O IMPACTO DAS OTIMIZAÇÕES APLICADAS POR UM COMPILADOR OTIMIZANTE

Rafael Rodrigues dos Santos (PIC/UEM), Anderson Faustino da Silva (Orientador), e-mail: anderson@din.uem.br.

Universidade Estadual de Maringá / Centro de Tecnologia / Maringá, PR.

Ciências Exatas e da Terra. Ciência da Computação.

Palavras-chave: otimização, compilador, análise

Resumo

Códigos gerados por compiladores podem não ter a melhor qualidade possível, devido à dificuldade de obter a sequência ótima de instruções em meio a inúmeras possibilidades. Portanto, se faz necessário analisar o impacto de cada otimização provida pelo compilador e desta forma montar o melhor conjunto possível de otimizações. Nesse contexto, esta pesquisa teve por objetivo apresentar uma análise do impacto das otimizações disponibilizadas pela infraestrutura de compilação LLVM nos programas do Benchmark cBench. Tal análise possue o atrativo de fornecerem subsídios para sistemas que tentam descobrir o melhor conjunto de otimizações a ser aplicado em um determinado programa.

Introdução

Compiladores modernos aplicam diversas otimizações, algoritmos de transformação de código, as quais são capazes de melhorar a qualidade do código final; sem contudo modificar a semântica do programa (Sebesta, 2009).

Na compilação, um código pequeno escrito em linguagem de alto nível é transformado em um código maior, com uma sequência de instruções em linguagem de montagem. Várias sequências distintas podem ser geradas e a escolha de uma delas deve considerar o tamanho de tal sequência, a velocidade de execução, o consumo de energia e memória, entre outros fatores.

Embora os compiladores modernos forneçam diversas otimizações, a aplicação de todas não garante um bom resultado (pode até piorar o código), já que cada otimização espera que o código fonte possua um conjunto de características específicas.

Desta forma, não basta adotar uma dentre as várias estratégias de seleção de otimizações presentes na literatura (Cavazos e outros, 2007), é preciso analisar o impacto que cada otimização causará no código alvo.

Materiais e métodos

Otimizações aplicadas

Das otimizações fornecidas pela infraestrutura de compilação LLVM (LLVM Team, 2017), foram analisadas apenas aquelas classificadas como transformações, isto é, aquelas que modificam a estrutura do código sem alterar a semântica.

Características analisadas

Programas de computador podem ser analisados em função de características estáticas ou dinâmicas. As características dinâmicas dizem respeito ao comportamento do programa durante a execução, sendo tal comportamento dependente de dados de entrada e da plataforma em que a execução ocorre. Características estáticas independem de plataforma e consideram apenas as estruturas algorítmicas do programa, sem levar em conta a variação que pode ocorrer em função dos dados de entrada e da plataforma de hardware.
Esta pesquisa considerou as características estáticas, especificamente aquelas propostas por Namolaru (Namolaru e outros, 2010). O objetivo foi verificar quais dessas características são afetadas pela aplicação de cada otimização.

Os experimentos

Os experimentos foram realizados em um computador com processador Intel Core I7-3820, 3.60 GHz e 24G de memória RAM. Foram utilizados 10 programas do Benchmark cBench (Fursin, 2017), são eles: bitcount, qsort, susan_c, susan_e, susan_s, bzip2d, bzip2e, jpeg_c, jpeg_d, lame. Os resultados apresentados indicam o valor médio entre 5 execuções.
Inicialmente foi avaliado o ganho de desempenho de cada otimização. Depois, foi analisado como cada otimização altera as características do programa.

Resultados e Discussão

Para comparar o ganho de desempenho provocado por cada otimização, foi calculada a taxa de melhoria, que é dada por: melhoria = (*speedup* – 1) / 100, sendo *speedup* a razão entre o tempo de execução com o uso da otimização e o tempo de execução sem qualquer otimização.
Das 79 otimizações avaliadas, 73 não conseguem obter um desempenho superior a 5% de melhoria média. Apenas 6 otimizações conseguem uma melhoria superior a 5%, a saber: gvn (8,22%); slp-vectorize (11,41%); scalarrepl (95,52 %); scalarrepl-ssa (97,72%); sroa (98,27%); e mem2reg (102,49%).

Tais resultados indicam que em um universo de dezenas de otimizações, uma quantidade pequena consegue alcançar um desempenho considerável. Além disto, os resultados também indicam que o ideal é aplicar um conjunto de otimizações e não uma única otimização isoladamente. Contudo, é importante considerar que o bom desempenho de uma determinada otimização nem sempre é constante para todos os programas.
Analisando as características afetadas positivamente pelas otimizações (Tabela 1) é possível perceber que as seis melhores otimizações reduzem a quantidade de instruções do programa, principalmente aquelas que fazem acesso à memória.

Tabela 1 – Características afetadas pelas melhores otimizações

Otimização	Características Reduzidas
gvn	nof_cfg_edges, nof_cfgcrit_edges, nof_bb, nof_1suc_bb, nof_pred_bb, nof_l15inst_bb, nof_assign_inst, nof_binop_int_inst, nof_binop_bitw_inst, nof_uncondbr_inst, nof_load_inst, nof_1pred_bb, nof_inst
slp-vectorize	nof_binop_int_inst, nof_load_inst
scalarrepl	nof_ge15le500inst_bb, nof_inst, nof_assign_inst, nof_memory_adress_inst, nof_load_inst, nof_store_inst,
scalarrepl-ssa	nof_ge15le500inst_bb, nof_inst, nof_assign_inst, nof_memory_adress_inst, nof_load_inst, nof_store_inst,
sroa	nof_ge15le500inst_bb, nof_inst, nof_assign_inst, nof_memory_adress_inst, nof_load_inst, nof_store_inst
mem2reg	nof_ge15le500inst_bb, nof_memory_adress_inst, nof_load_inst, nof_store_inst

Programas que manipulam uma grande quantidade de dados tendem a realizar diversos acessos à memória, o que degrada o desempenho. O ideal é que os dados acessados pelo programa sejam armazenados completamente na cache do processador, assim o custo de leituras à memória não existirá e consequentemente o tempo de execução do programa será reduzido.
Os programas analisados possuem diversos dados armazenados em memória, desta forma uma boa estratégia de compilação é utilizar otimizações que otimizem o acesso à memória tais como: scalarrepl, scalarrepl-ssa, sroa, e mem2reg. Portanto é possível concluir que programas com estruturas de acesso à memória serão beneficiados pelas otimizações: gvn, slp-vectorize, scalarrepl, scalarrepl-ssa, sroa, e mem2reg.
Outro ponto importante a ser observado é o fato da efetividade de uma otimização estar em sua capacidade de reduzir a quantidade de instruções de um programa. As outras otimizações analisadas neste projeto (descartando as 6 avaliadas nesta seção), não possuem uma efetividade neste quesito; embora sejam efetivas quando utilizadas em conjunto com outras otimizações (como é o caso do níveis de otimização). Portanto,

também é possível concluir que as otimizações gvn, slp-vectorize, scalarrepl, scalarrepl-ssa, sroa, e mem2reg são boas otimizações para reduzir a quantidade de instruções de um programa.
Por fim, outra informação que pode ser obtida dos resultados é o fato de programas com operações binárias se beneficiarem da aplicação da otimização gvn.

Conclusões

A escolha do conjunto de otimizações adequado na compilação de programas é uma tarefa difícil, pois cada programa tem características particulares que podem ser impactadas positiva ou negativamente pelas otimizações. Assim, tais características devem ser estudadas cuidadosamente para então escolher as otimizações mais adequadas.
Esta pesquisa mostrou que poucas otimizações produzem melhorias consideráveis, sendo que as mais notórias são aquelas que reduzem a quantidade de instruções que acessam a memória. Desta forma, programas que manipulam muitos dados armazenados em memória têm um desempenho melhor quando submetidos a esse tipo de otimização.

Agradecimentos

Ao Programa de Educação Tutorial (MEC/SESu) pelo fomento e pelo apoio institucional.

Referências

CAVAZOS, J., FURSIN, G., AGAKOV, F., BONILLA, E., O'BOYLE, M. F. P., TEMAM, O. **Rapidly Selecting Good Compiler Optimizations Using Performance Counters**. In Proceedings of the International Symposium on Code Generation and Optimization (Washington, DC, USA, 2007), IEEE Computer Society, pp. 185–197.

FURSIN, G. (2017). **Collective Benchmark - Enabling realistic benchmarking and optimization**. http://ctuning.org/cbench. Último acesso em 15/03/2017.

LLVM Team. **The LLVM Compiler Infraestructure**. www.llvm.org. Último acesso em 15/03/2017.

NAMOLARU, M., COHEN, A., FURSIN, G., ZAKS, A., FREUND, A. (2010). Practical **aggregation of semantical program properties for machine learning based optimization**. In Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems CASES '10 (pp. 197–206). New York, NY, USA: ACM.

SEBESTA, R. W. **Concepts of Programming Languages**, 9th ed. Addison-Wesley Publishing Company, USA, 2009.