Seleção de Transformações Baseada em Estatística

Ewerton Daniel de Lima [1]
Tiago Cariolano de Souza Xavier [1]
Anderson Faustino da Silva [1]

Resumo: Dentre diversas transformações providas por um compilador é um desafio, até mesmo para o mais experiente programador, saber quais gerarão o melhor código alvo para determinado código fonte. Neste contexto, o desenvolvimento de um seletor automatizado de boas transformações é um desafio nos dias atuais. O objetivo deste artigo é descrever uma abordagem estatística para selecionar boas transformações para um determinado código fonte. A abordagem estatística apresentada neste trabalho, apesar de simples, foi capaz de fornecer bons ganhos de desempenho, se comparados a trabalhos relacionados. Em um conjunto com dez programas, o *speedup* médio alcançado, em relação à abordagem mais agressiva da `LLVM` foi de $1,0514$ indicando um ganho de desempenho de $5,14\%$. No pior caso, a abordagem proposta obteve um *speedup* igual a $1,0$ e, no melhor caso, de $1,19$, indicando um ganho de desempenho de 0% e 19%, respectivamente.

Abstract: Among several transformations provided by the compiler, it is a challenge, even for the most expert programmer, to know which ones will generate the best target code for a particular input source code. Considering the transformation selection problem, the goal of this paper is to describe a statistical approach to automatically select good transformations for a source code. Although the statistical approach presented in this work may be considered simple, it was able to achieve good results compared to related works. Considering the most agressive optimization level of `LLVM`, the average speedup was $1,0514$, in a set of ten programs. These data indicantes a performance gain of $5,14\%$. In the worst case this approach obtained a speedup equal to $1,0$ and in the best case it reached $1,19$, indicating a performance gain of 0% e 19%, respectively.

1 Introdução

As transformações aplicadas por um compilador são pertinentes à etapa de otimização no processo de tradução de código fonte para código alvo e possuem o objetivo de modificar o código para que ocorra um ganho de desempenho.

[1] Universidade Estadual de Maringá
Departamento de Informática
Avenida Colombo, 5790 - Bloco C56 - Maringá/PR - CEP 87020-900
{ewertondanieldelima@gmail.com, tiago.cariolano@gmail.com,
anderson@din.uem.br}

Os compiladores que aplicam transformações ao código fonte geralmente fornecem dezenas delas para que possam ser escolhidas pelo usuário, pois nem toda transformação modificará o código de maneira benéfica, podendo, em alguns casos, até mesmo ocasionar uma perda de desempenho na qualidade do código gerado. Tal perda pode ocorrer devido as características do código fonte não se adequarem às características específicas de cada transformação.

Cada código fonte terá um conjunto de transformações que resultará no melhor desempenho. No contexto atual, desempenho depende do objetivo a ser alcançado, podendo ser redução do tamanho do código, redução do consumo de energia, otimização dos acessos à memória ou ainda a redução do tempo de processador.

Uma forma de garantir o resultado ótimo seria aplicar todas as possibilidades de conjuntos de transformações, avaliar cada caso e então escolher a melhor configuração possível. No entanto, não é difícil perceber que tal metodologia é inviável em termos de tempo de processamento. Considerando os conjuntos de transformações possíveis, a quantidade de avaliações para um determinado código seria de 2^n, onde n é o número de possíveis transformações a serem aplicadas. Assim, um código compilado com o Clang [1], que oferece a possibilidade de aplicação de 61 diferentes transformações, necessitaria de:

$$\#\mathcal{P}\left(S\right) = 2^n = 2^{61} = 2.305.843.009.213.693.952 \text{ avaliações}$$

($\#\mathcal{P}\left(S\right)$ se refere à cardinalidade do conjunto das partes de S, que é definida na literatura como 2^n, onde n é o número de elementos de S [2]).

Isto significa que para garantir o melhor subconjunto de transformações fazendo a experimentação de todas as combinações possíveis de um programa que compila e executa em 1 segundo seriam necessários cerca de 73 bilhões de anos de processamento.

Na tentativa de procurar soluções viáveis para o problema da seleção de transformações, várias abordagens foram propostas na literatura. Entre essas abordagens estão: algoritmo genético [3], aprendizagem de máquina [4], busca exaustiva [5], seleção aleatória [6], além de técnica estatística [7].

Neste contexto, o objetivo principal deste artigo é propor e avaliar um mecanismo de seleção de transformações baseado em similaridade estatística, chamado de `STBE`. A hipótese utilizada é que programas com características semelhantes tendem a ter bons conjuntos de transformações semelhantes. Partindo da premissa que esta hipótese seja verdadeira é possível construir uma base com informações sobre as características dinâmicas de vários programas e associar tais informações a conjuntos de transformações, os quais são melhores do que aquele utilizado pela estratégia mais agressiva (o nível mais alto de otimização, chamado aqui de *baseline*) de aplicação de transformações. Dispondo de uma base com tais

características, é possível buscar nela quais transformações são potencialmente boas candidatas para um determinado programa de entrada, utilizando similaridade estatística.

A principal contribuição deste artigo é demonstrar que é possível construir um mecanismo simples de seleção de transformações que proporcione *speedup*, em relação ao *baseline*, para a maioria dos programas aos quais é aplicado. O mecanismo desenvolvido é considerado simples por envolver apenas similaridade estatística, ao invés de estratégias mais elaboradas como: algoritmos genéticos e/ou algoritmos de aprendizagem de máquina.

Os resultados alcançados com `STBE` para o *benchmark* `SNU NPB Serial` demonstram que a estratégia utilizada é promissora, sendo capaz de encontrar bons *speedups* comparados ao *baseline* da `LLVM` [1, 8] e aos resultados apresentados na literatura [4, 9].

O texto a seguir está organizado como segue. A Seção 2 apresenta trabalhos realizados na área de seleção de transformações. A Seção 3 descreve o mecanismo para seleção de transformações baseado em similaridade estatística, o qual é proposto neste artigo. A Seção 4 apresenta a validação da hipótese formulada neste artigo. Por fim, a Seção 5 apresenta as conclusões, como também os trabalhos futuros.

2 Trabalhos Relacionados

Segundo Kulkarni *et al* [10], em alguns contextos, a abordagem de busca exaustiva pode ser utilizada. Por exemplo, em sistemas embarcados o objetivo tradicional é reduzir o tamanho do código, devido à limitação de espaço dos dispositivos. Assim, se forem consideradas apenas as transformações que influenciam no tamanho de código, pode ser viável fazer uma busca exaustiva pelo melhor resultado. De qualquer forma, a abordagem de busca exaustiva está restrita a contextos bem limitados.

A busca exaustiva foi utilizada no trabalho de Massalin [11]. O objetivo do trabalho de Massalin não é a seleção de transformações, mas a exploração de todas sequências de instruções possíveis de modo a encontrar qual é a menor, visando minimizar o tamanho do código gerado. Diferente deste trabalho, `STBE` é empregado para encontrar o melhor conjunto de transformações para um determinado código. Contudo, `STBE` é similar ao trabalho de Massalin por utilizar um espaço de busca sobre o qual o sistema irá operar.

No contexto específico de seleção de transformações, um estudo da aplicação de transformações para redução de tamanho de código pode ser visto no trabalho de Foleiss *et al* [12]. Restritos às aplicações de redes de sensores sem fio, os autores utilizaram a busca exaustiva para analisar o impacto das transformações no tamanho de código gerado. Para reduzir o espaço de busca os autores agruparam transformações similares, criando desta forma conjunto de transformações e então avaliando o agrupamento de tais conjuntos. Diferente do trabalho de Foleiss e Silva, o objetivo de `STBE` é encontrar o melhor conjunto de transformações

que reduza o tempo de execução e não é utilizada uma busca exaustiva, mas sim uma estratégia baseada em similaridade.

A questão da ordem de aplicação das transformações é explorada por Kulkarni *et al* [5] também com uma abordagem exaustiva. Este trabalho procurou demonstrar que o espaço de busca para encontrar a melhor ordem de aplicação das transformações não é tão grande se forem aplicadas algumas técnicas juntamente com a busca exaustiva. STBE não aborda a questão da ordem das transformações, porém, como o trabalho de Kulkarni *et al*, STBE tenta demonstrar que é possível obter bons resultados utilizando um espaço de busca pequeno.

Um trabalho que utiliza uma abordagem aleatória pode ser visto no desenvolvido por Lau *et al* [6]. Neste trabalho, implementado no contexto de máquinas virtuais [13] são geradas n versões de uma região de código compilada. Durante a execução do programa o sistema escolhe aleatoriamente uma das versões geradas. Na finalização do sistema, informações de execução são armazenadas, as quais auxiliarão um mecanismo estatístico na seleção da melhor versão de código para uma dada região. STBE é similar ao trabalho de Lau pelo fato de também utilizar uma abordagem aleatória para alcançar o objetivo desejado.

Os trabalhos de Cavazos *et al* [4] e Park *et al* [9] utilizam uma abordagem aleatória para gerar um espaço de busca inicial ao qual serão aplicados algoritmos de aprendizagem de máquina para escolher as melhores transformações. Em ambos trabalhos, cada *workload* utilizado para treino é executado 500 vezes com conjuntos de transformações escolhidos aleatoriamente e os resultados melhores que o nível mais agressivo de compilação são armazenados. Em seguida, um algoritmo de aprendizagem de máquina utiliza as informações armazenadas para decidir quais transformações aplicar ao código de entrada. STBE é similar a estes dois trabalhos pelo fato de também utilizar uma abordagem aleatória para gerar um espaço de busca inicial. Contudo, difere destes trabalhos pelo fato de utilizar similaridade para encontrar o melhor conjunto de transformações para um determinado código e não um algoritmo de aprendizagem de máquina.

Uma abordagem com técnicas estatísticas pode ser vista no trabalho de Haneda *et al* [7], o qual utilizou o teste de hipótese de Mann-Whitney [14] para verificar se determinada transformação afeta ou não, significativamente, o código gerado, permitindo assim decidir se é viável ou não selecioná-la. STBE é similar ao trabalho de Haneda no tocante a utilizar estatística para escolher transformações para um determinado código. Contudo, enquanto o trabalho de Haneda tem como objetivo identificar se uma transformação afeta o desempenho de um código, STBE tem por objetivo encontrar o melhor conjunto de transformações que afeta um determinado código.

3 STBE - Seletor de Transformações Baseado em Estatística

O Seletor de Transformações Baseado em Estatística (STBE) tem por objetivo ser um mecanismo capaz de encontrar um bom conjunto de transformações para um programa específico, com uso de similaridade estatística e probabilidade. Tal sistema foi projetado em uma arquitetura constituída por três componentes principais, a saber: uma base de informações; um mecanismo seletor e avaliador; e um verificador de similaridade. A Figura 1 apresenta como tais componentes estão interligados.

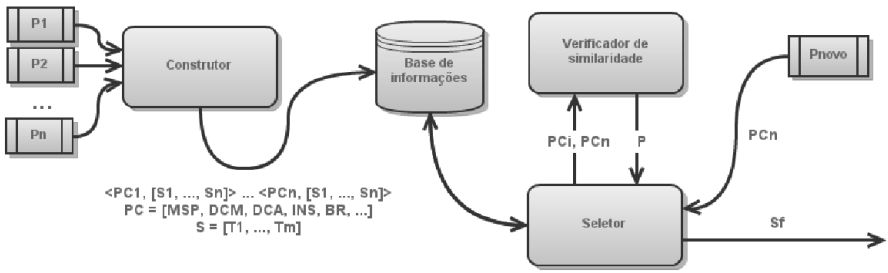

Figura 1. Arquitetura do STBE

Inicialmente, o construtor gera uma base de informações a partir de N programas ($P_1 ... P_n$). Esta base é composta por pares ordenados $< PC_1, [S_1, ..., S_n] > ... < PC_n, [S_1, ..., S_n] >$, onde PC_i é o conjunto de *performance counters* do programa e S_i é um bom conjunto de transformações para o programa P_i. Para o programa a ser avaliado (P_{novo}), o seletor coleta seus *performance counters* (PC_n) e os compara com os *performance counters* existentes na base, com o auxílio do modelo de similaridade. Tal modelo fornece então ao seletor um mecanismo de identificar quais programas da base são similares ao programa avaliado e desta forma selecionar o melhor conjunto de transformações possível.

3.1 Construção da Base de Informações

A base de informações é uma lista que armazena pares $< P, T >$, onde P é um programa e T é um conjunto de transformações. Um programa é representado por um conjunto de *performance counters*, sendo $P = \{p_1, p_2, ..., p_i, ..., p_n\}$, onde cada p_i é um *performance counter* diferente. Os elementos que pertencem ao conjunto T são transformações, assim, $T = \{s_1, s_2, ...s_m\}$, onde cada s_i é um conjunto de transformações, sendo m a quantidade máxima de transformações avaliadas.

Todos os *performance counters* disponíveis na arquitetura alvo são envolvidos no pro-

cessos de seleção. Assim, o tamanho de P, para um dado programa, é fixo. Por outro lado, o conjunto de transformações tem tamanho variado (configurado de maneira não-determinística) para cada programa e seus elementos (transformações) são escolhidos aleatoriamente. Dois conjuntos são particularmente especiais: *baseline* e $O0$. O primeiro é constituído pelas transformações disponíveis no nível mais agressivo de otimização e o segundo, é aquele que não contém nenhuma transformação. O conjunto $O0$ é utilizado para coletar os *performance counters*. Isso é feito para que nenhuma transformação afete o comportamento do programa em sua essência. O conjunto *baseline* é utilizado como um parâmetro de comparação. O objetivo é encontrar um conjunto que forneça resultados melhores que *baseline*, que apesar de ser a abordagem mais agressiva nem sempre é a que fornece o melhor resultado.

Para geração da base, são necessários *benchmarks* para serem executados e servirem como um "conjunto de treino". Para cada *benchmark*, as seguintes etapas devem ser executadas:

Coleta dos *performance counters* O programa a ser inserido na base é compilado sem nenhuma transformação e seus *performance counters* são coletados. Esse processo pode exigir mais do que uma execução do programa, devido ao fato da necessidade de N execuções para que todos os *performance counters* possam ser coletados, pois não existe como multiplexar todos em uma única execução.

Coleta do tempo de execução de *baseline* O programa é compilado com este nível de transformação e é executado. Seu tempo de execução é coletado e armazenado.

Compilação com transformações aleatórias O programa é compilado com um conjunto de transformações gerado aleatoriamente, é executado e seu tempo de execução é comparado com o de *baseline*. Se este conjunto gerar um tempo de execução menor ou igual a *baseline*, ele é inserido na base e associado aos *performance counters* coletados anteriormente, como também ao *speedup* alcançado. Essa etapa é repetida k vezes. É importante observar que neste processo não são tomados conjuntos repetidos e, portanto, exatamente k conjuntos diferentes são avaliadas para cada programa.

3.2 Mecanismo de Seleção

A geração da base e a seleção de transformações da mesma são processos independentes. No entanto, o processo de seleção está associado a uma base, que por sua vez pode ser trocada. De fato o seletor pode executar com qualquer base e até mesmo sem base, apesar desta não ser uma estratégia eficiente, pois não utilizar base, significa que o `STBE` irá explorar apenas busca aleatória.

Assim, antes de iniciar a seleção de transformações, o `STBE` deverá ter uma base associada a ele. Em seguida, ele recebe como entrada um código fonte e seleciona um bom

conjunto de transformações para ser usado durante a compilação do código fonte. O conjunto selecionado será um dos que estão na base, associado a algum programa similar ao programa de entrada ou um conjunto gerado aleatoriamente, nos casos em que não há informação suficiente na base.

O processo de seleção se inicia com a compilação do código de entrada sem aplicação de nenhuma transformação. O programa gerado é então executado e são coletados seus *performance counters*.

O próximo passo do processo de seleção é procurar na base de informações quais programas são similares ao programa de entrada, dados seus *performance counters*, dada a função $sim(P_1, P_2)$ que verifica a similaridade entre dois conjuntos de *performance counters* (P_1 e P_2). Em `STBE`, o seletor percorre todos os pares $< P_i, T_i > \in B$ (em que B é a base), para calcular a similaridade do programa a ser avaliado com os programas existentes na base. Isto é, $sim(P_i, P_n)$ é calculado para cada $i = 0...m$, onde m é o número total de programas armazenados na base. Esse cálculo resulta em uma distribuição de probabilidades. Assim, todos programas na base são candidatos à seleção.

O próximo passo é decidir quais programas da base serão selecionados para avaliar seus conjuntos de transformações. Cada programa da base neste momento possui uma probabilidade associada a ele, calculada pelo verificador de similaridade. Portanto, a probabilidade de um programa da base ser escolhido para serem utilizadas suas transformações é dada pelo índice de similaridade dele com o programa a ser avaliado, para o qual se quer encontrar boas transformações.

Um subconjunto de programas da base é selecionado probabilisticamente, baseando-se nas probabilidades calculadas pelo verificador de similaridade. Este processo resulta em uma base $B' \subseteq B$, que contém apenas os programas selecionados. Por fim, `STBE` explora esta nova base (B'), avaliando os conjuntos de transformações presentes nela, com o objetivo de encontrar o conjunto que maximiza o desempenho do programa avaliado.

3.3 Normalização dos *performance counters*

Para facilitar a análise da similaridade entre dois conjuntos de *performance counters*, existe a necessidade de normalizá-los. Há uma dificuldade em comparar programas sem normalizar os seus *performance counters*, pois quando há uma diferença entre seus tempos de execução, o que geralmente ocorre, há também uma grande diferença entre seus *performance counters*. Por exemplo, um programa que executa por horas geralmente terá muito mais falhas na *cache* do que um outro que executa por alguns minutos. Portanto, para facilitar a comparação, em `STBE` os *performance counters* são normalizados pelo número total de instruções. Seja $P = \{p_1, p_2, ..., p_i, ..., p_n\}$ o conjunto de *performance counters* sem normalização, o conjunto normalizado é dado por:

$$P_{normalizado} = \{\forall p \in P \bullet \frac{p}{TOT\_INS}\}$$

Onde TOT_INS é o total de instruções executadas.

Com essa normalização, o valor p de erros em predição de desvio, por exemplo, pode ser interpretado da seguinte maneira: *para cada instrução, houve* p *erros em predição de desvio*.

3.4 O Verificador de Similaridade

Medir a similaridade entre dois programas com exatidão não é uma tarefa trivial. Os programas são executados, em geral, em um ambiente que envolve sistema operacional [15] e um *hardware* específico [16]. As características específicas do sistema operacional e do *hardware* podem influenciar na execução do programa. Portanto, há uma dificuldade em calcular a similaridade entre dois programas.

Uma abordagem para medir a similaridade é avaliar o código fonte de cada programa, considerando informações estáticas tais como: número de funções, número de laços de repetição, número de chamadas de função, números e tipo de variáveis, dentre outras. Essa técnica pode ser denominada estrutural, ou seja, não está ligada com a execução do programa, mas sim com a sua estrutura. No entanto, ainda que dois programas sejam considerados similares na estrutura do código, sua execução pode ser muito diferente.

Outra abordagem para avaliar a similaridade entre dois programas é utilizar informações da execução do programa, utilizando *performance counters*. Com essa metodologia, a similaridade é baseada no processo executado, ao invés de no programa estático. No STBE, essa foi a abordagem utilizada para medir a similaridade entre os programas.

A implementação do verificador de similaridade utilizado no processo de seleção de transformações é baseado em técnicas estatísticas.

Em estatística, a similaridade entre dois conjuntos amostrais pode ser medida por meio do coeficiente de Jaccard [14], uma medida definida como a razão entre o tamanho da interseção e o tamanho da união dos dois conjuntos. Formalmente, dados dois conjuntos A e B a similaridade entre eles dada pelo coeficiente de Jaccard J é definida como:

$$J(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|}$$

Este índice inspirou a construção de um modelo para medir a semelhança entre dois conjuntos de *performance counters*.

Considerando $PC_1 = [pc_{11}, pc_{12}, ..., pc_{1i}, ..., pc_{1n}]$ e $PC_2 = [pc_{21}, pc_{22}, ..., pc_{2i}, ..., pc_{2n}]$ dois conjuntos de *performance counters* normalizados, uma abordagem para medir se estes são similares é verificar para cada $i = 1...n$ se $pc_{1i} = pc_{2i}$ e totalizar quantos *performance counters* têm valor idêntico. Esta seria a aplicação do índice de Jaccard, considerando que cada *performance counter* é um indivíduo.

O problema desta abordagem é que muito raramente um valor de *performance counter* será idêntico a outro. Portanto, a seguinte expressão foi especificada para medir a similaridade entre dois programas:

$$S(PC_1, PC_2) = \frac{1}{n} \sum_{i=1}^{n} \frac{min(pc_{1i}, pc_{2i})}{max(pc_{1i}, pc_{2i})}$$

Assim, quanto maior a diferença entre cada *performance counter* individualmente, menor será o valor de similaridade. É importante também observar que, quando os valores de ambos os programas são idênticos $S(PC_1, PC_2) = 1$.

4 Avaliação Experimental

Um conjunto de experimentos foi planejado e conduzido para avaliar a eficiência do seletor `STBE`, com o objetivo de responder às questões:

1. O `STBE` é capaz de encontrar sequências de transformações com *speedups* superiores àqueles obtidos por *baseline* do compilador Clang?
2. Como a qualidade dos resultados varia em função do número de tentativas do `STBE`?

Os experimentos realizados para avaliar `STBE` foi composto por uma base contendo programas do *benchmark* NPB-Serial, os quais são apresentados na Tabela 1. Esse *benchmarks* foi escolhido por conter diferentes aplicações utilizadas amplamente pela comunidade científica.

Para a construção de cada base foi utilizado $k = 500$ e para a seleção as sequintes quantidades de tentativas: 1, 5, 10 e 20. Além disto é importante destacar que a avaliação de um determinado programa é feita com a base contendo $N - 1$ programas, em outras palavras, é retirado da base as informações pertinentes ao programa a ser avaliado.

Na execução de cada experimento foi utilizado um computador contendo 2 processadores Intel(R) Xeon(R) E5504 2.00GHz, com 24GB de memória e sistema operacional GNU Linux Ubuntu 12 x86_64. Cada programa foi compilado com a infraestrutura de

Benchmark	Descrição
BT	*Solver* tridiagonal de blocos
CG	Gradiente conjugado, acesso irregular à memória e comunicação
DC	Cubo de dados - Movimentação de dados
FT	Transformada discreta de Fourier
EP	Aplicação paralela sem dependências
IS	Ordenação inteira, acesso aletatório à memória
LU	Resolvedor Gauss-Seidel
MG	Implementação de um método multigrid
UA	Aplicação com acesso dinâmico e irregular à memória
SP	Resolvedor Pentadiagonal escalar

Tabela 1. Programas do SNU NPB Serial [17]

T	**Opção**	T	**Opção**	T	**Opção**
T_1	-adce	T_2	-always-inline	T_3	-argpromotion
T_4	-bb-vectorize	T_5	-block-placement	T_6	-break-crit-edges
T_7	-codegenprepare	T_8	-constmerge	T_9	-constprop
T_{10}	-dce	T_{11}	-deadargelim	T_{12}	-die
T_{13}	-functionattrs	T_{14}	-globaldce	T_{15}	-globalopt
T_{16}	-gvn	T_{17}	-indvars	T_{18}	-inline
T_{19}	-instcombine	T_{20}	-instsimplify	T_{21}	-intervals
T_{22}	-ipconstprop	T_{23}	-ipsccp	T_{24}	-jump-threading
T_{25}	-licm	T_{26}	-loop-deletion	T_{27}	-loop-instsimplify
T_{28}	-loop-reduce	T_{29}	-loop-rotate	T_{30}	-loop-simplify
T_{31}	-loop-unroll	T_{32}	-loop-unswitch	T_{33}	-loweratomic
T_{34}	-lowerinvoke	T_{35}	-lowerswitch	T_{36}	-mem2reg
T_{37}	-memcpyopt	T_{38}	-mergefunc	T_{39}	-mergereturn
T_{40}	-partial-inliner	T_{41}	-partial-inliner	T_{42}	-reassociate
T_{43}	-regions	T_{44}	-scalarrepl	T_{45}	-sccp
T_{46}	-simplify-libcalls	T_{47}	-simplifycfg	T_{48}	-sink
T_{49}	-strip	T_{50}	-strip-dead-prototypes	T_{51}	-tailcallelim

Tabela 2. Transformações da LLVM utilizadas no experimento

compilação `LLVM` versão 3.1 [1, 8] e foram utilizadas 51 transformações presentes nesta infraestrutura, as quais são apresentadas na Tabela 2.

Como mencionado anteriormente, STBE utiliza *performance counters* para modelar as características dinâmicas dos programas que compõem a base do sistema, como também do programa a ser avaliado. Neste contexto, é importante destacar que cada hardware disponibiliza um conjunto específico de *performance counters*. Os disponibilizados no hardware utilizado nos experimentos são apresentados na Tabela 3, sendo todos estes utilizados por STBE. Tais *performance counters* forma coletados com a ferramenta PAPI [18], utilizada amplamente para análise de desempenho em arquiteturas de computadores.

Classe	**Performance Counters**
Cache	*L1_DCM, L1_ICM, L1_TCM, L1_LDM, L1_STM, L1_LDM, L1_STM L1_DCH, L1_DCA, L1_DCR, L1_DCW, L1_ICH, L1_ICA, L1_ICR L1_TCA, L1_TCR, L2_DCM, L2_ICM, L2_TCM, L2_LDM, L2_STM L2_LDM, L2_STM, L2_DCH, L2_DCA, L2_DCR, L2_DCW, L2_ICH L2_ICA, L2_ICR, L2_TCH, L2_TCA, L2_TCR, L2_TCW, L3_TCM L3_LDM, L3_DCA, L3_DCR, L3_DCW, L3_ICA, L3_ICR, L3_TCA L3_TCR, L3_TCW*
TLB	*TLB_DM, TLB_IM, TLB_TL*
Branch	*BR_UCN, BR_CN, BR_TKN, BR_NTK, BR_MSP, BR_PRC*
Intruções	*TOT_IIS, TOT_INS, FP_INS, LD_INS, SR_INS, BR_INS, LST_INS VEC_SP, VEC_DP*
Ciclos	*RES_STL, TOT_CYC*
Ponto Flutuante	*FP_OPS, SP_OPS, DP_OPS*

Tabela 3. *Performance counters* utilizados, os quais estão ativos no processador Intel(R) Xeon(R) E5504 2.00GHz

4.1 Speedup

Uma análise importante a ser realizada é verificar a quantidade de iterações que proporciona os melhores resultados. Isto pode ser verificado analisando a Figura 2 que apresenta o *boxplot* para os *speedups* alcançados na execução do STBE para os dez benchmarks e 1, 5, 10 e 20 iterações na seleção, respectivamente.

Para uma iteração apenas os programas BT e MG obtiveram *speedup* superior a $1,0$. Isso pode ser observado nos dois *outliers*, representados por sinais "+", presentes no gráfico e no valor da mediana (representada pela linha horizontal em cada *boxplot*) que é igual a $1,0$. Para cinco e dez iterações houve apenas um *outlier*, porém para cinco iterações a mediana teve valor igual a $1,0$, ou seja, a maioria dos programas apresentou *speedups* iguais a $1,0$ e para dez iterações a maioria dos programas obtiveram *speedup* superior a $1,0$. Portanto, bons resultados surgem a partir de dez iterações.

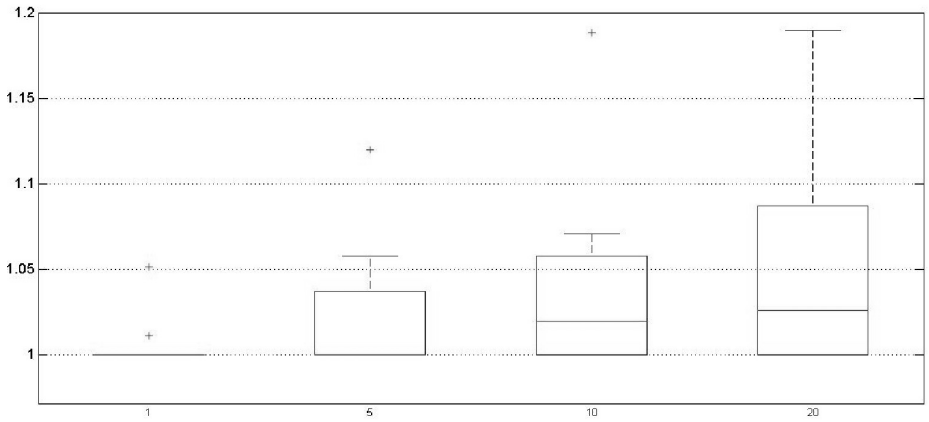

Figura 2. Boxplots para 1, 5, 10 e 20 iterações

Finalmente, para 20 iterações ocorre uma maior dispersão dos valores de *speedup* e não há *outliers*. Além disso, a maior parte dos valores ficaram concentrados acima da mediana indicando que a maior parte dos programas apresentou *speedup* superior a $1,0$.

Para uma visão mais detalhada dos resultados, a Figura 3 apresenta os valores de *speedup* para cada um dos programas utilizados no experimento. Cada grupo de colunas representa os *speedups* de um determinado programa para 1, 5, 10 e 20 iterações, respectivamente.

É possível observar que os programas CG, DC, EP e SP não obtiveram melhoria, ou seja, o `STBE` selecionou o *baseline* para esses programas. Para os outros programas, ou seja, a maioria, o *speedup* foi superior a $1,0$, sendo de $1,04$ para BT; $1,07$ para FT; $1,09$ para UA; $1,11$ para MG; e $1,19$ para LU. É importante ressaltar que tais resultados são promissores, pois um *speedup* de $1,04$ corresponde a 4% de ganho e um speedup de $1,19$ a um ganho de 19%. Isto demonstra que um mecanismo simples tem potencial e é promissor, pois a literatura apresenta ganhos desta ordem porém para sistemas mais complexos [4, 9].

Execuções com uma única iteração não apresentaram melhores resultados. Apenas dois programas obtiveram *speedup* já na primeira iteração. Por outro lado, as execuções com 20 iterações sempre apresentaram o melhor *speedup*. Assim, a melhor configuração para o `STBE`, com relação aos resultados observados pela análise desses *benchmarks* é de 20 iterações.

Os programas CG, DC, EP e SP podem não ter encontrado melhoria pelo fato de não haver, em suas respectivas bases, programas com um alto grau de similaridade com eles. Isso

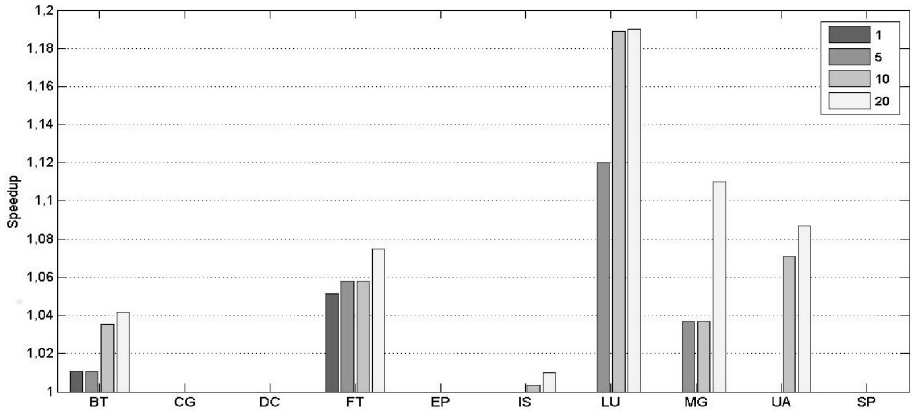

Figura 3. *Speedups* obtidos para cada programa

pode ser observado na Figura 4, que apresenta o melhor índice de similaridade encontrado para cada um dos programas durante o processo de seleção de transformações.

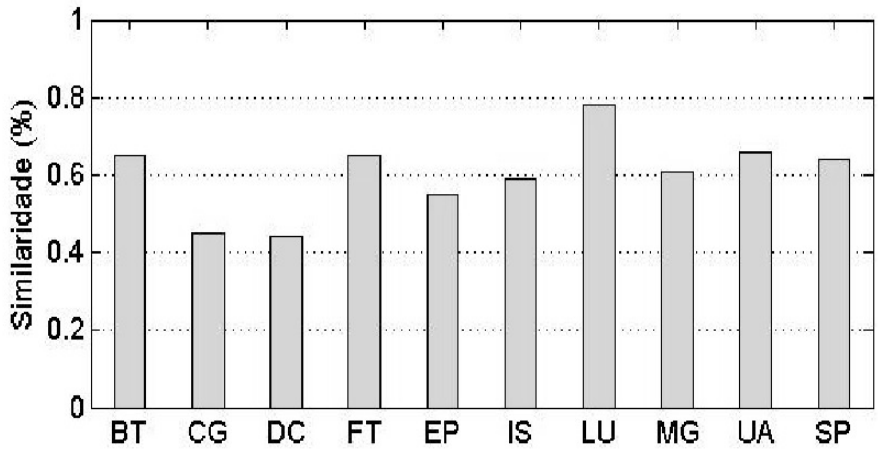

Figura 4. Maior índice de similaridade encontrado na base para cada programa

Os programas CG, DC e EP apresentam o menor índice de similaridade, ou seja, não houve em suas bases programas significativamente similares para que fossem buscadas trans-

formações deles. SP é uma exceção à essa regra. Apesar de ter um programa com uma boa similaridade na base, as sequências associadas a ele não foram suficientes para alcançar um *speedup* para SP.

Apesar de não haver uma melhora para o *speedup* de 4 programas, os resultados permitem concluir que o STBE foi capaz de obter um *speedup* razoável para a maioria das aplicações experimentadas. O melhor resultado foi um *speedup* de $1,19$. Além disso, o *speedup* médio, em relação ao *baseline*, foi de $1,0514$, indicando um ganho geral de $5,14\%$.

4.2 Os Melhores Conjuntos de Transformações

Uma questão importante a ser analisada é o conjunto de transformações que obtiveram um desempenho melhor que o do *baseline*. A Tabela 4 apresenta o melhor conjunto de transformações encontrada pelo STBE após 20 iterações, para cada *benchmark*. O nome de cada transformação pode ser visualizado na Tabela 2

Os melhores conjuntos encontrados por STBE apresentam algumas peculiaridades. Todos eles incluíram uma quantidade relativamente grande de transformações. Dentre 51 transformações disponíveis, os conjuntos, na ordem que aparecem na Tabela 4, apresentaram tamanho 47, 50, 48, 48, 48 e 33, respectivamente. Isso evidencia que a aplicação de poucas transformações não favorece um ganho de desempenho.

Outro ponto a ser destacado é a presença de determinadas transformações em todos melhores conjuntos encontrados: 21 transformações aparecem em todos os seis conjuntos, a saber: *-agrpromotion, -dce, -globaldce, -indvars, -inline, -instcombine, -ipconstprop, -ipsccp, -licm, -loop-rotate, -loop-simplify, -loop-unroll, -loop-unswitch, -lowerinvoke, -lowerswitch, -mem2reg, -memcpyopt, -regions, -sccp, -sink*, e *-tailcallelim*. Isto evidencia que algumas transformações quase sempre encontram oportunidades para melhorar o código. Entretanto, isso também pode ocorrer se os seis programas têm características muito semelhantes.

Algumas características de ordem das transformações também foram comuns para os melhores conjuntos encontrados. A transformação *-block-placement* ocorreu antes das transformações: *-dse, -ipsccp, -loop-rotate, -block-placement, -loop-simplify, -loop-unroll, -mem2reg, -sccp e -sink*. As transformações *-loop-reduce* e *-loop-rotate* ocorreram sempre antes da *-loop-unroll*. Essas características evidenciam que determinadas configurações de ordem são de fato benéficas para a maioria dos casos. Uma sugestão de ordem de aplicação das transformações é apresentada por Muchnick [19].

Programa	Melhor Conjunto Encontrado
BT	[T_9, T_{44}, T_{26}, T_{21}, T_{47}, T_5, T_{10}, T_{23}, T_{14}, T_{45}, T_{25}, T_{32}, T_{49}, T_{36}, T_{15}, T_{13}, T_{51}, T_{42}, T_2, T_{48}, T_{38}, T_{33}, T_{17}, T_{30}, T_{24}, T_3, T_{29}, T_{40}, T_{41}, T_{34}, T_{28}, T_4, T_{20}, T_{50}, T_{12}, T_{43}, T_{46}, T_{35}, T_{31}, T_1, T_{37}, T_{27}, T_{18}, T_{11}, T_6, T_{19}, T_{22}]
CG	*baseline*
DC	*baseline*
FT	[T_{19}, T_{42}, T_{28}, T_1, T_{51}, T_{27}, T_{34}, T_6, T_{33}, T_{17}, T_{43}, T_{18}, T_{13}, T_{32}, T_{44}, T_{50}, T_{14}, T_{35}, T_2, T_{16}, T_5, T_{29}, T_4, T_{31}, T_{26}, T_{25}, T_{10}, T_{49}, T_{30}, T_{38}, T_{20}, T_{37}, T_{40}, T_{41}, T_{24}, T_{23}, T_{12}, T_{36}, T_{11}, T_3, T_{22}, T_{48}, T_{15}, T_{46}, T_9, T_8, T_7, T_{45}, T_{39}, T_{47}]
EP	*baseline*
IS	[T_{19}, T_{37}, T_{39}, T_8, T_{25}, T_4, T_9, T_{49}, T_7, T_3, T_{50}, T_{27}, T_{40}, T_{41}, T_{26}, T_{42}, T_{24}, T_{22}, T_{14}, T_2, T_5, T_{30}, T_{36}, T_{21}, T_{48}, T_{16}, T_{23}, T_{33}, T_{17}, T_{32}, T_{34}, T_{10}, T_{28}, T_{35}, T_{43}, T_{15}, T_{13}, T_{38}, T_{46}, T_{29}, T_{31}, T_{44}, T_{18}, T_{11}, T_{45}, T_{51}, T_{47}, T_{20}]
LU	[T_{18}, T_{20}, T_{51}, T_{39}, T_1, T_{50}, T_{15}, T_{21}, T_{30}, T_{10}, T_{48}, T_{43}, T_{22}, T_4, T_{45}, T_{12}, T_9, T_{49}, T_3, T_{42}, T_{17}, T_{13}, T_{34}, T_{23}, T_{24}, T_{19}, T_{27}, T_{44}, T_2, T_8, T_{32}, T_{28}, T_{35}, T_6, T_{16}, T_7, T_{29}, T_{26}, T_{11}, T_{31}, T_{47}, T_{40}, T_{41}, T_{33}, T_{25}, T_{37}, T_{36}, T_{14}]
MG	[T_{43}, T_{26}, T_{51}, T_{27}, T_{50}, T_{37}, T_{49}, T_7, T_{14}, T_5, T_1, T_{29}, T_{39}, T_{33}, T_{30}, T_{20}, T_{44}, T_{22}, T_{15}, T_{45}, T_3, T_8, T_{34}, T_{38}, T_{18}, T_{28}, T_{35}, T_{24}, T_{36}, T_{31}, T_{13}, T_4, T_{46}, T_9, T_{42}, T_{25}, T_{48}, T_{32}, T_{21}, T_{17}, T_{47}, T_{23}, T_{10}, T_6, T_{16}, T_2, T_{19}]
UA	[T_{10}, T_8, T_5, T_{22}, T_{23}, T_{43}, T_{18}, T_{35}, T_{14}, T_{48}, T_{21}, T_1, T_{16}, T_{30}, T_{38}, T_{36}, T_{25}, T_3, T_{19}, T_{40}, T_{41}, T_{29}, T_{45}, T_{51}, T_{11}, T_{34}, T_{32}, T_{17}, T_7, T_{46}, T_{31}, T_{37}, T_{12}]
SP	*baseline*

Tabela 4. Melhores conjuntos de transformações encontradas pelo `STBE`

5 Conclusões e Trabalhos Futuros

A necessidade de produzir códigos com mais qualidade levou os projetistas de compiladores a implementarem dezenas de otimizações. Tais otimizações também são chamadas de transformações e têm o propósito de melhorar o desempenho do programa gerado.

Neste contexto surge o desafio de descobrir quais transformações proporcionarão o melhor ganho de desempenho a um determinado programa. A exploração de todas as possibilidades é inviável e isso motiva os pesquisadores a procurarem alternativas à abordagem

exaustiva.

Há contextos em que uma abordagem exaustiva pode ser utilizada e alguns trabalhos assim foram citados, apesar desses contextos serem bem específicos. Outros trabalhos abordaram o problema da seleção com busca aleatória. No entanto, uma investigação nesses trabalhos mostrou que outros mecanismos são utilizados para auxiliar na seleção. Dentre esses mecanismos estão algoritmos genéticos e de aprendizagem de máquina. Além destes, existem trabalhos que abordam o problema utilizando técnicas estatísticas.

Neste artigo foi proposto e avaliado o `STBE`, um sistema de seleção que utiliza conhecimento prévio armazenado em uma base, para decidir qual conjunto de transformações aplicar a um código de entrada. Essa decisão é baseada em similaridade, isto é, as características dinâmicas do programa de entrada são comparadas com as características dinâmicas dos programa da base e um coeficiente de similaridade é calculado para cada um deles. Desta forma, com tal coeficiente é possível escolher a partir de uma base de dados qual é o melhor conjunto de transformações a ser aplicado a um determinado código.

`STBE` obteve um *speedup* médio de $1,0514$ para os programas do *benchmarks* `SNU NPB Serial`, o que significa um *speedup* real de $5,14\%$. Dos dez programas avaliados, `STBE` conseguiu encontrar na base conjuntos de tranformações que forneceram um *speedup*, em relação ao *baseline* da `LLVM`, para seis programas. Desta forma, `STBE` demonstrou que é possível obter um mecanismo simples para a seleção de transformações que forneça bons resultados. Com o objetivo de melhorar os resultados do STBE, alguns trabalhos futuros são propostos:

Os próximos trabalhos com `STBE` são: (1) uma experimentação com um número maior de programas; (2) alteração da abordagem utilizada para geração da base; (3) adicionar uma estratégia para verificar a melhor ordem das transformações pertencentes ao melhor conjunto encontrado; e (4) comparar `STBE` com outras abordagens presentes na literatura.

Referências

[1] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, December 2002.

[2] João Roberto Gerônimo and Valdeni Soliani Franco. *Fundamentos da Matemática: uma introdução à lógica matemática, teoria dos conjuntos, relações e funções*. 2008.

[3] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding Effective Compilation Sequences. *SIGPLAN Notices*, 39(7):231–239, June 2004.

[4] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, Washington, DC, USA, 2007.

[5] Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. Exhaustive Optimization Phase Order Space Exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 306–318, Washington, DC, USA, 2006.

[6] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online Performance Auditing: Using Hot Optimizations Without Getting Burned. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 239–251, New York, NY, USA, 2006.

[7] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic Selection of Compiler Options Using Non-parametric Inferential Statistics. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Washington, DC, USA, 2005.

[8] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[9] Eunjung Park, Sameer Kulkarni, and John Cavazos. An Evaluation of Different Modeling Techniques for Iterative Compilation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 65–74, New York, NY, USA, 2011.

[10] Prasad A. Kulkarni, Stephen R. Hines, David B. Whalley, Jason D. Hiser, Jack W. Davidson, and Douglas L. Jones. Fast and Efficient Searches for Effective Optimization-phase Sequences. *ACM Transation on Architecture and Code Optimization*, 2(2):165–198, June 2005.

[11] Henry Massalin. Superoptimizer: A Look at the Smallest Program. *SIGARCH Computer Architecture News*, 15(5):122–126, October 1987.

[12] Juliano Henrique Foleiss, Anderson Faustino da Silva, and Linnyer Beatrys Ruiz. An Experimental Evaluation of Compiler Optimizations on Code Size. In *Proceedings*

of the Brazilian Symposium on Programming Languages, pages 1–15, São Paulo, São Paulo, Brazil, 2011.

[13] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA, 2005.

[14] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Boston, MA, USA, us ed edition, May 2005.

[15] Andrew S. Tanenbaum. *Modern Operating Systems*. Upper Saddle River, NJ, USA, 3rd edition, 2007.

[16] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA, 2006.

[17] Center for Manycore Programming. SNU NPB Serial, 2012. http://aces.snu.ac.kr/Center_for_Manycore_Programming/SNU_NPB_Suite.html.

[18] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting Performance Data with PAPI-C. In *Proceedings of the Parallel Tools Workshop*, pages 157–173, Dresden, Germany, 2009.

[19] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA, 1997.