DF-DTM: explorando redundância de tarefas em *Dataflow*

Leandro Rouberte[1], Alexandre C. Sena[2], Alexandre S. Nery[2], Leandro A. J. Marzulo[2], Tiago A. O. Alves[2], Felipe M. G. França[1]

[1]Programa de Engenharia de Sistemas e Computação - COPPE
Universidade Federal do Rio de Janeiro (UFRJ), Rio de Janeiro, Brasil

[2]Instituto de Matemática e Estatística
Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, Brasil

{leandrof, felipe}@cos.ufrj.br,
{asena, anery, leandro, tiago}@ime.uerj.br

Abstract. *Instruction Reuse is a technique adopted in Von Neumann architectures that improves performance by avoiding redundant execution of instructions (or traces of instructions) when the result to be produced can be obtained by searching an input/output history table for such instruction. Those techniques, however, are yet to be studied in the context of the Dataflow model, which has been gaining traction in the high performance community, due to its inherent parallelism. This paper proposes an approach for reuse in Dataflow, called DF-DTM (Dataflow Dynamic Task Memoization). Our technique supports reuse of individual nodes and subgraphs, which are analogous to instructions and traces, respectively. The potential of DF-DTM is evaluated by a series of experiments that analyze the behavior of redundant tasks in three relevant benchmark applications, resulting in reuse of up to* 97% *of the executed tasks.*

Resumo. *Reúso de Instruções é uma técnica adotada em arquiteturas de Von Neumann para melhorar o desempenho ao evitar a execução redundante de instruções (ou traços de instruções), quando o resultado a ser produzido pode ser extraído de um tabela com o histórico de operandos de entrada e saída da referida instrução. Entretanto, ainda é necessário estudar essas técnicas no contexto do modelo Dataflow, que tem se destacado na comunidade de computação de alto desempenho, devido ao seu paralelismo inerente. Este trabalho propõe uma abordagem para reúso em dataflow, chamada de* DF-DTM *(*Dataflow Dynamic Task Memoization*). A técnica suporta reúso no nível de nós e subgrafos, o que é análogo ao reúso de instruções e traços, respectivamente. O potencial do* DF-DTM *é avaliado com uma série de experimento com três aplicações relevantes, resultando em reúso de até* 97% *das tarefas executadas.*

1. Introdução

O modelo de execução *Dataflow* é um paradigma de computação que explora o paralelismo em aplicações de forma natural. Um programa *Dataflow* é descrito por um grafo direcionado, no qual cada vértice representa uma tarefa (operação) e cada aresta representa as dependências de dados entre tarefas. No modelo *Dataflow*, as tarefas são executadas de acordo com suas dependências, ao invés de seguir a ordem do programa, permitindo a

execução concorrente sem a necessidade de um contador de programa. Pesquisas recentes têm usado o referido modelo como uma alternativa atraente para programação paralela que, além de ser mais transparente para os usuários, consegue prover o desempenho desejado [Marzulo et al. 2014, Alves et al. 2014, Duran et al. 2011, Bosilca et al. 2012, Wozniak et al. 2013]. Além disso, arquiteturas *Dataflow* têm sido propostas por diferentes grupos de pesquisa [Swanson et al. 2003, Giorgi 2014] e já são uma realidade na indústria de aceleradores para computação de alto desempenho [Pell et al. 2013].

Além do aumento no número de núcleos de processamento, outras soluções no nível arquitetural foram estudadas nas últimas décadas para melhorar o desempenho em um *chip*. Uma dessas soluções, chamada *Dynamic Trace Memoization* (*DTM*) [da Costa et al. 2000], foi inspirada em uma técnica de aprendizado de máquina conhecida como *Memoization* [Michie 1968]. A ideia geral do *Memoization* é simples: evitar execuções redundantes se os operandos de entrada de uma instrução (ou grupos consecutivos de instrução, chamados de traços) são iguais a um conjunto de operandos visto anteriormente, tornando a re-execução desnecessária. Os ganhos de desempenho obtidos com o reúso de instruções ou traços podem ser substanciais, visto que a ocorrência de traços com operandos redundantes é comum em um programa.

Este trabalho discute e analisa o potencial de reúso no modelo de execução *Dataflow*. O objetivo deste estudo é usar os conhecimentos obtidos para incluir a técnica de reúso em aceleradores *Dataflow* em FPGA que estão sendo desenvolvidos pelos autores. O mecanismo, chamado de *DF-DTM* (Dataflow Dynamic Task Memoization), é inspirado nas técnicas tradicionais de reúso, adaptadas para o modelo de execução guiada por fluxo de dados. No *DF-DTM*, tarefas (ou nós) e subgrafos são análogos, respectivamente, às instruções e traços do *DTM* tradicional. Sequências redundantes de tarefas, os subgrafos, podem ser identificados durante a execução de um programa *Dataflow* e tanto instruções quanto subgrafos podem ser reusados. A biblioteca *Dataflow* Sucuri [Alves et al. 2014] foi modificada para implementar o *DF-DTM* e identificar em quais situações ocorre o reúso e como os aspectos de implementação podem afetá-lo.

Os resultados mostram que aplicações executando no modelo *Dataflow* podem se beneficiar com o reúso de tarefas e subgrafos. Três aplicações relevantes foram avaliadas: o LCS, um contador de palavras implementado com *Map-reduce* e o *Unbounded Knapsack*. Aproximadamente 97% e 55.4% das tarefas instanciadas foram reusadas para as aplicações LCS e *Map-reduce*, respectivamente. Além disso, o *Knapsack* não apresentou redundância. Experimentos variando o tamanho e a distribuição das estruturas de cache do *DF-DTM* mostram que é possível obter taxas de reúso significativas com poucas linhas de cache, indicando que uma futura implementação em *hardware* é viável.

O restante deste trabalho está organizado da seguinte forma: (*i*) na Seção 2 é feita uma introdução dos conceitos de reúso em *Dataflow* e explicação da sua implementação na biblioteca Sucuri; (*ii*) na Seção 3 são apresentados e discutidos os resultados experimentais; (*iii*) na Seção 4 é feita uma revisão de trabalhos relacionados; (*iv*) na Seção 5 é feita a conclusão e proposição de ideias para trabalhos futuros.

2. Reúso em *Dataflow*

Neste trabalho a biblioteca *Dataflow* Sucuri é usada como base para implementar e avaliar uma abordagem de reúso de tarefas e subgrafos redundantes, o *DF-DTM*. Nesta seção, são

discutidos os conceitos básicos da biblioteca Sucuri e apresentado o *DF-DTM*.

A Sucuri é uma biblioteca minimalista para programação *Dataflow* em Python, onde o desenvolvedor implementa funções do seu programa e as associa a objetos *Node* (nó), que são conectados com arestas, de acordo com suas dependências de dados. Os nós são inseridos em um objeto *Graph* (grafo), que é passado como parâmetro para o escalonador da Sucuri, responsável pela execução de acordo com a regra de disparo *Dataflow*. Essa abordagem permite despachar a execução de um nó quando todos os operandos de entrada estiverem disponíveis.

Cada nó Sucuri pode ser visto como uma tarefa estática. O escalonador da Sucuri conta com uma unidade de casamento que é responsável por verificar se a regra de disparo é satisfeita para algum nó, a cada operando recebido. Quando isso acontece, uma tarefa dinâmica é criada e colocada em uma fila de prontos no escalonador. A Sucuri possui um conjunto de trabalhadores (processos) que buscam tarefas dinâmicas na fila de prontos. Os trabalhadores consultam o grafo para chamar a função associada à tarefa e os resultados produzidos são enviados para o escalonador.

É importante notar que múltiplas tarefas estáticas podem ser associadas à mesma função e são, portanto, do mesmo tipo. Tarefas com o mesmo tipo são equivalentes a instruções com mesmo mnemônico em uma máquina de Von Neumann. Por outro lado, uma mesma tarefa estática pode ter múltiplas instâncias dinâmicas e possuem, portanto, o mesmo identificador. Tarefas com o mesmo identificador são equivalentes a instruções com o mesmo contador de programa (PC) em uma máquina de Von Neumann.

No *DTM* tradicional, o reúso de instruções é disparado quando os operandos de entrada (contexto de entrada) de uma certa instrução se repetem ao longo do tempo, gerando os mesmos resultados (contexto de saída). Quando isso ocorre, o PC é usado para endereçar uma cache de reúso que armazena os contextos de entrada e de saída para as instruções. Isso é equivalente a dizer que o reúso é apenas permitido entre instâncias dinâmicas de instruções com o mesmo PC (identificador). O reúso também poderia ser feito entre instruções com o mesmo mnemônico (tipo), resultando em um melhor aproveitamento de redundância. Entretanto, conforme descrito em [da Costa et al. 2000], é importante armazenar o estado do preditor de desvio para cada instrução, o que é totalmente relacionado com o contador de programa.

No modelo *Dataflow* não existe contador de programa e não há a necessidade de preditores de desvio. Sendo assim, no *DF-DTM*, o reúso pode ser feito tanto entre instâncias da mesma tarefa (com o mesmo identificador), quanto entre tarefas do mesmo tipo (relacionadas com a mesma função). Este último possui maior potencial de reúso, pois resultados produzidos por instâncias de um nó podem ser reutilizados por instâncias de outro nó.

É importante notar que algumas tarefas implementadas com a biblioteca Sucuri não são elegíveis para o reúso, seja por terem resultados não determinísticos (como funções que usam números randômicos) ou por terem operações não idempotentes que produzem efeitos colaterais.

Os contextos de entrada e saída de tarefas reusáveis são armazenados na Tabela de Reúso de Nós (TRN). Além disso, são armazenados o identificador do nó (ID) e o seu tipo. A Figura 1 mostra a implementação básica do *DF-DTM* na Sucuri e provê um exemplo

de uso da TRN. A figura também mostra as principais entidades da Sucuri: o grafo, os trabalhadores e o escalonador. O *DF-DTM* foi implementado dentro do escalonador da Sucuri. Quando a unidade de casamento despacha uma tarefa pronta para execução, ela verifica se a tarefa é elegível para reúso, buscando por tarefas do mesmo tipo e contexto de entrada na TRN. Se alguma entrada na TRN for encontrada a tarefa não é colocada na fila de prontos e o seu resultado, previamente armazenado na TRN, é repassado aos nós que dependem do mesmo. Na Figura 1 os nós do mesmo tipo possuem a mesma cor e a TRN possui algumas entradas com resultados anteriores para os nós *brancos*. Se o escalonador receber um operando com o valor a destinado a tarefa 3 (do tipo *branco*), ocorrerá um reúso, visto que já existe uma entrada com o valor a para nós do mesmo tipo de 3.

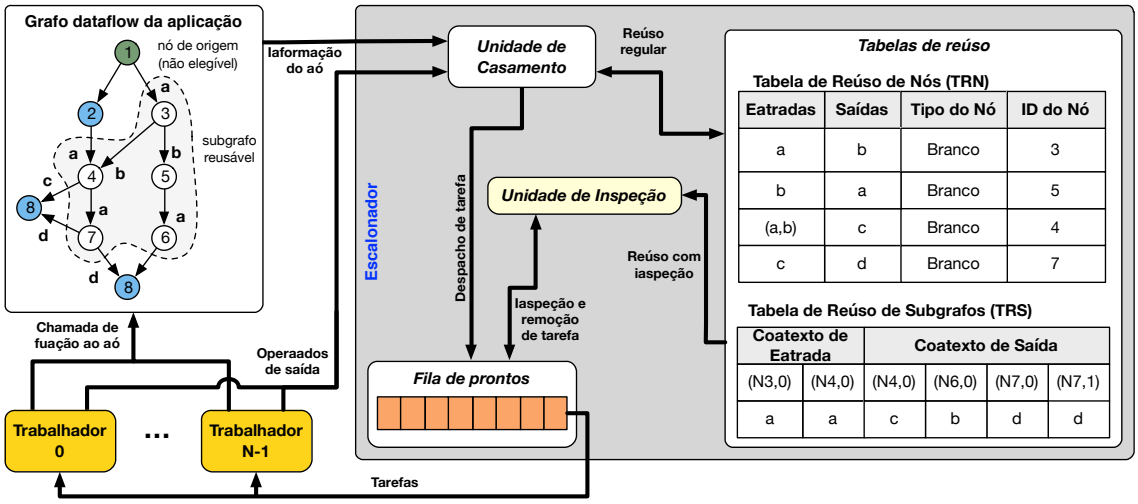


Figura 1. Implementação e exemplo do *DF-DTM*

O *DF-DTM* também inclui uma Tabela de Reúso de Subgrafos (TRS), que permite que o escalonador da Sucuri evite a execução de subgrafos redundantes. O reúso de subgrafos é equivalente ao reúso de traços no *DTM* tradicional [da Costa et al. 2000]. Subgrafos redundantes são construídos quando nós consecutivos são reusados. Por exemplo, na Figura 1 considere que a TRN já está preenchida e que a TRS está vazia. Se o nó 3 recebe a como entrada e o nó 4 recebe a como entrada esquerda, os nós 3, 4, 5 e 6 serão reusados em sequência, pois todos os seus operandos de entrada já estão na TRN. Isso significa que pode ser gerada uma entrada na TRS contendo todas as entradas e saídas para o subgrafo formado por esses nós. A construção do subgrafo termina quando não há mais nós redundantes nas arestas de saída do mesmo. A TRS armazena os contextos de entrada e saída do subgrafo contendo uma lista de tuplas (N_{id}, P, V) (identificador nó, porta de entrada ou saída e valor).

Um subgrafo é reusado quando as informações no contexto de entrada de um conjunto de nós prontos para execução correspondem aos dados de uma entrada da TRS. Além disso, a TRS é consultada antes da TRN, visto que é mais vantajoso reusar subgrafos do que nós independentes. É importante notar que o reúso de subgrafos é feito com base no identificador dos nós ao invés do tipo. O reúso de subgrafos pelo tipo de seus nós envolveria encontrar subgrafos isomórficos, que pode ser custoso. Por outro lado, o reúso pelo identificador pode limitar o potencial de reúso. A investigação de outras estratégias

de reúso de subgrafo é tema de trabalhos futuros.

Deixar a unidade de casamento como única responsável pela identificação das oportunidades de reúso na chegada de operandos pode não ser a melhor forma de explorar o potencial completo do *DF-DTM*. Por exemplo, em aplicações do tipo *fork-join* o casamento de todas as tarefas da etapa de *fork* é feito praticamente ao mesmo tempo, quando a TRN ainda está vazia, eliminando qualquer possibilidade de reúso. Entretanto, quando o número de trabalhadores é muito menor que o números de nós do *fork* muitas tarefas ficarão retidas na fila de prontos, aguardando a liberação de trabalhadores. Nesse caso, tarefas anteriores que terminam sua execução preencherão novas entradas na TRN com valores possivelmente redundantes aos das tarefas que ainda estão na fila de prontos. Para aproveitar essa oportunidade de reúso, um mecanismo de inspeção foi criado para procurar tarefas na fila de prontos cujos contextos de entrada sejam iguais ao de entradas recém criadas na TRN. Tais tarefas serão removidas da fila de prontos e reutilizarão os resultados já armazenados.

3. Análise Experimental

Esta seção apresenta uma série de experimentos que avaliam o potencial de reúso de tarefas redundantes em dataflow. Inicialmente, mediu-se as taxas de reúso alcançadas para diferentes tipos de *cache*. Em seguida, foi analisada a influência da granularidade das tarefas e do grau de paralelismo nas taxas de reúso. Finalmente, é apresentada uma avaliação do efeito de diferentes tamanhos de *caches* no potencial de reúso. Todos os experimentos foram conduzidos em uma máquina virtual localizada na nuvem *google cloud* com 8 VCPUS, 32 GB de RAM e 10 GB de armazenamento em disco. Além disso, já que os trabalhadores Sucuri estão executando em paralelo, a ordem em que cada operando chega ao escalonador pode variar em cada execução devido a condições de corrida. Portanto, todos os experimentos foram executados oito vezes com 8 trabalhadores. Foi usada a média aritmética dos resultados e o desvio padrão foi insignificante.

As principais métricas usadas nesses experimentos são: taxa de reúso máximo, taxa de reúso real e distribuição da taxa de reúso real. A taxa de reúso máximo em um grafo dataflow é calculada como $1 - \frac{D(N)}{N}$, onde $D(N)$ é o número de nós dinâmicos com contextos de entrada distintos e N é o número total de nós dinâmicos criados durante a execução dataflow. Por outro lado, a taxa de reúso real é o percentual de tarefas redundantes que foram detectadas. Finalmente, a distribuição da taxa de reúso real é usada para medir por qual mecanismo as tarefas redundantes foram detectadas (*i.e.*, busca na tabela TRN, TRS ou pelo mecanismo de Inspeção).

3.1. *Benchmarks*

Para avaliar o potencial de reúso em *Dataflow*, três aplicações foram implementadas usando a biblioteca Sucuri: o LCS (*Longest Common Sub-sequence*), uma contagem de palavras (*Word Counting*) usando o padrão *MapReduce* e o algoritmo *Unbounded Knapsack*. Os últimos dois existem no *DaSH Benchmark Suite* [Gajinov et al. 2014], específico para ambientes dataflow, e foram adptados para a biblioteca Sucuri.

LCS – Essa aplicação encontra uma subsequência comum de tamanho máximo entre duas sequências de caracteres. O algoritmo adota programação dinâmica e preenche uma matriz de pontuação com resultados de comparação dos caracteres das duas sequencias. Cada elemento da matriz depende dos vizinhos superior e esquerdo, seguindo

o padrão *wavefront*. A implementação para sucuri, baseada em [Alves et al. 2014], foi modificada para fazer a comparação de uma sequência com uma lista de sequências em um *pipeline*. O primeiro nó do *pipeline* recebe uma lista arquivos (caminhos) com as sequências a serem processadas. Cada arquivo é passado para um nó de leitura e as sequências lidas são encaminhadas para um sub-grafo *wavefront*. Finalmente, os resultados são encaminhados para um nó que escreve os dados em um arquivo, na mesma ordem de disparo pelo do primeiro nó. O primeiro e último nós do *pipeline* não são candidatos ao reúso. Nos experimentos é feita a comparação de uma sequência com outras sete, todas com 500 caracteres. O grão padrão utilizado foi de 1 elemento por nó.

MapReduce – O contador de palavras *MapReduce* [Gajinov et al. 2014] é uma aplicação paralela do tipo *fork-join* que lê múltiplos arquivos, cada um contendo palavras separadas por espaços, e calcula a frequência de ocorrência destas palavras em um outro arquivo específico. O modelo de programação do *MapReduce* descreve a criação de um programa na forma de funções mapeadoras e funções redutoras. As funções mapeadoras associam um dado valor a uma chave, gerando uma tupla. Neste *benchmark*, cada palavra é associada com o valor 1. A função redutora recebe as tuplas e aplica o operador agregador a elas. A redução é feita de forma hierárquica. A aplicação processa uma lista de arquivos em um *pipeline*. O primeiro e último nós do *pipeline* também não são candidatos ao reúso. Nos experimentos a análise de frequência de palavras foi executada para 10 arquivos, cada um contendo 3797 palavras.

Unbounded Knapsack – O terceiro benchmark é o problema da mochila sem restrição, *Unbounded Knapsack*, que consiste em, dado um conjunto de tuplas (w_i, v_i), onde w_i é o peso de um item i e v_i é o seu valor, encontrar o subconjunto de itens que maximizam o lucro sem exceder a capacidade C da mochila [P. C. Gilmore 1961, Gajinov et al. 2014]. O algoritmo usa uma matriz M de programação dinâmica, procurando por uma solução nesta matriz até encontrar a mochila de capacidade C. Cada elemento $M[i, j]$, exceto aqueles da primeira linha e da primeira coluna, é computado da seguinte maneira: $max(l \times v[i] + M[i-1, j-l*w[i])$ onde l varia de 0 a $j/w[i]$ (o número de itens i que a mochila comporta), $v[i]$ e $w[i]$ são os valores e os pesos de cada item i, respectivamente, e j é a capacidade da mochila que está sendo analisada. Quando o algoritmo termina, o valor máximo para a mochila é encontrado.

Nossa implementação dataflow utiliza um padrão *wavefront*, similar ao do algoritmo LCS. No entanto, o tamanho dos operandos cresce a medida que a execução avança no *wavefront*. Isto ocorre porque cada nó $N[i, j]$ da matriz *wavefront* irá precisar dos resultados de todos os outros nós à sua esquerda. Como nos outros dois benchmarks, o *unbounded knapsack* também usa um *pipeline* para resolver múltiplos problemas listados em um catálogo de arquivo. O primeiro e último nós do *pipeline* também não são candidatos ao reúso. O algoritmo *unbounded knapsack* foi executado para 10 mochilas com capacidade 200 e considerando 100 itens. O grão de tarefa usado por padrão foi de 1 elemento por partição de linha.

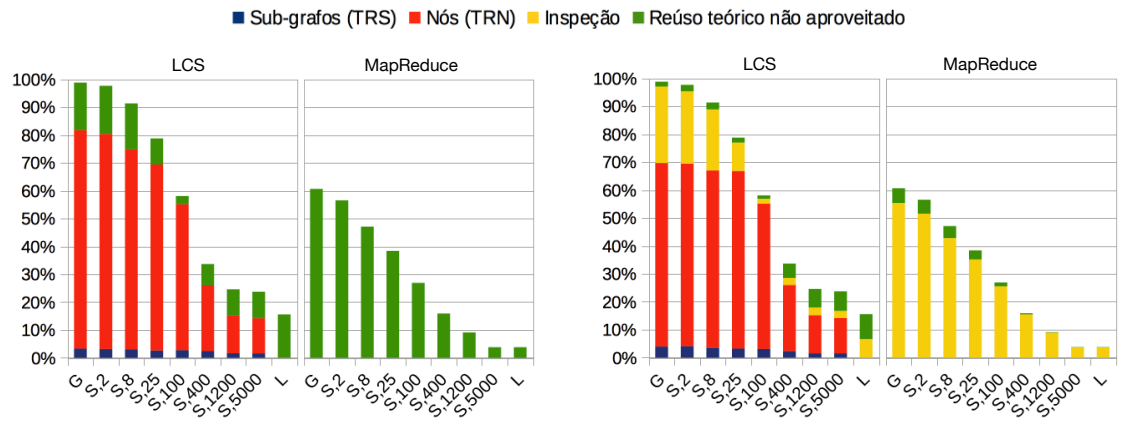
3.2. Tipos de Caches Vs. Taxas de Reúso

Em um sistema real, a cache de reúso poderia ser distribuída como ocorre em processadores *multicore* com caches locais L1 e L2 para cada núcleo. Portanto, foi adicionado um campo *GID* na TRN para simular os efeitos de caches distribuídas no reúso. As entradas

na tabela com o mesmo ID de grupo pertencerão à mesma cache. Esta abordagem foi usada em detrimento de implementar diferentes tabelas já que o propósito deste trabalho é medir o potencial de reúso das aplicações e não avaliar os custos de implementação.

Três tipos de caches TRN foram avaliadas: local (L), global (G) e compartilhada (S). As tabelas TRN locais são individuais para cada nó. A TRN global é uma tabela centralizada que armazena informação de reúso a respeito de todos os nós. As tabelas TRN compartilhadas são distribuídas e armazenam informação sobre grupos de nós. Uma TRN compartilhada com n tabelas (S,n) significa que os nós serão organizados em n grupos, cada um usando uma TRN diferente. O cálculo do *GID* de cada nó com identificador *ID* é dado por $GID = ID \mod n$.

É importante observar que o cálculo da taxa de reúso máximo é diretamente influenciado pelos tipos de cache. Em caches globais a informação de reúso é compartilhada por todos os nós. Já em caches compartilhadas, o reúso só é feito entre nós do mesmo grupo e em caches locais, entre nós com o mesmo identificador. No caso de caches compartilhadas, a política de agrupamento também pode influenciar as taxas de reúso. O estudo de outras políticas de agrupamento é tema de pesquisas em andamento.



(a) sem inspeção (b) com inspeção

Figura 2. Distribuição da taxa de reúso teórica para diferentes tipos de cache para as aplicações LCS e *Mapreduce*

A Figura 2(a) apresenta os resultados para a execução dos experimentos sem inspeção. O eixo-y mostra a taxa de reúso e o eixo-x o tipo de cache TRN. O gráfico mostra a contribuição do reúso de subgrafos separadamente. Note que para as duas aplicações, o reúso cai com o aumento do número de grupos nas caches compartilhadas, chegando ao pior valor para caches locais. Para o LCS o reúso máximo varia entre 98.8% e 15.6%. Já para o MapReduce o reúso máximo varia entre 60.6% e 0%. Em ambos os caso, os operandos reutilizados eram lista de tuplas contendo inteiros e caracteres. No *MapReduce* os redutores fazem o agrupamento hierárquico de palavras e possuem, portanto, contexto de entrada variável (mais operandos a cada nível da redução). Isto prejudica a taxa de reúso, pois operandos grandes dificilmente aparecem novamente em um programa. O *knapsack* não foi apresentado nas figuras porque não teve nenhuma redundância. Isto também pode ser explicado pelo fato dos operandos desta aplicação terem tamanhos variados. Um outro fator prejudicial para a taxa de reúso do *knapsack* é que todos os nós que processam

uma partição da linha produzem um operando diferente, já que todas as linhas representam uma tupla única (valor, peso). Por isso, para uma única execução do *knapsack* não é possível ter qualquer redundância. Pode ser que seja possível encontrar redundância para este problema particular quando considera-se mais de um grupo de itens. Nos experimentos realizados, o *knapsack* foi executado para uma sequência de 10 grupos de itens, mas nenhum redundância foi identificada.

Considerando a taxa de reúso real, para o benchmark LCS, os acertos na tabela TRN foram responsáveis por grande parte do reúso alcançado, sendo aproximadamente 78.5% para a cache global. Ainda que rara, o benchmark LCS revelou a existência de aproximadamente 3.31% subgrafos redundantes durante a execução. Nenhum reúso real foi alcançado considerando-se apenas as caches locais, ainda que tenham existido tarefas redundantes. Isto se deve à execução quase consecutiva de tarefas para um único nó, sem um intervalo de tempo considerável entre elas. Logo, quando a cache foi consultada, os trabalhadores não tinham ainda enviado os resultados necessários para pular a dada tarefa. Isto não foi um problema grave para os demais tipos de cache, pois os nós puderam checar o resultado produzido por outros nós.

Na aplicação MapReduce, nenhuma tarefa redundante foi detectada pela TRN ou TRS, em todos os tipos de cache. Isto pode ser explicado pelo fato de que para esse tipo de grafo, *fork-join*, existe um grande número de tarefas que são marcadas como prontas ao mesmo tempo. Por exemplo, o nó map, quando executado, irá liberar a execução de todos os redutores de primeiro nível ($red_{1,1}$, $red_{1,2}$, $red_{1,3}$...). Mesmo que $red_{1,1}$ e $red_{1,2}$ realizem o mesmo trabalho, $red_{1,2}$ não será capaz de usar os resultados de $red_{1,1}$, pois ele não estará na fila de prontos ainda. O reúso de computação de $red_{1,1}$ por $red_{1,2}$ somente seria possível com o mecanismo de Inspeção.

A Figura 2(b) mostra que o mecanismo de inspeção foi muito efetivo, aumentando substancialmente a taxa de reúso real. Para o LCS, considerando a cache global, a taxa de reúso real é de 97%, quase alcançando a taxa de reúso máximo de 98.8%. Observe que com o mecanismo de inspeção, a contribuição da cache TRN para a taxa de reúso real diminui, enquanto que da cache TRS aumenta. Isto pode ser explicado pelo fato da inspeção remover as tarefas da fila de prontos mais cedo. Portanto, a cache TRN não terá tido tempo de ser preenchida com tantos resultados, e não terá sido capaz de reusar nós continuamente. Um comportamento interessante foi que, para caches compartilhadas de tamanhos 100 a 5000, a contribuição do mecanismo de inspeção foi menor que para a cache local. Isto ocorre porque tais caches compartilhadas tiveram mais reúso de nós, com menos tarefas indo para a fila de prontos. Para o *MapReduce*, o mecanismo de inspeção foi capaz de atingir uma taxa de reúso real de aproximadamente 55.4% para o cenário de cache global, somente 5.2% menor que a taxa de reúso máximo, confirmando a suposição de que muitas das tarefas redundantes estavam na fila de prontos.

3.3. Granularidade Vs. Taxa de Reúso

Neste experimento foi avaliado como o aumento da granularidade das tarefas afeta as taxas de reúso máximo, considerando uma cache global. Na Figura 3, o eixo-x representa o tamanho da granularidade da tarefa e o eixo-y representa a taxa de reúso máximo. Como pode ser observado, a taxa de reúso máximo diminui drasticamente para ambos *benchmarks*. No caso do LCS, a taxa de reúso máximo de 98.8% para blocos $[1X1]$ (um

elemento da matriz) despenca para 44.5% e 9% para blocos $[2X2]$ e $[4X4]$, respectivamente. A medida que o grão cresce, as únicas tarefas redundantes são aquelas de leitura de arquivos para o arquivo de sequência que é comparado contra os demais arquivos de sequência. Para o benchmark MapReduce, as taxas de reúso caem de 60.7%, quando se utiliza 1 palavra por redutor, para 27.6% e 2.6%, quando se utiliza 2 e 4 palavras por redutor, respectivamente.

Este comportamento indica que, mesmo em aplicações com grande potencial de reúso, o grão deve ser o menor possível para se beneficiar do reúso. Modelos de programação dataflow adotam tarefas de granularidade mais grossa para mitigar os custos de gerenciamento de tarefas. Entretanto, o objetivo deste projeto é aplicar reúso no contexto de aceleradores *Dataflow* em FPGA, onde são empregadas tarefas de grão mais fino.

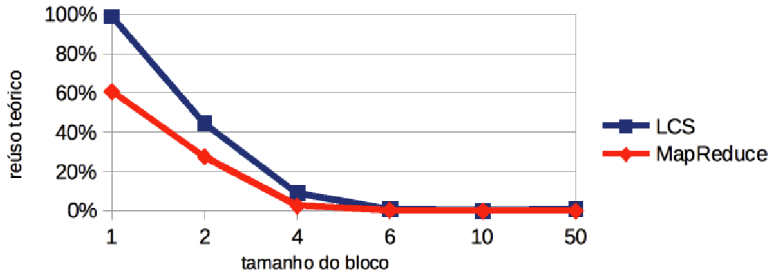


Figura 3. Comportamento do reúso máximo com a variação na granularidade das tarefas

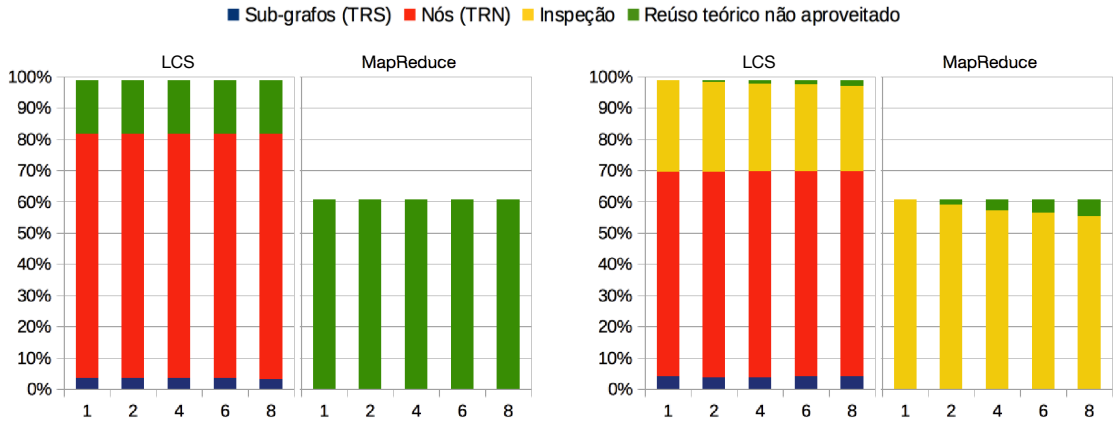
3.4. Paralelismo Vs. Taxa de Reúso

O *DF-DTM* pode ser aplicado tanto no nível de modelos de programação paralela quanto no nível de arquiteturas paralelas. Portanto é relevante estudar como o reúso de tarefas se comporta em um cenário de múltiplos trabalhadores, o que seria equivalente a ter múltiplos processadores dataflow para cada unidade de reúso. Como pode ser visto na Figura 4(a), virtualmente não há diferença entre os cenários, quando nenhum mecanismo de inspeção é aplicado. Esta é uma característica muito importante que possibilita que o modelo paralelo dataflow trabalhe em sinergia com a técnica de reúso para aumentar o desempenho de aplicações.

Já com o mecanismo de inspeção (Figura 4(b)), é possível observar que, para ambos *benchmarks*, a taxa de reúso diminui sutilmente a medida que o número de *threads* aumenta. A inspeção beneficia tarefas que ficam mais tempo na fila de prontos. Com mais trabalhadores executando, a taxa de consumo da fila de prontos aumenta e, consequentemente, menos tarefas esperam na fila. Para o *MapReduce*, a taxa de reúso foi mais prejudicada pelo aumento do número de trabalhadores do que o LCS, pois todo o reúso no *MapReduce* vem exclusivamente desse mecanismo.

3.5. Tamanho da Cache Vs. Taxa de Reúso

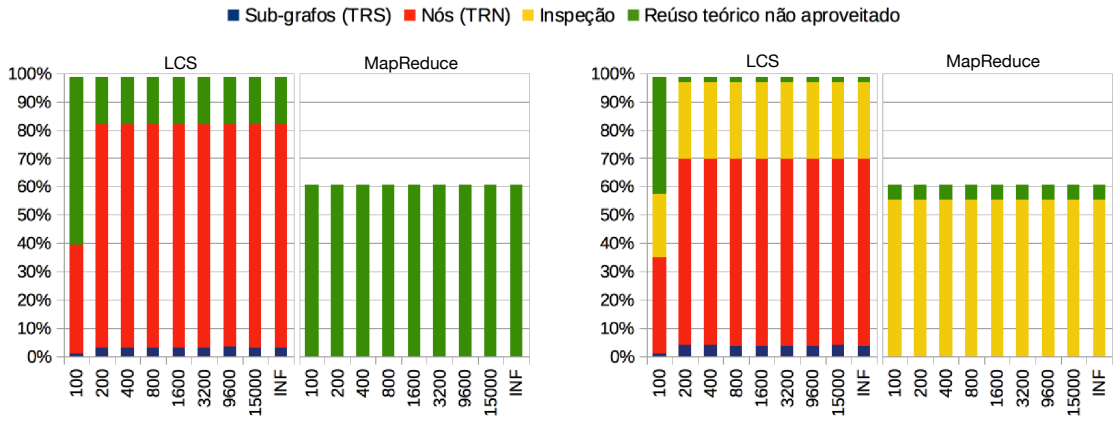
Já que a memória cache tem um papel crucial na técnica de reúso, é importante investigar qual o impacto de diferentes tamanhos de cache na taxa de reúso. Portanto, este experimento avalia a influencia do tamanho de cache na detecção de tarefas redundantes. Foram avaliadas caches globais com variação de tamanho e uma política de substituição LRU.



(a) sem inspeção (b) com inspeção

Figura 4. Comportamento da taxa de reúso com a variação no número de trabalhadores Sucuri

A Figura 5(a) mostra a taxa de reúso máximo e a taxa de reúso real (eixo-y) para cada tamanho de cache (eixo-x), sem o mecanismo de inspeção. Observa-se que, para o benchmark LCS, apesar de uma cache com 100 entradas alcançar somente 39.6% de reúso, a cache com apenas 200 entradas atingiu o mesmo grau de detecção de reúso que uma cache ilimitada (aproximadamente 81.8%). Isto acontece porque o tempo entre tarefas redundantes neste benchmark é pequeno o suficiente para permitir reúso perto do ótimo com uma cache de apenas 200 entradas. A Figura 5(b) mostra a importância do mecanismo de inspeção. Além do aumento da detecção de reúso para caches de 100 entradas para a aplicação LCS, o mecanismo de inspeção possibilita a detecção de reúso na aplicação MapReduce. A principal contribuição deste experimento é mostrar que uma cache pequena é o suficiente para atingir o mesmo desempenho de caches grandes.



(a) sem inspeção (b) com inspeção

Figura 5. Comportamento da taxa de reúso com a variação do tamanho da cache

4. Trabalhos Relacionados

Diversas abordagens para reúso foram propostas para arquiteturas tradicionais, baseadas no modelo de Von Neumann [Sodani and Sohi 1997, Tsai and Chen 2011,

Shibata et al. 2014], onde o fluxo de execução é guiado pelo contador de programa. Tais arquiteturas dificilmente conseguem explorar o paralelismo que geralmente pode ser identificado entre instruções independentes para uma dada aplicação, limitando o seu potencial de desempenho. Por esta razão, várias otimizações foram propostas ao longo dos anos para aumentar a vazão de operações na execução de código sequencial, usando, por exemplo, *hardware* especializado para previsão de desvios, especulação, execução fora-de-ordem, entre outras técnicas. O *DTM* [da Costa et al. 2000] pode ser considerado como uma destas otimizações, que tem o objetivo de reutilizar computações redundantes para evitar a execução de instruções ou sequências de instruções (traços) que tenham resultados anteriores armazenados em cache.

O *DF-DTM* é o primeiro mecanismo de reúso para máquinas e ambientes de execução baseados no modelo *dataflow* e amplamente inspirado no *DTM*. Como no paradigma dataflow operações são executadas segundo o fluxo de dados do programa, o paralelismo não é limitado por um contador de programa e quaisquer operações que já tenham recebido seus operandos podem executar em paralelo usando unidades funcionais diferentes. Além disso, o modelo dataflow não requer uso de registradores e todos os operandos são trocados diretamente entre as instruções ou tarefas produtoras e consumidoras destes operandos. Estas características permitem novas oportunidades de reúso, como foi possível observar nos experimentos.

5. Conclusões e Trabalhos Futuros

Este trabalho apresentou o *DF-DTM* (*Dataflow Dynamic Task Memoization*), um mecanismo que permite que resultados de tarefas redundantes sejam reutilizados, evitando execuções desnecessárias. O mecanismo foi implementado na biblioteca dataflow Sucuri e uma série de experimentos foi conduzida para verificar o potencial de reúso encontrado em aplicações *Dataflow*.

O *DF-DTM* apresentou grande potencial, atingindo taxas teóricas de reúso de até 98.8% e taxa de reúso real de até 97% para a aplicação LCS. Além disso, os experimentos mostraram que o mecanismo de inspeção teve papel crucial, aumentando as taxas de reúso real no LCS e abrindo oportunidades de reúso em aplicações importantes com o *MapReduce*, sem adicionar complexidade nas tabelas de reúso.

Este trabalho abre muitas oportunidades de contribuições futuras, como a avaliação de mais aplicações e o desenvolvimento de um simulador arquitetural para medir os custos do reúso. Estes passos são essenciais para a aplicação de técnicas de reúso no contexto de aceleradores *Dataflow* em FPGA, desenvolvidos pelos autores.

Agradecimentos

À FAPERJ (processo E-26/203.537/2015), ao CNPq e à CAPES pelo apoio dado aos autores deste trabalho.

Referências

Alves, T. A. O., Goldstein, B. F., França, F. M. G., and Marzulo, L. A. J. (2014). A minimalistic dataflow programming library for python. In *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*, pages 96–101.

Bosilca, G., Bouteiller, A., Danalis, A., Hérault, T., Lemarinier, P., and Dongarra, J. (2012). Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1-2):37–51.

da Costa, A. T., Franca, F. M. G., and Filho, E. M. C. (2000). The dynamic trace memoization reuse technique. In *Parallel Architectures and Compilation Techniques, 2000. Proceedings. International Conference on*, pages 92–99.

Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21:173–193.

Gajinov, V., Stipić, S., Erić, I., Unsal, O. S., Ayguadé, E., and Cristal, A. (2014). Dash: A benchmark suite for hybrid dataflow and shared memory programming models: with comparative evaluation of three hybrid dataflow models. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 4:1–4:11, New York, NY, USA. ACM.

Giorgi, R. e. a. (2014). TERAFLUX: Harnessing dataflow in next generation teradevices. *Microprocessors and Microsystems*, pages –.

Marzulo, L. A., Alves, T. A., França, F. M., and Costa, V. S. (2014). Couillard: Parallel programming via coarse-grained data-flow compilation. *Parallel Computing*, 40(10):661 – 680.

Michie, D. (1968). "Memo" Functions and Machine Learning. *Nature*, 218:19–22.

P. C. Gilmore, R. E. G. (1961). A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859.

Pell, O., Mencer, O., Tsoi, K., and Luk, W. (2013). *Maximum performance computing with dataflow engines*, pages 747–774.

Shibata, Y., Tsumura, T., Tsumura, T., and Nakashima, Y. (2014). An implementation of auto-memoization mechanism on arm-based superscalar processor. In *System-on-Chip (SoC), 2014 International Symposium on*, pages 1–8.

Sodani, A. and Sohi, G. S. (1997). Dynamic instruction reuse. In *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on*, pages 194–205.

Swanson, S., Michelson, K., Schwerin, A., and Oskin, M. (2003). Wavescalar. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 291–302.

Tsai, Y. Y. and Chen, C. H. (2011). Energy-efficient trace reuse cache for embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(9):1681–1694.

Wozniak, J., Armstrong, T., Wilde, M., Katz, D., Lusk, E., and Foster, I. (2013). Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 95–102.