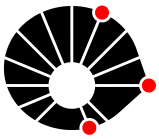

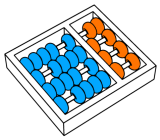

# Inlining de funções declaradas em módulos distintos no compilador LLVM

*Heitor Boschirolli Comel* *Sandro Rigo*

Relatório Técnico - IC-PFG-19-60
Projeto Final de Graduação
2019 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

# Inlining de funções declaradas em módulos distintos no compilador LLVM

Heitor Boschirolli Comel* Sandro Rigo†

## 1 Introdução

LLVM é uma coleção de tecnologias modulares e reutilizáveis que podem ser usadas para desenvolver o *front end* para qualquer linguagem de programação e o *back end* para um conjunto de instruções de qualquer arquitetura. Devido à sua modularidade o LLVM é bastante usado para trabalhos de otimização na área de compiladores, pois é possível alterar apenas uma de suas componentes e medir o impacto das modificações na compilação como um todo.

Dentre as otimizações feitas por um compilador está o que é chamado de *inline*. Este processo consiste em substituir uma chamada de função pelo código da função a ser chamada. Isso elimina as instruções de *call* e *return* necessárias para a chamada da função o que pode gerar ganhos de eficiência. O aumento no tamanho da função que ocorre como consequência do inline, porém, pode afetar a eficiência do programa de forma negativa, pois o aumento no número de instruções afeta a cache de instruções. Assim sendo, a seleção de quais chamadas de funções devem ser substituídas por um *inline* tem grande importância.

O compilador LLVM já possui um passe de inline no processo de otimização, mas ele é restringido a funções de um mesmo módulo; a instrução de *call* e a função chamada precisam estar definidas no mesmo arquivo. Essa limitação não pode ser facilmente contornada devido a forma como o LLVM é projetado, os passes de otimização não podem enxergar mais que um módulo por vez.

Para obter o efeito de *inlining* mesmo com as restrições impostas pelo compilador, fez-se um inline composto pela substituição das instruções de *call* e *return* por *jumps*. Isso deve gerar um ganho menor em performance quando comparado com o *inline* comum, mas a vantagem de que pode ser aplicado em mais funções permite que seja combinado com o *inline* já contido no LLVM para um ganho em performance.

Assim como no *inline* tradicional, a escolha das funções para realizar a substituição tem de grande importância.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.
†Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

## 2 Metodologia

O *inlining* feito por meio de jumps foi adicionado a um *fork* do compilador LLVM. Os passes de otimização escritos seguem os seguintes passos para realizar o *inlining*.

1. Clonar a função alvo da instrução de *call*.
2. Trocar o alvo da instrução de *call* para uma função clonada.
3. Trocar a instrução de *call* por um *jump* para uma função clonada.
4. Criar um rótulo global após o salto para a função clonada.
5. Substituir a instrução de *return* da função clonada por um *jump* para o rótulo global definido.
6. Corrigir a pilha do programa.

A sequência de passos acima descreve uma forma de realizar o *inlining* especificamente no compilador LLVM. Alguns dos passos realizados existem devido a restrições deste compilador e poderiam ser eliminados em se fossem feitos em outro compilador. De forma análoga, outros compiladores podem conter outras restrições e precisar de alguns passos não presentes na metodologia descrita aqui.

Para realizar o *inline* somente em um conjunto de funções contávamos um arquivo que descrevia em cada linha uma tripla (*caller* do inline, numero da chamada a sofrer *inline*, *callee* do *inline*). Este arquivo pode ser gerado manualmente, mas um programa para gerá-lo com funções que devem proporcionar um maior ganho de desempenho quando feito um *inline* está sendo feito.

Se fosse desejado fazer inline das duas chamadas da função **func2** no código da Figura 1, o arquivo para descrever os *inlines* precisaria ser como o mostrado na Figura 2.

```
1  void func2() {
2    ...
3  }
4
5  void main() {
6    A();
7    func2();
8    B();
9    A();
10   func2();
11   C();
12
13   return 0;
14 }
```

Figura 1: Exemplo de código na linguagem C em que se deseja fazer *inline* das duas chamadas da função **func2**.

```
main 2 func2
main 5 func2
```

Figura 2: Exemplo de arquivo para descrever o *inline* das duas chamadas da função **func2** no código da Figura 1

Para mostrar o efeito de cada etapa, considere o programa em C descrito na Figura 3. A Figura 4 mostra uma versão simplificada do código em linguagem de máquina gerado sem o uso de *inline*. A seguir são descritos como foram feitos, por que servem e que efeito cada passo tem no código compilado.

```
int func1() {
  return 3;
}

int main() {
  func1();
  return 0;
}
```

Figura 3: O código em linguagem de máquina gerado a partir deste exemplo será usado para demonstrar o efeito de cada etapa no programa.

## 2.1 Clonagem da função alvo

O *inlining* é feito na chamada de uma função **A** para uma função **B**. A função **B** é o que é clonado no nosso procedimento. Isso é necessário porque para retornar para a função **A** será feito nas próximas etapas um *jump* para um rótulo específico da função **A**. Assim sendo cada *inline* requer um clone. A função original é mantida pois podem haver chamadas que não sofrerão *inline*.

Para a clonagem foi feito um passe de otimização que operava na primeira representação intermediaria do LLVM - chamada simplesmente de *intermediate representation* ou IR. Como a função clonada ainda não é usada no programa é possível que outras otimizações o removam, para contornar isso o passe foi configurado para ser o ultimo a rodar na IR. Os passes feitos na segunda representação intermediária do LLVM - chamada de *Machine IR* -, são feitos depois dos passes da IR, ou seja, depois do passe descrito aqui, mas passes que operam na *Machine IR* não podem remover ou adicionar funções.

O nome dado aos clones criados foi usado a nosso favor. Eles eram a concatenação de um prefixo definido, do *caller*, do número da chamada do *caller* a sofrer *inline* e do *callee*. Isso simplifica os passos seguintes.

Na Figura 5 pode ser visto a funcao clonada em linguagem de máquina que é adicionada ao código da Figura 4 após o passe que clona funções ser adicionado ao compilador.

```
func1:                                  # @func1
.cfi_startproc
# %bb.0:
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq    %rsp, %rbp
.cfi_def_cfa_register %rbp
movl    $3, %eax
popq    %rbp
.cfi_def_cfa %rsp, 8
retq
.Lfunc_end0:
.size   func1, .Lfunc_end0-func1
.cfi_endproc
# -- End function
.globl  main                    # -- Begin function main
.p2align        4, 0x90
.type   main,@function
main:                                   # @main
.cfi_startproc
# %bb.0:
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq    %rsp, %rbp
.cfi_def_cfa_register %rbp
subq    $16, %rsp
movl    $0, -4(%rbp)
callq   func1
xorl    %ecx, %ecx
movl    %eax, -8(%rbp)          # 4-byte Spill
movl    %ecx, %eax
addq    $16, %rsp
popq    %rbp
.cfi_def_cfa %rsp, 8
retq}
```

Figura 4: Resultado simplificado de uma compilação do código da Figura 3 sem o uso de *inline*.

```
.globl  lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1 # -- Begin
function lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1
.p2align        4, 0x90
.type   lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1,@function
lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1: # @lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1
.cfi_startproc
# %bb.0:
subq    $8, %rsp
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq    %rsp, %rbp
.cfi_def_cfa_register %rbp
movl    $3, %eax
popq    %rbp
.cfi_def_cfa %rsp, 8
jmp     lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1_return
.Lfunc_end2:
.size   lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1,.Lfunc_end2-lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1
.cfi_endproc
# -- End function
```

Figura 5: Função clonada no programa compilado presente apos o passe de clone ser adicionado ao compilador.

### 2.2 Trocar o alvo da instrução de *call*

Uma vez que se tem a função clonada, trocou-se os alvos dos *calls* que iriam passar pelo *inline* para as funções de clonadas. A utilidade disso não está relacionada ao *inline* em si porque a instrução de *call* modificada sera removida no futuro, mas isso facilita as operações de substituição de *calls* por *jump* no futuro. O prefixo colocado na função clonada permite que saibamos pelo alvo de um *call* se ele deve ser trocado por um *jump* ou não.

Isso foi feito no mesmo passe em que a clonagem das funções.

### 2.3 Substituir a instrução de *call* por um *jump*

Esta e as subsequentes etapas foram todas realizadas na segunda representação intermediaria do LLVM, ela é específica da arquitetura para qual o programa está sendo compilado, portanto seria preciso escrever um procedimento para cada arquitetura que o *inline* seria suportado. Decidiu-se, por hora, restringir a arquitetura x86 da Intel, além de ser bastante comum era o que tínhamos na máquina em que estávamos testando.

O rótulo para qual o *jump* seria feito já estava presente no programa, pois o LLVM define as funções em suas representações intermediarias por meio de rótulos. Localizar os *calls* a serem substituídos também foi simples dado que os alvos destes eram para funções clonadas o que conseguíamos identificar por causa do prefixo que elas continham. O alvo do *call* também nos fornecia o nome do rótulo para o qual deveríamos saltar.

Para cada instrução que deveria ser substituída, simplesmente deletava-se o call antigo e criava-se um *jump* para um rótulo de mesmo nome do alvo da instrução de *call*.

Na Figura 6 pode-se o *caller* com a instrução de *call* trocada por uma instrução de *jump*.

### 2.4 Criar um rótulo global após o *jump*

Para criar um ponto de retorno do *inline* precisa-se de um rótulo logo abaixo do salto para a função clonada, para que esta possa voltar para o mesmo ponto quando a execução estivesse terminada.

O LLVM não possui funcionalidade especifica para criação de rótulos, apesar de suas representações intermediarias os usarem em vários momentos, tudo isso é feito de forma implícita. Isso nos forçou a modificar um pouco como o LLVM emitia instruções.

O primeiro passo foi localizar a instrução em que o rótulo seria colocado. Isso é bastante simples pois é sempre a instrução que segue o *jump* que inserimos na etapa anterior; fazendo essas duas etapas no mesmo passe tornou a tarefa trivial.

Em seguida modificou-se a classe *MachineInstruction* do LLVM; adicionou-se um novo atributo que indicaria se um rótulo deveria ser adicionado nesta instrução e qual seria o nome dele. Quando encontrava-se a instrução que precisava de um rótulo, atribuíamos o valor adequado ao atributo criado.

Durante a etapa de emissão de instruções, para cada uma delas adicionou-se uma verificação no atributo criado, se ele indicasse que um rótulo precisava ser emitido, colocava-o antes de prosseguir com a emissão de instruções. Isso garantia que o rótulo estava presente, mas ainda não permitia que instruções de *jump* fossem feitas para ele, pois se tratava de um rótulo desconhecido pelo compilador.

```
1  .globl  main                    # -- Begin function main
2  .p2align        4, 0x90
3  .type   main,@function
4  main:                                   # @main
5  .cfi_startproc
6  # %bb.0:
7  pushq   %rbp
8  .cfi_def_cfa_offset 16
9  .cfi_offset %rbp, -16
10 movq    %rsp, %rbp
11 .cfi_def_cfa_register %rbp
12 subq    $16, %rsp
13 movl    $0, -4(%rbp)
14 jmp     lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1
15 movl    %ecx, %eax
16 addq    $16, %rsp
17 popq    %rbp
18 .cfi_def_cfa %rsp, 8
19 retq
20 .Lfunc_end1:
21 .size   main, .Lfunc_end1-main
22 .cfi_endproc
23 # -- End function
```

Figura 6: *Caller* com a instrução de *call* substituída por um *jump*.

Para expor o rótulo criado ao compilador, declarava-se uma função de mesmo nome do rótulo, como já mencionado, nomes de funções são convertidos pelo LLVM em rótulos e a declaração de uma função o adiciona o rotulo na tabela do compilador.

Na Figura 7 pode-se ver o *caller* com o rótulo adicionado.

## 2.5 Substituir o *return* por um *jump*

Com o rótulo criado e exposto para o compilador o procedimento de substituir a instrução de *return* por um jump é bastante simples. O rótulo de retorno sempre era formado pelo nome da função a sofrer *inline* concatenado a *string* "_return". Com isso sempre sabíamos o nome do rótulo pois é possível sempre obter o nome da função em que se está. Por fim sabíamos se a instrução de *return* precisava ou não ser trocado ao analisar a função em que estávamos; ele precisava ser substituído se, e somente se, estava dentro de uma função clonada.

Sabendo o nome do rótulo e quando a substituição precisava ser feita, seguiu-se um procedimento muito similar a substituição da instrução de *call*. Adicionou-se o *jump* e em seguida removeu-se o *return*.

Na Figura 8 pode-se ver o *callee* com a instrução de *return* substituída por uma instrução de *jump*.

```
main:                                   # @main
.cfi_startproc
# %bb.0:
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq    %rsp, %rbp
.cfi_def_cfa_register %rbp
subq    $16, %rsp
movl    $0, -4(%rbp)
jmp     lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1
.globl
    lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1_return

lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1_return:
movl    %ecx, %eax
addq    $16, %rsp
popq    %rbp
.cfi_def_cfa %rsp, 8
retq
.Lfunc_end1:
.size   main, .Lfunc_end1-main
.cfi_endproc
# -- End function
```

Figura 7: *Caller* com o rótulo de retorno adicionado.

```
1  .globl  lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1 #
       -- Begin function
       lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1
2  .p2align        4, 0x90
3  .type   lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1,
       @function
4  lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1: #
       @lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1
5  .cfi_startproc
6  # %bb.0:
7  pushq   %rbp
8  .cfi_def_cfa_offset 16
9  .cfi_offset %rbp, -16
10 movq    %rsp, %rbp
11 .cfi_def_cfa_register %rbp
12 movl    $3, %eax
13 popq    %rbp
14 .cfi_def_cfa %rsp, 8
15 jmp
       lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1_return

16 .Lfunc_end2:
17 .size   lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1,
       .Lfunc_end2-
       lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1
18 .cfi_endproc
19 # -- End function
```

Figura 8: *Callee* com a instrução de *return* substituída por uma instrução de *jump*.

### 2.6 Corrigir a pilha do programa

As instruções de *call* e de *return* afetam a pilha do programa, pois o endereço de retorno é empilhado durante o *call* e desempilhado no *return*. Nos casos em que estas instruções são substituídas por *jumps* a pilha do programa ficava em um estado inválido, o que causava falhas de segmentação quando executado.

Para corrigir isso adicionou-se um *offset* na pilha no inicio de cada função clonada, isso fazia com que o restante dos elementos estivessem no local em que o programa esperava. Após o retorno das funções clonadas remove-se o *offset* da mesma forma que o endereço de retorno seria removido.

Nas Figuras 9 10 pode-se ver os *caller* e o *callee* com os *inlines* feitos e pilhas corrigidas.

```
1  .globl  lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1 #
       -- Begin function
       lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1
2  .p2align        4, 0x90
3  .type   lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1,
       @function
4  lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1: #
       @lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1
5  .cfi_startproc
6  # %bb.0:
7  subq    $8, %rsp
8  pushq   %rbp
9  .cfi_def_cfa_offset 16
10 .cfi_offset %rbp, -16
11 movq    %rsp, %rbp
12 .cfi_def_cfa_register %rbp
13 movl    $3, %eax
14 popq    %rbp
15 .cfi_def_cfa %rsp, 8
16 jmp
       lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1_return

17 .Lfunc_end2:
18 .size   lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1,
       .Lfunc_end2-
       lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1
19 .cfi_endproc
20 # -- End function
```

Figura 9: *Callee* com inline completo e pilha corrigida.

## 3 Testes

Para um estudo do impacto do *inlining* na performance é preciso concluir outra etapa do projeto que seleciona funções propícias para o *inline*; como esta etapa ainda não está concluída, foram feitos testes para garantir a corretude do que foi feito.

```
main:                                   # @main
.cfi_startproc
# %bb.0:
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq    %rsp, %rbp
.cfi_def_cfa_register %rbp
subq    $16, %rsp
movl    $0, -4(%rbp)
jmp     lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1
.globl
    lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1_return

lalavimuClone_main__lalavimuSeparator__1__lalavimuSeparator__func1_return:
addq    $8, %rsp
xorl    %ecx, %ecx
movl    %eax, -8(%rbp)          # 4-byte Spill
movl    %ecx, %eax
addq    $16, %rsp
popq    %rbp
.cfi_def_cfa %rsp, 8
retq
.Lfunc_end1:
.size   main, .Lfunc_end1-main
.cfi_endproc
# -- End function
```

Figura 10: *Caller* com inline completo e pilha corrigida.

Existem duas verificações a serem feitas para saber se o *inlining* ocorreu de forma correta. O código em linguagem de máquina precisa conter os rótulos e *jumps* inseridos e não conter as instruções de *call* removidas. Além disso, é preciso que o programa rode sem erros e tenha o mesmo resultado do original.

Além de alguns testes simples com programas escritos por nós, testou-se nossa versão modificada do LLVM na compilação do kernel Linux. Utilizamos *prints* para verificar se as funções que sofreram *inline* estavam sendo executadas, utilizamos o sistema para verificar que o kernel foi compilado corretamente e verificamos que as instruções foram modificadas como deveriam nos arquivos intermediários gerados pelo LLVM que contém o código compilado em linguagem de máquina.

## 4 Trabalho futuro

O próximo e mais importante passo é integrar o que foi feito neste trabalho com o que está sendo feito para determinar funções propicias a passarem pelo *inline*. Outras etapas interessantes podem ser a expansão do que foi feito para arquiteturas além de x86 e algumas otimizações no processo de compilação que podem ser feitas.