Exploring Dynamic Program Behavior with Frames and Phases

Divino César, Guido Araújo, Edson Borin
Institute of Computing
University of Campinas - UNICAMP
Campinas - São Paulo, Brasil
cesar@lsc.ic.unicamp.br, {guido,edson}@ic.unicamp.br

***Resumo*—The kind and amount of hardware resources demanded for the efficient execution of different programs are not the same. In fact, even the same program may have different requirements at different moments during its execution. However, current computers hardware is not designed to adapt itself during the execution of programs.**

The continuous need for energy efficient computation and the ever-decreasing size of transistors will enable the design and manufacturing of smart processors capable of adapting itself to meet the needs of the executing software. In this work, we investigate how program phase analysis and dynamic code optimization can be combined to achieve these goals. We first propose and evaluate a technique to perform online program phase detection based on the execution of frames built by the rePLay framework and then we show how the phases information can be used to improve the effectiveness of the rePLay framework.

***Keywords*-program phases; dynamic optimization; dynamic reconfiguration;**

I. Introdução

Fases são períodos durante a execução do programa, não necessariamente adjacentes, em que o desempenho de algum componente de hardware é aproximadamente o mesmo. Fases são a expressão de diferentes regiões de código sendo executadas e dos recursos que estas regiões utilizam durante a execução. Consequentemente a detecção de fases (mudanças de fase) tem implicações tanto na otimização do hardware [1]–[9] quanto em otimizações dinâmicas no código da aplicação [10]–[12].

A Figura 1 é um bom exemplo do comportamento fásico dos programas. Ela mostra a variação do desempenho de diversos componentes do *hardware* durante a execução do programa wave5 do *benchmark* SPEC CPU 2000 [13]. Cada ponto da abscissa representa 100 milhões de instruções executadas. A ordenada da esquerda representa as taxas de ocupação do *Reorder Buffer* (**Ruu**), erro de predição de valores (**Val**), erro na predição de alvo de desvio (**Addr**) e falta no acesso à *cache* de dados (**Data**). A ordenada da direita representa a taxa de erro na *cache* de instruções (**Inst**), a taxa de erro na predição de desvios (**Branch**) e o número de instruções executadas por ciclo (**IPC**). Nota-se que o comportamento não é estável durante toda a execução do programa, sendo alguns momentos caracterizados por alta localidade de acesso à memória (baixa taxa de falta na *cache* de dados), e outros com pouco potencial para paralelismo de instruções (baixo número de instruções por ciclo) . Nota-se também que, embora a variação de desempenho não seja na mesma proporção ou direção, existe correlação entre o desempenho dessas unidades. Os períodos em que o desempenho é aproximadamente o mesmo, são considerados como fases de execução. Fases podem ter tamanhos variáveis e são recorrentes.

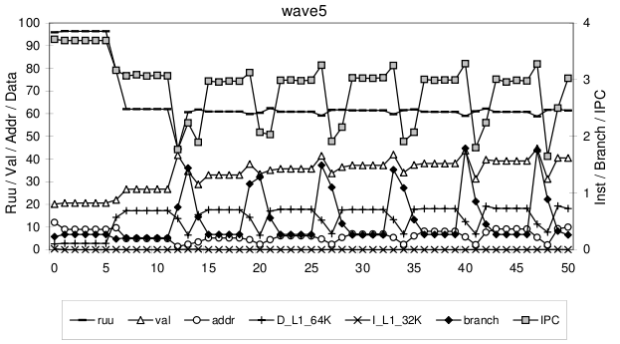


Figura 1: Comportamento fásico do programa wave5. Fonte [14].

Embora os programas apresentem comportamento fásico, os computadores, em geral, não são projetados para tirar proveito dessa característica. Entretanto, a crescente demanda por desempenho e eficiência no consumo de energia dos sistemas de computação requer que a utilização do *hardware* seja cada vez mais eficiente. Dessa forma, soluções que levam em conta o comportamento instantâneo das aplicações se tornarão cada vez mais comuns. Um exemplo de situação em que informações sobre fases pode ser útil é na reconfiguração de *caches*. Considere que em uma dada fase o programa apresenta baixa localidade, tendo essa informação disponível torna-se possível diminuir o tamanho da *cache* para reduzir o consumo de energia [1], [3], [6], [8]. Outro exemplo é a reconfiguração da largura da unidade de busca e despacho de instruções. Nas fases em que o programa não apresentar um nível razoável de paralelismo de instruções [8], [9] é possível reduzir a largura da unidade de busca e despacho do processador para diminuir o consumo de energia sem degradar o desempenho da execução. Adicionalmente, a captura de fases pode ser utilizada para direcionar otimizações de código [10], [11] e reduzir o tempo gasto em simulações [5], [7].

O *rePLay* [15] é um *framework* de tradução dinâmica de binários que baseia-se na correlação entre as instruções de desvio do programa para formar e otimizar regiões

978-0-7695-4847-0/12 $26.00 © 2012 IEEE
DOI

atômicas de código para melhorar o desempenho e reduzir o consumo de energia do processador. Estas regiões, chamadas de *frames*, são traços de instruções que incluem instruções de salto condicional. Sempre que o fluxo de controle não pode atingir o final do traço, seja por um desvio condicional ou uma exceção, o *hardware* descarta o estado especulativo e reexecuta o código original. Devido a essas propriedades, os *frames* apresentam novas oportunidades para otimizadores de código e o *framework* pode ser uma boa alternativa para permitir a realização de processadores mais eficientes. No entanto, para que as otimizações aplicadas sejam efetivas os *frames* devem cobrir uma parte significante da execução das instruções e a taxa de finalização dos *frames* deve ser alta.

Neste trabalho investigamos a relação entre a formação e a execução de *frames* e as fases dos programas. Primeiramente descrevemos como uma máquina que implementa o *framework rePLay* pode ser modificada para fazer a detecção de fases em tempo de execução. Em seguida, avaliamos o efeito das fases do programa na execução dos *frames*.

O texto está organizado da seguinte maneira. A próxima seção discute os trabalhos relacionados. A Seção III apresenta uma visão geral sobre o funcionamento do *framework rePLay*. A Seção IV descreve uma abordagem para captura de fases que se baseia na análise do conjunto de blocos básicos executados. A Seção V apresenta uma proposta de aplicação de uma técnica de captura de fases para aumentar a taxa de frames completados com o *framework rePLay*. Na Seção VI apresentamos as conclusões que obtivemos com os experimentos realizados.

II. Trabalhos Relacionados

Sherwood e Calder [14], mostraram a presença de correlação entre o desempenho de diversos componentes de *hardware*, tais como número de instruções executadas por ciclo e a taxa de acerto na predição de desvios. Em um trabalho subsequente, Sherwood e outros [7] propõem a técnica *Basic Block Vectors*, ou BBVs, capaz de identificar as fases de execução de um programa, monitorando apenas o fluxo dinâmico de instruções. A ideia é dividir o período de execução do programa em intervalos de tamanho fixo. Após a execução do programa cada intervalo é comparado com os demais produzindo assim uma "matriz de similaridade".

Em uma extensão do trabalho anterior [7], Sherwood e outros [16] propõem a utilização de técnicas de mineração de dados para fazer a detecção de fases com o objetivo de reduzir o tempo gasto em simulações de *hardware*. A ideia proposta é aplicar o algoritmo *K-means* para criar grupos de intervalos baseados em suas similaridades. Cada grupo corresponde a uma fase. Em seguida os autores mostram que, fazendo a simulação de apenas alguns intervalos por fase, é possível fazer uma estimativa confiável do comportamento de todo o programa. Estas técnicas serviram como base para diversos trabalhos [2], [9], [17] subsequentes.

Balasubramonian e outros [1] utilizam uma abordagem baseada em contadores de *hardware* para delinear as fases pelas quais o programa passa e reconfigurar a hierarquia de memória do sistema de acordo com as necessidades da aplicação em cada fase.

Dhodapkar e Smith [2], [18] fizeram um estudo comparativo entre diversas técnicas de detecção de fases e propuseram uma nova abordagem para captura de fases. As técnicas foram avaliadas utilizando vários critérios e a principal conclusão do trabalho foi a de que a técnica de BBV tem maior sensibilidade que as demais e produz fases com menor variação de desempenho. Eles ainda propuseram uma nova abordagem para captura de fase baseada no *Instruction Working Set* [18] e aplicaram a técnica na reconfiguração de um *hardware* multi-configurável.

Vijayn e Panomarev [9] também utilizam intervalos de execução fixos e vetores de *bits* para fazer a captura e predição de fases. Porém a ideia apresentada é consideravelmente mais simples que as abordagens anteriores, mais eficiente em termos de espaço e tempo de processamento e produz resultados equivalentes. Em vez de utilizar todas as instruções do intervalo para formar a assinatura, os autores propõem a utilização apenas das instruções de desvio. O restante do trabalho é desenvolvido de forma similar ao apresentado por Sherwood e outros [8].

Nagpurkar e outros [19] abstraem os conceitos utilizados nas técnicas apresentadas anteriormente e propõem um *framework* para captura de fases. A entrada de dados do *framework* é um conjunto de elementos de perfil da execução e seus componentes principais são uma unidade de computação de similaridade e uma de análise de similaridade. A saída do *framework* é uma sequência de elementos T, que indicam a transição entre fases, ou P que indicam que o comportamento está estável.

Lau e outros [20] investigam o uso de diversas estruturas ao nível de programa para fazer a captura de fases. As principais conclusões do trabalho foram que *Loop Frequency Vectors* produzem fases com variação de desempenho intra fase equivalentes àquelas obtidas utilizando-se *Basic Block Vectors* e que *Register Bit Vectors* produzem fases com menor variância que aquelas obtidas com BBVs além de exigirem menor espaço de armazenamento.

Em comum todos estes trabalhos têm o fato de proporem uma nova abordagem para detecção de fases ou a aplicação desta como base para alguma ação. Este trabalho difere dos demais no sentido de avaliar como a detecção de fases pode ser feita utilizando recursos de *hardware* já disponíveis no *framework* do *rePLay* e como as fases interferem na execução dos *frames*.

III. O *Framework RePLay*

O *rePLay* é um *framework* arquitetural que explora o comportamento dinâmico das aplicações para formar regiões de código com alto potencial de otimizações. O conceito central no *rePLay* é uma estrutura chamada de *frame*. O *frame* é uma região de código constituída por instruções coletadas a partir da execução de uma sequência de blocos básicos, ou um traço. Esta região difere de traços por ser executado atomicamente. Para permitir a execução de forma atômica, o *hardware* utiliza o

mecanismo de especulação, comum em arquiteturas com execução de instruções fora de ordem, para guardar os resultados gerados pelo *frame* até que o mesmo termine. Ao término do *frame*, caso todas as instruções foram executadas normalmente, o *hardware* grava os resultados de forma definitiva. Caso contrário, os resultados gerados pelo *frame* são descartados.

O *framework* é constituído por 4 componentes principais acoplados a uma unidade central de processamento, como mostrado na Figura 2.

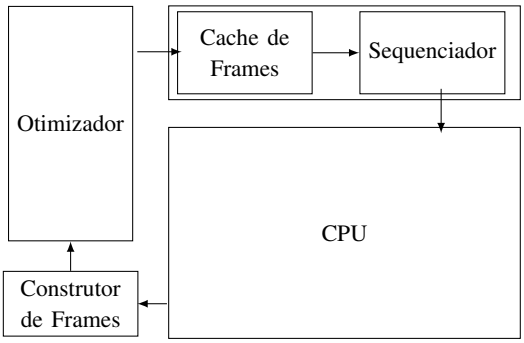


Figura 2: O *framework rePLay*

Toda instrução que termina de ser executada é encaminhada para a unidade de construção de *frames*, uma das funções desta unidade é identificar instruções de desvio que possam ter seu desvio facilmente predito pelo preditor de desvios. Para identificar a correlação entre as instruções de desvio duas tabelas são utilizadas: a CBBT (*Conditional Branch Bias Table*) e a IBBT (*Indirect Branch Bias Table*). Ambas armazenam contadores indicando a frequência que cada instrução de desvio seguiu (consecutivamente) em uma dada direção (desvio tomado ou não) em função do histórico dos desvios anteriores. Estas tabelas são indexadas a partir de uma função de espalhamento computada sobre os endereços dos alvos de desvios recentes. Durante a construção dos *frames*, quando uma instrução de desvio é encontrada, esta tabela é consultada para verificar a frequência com que o desvio seguiu na mesma direção, se esse valor for acima de um limiar a instrução é considerada "tendenciosa" e é incluída no *frame* caso contrário a construção do *frame* é encerrada.

As instruções são adicionadas ao *frame* em construção até ele atingir um tamanho máximo ou encontrar uma instrução de desvio que não seja tendenciosa. *Frames* que possuem um número mínimo de instruções e/ou blocos básicos são redirecionados para uma unidade de otimização de *frames* e em seguida são armazenados em uma *cache* de *frames*.

A unidade de sequenciamento é a responsável por disparar a execução dos *frames* no momento apropriado. Para fazer isto, ela verifica se o contexto de execução atual (histórico recente de alvos de desvio) é o mesmo daquele de quando o *frame* foi construído e informa à CPU se a execução deve seguir a partir da busca de um *frame* da *cache* de *frames* ou a partir da execução nativa de instruções.

A. Infraestrutura para experimentação

Para desenvolver este trabalho implementamos um simulador do *framework rePLay* utilizando a ferramenta Bochs 2.4.5 [21], um emulador de arquiteturas da família x86. Esta abordagem nos permitiu simular aplicações complexas, incluindo o sistema operacional. A Tabela I mostra o tamanho das estruturas no simulador desenvolvido.

Tabela I: Tamanho das estruturas no simulador.

CBBT	65536 entradas
IBBT	2048 entradas
Cache de *frames*	256 entradas totalmente associativa
Limiar de promoção	32
Histórico de alvos	6 últimos
Tamanho mínimo do *frame*	32 instruções ou 5 blocos básicos
Tamanho máximo do *frame*	256 instruções
Tabela de predição de *frames*	16384 entradas

Para validar a implementação utilizamos os programas 445.gobmk, 401.bzip2, 433.milc, 444.namd, 400.perlbench e 403.gcc, do *benchmark* SPEC CPU 2006 [22], com a entrada de referência e otimização base. A Tabela II apresenta os resultados que obtivemos com esta implementação. Nesta tabela, **Ini** é o total de vezes que a execução de um *frame* foi iniciada, **Com** é o número de execuções completas de um *frame* e **Cov** é o percentual de instruções executadas por *frames*.

Tabela II: Resultados com o *framework rePLay* implementado

Benchmark	Entrada	Ini.	Com.	Cov.
Gobmk	13x13.tst	2.192.010.858	85,48%	23,99%
	nngs.tst	6.162.818.145	87,00%	25,27%
	score2.tst	3.268.473.346	89,05%	28,58%
	trevorc.tst	2.166.811.691	85,84%	23,88%
	trevord.tst	3.258.451.946	87,06%	25,13%
	Média:	3.409.713.197	86.89%	25.37%
Bzip2	input.source	4.320.222.592	93,75%	44,39%
	chicken.jpg	1.751.259.164	92,47%	58,75%
	liberty.jpg	3.679.337.430	93,09%	58,99%
	in.program	6.106.053.749	93,02%	51,94%
	text.html	9.288.805.696	95,73%	54,47%
	in.combined	3.432.327.689	93,63%	46,36%
	Média:	5.029.135.726	93.61%	52.48%
Milc	su3imp.in	7.719.083.778	97,05%	61,96%
	Média	7.719.083.778	97.05%	61.96%
Namd	namd.input	21.865.120.288	96,92%	76,02%
	Média:	21.865.120.288	96.92%	76.02%
Perlbench	chkspam.pl	23.853.759.598	96,87%	57,91%
	diffmail.pl	7.767.149.990	97,01%	54,43%
	splitmail.pl	14.641.584.207	99,44%	74,01%
	Média:	15.420.831.265	97.77%	62.12%
Gcc	166.i	1.135.268.690	93,22%	56,42%
	200.i	2.678.383.817	95,21%	50,41%
	c-typeck.i	1.651.022.796	95,06%	64,83%
	cp-decl.i	1.487.819.652	92,84%	61,78%
	expr.i	1.474.779.410	95,62%	64,30%
	expr2.i	1.824.280.934	95,71%	66,75%
	g23.i	2.360.471.396	96,32%	66,97%
	s04.i	2.205.339.557	94,91%	65,95%
	scilab.i	939.721.786	93,68%	42,10%
	Média:	1.685.454.873	94.73%	59.95%

Os resultados indicam que em média 52% das instruções executadas estão contidas em *frames*, com taxa de finalização acima de 95%. Conjecturamos que a baixa cobertura vista nos resultados do 445.gobmk seja consequência da baixa correlação entre as instruções de desvio do programa.

IV. Detecção de Fases com *Basic Block Vectors*

A técnica de captura de fases utilizando *Basic Block Vector (BBV)* [7], [8] foi uma das primeiras a ser proposta e, apesar de ser simples, em geral produz resultados mais precisos que as demais abordagens para o problema [16], [18].

A intuição utilizada em grande parte das abordagens que utilizam BBVs para captura de fase é a de que mudanças de fase estão condicionadas diretamente ao código que está sendo executado naquele instante. Para explorar essa ideia dois requisitos são necessários: 1) é preciso identificar o momento em que a execução passa para outra região do código do programa - pois uma nova fase pode estar começando; 2) é necessário saber se a região atualmente em execução já foi executada - o programa está em uma fase executada anteriormente.

Em se tratando de captura de fase, uma região é um intervalo contínuo do fluxo de instruções do programa. Para saber se a região executada mudou ou se uma região executada previamente está sendo executada novamente, é preciso fazer uma comparação entre os intervalos. Porém, para fazer uma comparação precisamos de identificadores para os objetos comparados, portanto, também por um requisito de viabilidade e eficiência, uma assinatura precisa ser gerada para cada intervalo. Essa assinatura serve como um "resumo" da região de código executada durante o intervalo. Uma vez gerada a assinatura podemos fazer a comparação usando, por exemplo, distância manhattan ou euclidiana [16].

A. Basic Block Vector

Um bloco básico é uma sequência de instruções com apenas um ponto de entrada e apenas um ponto de saída [23]. Sherwood e outros [7], [8], [16] propuseram a utilização de um vetor de frequência de blocos básicos, o *Basic Block Vector* (BBV), como assinatura para os intervalos de execução do programa. Um BBV é um vetor que possui uma relação de um-para-um com os blocos básicos do programa alvo. A posição i de um BBV armazena a quantidade de vezes que o bloco básico B_i foi executado naquele intervalo multiplicada pelo número de instruções no bloco.

Manter um vetor com estas dimensões, principalmente em uma implementação em *hardware*, pode não ser viável. Por este motivo, frequentemente é utilizada alguma forma de espalhamento para permitir o armazenamento do BBV em uma estrutura menor [8].

B. Matriz de Similaridade

Uma matriz de similaridade é uma forma conveniente de visualizar a estrutura de fases de um programa. Nesta matriz cada linha e coluna representa um intervalo e estão dispostos na ordem de execução do programa. Cada célula (i, j) da matriz representa o nível de similaridade entre os intervalos i e j em escala de cinza. Quanto mais escuro o valor, maior a similaridade. A Figura 3 apresenta a matriz de similaridade da execução da aplicação gzip, do benchmark SPEC CPU 2000, com a entrada de dados *graphic*.

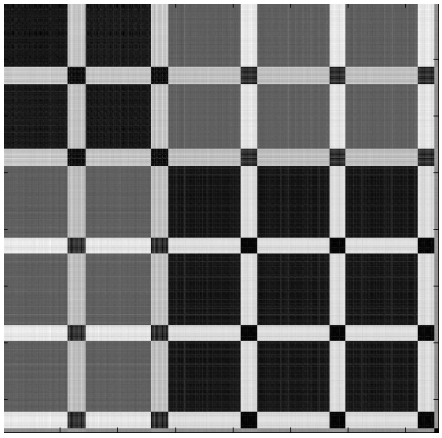


Figura 3: Matriz de similaridade para a execução da aplicação gzip com a entrada graphic

Para analisar a similaridade entre os intervalos considere inicialmente a diagonal principal da matriz. Para ver como o intervalo i relaciona-se com os próximos k intervalos, analise o segmento de reta formado pelos pontos com extremos em (i, i) e (i, k) - uma linha horizontal. Para ver a similaridade do intervalo i com os l intervalos anteriores analise o segmento de reta $(i, i)(l, i)$ - uma linha vertical. Veja que qualquer ponto da diagonal principal relaciona-se com uma sequência de intervalos subsequentes e/ou precedentes formando blocos de similaridade. Além disso, note que um intervalo i pode ser similar aos próximos k intervalos (formando um bloco), não similar aos intervalos de $k{+}1$ à $k{+}u$ e novamente similar aos intervalos $k{+}u{+}1$ à $k + u + v$, portanto caracterizando um comportamento recorrente de similaridade.

A recorrência destes blocos de similaridade é entendida como sendo o programa executando a mesma região de código em instantes de tempo diferentes. Como consideramos que uma fase é reflexo direto do código sendo executado, nestes dois instantes de tempo a execução do programa deve ser classificada como executando na mesma fase. Portanto, uma fase é um conjunto de intervalos que são similares mesmo que estes estejam temporalmente não adjacentes na execução do programa.

C. Captura de fases em tempo de execução

Sherwood e outros [8] apresentaram ainda um mecanismo de *hardware* para captura de fases durante a execução do programa. A técnica proposta é baseada em intervalos de tamanho fixo e usa BBV como assinatura para os intervalos. A Figura 4 mostra a arquitetura do mecanismo proposto.

Para tornar viável a implementação em hardware, uma versão reduzida do BBV é utilizada. Uma função de

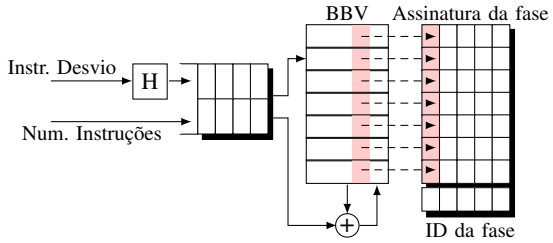


Figura 4: Mecanismo de captura de fases proposto por Sherwood e outros. Fonte: [8]

espalhamento é aplicada no endereço das instruções de desvio e o resultado é utilizado como índice para acessar uma posição do BBV que está sendo construído para o intervalo atual. Essa posição do vetor é incrementada com o número de instruções do bloco básico que foi mapeado.

Ao final da execução de um intervalo é preciso classificá-lo em uma fase já observada ou considerá-lo como representante de uma nova fase. Para fazer isso uma tabela com intervalos representando cada uma das fases já identificadas é mantida durante a execução do programa. Quando um intervalo precisa ser classificado, é feita a comparação de sua assinatura com a assinatura de cada um dos intervalos mantidos nesta tabela. Se a similaridade for acima de um limiar o intervalo é atribuído à fase que aquele intervalo representa, se nenhum dos intervalos atingir este limiar uma nova fase é "criada" adicionando-se o intervalo atual à tabela de intervalos representativos de fases.

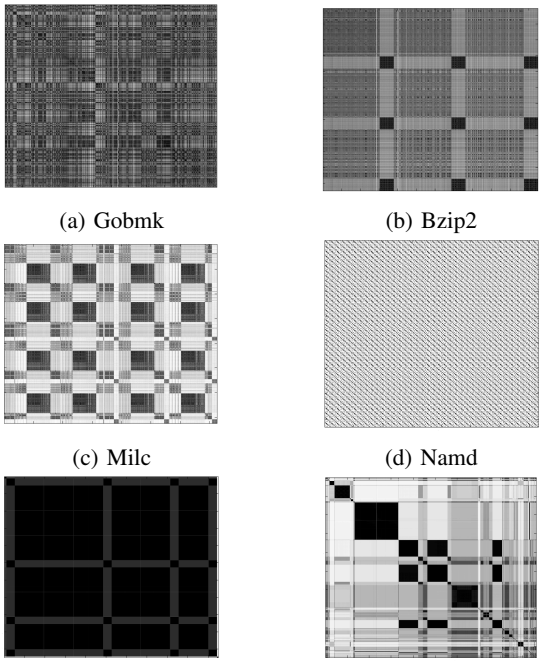
D. Infraestrutura para experimentação

Como parte deste trabalho desenvolvemos uma ferramenta para gerar BBVs, matrizes de similaridade e fazer a detecção de fases em tempo de execução. A implementação foi feita utilizando-se o emulador da arquitetura x86 Bochs 2.4.5 [21]. As Figuras 5a à 5f apresentam as matrizes de similaridade para os programas 445.gobmk-13x13.tst, 401.bzip2-input.source, 433.milc, 444.namd, 400.perlbench-diffmail.pl e 403.gcc-s04.i, do benchmark SPEC CPU 2006. Para permitir a representação em escala reduzida, cada célula das matrizes apresentadas corresponde à média aritmética de 4 células adjacentes da matriz de similaridade original.

Para coletar estes dados instrumentamos o emulador para gerar BBVs a cada 100 milhões de instruções executadas. Cada *benchmark* foi compilado com as otimizações base e executado com sua respectiva entrada de referência. Para programas que possuem mais de uma entrada mostramos a matriz gerada para a entrada que produziu um padrão fásico evidente. O padrão fásico é facilmente perceptível nas matrizes de similaridade, exceto na matriz do *benchmark* 444.Namd, que é composto de uma pequena fase que se repete durante toda a execução do *benchmark*.

V. *Frames* E FASES

A partir dos resultados obtidos com o *framework rePLay* e com a detecção de fases com BBVs, iniciamos



(a) Gobmk (b) Bzip2

(c) Milc (d) Namd

(e) Perlbench (f) Gcc

Figura 5: Validação da captura de fases utilizando *Basic Block Vectors*

uma investigação para determinar se as fases poderiam ser detectadas a partir do uso de *frames* e como a construção de *frames* por fase afetaria os resultados do *rePLay*.

Para realizar a captura de fases utilizando *frames* seguimos uma abordagem similar à descrita na Seção IV. Inicialmente o fluxo de instruções é dividido em intervalos de tamanho fixo de 100 milhões de instruções, usamos os *frames* que completaram execução para atualizar a assinatura do intervalo. Para gerar o índice da assinatura que deve ser atualizado aplicamos um *ou exclusivo* do endereço estendido da primeira instrução do *frame* com o vetor de *bits* que indica a direção das instruções de desvios internas ao *frame*. Em seguida, uma função de espalhamento é aplicada neste resultado para obter-se o índice da assinatura que deverá ser atualizada. A atualização é feita incrementando-se a entrada correspondente ao índice com o número de instruções contidas no *frame*.

Note que gerando a assinatura desta forma minimizamos a sobrecarga no *framework* do *rePLay*, pois a assinatura é atualizada apenas quando o *frame* conclui a execução, além disso a sobrecarga tende a ser reduzida em relação à abordagem de detecção de fases que usamos como base, porque nesta a assinatura é atualizada a cada bloco básico executado.

Para realizar a construção de *frames* por fase utilizamos uma implementação de captura de fases em tempo de execução similar à descrita na Seção IV. Durante a execução do programa uma tabela é utilizada para manter as estatísticas de cada fase. Cada linha i da tabela corresponde às estatísticas coletadas durante a execução da fase F_i. Quatro colunas são utilizadas, três delas são

exatamente iguais às últimas colunas da Tabela II e mais uma coluna para armazenar a quantidade de intervalos que foram classificados na fase F_i.

Durante a execução de uma fase o funcionamento do *rePLay* prossegue da forma convencional. Quando há uma mudança de fase, as estatísticas da fase anterior são persistidas juntamente com a configuração do *framework* . Se o programa já havia executado na fase atual as configurações e estatísticas do *framework* para aquela fase são restauradas.

Como não fazemos predição de fases, sabemos que o programa mudou de fase somente após a mudança ocorrer, ou seja, somente após a execução do primeiro intervalo da nova fase. Isto significa que começamos a coletar as estatísticas da fase a partir do seu segundo intervalo. O impacto disso nos resultados é mínimo pois em geral fases muito pequenas são na verdade transições entre fases e devem ser desconsideradas da análise, além disso, fases grandes (como as que reportamos na Tabela III) são compostas por dezenas de milhões de intervalos, e a remoção de um intervalo não afetaria significativamente as estatísticas. Nos experimentos que realizamos o tamanho do intervalo foi fixo em 100 milhões de instruções e um limiar de similaridade de 15 milhões de instruções foi utilizado. Nota-se que estes valores estão nas mesmas proporções daqueles utilizados no trabalho de Sherwood *et. al* [8]. Da mesma forma, o cálculo da similaridade entre os intervalos foi realizado utilizando-se distância manhattan.

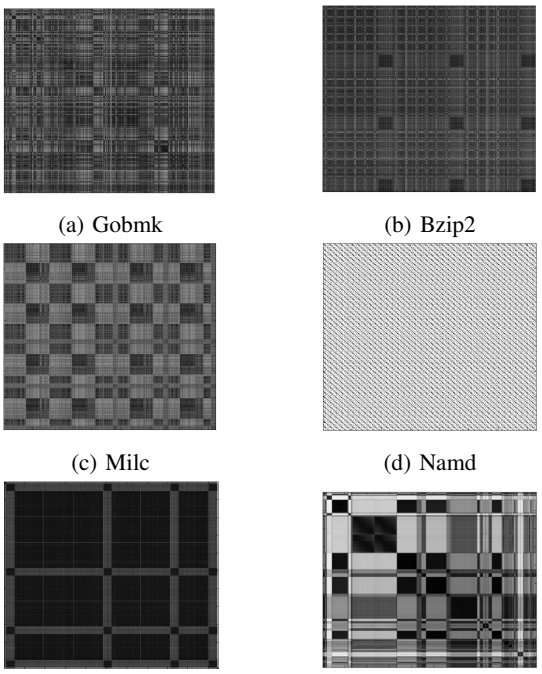
Nos próximos parágrafos descrevemos a metodologia que empregamos para implementar esta idéia e os resultados que obtivemos.

Todos os experimentos deste trabalho utilizaram os seguintes programas do *benchmark* SPEC 2006: 445.gobmk, 401.bzip2, 433.milc, 444.namd, 400.perlbench e 403.gcc. As aplicações foram compiladas com o compilador GCC 4.4.5 e executadas com as entradas de referência.

Para fazer a execução dos experimentos utilizamos o emulador Bochs 2.4.5 [21], executando uma imagem do sistema operacional Debian 5. Visto que o Bochs é uma máquina virtual de sistema, para que o código de instrumentação desconsidere as instruções de inicialização do sistema operacional, implementamos no simulador um gatilho que é ligado imediatamente antes da execução do *benchmark* e desligado após o término do mesmo.

As Figuras 6a à 6f apresentam as matrizes de similaridades geradas quando *frames* são utilizados para construir as assinaturas dos intervalos. Comparando os resultados com os obtidos com a técnica de BBVs, podemos observar que a abordagem empregada foi capaz de detectar os mesmos padrões de fases que a técnica *Basic Block Vectors*. Isto mostra que utilizando apenas informações relativas aos *frames*, que foram de fato completados durante os intervalos, é suficiente para gerar uma assinatura que possa ser utilizada para distingui-los.

Nota-se que as figuras estão mais escuras em relação às apresentadas na Seção IV, isto é devido à grande diferença entre o número de *frames* e o número de blocos básicos usados para gerar as matrizes nos dois casos. Em geral, quanto mais expressiva for a assinatura (figuras mais claras), maior tende a ser a sobrecarga no *framework* para gerar a assinatura (custo maior com perfilamento de código). Portanto, a abordagem que propomos se enquadra como um meio termo entre baixa sobrecarga de perfilamento do código e alta expressividade das assinaturas dos intervalos.



(a) Gobmk (b) Bzip2

(c) Milc (d) Namd

(e) Perlbench (f) Gcc

Figura 6: Captura de fases utilizando *frames*

Como trabalho futuro pretendemos conduzir uma abordagem analítica para comparação entre as técnicas de delineamento de fases apresentadas, pretendemos utilizar, por exemplo, a técnica de análise de covariancia [18] e/ou a análise de intervalo de confiança da média estimada [24].

Além de investigar a detecção de fases a partir da execução dos *frames*, verificamos se a taxa com que os *frames* são completados é afetada se mantivermos o estado arquitetural do rePLay orientado por fases. A ideia por trás disto é a de que expondo o comportamento estável das fases ao *framework* e persistindo a configuração por região de código (fase), as unidades componentes do *rePLay* não seriam afetadas por mudanças repentinas de código.

A Tabela III mostra os resultados da nossa implementação do *rePLay* para as cinco maiores fases de cada *benchmark* avaliado. A coluna **Cov.** representa o percentual de intervalos da execução da aplicação que foi classificado na fase; a coluna **Ini** o número de *frames* inicializados na fase e a coluna **Comp.** o percentual de *frames* inicializados que completaram a execução.

Os resultados estão próximos aos obtidos com o fluxo normal de operação do *framework*, mostrado na Tabela II. No entanto, obtivemos um ganho médio de 2% na taxa com que os *frames* são completados. Dado o grande

Tabela III: *frames* por Fase

Benchmark	Ini.	Cov.	Com.
Gobmk-13x13.tst	788.106.331	36,18%	88,46%
	507.583.725	24,13%	86,65%
	325.872.518	15,01%	85,74%
	124.776.860	6,06%	85,83%
	76.756.326	3,55%	89,37%
	Média:		87.21%
Bzip2-input.source	616.150.224	19,60%	87,30%
	222.633.430	4,29%	96,35%
	220.628.722	4,12%	95,80%
	210.297.380	3,94%	96,44%
	165.082.684	3,72%	96,24%
	Média:		94.43%
Milc-su3imp.in	629.858.395	8,65%	99,61%
	569.027.829	6,97%	96,82%
	392.385.617	5,01%	99,68%
	336.251.066	4,47%	97,57%
	263.766.112	3,81%	99,47%
	Média:		98.63%
Namd-namd.input	2.315.984.934	10,48%	96,22%
	2.064.156.187	9,45%	96,48%
	2.041.971.436	8,80%	96,42%
	1.370.728.313	6,92%	96,58%
	1.403.164.641	6,69%	96,33%
	Média:		97.00%
Perlbench-diff.pl	7.174.395.628	81,93%	97,72%
	1.180.887.853	15,57%	97,20%
	80.546.965	0,88%	95,93%
	3.921.042	0,55%	99,61%
	5.890.074	0,12%	99,45%
	Média:		97.98%
Gcc-s04.i	696.218.057	22,14%	99,72%
	151.213.378	18,04%	98,72%
	75.538.429	7,66%	91,58%
	218.998.650	7,05%	92,33%
	205.466.606	6,22%	92,81%
	Média:		95.03%

número de *frames* que são inicializados, isto significa que em alguns casos milhões de *frames* a mais completarão execução.

VI. Conclusões

Em um único segundo um processador pode executar bilhões de instruções e neste intervalo o comportamento do programa pode mudar drasticamente. Períodos em que o programa apresenta alto nível de paralelismo podem ser seguidos de momentos em que o fluxo de instruções é totalmente sequencial ou períodos de baixa localidade seguidos de períodos com alta localidade, etc.

Embora os programas apresentem este comportamento fásico, em geral os sistemas de computação atuais possuem uma implementação pouco flexível, com pouca capacidade para se adaptarem às diferentes fases de execução das aplicações. Neste trabalho propomos uma variação da técnica *Basic Block Vectors* para detecção de fases no *framework rePLay* e investigamos a utilização da informação de fases para aumentar o uso e a taxa de finalização dos *frames* no *framework rePLay*.

Nossos resultados indicam que a detecção de fases através do monitoramento da execução de *frames* é uma alternativa viável para a técnica de captura de fases baseada em *Basic Block Vectors* e pode ser empregada de forma efetiva em uma máquina que já implementa o *framework rePLay*. Além disso, mostramos que é possível aumentar em 2% a taxa de finalização dos *frames* se levarmos em conta as fases do programa durante a geração dos *frames* no *framework*.

Como trabalho futuro pretendemos medir a sobrecarga da integração entre as duas técnicas e avaliar como as informações relativas a fases podem ser utilizadas nas otimizações aplicadas pelo *rePLay*.

VII. Agradecimentos

Agradecemos à Fundação de Amparo a Pesquisa do Estado de São Paulo (FAPESP) pelo apoio financeiro recebido para realizar este trabalho - processos número 2011/05028-5 e 2011/00901-2.

Referências

[1] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 245–257.

[2] A. S. Dhodapkar and J. E. Smith, "Managing multi-configurable hardware via dynamic working set analysis," in *In 29th Annual International Symposium on Computer Architecture*, 2002, pp. 233–244.

[3] A. Gordon-Ross, J. Lau, and B. Calder, "Phase-based cache reconfiguration for a highly-configurable two-level cache hierarchy," in *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, ser. GLSVLSI '08. New York, NY, USA: ACM, 2008, pp. 379–382.

[4] J. Kim, S. Yoo, and chong Min Kyung, "Program phase-aware dynamic voltage scaling under variable computational workload and memory stall environment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, pp. 110–123, 2011.

[5] P. Ratanaworabhan and M. Burtscher, "Program phase detection based on critical basic block transitions," in *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, 2008, pp. 11–21.

[6] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," pp. 165–176, 2004.

[7] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, 2001, pp. 3–14.

[8] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," *SIGARCH Comput. Archit. News*, vol. 31, pp. 336–349, May 2003.

[9] B. Vijayn and D. Ponomarev, "Accurate and low-overhead dynamic detection and prediction of program phases using branch signatures," in *Computer Architecture and High Performance Computing, 2008. SBAC-PAD '08. 20th International Symposium on Computer Architecture and High Performance Computing*, 2008, pp. 3–10.

[10] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W.-m. W. Hwu, "Vacuum packing: extracting hardware-detected program phases for post-link optimization," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 35. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 233–244.

[11] J. Lu, H. Chen, P. Yew, and W. Hsu, "Design and implementation of a lightweight dynamic optimization system," *Journal of Instruction-Level Parallelism*, vol. 6, pp. 1–24, 2004.

[12] Y. Wu, M. Breternitz, J. Quek, O. Etzion, and J. Fang, "The accuracy of initial prediction in two-phase dynamic binary translators," *International Symposium on Code Generation and Optimization*, 2004.

[13] J. L. Henning, "Spec cpu2000: measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, jul 2000.

[14] T. Sherwood and B. Calder, "Time varying behavior of programs," University of California, Tech. Rep., 1999.

[15] S. J. Patel and S. S. Lumetta, "replay: A hardware framework for dynamic optimization," *IEEE Transactions on Computers*, vol. 50, pp. 590–608, 2001.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-X. ACM, 2002, pp. 45–57.

[17] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder, "Motivation for variable length intervals and hierarchical phase behavior," in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, 2005, pp. 135–146.

[18] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003, pp. 217–227.

[19] P. Nagpurkar, C. Krintz, M. J. Hind, P. F. Sweeney, and V. T. Rajan, "Online phase detection algorithms," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2006, pp. 111–123.

[20] J. Lau, S. Schoemackers, and B. Calder, "Structures for phase classification," in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*, 2004, pp. 57 – 67.

[21] O. Source, "Bochs - the cross platform ia-32 (x86) emulator," http://bochs.sourceforge.net/, 2011, [Acessado em 17 de agosto de 2012].

[22] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, pp. 1–17, Sep 2006.

[23] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006.

[24] S. V. Kodakara, J. Kim, D. J. Lilja, D. M. Hawkins, W.-C. W.-C. Hsu, and P.-C. Yew, "Cim: A reliable metric for evaluating program phase classifications," *Computer Architecture Letters*, vol. 6, no. 1, pp. 9–12, jan 2007.