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

佐々 政孝[†]　福岡 岳穂[††]　滝本 宗宏[†††]

静的単一代入形式（以下 SSA 形式と略す）は，変数の定義が字面上唯一になるように添字をつけた中間表現形式であり，コンパイラにおけるデータフロー解析や最適化を見通し良く行うのに適している．この SSA 形式に基づく変換と最適化をコンパイラ・インフラストラクチャCOINS に組み込んだ．その設計方針と実装および評価について述べる．実現したものは (i) SSA 形式への変換，(ii) SSA 形式からの逆変換，(iii) SSA 形式上の基本的最適化と高度最適化と種々の変換，である．本研究は次のような点に特徴がある．(a) 複数のアルゴリズムが知られているものについては，比較検討のうえ，採用するアルゴリズムを決定した．(b) 共通インフラストラクチャとして基本的な最適化を含めるとともに，研究基盤として活用できることの例示としてオリジナルなアルゴリズムによる最適化も含めた．(c) アルゴリズムの比較評価のためにアルゴリズムのバリエーションをオプションで指定できるようにした．(d) SSA 形式の分野で問題として示唆されてはいるが実際にどう解決するかが示されていない種々の点について，それぞれ 1 つの解決法を示した．(e) 参照実装として用いることができるように，読解性，保守性の良いコーディングを心がけ，以上の点を詳しい仕様書として提供した．

Realization of Static Single Assignment Form Optimization Module in a Compiler Infrastructure

Masataka Sassa,[†] Takeaki Fukuoka[††] and Munehiro Takimoto[†††]

Static single assignment form (hereafter SSA form) is an intermediate representation where the definition of each variable is textually unique. It is suited for dataflow analysis and optimization in compilers. We implemented transformation and optimization module based on this SSA form using a compiler infrastructure called COINS. In this presentation, we show its design, implementation and evaluation. The module includes (i) transformation into SSA form, (ii) back translation from SSA form, (iii) basic and advanced optimization in SSA form.

This research has the following characteristic features; (a) when there is a plurality of known algorithms we decided the algorithm to be implemented by thoroughly comparing them. (b) basic optimizations as well as original optimization algorithms are included. (c) variations of algorithms can be specified by command options. (d) we gave a solution to problems suggested but not solved in the SSA form. (e) readable and maintainable coding is made and detailed documents of the program are provided.

1. はじめに

静的単一代入形式（以下 SSA 形式）は，変数の定義が字面上唯一になるように添字をつけた中間表現形式であり，コンパイラにおけるデータフロー解析や最適化を見通し良く行うのに適している．

SSA 形式を用いると，最適化の実現容易性と最適化の実行効率が向上するといわれている．このためいくつかの最適化コンパイラ[9),12),13),21)]がその一部のパスで SSA 形式を採用するようになってきている．

しかし，一般の研究者や開発者が SSA 形式最適化の研究を行おうと思っても，そのベースとして使える適当なコンパイラがほとんど存在しなかった．SUIF[29)]などが思い浮かぶかもしれないが，SUIF は SSA 形式を扱わず，わずかに Machine SUIF[16)]が無用命令除去の SSA 最適化を含むのみであり，そのリリースも 2002 年以後は出ていない．

† 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
†† 株式会社ソニー・コンピュータエンタテインメント
Sony Computer Entertainment Inc.
††† 東京理科大学理工学部情報科学科
Department of Information Sciences, Faculty of Science and Technology, Tokyo University of Science

SSA 最適化に限らず，種々の最適化や多様な対象機種のコード生成のベースになることを目指して起こされたプロジェクトが「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」(1999 年度〜2004 年度）である．作成されたコンパイラ・インフラストラクチャは COINS（a COmpiler INfraStructure）と呼ばれる[6]．

本研究は，SSA 形式に基づく変換と最適化をこの共通インフラストラクチャCOINS に組み込むにあたっての設計方針，実装および評価について述べる．

実現したものは，COINS の低水準中間表現における，(i) SSA 形式への変換，(ii) SSA 形式からの逆変換，(iii) SSA 形式上のデータフロー解析と基本的最適化および高度最適化，(iv) SSA 形式上の最適化のために有用な様々の変換，である．

最適化モジュールをコンパイラ・インフラストラクチャに実装するにあたっては，研究目的だけではない種々の考察と決断が必要である．実装や設計の方針をきちんと立てること，論文などで既知のものを取り込むことと新たな研究とのバランス，既発表のアルゴリズムでも細部に誤りが残っていたり問題点が指摘されているものを措置すること，今後の参照実装となるように読みやすく保守しやすいコードとドキュメントを残すこと，などである．

本研究では，これらの点についての方針を立て，試行錯誤の段階をも経ながらひとまずの完成をみた．本研究は次のような点に特徴がある．(a) 複数のアルゴリズムが知られているものについては，比較検討のうえ，採用するアルゴリズムを決定した．(b) 共通インフラストラクチャとして基本的な最適化を含めるとともに，研究基盤として活用できることの例示としてオリジナルなアルゴリズムによる最適化も含めた．(c) アルゴリズムの比較評価のためにアルゴリズムのバリエーションをオプションで指定できるようにした．(d) SSA 形式の分野で問題として示唆されてはいるが実際にどう解決するかが示されていない種々の点について，それぞれ 1 つの解決法を示した．(e) 参照実装として用いることができるように，読解性，保守性の良いコーディングを心がけ，以上の点を詳しい仕様書として提供した．

次章以降では，この経験が伝わるよう努めつつ，これらについて述べる．

また，評価として SPEC ベンチマークによる SSA 形式最適化の効果も述べる．

これにより，利用者が容易に SSA 形式を扱うことができるようになり，共通インフラストラクチャが教育，研究用基盤として有用なものになったと考える．

2. 準　　備

準備として，コンパイラ・インフラストラクチャ COINS および SSA 形式について説明する．

2.1 コンパイラ・インフラストラクチャCOINS

COINS（a COmpiler INfraStructure）とは，新しいコンパイラ方式を容易に実験，評価できるようなコンパイラの共通インフラストラクチャである[6]．

図 1 にあるように，COINS は高水準中間表現 HIR（High-level Intermediate Representation）と低水準中間表現 LIR（Low-level Intermediate Representation）を持ち，ソース言語から HIR への変換部，HIR での各種最適化部，HIR から LIR への変換部，LIR での各種最適化部，LIR から機械語への変換部を組み合わせることによってコンパイラを実現できる．

現在，入力言語として C，Fortran 77，対象機種として SPARC，Intel x86 が正式サポートされ，その自由な組合せが可能となっている．

また，共通部分式除去などの基本的な最適化機能と，各種の最適化機能が備わっている．そのほか，ループ並列化，SMP 並列化，適用範囲に制限はあるが先進的な SIMD 並列化の機能も組み込まれている．

ただし，現状では，別名解析は手続き内の簡単なものが HIR レベルでのみ実装され，コード生成における命令スケジューリングなどは試験実装の段階である．

2.2 SSA 形式

SSA 形式について簡単に説明する．詳しくは教科書などを見られたい[1),19)]．

図 2 (a) のような通常形式のプログラムがあったとする．これを SSA 形式に変換すると図 2 (b) のように

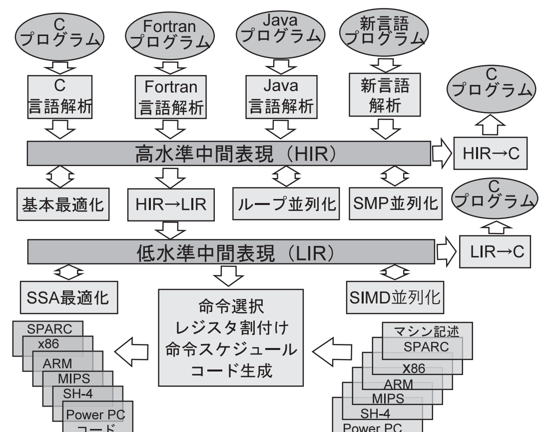
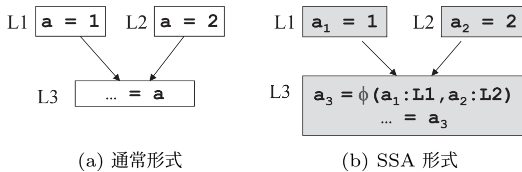


図 1 コンパイラ・インフラストラクチャCOINS
Fig. 1 Compiler infrastructure COINS.



(a) 通常形式　(b) SSA 形式

図 2 SSA 形式と ϕ 関数
Fig. 2 SSA form and ϕ function.

なる．まず，各変数の定義が字面上唯一となる．たとえば a について見ると，代入があるごとに変数の新しいバージョン（version）を作り，a_1，a_2 のように添字をつけて区別する．制御の合流点に同じ変数の異なるバージョンが到達するときは，ϕ 関数というものを用いる．たとえば，図 2 (a) を SSA 形式に変換しようとすると，最後の行で使用している a に対する定義を一意に決めることはできない．そこで，SSA 形式では図 2 (b) のような ϕ 関数を用いる．$\phi(a_1$:L1，a_2:L2) は，基本ブロック L1 から来たときは a_1 の値を返し，基本ブロック L2 から来たときは a_2 の値を返す仮想的な関数を表す．

3. 設 計 方 針

SSA 形式最適化モジュールの設計と実装にあたっては，次のような方針を定めた．

(1) 複数のアルゴリズムが知られているものについては，比較検討のうえ，採用するアルゴリズムを決定する．
(2) 共通インフラストラクチャとして基本的な最適化を含める．
(3) 研究基盤として活用できることの例示として，オリジナルなアルゴリズムによる最適化も含める．
(4) アルゴリズムの比較評価のために，個々のモジュールの起動やアルゴリズムのバリエーションをオプションで指定できるようにする．
(5) SSA 形式の分野で問題として示唆されてはいるが実際にどう解決するかが示されていない種々の点について，それぞれ 1 つの解決法を示す．
(6) 参照実装として用いることができるように，読解性，保守性の良いコーディングを心がけるとともに，詳しい仕様書を提供する．

以下，これらの方針との関連を示しながら，SSA 形式最適化モジュールの各部分の実現について述べる．

4. SSA 最適化部

作成された SSA 最適化モジュール（以後 SSA 最適化部と呼ぶ）の全体構成を**図 3** に示す．これは次の機能を備えた部分からなる．

(1) SSA 形式への変換（LIR→SSA 変換）
SSA 形式への効率良い変換を行う．
(2) SSA 形式からの逆変換（SSA→LIR 逆変換）
SSA 形式から実行効率の良い通常形式への変換を行う．
(3) SSA 形式上のデータフロー解析と基本的最適化および高度最適化
SSA 形式上でデータフロー解析を行い，最適化を行う．基本的な最適化をほぼ網羅しており，いくつかの高度な最適化も実現している．
(4) SSA 形式上の最適化のために有用な変換
SSA 形式上の最適化のために有用な種々の変換を行う．

以下，SSA 最適化部について，設計思想や採用の経緯を示しつつやや詳しく述べる．

4.1 SSA 最適化部の位置

まず，SSA 最適化をどのレベルの中間表現で実現するかについて検討した．当初は高水準中間表現（HIR）のうえでの実現も検討し，プロトタイプも作成した．しかし，HIR がソースプログラムの構造を保つ木構造を基にしているため，goto 文など非構造的なプログラム構造での ϕ 関数の扱いに困難が生じることが分かった[20]．そこで以後は低水準中間表現（LIR）のうえで実現することとした．結果的に HIR 上のこのプロトタイプは採用されなかったが，これを用いた実験や書かれたコードはその後の実装に役立った．

図 3 にあるように，SSA 最適化部は，COINS 基盤部から LIR の制御フローグラフ（CFG）を受け取り，それをまず SSA 形式の CFG に変換する．次いで，SSA 形式 CFG 上でデータフロー解析や種々の最適化および変換を行う．最後に SSA 形式 CFG を再び通常形式の LIR の CFG に戻し，基盤部へ渡す．基盤部は，そこから目的コードを生成する．

COINS 基盤部での LIR の処理はいくつものステップに分かれており，各段階で LIR の中身も変わっていくが，LIR がマシン非依存である間は，任意の段階で SSA 形式への変換と最適化を行うことができる．

なお，SSA 最適化部の行数は合計で約 15,000 行である．

4.2 SSA 形式への変換

SSA 形式への変換を行うためには，「設計方針」の (1) で述べたように，まず数多く提案されている既存の SSA 変換アルゴリズムの評価が必要である．そこで，有力と思われる 2 種類の SSA 変換のアルゴリズ

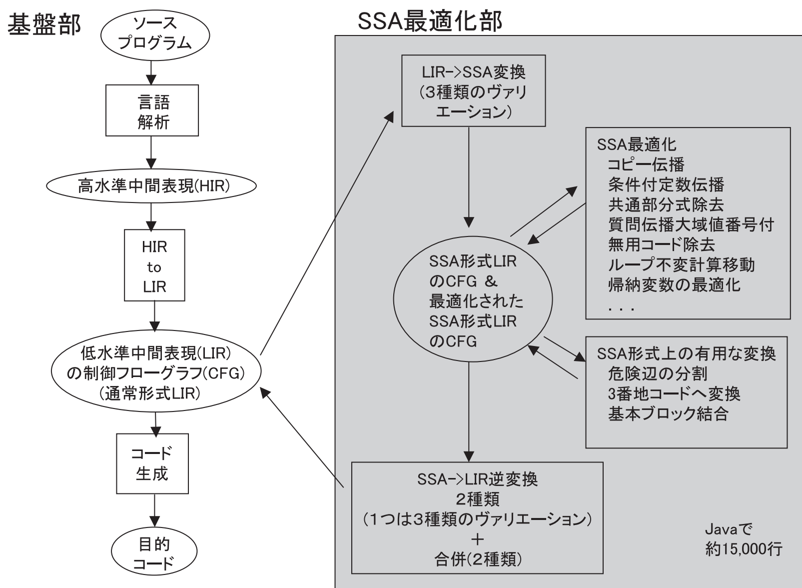


図 3　SSA 最適化部の構成
Fig. 3　Structure of SSA form optimizer.

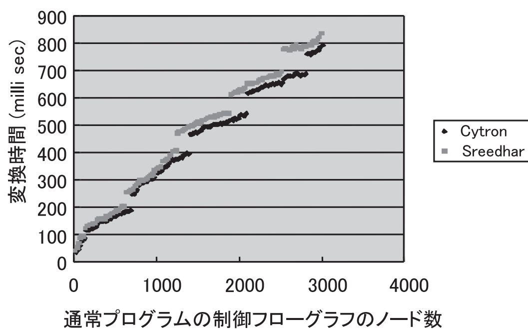


図 4　SSA 変換の変換時間の比較
Fig. 4　Comparison of SSA translation time.

ムについて，プロトタイプを作成して評価した．

通常のプログラムに対する SSA 変換の 2 種類の方法の変換時間の比較を**図 4** に示す．この図から，通常のプログラムでは，Cytron らによる方法[8]（制御フローグラフのすべてのブロックの支配辺境をあらかじめ求めておく方法）による変換時間が Sreedhar らによる方法[25]（DJ グラフというデータ構造を用いて支配辺境を求めるのと同時に ϕ 関数を挿入する方法）による変換時間よりもやや少なく良好であることが見てとれる☆．ちなみに，2 つの方法の変換結果に差はない．この評価結果とアルゴリズムの単純さから，Cytron の方法を採用した[20]．

具体的には，LIR の制御フローグラフのうえで変換を行う．SSA 変換の際に同時にコピー畳み込みを行ったり，無用な ϕ 関数を除去したりすることもできる．

SSA 形式には，最小（minimal）SSA，半ば刈り込んだ（semi-pruned）SSA，刈り込んだ（pruned）SSA の 3 つの形式がバリエーションとして提案されている[4]．「設計方針」の (4) で述べた方針に従い，アルゴリズムの比較評価ができるようにこれらのバリエーションをすべて実装し，オプションで指定できるようにした（後の評価の章でオプション名が出てくるため，オプション名を述べておく．これらはそれぞれ，`mini`，`semi`，`prun` で指定できる）．

4.3　SSA 形式からの逆変換

SSA 形式のままではコード生成ができないので，通常形式への逆変換が必要である．

SSA 形式から通常形式への逆変換についてはいくつかの危うい状況があることが知られている[4],☆☆．それを解決した主なアルゴリズムには Briggs らの方法[3],[4] と Sreedhar らの方法[26] がある．それらを詳細に比較検討したところ，SSA 逆変換の際に挿入されるコピー文の数などから，Briggs らの方法の適用後に合併（co-

☆ 制御フローグラフのあるブロックの支配辺境とは，そのブロックから制御フローグラフの辺をたどっていって初めてそのブロックの支配から外れたブロックである．ϕ 関数は各定義および挿入された ϕ 関数の支配辺境に置かれる．DJ グラフは，ϕ 関数の挿入の計算量を抑えるために，支配木に制御フローグラフの辺の一部を加えたデータ構造である．

☆☆ 素朴な逆変換アルゴリズムでは，コピー文を挿入して ϕ 関数を消去する．しかし，SSA 形式上で最適化を行った後に素朴な逆変換を施すと，結果が正しくなくなる場合がある．これを危うい状況と呼ぶ．日本語での説明は文献 14) などにある．

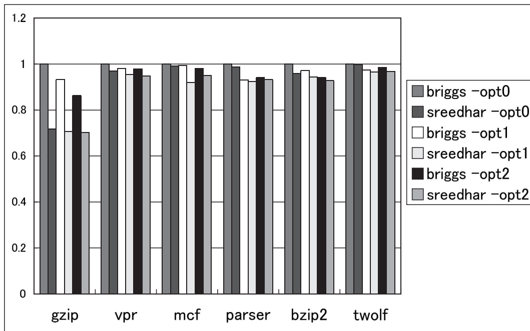


図 5 SSA 逆変換における Briggs 法，Sreedhar 法による目的コードの実行時間の相対比（レジスタ数 8）
Fig. 5 Ratio of runtime after SSA back translation by Briggs's and Sreedhar's methods (8 registers).

alescing）を行ったとしても，Sreedhar らの方法に優位性があると認められた[15]．そこで，既存のコンパイラでは採用が稀であったが，Sreedhar らの方法を採用した．なお，合併とは，コピー文で結ばれる異なる変数どうしをを 1 つの変数にまとめ，コピー文を削除するものである．

その後，東京工業大学の研究室で別途 Briggs らの方法を実装し，さらに Briggs らの方法の改善案も提案し，Sreedhar らの方法との比較を実測評価した[14),23)]．いずれの方法でも，レジスタ割当て時に合併を行っている．それぞれの方法を用いて逆変換した後の目的コードの実行時間の比較の一部を**図 5** に示す．

この図は，SUN Blade 1000（UltraSPARC III, 750 MHz×2，メモリ 1 GB）を用い，SPEC CINT2000 ベンチマークについて入力を ref とし，種々の最適化をかけたときの実行時間の相対比である（-opt0 は最適化なし，-opt1 は 4 つの最適化パス，-opt2 は 8 つの最適化パスをかけたもの，Briggs らの方法の最適化なしを 1 とした相対比）．

この図から Sreedhar らの逆変換法を用いた目的コードの実行時間が，Briggs らの逆変換を用いたものより小さいことが見てとれる．この結果，当初の選択が正しいことが実証された．

ちなみにその後，研究室で実装した Briggs らの方法による SSA 逆変換も COINS に組み込み，このインフラを用いて追実験ができるようにした．

Sreedhar らの逆変換法には，3 つのバリエーション Method 1，2，3 がある．「設計方針」の (4) で述べた方針に従い，アルゴリズムの比較評価のためにこれらのバリエーションをすべて実装し，オプションで指定できるようにした（オプション名はそれぞれ `srd1`，`srd2`，`srd3` である）．

また，SSA 逆変換後のコピー文を減らす目的で，2 種類の合併（SSA-based coalescing[26)] と Chaitin の方法[5)]）を実装し，これらもオプションで指定できるようにした．

4.4 SSA 形式上の最適化

4.4.1 SSA 形式上の基本最適化

「設計方針」の (2) で述べた方針に従い，SSA 形式上の最適化としては，まず共通インフラストラクチャとして基本的な最適化を含めることとした．

SSA 形式による基本的な最適化としては，次を実現した（括弧内はオプション名である）．

- コピー伝播（`cpyp`）
- 条件分岐を考慮した定数畳み込みと定数伝播[31)]（`cstp`）
- 支配関係に基づく共通部分式除去[19)]（`cse`）
- 無用コード除去（`dce`）
- ループ不変計算のループ外移動（`hli`）
- ループの帰納変数に関わる演算の強さの軽減と判定の置き換え[7)]（`osr`）
- 無用な ϕ 関数の除去[3)]（`rpe`）
- 空の基本ブロックの除去（`ebe`）

これらは，オプションの指定により，任意のものを任意の回数，任意の順序で適用できる．参考文献のないものは，通常形式での最適化から比較的容易に類推できるものである．無用コードの除去については 5 章で述べる．実装したアルゴリズムは文献 27) に記した．

4.4.2 SSA 形式上の高度最適化

一方，SSA 形式上の高度な最適化もいくつか実装した．また，「設計方針」の (3) で述べた方針に従い，この SSA 最適化部が研究基盤として利用できることの例として，既存のものでない新たに考案したアルゴリズムによる最適化も含めた．

これらの最適化としては，

- 大域的再結合 (global reassociation)[2)]（オプション名 `gra`）
- メモリ全体を 1 つの塊と見なす素朴な別名解析（デフォルトで適用．適用を止めるオプションは `ssa-no-memory-analysis`）
- 効率的な質問伝播を用いた大域的値番号付けによる部分冗長性除去（オプション名 `preqp`）

がある．後者 2 つは本研究で新たに考案したアルゴリズムである．

COINS の LIR では別名の情報が得られないので，別名が生じる可能性のある変数については安全のためメモリに置いたままにしている．「メモリ全体を 1 つの塊と見なす素朴な別名解析」とは，このような変数

についても，メモリへの書き込みがない間は，この変数に仮に別名があってもこの変数に変更はない，と見なす解析である．これにより，共通部分式の対象が広がるなどの効果がある．

たとえば，選択ソートの最内ループ

```
if (a[j] < min) {
    min = a[j];
    k = j;
}
```

では，a[j] が 2 回使用されるが，その間にメモリへの書き込みはない（min などのスカラ変数はレジスタにプロモートされている）．したがって，その間 a[j] の値に変更がないことが分かり，コンパイラは a[j] を読み込むコードに関する共通部分式除去を行ったのち，a[j] が 1 回だけメモリから読み込まれるようなコードを生成する．生成されるコードを疑似的に書くと次のようになる．

```
if (t = a[j], t < min) {
    min = t;
    k = j;
}
```

具体的な解析のアルゴリズムは文献 27) に記した．

なお，この最適化は，gcc の-O2 オプションに含まれる-fforce-mem オプション（メモリ上のオペランドを算術演算を行う前にレジスタにコピーする）などと類似の効果が見込まれる．ただしここで意図したのは，SSA 形式上でメモリの別名を扱う簡単な方法の開発である．

一方,「効率的な質問伝播を用いた大域的値番号付けによる部分冗長性除去」とは，質問伝播という方法により，SSA 形式の ϕ 関数が間に挟まれているような共通部分式も効率良く発見することができ，部分冗長性除去を含む統一的な冗長性除去を行うアルゴリズムである[27]．これについては別途発表予定である．

4.4.3 SSA 形式最適化のインフラとして有用な種々の変換

部分冗長性除去などのアルゴリズムでは，危険辺（critical edge）がないことを仮定しているものが多い．そこで，SSA 形式最適化のインフラとして有用な種々の変換処理として，次のようなものを実装した[27]（括弧内はオプション名）．

- ループ構造の変換（while 型ループを if-do-while 型ループへ変換する，共通のヘッダを持つ入れ子でないループをマージする）（デフォルトで適用．適用を止めるオプションあり）
- 危険辺の分割（esplt）[☆]
- 式の 3 番地コードへの変換（代入の右辺の演算子をたかだか 1 つにする）（divex）
- SSA グラフの作成（ssag）
- 基本ブロックの結合（cbb）

これらも，1 つ目のものを除き，オプションの指定により，任意のものを任意の回数，任意の順序で適用できる．

5. 実装にあたっての問題点の解決

実装にあたっては，解決すべき点がいろいろある．たとえば，既発表のアルゴリズムでも細部に誤りが残っているものがあり，それらを措置する必要がある．また，SSA 形式の分野で問題として議論されてはいるが実際にどう解決するかが示されていない種々の点がある．本実装では，それらにつき，次のようにそれぞれ 1 つの解決法を示した．

(1) Briggs の SSA 逆変換アルゴリズム[4] の実装上の注意の指摘と修正

(2) Sreedhar の SSA 逆変換アルゴリズム[26] の誤りの訂正と実装上の注意の指摘

(3) 積極的な最適化をどこまで許すかの決定

(4) 最適化アルゴリズムでの ϕ 関数の適切な取扱い

(1)，(2) は細部にわたるので，文献 27) に譲り，ここでは，(3)，(4) について無用コード除去を例にあげて説明する．

5.1 積極的な最適化の扱い

積極的な（aggressive）無用コード除去のアルゴリズムでは，プログラムの最終出力に影響を与えないコードは「死んでいる」ものとして除去する．

このようなアルゴリズムでは，出力を行わない無限ループは,「死んでいる」と見なされて消されてしまう．その結果のコードを実行すると，無限ループの次に書いてあるコードが実行され，場合によってはもともとは出なかった出力が行われてしまうことになる．たとえば，**図 6** (a) のようなプログラム断片は，無限ループが消されて図 6 (b) に変換されてしまう．実際，我々が既存の積極的な無用コード除去アルゴリズムをそのまま実装したところ，このような変換が行われた．

これでは，プログラムが持つ意味が変わってしまい，許容できない．Appel の本[1] の p.428 にも「積極的な無用コード除去は，出力のない無限ループを除去し，プログラムの意味を変えてしまう．多くの場合，これ

☆ 危険辺とは，複数の出力辺を持つ基本ブロックから複数の入力辺を持つ基本ブロックへの辺のことである．ある種の最適化では，危険辺があると文の挿入などが難しくなるため，危険辺を分割しておくことを前提とするアルゴリズムが採用されている．

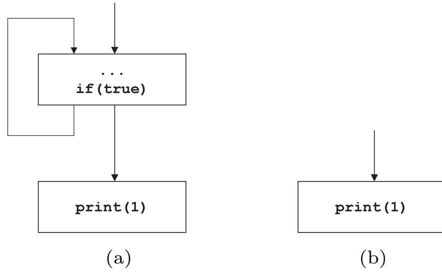


図 6 無限ループの扱い
Fig. 6 Handling infinite loop.

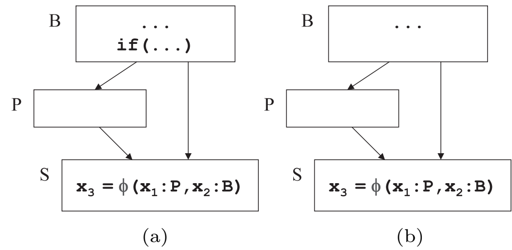


図 7 無用コード除去の危うい例
Fig. 7 Dangerous example of dead code elimination.

は許容できない.」とあり，上と同様な問題が指摘されている．そこではこの問題についての議論はあるが，解決法は示されていない．

それを避けるため，本実装では積極的な無用コード除去を行いつつ，ループから出るための条件分岐（図 6 (a) の `if(true)`）を「明らかに生きているコード」とした．これにより，図 6 (a) のような無限ループはそのまま保たれる．

なお，上記の積極的な無用コード除去の問題は，SSA 形式最適化に限らず，通常形式上の最適化でも起こる．

5.2 最適化アルゴリズムでの ϕ 関数の適切な取扱い

SSA 形式上での無用コード除去を行うアルゴリズムは，文献 1)，18)，19) に記述してあるものが一般的である．これらのアルゴリズムは，ϕ 関数を他のコードと同じに扱っている．また，これらのアルゴリズムでは制御フローを変更するようなコード列の変更も特定の条件のもとで許しており，その条件は最適化の対象となるプログラムが SSA 形式か否かで区別されていない．

しかし，そのアルゴリズムをそのまま適用すると，次のような最適化が行われてしまう．

図 7 (a) では，基本ブロック P が空であるので，既存のアルゴリズムでは，ブロック B の最後の条件文が除去され，図 7 (b) に変換される．しかし，図 7 (b) は，ブロック B の出口でどちらに分岐するのかが決まらず，ブロック S の ϕ 関数の意味が定まらないので正しくない．

そこで，本実現ではアルゴリズムを改訂し，ϕ 関数の引数と関連付けられているブロックが制御依存する条件分岐文を除去しないようにした．これにより，図 7 (b) への変換は行われない．

このようにすると，最適化が保守的すぎて無用コード除去の機会が減ってしまうと思うかもしれないが，そうはならない．たとえば，**図 8** (a) のコードは，ブロック S の ϕ 関数の引数がすべて等しい．このような

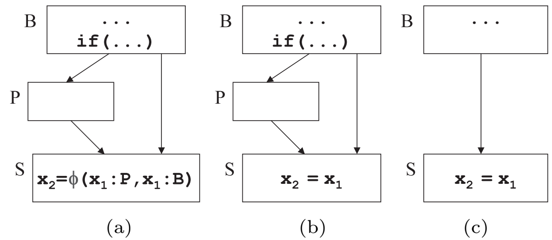


図 8 無用コード除去の望ましい例
Fig. 8 Good example of dead code elimination.

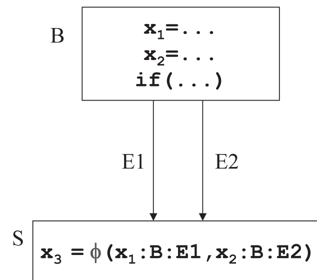


図 9 無用コード除去の積極的な適用結果
Fig. 9 Aggressive application of dead code elimination.

コードは，値番号付けなどを行った結果として得られることがある．このようなケースは次の手順で十分に最適化される．まず，無用 ϕ 関数除去の最適化によって，図 8 (b) のように ϕ 関数が通常の代入文に置き換えられる．その後無用コード除去により，ブロック P が除去され，ブロック B の条件文も消されて，図 8 (c) が得られる．

SSA 形式での無用コード除去アルゴリズムの改訂点は他にもあるが，省略する．要点は，SSA 形式特有の命令である ϕ 関数は他の命令と異なり，命令自身に制御フロー情報とデータフロー情報を含有しているので，SSA 形式上での無用コード除去では ϕ 関数を少し特別に扱う必要がある，ということである．詳しい説明とアルゴリズムは文献 27) に記した．

なお，図 7 (a) をさらに最適化していくと，**図 9** の

ような制御フローグラフが得られる，とする考え方もある．本実装では試験的に図 9 のような制御フローグラフも認めることとした．そのようなケースも扱えるよう，ϕ 関数の引数の出所には，図 9 の E1，E2 のように辺（edge）の情報もつけてある．ただし，この形からの SSA 逆変換は実装されていない．

6. 評　価

本実装はコンパイラ・インフラストラクチャとしてのものであるので，この章では SSA 最適化部の性能評価とともに，インフラストラクチャとしての機能についての評価も述べる．

6.1 SSA 最適化部の性能評価

SPEC ベンチマークおよび小規模テストプログラムについて，SSA 形式最適化を行った結果の目的コードの性能を測定した．

まず，SSA 最適化のお勧めセットを決めることにした．様々なコンパイル時オプションフラグを設けてあるので，それらを組み合わせることで，どのような最適化の組合せが良い性能を与えるかを測定した．測定は小規模テストプログラムと SPEC ベンチマークについて行ったが，後者の結果を**表 1** に示す．この結果，ssa1〜ssa3 のように多くの SSA 最適化のパスを適用すればするほど目的コードの性能が上がるというわけではないことが分かった．また，考えられる組合せのうち，この表の ssa4 が全体として良い結果を与えることが分かった．そこで，以後はこの ssa4 のオプションを SSA 最適化のお勧めとし，「-O2」と書くとこの最適化が行われるようにした．

また，この測定の過程で，多くのテストプログラムの目的コードを読んで解析したが，最内ループでの命令数が減っているのに実行時間が増えてしまうなどの現象が見られ，SPARC のようなスーパスカラマシンでは命令キャッシュやパイプライン処理などの計算機アーキテクチャの影響が大きいこと，が判明した．

次に，SPEC ベンチマークについての目的コードの実行時間を**表 2** に示す．これは COINS の基盤部に対して，HIR での最適化や他の最適化を適用せずに，SSA 最適化だけを適用した結果である（COINS 基盤部のバックエンドではデフォルトで局所的な最適化が行われているので，「COINS 基盤部最適化オプションなし」の場合でも分岐命令の最適化や基本ブロック内の定数畳み込みなどの簡単な最適化がされている）．

また，COINS 基盤部最適化オプションなしを 1 としたときの目的コードの実行時間の相対比を**図 10** に示す（図が煩雑になるので，ssa-preqp1 は除いた）．

表 1 SPEC CPU2000 C/F77 ベンチマークによる SSA 最適化器の組合せの評価（目的コードの実行時間）
SUN Fire V440（UltraSPARC III，1 GHz × 4，メモリ 8 GB）での結果．入力は test
以下は，ssa-opt のオプションとして次で指定される SSA 最適化を行ったもの
ssa1: prun/divex/cse/cstp/dce/hli/osr/hli/cpyp/cstp/cse/cpyp/cstp/dce/ebe/srd3
ssa2: prun/divex/cpyp/cse/cstp/dce/hli/cstp/osr/cse/cstp/cse/dce/cbb/ebe/srd3
ssa3: prun/gra/divex/cpyp/cse/cstp/dce/hli/cstp/osr/cse/cstp/cse/dce/cbb/ebe/srd3
ssa4: prun/divex/cse/cstp/hli/osr/hli/cstp/cse/dce/srd3

Table 1 Evaluation of combinations of SSA optimizers using SPEC CPU2000 C/F77 benchmarks.

benchmark	ssa1 (sec)	ssa2 (sec)	ssa3 (sec)	ssa4 (sec)
164.gzip	3.97	4.02	4.03	3.97
175.vpr	5.84	5.82	5.85	5.88
181.mcf	0.812	0.738	0.826	0.736
197.parser	6.73	6.77	6.82	6.72
254.gap	2.32	2.31	2.47	2.45
256.bzip2	26.6	26.3	26.6	26.3
300.twolf	0.623	0.622	0.645	0.623
171.swim	2.50	3.04	3.06	2.50
172.mgrid	108	123	123	107
173.applu	1.19	1.37	1.45	1.21
177.mesa	5.28	5.28	5.36	5.46
179.art	17.0	16.7	16.9	16.2
183.equake	3.13	3.32	3.17	3.34
188.ammp	24.0	22.4	22.6	23.4

表 2 や図 10 から以下のことがいえる．

まず，「メモリ全体を 1 つの塊と見なす素朴な別名解析」の効果が見て取れる．ssa-O2-no-memory-analysis は，前述の SSA の「-O2」最適化である ssa-O2 から 4.4.2 項で述べた「メモリ全体を 1 つの塊と見なす素朴な別名解析」だけをはずしたものである．この 2 つを比べると，「メモリ全体を 1 つの塊と見なす素朴な別名解析」は，2%から 12%程度の性能向上に寄与していることが分かる．2.1 節で述べたように，現状の COINS の別名解析は簡単なものが HIR で行われるだけで LIR では別名の情報は得られない．その制限の中では，「メモリ全体を 1 つの塊と見なす素朴な別名解析」は十分な効果をあげているといえる．

また，4.4.2 項で述べた「効率的な質問伝播を用いた大域的値番号付けによる部分冗長性除去」を含む ssa-preqp は，共通部分式除去までを行う前述の SSA の「-O2」最適化（ssa-O2）と比べて若干ではあるが目的コードの性能が上がっている（ssa-preqp の最適化には一部開発途中のものもある）．

SSA 最適化単体での結果は gcc や g77 の-O2 オプションの結果に一部は肉薄しているものの全体として

表 2 SPEC CPU2000 C/Fortran77 ベンチマークによる SSA 最適化部の評価（目的コードの実行時間）
SUN Blade 1000（UltraSPARC III, 750 MHz × 2, メモリ 1 GB）での結果．入力は ref
coins-noopt: COINS 基盤部最適化オプションなし
ssa-O2-no-memory-analysis: ssa-opt=prun/divex/cse/cstp/hli/osr/hli/cstp/cse/dce/srd3, ssa-no-memory-analysis で指定される SSA 最適化を行ったもの
ssa-O2: ssa-opt=prun/divex/cse/cstp/hli/osr/hli/cstp/cse/dce/srd3
ssa-preqp1: ssa-opt=prun/divex/cse/cstp/hli/osr/hli/cstp/preqp/dce/srd3
ssa-preqp: ssa-opt=prun/divex/cse/cstp/hli/osr/hli/cstp/cpyp/preqp/cstp/rpe/dce/srd3
gcc-O2: gcc -O2 または g77 -O2

Table 2 Evaluation of SSA optimizers using SPEC CPU2000 C/F77 benchmarks.

benchmark	coins-noopt (sec)	ssa-O2-no-memory-analysis (sec)	ssa-O2 (sec)	ssa-preqp1 (sec)	ssa-preqp (sec)	gcc-O2 (sec)
(CINT C)						
164.gzip	876	842	817	809	811	576
175.vpr	1075	905	780	771	773	538
181.mcf	499	475	469	475	465	438
197.parser	888	884	856	872	870	709
254.gap	776	747	705	700	696	591
255.vortex	821	839	788	769	778	589
256.bzip2	1064	934	887	885	882	479
300.twolf	1205	1165	1125	1117	1115	916
(CFP C)						
177.mesa	1126	1042	1020	1068	1004	840
179.art	735	682	626	628	627	447
183.equake	1120	1089	971	970	971	843
188.ammp	2061	1457	1436	1358	1434	992
(CFP F77)						
171.swim	7336	2234	2202	2211	2193	1968
172.mgrid	30446	3843	3338	3340	2480	2272
173.applu	20055	2939	2433	2700	1964	1600

は及ばない．ある意味で当然であると思うが，その原因として，gcc の目的コードではレジスタ・プロモーション☆や命令スケジューリングがなされているが，これはSSA 最適化だけでは処理できないこと，配列SSA などが提案されているものの SSA 最適化では非スカラ変数の扱いが確立されていないこと，現状の SSA 最適化に若干の改良の余地があること，があげられる．

COINS の全最適化を ON にして比べたいところであるが，COINS では高水準中間表現 HIR 上の最適化や別名解析，レジスタ・プロモーションや命令スケジューリングも行われているが，それらが試験実装の段階にあって SSA 最適化と組み合わせて SPEC ベンチマークの結果を得るに至っていない．

これらについては今後さらに検討する予定である．

☆ レジスタ・プロモーションとは，もともとメモリに置かれている変数をそれが使用される区間の間レジスタに乗せ，その区間が終わったらメモリに書き戻すことである．これにより，実行効率が上がる．もちろん，レジスタ割付けにより多くの局所変数はレジスタに乗せられるのだが，ここでは，それ以外の大域変数や配列の番地などを，それが使用される区間の間，レジスタに乗せておくことをいう．

ちなみに，SSA 形式最適化を行うコンパイラ・インフラストラクチャScale でも gcc ほどの性能は出ていない（7.2 節）．

なお前述のように，COINS ではレジスタ・プロモーションや命令スケジューリングを行うモジュールが試作されている．参考までにその効果を小規模なプログラムで計測したものを**図 11** に示す．マシンはSun Blade 1000 である．この図で「COINS O0」は COINS の最適化オプションなし,「ssa+S+R」は前述の ssa-O2 にさらに命令スケジューリングと大域変数のレジスタ・プロモーションの最適化も適用した結果である．図 11 の「ssa+S+R」では全体的に gcc の-O2 オプションより優れた性能を示している．

6.2 インフラとしての評価

6.2.1 SSA 最適化部の実装にあたっての問題点の解決

今回実現した SSA 最適化部のうち，基本的な最適化はすでに知られているものを実装したものが多い．しかし，既発表のアルゴリズムでも誤りがあったり，SSA 形式の分野で問題点が示唆されているが具体的

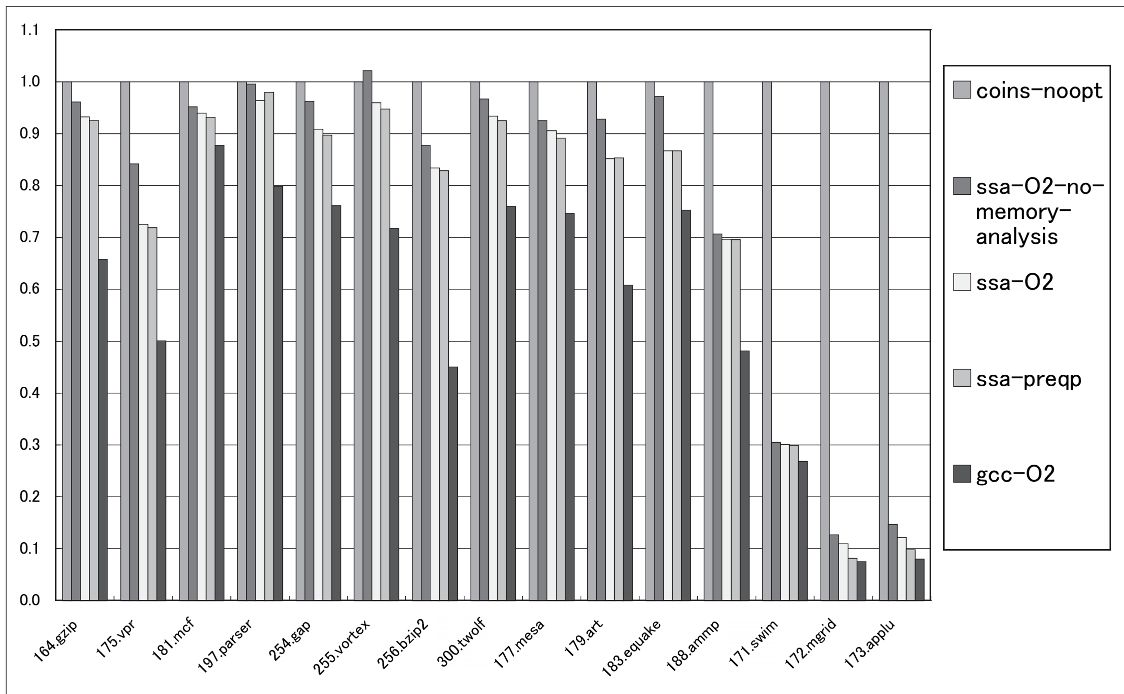


図 10 SPEC ベンチマークによる SSA 最適化器の評価(COINS 基盤部最適化オプションなしに対する目的コードの実行時間の相対比)
Fig. 10 Evaluation of SSA optimizer using SPEC benchmark (ratio of execution time of object code relative to COINS no optimization option).

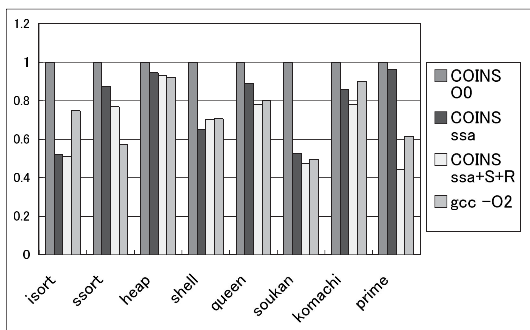


図 11 命令スケジューリングなどの適用結果
Fig. 11 Performance after instruction scheduling and others.

な解決が示されていないもの，SSA 形式に適用する際に注意が必要なもの，も少なからずあった．それらについて具体的な解決を与え，参照できる実装を行い，ドキュメントとして残したことは有益であったと思われる．とくに，

(1) SSA 変換・逆変換アルゴリズムの注意点の指摘と誤りの訂正
(2) 積極的な最適化をどこまで許すかの決定
(3) 最適化アルゴリズムでの ϕ 関数の適切な取扱いについて寄与できたと考える．

6.2.2 教育・研究への適用

COINS は，コンパイラの教育・研究の基盤とすることが目標の 1 つであった．SSA 最適化部については次のような実績がある．

(1) 講義への利用として，学部 4 年生と大学院生向けの COINS を用いたコンパイラの集中講義を開き，その一部として SSA 最適化の講義と演習を行った(2004 年 7 月)．このときに用いた豊富な例題とその使用法はウェブページで公開している[27]．また，学部教育レベルでは，東京工業大学の学士論文研究として，COINS インフラストラクチャを用いた研究がいくつかなされている．
(2) 研究レベルでは，前述した効率的な質問伝播を用いた部分冗長性除去がある．これは，もともとは COINS プロジェクト外のメンバが COINS を利用して独自のアルゴリズムの評価を始めたものであり，のちに，本システムに組み込むことになった．そのほか，東京工業大学の大学院生による研究も多い[14),17),28]．

なお，SSA 最適化部のソースプログラムのコメント，内部仕様書，外部仕様書，取扱いマニュアルは英

文化され，海外へのリリースの準備が整っている．

7. 関 連 研 究

一部のパスで SSA 形式を採用しているコンパイラはいくつかある[9),12),13),21)]．

しかし，本稿はコンパイラ・インフラストラクチャにおける SSA 形式最適化を主題としているので，ここではコンパイラ・インフラストラクチャを中心に関連研究を述べる．

7.1 SUIF と Machine SUIF

SUIF[29)] はコンパイラ・インフラストラクチャの中で最も良く知られていると思われる．SUIF は，最適化と並列化を協調させたコンパイラの研究を支援するために作られたフリーなインフラストラクチャで，スタンフォード大学を中心に作成された．米国の National Compiler Infrastructure（NCI）プロジェクトとして，DARPA と NSF の支援を受けた．

SUIF は，入力言語として，Fortran，C，C++，Java を受付ける．出力としては，C 言語，および Machine SUIF を経由して Alpha と x86 の機械語を生成できる．

SUIF の特徴として，拡張可能で高水準の表現と低水準の表現を混在できる中間表現を持つ，オブジェクト指向言語を扱うための中間表現がある，解析用インフラが整っている，Steensgaard の別名解析が組み込まれている，手続き間の配列解析や並列化解析がある，などがあげられる．しかし残念ながら SSA 形式についてはほとんど扱っていない．

また，SUIF は C++言語で書かれており，不要になったオブジェクトの回収にはやや困難があるといわれている．

SUIF はコンパイラの研究のテストベッドとして広く使われた．しかし，NCI の終了にともない，もはや保守がなされておらず，SUIF の一部を構成していた PGI 社も撤退している．

一方，Machine SUIF[16)] はコンパイラのバックエンドを作成するためのインフラストラクチャで，SUIF とともに用いる．これも NCI プロジェクトの 1 つであり，ハーバード大学で開発された．

Machine SUIF は，機械依存の最適化，プロファイルを用いた最適化，新しいアーキテクチャの評価，などに適し，制御フロー解析やデータフロー解析，レジスタ割当て，命令スケジューリング，スカラ最適化，コード配置などを含んでいる．ターゲット機械をパラメタ化することで，ターゲットにあまり依存しない解析と最適化ができる，という特徴がある．

しかし，SSA 形式の最適化については無用コード除去の最適化があるだけである．リリースも 2002 年以後は出ていない．

7.2 Scale

Scale[24)] は，新しい高性能アーキテクチャの研究のためにマサチューセッツ大学で開発中のコンパイラ・インフラストラクチャである．

Scale は，入力言語として，Fortran，C，Java を受け付け，出力としては，C 言語あるいは Alpha または SPARC v8 のアセンブリ言語のコードを生成する．

Steensgaard の別名解析，配列の依存テスト，一連のループ変換，SSA 最適化，多くのスカラ最適化を実現している．Scale は主に Java で書かれている．

SPEC ベンチマークの C プログラムから SPARC 機械語への変換で，オブジェクトコードの実行性能は gcc とほぼ同等から 1.7 倍（の遅さ）となっている．

SSA 最適化としては，疎な条件分岐を考慮した定数伝播（sparse conditional constant propagation），部分冗長性除去，コピー伝播，値番号付け，ループ不変コード移動，インライン化，ループ展開，を実装している．

我々が COINS の SSA 最適化部を開発中の数年前にサーベイしたときは，Scale はまだ C 言語への変換しか実現していなかった．しかし，現在では，Scale はアセンブリ言語も生成しており，Java で記述されていること，SSA 形式最適化を含んでいること，など COINS との類似性がある．ただし，COINS の特徴である，SMP 計算機用並列化や SIMD 命令を用いる命令レベル並列化などの並列化機能や，マシン記述によるリターゲット機能はない．

7.3 gcc

gcc[10)] では，数年来 Tree SSA と呼ばれる SSA 形式最適化の導入に向けての試みが別ブランチで行われていた．その時点では，SSA 形式最適化は実験的であると書かれていた．

その後，Tree SSA のブランチは 2004 年 5 月に本体にマージされ，2005 年 4 月にリリースされた gcc-4.0.0 にも含まれることとなった．

GCC Summit 2004 など[11)] によると，Tree SSA は，GIMPLE という 3 番地コードを基にした木構造中間表現を用いた SSA 形式最適化モジュールで，最適化として，無用コード除去，条件分岐を考慮した定数伝播，部分冗長性除去，支配木ベースの最適化（定数伝播，コピー伝播，冗長性除去など），集合体のスカラ置換（scalar replacement of aggregates），無用ストアの除去，などを含んでいる．

この最適化の内容は，COINS の SSA 最適化部に重なるものも多い．

しかし gcc については，ソースが大きく複雑で手を入れにくい，C 言語で書かれているので開発の際のバグ取りが難しくメモリ割当てや削除に誤りが入りやすい，RTL をファイルに書いて処理して再入力する手段がない，などの弱点が指摘されている．

これに対して，COINS の SSA 最適化部は，Java で記述されており，メモリ割当ての誤りがほとんどなくバグ取りがしやすい，ソースの大きさが手頃である，LIR をファイルに書いて処理して再入力することができる，などの利点がある．

7.4 ま と め

他のコンパイラ・インフラストラクチャと比べると，COINS やその SSA 最適化部は次の特徴がある．

- ソース全体が公開され，開発グループによって保守されている．
- Java 言語で書かれているので，C 言語や C++言語で書かれたコンパイラ・インフラストラクチャと比べて安全である．
- SSA 形式最適化が対象とする低水準中間表現 LIR は，ファイルへ出力して処理したものを再度入力できる，マシン記述により LIR からのコード生成系が自動生成できる，表示的意味論によりその意味が形式的に定義されている，などの特長を持つ．

8. お わ り に

本稿では，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（SSA 形式）最適化の実現について述べた．設計方針を立てて実施したことは，目標の実現に大きく寄与したと考える．

SSA 形式への変換，逆変換を共通インフラストラクチャに組み込むにあたっては，いくつかの方法を比較評価して，良いものを採用した．

SSA 形式はある程度普及してきたと思われるが，実装にあたっては，種々の解決すべき問題があった．本実現では，それぞれに 1 つの解決を与え，詳しいドキュメントとして残した．また，参照実装として用いることができるように，読解性，保守性の良いコーディングを心がけ，詳しい仕様書を提供した．

関連研究として，SSA 形式最適化を組み込んだコンパイラ・インフラストラクチャはいくつかあるが，教育，研究用基盤として使えるものは少ない．本実現では，アルゴリズムの比較評価のためにユーザが SSA 形式の変換や最適化の種々のバリエーションを用いたり，新しいモジュールを追加したりして，実験・評価を行えるよう，十分な配慮を行った．基本的な SSA 形式最適化もほとんど実装してある．

そこで，これをベースにした新たな研究が期待できる．実際，滝本ら[30]，須藤ら[28]，溝渕ら[17] の研究は，本研究により作成された SSA 形式最適化部を用いて，それに新たなアルゴリズムや手法を追加実装して評価したものである．

今後は，現状の COINS の最適化の効果が gcc などにまだ及ばない点を検討し，他の最適化と連携しつつ SSA 最適化の改良を施したい．

謝辞 COINS プロジェクトに協力してくださった方々に感謝する．本研究の一部は，文部科学省科学技術振興調整費，日本学術振興会科学研究費補助金，財団法人栢森情報科学振興財団の補助を受けた．

参 考 文 献

1) Appel, A.W.: *Modern Compiler Implementation in Java*, 2nd edition, Cambridge University Press (2002).

2) Briggs, P. and Cooper, K.D.: Effective Partial Redundancy Elimination, *Proc. Conference on Programming Language Design and Implementation*, pp.159–170 (1994).

3) Briggs, P., Harvey, T.J. and Simpson, L.T.: Static Single Assignment Construction, Version 1.0. (1996). ftp://ftp.cs.rice.edu/public/compilers/ai/SSA.ps

4) 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).

5) Chaitin, G.J.: Register Allocation and Spilling via Graph Coloring, *Proc. ACM SIGPLAN '82 Symp. on Compiler Construction, SIGPLAN Notices*, Vol.17, No.6, pp.98–105 (1982).

6) COINS Project：並列化コンパイラ向け共通インフラストラクチャCOINS ホームページ. http://www.coins-project.org/

7) Cooper, K.D., Simpson, L.T. and Vick, C.A.: Operator Strength Reduction, *ACM Trans. Prog. Lang. Syst.*, Vol.23, No.5, pp.603–625 (2001).

8) 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).

9) Fitzgerald, R., Knoblock, T.B., Ruf, E., Steensgaard, B. and Tarditi, D.: Marmot: An

Optimizing Compiler for Java, *Software — Practice and Experience*, Vol.30, No.3, pp.199–232 (2000).

10) GCC: GCC Homepage. http://gcc.gnu.org/

11) GCC: GCC Summit. http://www.gccsummit.org/2004/, http://www.gccsummit.org/2005/

12) IBM: Jikes Research Virtual Machine. http://jikesrvm.sourceforge.net/

13) Ishizaki, K., Takeuchi, M., et al.: Effectiveness of Cross-platform Optimizations for a Java Just-in-time Compiler, *OOPSLA '03: Proc. 18th annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages and Applications*, pp.187–204, ACM Press (2003).

14) 伊藤　陽，小濱真樹，佐々政孝：静的単一代入形式からの逆変換アルゴリズムの比較と評価，情報処理学会論文誌：プログラミング，Vol.46, No.SIG 14(PRO 27), pp.30–42 (2005).

15) 小濱真樹，中谷俊晴，佐々政孝：静的単一代入形式における正規化アルゴリズムの比較，日本ソフトウェア科学会大会論文集，第 19 回，1C-1 (2002).

16) Machine SUIF Homepage. http://www.eecs.harvard.edu/machsuif/

17) 溝渕裕司，立川　英，佐々政孝：変更文の移動を可能にした静的単一代入形式上での部分冗長性除去，日本ソフトウェア科学会第 7 回プログラミングおよびプログラミング言語ワークショップ（PPL2005）論文集，pp.261–275 (2005).

18) Morgan, R.: *Building an Optimizing Compiler*, Digital Press (1998).

19) 中田育男： コンパイラの構成と最適化，朝倉書店 (1999).

20) 中谷俊晴，加藤吉之介，佐々政孝，脇田　建：コンパイラ・インフラストラクチャにおける SSA 形式最適化プロトタイプシステムの実装，日本ソフトウェア科学会大会論文集，第 18 回，3D-2 (2001).

21) Paleczny, M., Vick, C. and Click, C.: The Java HotSpot Server Compiler, *USENIX 1st Java Virtual Machine Research and Technology Symposium* (*JVM '01*), pp.1–12 (2001).

22) Sassa, M., Nakaya, T., Kohama, M., Fukuoka, T. and Takahashi, M.: Static Single Assignment Form in the COINS Compiler Infrastructure, *Proc. SSGRR 2003w*, L'Aquila, Italy, No.54 (2003).

23) Sassa, M., Kohama, M. and Ito, Y.: Comparison and Evaluation of Back Translation Algorithms for Static Single Assignment Form, *Proc. IPSI-2004*, Prague, Czech, ISBN: 86-7466-117-3 (2004).

24) Scale Compiler Group: Scale homepage. http://www-ali.cs.umass.edu/Scale/

25) Sreedhar, V.C. and Gao, G.R.: A Linear Time Algorithm for Placing ϕ-Nodes, *Proc. 22nd ACM POPL*, pp.62–73 (1995).

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

27) 静的単一代入形式最適化システム 解説と外部仕様書．http://www.is.titech.ac.jp/˜sassa/coins-www-ssa/japanese/ あるいは http://www.coins-project.org/からリンクをたどる．

28) 須藤大二朗，佐々政孝： 比較照合法によるコンパイラ最適化器の正しさの検証，日本ソフトウェア科学会第 7 回プログラミングおよびプログラミング言語ワークショップ（PPL2005）論文集，pp.231–245 (2005).

29) SUIF. SUIF Homepage. http://www-suif.stanford.edu/

30) 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(PRO26), pp.16–26 (2005).

31) Wegman, M.N. and Zadeck, F.K.: Constant Propagation with Conditional Branches, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.4, pp.181–210 (1991).

(平成 17 年 7 月 5 日受付)
(平成 17 年 11 月 9 日採録)

佐々　政孝（正会員）

1948 年生．1970 年東京大学理学部物理学科卒業．1974 年同大学院博士課程中退，東京工業大学理学部情報科学科助手．1981 年筑波大学電子・情報工学系．1992 年東京工業大学理学部．現在同大学大学院情報理工学研究科数理・計算科学専攻教授．理学博士．プログラミング言語，コンパイラ，プログラミング環境に興味を持つ．著書『プログラミング言語処理系』（岩波書店）．日本ソフトウェア科学会，ACM，IEEE 各会員．

福岡　岳穂（正会員）

2001 年電気通信大学大学院情報システム学研究科情報ネットワーク学専攻修了．現在，(株) ソニー・コンピュータエンタテインメント．プログラミング言語およびその処理系，オペレーティングシステムに興味を持つ．日本ソフトウェア科学会各会員．

滝本　宗宏（正会員）

1994 年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了．1999 年東京理科大学理工学部情報科学科助手．現在，東京理科大学理工学部情報科学科講師．工学博士．プログラミング言語およびその処理系に興味を持つ．ACM，IEEE，日本ソフトウェア科学会各会員．