CPU のセキュリティ対策を可能にする Reduced Assembly Set Compiler の提案

坂井 弘亮[1]

概要:組込みシステムにおいては様々な CPU アーキテクチャが用いられる．一方,ソフトウェアの脆弱性対策として, CPU やコンパイラによるセキュリティ対策がある.しかし従来のコンパイラの多くでは,新たな CPU アーキテクチャへの対応は困難なものとなっており,それらに対応したコンパイラの開発が障壁となる.
そこで生成するアセンブリのパターンを限定することで新たな CPU アーキテクチャへの対応を容易にした Reduced Assembly Set Compiler (RASC) を提案する.RASC の実装であるCコンパイラ NLCC では,64 種類のアセンブリ・パターンを登録するだけで新たな CPU アーキテクチャに対応可能であり,さらに 28 種類のパターンはビルトインにより省略可能にすることができた.また x86 や ARM を含む 12 種類の CPU アーキテクチャに対応させた結果,平均8時間程度で対応できた.よって1日で新たな CPU アーキテクチャに対応できる 1day-compiler としての利用が可能である.

キーワード:コンパイラ, アセンブリ

Proposal of Reduced Assembly Set Compiler enabling security measures of the CPU

Hiroaki Sakai[1]

Abstract: Various CPU architecture is used in the embedded system. On the other hand, one of the vulnerability measures of the software includes the security measures with the CPU or the compiler. However, with the most of conventional compilers, the correspondence to the new CPU architecture becomes difficult, and the development of the compiler corresponding to them becomes the wall.
Therefore I suggest Reduced Assembly Set Compiler (RASC) which facilitated correspondence to the new CPU architecture by limiting the pattern of the formed assembly. In C compiler NLCC which was implementation of RASC, only 64 kinds of assembly patterns are registered and it was available for the new CPU architecture and 28 kinds of patterns are omissible by a built-in. In addition, as a result of corresponding to 12 kinds of CPU architecture including x86 and ARM, I was able to cope in an average of around eight hours. Thus, the use as 1day-compiler which can support the new CPU architecture in a day is possible.

Keywords: Compiler, Assembly

1. 背景

組込みシステムにおいては様々な CPU アーキテクチャが用いられる．一方,ソフトウェアの脆弱性対策のひとつとして,CPU によるセキュリティ対策がある.これにはセキュリティのための専用命令の追加やセキュリティに配慮した新たな CPU アーキテクチャの設計・開発が考えられるが,それらに対応したコンパイラの開発が障壁となる.またスタック・プロテクタに代表されるコンパイラのセキュリティ機構はソースコードを修正せずに適用できる点で有効であるが,新たなセキュリティ機構を検証・導入する際にはコンパイラの改造が前提となる.

これらのためには新たな CPU アーキテクチャに迅速に対応できるコンパイラが求められる.しかし従来のコンパイラの多くは高度・複雑な最適化による高速性を重視しており新たな CPU アーキテクチャへの対応は困難なものとなっている.

そこで生成するアセンブリのパターンを限定することで新たな CPU アーキテクチャへの対応を容易にした Reduced Assembly Set Compiler (RASC) を提案し,RASC 設計によるCコンパイラである NLCC の開発を行った.

2. Reduced Assembly Set Compiler の概要

Reduced Assembly Set Compiler (RASC) は出力するアセンブリのパターンを必要最低限のものに削減することで新たな CPU アーキテクチャへの対応を容易にする.

具体的には,複雑なアセンブリを生成するのでなく,定型化した単純なアセンブリ・パターン(アセンブリ・セット)の組み合わせによりアセンブリを生成する.アセンブリ・セットは以下の設計により定型化されており,1～数行程度のアセンブリの記述で容易に作成可能である.

(1) 演算用レジスタの限定

演算に用いるレジスタを限定し定型化する.

現代的な CPU アーキテクチャの命令セットでは,レジスタは汎用化されており,演算に対して多くの汎用レジスタが等価に利用できる.これは生成されるアセンブリの実行効率には寄与するが,レジスタの扱いが繁雑となりコンパイラのアセンブリ出力ロジックを複雑なものとする.

RASC では演算に用いるレジスタを限定し定型化する.これにより演算処理のアセンブリの生成を定型化・簡略化することが可能になる.

(2) 代入処理の限定

値の代入はレジスタ・ベースに限定する.

一部の CPU アーキテクチャでは,メモリからメモリへの値の代入が可能になっている.しかし現行の RISC アーキテクチャではロードストア・アーキテクチャにより必ずレジスタを経由する設計となっており,コンパイラもそれに準じた設計とすることで,値の代入処理をアーキテクチャ共通にする.

1) Global Fujitsu Distinguished Engineer

(3) 関数のプロローグ・エピローグの限定

関数のプロローグ(先頭の処理)とエピローグ(末尾の処理)は,スタックフレームの調整処理が行われるため関数内部でのレジスタの利用に左右される.とくに不揮発性のレジスタにおいてはプロローグでの保存とエピローグでの復旧が必要となるため,その利用数はスタックフレームのサイズに影響する.よって関数内部の処理で利用される不揮発性レジスタの見積もりが必要になり,プロローグ・エピローグの生成処理が煩雑になる.

関数のプロローグとエピローグの処理を限定することで,必要とするスタックフレームのサイズ計算を簡略化し,関数内部の処理への依存を不要にする.これにより関数のプロローグ・エピローグの処理を定型化する.

(4) ビルトインの活用

CPU が持つ命令には,ソフトウェア処理により代替可能な処理が多くある.

例えばビット反転は,-1 との排他的論理和により代替できる. \\
また符号拡張は,ビット判定と加算により代替できる. \\
他にも掛け算命令は,加算命令をループで繰り返すことで同等の処理が可能であり,割算も同様である.

それらの処理はソフトウェアによる代替処理のビルトインを持たせることで CPU への対応時に作成が必要なアセンブリ・セットの必要数を削減する.

(5) 最適化を限定

RASC ではアセンブリ・セットによるアセンブリ・パターンの固定化により,生成されるアセンブリの全体サイズは増加してしまうが,これは最適化によりある程度は軽減できる.

コンパイラによる最適化は,アーキテクチャ共通で行えるものとアーキテクチャ固有のものに分類される.アーキテクチャ固有の最適化はそのアーキテクチャの高速化のための専用命令を使うことなどで高速化に大きく寄与するが,CPU アーキテクチャの特有の処理が増加する原因ともなるため廃止し,共通的に行える最適化のみ採用する.

3. NLCC による実装

NLCC (No Look C Compiler)は NLUX(https://kozos.jp/nlux/)の一部として開発されているCコンパイラであり,以下の特徴がある

- セルフホストが可能(NLCC で NLCC をコンパイルできる)
- nllibc(NLUX の標準Cライブラリ)と連携し,NLUX に完全に閉じてのビルドが可能
- Debian/GNU Linux や FreeBSD の環境で,システムのヘッダファイルをインクルードしてのコンパイルが可能
- 構造体・共用体・可変長引数・ビットフィールド・構造体の引数渡しに対応
- RASC 設計により,各種 CPU アーキテクチャへの対応(クロスコンパイラの作成)が非常に低いコストで可能

NLCC は RASC として設計されており,多種 CPU アーキテクチャへの対応が簡便に行えるという特徴がある.以下に具体的な実装を示す.

3.1 演算用レジスタを２個に限定

演算に用いるレジスタは2個(R0/R1)に限定している.

演算処理のアセンブリ・パターンは R0/R1 限定として記述される.これにより記述するアセンブリを固定化し,アセンブリ・セットとして大幅に簡略化できている.

3.2 代入処理をレジスタ・ベースに限定

値の代入処理はレジスタ・ベースとなっている.このため多くの CPU アーキテクチャにおいてアセンブリ・セットの共通化が可能になっている.

表1は NLCC で定義されているアセンブリ・セットの一覧である.レジスタを R0/R1 に限定し,さらに代入処理をレジスタベースに限定することで,アセンブリ・セットを 64 種類に限定できている.

3.3 関数のプロローグ・エピローグの固定化

スタックフレームの各種サイズを固定値とすることで,スタックフレームの構造を固定にする.これにより関数のプロローグ・エピローグの処理が定型化され,関数の内部の処理に応じた調整が不要となっている.

表2は NLCC で定義されている,各種 CPU アーキテクチャのパラメータの一覧である.不揮発性のレジスタなど,各種レジスタ数などはスタックフレームのサイズに影響するが,これらのパラメータを固定値とすることで(処理やスタックフレームのサイズは無駄にはなるが)スタックフレームのサイズを固定とし,関数のプロローグ・エピローグ処理のアセンブリ・セットとしての定型化を可能としている.

3.4 ビルトイン

表3は NLCC で実装しているビルトインの一覧である.

これらのビルトインにより,一部のアセンブリ・セットは記述が不要となる.記述が不要なアセンブリ・セットは表1で省略が可となっているものであり,その個数は 28 である.よってビルトインを活用することで,動作に必要なアセンブリ・セットを最少で 36 個に削減できる.

ビルトインはソフトウェア処理による代替であるため,生成される実行コードのサイズと速度は一般に悪化する.しかしビルトインにより,一部のアセンブリ・セットを作成しなくとも動作・検証が可能となるため,CPU アーキテクチャ対応の初期段階での一時対応では有用である.

3.5 最適化

最適化は表1のアセンブリ・セットをいったんキューイングし, \\
不要なロジックを削除することで行う.これにより CPU アーキテクチャに依存しない共通処理で最適化し,最適化が CPU アーキテクチャの対応に影響しないようにしてある.

表4は amd64 における,gcc-13.3.0 と NLCC での機械語コードのサイズ比較である.サンプル・プログラムには nlcc の syntax.c を利用した.

NLCC では最適化をすることで約 52%の機械語コードサイズの削減効果があった.また NLCC では最適化時には gcc -O1 に対して約6倍のコードサイズ増加となっていた.なお狭い範囲での最適化でも,複数を組み合わせることで連鎖的に最適化がされ,実行コードの効果的なサイズ削減がされていた.

3.6 多種アーキテクチャへの対応

新たな CPU アーキテクチャへの対応は,表1のアセンブリ・セットを記述することで可能となる.

NLCC が対応している CPU アーキテクチャの一覧を表5に示す.実際に 12 種類の CPU アーキテクチャに対応させたところ,1アーキテクチャあたりの対応は概ね,基本対応が2時間,デバッグが6時間,合計8時間程度で可能であった.また対応行数の平均値は 544 行であり,600 行未満の対応で新たなアーキテクチャに対応可能になっている.

4. まとめ

新たな CPU アーキテクチャに対応させやすくするためのコンパイラの設計として,定型化した単純なアセンブリ・パターンの組み合わせによりアセンブリを生成する Reduced Assembly Set Compiler を提案し,その実装として NLCC を開発した.

アセンブリ・セットの数は 64 個,CPU アーキテクチャの対応数は 12 種類,対応行数の平均値は 544 行,対応に必要な平均時間は概ね8時間程度であった.また RASC は機械語コードのサイズ増加の懸念があるが,最適化時には gcc と比較して約6倍の機械語コードのサイズ増加となっており,このコードサイズ増加とそれによる実行速度低下を許せるならば,新たな CPU アーキテクチャに対応したコンパイラが迅速に欲しいというときに有効な

設計となる.

従来のコンパイラの多くは最適化による高速化を重要視している反面,新たな CPU アーキテクチャへの対応に非常なコストがかかる.しかし RASC 設計により多種アーキテクチャ対応を容易にし,1 日で新たな CPU アーキテクチャに対応できる 1day-compiler の開発が可能となることを示した.

今後は生成コードの実行速度や最適化の効果の詳細などの評価が必要である.

表 1　アセンブリ･セット一覧
Table 1　List of assembly-set.

名称	省略の可否	処理	内容
get_value	不可	R0 = val	定数値を R0 に代入
get_value_r1	不可	R1 = val	定数値を R1 に代入
get_address_stack	不可	R0 = SP + offset	スタックポインタを R0 に代入
get_address_stack_r1	可	R1 = SP + offset	スタックポインタを R1 に代入
get_address	不可	R0 = &label	ラベルのアドレスを R0 に代入
add_address	不可	R0 += offset	R0 をアドレス値として加算
get_r1	不可	R0 = R1	R1 を R0 に代入
set_r1	不可	R1 = R0	R0 を R1 に代入
memory_load	不可	R0 = *(R1 + offset)	メモリから R0 にロード
memory_store	不可	*(R1 + offset) = R0	メモリに R0 からストア
stack_load	不可	R0 = *(SP + offset)	スタックから R0 にロード
stack_store	不可	*(SP + offset) = R0	スタックに R0 からストア
stack_load_r1	不可	R1 = *(SP + offset)	スタックから R1 にロード
stack_store_r1	不可	*(SP + offset) = R1	スタックに R1 からストア
stack_expand	不可	SP -= size	スタックの獲得
stack_reduce	不可	SP += size	スタックの解放
funccall_reg_load	不可	R0 = arg	関数の引数を R0 にコピー
funccall_reg_store	不可	arg = R0	関数の引数に R0 をコピー
tmp_reg_load	不可	R0 = tmp	テンポラリレジスタを R0 にコピー
tmp_reg_save	不可	tmp = R0	テンポラリレジスタに R0 をコピー
tmp_reg_load_r1	不可	R1 = tmp	テンポラリレジスタを R1 にコピー
tmp_reg_save_r1	不可	tmp = R1	テンポラリレジスタに R1 をコピー
sign_extension_char	可	R0 = (char)R0	符号拡張 (signed char)
sign_extension_uchar	可	R0 = (unsigned char)R0	符号拡張 (unsigned char)
sign_extension_short	可	R0 = (short)R0	符号拡張 (signed short)
sign_extension_ushort	可	R0 = (unsigned short)R0	符号拡張 (unsigned short)
sign_extension_int	可	R0 = (int)R0	符号拡張 (signed int)
sign_extension_uint	可	R0 = (unsigned int)R0	符号拡張 (unsigned int)
calc_inv	可	R0 = R0 ^ -1	ビット反転 (R0)
calc_minus	可	R0 = -R0	符号反転
calc_op1	可		単項演算子一般
calc_add	不可	R0 = R0 + R1	加算
calc_sub	不可	R0 = R0 - R1	減算
calc_and	不可	R0 = R0 & R1	論理積
calc_or	不可	R0 = R0 | R1	論理和
calc_xor	可	R0 = R0 ^ R1	排他的論理和
calc_mul	可	R0 = R0 * R1	掛け算
calc_div	可	R0 = R0 / R1	割り算
calc_mod	可	R0 = R0 % R1	剰余算
calc_llshift	可	R0 = R0 << R1	左シフト
calc_rashift	可	R0 = R0 >> R1	算術右シフト
calc_rlshift	可	R0 = R0 >> R1	論理右シフト
calc_op2	可		2項演算子一般
branch	不可	goto label	無条件分岐
branch_zero	不可	if (!R0) goto label	R0 がゼロの場合に分岐
branch_nzero	可	if (R0) goto label	R0 が非ゼロの場合に分岐
branch_cmp_eq	不可	if (R0 == R1) goto label	分岐(equal)
branch_cmp_ne	可	if (R0 != R1) goto label	分岐 (not equal)
branch_cmp_lt	不可	if (R0 < R1) goto label	分岐 (less than)
branch_cmp_gt	可	if (R0 > R1) goto label	分岐 (greater than)
branch_cmp_le	可	if (R0 <= R1) goto label	分岐 (less equal)
branch_cmp_ge	可	if (R0 >= R1) goto label	分岐 (greater equal)
branch_cmp_ult	不可	if (R0 < R1) goto label	分岐 (unsigned less than)
branch_cmp_ugt	可	if (R0 > R1) goto label	分岐 (unsigned greater than)
branch_cmp_ule	可	if (R0 <= R1) goto label	分岐 (unsigned less equal)
branch_cmp_uge	可	if (R0 >= R1) goto label	分岐 (unsigned greater equal)
branch_cmp	可		分岐一般
function_call	可	function()	関数呼び出し

function_call_set	不可	fp = R0	関数呼び出しのポインタをR0に設定
function_call_pointer	不可	(*fp)()	ポインタ経由での関数呼び出し
function_start	不可		関数の先頭での処理
function_end	不可		関数の末尾での処理
function_register_save	不可		関数の先頭でのレジスタ保存
function_register_load	不可		関数の末尾でのレジスタ復旧

表 2　設定パラメータ一覧
Table 2　List of parameters.

名称	意味
word_size	整数値のバイトサイズ
pointer_size	ポインタ値のバイトサイズ
funccall_args_reg_number	関数の引数として用いるレジスタの個数
funccall_args_stack_number	関数の引数として用いるスタックのフィールド数
tmp_reg_number	一時利用するレジスタの個数
tmp_stack_number	一時利用するスタックのフィールド数
function_register_number	関数のプロローグで保存する，不揮発性(callee-saved)のレジスタの個数
function_saveparam_number	関数のプロローグで保存する，各種の値(戻り先アドレスなど)のフィールド数
stack_align_size	スタックのアラインサイズ
stack_correct_size	スタックの調整サイズ

表 3　ビルトイン一覧
Table 3　List of builtins.

名称	意味	内容
BUILTIN_EXTENSION	符号拡張	最上位ビットを参照し符号拡張ぶんのビットを加算する
BUILTIN_INV	ビット反転	オール1の値との排他的論理和
BUILTIN_MINUS	符号反転	0からの減算
BUILTIN_R1	R1 に対する処理	R1 によるアセンブリ・パターンを利用しない

表 4　最適化の効果
Table 4　Result of optimization.

コンパイル方法	サイズ(バイト)	gcc -O1 に対する割合(%)
gcc -O0	19966	137
gcc -O1	14546	100
gcc -Os	12498	85
nlcc (最適化無)	162917	1120
nlcc (最適化有)	85553	588

表 5　CPU アーキテクチャの対応状況
Table 5　Suppoted architectures.

アーキテクチャ名	対応行数	対応状況
x86	539	セルフビルド・全テストを確認完了
amd64	548	セルフビルド・全テストを確認完了
ARM	532	エミュレータ環境での動作を確認
MIPS	541	エミュレータ環境での動作を確認
PowerPC	524	エミュレータ環境での動作を確認
AArch64	556	エミュレータ環境での動作を確認
Thumb	619	エミュレータ環境での動作を確認
Thumb2	583	エミュレータ環境での動作を確認
MIPS16	564	エミュレータ環境での動作を確認
OSECPU	494	実験的対応
RISC-V	521	エミュレータ環境での動作を確認
RX	517	エミュレータ環境での動作を確認
平均	544	