LA-005

移植可能な Superoptimizer による最適な命令パターンの自動生成とそのパターンによる覗き穴最適化

Automatic Generation of Optimized Instruction Patterns using a Portable Superoptimizer and Peephole Optimization using their Patterns

蒲野 茂幸†‡
Shigeyuki Kabano

佐々木 晃†
Akira Sasaki

1．序論

コンパイラの最適化は、目的機械のアーキテクチャに独立な中間表現の上で行われることが多い。しかし、中間表現のコードが最適であったとしても、それを目的機械の命令列に書き換えると、冗長な命令が出てきてしまう。それは、中間表現の数命令に対応した命令 (SIMD 命令など)や機械特有の操作がうまく使えないことにより起こる。そこで、生成された目的機械の数命令の命令列に対してパターンマッチングを行い、効率のよい命令に置き換える覗き穴最適化が有効である。しかし、パターンマッチングに使うパターンを作成するには、知識や経験が必要になる。

本研究では、このパターンを自動生成する手法を考案し、COINS コンパイラ・インフラストラクチャ [1]上に組み込み性能を検証した。このパターンの自動生成には２通りの方法がある。

１つは、命令合成による方法[2]である。命令合成方式は、隣り合う２命令を合成して１命令として、その命令を実行する命令があるかどうか目的機械の命令セットから探し出すことによりパターンを生成する。この方法を、COINS コンパイラ・インフラストラクチャ上に組み込んでプロトタイプ実装した例がある[3]。

もう１つの方法が、 superoptimization[4]による方法である。superoptimization とは、命令列を実行した結果が同じ命令列を探す方法である。途中の命令実行工程には依存しないため、命令合成方式で起こる最適化パターンの偏りがないという利点がある。 superoptimizer の一例である GSO (Gnu SuperOptimizer)は、gcc の覗き穴最適化のパターンの発見に用いられている。[5]

本研究では、後者の superoptimizer を用い、パターンを自動生成する Bansal らの手法[6]を参考にした実装を行っている。Bansal らの手法との相違点として、命令列の探索を機械語レベルの中間表現上で行う点が挙げられる。これにより、目的機械の命令列を中間表現に変換し、生成した中間表現の命令列をアセンブリ言語の命令列に 変換する工程を追加するだけで多くの機械 に移植することができるという利点がある。本研究ではこの方法を COINS コンパイラ・インフラストラクチャ上に組み込んで実験し、いくつかの有用な置き換えパターンを発見することに成功し、そのパターン置き換えによる性能の改善も示せた。

2．superoptimizer と覗き穴最適化

superoptimization とは、与えられた命令列（目的命令列）と同じ効果が得られる最適命令列を、目的機械の命令セットのすべての命令をコストが少ない命令から 順に全探索することにより見つける手法である。すなわち、最適化したい命令列を目的命令列として superoptimizer の入力として与えることで、自動的により効率の良い命令列を求められる。superoptimizer の一実装として、前述の GSO がある。GSO では、入力として命令列ではなく関数（目的関数）をとる。目的関数とは、例えば if(a>b)a++;のようなもので、これを実行して同等の結果を与える最適命令列をアセンブリ言語で出力する。GSO ではこの目的関数が数十個用意されていて、手動で追加できるようになっている。ただし、GSO で生成したパターンを GCC で使うためには、GCC のコード生成部にこのパターンを置き換える記述を書かなければならない。

覗き穴最適化とは、コードを局所的に見て、１つの目的命令列をそれと同等のより 効率の良い最適命令列と置き換えるパターンマッチングシステムである。通常、パターンマッチのルールは人間が記述しなければならないが、手間や経験が必要となる。そこで、 Bansal らは、superoptimization を用いて目的命令列の生成から、 それに対する最適命令列の生成を行い、さらにそのパターンを使った覗き穴最適化までを自動で行う手法を提案した。しかし、この手法では様々なアーキテクチャに対応させることが困難である。特に、アーキテクチャごとに別々の superoptimizer を用意する必要が生ずる。

本研究では、Bansal らの手法を参考に、より移植性の高い、パターン自動生成を行う覗き穴 最適化器を設計した。superoptimization については、GSO を使うことにより移植性を高めた。

3．移植可能な 自動パターン生成を行う覗き穴最適化器の設計

本節では、提案する自動パターン生成の手法とそのパターンを用いる覗き穴最適化器の設計について述べる。図１には、最適化器全体のフローチャートを示した。全体の構成は、Bansal らの手法[6] を参考にしている。図の太線の部分はパターン生成のフローで、細線の部分は覗き穴最適化のフローである。

† 法政大学 情報科学部

‡ 現在は東京工業大学 情報理工学研究科

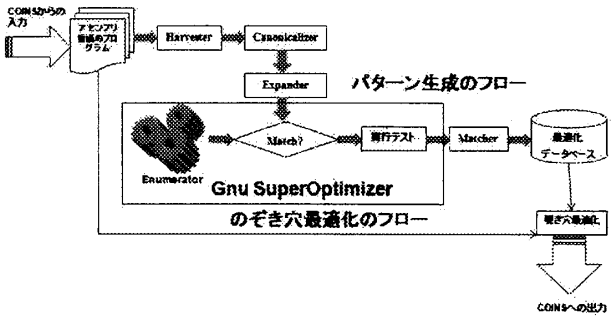
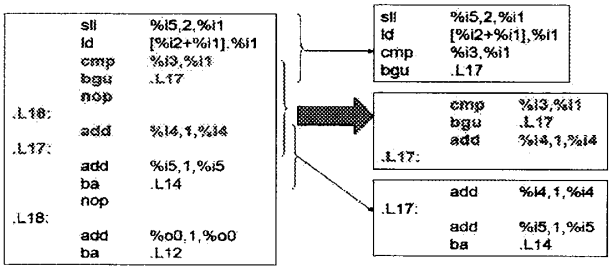


図１．移植可能な自動パターン生成を行う覗き穴最適化器

3.1 自動パターン生成を行う覗き穴最適化器

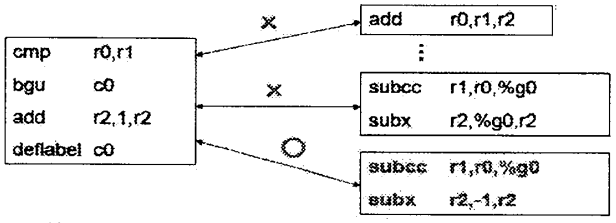
3.1.1 命令の切り出し(harvester)



Harvester は、プログラムから命令列を切り出す。これが、覗き穴最適化の目的命令列の候補となる。ここでは、命令の切り出し方の説明を上記のSPARC の命令列の例を使って行う。まず、superoptimizer は命令の実行結果のみを必要とするので、実行結果に影響を与えないnop や、不要なラベル.L16 を除去する。ブランチ命令を除去することを目標としているので、ここでのnop命令は簡単に除去する。上記の例では、右中段の命令列がcmp 命令から４命令を切り取った結果である。

Canonicalizer は、オペランドとなるレジスタや記号定数を一貫性のある名前に置き換えることで、命令列のパターンを減らす。この正規化した命令列が、覗き穴最適化での目的命令列の候補(superoptimizer が探索する目的命令列) となる。新しいレジスタが出現したときは、r0, r1.. のように昇順に置き、記号定数やアドレスの場合も同様に c0, c1.. のように置く。

3.1.2 superoptimization

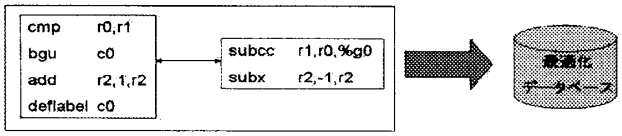


２節で説明したsuperoptimizationにより、最適な命令列を発見できる。これを実装するには、図１のEnumeratorと実行テストが必要になる。

Enumerator は、目的機械の命令列をコストが少ない順に出力する。Match?の処理は、１つのテストデータを使用して、生成された命令列と、目的命令列と実行結果がマッチするかどうかを判断する。これでマッチした命令列が、最適命令列の候補となる。

実行テストでは、目的命令列と最適命令列の候補が同等の出力レジスタと状態コードの結果を出力するかどうか を、入力レジスタに値を入力して実行することによりテストする。この値は、特別な値(すべて0bitやすべて1bit)とランダムな値となっている。

3.1.3 最適化データベースへの登録



最適化データベースに、目的命令列と最適命令列のパターンの対（の集合）を保存する。これにより、多くのプログラムをsuperoptimizer に通すことにより、パターンの数を増やすことができる。また、それとは別にsuperoptimizerで最適な命令列を得られなかった目的命令列も保存する。これにより、以前の実行で最適命令列が見つからなかった目的命令列は、探索を省略できる。

3.1.4 覗き穴最適化

覗き穴最適化では、データベースのパターンを使用して覗き穴最適化を実行する。パターンマッチングの際に、生存情報が必要になることがある。たとえば、{mov r0, r1; mov r1, r2} の目的命令列を{mov r0, r2} の最適命令列に置き換えることができるためにはレジスタr1 が死んでいることを確認しなければならない。

3.2 中間表現の導入(Expander,Matcher)

3.1 節では、パターン生成を SPARC の命令セットを使って説明した。しかし、他のアーキテクチャでパターン生成を行うには新たにその機械向けのパターン生成器を作らなければならない。そこで、本研究では図２のように機械語レベルの中間表現を導入することによって、目的機械に非依存である中間表現上でパターン生成を行い、そのパターンを変換することにより、SPARC のパターンを生成した。これによって、新しい x86 のような機械に対応させる場合、中間表現との対応関係を書くだけで容易に移植できる。

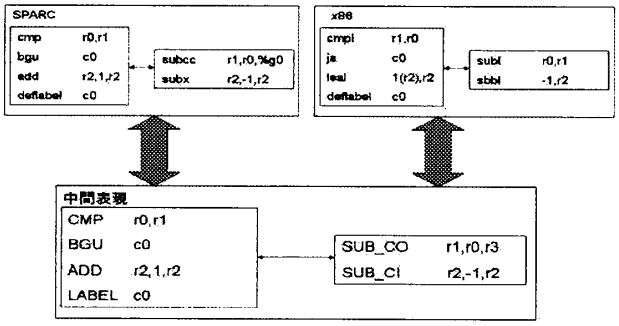


図２．中間表現との対応関係

本研究では、中間表現は GSO のものを改良して使用しており、表１のように SPARC や X86 などのアセンブリ言語と中間表現が対応している。

表１．中間表現

中間表現	SPARC	x86
ADD s1,s2,d	add s1,s2,d	incl d (if s2 = 1) decl d (if s2 = -1)
ADD_CI s1,s2,d	addx s1,s2,d	
ADD_CO s1,s2,d	addcc s1,s2,d	addl s2,d
ADD_CIO s1,s2,d	addxcc s1,s2,d	adcl s2,d

本研究で提案する覗き穴最適化器では、中間表現を導入するため Bansal らの手法に新たに、Expander と Matcher を

追加した。

Expander は、アセンブリ言語の命令列を中間表現の命令列に書き換える。Matcher では、中間表現の命令列をアセンブリ言語の命令列に書き換える。このExpander から Matcher までのsuperoptimizerの操作は、中間表現で行われる。これにより、新しい目的機械に対応した覗き穴最適化器を作成する場合、新しい機械と中間表現の変換規則（機械語→中間表現、中間表現→機械語）を記述し、これをExpanderとMatcherに適応することにより、容易に行える。

また、Enumeratorでのコスト計算は、中間表現の命令数で与えている。そこで、中間表現は、コストが低く、等しい命令だけに実装を制限することにした。例えば、足し算や引き算などは実装を行ったが、掛け算や割り算は実装をしないことにした。しかし、足し算や引き算などと比べてコストの大きいメモリアクセス命令の実装を行った。これによって、4.2.1節の1での冗長なロード命令除去のパターンは、コストが低い命令列を発見したことにはならなくなる。これに対応するために、同じ命令数の命令列を発見したときでも、それをパターンとしてデータベースに保存するようにした。

中間表現の各命令に対しては意味動作が定義されるが、これは状態コードの変化を適切に表したものでなければならない。本研究で基にしたGSOでは、状態コードはキャリーのみデフォルトでサポートされている。しかし、ブランチ命令を最適化の対象とするために、他の状態コード(0判定など) も追加した。

4．実装と実験

4.1 COINS への実装

本節では、3節で示した手法のプロトタイプ実装について述べる。この実装は、目的機械をSPARC アーキテクチャとし、COINS コンパイラインフラストラクチャ[1] 上に組み込む形で行った。COINS は、新しいコンパイラ方式を容易に実験、評価できるような共通インフラストラクチャで、全体の構成は図 3 のようになっている。COINS では、マシン記述部(図 3 右下) に覗き穴最適化のための関数(peepHoleOpt) を記述できるようになっている。そこで、COINS へ本研究の覗き穴最適化器を組み込むために、peepHoleOpt では、パターン生成を行う関数と、そのパターンで覗き穴最適化を行う関数を呼び出している。この2つの関数は、他の機械への移植も考えて、機械依存の部分をできるだけ少なく、わかりやすく記述した。

4.1.1 パターン生成を行う関数

ここでは、superoptimizer では対象とはしない遅延命令などを除いた命令列を、目的機械を指定したsuperoptimizerの入力にして、superoptimizer にパターンを作成させる。superoptimizer の内部は、3節で説明した。

4.1.2 覗き穴最適化を行う関数

この関数では、3節で説明したようにデータベースのパターンを使用して覗き穴最適化を実行する。覗き穴最適化では、生存情報が必要になる。通常は生存情報を解析するには、制御フローグラフを作らなければならないが、ここではCOINS のレジスタ割付で使用する生存情報を用いる。

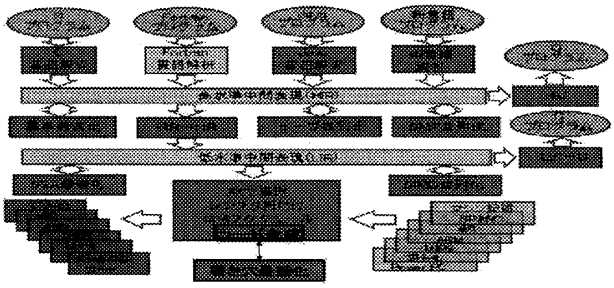


図 3．COINSへの組み込み

4.2 実験

本節では、4.1 節で述べたSPARC 用覗き穴最適化器を用いて発見されたパターンについて述べる。また、覗き穴最適化による効果を調べるため、カーネルプログラムでの実行時間の改善を測定した。

4.2.1 発見されたパターン

発見されたパターンは、現段階で下記の3つの種類に分けられる。それぞれの種類では、他のブランチ命令を使ったパターンや、3番目のブランチ命令の除去での加算(add r2,1,r3) を減算(sub r2,1,r3) に変えたようなパターンも発見された。

1. 冗長なロード命令の除去

```
st      r0, c0          st      r0,c0
ld      c0,r1     ⇒     mov     r0,r1
```

ロード命令をムーブ命令に置き換えることによりメモリアクセスが発生しないので、命令実行時間が改善される。

2. ブランチ命令を条件ムーブ命令に置き換える(10パターン発見)

```
cmp     r0,0            cmp     r0,0
be      c0        ⇒     movne   %icc,1,r1
mov     1,r1
deflabel c0
```

条件ムーブ命令を使うことにより、ブランチ命令を除去できる。これにより、コードサイズを1命令減らすことができるので、実行時間の改善もされる。

3. ブランチ命令の除去(14パターン発見)

```
cmp     r0,r1           subcc   r1,r0,%g0
bgu     c0        ⇒     subx    r2,-1,r3
add     r2,1,r3
deflabel c0
```

キャリーをうまく使うことにより、コードサイズを1命令減らしている。また、このようなパターンは命令合成では発見できない。

4.2.2 プログラムによる実験結果

実験は、整数要素の配列計算を行うプログラムを使い、発見されたパターンによる覗き穴最適化の実行時間の改善の効果を実証するために行った。テストプログラムは、表2に示す。COINSで-O2 のオプションで最適化された命令列を覗き穴最適化器の入力として使用する。図4に本研究の覗き穴最適化を使用しない場合と使用した場合とのカーネルプログラムにおいての実行時間の比較を示す。

表2: 4百万個の要素の配列で実験に使用したカーネルプログラム

カーネル名	擬似コード
leu+	*if* (*a*[*i*] <= *b*[*i*]) *d*++;
ne0-	*if* (*a*[*i*] ! = 0) *d*−−;

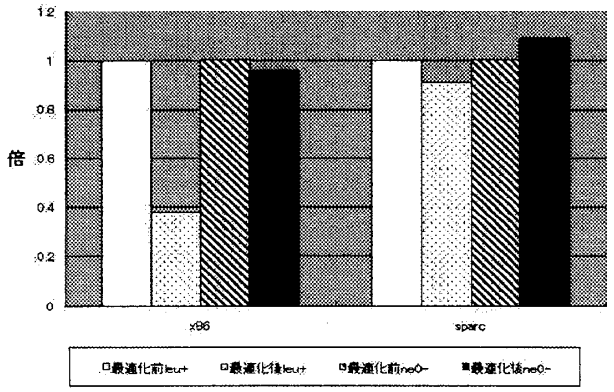


図4：カーネルプログラムの実行時間

ここで、SPARC上での実験は、本研究で実装した覗き穴最適化器を適用した結果である。x86の実験結果は、GSOによる結果のパターンを、COINSでコンパイルしたx86のコードに、手動で組み込んで得られたもので、SPARCの実験結果との比較のために行った。なお、SPARCの実験環境は、UltraSPARC II 400MHz、メモリ512MBである。また、SPARCとの比較には、AMD Athlon 64 3000+ 1.81GHz、メモリ512MBというx86マシンを使用した。4.2.1節で示した3番目のパターンである*leu+*において、使用しない場合と比べて、x86では62%、SPARCでは9%ほどの性能の改善が見られた。しかし、同じ3番目のパターンである*ne0-*においては、x86では4%しか改善されず、SPARCにおいては、9%実行時間が増加した。

5．考察

本研究では、移植性を考慮した、命令パターンの自動生成および、そのパターンを用いる覗き穴最適化の工程を設計した。また、プロトタイプ実装としてSPARCを目的機械として用い、COINSコンパイラに実装した。

本研究の特徴は、パターン生成のharvester以降のすべての工程を機械語レベルの中間表現で行った点である。これにより、表1に示したような目的機械の機械語と中間表現との対応関係を書くことにより、中間表現上でパターン生成を行うことができる。よって、新たにパターン生成器を作成せずに済むという利点がある。

生成されたパターンの特徴として、superoptimizerの特性上、即値オペランドを使用したパターン生成は少ないが、多くのブランチ命令を除去するパターンがあった。また、そのパターンは、カーネルプログラムにおいて実行時間を改善することが実験より実証された。しかし、SPARCにおいては、x86より性能改善の割合が少なく、実行時間が増加してしまうパターンも現れた。原因は不明であるが、パターンを置き換える際に、さらに計算コストの考慮を行う必要がある。また、SPEC CINT2000での実験では性能改善の効果はなかった。

今後の課題として、さらに性能改善を図るために、SIMD命令を中間表現に追加することが考えられる。SIMD命令は、SIMD型のマルチメディア向けの拡張命令であり、コンパイラの対応は不十分である場合がある。しかし、SIMD命令をうまく活用することにより、高速化が期待できるので、COINSでの研究が行われている。[8]

また、実行テストにおいて、今まで間違った命令列は出力されていないが、包括的なテストではないので、正しさを証明するためには高信頼のテストが必要である。

また、COINSへの組み込みにおいて、機械依存部とそうでない部分を完全に分離する(違う関数にする)ことにより、さらに移植しやすいようにすることも必要である。

6．結論

本研究では、移植性の高い覗き穴最適化器の実装手法の提案を行った。この手法では、命令パターンを自動生成し、そのパターンに基づいた覗き穴最適化を行う。本研究のプロトタイプ実装では、いくつかの有効な最適化パターンを発見でき、そのパターンでの命令置き換えである程度の実行時間の改善が見られた。今後、5節で述べたように、実際にコンパイラに組み込むために、superoptimizerのSIMD命令への対応や、覗き穴最適化を移植可能にすることなどについて改善が必要である。さらに、包括テストを併用する等の方法で命令パターンの正確性を保証することが、実用化するために最も重要なことである。

謝辞

本研究の一部は科研費(課題番19700031)の助成を受けたものである。東京工業大学大学院情報理工学研究科の佐々政孝教授と査読者からは、貴重なコメントを頂いた。ここに謝辞を申し上げる。

参考文献

[1] COINSProject. Coinsproject home page. http://www.coins-project.org/.
[2] Jack W. Davidson and Christopher W. Fraser,"Automatic generation of peephole optimizations," SIGPLAN Not., Vol. 39, No. 4, pp. 104-111, 2004.
[3] 佐原聡一郎、佐々政孝、"自動的な命令合併を行う覗き穴最適化器の設計と実装、" 日本ソフトウェア科学会大会論文集、第22回、5B-2、2005年9月.
[4] H. Massalin, "Superoptimizer: A look at the smallest program," In Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), pages 122-126, 1987.
[5] T. Granlund and R. Kenner, "Eliminating branches using a superoptimizer and the gnu C compiler," In Proceedings of the ACM SIGPLAN `92 Conference on Programming Language Design and Implementation, volume 27, pages 341-352, San Francisco, CA, June 1992.
[6] Sorav Bansal and Alex Aiken, "Automatic Generation of Peephole Superoptimizers," In Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems, pages 394-403, October 2006.
[7] Keith D. Cooper and Linda Torczon, Engineering a Compiler, Morgan Kaufmann Publisher, 2003.
[8] 藤波順久、阿部正佳、"SIMD型拡張命令をもっと使った最適化への道のり," 第43回プログラミング・シンポジウム報告集、2002年1月.
[9] SPARC International, Inc. "The SPARC Architecture Manual Version 9," http://www.sparc.com/standards/V9-R1.4.7.pdf
[10] 中田育男, コンパイラの構成と最適化, 朝倉書店, 1999.