O Potencial do Uso de Estimativas de Desempenho na Exploração de Conjuntos de Otimizações

Vanderson Martins do Rosario [1]
Anderson Faustino Silva [1]

Data de submissão: 17.05.2015
Data de aceitação: 01.11.2015

Resumo: Compiladores modernos tradicionalmente adotam estratégias de maior generalidade. Em contrapartida, para se aproveitar das especificidades de cada programa, surgem os compiladores iterativos. Esses exploram diferentes conjuntos de otimizações com o objetivo de encontrar o melhor para cada programa, maximizando uma função objetivo. Quando estamos buscando melhorias de desempenho, essa função é o tempo de execução. Uma forma prática de se obter o tempo de execução de um programa é executando o mesmo, porém a execução pode ser demorada tornando a exploração inviável. Para isso, uma solução é a estimativa de desempenho. Nesse artigo apresentamos uma ferramenta de estimativa de desempenho para auxiliar a tarefa de exploração do espaço de otimizações por compiladores iterativos. Diferentes técnicas foram avaliadas, onde foi possível mostrar que mesmo com estimativas aproximadas pode-se obter bons resultados. Além disso, as estimativas reduziram o tempo da exploração em 2,34x na seleção entre quatro conjuntos.

Abstract: Modern compilers prefer to be generic and retargetable. In contrast, to take advantage of program's specificities, there are iteratives compilers, which explore different optimization sets for each program using an objective function. When this search is done for the increase of performance, the objective function is the runtime. One practical way to get the runtime of a program is by executing it, but if the program runtime is expensive this can end up being impractical and performance estimation is an alternative. In this paper, we analyze the impact of the performance estimation precision over the exploration of optimization space and also present a tool to guide LLVM compiler over this task. A variety of estimation techniques are evaluated and we show that even with approximate estimates it is possible to have good results. Furthermore, the estimations were able to reduce in up 2.34x the selection time between four sets.

[1]Departamento de Informática, UEM, Bloco C56
{ra67620@uem.br} {anderson@din.uem.br}

1 Introdução

Atualmente existe um vasto, bem como crescente, leque de diferentes tipos de programas computacionais que são executados nos mais diferentes ambientes e com os mais diversos objetivos possíveis. Isso é acompanhado por uma grande diversidade de complexos processadores [1], cada um com suas especialidades. Essas características se tornam um grande problema para os compiladores [2][3], dada a inviabilidade em criar um compilador especializado em cada item no espectro da computação. Na verdade, pelo contrário, os compiladores acabam optando pela generalidade, gerando códigos não só para as mais diversas arquiteturas [4] mas também para os mais diversos programas e linguagens [5][6], com pouca ou nenhuma preocupação com suas especificidades.

Para contornar esse problema, a literatura propôe o uso de um compilador iterativo [7][8][9]. Esse, guiado pelo *feedback* de uma função objetiva, pode se adaptar e a cada iteração gerar um código com melhores resultados. Se repetido, esse processo deve convergir para a geração de um código extremamente otimizado. Além disso, dado o fato que diferentes programas quando aplicados a diferentes conjuntos de otimização geram melhorias de desempenho diferentes, ou seja, se cada programa possui um conjunto ótimo de otimizações, um compilador iterativo deveria convergir para esse conjunto. Portanto, um compilador iterativo encontra diferentes conjuntos de otimizações para diferentes programas em diferentes arquiteturas, levando em consideração suas especificidades.

Diversas funções objetivo são úteis atualmente na computação, entre elas podemos citar: o tempo de execução, o uso de recursos como a memória ou rede e o gasto de energia. Em nosso projeto em particular estamos interessados no tempo de execução dos programas. Para geração do valor dessa função objetivo, em outras palavras, o tempo de execução, normalmente se executa o código com alguma ferramenta de medição de tempo. Nesse caso a exploração do espaço exploratório pode levar um tempo considerável. Por exemplo, para um programa que demore 20 segundos para compilar e 300 segundos para executar, cada visita leva 320 segundos. Se a heurística realizar 20 visitas, a busca irá demorar 6400 segundos, ou seja, 1 hora e 47 minutos. Normalmente são feitas diversas visitas para diversos tipos de programas, o que pode acabar levando meses. Isso trás um problema que torna, inicialmente, inviável o uso de compiladores iterativos.

A fim de resolver esse problema alguns autores propuseram o uso de aprendizagem de máquina [10][11][12][13], de forma que o compilador aprenda quais otimizações são mais vantajosas, em uma fase de treinamento. Assim, todo o processo custoso da execução dos programas para obtenção das melhores otimizações, não são mais senti-

das pelo usuário final do compilador. Entretanto, ainda existe a fase de treinamento onde esse custo continua existindo.

Como alternativa para redução deste custo, alguns autores sugerem o uso de estimativa de desempenho, ao invés da execução do programa [3][14]. A estimativa pode ser ordens de magnitude mais rápida do que a execução de um programa, o que trás diversos benefícios para a exploração de conjuntos de otimizações. Contudo, uma estimativa precisa nas arquiteturas atuais é uma tarefa extremamente complexa, principalmente porque os processadores utilizam diversas técnicas para acelerar a execução de um código, tais como o uso de hierarquia de caches [15], instruções vetoriais [1] e previsão de desvios condicionais [16].

O presente artigo tem como objetivo apresentar uma ferramenta de estimativa de desempenho para a exploração de conjuntos de otimizações em compiladores LLVM. Os resultados experimentais demonstram que a ferramenta é rápida, além de ser capaz de auxiliar uma heurística na busca por bons conjuntos de otimizações.

O restante desse artigo está organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados; primeiro os relacionados a estimativa de desempenho e em seguida os relacionados a seleção de otimizações. A Seção 3 apresenta a ferramenta GESO, suas funcionalidades e sua arquitetura. A Seção 4, por sua vez, apresenta os resultados obtidos por experimentos realizados utilizando-se a ferramenta. Por fim, a Seção 5 apresenta as conclusões e também os trabalhos futuros.

2 Trabalhos Relacionados

Durante a revisão bibliográfica, no que se refere a exploração de conjuntos de otimizações para um programa, não foram encontrados artigos que comparassem o uso de heurísticas com estimativas com os que usassem o tempo real de execução. Em [14], por exemplo, é apresentado um estimador de desempenho para arquitetura Itanium, mas não é feita a comparação dos resultados com o uso dos tempos reais. Por conseguinte, dividimos os trabalhos relacionados em duas categorias, os que apresentam estudos sobre a estimativa de desempenho e os que apresentam formas de explorar conjuntos de otimizações.

2.1 Estimativa de Desempenho

O problema de estimar o tempo de execução de um programa é vastamente investigado por pesquisadores da área de sistemas operacionais em tempo real [17][18][19][20][21][22]. Sistemas em tempo real caracterizam-se pela presença de restrições no tempo de execução de uma tarefa. Para que o escalonador [23] tenha conhecimento

do tempo que levará para executar um processo é preciso estimar o tempo máximo de execução do mesmo. Esse problema é conhecido como *worst case execution time* ou simplesmente WCET.

Existem duas formas de se obter o WCET de um programa [17], a saber: por meio de análise dinâmica ou análise estática. A análise estática tem sido preferido entre os sistemas em tempo real e garante limites mais seguros. Porém, a complexidade de uma análise estática exata é alta. Encontrar o maior entre todos os caminhos possíveis em um código não é trivial, na verdade, esse é um problema NP-Difícil. Além disso, na análise estática é preciso modelar a arquitetura alvo [18]. Isso gera dois problemas: primeiro, torna a implementação vinculada à arquitetura, e segundo, os processadores modernos possuem diversas características que são difíceis de modelar e simular.

Por outro lado, se utilizarmos análise dinâmica, não existe a necessidade de modelar a arquitetura e sua complexidade [19]. Entretanto, certas informações não podem ser adquiridas de forma dinâmica e algumas técnicas estáticas são aplicadas para, por exemplo, medir a latência de acesso à memória com cache. Essas técnicas são chamadas de híbridas e permitem a implementação de grande parte do sistema de cálculo do WCET de forma independente de plataforma.

Em [20], os autores examinaram o problema de estimar o WCET de um programa por meio de análise estática em uma arquitetura específica. Para isso, os autores dividem o problema em duas partes: encontrar o fluxo de execução, entre os fluxos possíveis, que seja o pior caso e a modelagem da arquitetura alvo para simulação desse fluxo. Por fim, os autores apontam que a maior dificuldade se encontra na segunda parte, já que muitos processadores possuem *pipelines* e sistemas de memória cache [24], e esses são difícies de simular.

Ainda em [20] é evidenciado a importância de que as análises sejam feitas sobre código assembly, para que essas possam capturar todos os efeitos das otimizações do compilador e da implementação da arquitetura. Todavia, os autores alertam para o fato de que estimar o tempo de execução de um programa é indecidível e é equivalente ao problema da parada.

Em [21] é apresentado uma forma de estimar o WCET sem a necessidade da modelagem da arquitetura alvo, por meio de uma análise híbrida. Para isso, é derivado um conjunto de equações lineares a partir da execução de uma versão instrumentada do programa. A instrumentação do programa se dá pela inserção de instruções que capturem informações sobre a execução de blocos de código. Com suficientes entradas pode-se criar diferentes equações lineares para solucionar o sistema de equações. Com o sistema resolvido, obtemos o tempo gasto por cada bloco. Os autores levam em

consideração que o tempo de execução é igual a soma do tempo individual de cada bloco multiplicado pela sua frequência de execução. Ou seja, resolvidas as equações lineares saberemos o tempo de execução de cada bloco e com análise de fluxo podemos encontrar o fluxo de execução dos blocos que leve ao pior caso do tempo de execução. Portanto, multiplicando as frequências no pior caso com os tempos encontrados pelo sistema de equações encontramos o WCET.

Em [19] da mesma forma que em [21], os autores calculam o tempo de execução (Equação 1), como a soma do tempo de execução das instruções ($T_{execução}$) com o tempo de latência da memória ($T_{Memória}$).

$$T_{execução} = T_{Instruções} + T_{Memória} \tag{1}$$

O tempo de execução das instruções (Equação 2), é dado como no trabalho [21].

$$T_{Instruções} = \sum_{i=1}^{N_{blocos}} T_{B_i} * N_{B_i} \tag{2}$$

No cálculo da latência ao acesso à memória ($T_{Memória}$) (Equação 3), utiliza-se N_{C_i} como a quantidade de acessos a cache de nível C_i e L_{C_i} como a latência do acesso à cache nesse nível. Também, é utilizada a quantidade de acessos à memória principal como $N_{principal}$ e a latência de seu acesso como $L_{principal}$.

$$T_{Memória} = \sum_{i=2}^{N_{cache}} N_{C_i} * (L_{C_i} - L_{C_1}) + N_{principal} * L_{principal} \tag{3}$$

Em [22] é proposto uma ferramenta para cálculo do WCET baseada na representação intermediária da LLVM, o LLVM IR (LLVM IR, do inglês LLVM *Intermediate Representation*). A principal vantagem de uma abordagem sobre uma representação intermediária é que torna a ferramenta modular e fácil de migrar para diferentes arquiteturas. No trabalho, são fornecidos para as análises as descrições da plataforma alvo e a representação intermediária LLVM do programa. Essas então estimam o tempo de execução para esse programa.

2.2 Seleção de Otimizações

Em [14], os autores mostram que é imprescindível que compiladores apliquem otimizações (também chamadas de transformações) agressivas para obtenção de bons

desempenhos nos processadores modernos. Essas otimizações podem explorar diversas capacidades dos processadores [2], como, o acesso não uniforme a recursos, o paralelismo a nível de instruções, o uso de diversos níveis de memória e o suporte à execução especulativa. No entanto, o que acaba acontecendo é que as otimizações trocam benefícios em uma característica por outra. Como é o exemplo da otimização *loop unrolling* que aumenta o nível de paralelismo de instruções do código, mas por outro lado diminui o desempenho com a cache. Assim, torna-se clara a necessidade dos compiladores em determinar corretamente quando e onde aplicar cada otimização.

Entretanto, a complexidade em relacionar o impacto entre as otimizações inviabiliza decidir quais otimizações devem ou não ser aplicadas. Para resolver esse problema diversos autores propõem o uso de compiladores iterativos [3][25][10]. Esses geram diversas conjuntos de otimizações, usando algum método aleatório ou heurístico, e escolhem a melhor.

A busca por um conjunto de otimizações é um problema de otimização combinatória. Cooper et al. [3] propõem um compilador adaptativo que aprende a criar bons conjuntos de otimizações por meio da geração de conjuntos aleatórios. E para avaliar e selecionar o melhor conjunto eles usam uma função objetivo, no caso a quantidade estática de instruções.

Triantafyllis et al. [14] afirmam que mesmo que a abordagem do compilador iterativo sejá promissora, eles nem sempre são úteis para compilação de propósito geral e sofrem de um tempo inaceitável para compilação.

Em [25] e [10] são apresentados abordagens baseadas em aprendizado de máquina. Esses descobrem bons conjuntos de otimizações para um conjunto de programas de treino e criam uma base de conhecimento. Depois de treinada, essa base de conhecimento guia o compilador em qual conjunto de otimizações aplicar para um novo programa. Abordagens como essas, depois da etapa de aprendizagem, não precisam mais explorar o espaço de otimizações, porém ainda é necessário avaliar diversos conjuntos na fase de aprendizagem. Portanto, até mesmo essas abordagens podem se beneficiar de uma ferramenta de estimativa de desempenho.

3 GESO: Guia para Exploração de Conjuntos de Otimizações

O Guia para Exploração de Conjuntos de Otimizações[2] (GESO) é uma ferramenta para auxiliar e acelerar compiladores iterativos LLVM. O GESO possui três funcionalidades básicas, a saber:

[2]https://github.com/vandersonmr/GEOS

1. Cálculo de estimativas de desempenho para códigos LLVM;

2. Criação de cópias das informações de entrada (código LLVM e informações de *profiling*) para que as heurísticas possam se recuperar de suas ações (*backtracking*); e

3. Manter a coerência das informações e reconstruí-las, principalmente a frequência de execução, após a aplicação das otimizações.

Todas essas funcionalidades foram implementadas sobre a infraestrutura do Low Level Virtual Machine (LLVM) [26] e são aplicadas sobre a representação intermediária LLVM. O diagrama da Figura 1 mostra a arquitetura de um compilador iterativo com o GESO.

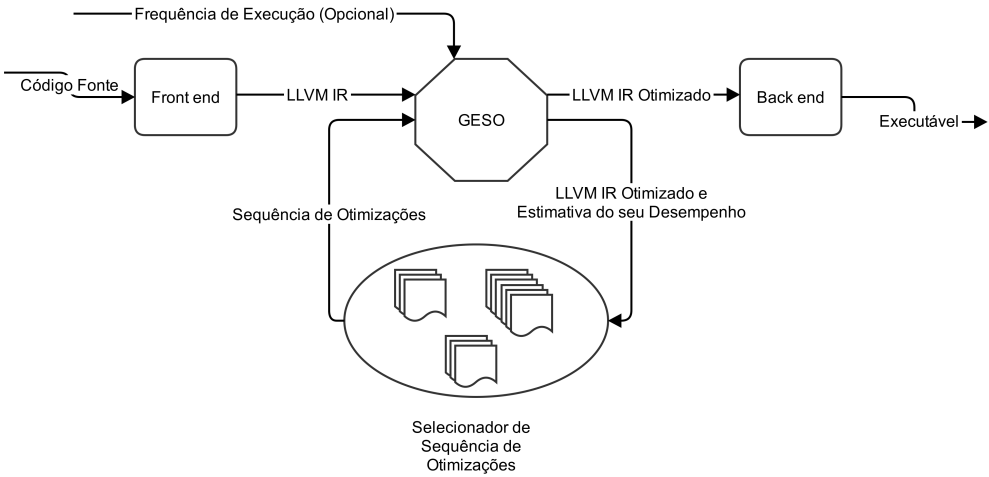


Figura 1: Arquitetura de um compilador iterativo com o GESO.

O *front end* e o *back end* são identicos ao de um compilador comum, sem a necessidade de alterações. O GESO é adicionado entre esses dois componentes mencionados e juntamente com uma heurística (Selecionador de Conjuntos de Otimização) escolhe qual o conjunto de otimização aplicar na representação intermediária e a aplica. Note que a heurística poderá fazer diversas chamadas ao GESO, realizando diversas iterações até escolher o conjunto final.

3.1 Estimativa de Desempenho

Foram implementados cinco métodos para cálculo do custo de execução, cada um como um módulo separado. São eles: o número de instruções; o custo de instruções; a quantidade de saltos incondicionais; o número de acessos a dados na memória; o custo de chamadas a funções externas; e cache de instruções.

3.1.1 Número de instruções Este método é a forma mais simples de se estimar o desempenho. Nela se leva em consideração apenas a quantidade de instruções LLVM executadas. O método implementado é similar a função objetivo implementada por Cooper et al [3], contudo, em nível intermediário.

A Equação 4 reflete o cálculo feito. Para cada função f é percorrido seus blocos básicos bb e somada a quantidade de instruções vezes a frequência de execução desse bloco básico.

$$E_{NúmeroInstruções}(f) = \sum_{bb}^{f} \sum_{i}^{bb} (1 * FrequênciaExecução(bb)) \tag{4}$$

Mesmo bastante simples, principalmente por levar em consideração apenas as instruções em nível intermediário, esta abordagem é interessante pelo fato de que em arquiteturas CISCs o assembly da LLVM IR reflete bastante o código da máquina.

3.1.2 Custo de instruções Este método leva em consideração informações sobre os diferentes custos para executar cada instrução na arquitetura em questão. Cada instrução da LLVM IR tem um peso para uma determinada arquitetura, o qual foi especificado na implementação do módulo de geração de código da arquitetura.

Além disso, também é levada em consideração a vetorização na geração do código de máquina, de forma que grupos de instruções podem ter seus pesos diminuídos. Principalmente pelo fato que na geração de código de máquina essas instruções deverão ser substituídas por instruções vetorias.

A Equação 5 reflete a implementação do cálculo da estimativa. Diferente do Número de Instruções, agora o custo das instruções não é mais constante e sim definido por uma função ($CustoExecução(i)$).

$$E_{Instruções}(f) = \sum_{bb}^{f} \sum_{i}^{bb} (CustoExecução(i) * FrequênciaExecução(bb)) \tag{5}$$

3.1.3 Quantidade de saltos condicionais Este método utiliza o total de saltos condicionais, ou seja, aqueles que por não poderem ser totalmente previstos podem causar a limpeza do *pipeline*. A frequência média com que os saltos não são previstos na arquitetura deve ser passada como parâmetro. As equações 6a e 6b mostram o cálculo.

$$CustoBranch(bb) = \begin{cases} 1 & \text{se } bb \text{ termina com um desvio condicional} \\ 0 & \text{caso contrário} \end{cases} \tag{6a}$$

$$E_{Branchs}(f) = \sum_{bb}^{f}(CustoBranch(bb) * FrequênciaExecução(bb) * MediaDeFalhas) \tag{6b}$$

3.1.4 Número de acessos a dados na memória Este método conta a quantidade de instruções no assembly gerada que acessam a memória, ou seja, que não acessam apenas registradores. Note que ela deve ser executada depois da alocação de registradores, já que usa as informações do uso dos registradores. As equações 7a e 7b mostram o cálculo.

$$CustoReg(i) = \begin{cases} 1 & \text{se } i \text{ acessa a memória} \\ 0 & \text{se } i \text{ apenas acessa registradores} \end{cases} \tag{7a}$$

$$E_{Registradores}(f) = \sum_{bb}^{f}\sum_{i}^{bb}(CustoReg(i) * FrequênciaExecução(bb)) \tag{7b}$$

3.1.5 Custo de chamadas a funções externas Este método calcula o custo das chamadas à funções externas por meio do *profiling* de execuções anteriores. É importante ressaltar que as otimizações não têm efeito em chamadas externas, pois tais são aplicadas antes do processo de ligação com bibliotécas externas, limitando o *speedup* máximo que as otimizações podem alcançar. As equações 8a e 8b mostram o cálculo do custo total das chamadas.

$$CustoCall(i) = \begin{cases} CustoDaFunção & \text{se } i \text{ for uma instruções call para uma função externa} \\ 0 & \text{caso contrário} \end{cases} \tag{8a}$$

$$E_{LibCall}(f) = \sum_{bb}^{f} \sum_{i}^{bb} (CustoCall(i) * FrequênciaExecução(bb)) \tag{8b}$$

As Figuras 2 e 3 mostram como é feita a instrumentação para geração do peso de cada chamada a funções externas. Esses pesos são salvos em um arquivo que servirá como entrada para a estimativa de desempenho. Na equação 8a esse peso é nomeado como *CustoDaFunção*.

```
if . then . i :
%81 = load %struct . _IO_FILE ** @stderr , align 8 , ! tbaa !13
%fputc3 . i = call i32 @fputc ( i32 10 , %struct . _IO_FILE * %81 ) #8
br label %for . inc . i
```

Figura 2: Bloco básico antes da instrumentação.

```
if . then . i :
%81 = load %struct . _IO_FILE ** @stderr , align 8 , ! tbaa !13
call void ( i32 , ...) * @start_measures ( i32 1)
%fputc3 . i = call i32 @fputc ( i32 10 , %struct . _IO_FILE * %81 ) #8
call void (...) * @get_results ()
br label %for . inc . i
```

Figura 3: Bloco básico depois da instrumentação.

3.1.6 Cache de instruções Por último, este método simula o comportamento da cache para tentar encontrar localidades espaciais e temporais na cache de instruções. Para isso foi implementado um algoritmo nos moldes propostos por Mueller e Whalley [27]. A análise é feita sobre o código de máquina e os endereços de memória das instruções são aproximados.

3.2 Frequência de Execução e Sua Manutenção

Para que a estimativa de desempenho possa ser feita depois da aplicação das otimizações é preciso que as informações de *profiling* sejam reajustadas. Muitas otimizações quando aplicadas a uma função modificam seu grafo de controle de fluxo, seja adicionando novos blocos básicos, removendo blocos básicos ou movendo-os.

Os dados sobre a frequência de execução são mantidos em dois níveis: diretamente nos blocos básicos e nas arestas dos saltos condicionais. A frequência dos blocos básicos pode ser calculado a partir das frequências dos saltos incondicionais, portanto, apenas essas são mantidas quando o código é persistido .

As informações sobre custo de chamadas externas e também a frequência dos saltos são armazenadas como metadados para que seja possível persisti-las em arquivo juntamente com a representação intermediária, fazendo com que as heurísticas possam criar pontos de retornos. A Figura 4 mostra como um metadado é persistido em um arquivo. Nesta figura podemos ver a frequência (última linha) de cada possível salto da instrução br (branch).

```
"22":                                             ; preds = %"21", %"16"
%.unr = phi i32 [ %.pre, %"16" ], [ %.unr.ph, %"21" ]
%indvars.iv.unr = phi i64 [ 1, %"16" ], [ %indvars.iv.unr.ph, %"21" ]
%9 = icmp ult i32 %5, 2
br i1 %9, label %"28", label %"23", !prof !18
...
!18 = !{!"branch_weights", i32 24900, i32 25000}
```

Figura 4: Bloco básico com metadados sobre a frequência de execução.

Os metadados são armazenados com o nome de branch_weights, já que esse é o padrão adotado pela LLVM. As transformações, quando identificam esses metadados, tentam mantê-los consistentes, mas isso nem sempre é possível. Por esse motivo, sempre depois da aplicação das otimizações, as informações sobre a frequência dos saltos são propagadas pelos blocos básicos do grafo de controle de fluxo.

3.3 Complexidade de Tempo

As estimativas possuem complexidade de tempo que são dependentes apenas do tamanho do código, ou seja, por mais que um código seja executado com uma enorme frequência, isso não afeta o tempo de execução da estimativa como afeta o tempo de execução.

Na subseção 3.1 foram apresentadas algumas formas de calcular o custo de um código e pudemos notar que todos são dependentes do número de blocos básicos e/ou número de instruções, já que suas somatórias são sobre esses itens. O fato do tempo das estimativas estar ligado apenas com o tamanho do código e não com a frequência faz com que um laço de tamanho k que é executado n vezes tenha complexidade de tempo para execução de O(kn), enquanto a complexidade de tempo para a estimativa seja de apenas O(k). Por esse motivo ocorre um ganho de desempenho para maior parte dos códigos no uso dessas estimativas, mesmo quando combinadas.

4 Resultados

O GESO foi avaliado em um computador que possui dois processador Xeon E5504, 24 GB RAM e executa o sistema operacional Ubuntu 12.04.5 LTS (GNU/Linux 3.13.0-45-generic x86_64). Todos os experimentos foram feitos com o processador travado em 1.86 GHz e executados no primeiro núcleo da primeira CPU. Esse núcleo foi travado via kernel para os experimentos.

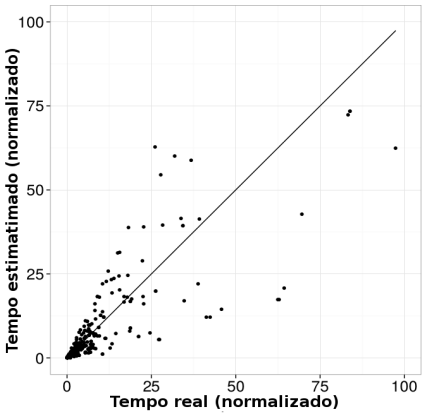
Foram utilizados 60 *benchmarks* compilados com o Clang 3.6 [28] com os conjuntos de otimização O0, O1, O2 e O3, totalizando 240 executáveis. Todos os *benchmarks* foram retirados do LLVM-test[3] e do Polybench[4].

4.1 Desempenho das Estimativas

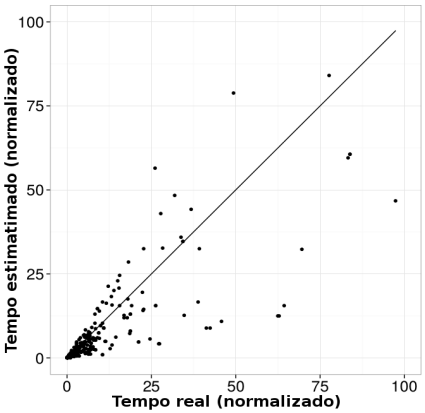
Para medirmos a precisão dos métodos de estimativa, executamos todos os *benchmarks*, medimos seus tempos de execução com a ferramenta time e estimamos o tempo de execução com cada método. Foi calculada então a correlação (Pearson) [29] entre o tempo real de execução e cada estimativa. Os resultados da correlação foram, 0,794 para Número de Instruções, 0,827 para Custo de Instruções, 0,847 para Número de Acessos a Dados na Memória, 0,762 para Saltos Condicionais e 0,789 para a Cache de Instruções.

Os gráficos das Figuras 5 e 6 mostram algumas dessas comparações visualmente. O eixo das abscissas representa o tempo real de execução em segundos. Já o eixo das ordenadas representa o desempenho estimado normalizado. Cada ponto representa a estimativa e tempo real de execução para um *benchmark*.

[3]https://github.com/llvm-mirror/test-suite
[4]http://web.cse.ohio-state.edu/ pouchet/software/polybench/

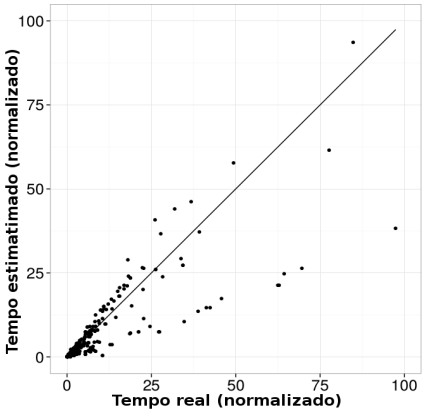


(a) Número de instruções

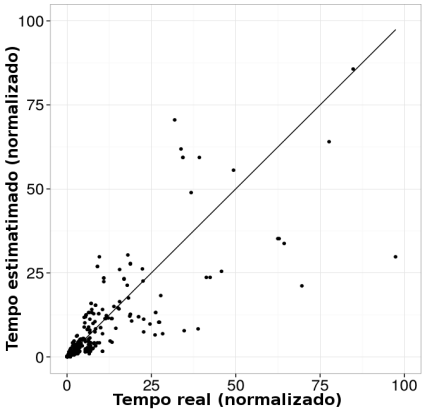


(b) Custo de instruções

Figura 5: Estimativa de desempenho e o tempo real para diversos *benchmarks* para diferentes métodos de estimativa.



(a) Número de Acessos a Dados na Memória



(b) Cache de Instruções

Figura 6: Estimativa de desempenho e o tempo real para diversos *benchmarks* para diferentes métodos de estimativa.

Os gráficos das Figuras 5 e 6 mostram a comparação entre as estimativas e os desempenhos reais de cada *benchmark*. Quanto mais alinhados melhor foram os resultados da estimativa. Uma linha indica o melhor resultado possível para servir como comparação. Quanto mais próximo da linha, mais próximo foi o resultado da estimativa em relação ao desempenho real. Como não existe relação forte entre a quantidade de saltos incondicionais (como também entre a quantidade de chamadas a funções externas) e o tempo de execução, não apresentamos seu gráfico.

Criamos uma nova estimativa por meio da soma ponderada entre diversas estimativas, como apresentado na Equação 9. Os valores de seus pesos foram definidos por meio de estimativas com diferentes valores. Foram escolhidos os valores que obtiveram as melhores estimativas, a saber: $c_1 = 2$, $c_2 = 1$, $c_3 = 0,8$, $c_4 = 0,05$. A correlação dessa nova estimativa e o tempo real de execução é de 0,853. O gráfico da Figura 7 compara os dois visualmente.

$$E_{Desemp.} = \sum_{f}^{Módulo} (c_1 E_{Mem}(f) + c_2 E_{Instruções}(f) + c_3 E_{Saltos}(f) + c_4 E_{Calls}(f)) \quad (9)$$

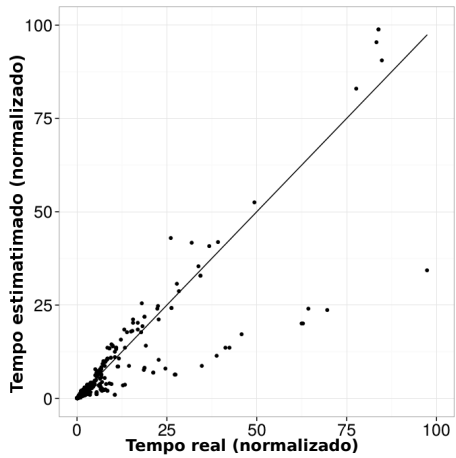
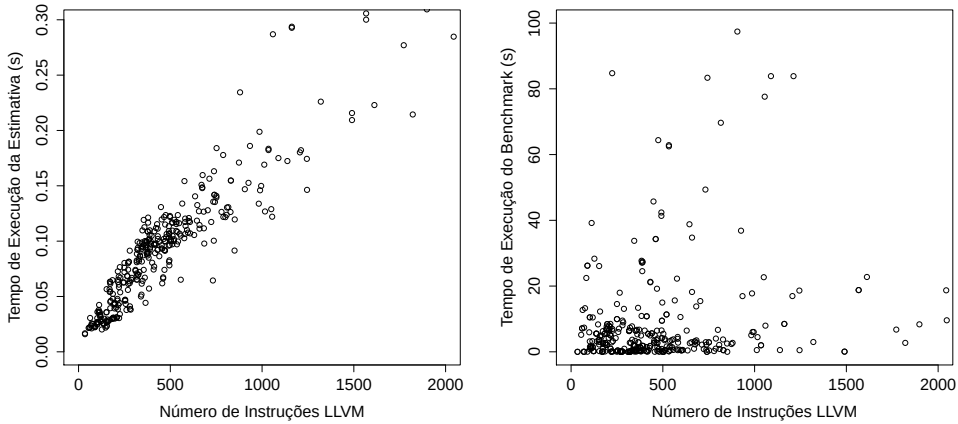


Figura 7: Correlação entre a estimativa do tempo de execução e o tempo real para diversos *benchmarks* utilizando a somatória ponderada de todas as técnicas de estimativa de desempenho.

Como nos gráficos das Figuras 5 e 6, a Figura 7 apresenta um gráfico com a comparação entre a estimativa de desempenho e o tempo real de execução. No entanto, nesse é utilizado os valores obtidos pela Equação 9 como estimativa. Note que os resultados foram superiores ao do uso de independentes abordagens. Isto indicada que a melhor estratégia, para estimar o tempo de exeucução de um determinado programa, é utilizar uma composição de abordagens.

Ainda, medimos o tempo para execução das estimativas para cada *benchmark*. Os gráficos da Figura 8 mostram o tempo para execução real e o tempo para a execução da estimativa de desempenho (eixo das ordenadas) de cada *benchmark* ordenados pelos seus números de instrução LLVM (eixo das abscissas).



(a) Estimativas

(b) Tempo Real

Figura 8: Tempo de execução da estimativa e dos *benchmarks* ordenados pela quantidade de instruções LLVM.

Além de visível a correlação entre o tempo de execução das estimativas e o tamanho do código (Figura 8a), na Figura 8b podemos perceber a grande diferença entre o tempo de execução do *benchmark* com o da estimativa, como tínhamos apontados na discussão sobre a complexidade de tempo na Subseção 3.3.

4.2 Seleção do Melhor Conjunto

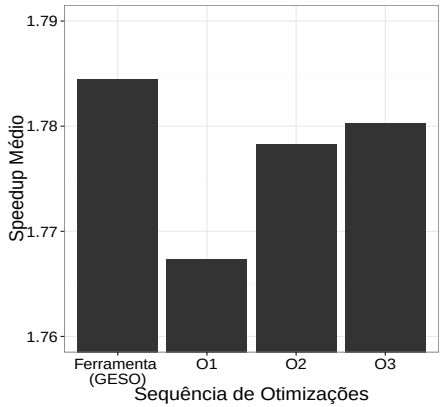
Implementamos uma ferramenta que, com auxílio do GESO, escolhe qual conjunto de otimizações é mais eficaz entre a O1, O2 e O3 para um código de entrada.

O código a seguir mostra o núcleo da ferramenta implementada em C++ utilizando o GESO. As configurações das estimativas são passadas por parâmetro e são colocadas na variável Opts. Inicialmente, é calculado a estimativa para o módulo LLVM (variável que contêm a representação intermediária do código de entrada) sem nenhuma otimização. Após são aplicados os conjuntos de otimizações O1, O2 e O3, sendo o módulo resultante é salvo na variável PO. A variável BestO mantêm qual conjunto obteve a melhor estimativa, enquanto BestCost mantém o valor da melhor estimativa.

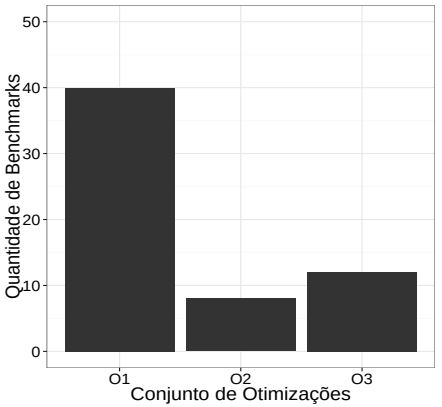
Algoritmo 1: Alinhamento Global de Needleman-Wunsch

```
Output: BestO
double BestCost = GEOS::analyseCost(PModule, Opts);
auto BestO = 0;
for i = 1; i ≤ 3; i++ do
    PassSequence Passes;
    Passes.setOLevel(i);
    ProfileModule *PO = GEOS::applyPasses(PModule, Passes);
    auto CostO = GEOS::analyseCost(PO, Opts);
    if BestCost > CostO then
        BestCost = CostO;
        BestO = i;
return BestO;
```

A Figura 9 mostra que essa ferramenta, sobre o conjunto de *benchmarks* utilizados, foi capaz de obter um *speedup* superior ao obtido por O1, O2 e O3.



(a) Comparação do *speedup* médio entre a ferramenta com o GESO e os conjuntos O1, O2 e O3.



(b) Frequência das escolhas da ferramenta com o GESO.

Figura 9: Resultados do uso das estimativas de desempenho para escolha dos conjuntos O1, O2 e O3.

Ainda, um resultado bastante interessante foi o fato que a maior parte das escolhas foram no O1. Para a maioria dos *benchmarks* para os quais o conjunto O1 foi escolhida, não houve melhoria no resultado. Entretanto, a diferença entre o *speedup* do O1 para o O2 e O3 era insignificante, praticamente idênticos. Isso mostrou que nem sempre é necessário aplicar otimizações mais complexas, já que essas nem sempre obtêm resultados significativos.

O uso de estimativas reduziu o tempo de resposta. A Tabela 1 apresenta os comparativos dos tempos. Note que para a execução da ferramenta de seleção dinâmica foi preciso executar todos *benchmarks* compilados em O0 para obter as informações da frequência de execução. Portanto, o tempo para execução da ferramenta foi de 17m + 56s, o que é $2,34$x mais rápido do que executar todos os 240 executáveis, e para cada escolher a melhor conjunto.

Tabela 1: Tempos de execução.

	Todos *benchmarks* (O0)	Todos *benchmarks* compilados em O0, O1, O2 e O3
Tempo para execução	17m	42m20s
Tempo para estimativa	17m + 31s	17m + 56s

É importante observar a qualidade dos resultados obtidos utilizando a ferramenta de estimativa. O uso de estimativas gerou um *speedup* médio de 1,7811, enquanto a execução real obteve um *speedup* médio de 1,7821. Isto indica que o uso de estimativas é tão preciso quanto uma execução real.

4.3 Conjuntos Aleatórios

Como é preciso executar o código uma vez para obter informações da frequência de execução, essa execução limita nosso *speedup* máximo, ou eficiência, como na lei de Amdahl. Esse limite diminui quando aumentamos a quantidade de conjuntos avaliados. Por isso, implementamos o Algoritmo 2, similar ao Algoritmo 1, no qual, agora, geramos 100 conjuntos aleatórios e selecionamos o melhor, não com o objetivo de encontrar boas sequências, já que é puramente aleatório, mas de avaliarmos o impacto das estimativas no desempenho de um compilador iterativo.

Algoritmo 2: Alinhamento Global de Needleman-Wunsch - Conjuntos aleatórios

```
Output: BestO
double BestCost = GEOS::analyseCost(PModule, Opts);
auto BestSet = null;
for i = 0; i ≤ 100; i++ do
    PassSequence Passes;
    Passes.randomize(30);
    ProfileModule *PO = GEOS::applyPasses(PModule, Passes);
    auto CostSet = GEOS::analyseCost(PO, Opts);
    if BestCost > CostSet then
        BestCost = CostSet;
        BestSet = PO;
return BestO;
```

Para isso, no Algoritmo 2, geramos 100 conjuntos aleatórios de 30 otimizações e selecionamos o que resulta o melhor desempenho por estimativa ou execução.

Como esperado a eficiência aumentou com o uso de mais estimativas, vide Tabela 2. Em nosso primeiro experimento selecionávamos entre 4 conjuntos o melhor e com o uso de estimativas tivemos um aumento de desempenho de 2,34x, portanto eficiência de 58%. Já ao selecionar o melhor entre 100 conjuntos, tivemos uma melhora de desempenho de 61,9x, ou seja, eficiência de 62%.

Tabela 2: Tempos de execução.

	Todos *benchmarks* compilados com os 100 conjuntos aleatórios
Tempo para execução	29h38m
Tempo para estimativa	17m + 11m42s

O *speedup* médio pelos conjuntos de otimizações encontrado na execução do Algoritmo 2 para os *benchmarks* foi de 1.75, inferior ao O1.

5 Conclusões e Trabalhos Futuros

Neste artigo apresentamos uma ferramenta capaz de auxiliar na busca por bons conjuntos de otimizações, por meio de estimativa de desempenho. Mostramos quais são as principais funcionalidades necessárias para essa tarefa e também como a estimativa de desempenho pode ser calculada.

Os resultados indicam que é possível reduzir o tempo de resposta de um compilador iterativo, principalmente quando esse necessíta de vários testes de desempenho. Além disso, mostramos que as estimativas não precisam ser complexas para obtermos resultados positivos. Isso demonstra a existência de espaço para avanços e pesquisa no uso de estimativas em compiladores iterativos.

Diversos aprimoramentos podem ser feitos sobre as estimativas de desempenho, como também no uso de heurísticas para identificar conjuntos de otimizações. Entre os trabalhos em andamento estão: (1) uso de técnicas avançadas já aplicadas na pesquisa de WCET; (2) uso heurísticas e mecanismos mais complexos para busca de conjuntos de otimizações; e (3) identificar quais informações são realmente importantes para responder com um certo grau de certeza se um código A é mais rápido que B.

Referências

[1] D. A. Patterson and J. L. Hennessy, *Computer Architecture*. Morgan kaufmann, 2011.

[2] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan kaufmann, 1997.

[3] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *Journal of Supercomputing*, vol. 23, p. 2002, 2001.

[4] M. Ganapathi, C. N. Fischer, and J. L. Hennessy, "Retargetable compiler code generation," *ACM Computing Surveys (CSUR)*, vol. 14, no. 4, pp. 573–592, 1982.

[5] F. Chow, "Intermediate representation," *Queue*, vol. 11, no. 10, p. 30, 2013.

[6] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 75–86, IEEE, 2004.

[7] E. Park, S. Kulkarni, and J. Cavazos, "An Evaluation of Different Modeling Techniques for Iterative Compilation," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, (New York, NY, USA), pp. 65–74, ACM, 2011.

[8] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle, "Iterative Compilation," in *Proceedings of the Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation*, (London, UK, UK), pp. 171–187, Springer-Verlag, 2002.

[9] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, and H. A. G. Wijshoff, "Iterative Compilation in Program Optimization," in *Proceedings of the Compiler for Parallel Computers*, pp. 35–44, 2000.

[10] E. D. de Lima, T. C. de Souza Xavier, and A. F. da Silva, "Seleção de transformações baseada em estatística," *Revista de informática Teórica e Aplicada*, vol. 20, 2013.

[11] T. M. Mitchell, *Machine Learning*. New York, NY, USA: McGraw-Hill, Inc., 1 ed., 1997.

[12] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using Machine Learning to Focus Iterative Optimization," in *Proceedings of the International Symposium on*

Code Generation and Optimization, (Washington, DC, USA), pp. 295–305, IEEE Computer Society, 2006.

[13] J. Thomson, M. O'Boyle, G. Fursin, and B. Franke, "Reducing Training Time in a One-shot Machine Learning-Based Compiler," in *Proceedings of the International Conference on Languages and Compilers for Parallel Computing*, (Berlin, Heidelberg), pp. 399–407, Springer-Verlag, 2010.

[14] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, (Washington, DC, USA), pp. 204–215, IEEE Computer Society, 2003.

[15] D. Levinthal, "Performance analysis guide for intel core i7 processor and intel xeon 5500 processors," 2009. https://software.intel.com/sites/products/collateral/hpc/vtune/ performance_analysis_guide.pdf.

[16] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*. Morgan kaufmann, 2013.

[17] J. Engblom, P. Altenbernd, and A. Ermedahl, "Facilitating worst-case execution times analysis for optimized code," in *Proceedings of the Euromicro Workshop on Real-Time Systems*, pp. 146–153, 1997.

[18] S. Schaefer, B. Scholz, S. M. Petters, and G. Heiser, "Static analysis support for measurement-based wcet analysis," in *Proceedings of the International Conference on Embedded and Real-time Computing Systems and Applications*, 2006.

[19] K. Yamamoto, Y. Ishikawa, and T. Matsui, "Portable execution time analysis method," in *Proceedings of the International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 267–270, 2006.

[20] Y.-T. S. Li, S. Malik, and A. Wolfe, "Efficient microarchitecture modeling and path analysis for real-time software," *16th IEEE Real-Time Systems Symposium*, 1995.

[21] M. Lindgren, H. Hansson, and H. Thane, "Using measurements to derive the worst-case execution time," in *Cheju Island, South Korea*, pp. 15–22, 2000.

[22] A. Machado, "Análise de tempo de execução utilizando llvm," tech. rep., Departamento de Informática e Estatística da Universidade Federal de Santa Catarina (UFSC), 2007.

[23] E. D. Jensen, C. D. Locke, and H. Tokuda, "A time-driven scheduling model for real-time operating systems.," in *RTSS*, vol. 85, pp. 112–122, 1985.

[24] W. Stallings, *Arquitetura e organização de computadores*. Pearson, 2010.

[25] S.-H. Hung, C.-H. Tu, H.-S. Lin, and C.-M. Chen, "An automatic compiler optimizations selection framework for embedded applications," in *Proceedings of the International Conference on Embedded Software and Systems*, ICESS '09, (Washington, DC, USA), pp. 381–387, IEEE Computer Society, 2009.

[26] "The llvm compiler infrastructure." Acesso em: http://llvm.org/.

[27] F. Mueller and D. B. Whalley, "Fast instruction cache analysis via static cache simulation," in *Proceedings of the 28th Annual Simulation Symposium*, pp. 105–114, 1994.

[28] "clang: a c language family frontend for llvm." Acesso em: http://clang.llvm.org/.

[29] R. M. Heiberger and B. Holland, *Statistical Analysis and Data Display: An Intermediate Course with Examples in S-Plus, R, and SAS*. Springer, 2004.