラムを副プログラムごとにコンパイルする形態だけでなく、副プログラム間にまたがった最適化などを行うことも可能にしている。

4.1.3 仕様の拡張

入力言語を確定させた上でのコンパイラ開発では、その言語の諸項目をすべて表現できる中間表現を決めてから詳細設計に進むことが要求される。しかし、COINS では、多くの言語に対応できることを目指しているので、可能性のある言語の諸項目をすべて表現できることを確かめてから詳細設計を始めるというのは、現実的でない。そこで、特性の異なる数個の言語を視野に入れて基本仕様を決めたあと、言語を追加するたびにどうしても必要な機能があれば追加してゆくという方法をとった。このような機能拡張を容易にするため、コンパイラの各モジュールは中間表現を入力して中間表現を出力する形をとり、モジュール間で個別に情報を受け渡すことをしない。後者のようにすると、機能追加に伴う改造量が増えるためである。

C を実現したあと FORTRAN を実現するときは、C にない COMMON と複素数を表現する必要があった。COMMON については、それを抽象化した機構も検討したが、固定レコード（C の構造体）と同様に扱うことでも実現可能なので、現在はその方法を採用している。複素数については複素数型を導入することも検討したが、実部と虚部が陽に表現されている固定レコード表現の方が最適化がしやすいと考えて、固定レコードに複素数であることを示すフラグをつけて、複素数としての扱いをしやすくする、という拡張にとどめた。FORTRAN のために修正を要した HIR 関連のモジュールは、最適化部、並列化部も含めて 4 個である。

32 ビット機のコンパイラを実現したあと、64 ビット機のコンパイラを作成したが、そのさい、修正したモジュールは 4 個で、それらは初期設計時の検討不

足が原因であったものだけである。

上記の追加・修正量は、COINS 全体のモジュール数 599 に比べるとわずかであり、設計方式には大きい誤りはなかったと言えよう。

オブジェクト指向言語については、Java のクラスファイルを入力として機械語のコードを生成するコンパイラ（Ahead-of-Time コンパイラと呼ばれる）を検討したが、実行時環境の実現に問題があって未完で終わった。Java では実行時環境に多くの機能が要求され、その作成は容易ではないので、gnu の Java コンパイラ（gcj [8]）の実行時ライブラリを解析して [9] それを使うことにし、通常の処理はほぼ実現できたのであるが、例外処理が未完となった。例外処理の実現を難しくした 1 つの理由は、HIR で try-catch の構造を表現できなかったことであった。これは今後の仕様拡張の課題の 1 つである。

4.1.4 具体表現

HIR の物理表現は木構造である。テキストで表すとき、節は

（ オペレータ 型 第 1 子 第 2 子 … 第 n 子 ）

葉は

< 種別 記号 型>

と表す。たとえば

```
for (i=0; i<10; i=i+1) {
  a[i]=i;
  ...
}
```

から作成される HIR は

```
(for void
  (assign int <var i int> <const 0 int>)
  (cmpLT bool <var i int> <const 10 int>)
  (block void
   (assign int    // a[i]=i;
    (subs int <var a <VECT 10 0 int>>
              <var i int>)
     <var i int>)
      ....
  )
  (assign int     // i=i+1;
   <var i int>
   (add int <var i int> <const 1 int>) ) )
```

となる。第 1 子、第 2 子、… はほぼ実行順序に合わせて並べるので、i=i+1 に相当する部分はループ本体の後ろに置かれる。

このような HIR を生成するために、上記の for, assign, var, cmpLT, block, subs, のそれぞれに対応して、その部分木を作成するメソッドが用意されている。ソースプログラムの各構成要素にほぼ 1 対 1 に対応させてそれらのメソッドを呼べば HIR を作成できるので、フロントエンドの作成が比較的簡単である。

実際に、FORTRAN のフロントエンドを開発する際には、最近のプログラミング言語にはない、算術 GOTO など FORTRAN 特有の機能を、どのような HIR の木として表現するかには検討を要したが、それを決めたあと、ソースプログラムを HIR の木に変換するプログラムを書くのは、HIR 作成用メソッドを使うことにより、比較的容易であった。

4.2 低水準中間表現 LIR

低水準中間表現の例としては、GCC の RTL [10] があり、実際にそれを使って多くのコンパイラが作成されている。COINS の低水準中間表現 LIR を設計するときもそれが参考になり、基本的には類似したものになっている。しかし、COINS の LIR は、コンパイラのモジュールの独立性を高めるため、次のような特徴を持っている [11]。

1. 独立したプログラミング言語である
2. 機種独立性が高い

それらについて以下に説明する。

4.2.1 独立したプログラミング言語

通常の中間表現は、コンパイル過程で一時的に使われるものであり、独立したプログラミング言語とはなっていない。たとえば、記号表は中間表現とは別に定義されており、中間表現ではそれを参照するという形式をとっているから、その記号表がなければ情報が不完全なものとなる。

COINS の LIR は独立した完全な 1 つのプログラミング言語として定義されており、その仕様は表示的意味論を使って厳密に定義されている [12]。したがって、LIR を扱うモジュールはその言語仕様に従うだけでよく、それ以外の情報を参照する必要はないので、他のモジュールとは独立に開発することができ、また独立に利用することも出来る。たとえば、ある

言語のフロントエンドを COINS を使わずに開発し、その出力を LIR とすることで、COINS のバックエンドの機能を利用することが出来る。また、LIR をファイルに出力して、外で処理をし、ふたたび読み込んで処理を続けることができるが、GCC の RTL ではそれはできない。

その LIR の形の例をソースプログラムと対応させて以下に示す。ソースプログラムはファイル ex1.c にあり、

```
int x;
int func(int y){
    return x+y;
}
```

とする。それから生成される LIR は以下のようになる。左端の番号と"//"以降は説明のために付けたものである。LIR の物理表現は、木構造であり、命令列は木構造の各命令をリストでつないだ形をしている。テキストで表現するときは木構造の子供は字下げして示す。

```
 1(MODULE "ex1.c"
 2 (SYMTAB  //大域的記号表
 3  ("func" STATIC UNKNOWN 4 ".text" XDEF)
 4  ("x" STATIC I32 4 ".bss" XDEF))
 5 (FUNCTION "func" // 関数 func の定義本体
 6  (SYMTAB  //局所的記号表
 7   ("y.1" FRAME I32 4 0)   // 引数 int y
 8   ("returnvalue.2" FRAME I32 4 0))
 9  (PROLOGUE (0 0)
10   (MEM I32 (FRAME I32 "y.1")))
11  (DEFLABEL "_lab1")  // ラベル定義 _lab1
12  (SET I32
13    (MEM I32 (FRAME I32 "returnvalue.2"))
14    (ADD I32 (MEM I32 (STATIC I32 "x"))
15             (MEM I32 (FRAME I32 "y.1"))))
16  (JUMP (LABEL I32 "_epilogue"))
17  (DEFLABEL "_epilogue")
18  (EPILOGUE (0 0)
19   (MEM I32 (FRAME I32 "returnvalue.2"))))
20 (DATA "x" (SPACE 4))) // int x; の領域
```

LIR プログラム（LIR はプログラミング言語であるので、それで表現されたものは LIR プログラムと呼ぶことが出来る）はモジュール（MODULE）と呼ばれる。モジュールは、大域的記号表（2-4 行）、いくつかの関数の定義（5-19 行）、データ（20 行）などからなる。関数は、局所的記号表（6-8 行）、命令列（9-19 行）からなる。ここで、I32 は 32 ビット整数型を表し、SET は代入で、最初の子供が左辺、次の子供が右辺を表す。また、MEM はメモリにあること、FRAME はスタックの中にとられるフレーム（関数の呼びだしごとに確保され、その関数の局所変数などの場所を含む）にあること、STATIC は静的変数であることを表す。なお、"returnvalue.2"は関数の戻り値を入れるための変数である。関数の命令列は、PROLOGUE 式で始まり、 EPILOGUE 式で終わる。

4.2.2 機種独立性

LIR は機種独立性が高く、バックエンドの全フェーズで共通に使われ、各フェーズは LIR から LIR への変換系である。ただし、最後のアセンブリコードを生成するところだけが、LIR への変換ではない。GCC の RTL は対象機種の命令選択をした後の表現に使われるものであるが、LIR は命令選択をする前の表現にも使われる。LIR の末端オペランドには機種に依存する情報を含むものもあるが、LIR の文法自体は機種に依存しない。たとえば仮想レジスタも物理レジスタも文法的には同じ扱いであり、ロードはメモリ参照、ストアは SET 式というように、機種に依存しない形に抽象化されている。

このことによって、

1. バックエンドの全フェーズを機種独立な形で書くことが出来る
2. バックエンドの全フェーズでライブラリを共用できる

という利点が得られる。コード生成のための機械語の命令選択をした後でも、機械語命令に対応する LIR 形式になっているから、その後のレジスタ割付けなども機種独立な形で書ける。その結果、バックエンド全体が機種独立になる。

LIR で表現された式 (L 式）は任意の複雑度を持つことができ、末端オペランドはメモリ、レジスタ、定数、関数名、ラベルを表す。機種依存性の高い関数の入口、出口処理は PROLOGUE, EPILOGUE として抽象化されている。HIR から変換された直後の L 式では、ソースプログラムの 1 つの代入文が 1 つの SET 式になり、複雑な L 式になる場合もある。コー

ド生成をするためにそのL式に対して機械語の命令を選択（これがコード生成とか命令選択とか呼ばれる）をした後では、複雑な式は、計算の途中結果を示す仮想レジスタを導入して分解され、対象機種の1つあるいは数個の機械語に直接対応した単純なL式の列に変換される。レジスタ割り付けでは仮想レジスタは実レジスタに変換される。これらの変換はいずれもL式からL式への変換なので、そのプログラムは、どの対象機種にも適用できるモジュールとして書ける。

先ほどの例の 12-15 行は、命令選択の直前では次のようになっている。

```
(SET I32 (REG I32 "returnvalue.2%")
         (ADD I32 (MEM I32 (STATIC I32 "x"))
             (REG I32 "y.1%")))
```

ここで、(REG I32 "...") は、レジスタを表し、"..."がレジスタの名前である。それがマシンの実レジスタ名でないときは、仮想レジスタ名である。今の場合は、レジスタ割付け前なので仮想レジスタ名になっている。引数の y は、LIR が作られたときはフレームの変数の形をしていたがここでは"y.1%"という名前の仮想レジスタになっている。これに対して、対象機種を SPARC として、後で述べる方法によって命令選択を行うと

```
(SET I32 (REG I32 ".T2%")
         (STATIC I32 "x"))
(SET I32 (REG I32 ".T1%")
         (MEM I32 (REG I32 ".T2%")))
(SET I32 (REG I32 "returnvalue.2%")
         (ADD I32 (REG I32 ".T1%")
                  (REG I32 "y.1%")))
```

となるが、最初の SET 式は x のアドレスを仮想レジスタ.T2%にセットする命令に対応し、2 番目は.T2%番地の内容を仮想レジスタ.T1%に載せる命令、3 番目は仮想レジスタ.T1%と y.1%の内容を加えたものを仮想レジスタ returnvalue.2%にセットする命令、にそれぞれ対応する。この後でレジスタ割付けを行っても、レジスタ名が仮想レジスタ名から SPARC マシンの実レジスタ名に代わるだけで、LIR の形は保っている。

5 コンパイラ・インフラストラクチャとしての特徴

コンパイラのインフラストラクチャに求められる要件は以下のとおりである。

- 基本的な機能は一通りそろっている
- 新しい言語のフロントエンドが比較的容易に作成出来る
- 新しい機種のバックエンドが比較的容易に作成出来る
- 新しい機能の追加や既存の機能の変更などが容易である
- インフラストラクチャの内容を理解するためのドキュメントやツールがある

COINS では、それらをほぼ実現した。ここでは、その中で COINS の特徴的な以下の項目について述べる。

1. 最適化の機能が充実している
   (a) HIR に対しては並列化モジュールもある
   (b) LIR に対して SSA 最適化のモジュールが充実している
   (c) 各種の最適化の適用の順序が任意に指定できる
   (d) バックエンドにも基本的な最適化モジュールがあり、レジスタ割付けにも工夫がある
2. バックエンドの拡張性が高い
   (a) 対象機種変更容易なコード生成系である
   (b) コード生成系のためのマシン記述が比較的容易である
   (c) 機能拡張が容易である

最適化の一環として SIMD 並列化の機能を持っている [13] [14] のも COINS の特徴であるが、完成度が十分とは言えないので、ここでは述べない。また、ドキュメントやツールの整備にも努力した [15] [16] [17] が、それは当然備えるべきものともいえるので、ここでは述べない。

5.1 最適化機能の充実

5.1.1 HIR での最適化と並列化機能

基本的な機能として、制御フローグラフの構築、データフロー解析などの機能を備え、それを使って、コピー伝播、定数畳み込み、共通部分式削除、部分冗長性除去、無用命令除去、ループ展開、インライン展

開などのモジュールが作られている。また並列化については、十分ではないが、もっとも基本的なものとして do-all 型ループの並列化変換、粗粒度並列化のためのマクロタスクグラフの作成、マクロタスクの実行スケジューリングの機能を備えている。

5.1.2 SSA 最適化

SSA 最適化のためには、通常の中間表現を SSA 形式に変換し、その SSA 形式の上で最適化変換をしてから、バックエンドの他のフェーズに渡すためにもとの中間表現形式に戻すのが一般的な方法である。そのそれぞれについて、研究・提案されているほとんどの機能を備えるようにした。

SSA への変換に関しては、制御フローグラフの合流点に置く ϕ 関数命令の多少によって、minimum、semi-pruned、pruned の 3 つの方法が提案されているが、変換時間と SSA 形式上での最適化にかかる時間とのトレードオフから見てどれが最適であるかという定説はない。そこで、これらをすべて備えた。コンパイラのオプション指定でその中から 1 つを選んで使うことが出来る。

SSA から通常形式の中間表現に戻す変換に関しては、当初考えられた単純な変換[18] では間違った変換をしてしまうことが知られており、Briggs ら[19] と Sreedhar ら[20] による修正アルゴリズムが提案されている。また、後者については、変換結果にコピー命令がなるべく入らないようにする方法もいくつか提案されている。それらについても、比較評価が出来るように、すべてを備えた。

SSA 形式の上での最適化については、一般に提案されているものはほとんどすべて備えた。それらは、演算の強さの軽減と判定の置き換え、共通部分式削除、コピー伝播、条件付定数伝播、ループ不変式移動、無用命令除去である。さらに、本プロジェクトのメンバが新たに開発した、質問伝播[21][22] に基づく要求駆動型部分冗長除去も備えた。これらの最適化モジュールは、すべて SSA 形式から SSA 形式への変換を行う独立したモジュールとなっているので、それらを任意の順序で適用することができる。

また、これらの最適化につながる補助的な機能として、危険辺の除去、無用 ϕ 除去、空ブロック除去、基本ブロックの連結、SSA グラフの作成、式の 3 番地コードへの分割を備えた。詳しくは参考文献[23] を参照されたい。

5.1.3 任意の最適化順序の指定

最適化コンパイラの設計でいつも問題になるのは、多くの最適化機能を備えても、それらの適用の効果は対象とするソースプログラムによって異なるので、すべてのプログラムに対する最適な適用順序はないということである。そこで、HIR 最適化でも SSA 最適化でも、最適化処理の順序を任意に指定できるようにした。同じ最適化を複数回適用することも可能である。一方、標準的な処理順序については、ひとまとめにして指定可能にした。

SSA 最適化については、いろいろな適用順序を試し、その結果、多くのテストプログラムに対して比較的良い結果を出す SSA 最適化の列を「-O2」の最適化として採用した.

5.1.4 バックエンドでの最適化とレジスタ割付け

バックエンドでもっとも重要なものは命令選択とレジスタ割付けであり、その結果が目的コードの効率を大きく左右するから、これらも最適化の一貫と考えられる。レジスタ割付けのためにはフロー解析などのプログラム解析も行っており、それらの情報を使って簡単にできる最適化、たとえば無駄なジャンプ命令の削除、等も行っている。それらのモジュール構成は後で述べることにして、ここでは、レジスタ割付けについて述べる。

通常のコンパイラでは関数の局所変数は実行時のスタックフレームに割り付けられ、レジスタはそれらの変数を使った計算の途中結果に対して割り付けられる。LIR の PROLOGUE や EPILOGUE の 2 番目の項目はそのフレームの大きさなどを表現するものであり、COINS のコード生成系の最初のバージョンでもそのように使っていた。それに対して、次のバージョンでは、配列以外の局所変数（のうちアドレスをとる操作が行われていないもの）は最初に全部を仮想レジスタに割り付け、それに対してレジスタ割付けを行ったときに、レジスタ割り付けができなかったものだけをフレームに置く方式にしたので、この項目を使わないことになった。この後者の方式で、効率のよい

レジスタ利用が実現できている。

レジスタ割付けのアルゴリズムとしてはグラフ彩色法[2][3]がよく知られており、具体的な方法はいろいろ提案されているが、COINSでは標準的と考えられる方法[24]を基本としている。ただし、以下のような工夫を加えている。

(a) **干渉グラフのエッジに重みを付ける**

同じビット数の2つの変数が干渉しているときはそのエッジの重みは1とするが、ビット数が異なる場合は重みも異なる。たとえば、8ビット変数と16ビット変数が干渉している場合は、8ビットの変数側から見れば、16ビットのレジスタは8ビットレジスタ2つに相当するから、8ビット変数から16ビット変数へのエッジの重みは2とする。逆に、16ビットの変数側から見れば、8ビットレジスタが使われてしまえばそのレジスタに16ビットレジスタを割付けることは出来ないから、16ビット変数から8ビット変数へのエッジの重みは1とする。ある変数から出ているエッジの重みの和がそのビット数のレジスタの個数以下なら、その変数に対してはレジスタを割付けることが保証できるから、そのノードを干渉グラフから取り除いて残りのグラフについて割付ければ良い。これはグラフ彩色法のsimplifyと呼ばれる操作である。

(b) **スピルの選択法**

前項に述べたsimplifyを繰り返してノードがなくなればレジスタ割付けに成功したことになるが、それが出来なくなった場合にはどれかのノード（変数）をレジスタ割付けから外してメモリーに割り当てる（スピルするといわれる）ことにして、干渉グラフを作り直してレジスタ割り当てをやり直すことになる。スピルするノードの選び方としては、スピルのコスト（スピルしたことによって必要になるロード/ストア命令のコスト）が最小のものを選ぶのが通常の方法である。COINSでは、スピルするノードの選び方に新しい工夫をしている。それは、他のレジスタ割付けを妨害する度合いの低いものを選ぶ方法である。たとえば、以下のプログラム

```
0:  x = ...
1:  w = ...
2:  ..= w ...
3:  ..= x ...
```

図2 干渉グラフ

図3 妨害グラフ

では x の生存区間は [0-3) であり、w の生存区間は [1-2) である。生存区間に共通部分があるから干渉グラフでは x と w は干渉し、両者は同格である（図2）。しかし、スピルに関しては同格ではない。もし、x をスピルした場合は、w の生存区間に x の定義や参照は入っていないから、w にレジスタを割付ける際に x の影響はない（x と w のためにレジスタを1個使うだけですませることも出来る）。逆に w をスピルした場合は、x の生存区間に w の定義や参照が入っているので、w のために使うレジスタが x のレジスタと干渉してしまう（x と w のためにレジスタが2個必要になる）。このことを w が x へのレジスタ割付けを妨害すると考えて、図3のような妨害グラフで表現し、このグラフを使って、妨害のされ方が大きく、妨害のし方が小さいものを選んで、その中で（通常の方法である）スピルコストの小さいものをスピルの対象として選んでいる。結局この例では x をスピルする。

以上の他にもいくつかの最適化機能が追加されているが、それについてはバックエンドの機能拡張の項で述べる。

5.2 バックエンドの拡張性

5.2.1 対象機種変更容易なコード生成系

コード生成系を対象機種変更容易なものとする方法としては、中間表現のパターンを木文法（tree grammar）で表現し、その生成規則にマッチした中間表現に対して生成規則に対応するコードを生成する方法が知られている[2][3]。COINSでもその方法の1つで実際に使われた実績のある方法[25]を使っている。この方法では、コンパイル時にダイナミック・プログラミングによって最小コストのコードを生成する。その他の方法として、コード生成系を生成すると

きにそのダイナミック・プログラミングをやっておく方法もあるが、その方法では、コストが定数でしか表現できないことと、コード生成系を生成するのが容易ではないことから選ばなかった。

COINS のマシン記述では、LIR のパターンを木文法で表現し、生成するアセンブリ命令（アセンブリ言語で表現された命令）とその命令のコストを生成規則の属性として記述しておく。コード生成系では、与えられた LIR の木に対して生成規則とのパターンマッチングを行い、マッチするいくつかのパターンのうちのコストの和が最小となるものを選ぶ。

COINS では、細かな点でいくつかの工夫をしている。まず、命令のコストは 2 種類書けるようにしている。その 1 番目に速度のコスト（たとえばレイテンシ）、2 番目にスペースのコスト（たとえばバイト数）を書くことを想定している。コンパイルのオプションによって、どちらのコストの和が最小になるものを選ぶかを指定することが出来る。

前章でも述べたように、バックエンドのすべての処理が LIR に対する処理となるように、パターンマッチングをした後も LIR 形式を保っている。すなわち、命令選択ではマッチングした生成規則によって（ソースプログラムの形に対応した）LIR から（対象機種の機械語に対応した形に分解された）LIR への変換を行うだけである。木文法の生成規則には、対応するアセンブリ命令とそのコストが属性として記されているが、命令選択の際にはコストの属性を利用するだけでアセンブリ命令の属性は無視する。バックエンドの最後でアセンブリ命令の出力をする時に、もう一度パターンマッチングを行って、対応する生成規則の属性に書かれているアセンブリ命令が出力される。

5.2.2 マシン記述の容易性

COINS では、マシンの特性や上記の形の生成規則を TMD（Target Machine Description）として記述し、TMD から Java への変換プログラムによってコード生成系を生成している [16]。PROLOGUE 式に対するコード生成などのように、生成規則だけでは表現しにくい部分は、TMD の中に Java でその処理を記述している。

今までに作成された TMD の行数と開発に要した期間（1 人で従事した期間）を表 2 に示す。「TMD」欄が TMD で記述した行数であり、「内 Java」欄はその中で Java で記述した部分の行数である。「生成」欄は、TMD から生成された Java プログラムの行数で、参考のために載せてある。

表 2 マシン記述に要した期間と行数

機種	期間	TMD	内 Java	生成
MIPS	3ヶ月	2081	1379	4484
SH-4	6ヶ月	3568	2467	6677
PowerPC	6ヶ月	5016	2913	24498
SPARC	不明	1949	802	4993
x86	不明	2391	985	3487
ARM	6ヶ月	3052	2176	3822
Alpha	2ヶ月	1271	736	4303

SPARC と x86 の記述はマシン記述の仕様や処理系の開発と並行して行ったので、正確な期間は分からない。それ以外はいずれも数ヶ月で開発できており、しかも学部生の卒業研究として作成したものがほとんどであることから、記述が比較的容易であることがみてとれる。

以上を記述した後、生成規則などの記述力を再検討し、TMD の記述法を改良した。具体的には、マクロ機能をより柔軟なものにすることに加えて、以下が挙げられる。

(a) 共通 TMD ファイル

多くのマシン記述に共通に現れるものはその部分を別ファイルにしておいて、TMD から Java への変換の際にそれらを取り込む。

(b) 関数呼び出しの標準化

LIR では関数のプロローグ、エピローグ、呼び出し (call) は抽象化された表現になっている。これは機種独立な表現であり、機種独立な処理には適しているが、命令選択の前には、機種特有の呼びだし系列の情報などに変換する必要がある。それは、TMD の中では Java による記述の大きな部分を占めている。しかし、それにも多くの機種に共通な部分がある。そこで、その記述を標準化し、機種特性をパラメタ化した。

(c) 近接アドレス指定の標準化

命令のアドレス部に十分なビット数を使えない機種では、PC 相対アドレスなどで、比較的近接したアドレスしか使えない場合がある。アセンブラのプログラムでそのような近接アドレスを活かすことを TMD で記述することは容易ではなく、SH-4 マシンの TMD でも、定数を、それを使う命令の近くに置くための Java 記述などが大きくなっている。そのような機種は SH-4 に限らないので、近接アドレスの処理を標準化して、それぞれの機種ではその機種の特性を Java のクラスとして宣言しておくことによって、機種の特性に合わせた近接アドレス処理が行われるようにした。

たとえば、TMD では

```
        ldr     %r0,=W12345
```

という命令が生成されるようにしておくだけでよく(これはワード型の定数 12345 をロードする命令)、このアセンブリ命令が生成された後で、ポストプロセスとして、その機種で許される近接アドレスを使って次のように変換される。

```
        ldr     %r0,.LB1
        ...
.LB1:
        .word 12345
```

以上の 4 つの改良結果を利用して ARM マシンの TMD を書き直したところ、以前の記述（arm.tmd ファイル、3052 行、コメント等を除けば 2245 行）に比べて、新しく記述した同等機能の記述量（2426 行、コメント等を除けば 1701 行）が実質的な行数で約 25 %少なくなった。これは、記述がさらに容易になったことを示している。

5.2.3 バックエンドの機能拡張の容易性

バックエンドのモジュール構造が分かりやすく分割されていて、モジュールの追加・変更が容易であれば、バックエンドの機能拡張が容易になる。COINS のバックエンドの全体のフェーズの動きは次のように記述されている。

```
  unit.apply(new String[] {
      "IntroVirReg",
      "EarlyRewriting",
      "+AfterEarlyRewriting",
      "If2Jumpc”,
      "+BeforeBasicOpt",
      "SimpleOpt",
      "JumpOpt",
      "PreHeaders",
      "LoopInversion",
      "+AfterBasicOpt”,
      "+BeforeCodeGeneration",
      "LateRewriting",
      "+AfterLateRewriting",
      "ToMachineCode",
      "+AfterToMachineCode",
      "ConvToAsm"  });
```

ここで、unit は、LIR の 1 つの MODULE であり、コンパイルの対象となっているソースプログラムがコンパイルの過程で LIR の形に変換されてバックエンドに渡されているものである。それに対して、上記の各文字列に対応するメソッドが適用される。文字列の中の"+"の意味は後述する。最初の"IntroVirReg"は、関数に局所的な単純変数を仮想レジスタに割付ける処理をする。次の"EarlyRewriting"は、TMD の記述に従って LIR の変換処理を行う。このように TMD には生成規則による命令選択だけでなく、その前後に行う変換処理も記述できる。関数の PROLOGUE を機種特有のものに書き換えるのはこれによって行われる。たとえば、TMD 記述の中で、

```
    (defrewrite (before)
      (to (after))
      (phase early) )
```

と書いてあれば、それは"EarlyRewriting"で実行され、*before* が *after* に書き換えられる。"(phase late)"ならば、上記の後ろの方にある"LateRewriting"で実行される。

"ToMachineCode"では、関数単位に次のものが実行される。

```
  apply("LiveRange");
  apply("JumpOpt");
  apply("JumpCanon");
  apply("+BeforeFirstInstSel");
  apply("InstructionSelection");
  apply("+AfterFirstInstSel");
  for (;;) {
    if (apply("RegisterAllocation"))
          break;
    apply("+BeforeSecondInstSel");
    apply("InstructionSelection");
    apply("+AfterSecondInstSel");
```

```
  }
  postProcess();
```

上記の文字列の中で先頭に"+"がついているもの、たとえば、

```
  apply("+AfterFirstInstSel");
```

を実行しても、通常は何も起こらない。しかし、その名前で登録されているオブジェクトがあると、そのオブジェクトの doIt メソッドを実行する。すなわち、これは任意の機能を追加するためのフックである。たとえば、

```
class NewClass {
  class NewTrigger {
    void doIt(...){...}
  }
  static void attach() {
    root.addHook("+AfterFirstInstSel",
                 new NewTrigger());
  }
}
```

というクラスを作っておいて、コンパイラのオプションでクラス名 NewClass を指定すると、バックエンドで NewClass.attach メソッドを呼び、NewTrigger のオブジェクトを"+AfterFirstInstSel"に登録する。その結果、

```
  apply("+AfterFirstInstSel");
```

によって、上記の doIt メソッドが実行される。このようにして、COINS の本体に手を加えずに機能を追加することが出来る。

実際に、この方法で、"+AfterEarlyRewriting"で実行されるレジスタ・プロモーション [26]、レジスタ割付け前（"+AfterFirstInstSel"）とレジスタ割付け後（"+AfterToMachineCode"）に実行される命令スケジューラ、レジスタ割付け前（"+AfterFirstInstSel"）に実行されるソフトウェア・パイプライニング [27]、などの機能が実現されている。なお、これらのプログラムの対象は特定の機種の命令選択をした結果の LIR であるが、それは機種独立な LIR の形であるので、プログラム自身は特定の機種によらずに書かれており、実際に同じプログラムが複数の機種に適用できる。

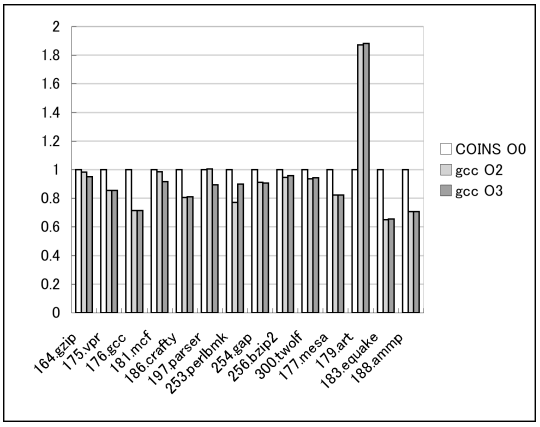

図 4　COINS の性能（SPEC2000, x86）

6　性能評価

コンパイラの生成する目的コードの性能評価は、ベンチマークとしてよく使われている SPEC CPU2000 ベンチマークと、典型的な小さなプログラムを使って行った。SPEC ベンチマークのような大規模なプログラムは全体的な評価をするのに適しており、小さなプログラムは性能の違いの解析をするのに適している。評価はフリーのコンパイラで性能的にも定評のある GCC の最適化レベル 2 あるいは 3 で得られるものと比較することで行った。以下のいずれの図も各ベンチマークの目的コードの実行時間に関して COINS で最適化を何も指定しない場合の実行時間を 1 とした相対値で表している。

6.1　SPEC ベンチマークでの評価

6.1.1　x86 での評価

図 4 は、ノートブック EPSON Pro-1000（Pentium 4, 1.8 GHz, 256 MB）で、COINS の最適化を何も指定しない場合と GCC（バージョン 2.95.4）の最適化レベル 2、3 との結果をを比較したものであり、1 つを除いて GCC より時間がかかっているがあまり大きな差はない。COINS で最適化を何も指定しない場合はバックエンドで簡単な最適化とマシン記述による命令選択とレジスタ割付けをやっているだけであるが、x86 のようにレジスタの個数が少ない機種ではレジス

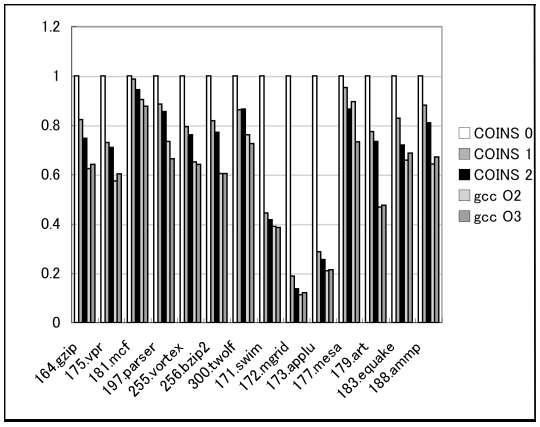

図 5 COINS の性能（SPEC2000, SPARC）

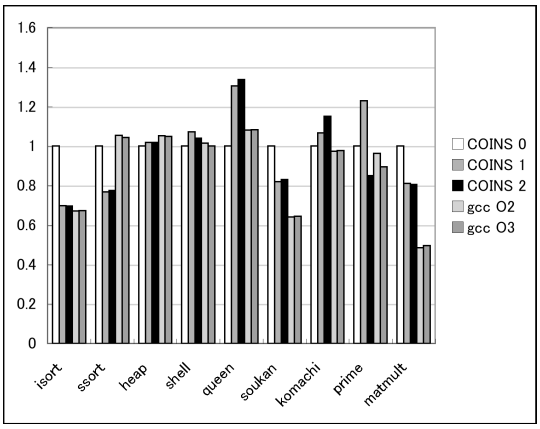

図 6 COINS の性能（小プログラム, x86）

タ割付けの効果が大きく、それだけで比較的良い性能が出ている。

しかし、さらに COINS が備えている多くの最適化を適用してみても、性能が向上せず、かえって悪くなるものも多い。多くの最適化はレジスタの個数に余裕があるときは良い効果をもたらすが、x86 のようにレジスタの個数が少ない機種では逆効果になる場合がある。

6.1.2 SPARC での評価

図 5 は、Sun Ultra 60（UltraSPARC-IIs, 450MHz デュアル, 1 GB）で、COINS で最適化をいろいろ指定した場合と GCC（バージョン 3.3）の最適化レベル 2、3 との結果を比較したものである。COINS 0 は最適化指定なし、COINS 1 は SSA 最適化の指定、COINS 2 はそれにさらに命令スケジュールとソフトウェア・パイプライニングの指定をしたものである。図では、最適化をしないと GCC と大きな差があるが最適化をすれば近づくことが分かる。

しかし、GCC との間にはまだ差がある。その原因はまだ分かっていないが、GCC では機種特有の最適化をしていると思われるのに対して、COINS では、インフラストラクチャであることを重視して、最適化はすべて機種独立な形で行っているのが 1 つの原因である可能性がある。

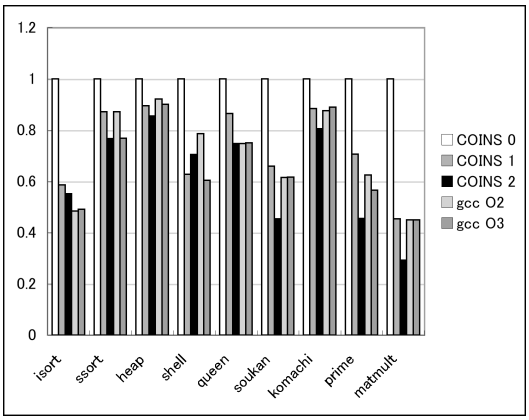

図 7 COINS の性能（小プログラム, SPARC）

6.2 小さなプログラムによる評価

プログラムによる最適化効果の違いなどを評価するために、小さなプログラムによる評価も行った。プログラムは、挿入ソート（isort）、選択ソート（ssort）、ヒープソート（heap）、シェルソート（shell）、8 クイーン（queen）、相関係数（soukan）、小町算（komachi）、素数（prime）、行列積（matmult）の 9 つである。最適化の指定は、前節の SPARC の場合とほとんど同じであるが、COINS 2 に大域変数レジスタ・プロモーションが付け加えられている点が異なる。それを付け加えた理由は後で述べる。

6.2.1 x86 での評価

図 6 は、DynaBook G9（Pentium 4, 2.4 GHz, 512 MB）での結果を比較したものである。GCC のバー

ジョンは 3.4.4 である。heap, shell, queen については最適化の効果はなく、何も最適化をしない COINS 0 が最速の結果を出している。これは x86 マシンでの最適化の難しさを示していると思われる。soukan と matmult は浮動小数点演算の多いプログラムであるが、COINS の最適化効果は GCC ほど出ていない。その 1 つの原因は x86 のスタック型浮動小数点レジスタのセットを、機種独立な命令スケジューラなどでは 1 つのレジスタとして扱わざるを得ず、最適化の効果が出せないからである。prime で COINS 1 の性能が出なかったのは、大域静的変数がループの中で多用されていて、それに対する最適化が出来ていなかったからである。これを踏まえて、COINS 2 に、ループの中での大域静的変数の使い方を調べて、可能ならばループ中ではレジスタに割付けるレジスタ・プロモーション [26] の最適化を追加した。この追加は、バックエンドの機能追加の容易性を検証する実験ともなった。その結果 COINS 2 では GCC を上回る性能となった。

6.2.2 SPARC での評価

図 7 は、Sun Blade 1000（UltraSPARC-III, 750 MHz デュアル, 1GB）での結果を比較したものである。GCC のバージョンは 3.4.2 である。9 件のプログラム中の 7 件で COINS 2 が最速の結果を出している（GCC とほとんど同じものも含めて）。SPARC は x86 に比べてレジスタ数が多いので、汎用のアルゴリズムでの最適化の効果が出やすいといえる。特に prime, soukan, matmult が GCC より大分性能が向上しているが、prime は前項に述べた理由によるものであり、soukan, matmult は GCC では行われていないソフトウェア・パイプライニングの効果によるものである。

7 おわりに

COINS コンパイラ・インフラストラクチャとして、当初の目標を満たすものを開発することが出来た。インフラストラクチャとして使いやすいものとすることを心がけてきたが、その実際の評価には、COINS の使用者からの評価を待つ必要がある。

今後やるべきこととしては、COINS の保守・改良はもちろんであるが、今まで十分には行えなかった並列化の機能や、64 ビット対応の強化などもある。

謝辞：COINS は 2000 年度から 5 年間の科学技術振興調整費のプロジェクト「並列化コンパイラ向け共通インフラストラクチャの研究」として開発し、さらに、2006 年の 1 月から 8 月までは、IPA の次世代ソフトウェア開発事業の 1 つのプロジェクトとして機能追加を行ったものである。前者のプロジェクトの研究運営委員会および研究 WG で助言を頂いた方々、COINS の開発に多大の努力を傾注されたプロジェクトのメンバー、学生アルバイトとして参加された笹田耕一（Fortran 担当）、荒川傑（Java 担当）の皆さんに深く感謝します。

また、8000 件のテストプログラムを快く貸してくださった引地信之（SRA 社）、学生の Java コンパイラの指導に貴重な時間を割いてくださった千葉雄司（日立システム開発研究所）、早期に COINS を教育に利用 [28] して貴重なご意見をくださった山之上卓（鹿児島大）の諸氏に深く感謝します。

参 考 文 献

[1] http://www.coins-project.org/
[2] 中田育男：コンパイラの構成と最適化、朝倉書店、1999.
[3] Cooper, K. and Torczon, L.: Engineering a Compiler, Morgan Kaufmann, 2004.
[4] http://www.jp.arm.com/products/processors/architecture.html
[5] http://www.spec.org/benchmarks.html
[6] http://ist.ksc.kwansei.ac.jp/ ishiura/testgen/
[7] Feldman, S.I., Gay, D.M., Maimone, M.W., Schryer, N.L.: A Fortran-to-C Converter, Computing Science Technical Report No. 149 (Bell Lab.), Originally issued May 16, 1990. Last updated March 22, 1995.
[8] http://gcc.gnu.org/java/
[9] http://www.coins-project.org/contributions/CoinsJava/
[10] http://gcc.gnu.org/onlinedocs/gccint/RTL.html
[11] Abe, S., Hagiya, M. and Nakata, I.: A Retargetable Code Generator for the Generic Intermediate Language in COINS, 情報処理学会論文誌：プログラミング, Vol. 46, No. SIG 14 (PRO 27), pp. 12–29, 2005.
[12] http://www.coins-project.org/shiyosho.html の「低水準中間表現仕様書」
[13] Suzuki, M., Fujinami, N., Fukuoka, T., Watanabe, T., Nakata, I.: SIMD Optimization in COINS

Compiler Infrastructure, Post Proc. 8th International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA2005), pp. 131–140, 2005.

[14] 鈴木貢, 藤波順久, 福岡岳穂, 渡邊坦, 中田育男：マルチメディア SIMD 命令活用のためのデータサイズ推論, 情報処理学会論文誌：プログラミング Vol. 45, No.SIG 5(PRO 21), pp. 1–11, May 2004.

[15] http://www.coins-project.org/advanceduse.html

[16] http://www.coins-project.org/doc.html

[17] http://www.coins-project.org/international/use.html

[18] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K.: Efficiently computing static single assignment form and the control dependence graph, ACM Trans. Prog. Lang. Syst., Vol. 13, No. 4, pp. 451–490, 1991.

[19] Briggs, P., Cooper, K.D., Harvey, T.J. and Simpson, L.T.: Practical Improvements to the Construction and Destruction of Static Single Assignment Form, Software – Practice and Experience, Vol. 28, No. 8, pp. 859–881, 1998.

[20] Sreedhar, V.C., Ju, R D.-C., Gillies, D.M. and Santhanam, V.: Translating Out of Static Single Assignment Form, Proceedings of the 6th International Symposium on Static Analysis, Lecture Notes in Computer Science, Vol. 1694, Springer-Verlag, pp. 194–210, 1999.

[21] 滝本宗宏，福岡岳穂，佐々政孝，原田賢一: 疎な要求駆動型データフロー解析，情報処理学会論文誌：プログラミング, Vol. 46, No. SIG 11 (PRO 26), pp. 16–26, 2005.

[22] Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Global Value Numbers and Redundant Computations, Proc. Principles of Programming Languages (POPL'88), pp. 12–27, 1988.

[23] 佐々政孝，福岡岳穂，滝本宗宏：コンパイラ・インフラストラクチャにおける静的単一代入形式最適化部の実現，情報処理学会論文誌：プログラミング，Vol. 47, No. SIG 2 (PRO 28), pp. 30–43, Feb. 2006.

[24] George, L. and Appel, A.W.: Iterated Register Coalescing, ACM TOPLAS, Vol. 18, No. 3, pp. 300–324, 1996.

[25] Fraser, C.W., Henry, R.R. and Proebsting, T.A.: BURG – Fast Optimal Instruction Selection and Tree Parsing, SIGPLAN Notices, Vol. 27, No. 4, pp. 68–76, 1992.

[26] Cooper, K.D. and Lu, J.: Register promotion in C programs, PLDI' 97 (Proc. of the ACM SIGPLAN 1997 conference on Programming language design and implementation), pp. 308–319, 1997.

[27] 中田育男：ソフトウェア・パイプライニングの一実現法、情報処理学会論文誌：プログラミング, Vol. 47, No. SIG 16 (PRO 31), pp. 44–51, Oct. 2006.

[28] 山之上 卓：並列化コンパイラ向け共通インフラストラクチャCOINS を利用したコンパイラ教育の試み、情報処理学会 SSS2004 情報教育シンポジウム論文集、pp. 155–158, Nagano, Japan, 28–30 Aug. 2004.