UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Programa de Pós-Graduação em Computação

Dissertação

Potencial de Reuso de Traços em Arquiteturas ARM

Rodrigo Costa de Moura

Pelotas, 2015

Rodrigo Costa de Moura

Potencial de Reuso de Traços em Arquiteturas ARM

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Mestre em Ciência da Computação

Orientador: Prof. Dr. Maurício Lima Pilla
Coorientador: Prof. Dr. Laércio Lima Pilla

Pelotas, 2015

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação na Publicação

M929p Moura, Rodrigo Costa de

Potencial de reuso de traços em arquiteturas ARM / Rodrigo Costa de Moura ; Maurício Lima Pilla, orientador ; Laércio Lima Pilla, coorientador. — Pelotas, 2015.

63 f. : il.

Dissertação (Mestrado) — Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2015.

1. Reuso de valores. 2. Reuso de traços. 3. Arquitetura ARM. I. Pilla, Maurício Lima, orient. II. Pilla, Laércio Lima, coorient. III. Título.

CDD : 005

Elaborada por Aline Herbstrith Batista CRB: 10/1737

UNIVERSIDADE FEDERAL DE PELOTAS
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
CENTRO DE DESENVOLVIMENTO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

PPGC
Programa de Pós-Graduação em Ciência da Computação

ATA DE APRECIAÇÃO SOBRE A DEFESA DE DISSERTAÇÃO DE MESTRADO

NOME DA ESTUDANTE		MATRÍCULA
RODRIGO COSTA DE MOURA		12121019
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO		MESTRADO

MEMBROS DA BANCA EXAMINADORA	TÍTULO	ASSINATURA
PROF. MAURÍCIO LIMA PILLA (ORIENTADOR)	Doutor	
PROF. LAÉRCIO LIMA PILLA (CO-ORIENTADOR, UFSC)	Doutor	
PROF. EDUARDO ANTONIO CESAR DA COSTA (UCPEL)	Doutor	
PROF. ADENAUER CÔRREA YAMIN (UFPEL)	Doutor	

APRECIAÇÃO SOBRE A DISSERTAÇÃO

NÃO SIGILOSA

Em 30 de novembro de 2015 os membros acima nomeados para a defesa da Dissertação da **RODRIGO COSTA DE MOURA,** matriculado no Programa de Pós-Graduação em Computação, consideraram aprovada, estabelecendo o título definitivo da Dissertação como sendo " Potencial de Reuso de Traços em Arquiteturas ARM", e estabelecendo um prazo máximo de 30 dias para as correções e entrega da versão definitiva.

DADOS PESSOAIS DOS MEMBROS DA BANCA EXAMINADORA

NOME COMPLETO	CPF	ANO NASCIMENTO	TITULAÇÃO		
			Área	Local	Ano
MAURÍCIO LIMA PILLA	933.456.580-20	24/12/1976	CIÊNCIA DA COMPUTAÇÃO	UFRGS	2004
LAÉRCIO LIMA PILLA	015.436.430-43	26/05/1987	CIÊNCIA DA COMPUTAÇÃO	UFRGS	2014
EDUARDO ANTONIO CESAR DA COSTA	639.797.564-91	24/09/1964	CIÊNCIA DA COMPUTAÇÃO	UFRGS	2002
ADENAUER CÔRREA YAMIN	291.222.010-68	04/09/1959	CIÊNCIA DA COMPUTAÇÃO	UFRGS	2004

1ª Via – Coordenador do Curso; 2ª Via – Orientador; 3ª Via – PRPPG.
DISTRIBUIÇÃO A CARGO DA COORDENAÇÃO DO PROGRAMA.

AGRADECIMENTOS

Chegando neste momento, é preciso agradecer todos aqueles que contribuíram com a conclusão desta etapa. Primeiramente agradeço aos meus pais, por me mostrarem desde cedo a importância do estudo e da busca por crescimento pessoal.

Devo agradecimentos também aos amigos do laboratório LUPS, colegas ou professores, pelas trocas de conhecimento que contribuíram com o trabalho e principalmente por terem tornado esse período bastante divertido.

Agradeço ao meu orientador, Professor Pilla, pelos ensinamentos, por toda paciência neste longo período e pelas conversas sobre cerveja.

Será que eu fiz alguma coisa de errado hoje
ou será que o mundo sempre foi assim,
só que eu estava encucado demais para perceber?
— ARTHUR DENT, ENQUANTO OLHAVA PARA DENTRO DO CANECO.

RESUMO

MOURA, Rodrigo Costa de. **Potencial de Reuso de Traços em Arquiteturas ARM**. 2015. 63 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2015.

O constante aumento da necessidade de alto poder de processamento faz com que os projetos computacionais fiquem cada vez mais complexos. Como há grande variedade de aplicações, desenvolver *hardware* dedicado muitas vezes é inviável. Paralelamente ao aumento do poder computacional, pode ocorrer o aumento do consumo de energia. Dessa forma, desenvolveram-se técnicas que visam atender a demanda por alto poder de processamento e também para reduzir o consumo de energia. Uma dessas estratégias se refere ao Reuso de Valores, onde são exploradas as partes recorrentes e previsíveis que compõem os programas. O objetivo destas técnicas é evitar a re-execução de instruções já conhecidas, reduzindo o número total de instruções executadas e mantendo o contexto original da aplicação.

Técnicas de Reuso de Valor memorizam as execuções anteriores, sejam de instruções, blocos ou traços, com a esperança de reutilizá-las quando estas surgirem novamente com os mesmos contexto de entrada. Mesmo com o grande potencial apresentado pelas técnicas de reuso, tanto em aumento do poder de processamento quando na redução do consumo energético, o emprego do reuso de valores não foi avaliado em uma das arquiteturas de maior popularidade: a arquitetura ARM.

Os processadores ARM são facilmente encontrados em dispositivos móveis como celulares, *smartphones*, *tablets* e calculadoras. Essa categoria de dispositivos demanda cada vez mais por poder de processamento, porém, sempre mantendo o compromisso com o baixo consumo de energia.

Neste trabalho, são avaliadas as características da arquitetura ARM e seu conjunto de instruções. Posteriormente, é analisado o potencial de reuso utilizando o conjunto de *benchmarks* MiBench. Para isso, é apresentada uma estratégia de reuso de traços e sua estrutura de armazenamento, onde são avaliadas diferentes formas de estruturar traços e os seus impactos em *buffers* de diferentes tamanhos. Os testes com os *benchmarks* MiBench mostram que é possível alcançar 18,4% de reuso médio com uso de um *buffer* de traços de 32 KiB.

Palavras-chave: Reuso de Valores, Reuso de Traços, Arquitetura ARM.

ABSTRACT

MOURA, Rodrigo Costa de. **Traces Reuse Potential in ARM Architectures**. 2015. 63 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2015.

The continuous increase in the need for high processing power makes computer designs increasingly complex. Since most of the applications are general purpose, the development of dedicated hardware is often unfeasible. In parallel with the increase in computing power, energy consumption may also be increased. Thus, techniques were developed in order to provide high processing power while reducing or mantaining energy consumption. One of these strategies is called Value Reuse, which exploits recurring and predictable parts that make up most of the executed instructions. The goal of these techniques is to avoid the re-execution of known instructions, which reduces the number of executed instructions and preserves the original context of the application.

Value reuse techniques memorize previous executions of instructions, blocks, or traces, with the hope of reusing them when they arise with the same input context again. Although showing great potential for both performance and energy consumption improvement, trace reuse techniques have not been studied for one of the most widely available computer architectures yet: the ARM architecture.

ARM processors are common in mobile devices such as cell phones, smartphones, tablets and calculators. This category of devices increasingly demands for processing power, while always committed to a low power consumption.

In this work, the architectural features of ARM and its instruction set are evaluated. Afterwards, the potential of reuse of the MiBench benchmarks is analyzed. For this, a trace reuse strategy and its storage structure are presented, which are evaluated with different ways of structuring traces and their impacts in buffers of different sizes. The experiments with MiBench benchmarks show that it is possible to achieve 18.4% of reuse on average using a trace buffer of 32 KiB.

Keywords: Value Reuse, Trace Reuse, ARM Architecture.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ARM	*Advanced Risc Machine*
CISC	*Complex Instruction Set Computer*
CPSR	*Current Program Status Register*
DMA	*Direct Memory Access*
DTM	*Dynamic Trace Memoization*
FIFO	*First In, First Out*
FIQ	*Fast Interrupt Request*
FPU	*Float Point Unit*
ILP	*Instruction-Level Parallelism*
IPC	Instruções por Ciclo
IRQ	*Interrupt Request*
I2C	*Inter-Integrated Circuit*
LCD	*Liquid Crystal Display*
LRU	*Least Recently Used*
MMU	*Memory Management Unit*
OTG	*On-The-Go*
PWM	*Pulse Width Modulation*
RISC	*Reduced Instruction Set Computing*
RST	*Reuse throuth Speculation on Traces*
SPI	*Serial Peripheral Interface*
SSI	*Synchronous Serial Interface*
SVC	*Supervisor Call*
TTL	*time-to-live*
UART	*Universal Asynchronous Receiver/Transmitter*
UND	*Undefined*
USB	*Universal Serial Bus*

SUMÁRIO

1 INTRODUÇÃO

A cada momento *softwares* mais robustos são desenvolvidos, exigindo sempre processadores mais poderosos. O constante aumento da demanda por alto poder de processamento faz com que os projetos computacionais fiquem cada vez mais complexos. Como grande parte das aplicações são de uso geral, desenvolver *hardware* dedicado muitas vezes é inviável. Paralelamente ao aumento do poder computacional, acontece o aumento do consumo de energia. Dessa forma, desenvolveram-se técnicas que visam atender a demanda por alto poder de processamento, além de reduzir o consumo de energia em processadores de uso geral.

Uma forma de aumentar o desempenho de processadores de uso geral é explorando a Localidade de Valores. A Localidade de Valores, de forma geral, é a capacidade de se prever um valor que será gerado futuramente, baseando-se nos valores anteriores. De acordo com Gabbay et al. (1996), programas são inerentemente constituídos em grande parte por computação redundante e previsível. Essa característica nos permite saber previamente, durante a execução de um programa, parte dos valores que serão gerados nas próximas execuções, baseando-se apenas nas execuções anteriores. Isso é possível pois os programas, em sua maioria, são compostos por laços e chamadas de sub-rotinas, que geram padrões e repetições durante a execução.

A exploração da Localidade de Valores é utilizada em técnicas já conhecidas, como a previsão de desvios. Nesse caso, a localidade de valores é explorada analisando os desvios anteriores para adiantar os desvios futuros. Outra técnica que explora a localidade de valores é o uso de memória *cache*, a qual explora a localidade temporal e espacial de endereços na memória para minimizar o tempo dos próximos acessos a dados e instruções em memória.

Sendo possível saber o resultado de uma parcela das computações antes delas ocorrerem, é possível evitar suas execuções e economizar recursos computacionais. No trabalho feito por Sodani et al. (1998), estimou-se que das execuções de programas em processadores MIPS, em alguns casos até 90% das instruções executadas são repetições de instruções anteriores e outros 5% são apenas variações, portanto,

previsíveis. Isso significa que, neste caso, apenas 5% das instruções executadas geram novos resultados, o restante sendo apenas repetições ou variações previsíveis destas execuções. As altas parcelas de computação redundante indicam o grande potencial de economia de recursos, caso essas parcelas sejam suprimidas.

A técnica de Reuso de Valores consiste basicamente em explorar a localidade de valores no momento da execução, visando suprimir a parcela de computação redundante e previsível (GONZÁLEZ; TUBELLA; MOLINA, 1998). A tarefa de reduzir a redundância em programas é tradicionalmente encarada pelo compilador (AHO; ULLMAN; SETHI, 1995). Porém, grande parte das computações dependem de valores que serão gerados apenas durante a execução, valores esses que não podem ser identificados pelo compilador. Assim, a técnica de reuso de valores trabalha durante a execução, armazenando os resultados das execuções anteriores e comparando-os com as futuras, buscando identificar padrões e repetições. O objetivo consiste em aumentar o desempenho de processadores de uso geral diminuindo o total de instruções executadas. Isso é feito reaproveitando as sequências de execuções recorrentes para que estas não sejam re-executadas. Dessa forma pode-se atingir tanto aumento no poder de processamento quanto redução no consumo de energia, de acordo com o enfoque dado a técnica.

Para que o reuso seja possível, é necessária a adição de um *buffer* próximo ao processador no mesmo nível da memória *cache* L1 (COPPIETERS et al., 2015). Esse *buffer* deve armazenar informações sobre execuções anteriores para que seja possível compará-las com as futuras. Adicionalmente, são necessários mecanismos para testar os contextos e efetivamente reusar as instruções.

Tendo em vista a necessidade de um *hardware* adicional para a implementação da técnica de reuso, diversos fatores devem ser avaliados para garantir a viabilidade da técnica. Esses fatores dizem respeito basicamente ao formato do *buffer* e às políticas de armazenamento dos dados que o utilizam. Para avaliar a forma de uso do *hardware* adicional, deve-se considerar as características da arquitetura utilizada. O conjunto de instruções, o tamanho e a quantidade de registradores, as formas de endereçamento e outros fatores específicos de cada arquitetura necessitam ser considerados.

Mesmo com o grande potencial apresentado pelas técnicas de reuso, tanto em aumento do poder de processamento quando na redução do consumo energético (LAURINO et al., 2005; TSAI; CHEN, 2011; COPPIETERS et al., 2015), o emprego do reuso de valores não foi avaliado em uma das arquiteturas de maior popularidade: a arquitetura ARM. A categoria de dispositivos móveis demanda cada vez mais por poder de processamento, mantendo sempre o compromisso com o baixo consumo de energia. Nesses dispositivos são facilmente encontrados processadores do tipo ARM. Devido suas características de simplicidade e baixo consumo energético, os processadores ARM se tornaram referência para computação embarcada e dispositivos portáteis.

Um dos principais desafios para implementar reuso em processadores ARM está relacionado às instruções condicionais. Do conjunto de instruções ARM, essas podem ser executadas ou não de acordo com o estado de um conjunto de bits de condição no momento da execução Além disso, por estes processadores prezarem pela simplicidade e economia de energia, o reuso de valores em processadores ARM deve demandar pouco hardware adicional.

1.1 Objetivos

O objetivo principal desta dissertação é estimar o potencial da aplicação de técnicas de reuso de valores em processadores ARM. Além disso, o presente trabalho busca quantificar a redundância de execução nestes processadores, bem como determinar as formas mais adequadas de explorar esta redundância. Para isso, é feita uma análise do comportamento da execução do conjunto de *benchmarks* MiBench sobre o simulador de arquitetura ARM Sim-Panalyzer. A partir desta análise é apresentada uma estratégia de reuso no nível de traços de instruções e sua estrutura de armazenamento, onde são avaliados os impactos de limitações na construção dos traços e seus efeitos em *buffers* de diferentes tamanhos. Por fim, são apresentados os potencias de reuso de traços utilizando diferentes configurações de *buffer*.

1.2 Metodologia

A primeira etapa desta dissertação envolve uma análise bibliográfica sobre as técnicas que exploram a localidade de valores em vários níveis. Paralelamente, são avaliadas as características dos processadores ARM no que se refere ao reuso. Com isso, o trabalho apresenta o domínio de reuso, que se refere ao conjunto de instruções da arquitetura ARM que poderão ser exploradas pelas técnicas de reuso. Para a avaliação da escolha do domínio de reuso, estimativa de redundância e demais testes, o simulador de arquitetura ARM Sim-Panalyzer é utilizado, juntamente com o conjunto de *benchmarks* MiBench como carga de trabalho.

Partindo das análises da execução do simulador e dos estudos sobre técnicas de reuso e processadores ARM, é proposta uma estrutura para armazenamento de traços de instruções, bem como uma política para construção, armazenamento e reuso de traços. Com isso, são feitas modificações no simulador Sim-Panalyzer para que este intercepte as instruções executadas e possa simular o funcionamento do mecanismo de reuso de traços proposto. Assim, a presente dissertação avalia o potencial de reuso de traços, aplicando variações no tamanho do *buffer* e limitações na construção dos traços.

1.3 Resultados

Observou-se que a execução dos *benchmarks* MiBench sobre o simulador Sim-Panalyzer possui uma média de 34,1% de computação redundante. Além disso, em média, 57,3% do total de instruções não operam com ponto flutuante nem fazem acesso a memória. No trabalho apresentado, observou-se que com um mecanismo de reuso de traços, utilizando um *buffer* de 32 KiB, é possível reutilizar em média 18,4% da computação redundante existente na execução do simulador Sim-Panalyzer utilizando o conjunto de *Benchmarks* MiBench.

1.4 Estrutura do Texto

O Capítulo 2 trata da Localidade e Redundância de Valores e do comportamento das aplicações que lhe conferem essas características. São apresentadas as técnicas gerais de reuso e previsão de valores, bem como é feita uma análise e comparação dos trabalhos relacionados a essas técnicas. O Capítulo 3 apresenta a arquitetura de processadores ARM, sua principais características arquiteturais e avalia o conjunto de instruções e a execução condicional no que se refere ao reuso. Na sequência, o Capítulo 4 apresenta a metodologia utilizada no desenvolvimento do trabalho, discute a política de reuso adotada e apresenta as ferramentas utilizadas. No Capítulo 5 são apresentados os potenciais de reuso de traços na arquitetura ARM, onde é apresentado o limite de reuso atingido pelos testes realizados, além dos potenciais de reuso utilizando diferentes limitações na construção dos traços para diferentes tamanhos de *buffers*. Por fim, no Capítulo 6 é apresentada a conclusão do presente trabalho e são apontados trabalhos futuros.

2 LOCALIDADE E REDUNDÂNCIA DE VALORES

Uma definição para Localidade de Valores é a probabilidade de recorrência de um valor em uma determinada área de armazenamento, como posições em *cache*, em memória ou em registradores (LIPASTI; WILKERSON; SHEN, 1996). Um outro enfoque define como o potencial existente em um programa de prever os valores gerados durante a execução (GABBAY, 1996). Assim, a previsibilidade pode ser definida como *Temporal*, quando diz respeito à probabilidade de uma execução gerar o mesmo resultado de outra já executada em um tempo próximo; ou pode ser definida como previsibilidade *Espacial*, que corresponde à probabilidade de uma execução gerar uma variação de uma execução anterior.

Programas são constituídos em grande parte por computação redundante e previsível (GABBAY, 1996). Dessa forma, foram desenvolvidas técnicas baseadas na localidade de valores, visando explorar essa característica para atingir melhores níveis de desempenho em processadores.

Analogamente ao uso de memórias *cache*, que exploram a localidade espacial e temporal no acesso a valores para diminuir a latência de acesso à memória principal, pode-se explorar a localidade de valores no nível de execução para evitar a execução de tarefas redundantes.

Neste Capítulo serão apresentadas as técnicas de reuso de valores que exploram a localidade e redundância de valores, bem como a técnica de previsão de valores. Além disso, é feita uma análise dos trabalhos relacionados e, por fim, é apresentada uma discussão sobre as principais técnicas.

2.1 Reuso de Valores

O reuso de valores (LIPASTI; WILKERSON; SHEN, 1996; SODANI; SOHI, 1998) surgiu da observação da execução de programas, onde percebeu-se que muitas das instruções executadas em processadores superescalares se repetem, executando com os mesmos operandos e gerando os mesmos resultados. Da mesma forma que o uso de *cache* reduz o número de acesso a memória principal, o reuso de valores tem

o objetivo de reduzir o número de instruções executadas. A redução de redundância de instruções é uma tarefa tradicionalmente encarada pelo compilador (AHO; ULLMAN; SETHI, 1995), que tenta reduzir trechos de código que não produzam novos resultados. Porém, muitas vezes é difícil identificar essas redundâncias em tempo de compilação, pois os valores a serem operados podem não ser conhecidos.

O reuso de valores é uma técnica não especulativa, isto é, trabalha apenas com valores conhecidos, sem fazer previsões. Conforme a execução da computação acontece, as entradas e saídas são armazenadas em uma tabela indexada, para posterior consulta. Na próxima vez que essa computação for executada, suas entradas serão comparadas com as entradas da tabela. Caso sejam iguais, os valores de saída armazenados na tabela são copiados diretamente para os registradores de saída. Dessa forma, alguns dos estágios do *pipeline* não serão utilizados, causando uma importante economia de recursos.

Além das vantagens relacionadas a redução das instruções executadas, como economia de energia e redução no tempo de execução, é possível destacar também (COSTA, 2001; SODANI, 2000):

- Resultados das computações ficam disponíveis mais cedo, permitindo que instruções dependentes executem antes;
- Dependências de dados são reduzidas, pois instruções dependentes podem ser reusadas em paralelo; e
- As execuções geradas por previsões de desvios incorretas, ao invés de serem descartadas, podem ser reutilizadas por execuções futuras.

Para que o reuso seja possível, é necessário prover uma estrutura de memória capaz de armazenar as instruções já executadas. Além disso, deve-se adicionar um mecanismo que intercepte as instruções, os valores dos operandos e os resultados das execuções no momento em que elas acontecem. Todo este *hardware* adicional necessita trabalhar no mesmo ritmo em que as instruções são executadas, dessa forma, a estrutura de memória deve estar localizada próxima ao processador. O *hardware* para implementar o mecanismo de reuso induz a adição de alguns custos na arquitetura:

- Aumento na complexidade do projeto;
- Consumo energético adicional; e
- Aumento no valor de produção, devido à adição de *hardware*.

As diferentes estratégias que implementam alguma técnica de reuso seguem basicamente os mesmos passos, como definido em (PILLA, 2004):

- Uma unidade de um conjunto de computações é armazenada com seus valores de entrada e saída;
- Em uma próxima execução, é identificada uma oportunidade de reuso;
- Desta oportunidade de reuso, são comparadas as entradas com aquelas anteriormente armazenadas; e
- Caso as entradas sejam iguais, o valores de saída armazenados anteriormente são reutilizados, evitando a re-execução.

Dentre as abordagens que utilizam reuso de valores, a diferença básica entre elas está na unidade do reuso (HUANG; LILJA, 2000), podendo ser de vários níveis, como reuso de instruções, reuso de blocos básicos ou reuso de traços de instruções. Todas essas estratégias buscam, de formas diferentes, minimizar os custos causados pela adição de *hardware* e maximizar a possibilidade de reuso.

2.1.1 Reuso de Instruções

Reuso de Instruções é uma técnica onde a unidade reutilizada é uma única instrução. Assim, quando uma instrução é executada, suas entradas são comparadas com as das execuções anteriores e, caso sejam iguais, essa instrução não precisa ser executada novamente. Dessa forma, basta aproveitar o valor gerado pela execução anterior.

O mecanismo necessário para prover o reuso de instruções envolve um *buffer*, utilizado para armazenar as informações que poderão ser reutilizadas. É necessário também um teste de reuso, que verificará se as entradas das instruções correntes correspondem as entradas das instruções armazenadas no *buffer*. Na Figura 1 é apresentado um modelo básico para reuso de instruções utilizando um *buffer* de instruções indexado pelo PC (*Program Counter*) (SODANI, 2000).

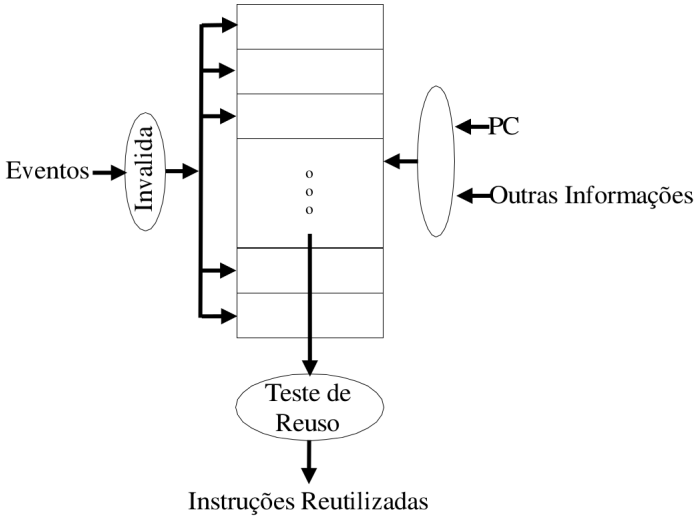

Figura 1: *Buffer* de Reuso Genérico

As informações de cada entrada do *buffer* de reuso se referem a uma instrução, que deve conter seu PC, os registradores de entrada e seus valores, bem como os registradores de saída e os valores gerados pela execução.

O *pipeline* de instruções apresentado na Figura 2 mostra um mecanismo de reuso de instruções acoplado. O teste do reuso é feito em paralelo com a busca da instrução, não adicionando custos aos estágios originais do *pipeline*. Dessa forma a adição do reuso de instruções não prejudica o desempenho da arquitetura, mesmo quando as instruções não forem reutilizadas.

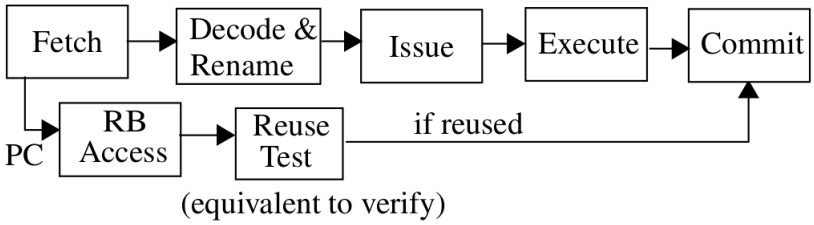

Figura 2: *Pipeline* com reuso de instruções (SODANI; SOHI, 1998)

Neste modelo, a cada instrução executada, suas entradas são comparadas com as presentes no *buffer* de reuso. Caso as entradas sejam iguais, as saídas do *buffer* de reuso referentes a esta instrução são passadas diretamente para o último estágio do *pipeline*. Caso o teste de reuso falhe, a instrução segue seu fluxo normal no *pipeline*.

2.1.2 Reuso de Blocos Básicos

O reuso de blocos básicos (HUANG; LILJA, 1999) é uma abordagem que visa reutilizar conjuntos de instruções ao invés de apenas uma única, suprimindo a execução de um bloco de instruções em apenas um momento. Blocos básicos são sequências

dinâmicas de instruções, construídas durante a execução, onde sua última instrução é um desvio. Para possibilitar o reuso, os blocos básicos são estruturas dinâmicas representadas por um conjunto de dados de entrada e um conjunto de dados de saída. Dados gerados por instruções intermediárias que não compõem o resultado final não precisam ser armazenadas, o que diminui o espaço necessário para o armazenamento se comparado ao reuso de instruções. Os conjuntos de dados de saída são utilizados por outros blocos básicos e, caso um bloco produza um resultado que não seja utilizado por nenhum outro bloco básico, esta saída não precisa ser armazenada. Estes resultados não utilizados são chamados de *saídas mortas* (HUANG; LILJA, 1999). As entradas e saídas dos blocos básicos são armazenados em um *buffer*, onde cada bloco pode aparecer mais de uma vez no *buffer* com entradas e saídas diferentes. Na Figura 3 é representada uma estrutura básica para o armazenamento de uma entrada no *buffer* para reuso de blocos.

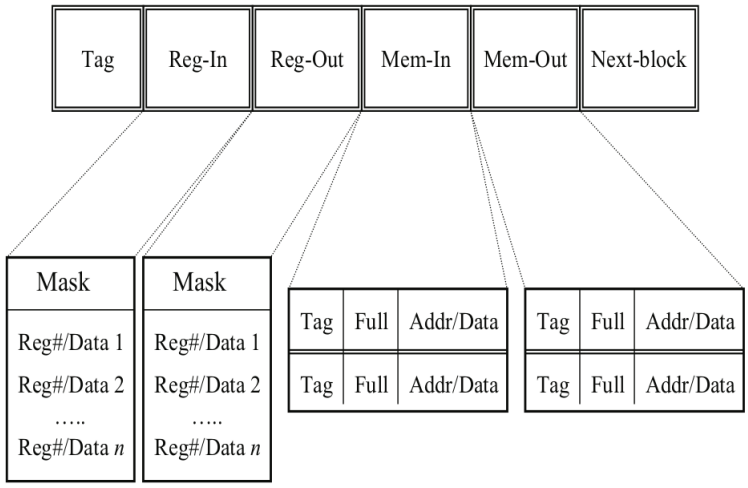

Figura 3: Exemplo de estrutura para Reuso de Blocos (HUANG; LILJA, 1999)

Os campos básicos para um *buffer* de blocos básicos, conforme a Figura 3, são:

- **tag** - Endereço do início do Bloco Básico;
- **Reg-in** - Conjunto de registradores de entrada com seus valores;
- **Reg-out** - Conjunto de registradores de saída com seus valores;
- **Mem-in** - Conjunto de valores em memória que compõem a entrada do bloco;
- **Mem-out** - Conjunto de valores em memória que compõem a saída do bloco;
- **Next-block** - Endereço da próxima instrução após o fim do bloco.

Neste modelo, além da limitação do tamanho do *buffer*, é importante que haja também uma limitação no tamanho dos blocos para evitar o excesso de valores de

entrada e saída a serem armazenados. Quanto mais instruções formarem o traço, maiores serão os conjuntos de valores de entrada e saída.

2.1.3 Reuso de Traços de Instruções

Os traços de instruções (GONZÁLEZ; TUBELLA; MOLINA, 1999; COSTA; FRANÇA; FILHO, 2000; PILLA, 2004; LAURINO et al., 2005) são sequências de instruções executadas. Diferentemente dos blocos básicos, os traços podem conter instruções de desvio, conforme representado na Figura 4.

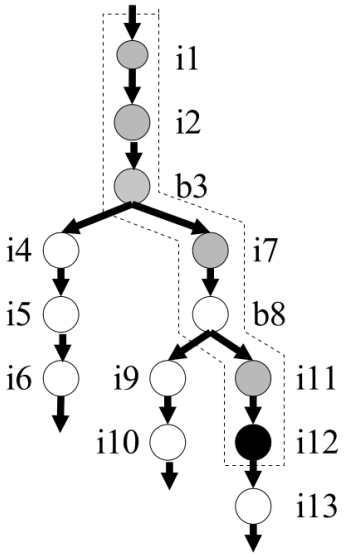

Figura 4: Formação de traços (PILLA, 2004)

No que se refere ao reuso, os traços possuem um contexto de entrada e um contexto de saída. O contexto de entrada compreende os valores a serem operados nas instruções, enquanto o contexto de saída se refere ao conjunto de valores gerados pela execução destas instruções. Para possibilitar o reuso, esses contextos podem ser armazenados em *buffers*, de forma a modelar os traços com potencial de serem reutilizados, conforme Figura 5.

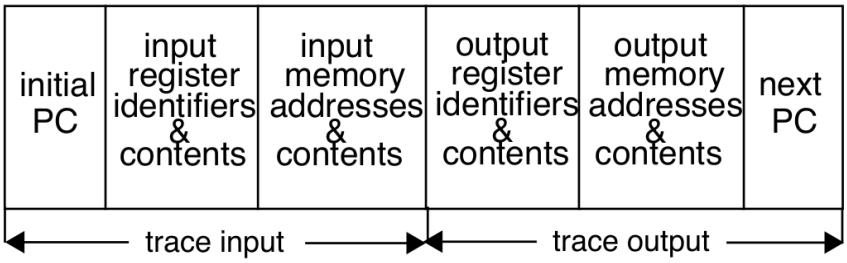

Figura 5: Representação de um traço no *buffer* (GONZÁLEZ; TUBELLA; MOLINA, 1999)

Assim como no reuso de instruções, os traços devem passar por um teste para

determinar se o reuso será ou não efetuado. A medida que as instruções são buscadas, elas são comparadas com as instruções que formam os traços já armazenados no *buffer* de reuso. Os traços são considerados redundantes quando seus contextos de entrada forem idênticos.

A abordagem do reuso de traços possibilita que várias instruções sejam reutilizadas simultaneamente, suprimindo sequências de execuções redundantes. Diferentemente das abordagens de mais baixo nível, que reutilizam apenas uma instrução por vez ou blocos de instruções, o reuso de traços necessita de estruturas de armazenamento e politicas de reuso mais complexas, visto que os traços além de ter tamanho variável, podem conter instruções de desvio. Assim como no reuso de blocos, existe a necessidade de limitar o tamanho dos traços para evitar o excesso de valores dos contextos de entrada e saída a serem armazenados.

Uma das vantagens do reuso de traços é que, mesmo que o reuso de um traço não seja possível por haver diferenças nos valores do contexto de entrada, este traço ainda pode ser aproveitado para evitar a busca e decodificação das instruções que o formam. Neste caso, o *buffer* de reuso passaria a entregar instruções ao processador, o que traz a possibilidade de evitar eventuais previsões de desvios (TSAI; CHEN, 2011).

2.2 Previsão de Valores

A previsão de valores é uma técnica especulativa que visa aumentar o desempenho de processadores antecipando a execução das próximas computações, aproveitando-se da redundância de instruções existente (LIPASTI; WILKERSON; SHEN, 1996). A especulação pode ser utilizada para prever os valores que serão gerados pelas próximas computações, fazendo com que elas não precisem ser executadas. Essa previsão é feita baseando-se nos valores das execuções anteriores, previamente armazenados. Esse mecanismo visa reduzir o tempo total de execução de programas, explorando a antecipação da aquisição do resultado.

Ao se basear nos valores anteriores para especular os valores futuros, pode-se fazer uma *previsão de último valor* ou *previsão de deslocamento* (GABBAY, 1996). Na *previsão de último valor*, analisa-se o valor gerado pela última execução da instrução, previamente armazenado em uma tabela indexada, e utiliza-se este valor como previsão. Neste caso, a profundidade do mecanismo é 1, pois analisa apenas uma instrução anterior, porém é possível armazenar e analisar um número maior de execuções anteriores, aumentando a profundidade do mecanismo. Na *previsão de deslocamento*, analisa-se os valores gerados pelas duas últimas execuções da instrução. Dessa forma calcula-se o valor de deslocamento entre os resultados, o qual é utilizado para prever o próximo valor.

Como outra forma de explorar a previsão de valores, ao invés de prever as saídas das próximas computações, pode-se prever os valores de entrada ainda não conhecidos das instruções que aguardam para serem executadas (PILLA, 2004). Com essa previsão, as instruções ficam prontas para execução mais cedo, pois as esperas causadas pelas dependências de dados são reduzidas.

Em um mecanismo de previsão de valores, as previsões incorretas devem ser observadas. Nesses casos, é necessário que a arquitetura avalie se a previsão teve ou não sucesso. Assim, quando é feita uma previsão incorreta, o sistema precisa recuperar o contexto de execução anterior a especulação. Esse problema é semelhante às previsões de desvio, que também são sujeitas a falhas, onde seus contextos anteriores aos desvios errados devem ser recuperados.

Além da arquitetura necessitar avaliar a corretude das previsões, é necessário que ela proveja um mecanismo de recuperação de contexto para contornar as especulações erradas. Dessa forma, quando uma previsão errada for detectada, todas as instruções executadas após a previsão deverão ser descartadas. Segundo (LAURINO et al., 2005), existe uma grande semelhança entre os sistemas para recuperação de previsão de desvios e os sistemas necessários para a recuperação de previsão de valores. Dessa forma, o mesmo mecanismo presente na arquiteturas atuais para recuperação de desvios incorretos pode ser usado, com pequenas modificações, no sistema de previsão de valores.

2.3 Trabalhos sobre Localidade de Valores

Nesta seção serão abordados trabalhos que propõem modelos e técnicas para explorar a localidade de valores no nível de arquitetura. Os trabalhos apresentam, de forma geral, arquiteturas para prover mecanismos de reuso ou previsão de valores. Ao final da seção é feita uma análise e comparação das técnicas.

2.3.1 RST: *Reuse through Speculation on Traces*

Baseado no DTM (COSTA; FRANÇA; FILHO, 2000), a estratégia da arquitetura RST (PILLA, 2004) consiste em prover para a arquitetura MIPS um mecanismo de reuso de valores combinado com técnicas de previsão de valores. Nesta abordagem, Pilla (2004) busca que o acréscimo de *hardware* não seja tão grande se comparado com as duas outras técnicas isoladas. Utilizar apenas reuso tem como vantagem a redução da quantidade total de instruções executadas, devido a redução da redundância de execução. Porém, ao utilizar técnicas de reuso no nível de traços, um limitante de desempenho está na necessidade de esperar até que todas as entradas das instruções que formam os traços estejam disponíveis, para assim poder fazer o teste de reuso. A combinação do reuso de valores com previsão de valores, denomi-

nado reuso especulativo, visa permitir que traços de instruções possam ser reusados antes que todos os operandos estejam disponíveis. Utilizando a previsão de valores, é possível prever alguns dos operandos das instruções que formam os traços. Dessa forma é possível antecipar o teste de reuso, pois os traços estarão prontos mais cedo.

Na arquitetura RST o reuso pode ser feito no nível de instruções e também no nível de traços. Porém, o reuso especulativo só pode ser feito no nível de traços. O reuso de traços pode ser feito regularmente, sem previsão; ou especulativamente, com previsão dos operandos que formam o traço. A construção dos traços é feita dinamicamente durante a execução do programa. Os traços podem ser formados por instruções pertencentes a um conjunto previamente definido, denominado domínio de reuso. Na arquitetura RST, o domínio de reuso não inclui instruções de acesso à memória nem instruções que operem sobre valores em ponto flutuante.

Um traço é construído à medida que instruções pertencentes ao domínio do reuso são encontradas, parando ao encontrar uma instrução fora deste domínio. Este traço é então armazenado, juntamente com os valores dos registradores de entrada e pelos valores gerados pelas execuções. Quando uma próxima execução alcançar este traço, é feito o teste de reuso. Caso as entradas do traço corrente sejam iguais aos valores do traço armazenado, as saídas armazenadas são copiadas para os registradores de saída. Se algumas entradas não estiverem disponíveis para comparação, é feita uma especulação.

Na arquitetura RST, ao adicionar previsão de valores, não são necessárias novas tabelas para armazenar os valores especulados, se comparado a abordagem que utiliza apenas reuso de valores. Assim, a arquitetura RST utiliza pouco *hardware* adicional para implementar o reuso especulativo em comparação ao *hardware* necessário para prover o reuso não especulativo.

2.3.1.1 Tabelas de Reuso

Na arquitetura RST, conforme instruções são executadas e demonstram potencial para reuso, elas são colocadas na tabela *Memo_Table G*. Essa tabela serve para a memorização global e é destinada para reuso de instruções. A Figura 6 mostra um entrada da *Memo_Table G* que corresponde a uma instrução armazenada.

bits	*30*	*32*	*32*	*32*	*1*	*1*	*1*
	pc	sv_1	sv_2	res/targ	jmp	brc	btk

Figura 6: RST - Entrada em *Memo_Table G* (PILLA, 2004)

Cada entrada da tabela possui os seguintes campos:

- **pc** - Endereço da instrução;

- **sv**$_1$ e **sv**$_2$ - Operandos de entrada;
- **res/targ** - Resultado ou endereço alvo;
- **jmp** - Quando setado, indica que a instrução é um desvio incondicional;
- **brc** - Quando setado, indica que a instrução é um desvio condicional;
- **btk** - Especifica se o desvio será tomado ou não.

Ao identificar traços, estes são armazenados na *Memo_Table T*, conforme ilustrado na Figura 7. Essa tabela é destinada apenas para traços de instruções.

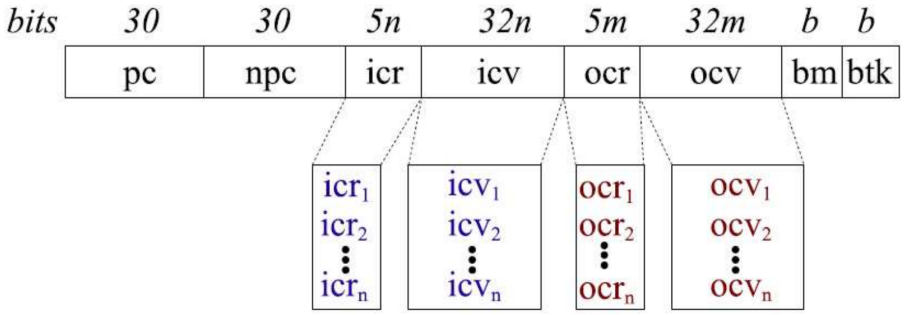

Figura 7: RST - Entrada em *Memo_Table T* (PILLA, 2004)

Cada traço armazenado em *Memo_Table T* possui os seguintes campos:

- **pc** - Endereço da primeira instrução do traço;
- **npc** - Endereço da primeira instrução após o fim do traço;
- **icr** - Nome dos registradores do contexto de entrada;
- **icv** - Valores dos registradores do contexto de entrada;
- **ocr** - Nome dos registradores do contexto de saída;
- **ocv** - Valores dos registradores do contexto de saída;
- **bm** - Mapa de bits usado para indicar desvios dentro de um traço;
- **btk** - Especifica a direção dos desvios dentro de um traço.

2.3.1.2 Formação de Traços

Em (PILLA, 2004), foram comparadas duas estratégias de formação de traços, uma tradicional e outra mais agressiva:

- ***reused-only*** Nessa abordagem de formação de traços, apenas instruções que que já foram executadas anteriormente podem formar os traços. Dessa forma, a tabela *Memo_Table G* é consultada como referência para permitir ou não a entrada das instruções na formação do traço.

- ***reusable*** Essa abordagem, mais agressiva, permite que instruções que não pertencem a *Memo_Table G*, ou seja, que ainda não foram executadas façam parte do traço.

A análise feita em (PILLA, 2004), mostrou que a abordagem *reused-only* obtém melhores resultados devido ao uso de instruções que já apresentaram redundância.

2.3.1.3 Resultados

A arquitetura RST, utilizando a abordagem *reused-only*, apresentou *speedups* de 1,29 sobre a arquitetura base sem reuso e previsão, e 1,09 de *speedup* sobre a arquitetura com apenas reuso de traços. Além disso, o trabalho observou que é mais eficiente utilizar a estrutura de memória adicional para implementar a arquitetura RST do que simplesmente aumentar a *cache* L1.

2.3.2 Arquitetura RSTm

A arquitetura RST originalmente não inclui instruções de acesso à memória nos traços. Dessa forma, no momento em que os traços são formados, caso uma instrução de acesso à memória seja encontrada, ocorrerá a finalização do traço. O estudo feito em (LAURINO, 2007) mostra que essas instruções são responsáveis por 45% da finalização de todos traços criados. O trabalho de (LAURINO, 2007) propôs a inclusão de instruções de acesso à memória como forma de aumentar o tamanho médio dos traços e, assim, minimizar problemas de dependências de dados.

Para possibilitar a inclusão de instruções de acesso a memória nos traços, a arquitetura RSTm inclui a tabela *Memo_Table L*, apresentada na Figura 8. A tabela *Memo_Table L* possui os seguintes campos:

- **pc** - endereço da instrução de *load*;

- **sv1** - valor do operando fonte;

- **type** - mapa de bits que indica o tipo de acesso (*half*, *single*, *double*);

- **maddr** - endereço de acesso à memória;

- **valid** - *flag* que indica se o valor de leitura/escrita é valido;

- **res** - valor de leitura/escrita.

pc	sv1	type	maddr	valid	res
30b	32b	2b	32b	1b	32b

Figura 8: *Memo_Table L* (LAURINO, 2007)

A inclusão de instruções de acesso à memória aos traços traz um problema de concorrência, pois estas instruções em um traço podem ter suas posições de memória alteradas por instruções de fora deste traço. Para solucionar esse problema, a arquitetura RSTm marca como inválido o valor da operação de leitura na tabela *Memo_Table L* quando esta for alterada por uma instrução de fora do traço. No momento do teste do reuso, caso a entrada da *Memo_Table L* esteja marcada como inválida, o reuso do traço não é realizado.

A adição de instruções de acesso à memória aumentou em 4,77% o desempenho da arquitetura RST quando utilizando a estratégia *reused-only* para formação de traços, e em 2,17% com uso da estratégia *reusable*.

2.3.3 Contrail Processor Architecture

No processador Contrail (KOUSHIRO; SATO; ARITA, 2003) existem duas linhas de execução. Uma delas é chamada de *speculation stream*. Nesta linha de execução acontece a maior parte das computações, utilizando as unidade de processamento mais rápidas. Na *speculation stream* a arquitetura utiliza especulação de traços, assim, uma parcela significativa de código que é especulado, não precisa ser executado.

Ao se fazer especulações, algumas delas podem não estar corretas. Para esses casos, uma outra linha de execução é utilizada para fazer a verificação da previsão. As unidades de processamento utilizadas para fazer essa verificação podem ser mais lentas que as principais, visando redução de consumo. O ganho dessa estratégia ocorre de duas formas: com a redução do número de instruções executadas na linha principal, e pela possibilidade de adiantar instruções que seriam dependentes. Isso é possível pois, ao fazer especulações, instruções que dependem dos dados especulados ficam disponíveis mais cedo para serem executadas. Na Figura 9 é apresentada a organização da execução no processador Contrail.

Caso as especulações estejam erradas, a outra linha de execução é responsável por fazer a verificação. Assim, a linha principal de execução deve retomar o contexto da execução no momento da especulação errada, perdendo assim, todas instruções executadas após a previsão. Considerando as taxas de previsões incorretas e mantendo o foco na economia de energia, a arquitetura proposta aponta um potencial de redução de 40% no consumo de energia mantendo o desempenho original.

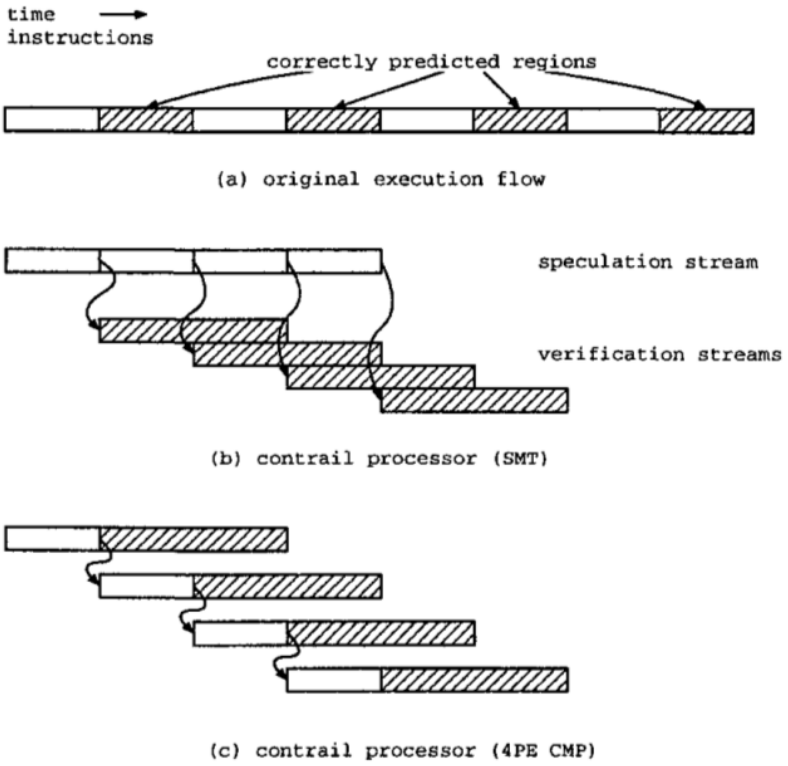

Figura 9: Execução em Processador Contrail (KOUSHIRO; SATO; ARITA, 2003)

2.3.4 TR Cache

Partindo do princípio que um aumento de eficiência na entrega de instruções ao processador resulta em um aumento de desempenho no mesmo, em (TSAI; CHEN, 2011) é proposta uma estrutura de memória chamada *Trace Reuse Cache* (*TR Cache*), destinada a processadores ARM. A ideia básica consiste em inserir um *buffer* para traços no mesmo nível da memória *cache* de instruções, como uma segunda fonte de instruções para o processador. Por estar no mesmo nível da *cache* de instruções, o custo para entrega de instruções ao processador permanece o mesmo.

O *buffer* de reuso de traços é alimentado dinamicamente pelos traços formados pelas instruções que já passaram pelo *pipeline*. Diferentemente de outras estratégias de reuso, na *TR Cache* os traços contém apenas as instruções, e não os resultados de suas execuções. O mecanismo proposto identifica os traços observando o *pipeline* de instruções, sem interferir no seu andamento. Conforme os traços são identificados, estes são armazenados no *buffer*. Caso um novo traço identificado já esteja presente no *buffer*, a *cache* principal é alterada para o *buffer* de reuso de traços, que passa a ser responsável pela entrega de instruções ao processador. Essa técnica visa aumentar o número de instruções entregues ao processador, visto que os traços armazenados no *buffer* já tiveram suas instruções de desvio resolvidos e as dependências de dados reduzidas.

Esta técnica proporciona um número menor de paradas no *pipeline*, aumentando o IPC e causando uma consequente redução no consumo energético. De acordo com (TSAI; CHEN, 2011), um *buffer* para reuso de traços com 2048 entradas é capaz de prover uma redução de 75% no consumo de energia para uma *cache* de instruções de 16kB, enquanto o IPC atinge um ganho de 21%.

2.3.5 Arquitetura DCE

A arquitetura DCE (*Dynamic Conditional Execution*) (SANTOS, 2003) visa reduzir o custo de previsões incorretas em processadores superescalares explorando a execução multifluxo. A arquitetura busca e executa os vários caminhos gerados pelas instruções de desvio, obedecendo algumas restrições relacionadas a tamanho e a complexidade. Com isso, a arquitetura busca reduzir o número de previsões e, consequentemente, evitar as penalidades geradas pelas previsões incorretas.

Ao encontrar uma instrução de desvio condicional, a arquitetura busca ambos os caminhos possíveis, até que a instrução que gerou o desvio seja solucionada. Neste momento, é mantido apenas o fluxo correto e descartados os demais. Ao executar diversos fluxos, as instruções precisam ser replicadas para manter a semântica original do programa. Essas réplicas são geradas e mantidas até que o desvio que gerou o fluxo seja resolvido. As múltiplas réplicas podem facilmente saturar o *pipeline*, causando perdas de desempenho. Dessa forma, em (SANTOS, 2004) foi desenvolvido um mecanismo de Reuso de Valores para a arquitetura DCE visando reduzir o volume de instruções redundantes.

No mecanismo proposto por Santos (2004), primeiramente foi aplicado o reuso de instruções, inserindo um estágio no *pipeline* após a renomeação de registradores. Nesse estágio é feito o teste de reuso e a atribuição do valor destino. Ao passarem por esse estágio, as instruções são armazenadas em um *buffer* para posterior uso. Nas próximas execuções, o *buffer* é consultado e, caso os operandos de origem correspondam aos armazenados no *buffer*, o valor de saída é copiado para o registrador de destino. A instrução reusada é então encaminhada para o *buffer* de reordenamento, onde aguardará pela resolução da instrução que gerou o desvio. Essa abordagem atingiu cerca de 15% de instruções reusadas. Porém, mesmo sendo reusadas, essas instruções ocupam espaço no *pipeline*. Mesmo liberando unidades funcionais do *pipeline*, as instruções ainda permanecem no *buffer* de reordenamento para esperar a graduação dos resultados em ordem.

Quando aplicado o reuso de traços, este problema foi reduzido, pois instruções que fazem parte do traço e não produzem resultados que serão utilizados por instruções posteriores podem ser retiradas do *pipeline*, o que reduziu o numero de instruções no *buffer* de reordenamento. Com essa abordagem o trabalho atingiu, em alguns casos, ganhos de 80% se comparado ao DCE original.

2.3.6 Comparação

Nesta seção foram apresentados trabalhos que exploram a localidade de valores buscando aumento no poder de processamento ou redução do consumo energético.

Na arquitetura original do RST (PILLA, 2004), é feita a combinação das técnicas de reuso e previsão de valores, em uma abordagem de Reuso Especulativo. Nessa abordagem, sem incluir instruções de acesso à memória, foi atingido um *speedup* de 1,09 sobre a mesma arquitetura sem especulação. Já em (LAURINO et al., 2005) foi proposto o RSTm, que incluía instruções de acesso à memória na arquitetura RST. Essa inclusão obteve um ganho de 4,77% em relação a abordagem original. A estratégia do *Contrail Processor Architecture* (KOUSHIRO; SATO; ARITA, 2003) utiliza apenas Previsão de Valores como estratégia para explorar a localidade de valores, focando no mecanismo de recuperação das especulações. Esse mecanismo visa reduzir o consumo energético, utilizando unidades funcionais de baixo consumo para fazer as verificações das especulações. Com essa estratégia, o trabalho indica a possibilidade de economizar até 40% de energia. Na arquitetura TR Cache (TSAI; CHEN, 2011) é proposto um mecanismo adicional de entrega de instruções ao processador, utilizando para isso um *buffer* de traços. Ao entregar instruções de traços já conhecidos, a arquitetura visa aumentar o IPC, visto que estes traços já tiveram seus desvios resolvidos e as dependências de dados reduzidas. Na arquitetura DCE (SANTOS, 2003), (SANTOS, 2004) é proposto um modelo de execução multifluxo com aplicação de técnicas de reuso de valores para redução de replicações. Ao utilizar reuso de traços, a arquitetura conseguiu liberar recursos do *pipeline*, atingindo um desempenho 80% superior a estratégia de execução multifluxo sem reuso.

Observa-se nos trabalhos apresentados que o reuso de traços é a opção mais interessante, visto necessitarem estruturas de armazenamento e políticas de reuso relativamente simples. Além disso, os trabalhos utilizam o reuso e a previsão como forma de reduzir as dependências de dados e solucionar previsões de desvios, atribuindo a isto uma parcela do desempenho atingido pelas técnicas. O emprego de técnicas e previsão de valores, sozinhas ou combinadas com o mecanismos de reuso, atingem resultados positivos, porém necessitam de mecanismos adicionais para verificação de erros. A inclusão de instruções de acesso a memória em mecanismos de reuso possibilitam o aumento do comprimento dos traços, trazendo ganhos de desempenho. Porém, para que isso seja possível são necessárias estruturas de armazenamento especiais, além de políticas para verificação da coerência dos valores armazenados nas tabelas de reuso com os valores em memória.

Para todas as técnicas, sempre é necessário *hardware* adicional, bem como estruturas de memória junto a memória *cache* L1. As características dos *buffers* para reuso são condicionadas, além de outros fatores, ao formato e ao tipo das instruções a serem reusadas. Diferentes tipos de instruções, que operem com inteiros, ponto flu-

tuante ou operações em memória, demandam estruturas de armazenamento distintas. Dessa forma, os *buffers* terão formatos e tamanhos diferentes para cada arquitetura que forem projetados.

3 ARQUITETURA ARM

A arquitetura ARM (*Advanced Risc Machine*) (ARM, 2015a) é uma categoria de processadores superescalares do tipo RISC (*Reduced Instruction Set Computer*), que presam pela simplicidade e baixo consumo de energia. A simplicidade dos processadores ARM e suas características que lhes conferem baixo consumo energético, tornaram esses processadores referência para computação embarcada e dispositivos portáteis. São facilmente encontrados em celulares, *smartphones*, *tablets*, calculadoras, computadores de bordo, impressoras, equipamentos de rede, etc.

A cada dia, novos dispositivos portáteis são desenvolvidos, bem como equipamentos com pequenas unidades de processamento para atividades específicas. O aumento das funções que cada equipamento pode realizar torna os projetos cada vez maiores e mais complexos. Nesses casos, é necessário adotar soluções práticas e modulares. Em um *smartphone*, por exemplo, para processar os dados dos acelerômetros, da câmera, e gerenciar uma conexão de rede, é menos custoso utilizar um processador de uso geral do que desenvolver hardware especializado para gerenciar cada uma dessas tarefas.

As características e funcionalidades providas pelos processadores ARM se mostraram muito adequadas para o desenvolvimento de projetos de dispositivos móveis e sistemas embarcados em geral, como pode ser comprovado pela sua grande disseminação nesses seguimentos.

3.1 História

A arquitetura ARM teve início em 1983 (LEVY, 2005), quando a empresa Acorn Computers necessitava de um microprocessador de 16 bits para seu próximo computador. Porém, não existiam processadores que suprissem suas necessidades, pois os processadores existentes eram mais lentos que os módulos de memória disponíveis. Além disso, os processadores possuíam instruções complexas, onde algumas delas necessitavam de centenas de ciclos para executar, causando latências elevadas.

Dessa forma, a Acorn começou a desenvolver seu próprio processador. Porém, no

início dos anos 80, com a grande complexidade das arquiteturas de processadores, poderia levar vários anos para que uma empresa com experiência desenvolvesse um novo processador.

Assim, a Acorn baseou-se no projeto Berkeley RISC 1 (SéQUIN; PATTERSON, 1982), que mostrou que era possível construir um processador simples com desempenho comparável com os mais avançados processadores CISC (*Complex Instruction Set Computer*) do momento. Seguindo a abordagem RISC, em 1985 foi apresentado o processador chamado Acorn RISC Machine (ARM), o qual se tornou o primeiro processador RISC comercial. Este primeiro processador é denominado ARM *version* 1, sucedido pelo ARM *version* 2 em 1987, que continha suporte a coprocessador. Com adição de uma memória *cache* integrada no processador, surgiu o ARM *version* 3.

Em 1992, a arquitetura ARM entrou no mercado de sistemas embarcados, tendo um processador desenvolvido para o PDA Apple Newton. O processador contava com endereçamento de 32 bits, suporte a MMU e instruções de multiplicação e acumulação de 64 bits. Neste empreendimento conjunto entre a Acorn e a Apple, a empresa criada foi chamada de ARM, Advanced RISC Machines.

3.2 Características

ARM é uma arquitetura do tipo RISC e possui as características tipicas desta arquitetura (ARM ARCHITECTURE REFERENCE MANUAL, 2005)

- Grande e uniforme banco de registradores;
- Arquitetura do tipo *load/store*, onde as operações ocorrem apenas em registradores, não diretamente na memória;
- Modos de endereçamento simples, com todos endereços de *loads* e *stores* determinados por conteúdo de registradores e parâmetros de instruções;
- Instruções de tamanho fixo e uniforme;

Dentre as características que promoveram a grande popularidade dos processadores ARM, destacam-se (RYZHYK, 2006):

- Sendo feita uma comparação entre um *core* de utilização geral e outro do tipo ARM, este último é mais simples. Isto implica que um processador ARM pode ser construído com um número relativamente pequeno de transistores, permitindo a inserção de uma maior quantidade de componentes voltados para aplicações específicas;
- Tanto o *set* de instruções como o *pipeline* de uma arquitetura ARM são construídos de modo a consumir a menor quantidade de energia possível.

- Esta categoria de arquiteturas é altamente modular. Somente o *pipeline* de inteiros é um componente obrigatório dentro de uma processador ARM. Assim, é possível construir processadores baseados em ARM com bastante flexibilidade.

A alta modularidade dos processadores ARM permitiu que fossem desenvolvidos processadores destinados para diversos fins. Um processador indicado para aplicações multimídia, por exemplo, pode conter módulos como acelerador gráfico, controlador de LCD (*Liquid Crystal Display*) e decodificador de vídeo. Já para um processador voltado para automação industrial pode ser mais interessante que contenha um maior número de temporizadores, PWM (*Pulse Width Modulation*), sensor de temperatura e temporizador tipo *Watchdog*.

Dentre os módulos normalmente encontrados, pode-se destacar (FREESCALE, 2014; CHON; VALENZUELA, 2013):

- **Geral:** *TouchScreen*, Temporizador *Watchdog*, DMA (*Direct Memory Access*), PWM, MMU (*Memory Management Unit*), FPU (*Float Point Unit*);
- **Conectividade:** I2C, SPI, SSI, UART, USB, USB OTG, ETHERNET;
- **Multimídia:** Acelerador 2D, Controlador de LCD, Decodificador de Vídeo, Interface para câmera, Pré e Pós-Processador de Imagens;

A arquitetura ARM possui um conjunto de instruções com tamanho fixo em 32 bits (FURBER, 2000), à exceção da extensão *Thumb* que permite executar instruções de 16 bits. Este tipo de instrução é capaz de melhorar o desempenho do processador em aplicações específicas, por criar um código mais compacto, é ideal para aplicações com restrições de banda de memória (ARM7TDMI - TECHNICAL REFERENCE MANUAL, 2004).

A arquitetura ARM possui um conjunto de 37 registradores de 32 bits. Destes, 6 são utilizados como registradores de *status* e o restante é usado para propósito geral (SEAL, 2000). Os registradores são organizados em dois conjuntos que se sobrepõem, onde o modo de processamento atual controla qual conjunto de registradores está disponível. Assim, somente é possível ter acesso a 16 registradores, independente do modo de execução.

A arquitetura ARM suporta sete modos de execução, as quais possuem diferentes permissões para acessar o banco de registradores, como pode ser visualizado na Tabela 1. São eles:

- *User*: Execução normal de programas;
- *Fast Interrupt* (FIQ): Possui suporte à transferência rápida de dados;
- *Normal Interrupt* (IRQ): Empregado em interrupções gerais;

- *Software Interrupt* (SVC): Usado para chamadas de serviços do sistema operacional;
- *Abort*: Serve como resposta para possíveis falhas em memória;
- *System*: Executa operações privilegiadas do sistema operacional;
- *Undefined* (UND): Pode ser utilizado como modo de emulação de coprocessador.

Os modos privilegiados contém registradores especiais que são habilitados somente nestes modos, os quais possuem funções específicas. É o caso dos registradores destacados em cinza na Tabela 1.

Tabela 1: Registradores e os modos de execução na arquitetura ARM (KNAGGS; WELSH, 2004)

User	FIQ	IRQ	SVC	System	Abort	UND
r0	r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7	r7
r8	r8 fiq	r8	r8	r8	r8	r8
r9	r9 fiq	r9	r9	r9	r9	r9
r10	r10 fiq	r10	r10	r10	r10	r10
r11	r11 fiq	r11	r11	r11	r11	r11
r12	r12 fiq	r12	r12	r12	r12	r12
r13	r13 fiq	r13 irq	r13 svc	r13	r13 abt	r13 und
r14	r14 fiq	r14 irq	r14 svc	r14	r14 abt	r14 und
r15	r15	r15	r15	r15	r15	r15
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
SPSR	SPSR fiq	SPSR irq	SPSR svc	SPSR	SPSR abt	SPSR und

3.3 Instruções

O *set* de instruções da arquitetura ARM, apresentado na Tabela 2, é composto basicamente por instruções de adição, subtração, operações lógicas, desvio e acesso a memória. Este conjunto totaliza 36 instruções básicas (KNAGGS; WELSH, 2004).

3.3.1 Instruções Condicionais

Uma característica particular relacionada à arquitetura ARM diz respeito à execução condicional de instruções. No conjunto de instruções ARM, estas podem

Tabela 2: *Set* de instruções da arquitetura ARM

Operador	Significado
ADC	Add with carry
ADD	Add
AND	Logical AND
BAL	Unconditional branch
B(cc)	Branch on condition
BIC	Bit clear
BLAL	Unconditional branch and link
BL(cc)	Conditional branch and link
CMP	Compare
EOR	Exclusive OR
LDM	Load multiple
LDR	Load register (Word)
LDRB	Load register (Byte)
MLA	Multiply accumulate
MOV	Move
MSR	Load SPSR or CPSR
MSR	Store to SPSR or CPSR
MUL	Multiply
MVN	Logical NOT
ORR	Logical OR
RSB	Reverse subtract
RSC	Reverse subtract with carry
SBC	Subtract with carry
SMLAL	Mult accum signed long
SMULL	Multiply signed long
STM	Store multiple
STR	Store register (Word)
STRB	Store register (Byte)
SUB	Subtract
SWI	Software interrupt
SWP	Swap word value
SWPB	Swap byte value
TEQ	Test equivalence
TST	Test
UMLAL	Mult accum unsigned long
UMLLL	Multiply unsigned long

ser executadas condicionalmente ou não, dependendo do estado de um conjunto de bits de condição no momento da execução. Dentro das instruções 4 bits definem 16 condições diferentes para uma determinada instrução ser executada. Esta instrução será executada dependendo do estado destes bits condicionais no registrador de *status* CPSR (*Current Program Status Register*).

Na Tabela 3 são mostradas as 16 situações condicionais que podem ocor-

rer. Quando uma instrução for condicional, o sufixo da condição aparece junto ao mnemônico da instrução. Na prática, todas as instruções do conjunto ARM são condicionais, pois uma das condições significa executar sempre (*AL*). Caso nenhuma situação condicional seja especificada, a condição (*AL*) será assumida. A atualização do estado dos bits de condição no registador CPSR é feito conforme o resultado da execução de instruções com o sufixo *S*.

Tabela 3: Sufixos condicionais da arquitetura ARM

Sufixo	Descrição
CS	Carry set
CC	Carry clear
EQ	Equal
VS	Overflow set
VC	Overflow clear
GT	Greater than
LT	Less than
GE	Greater than or equal
LE	Less than or equal
PL	Plus (Positive)
MI	Minus (Negative)
HI	Higher than
LO	Lower than
HS	Higher or same
LS	Lower or same
AL	Always

4 METODOLOGIA

Como forma de analisar o comportamento da execução de programas na arquitetura ARM, foi executado o conjunto de *benchmarks* MiBench sobre o simulador SimPanalyzer, como feito em (NACHTIGALL et al., 2012). Para simular o reuso de traços, foram feitas modificações no simulador para interceptar as instruções executadas, bem como os valores dos registradores a cada execução. Essas modificações não interferem na execução normal dos programas. Com acesso às instruções e aos valores dos registradores, foi possível simular o reuso de traços.

4.1 Domínio de Reuso

Primeiramente foi definido o domínio de reuso, que corresponde ao grupo de instruções que poderão compor os traços. No trabalho de (LAURINO et al., 2005) utilizando o processador MIPS, foi observado que a inclusão de instruções de acesso a memória nos traços demandam estruturas de armazenamento mais complexas e mecanismos extras para o controle da coerência dos dados em memória. Em (COPPIETERS et al., 2015), é observado que, além de necessitarem estruturas maiores para armazenamento, instruções que operam com ponto flutuante são menos propensas a apresentar redundância se comparadas com instruções que operam com inteiros.

De acordo com estas análises, os testes realizados utilizam como domínio de reuso apenas instruções que operem com inteiros e que não façam acesso a memória, visando abranger o maior grupo de instruções, com maiores chances de apresentar redundância e que demandem estruturas de armazenamento menores e mais simples. Isso significa que apenas instruções deste grupo poderão compor os traços e, consequentemente, apenas estas poderão ser reutilizadas.

4.2 Política de Reuso de Traços

Após definir o domínio de reuso, foi possível determinar a estrutura básica para representar os traços, a qual é apresentada na Figura 10. Cada entrada da estrutura

de armazenamento possui os seguintes campos:

Figura 10: Estrutura para representação de um Traço

- **pc-inicial:** Endereço da primeira instrução que forma o traço - 32 bits
- **pc-final:** Endereço da primeira instrução após o fim do traço - 32 bits
- **regs-map-in:** Registradores de entrada (mapa de bits) - 16 bits
- **regs-map-out:** Registradores de saída (mapa de bits) - 16 bits
- **regs-val-in:** Valores dos registradores de entrada - estrutura para até 16 registradores de 32 bits
- **regs-val-out:** Valores dos registradores de saída - estrutura para até 16 registradores de 32 bits
- **cpsr-map-in:** *Flags* condicionais de entrada (mapa de bits) - 4 bits
- **cpsr-map-out:** *Flags* condicionais de saída (mapa de bits) - 4 bits
- **cpsr-val-in:** Valores das *flags* condicionais de entrada - 4 bits
- **cpsr-val-out:** Valores das *flags* condicionais de saída - 4 bits

A estrutura representa um traço a ser armazenado no *buffer* de reuso. Este traço deve ser composto por uma sequência de instruções pertencentes ao domínio de reuso, finalizado por uma instrução fora deste domínio. A formação dos traços se dá em tempo de execução, seguindo os passos descritos:

1. Armazenar o endereço da primeira instrução em *pc-inicial*
2. Para cada instrução pertencente ao domínio do reuso:
    - Identificar os registradores de entrada
    - Destes registradores, marcar em *regs-map-in* todos que não estejam em *regs-map-out*
    - Armazenar os valores destes registradores em *regs-val-in*
    - Identificar os registradores de saída

- Marcar estes registradores em *regs-map-out*
- Armazenar os valores dos registradores de saída, substituindo eventuais valores antigos, em *regs-val-out*
- Identificar as *flags* condicionais
- Caso estas *flags* não estejam marcadas em *cpsr-map-out*, marcar em *cpsr-map-in*
- Armazenar os valores destas *flags* em *cpsr-val-in*
- Para instruções que atualizem o registrador CPSR:
  - Identificar *flags* atualizadas pela instrução
  - Marcar estas *flags* em *cpsr-map-out*
  - Armazenar o valor destas *flags*, substituindo eventuais valores antigos, em *cpsr-val-out*

3. Ao encontrar uma instrução fora do domínio de reuso, armazenar o endereço desta instrução em *pc-final*.

A estrutura apresentada na Figura 10 e o mecanismo de formação de traços foram implementados dentro do simulador Sim-Panalyzer onde são interceptadas as instruções, os valores dos operandos e o resultado das execuções. Partindo desta estrutura, foi possível analisar o comportamento das execuções e realizar os testes de reuso apresentados no Capítulo 5.

4.3 Ferramentas

Nesta seção são apresentadas as ferramentas utilizadas para a realização dos testes da presente dissertação. Os testes foram feitos tendo como base o simulador de arquitetura ARM Sim-Panalyzer e utilizando como carga de trabalho o conjunto de *benchmarks* MiBench.

4.3.1 Simulador Sim-Panalyzer

O simulador Sim-Panalyzer (MUDGE; AUSTIN; GRUNWALD, 2003) é o software empregado para simular uma arquitetura ARM em um computador de uso geral. Esta ferramenta foi baseada em um outro simulador, chamado SimpleScalar (AUSTIN, 1997), voltado para simular arquiteturas como x86, PISA, Alpha, além de ARM. O principal objetivo da ferramenta Sim-Panalyzer é criar um estimador adiantado de potência, permitindo avaliar ganhos e perdas na relação entre desempenho e dissipação de energia. O simulador implementa o conjunto de instruções ARM7

(ARM7TDMI - TECHNICAL REFERENCE MANUAL, 2004) e o pipeline do processador StrongArm (WITEK; MONTANARO, 1996). Sim-Panalyzer possui código fonte disponível, necessário para para aplicar as modificações referentes ao reuso.

4.3.2 Benchmarks MiBench

MiBench (GUTHAUS et al., 2002) é um conjunto de *benchmarks* gratuito e comercialmente representativo. Deste conjunto foram utilizadas 23 aplicações, as quais são divididas em 6 categorias (GUTHAUS et al., 2001):

- Controle industrial e automotivo:
  - **basicmath:** Executa cálculos matemáticos básicos que não possuem *hardware* dedicado em processadores RISC;
  - **bitcount:** Testa o desempenho de processadores quanto a manipulação de bits;
  - **quicksort:** Ordena *arrays* de *strings* utilizando o algoritmo *quicksort*;
  - **susan (corners):** Algoritmo para reconhecimento de imagens;
- Redes:
  - **dijkstra:** Utiliza o algoritmo de dijkstra para calcular o menor caminho entre cada par de nodos em uma matriz de adjacências;
  - **patricia:** Testa uma estrutura de árvore Patricia;
- Segurança:
  - **blowfish encode/decode:** Cifra simétrica de blocos com tamanho variável de chave;
  - **rijndael encode/decode:** Cifra de blocos com tamanhos possíveis de 128, 192 e 256 bits;
  - **sha:** *Secure Hash Algorithm*, algoritmo de *hash* utilizado em criptografia.
- Dispositivos de consumidor:
  - **jpeg encode/decode:** Algorítimo para compressão e descompressão de imagens;
  - **mad:** Decodificador de áudio MPEG de alta qualidade;
  - **tiff2bw:** Conversor de imagens TIFF coloridas para preto e branco;
  - **tiff2rgba:** Converte uma imagem TIFF colorida em outra TIFF no padrão RGB;

  - **tiffdither:** Utilizado para reduzir o tamanho e a resolução de imagens monocromáticas TIFF;
  - **tiffmedian:** Converte uma imagem TIFF para outra com a palheta de cores reduzida.

- Automação de serviços:
  - **ghostscript:** Interpretador de linguagem *postscript* sem interface gráfica;
  - **ispell:** Verificador ortográfico;
  - **rsynth:** Sintetizador de texto em voz;
  - **stringsearch:** Busca de palavras em frases;

- Telecomunicações:
  - **CRC:** *Cyclic Redundancy Check*, utilizado para detectar erros em transmissões de dados;
  - **FFT/IFFT:** *Fast Fourier Transform*, realiza a transformada rápida de Fourier e sua inversa, aplicada em processamento digital de sinais;
  - **ADPCM encode/decode:** *Adaptive Differential Pulse Code Modulation*, utilizado para representar digitalmente sinais analógicos;
  - **GSM encode/decode:** *Global Standard for Mobile communications*, padrão para codificação e decodificação de voz.

MiBench possui código fonte disponível e executáveis previamente compilados para a arquitetura ARM, o que agilizou os testes realizados.

5 POTENCIAL DE REUSO DE TRAÇOS NA ARQUITETURA ARM

Neste Capítulo são apresentados os resultados obtidos através da simulação de reuso de traços em processadores ARM. É apresentado o limite máximo do reuso, considerando um mecanismo sem limitações de armazenamento. Além disso, são discutidos os potenciais de reuso utilizando diferentes tamanhos de *buffers* com limitações na estrutura de armazenamento dos traços. Com essas análises, são apresentadas configurações para a implementação dos *buffers* de reuso, de acordo com seus tamanhos.

5.1 Limites de Reuso

Utilizando a estratégia para formação de traços e a estrutura para armazenamento, ambos apresentados na Seção 4.2, foi realizada a execução dos *benchmarks* MiBench no simulador Sim-Panalyzer modificado para simular o reuso de traços. A partir deste conjunto de execuções, foi possível quantificar o percentual das instruções que fazem parte do domínio do reuso, conforme apresentado na Figura 11.

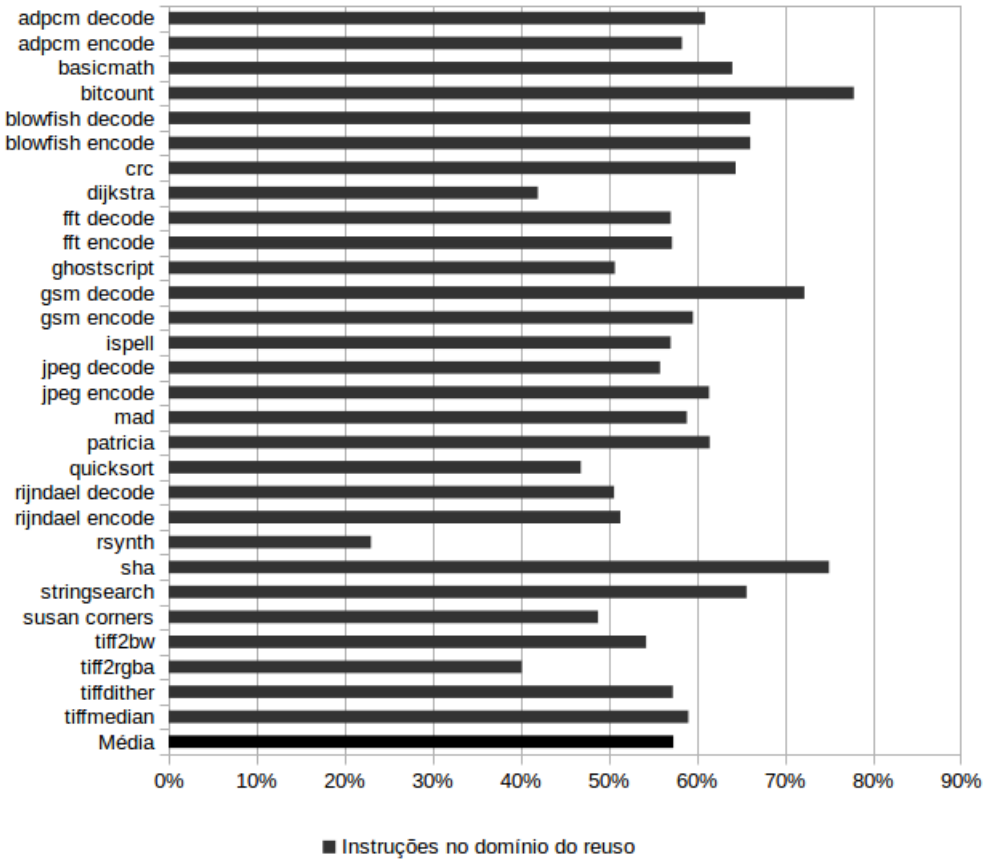

Figura 11: Percentual de instruções no Domínio de Reuso.

Nessa análise, observa-se que em média 57,3% do total de instruções operam com inteiros e não fazem acesso a memória, as quais são as instruções utilizadas nos testes apresentados a seguir. Isso significa que um dos limites para o mecanismo de reuso é a parcela de instruções que podem compor os traços. Neste caso, o reuso de traços será aplicado, em média, a 57,3% da computação executada pelos *benchmarks*.

Em um primeiro momento, foi realizada a execução do reuso de traços utilizando um *buffer* sem limitações, com o objetivo de quantificar o reuso máximo possível. O potencial da técnica de reuso de traços, quantificado pela parcela de computação redundante suprimida pelo emprego do reuso é apresentado na Figura 12. O gráfico mostra a parcela de instruções reutilizadas pelo mecanismo de reuso de traços em relação ao total de instruções executas pelo simulador e também em relação as instruções do domínio do reuso.

Como pode ser visto, o reuso médio atingido foi de 34,1% ao considerar todas instruções, o que representa a parcela de computação dos *benchmarks* que teriam a execução evitada pela aplicação do reuso de traços. Considerando apenas as instruções que compõem os traços, foi identificada a redundância média de 60,5%.

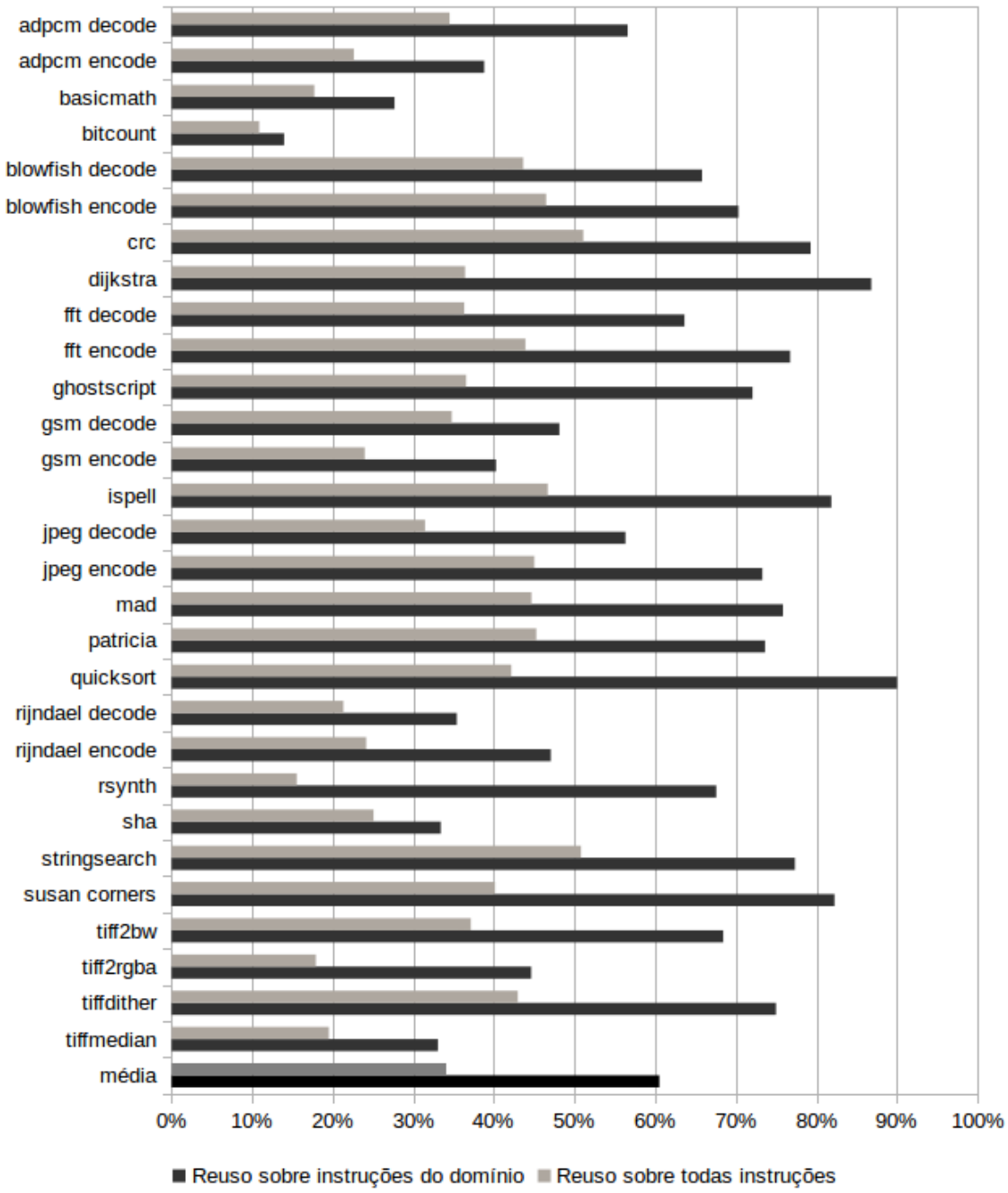

Figura 12: Percentual de instruções reutilizadas sem limitação do tamanho de *buffer*.

Os testes realizados para definir o reuso máximo consideram um *buffer* ilimitado em número de traços que podem ser armazenados e na quantidade de registradores de entrada e de saída que podem compor os traços. Assim, o mecanismo ilimitado garante espaço para armazenar até 16 registradores para o contexto de entrada e 16 no contexto de saída de cada traço. Para analisar se os traços formados realmente utilizariam a estrutura proposta, foi quantificada a parcela de traços que seria abrangida para todas as limitações de registradores de entrada e de saída. Para fins de melhor visualização, na Figura 13 são apresentadas 16 destas combinações. Foi possível concluir que para representar todos os traços construídos durante a execução do conjunto de *benchmarks* não são necessários mais que 11 registradores de entrada e 11 de saída.

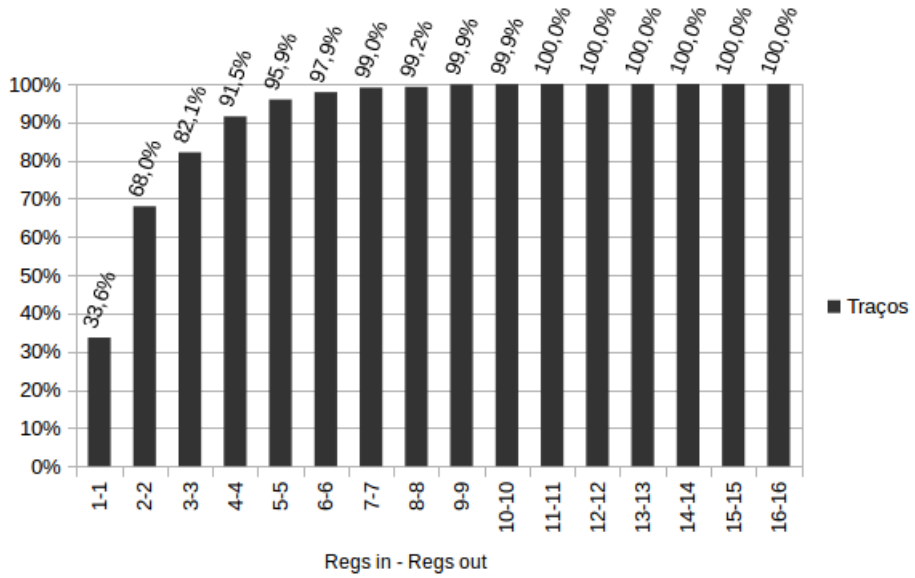

Figura 13: Percentual de traços armazenados com limitações no número de registradores de entrada e saída.

Com essa análise, é possível definir o tamanho do *buffer* necessário para atingir o reuso máximo. O cálculo do tamanho do *buffer* máximo necessário, baseado na quantidade de traços formados pelo *benchmark* mais longo (CRC) e pelas limitações de 11 registradores para o contexto de entrada e 11 para o contexto de saída, é apresentado na Tabela 4. Dessa forma, observamos que para atingir o reuso máximo de 34,1% apresentado na Figura 12 seria necessário um *buffer* de 400 MiB.

Tabela 4: Buffer máximo

Máximo de traços construídos	Bits por traço	Tamanho do Buffer
4.113.122	816	400,1 MiB

5.2 Reuso de traços com limitações de *buffer*

Como mecanismos de reuso necessitam de estruturas de memória adicionais junto ao processador, de forma semelhante às *caches* L1 (COPPIETERS et al., 2015), se faz necessário que os *buffers* de reuso durante a simulação tenham tamanhos limitados. Considerando que as *caches* L1 mais utilizadas tenham tamanho aplicável para implementar o *buffer* do mecanismo de reuso, os testes realizados consideraram *buffers* no intervalo de 1 KiB à 1024 KiB. Nesta faixa é possível encontrar os tamanhos de *cache* L1 mais aplicadas em processadores ARM (ARM, 2013, 2014a,b,c, 2015b), além de valores extremos para comparação.

Conforme ilustrado anteriormente na Figura 13, reduções maiores nos registradores, como 7 registradores para o contexto de entradas e 7 para o de saída, abrangem quase a totalidade dos traços. Uma redução como esta causaria poucas perdas pelos traços que não seriam representados. Porém, a estrutura para armazenar cada traço seria menor, reduzindo o tamanho total do *buffer* necessário. Partindo desta análise, o trabalho buscou determinar formas mais eficientes de armazenar dados para reuso. Nos testes executados, é aplicada a combinação de duas limitações para o uso do *buffer* de reuso de traços:

- Limite máximo de registradores de entrada e de saída de cada traço;
- Limite máximo de traços dentro do *buffer*.

Considerando um *buffer* de tamanho total fixo, ao se alterar a quantidade máxima de registradores de entrada e de saída, o tamanho de cada entrada no *buffer* para armazenar um traço irá variar, o que implica em uma mudança no total de traços que caberão no *buffer*. Essa alteração na quantidade de registradores causa o seguinte efeito:

- Com menos registradores na estrutura de armazenamento de traços:
  - Alguns traços não poderão ser armazenados devido a limitação nos registradores de entrada e saída, diminuindo o reuso;
  - O *buffer* poderá comportar mais traços, aumentando o reuso.
- Com mais registradores na estrutura de armazenamento de traços:
  - Mais traços poderão ser representados na estrutura de armazenamento, aumentando o reuso;
  - O *buffer* poderá comportar menos traços, diminuindo reuso.

Partindo desta análise, foram feitos testes para determinar a melhor combinação entre a limitação nos registradores que compõem os contextos de entrada e saída e o total de traços no *buffer*. Com a análise dos testes, as melhores combinações foram determinadas pelo maior reuso atingido, representado pelo percentual de instruções que seriam reutilizadas. Na Tabela 5, são apresentadas as configurações dos *buffers* que obtiveram os melhores resultados de reuso médio com a execução dos *benchmarks* MiBench.

Tabela 5: Configurações dos *buffers* de reuso.

Buffer	Traços	Regs-in	Regs-out	Tamanho de uma entrada	Parcela de traços representável no buffer
1 KiB	22	4	4	46 bytes	91,50%
2 KiB	44				
4 KiB	89				
8 KiB	178				
16 KiB	356				
32 KiB	655	5	4	50 bytes	94,53%
64 KiB	1213	5	5	54 bytes	95,93%
128 KiB	1598	7	10	82 bytes	99,34%
256 KiB	3196				
512 KiB	6393				
1024 KiB	12787				

De acordo com as estruturas apresentadas na Tabela 5, observa-se que as melhores configurações para *buffers* menores são alcançadas aplicando maiores restrições nos registradores de entrada e de saída, como é o caso dos *buffers* de 1 KiB a 16 KiB.

Nesses casos, a maior limitação no reuso é atribuída ao baixo número de traços que cabem no *buffer*. Já para os *buffers* maiores, a limitação causada pela quantidade de traços no *buffer* é menos significativa, então é possível aceitar traços que possuam mais registradores de entrada e saída.

Esse comportamento pode ser visto na Figura 14, onde é apresentada a concentração de traços por KiB de memória. Nesta Figura é apresentada a quantidade de traços que cabem em cada KiB de memória, para cada configuração de *buffer*. Para os *buffers* maiores, de 128 KiB à 1024 KiB, observa-se uma concentração de 12,5 traços para cada KiB de memória. Já para os *buffers* menores, a concentração fica em 22 traços por KiB. Dessa forma, foi observado que em *buffers* menores a alta concentração de traços visa compensar a reduzida capacidade total de armazenamento, principal fator que limita o reuso. Este comportamento não ocorre para *buffers* a partir de 128 KiB, que possuem grande capacidade total de armazenamento. Nestes casos, pode-se reduzir a concentração de traços sem grandes perdas para o reuso, em troca de permitir mais registradores nos contextos de cada traço.

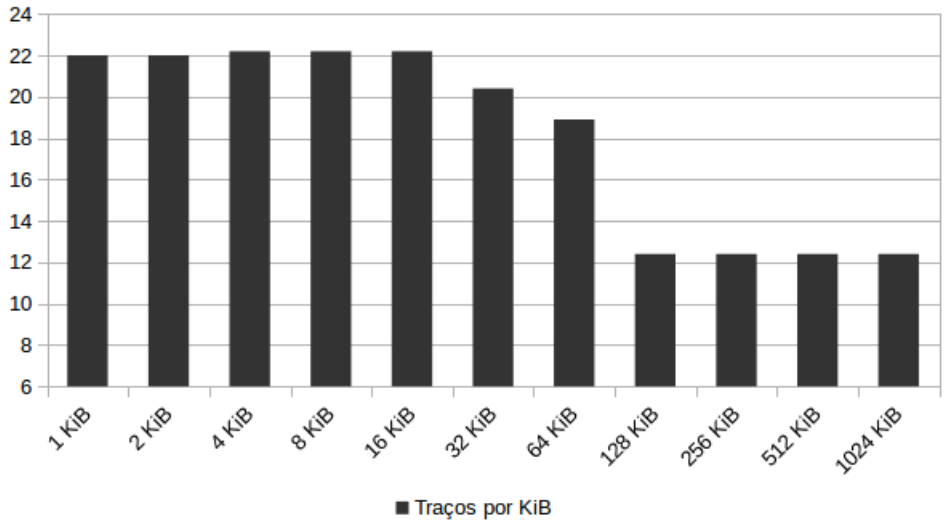

Figura 14: Concentração de traços por KiB

Na Figura 15 são apresentados os percentuais de reuso atingido pelas configurações descritas na Tabela 5. O gráfico apresenta a parcela de computação redundante suprimida pelo mecanismo de reuso em relação ao total de instruções executadas pelo simulador. Os resultados obtidos confirmam a aplicabilidade de reuso de traços em processadores ARM utilizando *buffers* de tamanhos similares aos utilizados em *caches* L1. Analisando a linearidade do reuso com diferentes tamanhos de *buffers*, mesmo estes tendo configurações diferentes, observou-se que a estratégia de limitação dos registradores de entrada e saída torna o uso dos *buffers* mais eficientes. Além disso, o *buffer* de 32 KiB atingiu 18,4% de reuso final, o que corresponde a mais da metade do potencial de reuso máximo, que necessitaria de um *buffer* de 400 MiB para atingir 34,1% de reuso final. Com isso é possível concluir que mesmo com *buffers* pequenos, é possível explorar grandes parcelas da redundância de execução.

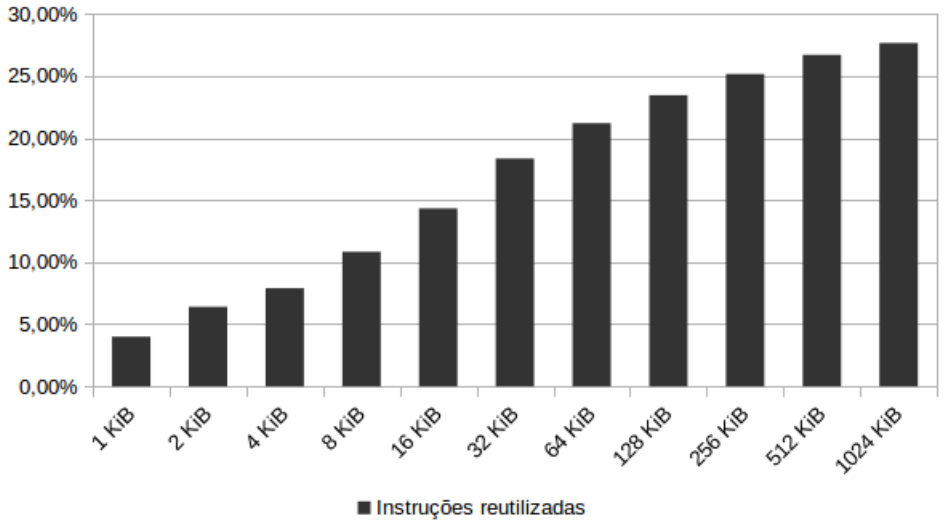

Figura 15: Percentuais de reuso de instruções com limitações de tamanho de *buffer*.

5.3 Viabilidade de implementação dos *buffers*

Considerando que o *buffer* de reuso deverá ficar localizado junto ao processador no mesmo nível da *cache* L1 e que esta estrutura deve ser acessada em tempo de execução sem adicionar custos à execução normal, é importante que o *buffer* seja acessado em frequência compatível com a operação do processador. Utilizando a ferramenta CACTI (THOZIYOOR et al., 2008), foi feita a avaliação do tempo de acesso para o *buffer* de 32 KiB com a configuração apresentada na Tabela 5 considerando tecnologia de 32 nm. Com essa análise, foi observado que é possível acessar o *buffer* na frequência de até 3,12 GHz, o que é superior a frequência de operação dos processadores ARM atuais (ARM, 2015b).

5.4 Comprimento dos traços

Ao se utilizar o reuso no nível de traços, a principal vantagem é a possibilidade de evitar a execução de um conjunto de instruções em um único momento. Na Figura 16, são apresentados os comprimentos médios dos traços por *benchmark*. O comprimento corresponde a quantas instruções compõem cada traço, independente do número de registradores do contexto de entrada e saída. A média do comprimento dos traços dos *benchmarks* ficou em 3,4 instruções.

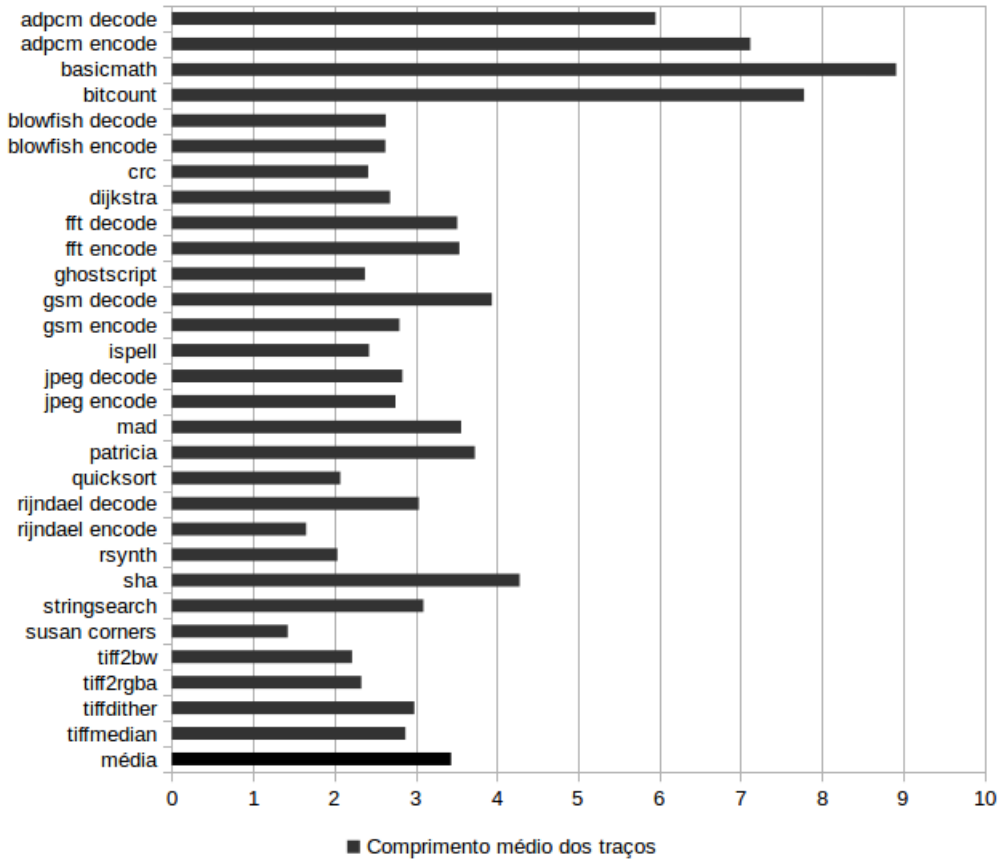

Figura 16: Comprimento médio dos traços

Embora traços de mesmo comprimento possam ter quantidades diferentes de registradores de entrada e saída, é possível observar que traços mais longos tendem a ter mais registradores nos contextos de entrada e saída. Na Figura 17, onde são apresentadas as médias dos registradores de entrada e saída dos traços gerados para cada *benchmark*, observa-se que aplicações como *ADPCM encode/decode* e *Sha*, que possuem traços mais longos (Figura 16), também possuem uma maior média de registradores de entrada e saída no contexto dos traços.

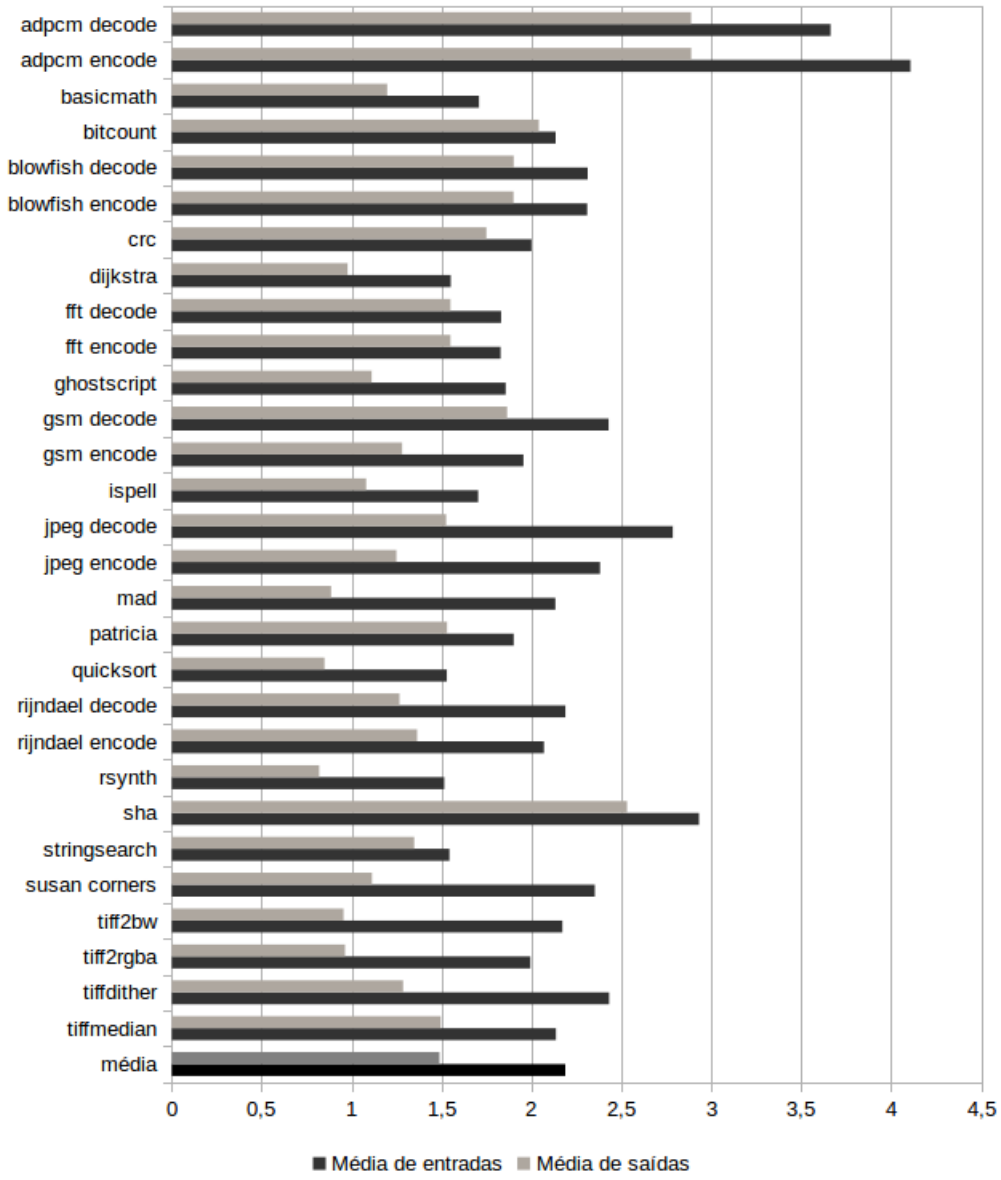

Figura 17: Média de registradores de entrada e saída

De acordo com essa análise, na Figura 18 é feita uma comparação entre o reuso final atingido com as diferentes configurações de *buffer* e os traços reutilizados dentre aqueles que puderam ser representados no *buffer*. No gráfico observa-se um desvio no comportamento do reuso para os *buffers* de 64 KiB e 128 KiB. Para estes dois *buffers*, o reuso de traços permanece praticamente inalterado, com 52,2% e 52,8% respectivamente. Porém, o reuso de instruções teve crescimento entre os *buffers*, de 21,2% para 23,5%, seguindo o comportamento de crescimento dos outros *buffers*. Essa diferença se dá devido ao comprimento dos traços reusados em cada *buffer*. Como apresentado anteriormente na Tabela 5, o *buffer* de 64 KiB permite até 5 registradores para o contexto de entrada e 5 para o contexto de saída. Já a configuração do *buffer* de 128 KiB pode armazenar traços com até 7 registradores no contexto de

entrada e 10 no contexto de saída. Isso permite que com o *buffer* de 128 KiB seja possível armazenar e reutilizar traços mais longos. Dessa forma, mesmo tendo taxas de reuso de traços muito próximas, na configuração do *buffer* de 128 KiB os traços possuem mais instruções do que os traços na configuração do *buffer* de 64 KiB. Assim, reutilizando traços mais longos, atinge-se um melhor reuso final com o *buffer* de 128 KiB.

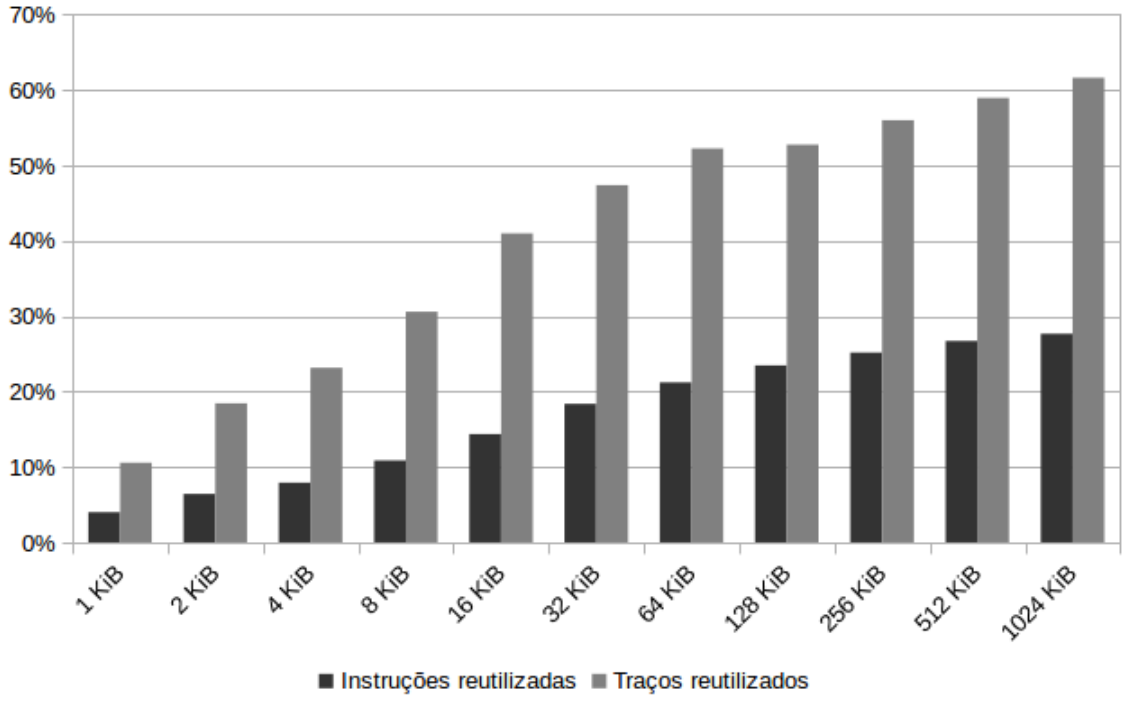

Figura 18: Reuso de traços em comparação com o reuso final

6 CONCLUSÕES

No presente trabalho foi apresentada uma estratégia de reuso de traços para a arquitetura de processadores ARM utilizando *buffers* de diferentes tamanhos. Para isso, foram consideradas as características das instruções ARM para a definição da estrutura do *buffer*; foi analisado o comportamento da execução do conjunto de *benchmarks* MiBench sobre o simulador de arquitetura ARM Sim-Panalyzer para identificar o potencial de reuso máximo; foram definidas limitações na construção dos traços, visando aumentar a eficiência do uso do *buffer* de acordo com o seu tamanho; e, por fim, foram apresentados o potenciais de reuso para *buffers* de 1 KiB à 1024 KiB, utilizando diferentes configurações para a construção dos traços.

Os testes realizados demonstraram que não são necessários mais que 11 registradores para o contexto de entrada e 11 para o contexto de saída para a estrutura de armazenamento de traços. Isso significa que a quantidade máxima possível de traços pode ser construída com essa limitação, equivalendo a uma estrutura sem limitação. Além disso, observou-se que mesmo limitações maiores nos contextos de entrada e saída dos traços podem atingir resultados muito próximos do máximo possível. Esse comportamento demonstra que a limitação no contexto de entrada e de saída durante a construção dos traços é uma estratégia útil para diminuir o tamanho dos traços sem causar grandes alterações na probabilidade de reuso.

Ao se implementar diferentes limitações no contexto de entrada e de saída dos traços, concluiu-se que *buffers* de diferentes tamanhos atingem seus melhores resultados com limitações diferentes. *Buffers* maiores atingem melhores resultados com menos limitações na formação dos contextos, enquanto *buffers* menores são mais eficientes ao se aplicar mais limitações. Esse comportamento deve-se ao tamanho final dos traços e a quantos traços cada *buffer* pode comportar. Dessa forma, conclui-se que a quantidade de traços em *buffers* menores é um fator determinante para o reuso, mesmo que o conjunto de traços com possibilidade de reuso seja menor.

Ao se restringir o uso do *buffer* de acordo com limitações nos registradores de entrada e saída, existe um impacto no comprimento dos traços. Como uma das vantagens em se reutilizar traços é a possibilidade de suprimir sequências de instruções

em um só momento, reutilizar traços mais longos tende a obter melhores resultados, como pôde ser visto comparando os *buffers* de 64 KiB e 128 KiB. Embora o reuso de traços destes *buffers* seja equivalente, a configuração do *buffer* de 128 KiB obteve melhores resultados por reutilizar traços mais longos.

Nos testes com o simulador Sim-Panalyzer utilizando como carga de trabalho o conjunto de *benchmarks* MiBench, identificou-se que 57,3% das instruções executadas não operam com ponto flutuante nem fazem acesso à memória. Aplicando a técnica de reuso de traços apenas neste grupo de instruções, observou-se a redundância média de 60,5%, o que representa o potencial de reuso máximo de 34,1% ao considerar todas instruções executadas. Para atingir o reuso máximo, foi estimado que seria necessário um *buffer* de 400 MiB. Aplicando o reuso de traços utilizando *buffers* limitados, foi possível atingir 18,4% de reuso final médio utilizando um *buffer* de 32 KiB. Esse resultado demonstrou a possibilidade de atingir mais da metade do potencial de reuso máximo, adicionando um *buffer* de tamanho similar ao utilizado em *caches* L1 de processadores modernos.

Por fim, os resultados de reuso obtidos confirmam o potencial da técnica de reuso de traços em processadores ARM, além de demonstrarem que é possível utilizar *buffers* de tamanhos realista, ao se pensar em implementações em *hardware*. A linearidade dos resultados de reuso final para os *buffers* do intervalo de 1 KiB a 1024 KiB, mesmo aplicando diferentes estruturas para armazenamento de traços, demonstram a importância de considerar o tamanho total do *buffer* no momento de definir a estratégia de formação e armazenamento de traços.

O trabalho desta dissertação teve seu desenvolvimento parcial apresentado nos seguintes artigos:

- MOURA, R. C. de; OLIVEIRA TORRES, G. de; PILLA, M. L. Reuso de Valores na Arquitetura ARM. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, 13., 2013, Porto Alegre, RS. Anais. . . SBC, 2013.

- MOURA, R. C. de; OLIVEIRA TORRES, G. de; PILLA, M. L. Estimando o Potencial de Reuso de Instruções na Arquitetura ARM. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, 14., 2014, Porto Alegre, RS. Anais. . . SBC, 2014. p.105–106.

- OLIVEIRA TORRES, G. de; MOURA, R. C. de; PILLA, M. L. Estimativa de desempenho utilizando reuso de instruções em arquiteturas ARM. In: WSCAD-WIC - WORKSHOP DE INICIAÇÃO CIENTÍFICA, 2013, Porto Alegre, RS. Anais. . . SBC, 2013. p.218–221.

6.1 Trabalhos Futuros

Por ser um trabalho inicial sobre a exploração do reuso de valores em processadores ARM, diversos fatores já tratados em outras arquiteturas não puderam ser avaliados, ficando para trabalhos decorrentes deste. Como sugestões de trabalhos futuros podemos destacar:

Avaliar a disponibilidade dos valores dos operandos das instruções. Ao se fazer o teste de reuso, os valores dos registradores podem ainda não estar disponíveis para se fazer a comparação. Neste caso, o reuso seria atrasado. Em (PILLA, 2004), foi proposto o uso de especulação de valores para prever as entradas ainda não disponíveis, o que trouxe resultados positivos nos testes realizados sobre a arquitetura MIPS. Essa estratégia pode ser avaliada em processadores ARM.

Quanto a formação dos traços, nos testes apresentados, qualquer instrução do domínio de reuso poderia compor os traços. Porém, trabalhos relacionados avaliam a possibilidade de utilizar apenas instruções que já tenham apresentado redundância (PILLA, 2004; LAURINO, 2007). A comparação destas duas estratégias (*reusable* / *reused-only*) no reuso de traços em processadores ARM pode ser avaliada.

Considerando que uma grande parcela dos traços identificados possuem apenas uma instrução, é possível avaliar a possibilidade de inserir uma estrutura para reuso de instruções combinada com o reuso de traços. Dessa forma, seria possível diminuir o tamanho total da memória adicional e manter o mesmo nível de reuso.

Pode-se testar diferentes políticas de substituição de traços no *buffer*, como utilizar um campo TTL (*time-to-live*) para cada traço, aplicar a política LRU *Least Recently Used* ou utilizar uma estrutura FIFO (*first in, first out*). Para determinar qual seria a melhor opção, é necessário avaliar principalmente duas características da redundância:

- Quantas vezes em média cada traço é reutilizado;
- Qual a proximidade entre a formação do traço e o seu reuso.

Ao se acrescentar uma estrutura de memória junto a *cache* L1, naturalmente o custo do processador será elevado. Dessa forma, é importante comparar o ganho de desempenho da adição dessa memória para o mecanismo de reuso em relação ao ganho de desempenho ao utilizar a mesma memória apenas para aumentar a *cache* L1. De acordo com o desempenho das duas abordagens, existe a possibilidade de utilizar parte da *cache* L1 para implementar o *buffer* de reuso, sem adição de memória extra. Com isso, é possível ao invés de focar no aumento do poder de processamento, explorar a redução da computação para reduzir o consumo enérgico, mantendo o poder de processamento original.

No trabalho apresentado foram consideradas apenas instruções que operem com inteiros e não façam acesso a memória. Porém, no trabalho de Laurino (2007), a

inclusão de instruções de memória no mecanismo de reuso para o processador MIPS, apesar de aumentar a complexidade do projeto, apresentou resultados positivos em relação a arquitetura base. O impacto da inclusão de instruções de acesso a memória no reuso de traços da arquitetura ARM pode ser avaliado.

REFERÊNCIAS

AHO, A. V.; ULLMAN, J. D.; SETHI, R. **Compiladores** : Princípios, Técnicas e Ferramentas. [S.l.]: LTC, 1995. 344p.

ARM. **ARM Cortex-A15 MPCore Processor - Technical Reference Manual**. Disponível em: <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438i/DDI0438I_cortex_a15_r4p0_trm.pdf>. Acesso em: outubro de 2015.

ARM. **ARM Cortex-A17 MPCore Processor - Technical Reference Manual**. Disponível em: <http//infocenter.arm.com/help/topic/com.arm.doc.ddi0535b/DDI0535B_cortex_a17_r1p0_trm.pdf>. Acesso em: outubro de 2015.

ARM. **ARM Cortex-A53 MPCore Processor - Technical Reference Manual**. Disponível em: <http//infocenter.arm.com/help/topic/com.arm.doc.ddi0500f/DDI0500F_cortex_a53_r0p4_trm.pdf>. Acesso em: outubro de 2015.

ARM. **ARM Cortex-A57 MPCore Processor - Technical Reference Manual**. Disponível em: <http//infocenter.arm.com/help/topic/com.arm.doc.ddi0488g/DDI0488G_cortex_a57_mpcore_trm.pdf>. Acesso em: outubro de 2015.

ARM. **ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile**. Disponível em: <http//infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.h/index.html>. Acesso em: outubro de 2015.

ARM. **ARM Cortex-A72 Processor**. Disponível em: <http//www.arm.com/products/processors/cortex-a/cortex-a72-processor.php>. Acesso em: outubro de 2015.

ARM Architecture Reference Manual. [S.l.: s.n.], 2005. Disponível em: <http://silver.arm.com/download/download.tm?pv=1073121>. Acesso em: outubro de 2015.

ARM7TDMI - Technical Reference Manual. Disponível em: <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf/>. Acesso em: outubro de 2015.

AUSTIN, T. **SimpleScalar LLC**. [S.l.: s.n.], 1997. Disponível em: <http://www.simplescalar.com/>. Acesso em: julho de 2015.

CHON, S.; VALENZUELA, A. **Getting Started on TI ARM embedded processor development – The Basics**. [S.l.]: Texas Instruments, 2013. Disponível em: <http://www.ti.com/lit/wp/spry242/spry242.pdf>. Acesso em: outubro de 2015.

COPPIETERS, A. M.; OLIVEIRA, S. de; PILLA, M. L.; FRANÇA, F. M. G.; COSTA, A. T. da. **Decanting the Contribution of Instruction Types and Loop Structures in the Reuse of Traces**. [S.l.]: COPPE - Universidade Federal do Rio de Janeiro, 2015.

COSTA, A. T. D. **Explorando Dinamicamente o Reuso de Traces em Nível de Arquitetura de Processador**. 2001. 181p. Phd thesis — Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ.

COSTA, A. T. D.; FRANÇA, F. M. G.; FILHO, E. M. C. The Dynamic Trace Memoization Reuse Technique. In: IN 9TH PACT, P. 92-99, 2000, IEEE CS, 2000. **Anais...** [S.l.: s.n.], 2000. p.92–99.

FREESCALE. **Embedded Solutions Based on ARM Technology**. [S.l.]: Freescale Semiconductor, 2014.

FURBER, S. **ARM System-on-Chip Architecture**. 2nd.ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

GABBAY, F. **Speculative Execution based on Value Prediction**. [S.l.]: EE Department TR 1080, Technion - Israel Institue of Technology, 1996.

GONZÁLEZ, A.; TUBELLA, J.; MOLINA, C. **The Performance Potential of Data Value Reuse**. [S.l.: s.n.], 1998.

GONZÁLEZ, A.; TUBELLA, J.; MOLINA, C. Trace-Level Reuse. In: ICPP, 1999. **Anais...** [S.l.: s.n.], 1999. p.30–.

GUTHAUS, M. R.; RINGENBERG, J. S.; ERNST, D.; AUSTIN, T. M.; MUDGE, T.; BROWN, R. B. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: WORKLOAD CHARACTERIZATION, 2001. WWC-4. 2001 IEEE INTERNATIONAL WORKSHOP, 2001, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2001. p.3–14. (WWC '01).

GUTHAUS, M.; RINGENBERG, J.; AUSTIN, T.; MUDGE, T.; BROWN, R. **MiBench Version 1.0**. Disponível em: <http://www.eecs.umich.edu/mibench/>. Acesso em: julho de 2015.

HUANG, J.; LILJA, D. Exploiting basic block value locality with block reuse. In: HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 1999. PROCEEDINGS. FIFTH INTERNATIONAL SYMPOSIUM ON, 1999. **Anais...** [S.l.: s.n.], 1999. p.106–114.

HUANG, J.; LILJA, D. J. Exploring Sub-Block Value Reuse for Superscalar Processors. **Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques**, Los Alamitos, CA, USA, v.0, p.100, 2000.

KNAGGS, P.; WELSH, S. **ARM**: Assembly Language Programming. [S.l.]: School of Design, Engineering & Computing, Bournemouth University, 2004.

KOUSHIRO, T.; SATO, T.; ARITA, I. A trace-level value predictor for Contrail processors. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.31, n.3, p.42–47, June 2003.

LAURINO, L. S. **Reuso especulativo de traços com instruções de acesso à memória**. 2007. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul, Porto Alegre, RS.

LAURINO, L. S.; PILLA, M. L.; SANTOS, T. S. G. dos; NAVAUX, P. O. A. Reuso de Traços com Loads em Arquiteturas Superescalares. **WSCAD**, [S.l.], 2005.

LEVY, M. **The history of the ARM architecture**: From inception to IPO. [S.l.: s.n.], 2005.

LIPASTI, M. H.; WILKERSON, C. B.; SHEN, J. P. Value locality and load value prediction. **SIGPLAN Not.**, New York, NY, USA, v.31, n.9, p.138–147, Sept. 1996.

MUDGE, T.; AUSTIN, T.; GRUNWALD, D. **Sim-Panalyzer 2.0 Reference Manual**. [S.l.: s.n.], 2003.

NACHTIGALL, M. G.; ARAUJO, A. S.; FAVARETTO, R. M.; PILLA, M. L. Perfis de consumo e desempenho de Benchmarks no Sim-Panalyzer. In: ERAD - ESCOLA REGIONAL DE ALTO DESEMPENHO, 2012. **Anais...** [S.l.: s.n.], 2012. p.197–200.

PILLA, M. L. **RST**: Reuse through Speculation on Traces. 2004. 157p. Phd thesis — Universidade Federal do Rio Grande do Sul, Porto Alegre, RS.

RYZHYK, L. **The ARM Architecture**. [S.l.]: Chicago University, 2006.

SANTOS, R. R. dos. **DCE**: The Dynamic Conditional Execution in a Multipath Control Independent Architecture. 2003. 128p. Phd thesis — Universidade Federal do Rio Grande do Sul, Porto Alegre, RS.

SANTOS, T. G. S. dos. **Reusing Values in a Dynamic Conditional Execution Architecture**. 2004. 115p. Phd thesis — Universidade Federal do Rio Grande do Sul, Porto Alegre, RS.

SEAL, D. **Arm Architecture Reference Manual**. [S.l.]: Addison-Wesley Longman Publishing Co., 2000.

SODANI, A. **Dynamic Instruction Reuse**. [S.l.]: University of Wisconsin–Madison, 2000.

SODANI, A.; SOHI, G. S. Understanding the Differences Between Value Prediction and Instruction Reuse. In: MICRO'98, 1998. **Anais...** [S.l.: s.n.], 1998. p.205–215.

SéQUIN, C. H.; PATTERSON, D. A. **Design and Implementation of RISC I**. [S.l.]: EECS Department, University of California, Berkeley, 1982. (UCB/CSD-82-106).

THOZIYOOR, S.; MURALIMANOHAR, N.; AHN, J. H.; JOUPPI, N. **CACTI 5.1**. [S.l.]: HP Laboratories, Palo Alto, 2008. Disponível em: <http://www.hpl.hp.com/techreports/2008/HPL-2008-20.pdf>. Acesso em: setembro de 2015.

TSAI, Y.-Y.; CHEN, C.-H. Energy-Efficient Trace Reuse Cache for Embedded Processors. **IEEE Trans. VLSI Syst.**, [S.l.], v.19, n.9, p.1681–1694, 2011.

WITEK, R. T.; MONTANARO, J. StrongARM: A High-Performance ARM Processor. In: COMPCON, 1996. **Anais...** [S.l.: s.n.], 1996. p.188–191.