# Relatório Parcial do Edital Único PRPq 01/2010

**Projeto: AlocReg – Incorporação de um Novo Alocador de Registradores em LLVM**

# FAPEMIG/PROBIC 2010

Aluno: Matheus Lima Diniz Arauújo
Coordenadora: Prof.$^a$ Mariza Andrade da Silva Bigonha

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Belo Horizonte

13 de agosto de 2011

# 1 Implementação do Alocador Global de Registradores Baseado em Crescimento de Domínios Ativos e Combinação de Registradores em LLVM

O objetivo desse documento é apresentar de forma simples e objetiva os primeiros passos no desenvolvimento de um novo Alocador de Registradores para o compilador LLVM.

A alocação de registradores é um dos problemas mais importantes para otimização de código [4, 5]. Registradores são memórias pequenas, caras e rápidas que existem dentro da CPU, e que guardam valores frequentemente usados durante a execução de programas. Alocação de registradores é a fase do compilador que decide quais valores devem ser atribuídos a estes registradores. Geralmente, existem poucos registradores na máquina, menos que o necessário, e isto faz com que alguns valores contidos em registradores sejam derramados para a memória. Derramar um valor para a memória significa que este valor será acessado via memória, e não via registradores, fazendo com que o tempo de execução do programa aumente. Para se ter uma idéia, o x86, a arquitetura de computadores mais popular entre os fabricantes de *desktops*, possui somente 7 registradores de propósito geral. Logo, compiladores precisam maximizar, tanto quanto possível, o número de variáveis associadas a registradores. O problema de se encontrar uma alocação de registradores ótima é NP-completo, logo deve-se buscar heurísticas para resolvê-lo. Este trabalho se propõe a implementar uma nova heurística para alocação de registradores baseado no algoritmo de Crescimento de Domínios Ativos proposto por [**?**] e modificado por [2] (*Live Range Growth*) no compilador LLVM [7]. O resultado da pesquisa proposta será o próprio alocador de registradores incorporado em LLVM.

A soluo baseada em Crescimento de Domnios Ativos foi proposta por Ottoni e Araújo [**?**] para processadores dedicados DSPs *(Digital Signal Processors)* com o objetivo de resolver o problema da alocao de referncias a vetores em registradores de endereamento dentro de laos que contm mais de um bloco bsico. Nossa proposta é utilizar este mtodo para a alocao global de registradores em processadores de propsito geral. O problema central para se obter uma solução baseada em Crescimento de Domínio Ativo (*Live Range Growth*) para a Alocação Global de Registradores é o cálculo do menor custo associado a um dado domínio ativo. Neste algoritmo, considera-se um domínio ativo (*live range*) como sendo um conjunto de variáveis que são alocadas a um mesmo registrador. Logo, todas as variveis de um mesmo domnio ativo são atribuídas ao mesmo registrador, fazendo com que todos os usos e definies destas variveis sejam feitos através deste registrador. Uma *web* é a combinação de correntes du (definição-uso) que se interceptam, isto é, que contêm um uso em comum [**?**]. Inicialmente, cada *web* colocada separadamente em um domnio ativo. A seguir, usado um algoritmo heurstico, denominado Crescimento de Domnios Ativos, o qual faz uma juno sucessiva de pares de domnios ativos, at que o nmero total deles atinja o nmero de registradores disponveis na arquitetura em questo. Para decidir o par de domnios ativos unidos a cada iterao do algoritmo, todas as combinaes de pares de domnios ativos so avaliadas, e a que resulta em um menor custo escolhida. O custo de um domnio ativo medido pelo nmero de instrues de *load* e *store* necessárias para o ajuste do registrador. Assim, o problema central desta tcnica é a determinao, para um dado domnio ativo, do nmero destas instrues para manter o registrador com a varivel correta durante o fluxo de execução do programa. Para isso, essa técnica tenta contornar os problemas da coloração de grafos fazendo: (a) análise de fluxo de controle de programa, (b) análise de variáveis vivas,

e (c) análise de fluxo de dados denominada Análise de Alcançabilidade e Consistência de Registradores, [2]

Para tornar mais claro este guia, o texto foi organizado da seguinte forma: a Seção 2 apresenta uma introdução sobre o que é o compilador LLVM e porque escolher este compilador para desenvolver não só alocadores de registradores mas qualquer outra otimização para compiladores. Seção 3 apresenta uma visão suscinta dos atuais alocadores de registradores que compõe as versões padrões do LLVM e por fim, a Seção 4 descreve passo a passo como começar a implementação de um novo alocador de registradores para o LLVM. A Seção 5 conclui esse texto.

# 2 O Compilador LLVM

LLVM (Low Level Virtual Machine) [1]. é uma infraestrutura de compilador escrita em C++ que foi desenvolvida com o propoósito de otimizar em tempo de compilação, ligação e execução de programas escritos em diferentes linguagens, como C, C++, Pyton, dentre outras. Atualmente o LLVM está sobre os domiínios da Apple Inc.

Um conceito muito importante em LLVM é denominado *passes*. *Passes* são pequenos módulos de código que juntos formam praticamente todo o LLVM. Para realizar qualquer modificação, seja para otimizar ou desenvolver novos recursos no LLVM, é preciso saber como implementar estes *passes*. Alocadores de Registradores, Geradores de Código de Máquina, ou mesmo um contador de funções para um programa pode ser um *pass*. Com este sistema de *passes*, o LLVM possui a característica de ser muito modular, demonstrando um código intuitivo, e assim tornar relativamente fácil realizar otimizações e modificações.

## 2.1 Estrutura do LLVM

O LLVM pode ser dividido por 2 partes básicas: FrontEnd e BackEnd, esta forma de divisão é comum para praticamente todos os compiladores.

**FrontEnd.** O *FrontEnd* analisa o código fonte e gera a representação interna, ou representação intermediária (IR), do programa para o compilador. A representação interna do LLVM é baseada no padrão SSA, (*Static Single Assigment*) [8]. Essa representação é comum para todas as partes da compilação do programa.

**BackEnd.** O *BackEnd* do LLVM analisa o IR gerado no *FrontEnd*, otimiza-o e depois realiza a geração de código, ou seja, gera o código de maquina final para um determinado sistema.

Este texto focará na parte da otimização do *BackEnd*, para ser mais especifico na parte de otimização da alocação de variáveis do programa a ser compilados nos registradores do sistema LLVM.

[1] [llvm] http:llvm.org

**Porque o LLVM?** A escolha se deu por duas razões. Primeiro por que o LLVM possui uma estrutura bastante modular, com um sistema de *passes* eficiente; o código é bem escrito e comentado; possui uma comunidade bem entusiasmada onde são trocadas informações, perguntas e respostas. Na lista:

`http://lists.cs.uiuc.edu/mailman/listinfo/llvmdev`

é possível encontrar diversas soluções para problemas relacionados ao desenvolvimento de recursos para LLVM. Em segundo lugar por que já havíamos implementado e testado o algoritmo de alocação de registradores baseado em crescimento de domínios ativos e combinação de Registradores no GCC [6].

# 3 Alocadores de Registradores

## 3.1 Alocador Global de Registradores Baseado em Crescimento de Domínios Ativos e Combinação de Registradores

O problema central para se obter uma solução baseada em Crescimento de Domínio Ativo (*Live Range Growth*) para a Alocação Global de Registradores é o cálculo do menor custo associado a um dado domínio ativo. Neste algoritmo, considera-se um domínio ativo (*live range*) como sendo um conjunto de variáveis que são alocadas a um mesmo registrador. Logo, todas as variáveis de um mesmo domínio ativo são atribuídas ao mesmo registrador, fazendo com que todos os usos e definições destas variáveis sejam feitos via este registrador. Uma *web* é a combinação de correntes du (definição-uso) que se interceptam, isto é, que contêm um uso em comum. Inicialmente, cada *web* é colocada separadamente em um domínio ativo. O algoritmo heurístico, denominado Crescimento de Domínios Ativos, faz uma junção sucessiva de pares de domínios ativos, até que o número total deles atinja o número de registradores disponíveis na arquitetura em questão. Para decidir o par de domínios ativos unidos a cada iteração do algoritmo, todas as combinações de pares de domínios ativos são avaliadas, e a que resulta em um menor custo é escolhida. O custo de um domínio ativo é medido pelo número de instruções de *load* e *store* necessárias para o ajuste do registrador. Assim, o problema central desta técnica é a determinação, para um dado domínio ativo, do número destas instruções para manter o registrador com a variável correta durante o fluxo de execução do programa.

## 3.2 Alocadores de Registradores no LLVM

A maioria dos alocadores de registradores utilizam a técnica de coloração de grafos para resolver o problema e realizar o menor número de derramamentos possível para a memória. Apesar dessa técnica ser intuitiva, o tempo de alocação para algoritmos baseado neste método é relativamente caro. Como exemplo de outras técnicas temos [11], [12]

O LLVM em sua versão 2.8 possui três alocadores de registradores diferentes, o *Linear Scan* [12], [9], o *Greedy Register Allocator* [10] e o Alocador de Registrador Básico [1].

**Linear Scan Register Allocator.** O *Linear Scan* usa um algorítimo de alocação de registradores globais, não baseado na coloração de grafos. Ele se comporta da seguinte maneira: Dado o tempo de vida (*live range*) das variáveis em uma funç ao, o algoritmo do *Linear Scan* escaneia todos os tempos de vida alocando as variáveis de forma gulosa. O algoritmo é simples, eficiente e produz um código relativamente limpo. É útil em situações que requerem um bom tempo de compilação e um claro entendimento. O *Linear Scan* era usado como o Alocador de Registrador default do LLVM até a Versão 2.8.

**Greedy Register Allocator** . O *Greedy Register Allocator* como o próprio nome já menciona, também é baseado em soluções gulosas para o problema da alocação de registradores. Ele possui um comportamento semelhante ao *Linear Scan* mas é um pouco mais esperto. Ele possui as seguintes principais características:

- Gera binários em média 2% menores e 10% mais rápidos.
- É mais rápido que o Linear Scan em tempo de compilação para programas que possuem funções pequenas, mas perde ao compilar grandes funções. A partir da versão 2.9 do LLVM, o *Greedy Register Allocator*, se tornou o alocador de registradores default.

**Basic Register Allocator.** O *Basic Register Allocator* é um alocador básico para o LLVM, de forma a ser considerado um exemplo didático de como criar um alocador de registradores para o mesmo. Esse alocador não promove otimização de código e é inviável para o uso na compilação de programas. Ele analisa a vida dos registradores e derrama-os (spills) sempre que não há registradores disponíveis, se mostrando uma péssima heuriística. Utilize-o como base para implementar seus próprios alocadores.

# 4 Implementação de um Alocador de Registradores em LLVM

Para implementar um novo alocador de registradores em LLVM, iniicialmente é necessário configurar e instalar o LLVM. As próximas seções mostram suscintamente como fazê-lo.

## 4.1 Instalação e Configuração do LLVM

O LLVM é um compilador complexo, portanto possui inumeráveis configuração de compilação, instalação e execução. Esta é mais uma característica que comprova a sua *maleabilidade* para diversas arquiteturas e linguagens. Nesta seção é descrito como compilar e executar o LLVM a partir do seu fonte.

1. Inicialmente é necessário fazer o *download* do código fonte, no *site:*

`http://llvm.org/releases/`

Nesse *site* encontra-se as últimas versões do LLVM, selecione a versão mais nova. Observe que este tutorial foi realizado baseado na Versão 2.9. Após o *download*, descompacte os arquivos, a pasta criada será o *root* do LLVM, e via terminal, realize os comandos:

```
./configure
make
```

O primeiro comando tem o objetivo de configurar o código fonte do LLVM a ser compilado, de modo a adequar ao seu sistema. O segundo comando compila o código.

2. Após o primeiro passo feito, já se tem o *BackEnd* do LLVM instalado e utilizável em sua máquina. Mas não adianta tentar compilar um código em C por exemplo, pois é necessário o FrontEnd do compilador. Para tal, é necessário fazer o download do código fonte do *LLVM-GCC FrontEnd* na mesma página do *download* anterior. Para compilar e executar, basta seguir as instruções do arquivo README.llvm contido no *download* acima. Siga estritamente os comandos neste arquivo,principalmente na hora de escolher a arquitetura do sistema, pois caso contrário o LLVM-GCC pode não compilar.

Pronto. Uma vez efetuado corretamente os passos mostrados, vá ao diretório `install` criado e via terminal acesse o diretório `bin`. Tente compilar um arquivo `.c` com o binário `llvm-gcc`, usando os mesmos argumentos do *gcc*, para assim certificar que tudo foi instalado corretamente. Utilize o `llvm-gcc` para gerar *bytecodes(.bc)* dos programas a serem compilados.

## 4.2 Implementando o seu primeiro PASS

Nesta seção discutiremos a estrutura básica de um *Pass* por meio de um exemplo bem simples, depois será discutida como deveria ser a estrutura de um *pass* para alocadores de registradores. Crie um diretório dentro do diretório em algum lugar no *root* do LLVM para que sejam, guardados os *passes* criados.

**Makefile** Para que o seu *pass* seja compilado e ligado (*linked*) à estrutura do LLVM é necessário criar um `Makefile` para ele com o seguinte conteúdo:

```
# Makefile for hello pass

# Caminho até o diretório de maior nível do LLVM

LEVEL = ../../..

# Nome da Biblioteca a ser construiída

LIBRARYNAME = Hello
```

# Fazer com que o pass seja carregado por ferramentas do LLVM

```
LOADABLE_MODULE = 1
```

# Incluir o *makefile* na implementação

```
include $(LEVEL)/Makefile.common
```

**Pass Básico** O *pass* que será explicado possui a simples função de imprimir no terminal o nome das funções, não externas, que o programa a ser compilado possui. Desta forma o *pass* não modifica o programa, apenas o inspeciona. Vamos chamar o arquivo que conterá o pass de `hello.cpp`.

```
1. #include "llvm/Pass.h"
2. #include "llvm/Function.h"
3. #include "llvm/Support/raw_ostream.h"
4. using namespace llvm;
5. namespace {
6. struct Hello : public FunctionPass {
7. static char ID;
8. Hello() : FunctionPass(ID) {}
9. virtual bool runOnFunction(Function &F) {
10. errs() << "Hello: " << F.getName() << "\n";
11. return false;
12. }
13. };
14.
15. char Hello::ID = 0;
16. static RegisterPass<Hello> X("hello", "Hello World Pass", false, false);
17. }
```

**Linhas de 1 a 3:** É necessário incluir os arquivos:

- `Pass.h`, pois estamos escrevendo um *pass*.
- `Function.h`, pois estamos operando sobre as funções do programa.
- `raw_ostream.h`, pois estamos imprimindo algo.

**Linha 5:** É iniciado um *namespace* anonimo. *Namespace anonimo* em C++ é equivalente à palavra-chave *static* em C, no escopo global. Desta forma torna as coisas que são declaradas dentro deste *namespace* apenas visível pelo arquivo corrente.

**Linha 6:** Declaração do *pass* em si. Declara que a classe *Hello* que estamos criando é subclasse da classe `FunctionPass`. Classe esta responsável por operar funções do programa compilado.

**Linhas 7 e 8:** Declara um identificador para o *pass*. Cada *pass* do LLVM possui seu identificador. O ID é configurado em `FunctionPass()`.

**Linhas 9 e 13:** Sobrescreve o método abstrato `runOnFunction()`, herdado da classe `FunctionPass`. É dentro deste método que é implementado toda a função do *pass* a ser gerado. No exemplo é apenas impresso o nome das funções do programa compilado (`F.getName()`) na saída `errs()`.

**Linha 15:** O ID do nosso *pass* é inicializado. O LLVM usa o endereço do ID para identificar o *pass* e não o seu conteúdo, portanto podia ser qualquer valor diferente de 0.

**Linha 16:** Por último registramos a nossa classe `Hello`, dando a ela um argumento que será usado na linha de comando ao compilar o *pass*, no caso `hello`, o nome de nosso pass será `Hello World Pass`, os últimos 2 argumentos descreve o seu comportamento. O primeiro `false` representa que o pass irá não somente passar pelo CFG(Control Flow Graph), então é `false`. O segundo `false` significa que o nosso *pass* não é um `AnalysisPass(somos FunctionPass)`.

## 4.3 Compilando e Executando o PASS no LLVM

Agora que você possui o seu `pass hello.cpp` e o `Makefile`, basta realizar o sequinte comando no terminal:

```
./gmake
```

Assim será gerado o arquivo `Hello.so` no diretório

```
($LEVEL)/Debug+Asserts/lib/
```

este arquivo é o seu *pass* compilado.

Para executá-lo use os seguintes comandos:

```
$ opt -load ../../../Debug+Asserts/lib/Hello.so -hello < hello.bc > /dev/null
```

Este comando usa a ferramenta do LLVM `opt` para acessar o seu *pass*, da seguinte forma: −`load argumento`, onde *argumento* é o caminho de seu *pass*.

−`hello`, quando o nosso *pass* foi registrado, este comando passa a ser válido e assim o *pass hello* é executado sobre o `bytecode hello.bc`. Este `hello.bc` é um programa que foi gerado a partir de uma compilação do *FrontEnd*, o LLVM-GCC , gera arquivo `.bc` a partir de arquivos `.c`.

*/dev/null* significa ignorar os resultados do `opt`. Após executar este comando, será impresso no terminal os nomes das funções que `hello.bc` possui. Concluindo assim o objetivo do *pass*.

### 4.3.1 Pass para um Alocador de Registrador

Esta seçã apresenta uma rápida introducção a respeito de como os Alocadores de Registradores funcionam no LLVM. Somente com este material não será possível implementar um alocador

de registradores, mas será mostrado alguns pontos necessários para o seu desenvolvimento. Vamos supor que o nosso alocador de registradores se chame *Exemple Register Allocator*.

**Configurando o LLVM para Reconhecer o seu Pass.**

É necessário saber que no arquivo

```
../lib//include/llvm/CodeGen/passes.h
```

existe a declaração de todos os *passes* que serão utilizados pelo *BackEnd* do LLVM, portanto é necessário que a declaração da função que cria o seu *pass* esteja lá. Por exemplo:

```
Functivos *createExampleRegisterAllocator();
```

Agora o *pass* do alocador `existe` para o LLVM, mas ainda é necessário configurar para que ele seja utilizado na alocação de registradores. A função que determina qual alocador será utilizado de acordo com a preferência do usuário é a `createRegisterAllocator()` que está implementada no arquivo

```
../lib/CodeGen/passes.cpp
```

se deseja criar um novo alocador de registrador é necessário configurar este arquivo, incrementando-o com o seu alocador.

**Estruturas do *pass* que possui um alocador de registradores.**

**Entrada.** Para entender como um alocador de registrador funciona na prática, é preciso saber como é realizado as entradas do seu algoritmo. Em um compilador, os alocadores recebem como entrada *blocos básicos*. Blocos básicos são blocos de instruções consecutivas sem desvios, exceto no seu fim, possuindo uma única e entrada, e normalmente correspondem a uma função ou rotina em uma determinada linguagem. A partir da análise desses blocos básicos, o alocador pode observar a duração do tempo de vida das variáveis e assim saber quantas e quais vaáveis estão vivas em um determinado instante da execução do programa. Vale destacar que o LLVM possui uma biblioteca muito útil para realizar análises dos blocos básicos, no arquivo

```
../lib/CodeGen/RegAllocBase.h
```

pode-se encontrar funções extremamente úteis, as quais economizam muito tempo na implementação das estruturas do seu *pass*. Em

```
\../lib/CodeGen/MachineBasicBlock.cpp
```

está disponível um grande leque de funções para manipular os blocos básicos.

**Heurística.**

Com o conhecimento de como é realizado a entrada, parte-se para a implementação do que realmente compõe a heurística do alocador de registradores a ser implementada. Possuindo a heurística em mente, basta convertê-la para a estrutura do LLVM. Para auxiliar nesse empreendimento o uso do *Basic Register Allocator*

`(../lib/CodeGen/RegAllocBasic.cpp)`

fundamental e extremamente recomendado, também é importante observar o comportamento dos outros alocadores como o *Linear Scan*

`(../lib/CodeGen/RegAllocLinearScan.cpp)`

Baseando nos atuais alocadores é possível se ter uma noção de como o *pass* deve se comportar e como conseguir determinadas informaçõ̃es do LLVM que podem ser úteis para a sua heurística, como o sistema de *Coalescing* , *PhiElimitation* e *Spills*, presentes na maioria dos alocadores descritos.

**Saída.**

Como saída do alocador, tem-se as variáveis derramadas para a memória e determinadas variáveis associadas aos registradores. Uma função virtual importante que está contida no arquivo *RegAllocBase.h* é a `selectOrSpills()`, ela seleciona quais variáveis devem ser derramadas e quais devem receber os alocadores. O `RegAllocBase.h` possui a função `addMBBLiveIns()`, cuja finalidade é associar cada vida de uma variável escolhida a um registrador físico. Em relação aos derramamentos para a memória temos a função `spillInterferences()` ou `Spiller()`.

Na Figura 1 pode-se observar graficamente como seria composto o nosso *pass*. A classe do alocador seria criada pela função `FunctionPass*` que você declarou em `passes.h` explicado anteriormente. Ao ser criada a classe, o construtor é ativado, a maioria dos alocadores utilizam a função `Init()` do `RegAllocBase` para implementar o seu construtor. Quando necessário, o compilador invocará o seu compilador, e executará a função `runOnMachineFunction()`, esta função é o esqueleto do seu alocador, ela executa a entrada, o algoritmo da heurística escolhido e por fim associa os tempos de vida *LiveRanges* das variáveis aos registradores.

### 4.3.2 Recomendações de Estudos ao Implementar um Alocador

Para implementar um alocador de registradores é muito importante saber como o mesmo funciona na teoria, de forma a focar na heurística desejada. Mas também é necessário saber como ele funciona na prática dentro de um compilador. Um bom livro que trata de ambos assuntos é o *Modern Compiler Implementation in Java* [3], nele é detalhado a implementação de um alocador de registrador de forma bastante clara e exemplificada. É importante frisar que a lista de e−mail

`llvmdev@cs.uiuc.edu`

é bastante movimentada, nela diversos assuntos são discutidos e muitas dúvidas podem ser sanadas. O próprio site

`http://llvm.org/docs/`

possui diversos artigos e tutoriais para que seus usuários possam aproveitar ao máximo a infraestrutura do LLVM. Para ser mais específico sugiro foco nos seguintes tópicos:

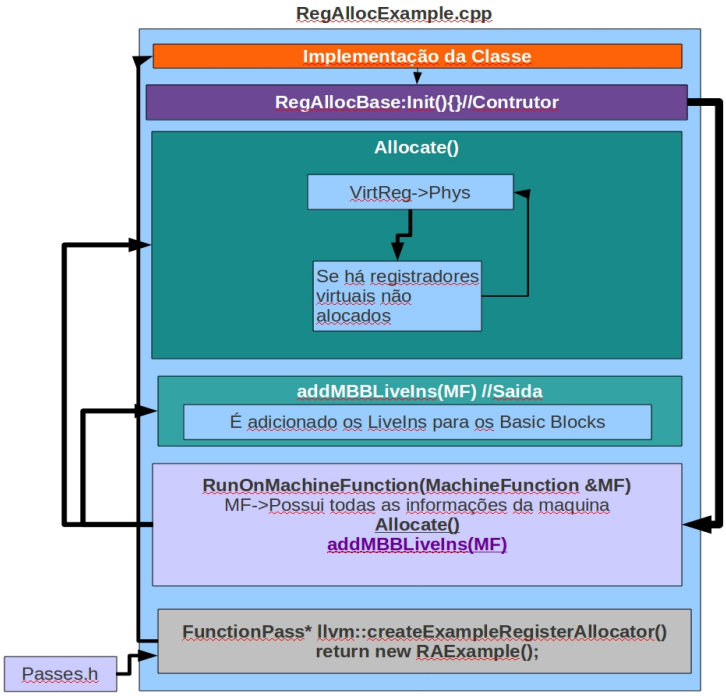


Figura 1: Exemplo da estrutura da classe.

- *Introduction to the LLVM Compiler System*
- *The LLVM Getting Started Guide*
- *Writing an LLVM pass*
- *Writing an LLVM Backend*
- *The LLVM Target-Independent Code Generator*

Por fim, e ao meu ver, o melhor material para estudos são os próprios códigos que compõem o LLVM, a sua clareza é indiscutível.

# 5 Conclusão

Nesse texto foram apresentados conceitos elementares em relação a como é um compilador, em específico o *LowLevel Virtual Machine* (LLVM). Foram discutidas as estruturas que um o LLVM possui e as características que motivam a escolhê-lo como base para implementar códigos para otimizações em compiladores. Discutiu-se como seria realizado a implementação de um alocador de registradores no LLVM. Inicialmente é preciso ter em mente como é o desenvolvimento de um *pass*, para que então possamos focar nos conceitos e diretrizes necessárias

na criação de um alocador de registradores. Esse documento está longe de ser um manual para o desenvolvimento de alocadores de registradores para o LLVM, mas a partir dele podemos ter uma real noção dos primeiros passos necessários para se tornar um bom desenvolvedor de uma ferramenta extremamente poderosa que é o LLVM.

## Referências


[1] Regallocbasic.cpp - basic register allocator. `http://llvm.org/docs/doxygen/html/RegAllocBasic_8cpp_source.html`. Accessed: 13/08/2011.

[2] Bigonha Mariza A. S Bigonha Roberto S. Ambrosio, L.L. Alocação global de registradores baseada em crescimento de domínios ativos e combinação de registradores. *8º Simpósio Brasileiro de Linguagens de Programação*, 1:157–171, 2004.

[3] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.

[4] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, 1992.

[5] Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[6] Brian J. Gough. *An Introduction to GCC*. Network Theory Ltd, 1st edition, 2005.

[7] Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[8] Allen Leung and Lal George. Static single assignment form for machine code. In *PLDI*, pages 204–214. ACM, 1999.

[9] Hanspeter Mossenbock and Michael Pfeiffer. Linear scan register allocation in the context of SSA form and register constraints. In *CC*, pages 229–246. LNCS, 2002.

[10] Fernando M Q Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *In Proceedings of APLAS 05, Asian Symposium on Programming Languages and Systems*, pages 315–329, 2005.

[11] Fernando Magno Quintao Pereira. *Register Allocation by Puzzle Solving*. PhD thesis, University of California, Los Angeles, 2008.

[12] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *TOPLAS*, 21(5):895–913, 1999.