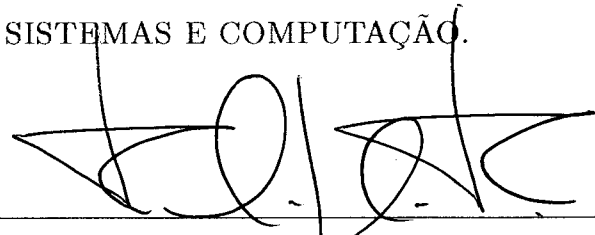
MEMORIZAÇÃO E REUSO DINÂMICO DE TRAÇOS EM UMA ARQUITETURA DE PROCESSADOR JAVA

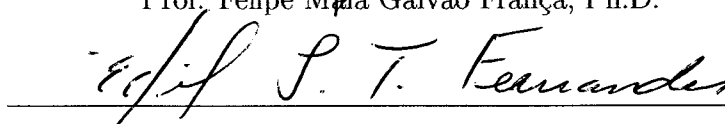
Bruno Rodrigues Silva

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

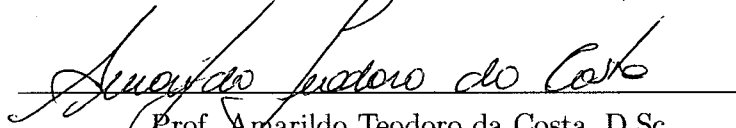
Aprovada por:



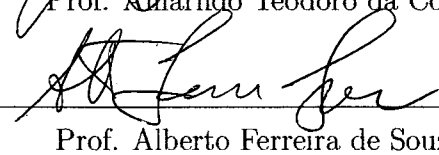
Prof. Felipe Maia Galvão França, Ph.D.



Prof. Edil Severiano Tavares Fernandes, Ph.D.



Prof. Amarildo Teodoro da Costa, D.Sc.



Prof. Alberto Ferreira de Souza, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

FEVEREIRO DE 2006

SILVA, BRUNO RODRIGUES

Memorização e Reuso Dinâmico de Traços em uma Arquitetura de Processador Java [Rio de Janeiro] 2006

XVI, 106 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2006)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

1. Reuso Dinâmico de traços
2. Arquitetura Java
3. Sistemas Embarcados

I. COPPE/UFRJ II. Título (série)

*"Nunca ande pelo caminho traçado,*

*pois ele conduz somente até onde os outros foram".*

*Alexandre Graham Bell*

# Agradecimentos

Primeiramente a Deus que, com sua misericórdia infinita, sempre esteve ao meu lado, mesmo não sendo eu digno de tantas bênçãos. A cada dia vejo o quão pequeno sou diante de tanta magnitude e bondade do Pai, e a cada dia vejo que DEUS É FIEL!

Ao professor Felipe Maia Galvão França que, com seu humor inteligente e de forma tão simples e humilde, me acolheu como seu orientando e possibilitou que a minha experiência de viver dois anos em função do mestrado fosse a mais proveitosa possível. Agradeço pela sugestão do tema para o desenvolvimento dessa pesquisa e deixo claro que seus ensinamentos ficarão para sempre em minha memória. Sem você não teria conseguido.

À minha família que sempre acreditou no meu potencial e mesmo com todas as dificuldades sempre apoiou meus estudos no que pôde. Mãe, mais uma vez obrigado por ter me conscientizado da importância dos estudos.

Ao grande amigo Renato Afonso Conta Silva, eterno companheiro desde os tempos de graduação. No passado lutamos juntos por dias melhores e hoje somos gratos a Deus pois esses dias chegaram.

Às grandes amizades formadas durante os dois anos do mestrado: João Maurício, Bernardo, Patrícia, Raquel, Fernanda, Fabiano, Danilo, Eduardo Melione e aos

novatos do LAM: Ivomar, Leandro e Elias. Pessoas que jamais serão esquecidas.

À Universidade Federal do Rio de Janeiro, e em especial ao Programa de Engenharia de Sistemas e Computação da COPPE por ter contribuído para minha formação e ter tão bem recebido esse mineiro.

A todos os funcionários do Programa de Engenharia de Sistemas e Computação da COPPE, e em especial à Cláudia, Sônia, Adilson, Itamar, Solange, Lurdes e Mercedes. Vocês foram minha família nesses dois anos de luta.

Ao NACAD - Núcleo de Atendimento de Computação de Alto Desempenho da COPPE, pelo uso do Cluster Mercury, extremamente importante durante a execução dos experimentos.

Ao Depto. de Informática da UFRGS e em especial ao professor Luigi Carro e ao amigo Antônio Carlos Beck Filho pela grande atenção dispensada no início dessa pesquisa.

Ao professor Ulisses Leitão que desde os tempos de graduação sempre acreditou e apostou no meu potencial e sempre me apoiou nas decisões mais difíceis. Compartilho essa conquista com você, amigo.

Ao apoio financeiro da CAPES, CNPq e Instituto Doctum de Tecnologiá.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

# MEMORIZAÇÃO E REUSO DINÂMICO DE TRAÇOS EM UMA ARQUITETURA DE PROCESSADOR JAVA

Bruno Rodrigues Silva

Fevereiro / 2006

Orientador: Felipe Maia Galvão França
Programa: Engenharia de Sistemas e Computação

Esta pesquisa realiza uma análise do comportamento do mecanismo *DTM (Dynamic Trace Memoization)* implementado em uma arquitetura de processador Java. Tal mecanismo se propõe à memorizar e reusar dinamicamente traços redundantes, *i.e.*, sequências de instruções redundantes, buscando evitar a re-execução destes traços quando eles forem novamente alcançados com o mesmo conjunto de valores de entrada observados e registrados anteriormente durante a execução do programa. Para tanto, o mecanismo DTM foi modificado para se adaptar à arquitetura Java, dando origem ao mecanismo *JDTM (Java Dynamic Trace Memoization)*. Foi avaliada a sua efetividade através de simulação dirigida à execução utilizando o simulador de um microprocessador *pipeline* escalar Java. De acordo com os resultados obtidos da execução de um conjunto de programas típicos de sistemas embarcados, alcançou-se a aceleração de 12% (média harmônica). Esta aceleração é justificada pela redução do número de instruções executadas e pela redução do número de bolhas inseridas no *pipeline* em vista das dependências de recursos, de dados e de controle.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DYNAMIC TRACE MEMOIZATION AND REUSE IN A JAVA PROCESSOR ARCHITECTURE

Bruno Rodrigues Silva

February / 2006

Advisor: Felipe Maia Galvão França

Department: Computing and Systems Engineering

This research make an analysis of the behavior of DTM (Dynamic Trace Memoization) mechanism implemented in a Java processor architecture. Such mechanism dynamically memorize and reuse redundant traces, i.e., sequences of redundant instructions to prevent the re-execution of them when they are reached again with the same set of input values observed previously during the execution of a program. The DTM mechanism was modified for the Java architecture, originating the JDTM (Java Dynamic Trace Memoization) mechanism. Using a Java scalar pipelined microprocessor simulator was employed to evaluate the effectiveness of the mechanism. According to the results of the execution of a typical programs set of embedded systems, it reached 12% speedup (harmonic average). This speedup is justified by the reduction in the number of executed instructions and by the reduction in the number of inserted bubbles in pipeline because of the resources, data and control dependencies.

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Palavras-chave

1. Reuso Dinâmico de Traços.

2. Arquitetura Java.

3. Sistemas Embarcados

# Capítulo 1

# Introdução

A demanda por processadores com maior capacidade de processamento e menor custo vem crescendo consideravelmente a medida que o aumento da complexidade do *software* se torna algo evidente e necessário. Como forma de atender a essa demanda, o projeto de computadores pode se apoiar em duas alternativas para aumento de desempenho:

1. Redução do tempo do ciclo de *clock*, principalmente através de melhorias na tecnologia de semicondutores.

2. Melhorias no nível arquitetural, com a concepção de novos mecanismos para permitir a execução de mais instruções por ciclo de *clock*.

A primeira alternativa é algo que se tornou momentaneamente insustentável para a Intel Corp, por exemplo. A companhia passará a adotar um conceito para seus processadores que não prioriza a velocidade do *clock* [15]. Tanto que recentemente desistiu do projeto Tejas (chip antes previsto para o segundo trimestre de 2005, que venceria a barreira de 4 GHz). Especula-se que a decisão ocorreu em meio a preocupações de que os novos processadores estavam sujeitos a superaquecimento devido a gasto excessivo de energia, e consequente obrigatoriedade de sistemas de

refrigeração sofisticados. No mercado de sistemas embarcados, altas freqüências de ciclo de *clock* são ainda mais preocupantes devido ao excessivo consumo de energia.

A segunda alternativa foca todo o esforço em melhorias na implementação da arquitetura do processador. Tais melhorias visam, entre outras coisas, explorar o paralelismo espacial/temporal no nível de instrução, aumentando assim o número de instruções executadas por unidade de tempo através da redução do *CPI - Ciclos por Instrução.*

Soluções voltadas ao processamento paralelo são alternativas arquiteturais a fim de melhorar o desempenho sem aumentar a frequência de operação do processador. Por exemplo, como solução ao problema de superaquecimento, a Intel Corp. passou a apostar nos *chips dual core*, que combinam dois processadores para aumentar o desempenho.

Pela equação de desempenho apresentada a seguir [10] (Equação 1.1), temos que o tempo de um processador para realizar uma tarefa pode ser definido por:

$$Tempo\ da\ CPU = IC * CPI * T \tag{1.1}$$

Onde:

- $IC$ é o número total de instruções executadas;
- $CPI$ é o número médio de ciclos por instrução, e
- $T$ é o tempo médio de um ciclo.

Analisando a Equação 1.1, observa-se que o tempo de execução de um processador pode ser melhorado pela redução do número de instruções necessárias ao cumprimento de uma determinada tarefa. Uma solução seria a concepção de me-

canismos que identifiquem e eliminem a redundância existente em um programa, o qual é objeto de investigação desta pesquisa.

## 1.1 Motivação

Pesquisas recentes vêm contribuindo com novas opções de implementação das arquiteturas existentes de forma a explorar a redundância implícita na execução de programas. Estes estudos sugerem que, durante a execução de um programa, muitas instruções (e grupos de instruções) são executadas repetidamente com as mesmas entradas gerando os mesmo resultados [20, 18]. Sodani e Sohi mostraram que em muitos *benchmarks*, cerca de 80% à 90% das instruções dinâmicas produzem resultados já computados [22]. Isto porque diferentes instâncias de uma mesma instrução estática produzem resultados repetidos se seus operandos são repetidos (no caso comum). Adicionalmente, o resultado de uma instrução pode ser repetido mesmo se seus operandos não o são (e.g. o resultado de uma instrução de comparação pode ser o mesmo com uma ampla variedade de valores de entrada). Entretanto, em alguns casos o resultado de uma instrução pode não ser repetido mesmo sendo verificada a repetição de seus operandos (e.g., uma instrução *load* que lê diferente valores do mesmo endereço de memória).

Sodani e Sohi realizaram uma análise empírica da repetição de resultados, onde foi constatado que, em vários programas a maior parte desta redundância se deve à um pequeno número de instruções estáticas. Utilizando diferentes conjuntos de entrada para a execução dos programas, eles observaram também que a repetição é uma característica de como a computação é expressa em um programa e não uma característica dos valores de entrada como se poderia imaginar [21].

Segundo Lipasti e Shen [14], este fenômeno, denominado *Localidade de Valores*, existe pela forma na qual programas são projetados. Eles observaram também que

em programas tais como ambientes de execução e sistemas operacionais existem várias penalidades de desempenho devido à natureza de propósito geral dessas aplicações. Isto é, programas são implementados para manipular não somente condições excepcionais e entradas inválidas que raramente são encontradas, mas também são freqüentemente projetados com reuso de código. Isto implica diretamente em redundância das instruções que compõem o código de tais programas, o que significa que a maioria das instruções estáticas exibem muito pouca variação nos valores que elas produzem durante o curso da execução de um programa.

Pode-se ilustrar essa situação através do exemplo da Figura 1.1. O método `int busca (int x)` procura pelo valor de um inteiro `x` (chave de busca) no *array* `vetorDados` de um tamanho `n` qualquer. O método `void main ()` realiza duas chamadas ao método `int busca ()`, pesquisando um elemento diferente no mesmo *array* à cada chamada. Quando o método `int busca (int x)` for chamado, ele acessa elemento por elemento do *array* até encontrar uma correspondência com o valor de `x` ou até que o final do *array* seja alcançado. A Figura 1.1(b) exibe as instruções estáticas em um nível intermediário, referente ao código do método `int busca (int x)`. A Figura 1.1(c) exibe as instruções dinâmicas desse código que são geradas pela primeira chamada ao método `int busca (int x)`. Em cada iteração do loop, a instrução 2 depende do parâmetro `n`, as instruções 3 e 4 dependem do *array* `vetorDados`, a instrução 5 depende do *array* assim como do valor a ser pesquisado, e a instrução 6 depende da variável de indução. Mesmo se o método `int busca (int x)` for chamado novamente (Figura 1.1(d)) com uma chave de busca diferente da execução anterior, ele pesquisará no mesmo *array* e logicamente no mesmo tamanho `n`. Logo, todas as diferentes instâncias das instruções de 1 à 4 e 6 produzem os mesmos resultados que foram produzidos previamente quando o método `int busca (int x)` foi chamado (um total de `n` instâncias das instruções 2,3,4 e 6). Somente as instâncias da instrução 5 produzem resultados diferentes da última execução do método. O método `int busca (int x)` foi escrito para operar sobre

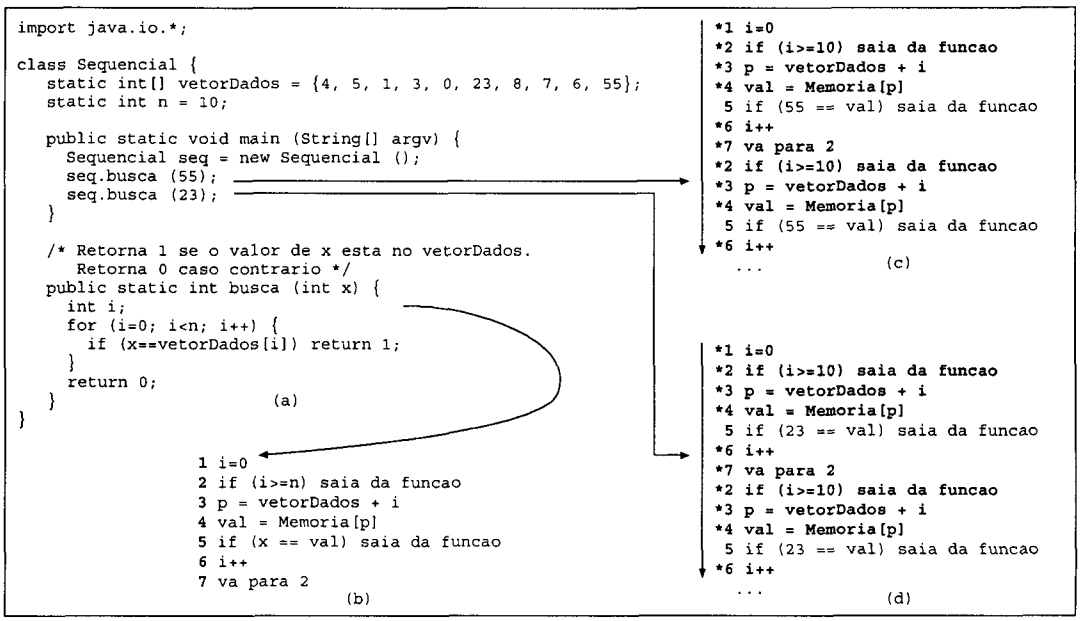



Figura 1.1: Cenário ilustrando redundância em um programa Java. As instruções dinâmicas marcadas com '*' realizam a mesma computação para ambas as chamadas ao método `int busca (int x)`

um *array* genérico, mas nesse caso apenas um dos parâmetros (chave de pesquisa) foi alterado entre uma chamada e outra, o que justifica a redundância dos resultados das instruções 1 à 4 e 6. Além disso, mesmo se todos os parâmetros fossem diferentes para a segunda chamada, ainda sim as *min(n1, n2)* instâncias da instrução 6 na segunda chamada do método `int busca (int x)`, produziriam a mesma sequência de valores da chamada anterior.

Portanto, a exploração de toda essa computação redundante pode ser uma alternativa arquitetural viável a fim de aumentar o desempenho de novas gerações de processadores. A próxima subseção apresenta o estado da arte na concepção de mecanismo que exploram a computação redundante através do *Reuso de valores* e *Predição de valores.*

## 1.2 Trabalhos Relacionados

Existem dois fatores que limitam a quantidade de paralelismo no nível de instrução que pode ser extraído de programas seqüenciais: *Fluxo de Controle* e *Fluxo de Dados*. O Fluxo de Controle limita o paralelismo pela imposição de serialização nos pontos de desvio e retorno do grafo de fluxo de controle do programa. O Fluxo de Dados limita o paralelismo pela imposição de serialização em pares de instruções que são dependentes de dados (*i.e.* uma instrução necessita do resultado de outra antes de iniciar a execução).

Duas técnicas têm sido propostas para evitar a execução serial de instruções apresentando dependência de dados verdadeira ("*RAW - Read After Write*"):

- **Predição de valores:** É uma técnica *especulativa* que explora redundância em programas através da predição dos valores que serão produzidos ou usados pelas instruções. Os valores preditos podem ser de três tipos:
  - ***constant:*** onde o mesmo valor ocorre novamente;
  - ***stride:*** onde existe um intervalo fixo entre dois valores subseqüentes;
  - ***non-stride:*** onde existe nenhuma ou uma complexa correlação entre dois valores subseqüentes;

  A Figura 1.2 esboça uma simples implementação de predição de valores chamada *Last Value Predictor* [17], que é adequada para o predição de valores do tipo *constant*. Neste caso, o endereço da instrução é utilizado para acessar a tabela de predição, onde o valor da predição está armazenado.

  No entanto, o uso do *Last Value Predictor* impede a predição de valores do tipo *stride*, *i.e*, uma sequência de valores baseados em um valor inicial e com intervalo fixo, tais como: *1, 2, 3, 4, 5, 6, ... ou 1, 3, 5, 7, 9,...* Para efetuar a predição de valores deste tipo, pode-se utilizar o *Stride Predictor* (Figura

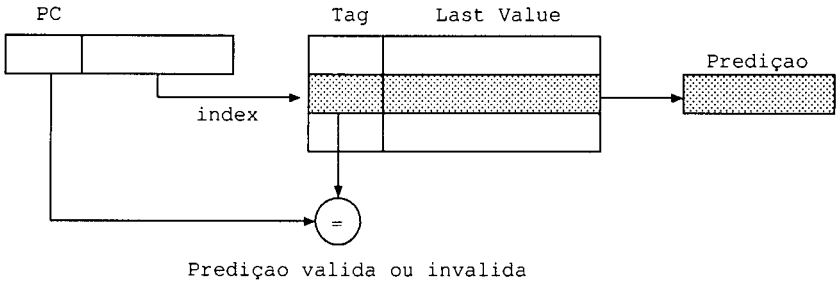



Figura 1.2: Técnica de predição de valores - *Last Value Predictor.*

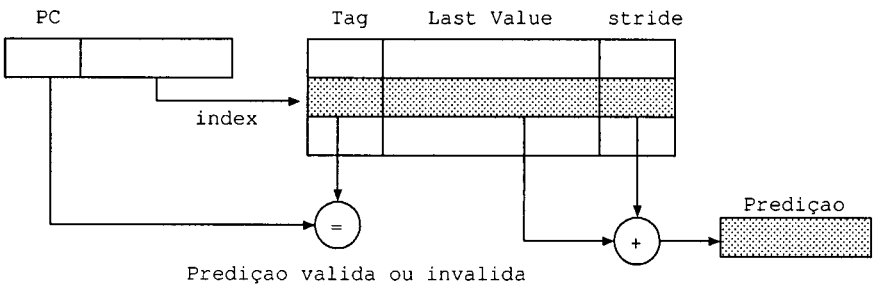



Figura 1.3: Técnica de predição de valores - *Stride Predictor.*

1.3), o qual estende o *Last Value Predictor* pela adição do campo *stride* que contém a diferença, *i.e*, o intervalo entre os valores da sequência. Uma predição é realizada somando-se o valor do campo *Last Value* com o valor do campo *stride.*

- **Reuso de valores:** É uma técnica *não especulativa* que explora o fato de que muitas instruções ou seqüências de instruções dinâmicas são repetidamente executadas e a maioria destas repetições possuem os mesmos valores de entrada e, assim, geram o mesmo resultado [21]. Técnicas de Reuso de valores exploram este fato através da *"buferização"* de valores de entradas e seus correspondentes resultados associados à uma instrução ou traço por exemplo. Quando uma instrução ou traço é encontrado novamente e seus valores de entrada atuais correspondem aos valores armazenados no *buffer*, a execução pode ser evitada pois o resultado pode ser obtido do próprio *buffer.* A Figura 1.4 esboça o

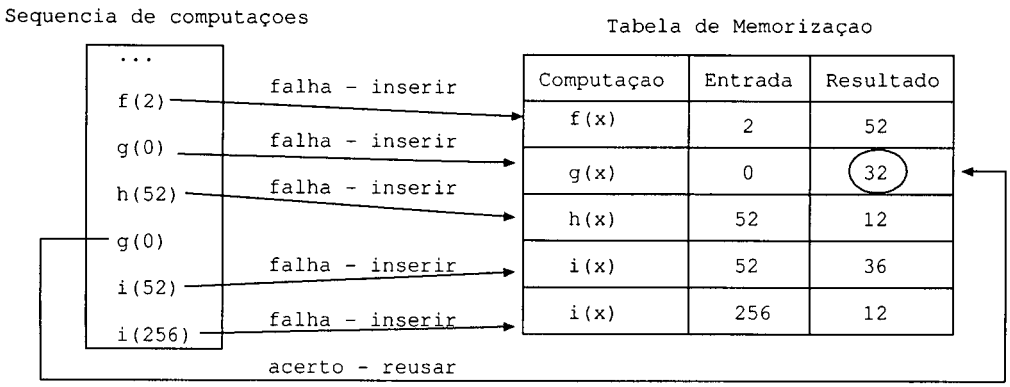



Figura 1.4: Técnica de Reuso de valores.

funcionamento da técnica de reuso de valores, onde `f`, `g`, `h` e `i` representam computações arbitrárias. Antes da execução de cada computação é realizada uma consulta à tabela de memorização utilizando um identificador qualquer da computação e o respectivo valor de entrada. Em caso de falha, a instância da computação é inserida. Caso contrário, o resultado é obtido da tabela e a computação deixa de ser executada.

### 1.2.1 Explorando a redundância através de Reuso de valores

Como forma de explorar a redundância através de Reuso de valores no nível de instruções, Sodani e Sohi [20] propuseram um mecanismo próprio para implementação em *hardware* denominado *Dynamic Instruction Reuse*, que memoriza instruções dinâmicas, *i.e.*, valores dos operandos de entrada e resultado, buscando oportunidades de reusar estes resultados. Além disso, eles também fizeram uma estimativa de qual a percentagem de redundância apresentada em programas que pode ser capturada pelo mecanismo *Dynamic Instruction Reuse* [22].

Utilizando outra abordagem, da Costa [2] e da Costa *et al.* [3] propõem o reuso no nível de traços de instruções, dando origem ao mecanismo *DTM - Dynamic Trace Memoization*, o qual busca memorizar e reusar traços de instruções redundantes em

uma arquitetura superescalar de 3 operandos do tipo RISC, reduzindo (i) o número de instruções executadas, (ii) desvios previstos incorretamente e (iii) os caminhos críticos determinados por dependências verdadeiras.

Gonzalez *et al.* apresentaram um estudo do potencial de reuso de instruções e traços. Para isso, eles definiram uma estrutura de memorização denominada *Reuse Trace Memory (RTM)*, que armazena informações sobre os traços selecionados idealmente para posterior reuso. Eles também apresentaram os limites superiores para redundância e aceleração que podem ser explorados pelo reuso de traços [9].

Todas as pesquisas relatadas têm como alvo uma arquitetura de 3 operandos do tipo RISC. Programas Java compilados para *bytecodes*[1] são diferentes de programas C compilados para uma arquitetura RISC porque (i) eles são altamente orientados à objetos com muitas chamadas de métodos ligados dinamicamente, e (ii) são compilados para uma arquitetura baseada em pilha. A fim de investigar o comportamento do mecanismo DTM sobre a arquitetura Java e avaliar seu impacto no desempenho devido ao item (ii), Silva *et al.* [19] propuseram uma variação do mecanismo DTM, dando origem ao mecanismo *JDTM - Java Dynamic Trace Memoization*, que memoriza e reusa dinamicamente traços de *bytecodes* Java compilados para uma arquitetura de processador Java que executa *bytecodes* nativamente.

### 1.2.2 Explorando a redundância através de Predição de valores

Lipasti e Shen [14], visando ultrapassar os limites do Fluxo de Dados devido à serialização de seqüências de instruções dependentes de dados, exploraram a Localidade de Valores no contexto de predição de valores através de execução especulativa. Desta forma, instruções dependentes podem iniciar a execução evitando aguardar que seus operandos de entrada sejam produzidos por instruções anteriores na sequên-

[1] Os termos instrução e bytecode serão usados indistintamente neste trabalho.

cia de execução.

### 1.2.3 Semelhanças e diferenças entre Reuso de valores e Predição de valores

Ambas as técnicas buscam reduzir o tempo de execução de programas pela redução das restrições impostas pelo Fluxo de Dados. Para isso, elas usam a redundância em programas para determinar especulativamente (Predição de valores) ou não-especulativamente (Reuso de valores) o resultado de instruções sem executá-las. A vantagem é que as instruções não precisam esperar pela conclusão das "instruções produtoras" para iniciarem a execução. Elas podem executar mais cedo usando os resultados obtidos pelas duas técnicas acima, reduzindo, assim, as restrições do fluxo de dados.

A abordagem de Reuso de valores não é especulativa e, portanto, é mais conservadora. Por exemplo, se os valores dos operandos de entrada de uma instrução ou traço não estiverem disponíveis a tempo do teste de reuso, tal instrução ou traço não será reusado; ou, uma instrução que produz o mesmo resultado mas com diferentes entradas (por exemplo, operações lógicas e *loads*), também não serão identificados por tal abordagem. Porém, Predição de valores pode fazer previsões corretas para cada caso, visto que essa técnica não depende dos valores de entrada estarem disponíveis, nem serem os mesmos da última execução. Logo, o Reuso de valores captura menos redundância em programas do que Predição de valores. Porém, o Reuso de valores não impõe penalidades devidas à previsões incorretas. Além do mais, na abordagem de Predição de valores, as instruções precisam de qualquer forma serem executadas para confirmar ou não a especulação. No caso do Reuso de valores, as instruções reusadas não necessitam ser executadas, reduzindo assim a demanda por recursos do processador e o número total de instruções executadas.

Uma abordagem híbrida tira proveito dos benefícios de ambas as técnicas. Esse

é o caso da abordagem de Pilla *et al.* [17], onde é proposto e avaliado um mecanismo implementado em *hardware* capaz de realizar predição de valores para reuso de traços através de especulação. *Reuse through Speculation on Traces (RST)* [16], como foi chamado, aumenta o reuso de traço e oculta dependências verdadeiras pela combinação de Predição de valores e Reuso de valores. RST possibilita que traços sejam *regularmente reusados*, quando todos os valores de entrada estão disponíveis e correspondem com os valores armazenados no *contexto de entrada do traço*, ou *especulativamente reusados* quando existem valores ainda não conhecidos no contexto de entrada de um traço. Além disso, traços que não poderiam ser reusados em outras abordagens podem ser reusados explorando sua previsibilidade e não somente sua redundância.

### 1.2.4 Outros Trabalhos

RST, assim como DTM e JDTM não inclui instruções de acesso à memória nos traços construídos, evitando problemas de inconsistência com a memória de dados. Uma variação do mecanismo DTM foi apresentada por Viana [4], onde essa classe de instruções pode ser inclusa nos traços construídos e reusados. Da mesma forma, uma variação do mecanismo RST foi apresentada por Laurino *et al.* [13], onde instruções de leitura de dados da memória (*loads*) são inseridas no conjunto de instruções válidas para serem utilizadas na construção e conseqüente reuso especulativo dos traços.

## 1.3 Objetivos e contribuições

Este trabalho examina o reuso de computação redundante (Reuso de valores) no contexto de aplicações embarcadas Java através de um mecanismo implementado em *hardware* que memoriza e reusa dinamicamente traços de *bytecodes* Java.

Além do impacto no desempenho do processador que incorpora tal mecanismo, é objetivo dessa investigação identificar:

- O percentual de instruções reusadas. Ou seja, o percentual de reuso exposto pelo mecanismo;
- A distribuição percentual do número de instruções encapsuladas nos traços reusados;
- As melhores opções em termos de tamanho e associatividade das tabelas de memorização utilizadas pelo mecanismo;
- A distribuição percentual do número de valores no contexto de entrada e saída dos traços reusados;
- A distribuição do número de instruções de desvios realizados encapsulados nos traços reusados;
- As instruções que limitam o tamanho dos traços construídos;
- Trabalhos futuros a serem realizados no contexto de exploração de redundância em aplicações embarcadas Java.

O mecanismo de memorização e reuso de traços de *bytecodes* proposto neste trabalho possibilitou, em média, um percentual de reuso de 22% dos *bytecodes* executados em 8 *benchmarks* típicos de sistemas embarcados, o que provocou um incremento, em média harmônica, de 12% no desempenho do processador substrato. Este trabalho realizou também um estudo sobre a sensibilidade do mecanismo à variações no tamanho e na associatividade das tabelas de memorização, bem como identificou a melhor relação custo/benefício em se tratando do número de entradas das duas tabelas de memorização utilizadas pelo mecanismo.

## 1.4 Estrutura do Trabalho

Este trabalho está dividido em 5 partes. A primeira parte compreende o presente capítulo introdutório com a motivação da pesquisa, trabalhos relacionados e objetivo. O Capítulo 2 apresenta o conhecimento de fundo, descrevendo de forma resumida (i) o mecanismo DTM como foi originalmente desenvolvido por da Costa [2], (ii) a arquitetura do processador utilizado como substrato para a implementação da solução proposta neste trabalho e (iii) o simulador ciclo-a-ciclo utilizado na realização dos experimentos. A terceira parte é composta pelo Capítulo 3, onde é apresentada a solução encontrada para memorização e reuso de traços de *bytecodes*. Neste, é apresentado o mecanismo *JDTM (Java Dynamic Trace Memoization)* e suas principais diferenças em relação ao mecanismo DTM. O Capítulo 4 descreve os experimentos e a análise dos resultados obtidos. Por fim, o último capítulo aborda as principais conclusões e os possíveis trabalhos futuros.

# Capítulo 2

# *Background*

O presente capítulo apresenta a base de conhecimento usada no desenvolvimento desta pesquisa. Para tanto é descrita a arquitetura do processador *FemtoJava Low Power 32 bits* [6] utilizado como processador substrato para a implementação do JDTM, bem como o simulador *CACO-PS* [7] utilizado nos experimentos. Também é apresentado o mecanismo *Dynamic Trace Memoization*, como foi originalmente desenvolvido para a arquitetura MIPS [10].

## 2.1 Processador FemtoJava

### 2.1.1 FemtoJava multiciclo 8 *bits*

Nos dias atuais Java é uma das tecnologias mais populares no ambiente de aplicações embarcadas. A maioria dos aparelhos celulares modernos suportam Java, assim como outros sistemas embarcados (*personal digital assistants* e *pagers*, por exemplo).

Visando explorar o crescente mercado de sistemas embarcados, Ito *et al.* [12] apresentam uma nova estratégia de projeto para implementação de aplicações em-

barcadas descritas unicamente em Java, de forma à manter a compatibilidade de *software* durante todo o processo de projeto. Além disso, o *hardware* destino de tais aplicações é um único *chip FPGA (Field Programable Gate Array)*, devido ao seu baixo custo e fácil reconfiguração.

Esta nova estratégia baseia-se em um ambiente de projeto para aplicações específicas através da geração de processadores Java. Neste ambiente, denominado *SASHIMI* (*Systems As Software and Hardware In MIcrocontrollers*), o projetista fornece uma aplicação Java para ser analisada e otimizada para execução em um *ASIP* (*Application Specific Instruction Set Processor*), somado à um *ASIC* (*Application Specific Integrated Circuit*) opcional, ambos sintetizados em um único *chip FPGA*. Esta abordagem é também caracterizada pela alta integração de funções, ambiente de execução simplificado, nenhuma necessidade de um novo compilador e compatibilidade de *software*.

Para dar suporte ao SASHIMI no nível de *hardware*, foi projetado um processador Java *multiciclo de 8 bits* chamado *FemtoJava*, que executa *bytecodes* nativamente [11]. Este processador, destinado à execução de aplicações específicas do mercado de sistemas embarcados, possui uma arquitetura *Harvard* de pilha com conjunto de instruções reduzido e pequeno tamanho. A ferramenta SASHIMI realiza uma análise dos *bytecodes* de uma aplicação e gera a unidade de controle do FemtoJava, através de uma descrição em *VHDL* (*VHSIC Hardware Description Language*), suportando somente as instruções usadas pela aplicação. Conseqüentemente, o tamanho da unidade de controle é diretamente proporcional ao número de instruções diferentes usadas.

### Por que usar uma arquitetura de Pilha?

Máquinas de pilhas são conhecidas pelo seu mecanismo de execução simplificado. Estas máquinas não codificam informações dos operandos na palavra de instrução, resultando em um código mais compacto e portável visto que elas necessitam de uma

menor quantidade de *bits* para codificar uma instrução e não são tão dependentes da organização do banco de registradores. Vale mencionar que programas compilados para máquinas *RISC* (*Reduced Instruction Set Computer*) podem ser 1.5 à 2.5 vezes maiores que suas versões compiladas para máquinas *CISC* (*Complex Instruction Set Computer*) [11]. Além disso, o programa de uma máquina de pilha pode ser 2.5 à 8 vezes menor que o mesmo programa compilado para uma máquina *CISC* [11].

Atualmente várias soluções focadas em sistemas embarcados possuem excessivos recursos para atender: *RTOS* (*Real Time Operating System*); implementações específicas da *JVM* (*Java Virtual Machine*); suporte *multithreading* e *garbage colector*. Entretanto, para muitas aplicações embarcadas simples, estes mecanismos impõem grande *overhead*, relativo ao custo de implementação e potência. Para estas aplicações mais simples, é necessário desconsiderar possíveis críticas quanto ao desempenho da arquitetura de pilha e avaliar a sua capacidade em atender as necessidades de aplicações específicas.

Alguns benefícios, problemas e questões abertas no uso de processadores baseados em uma arquitetura de pilha para suportar execução nativa de *bytecodes* Java, são discutidos em por Ito *et al.* [11].

### 2.1.2 Breve descrição de alguns *bytecodes* Java

Ao contrário de máquinas *RISC* ou *CISC* que possuem uma arquitetura orientada a acumulador ou a registrador, Java possui uma arquitetura orientada a pilha. Logo, o modelo de programação é um pouco diferente de outros modelos presentes em processadores mais comuns como *MIPS*, *Intel x86* e outros que se baseiam em transferência de valores entre registradores. Esta subseção visa descrever de forma sucinta, através de um exemplo de código Java, alguns dos 201 *bytecodes* disponíveis no conjunto de instruções da arquitetura Java.

Na Figura 2.1 pode-se observar duas seqüências de instruções para a operação `C = A + B`, uma para uma arquitetura do tipo *load-store*, e outra para uma arquitetura de pilha, assumindo que os valores das variáveis `C`, `A` e `B` estão inicialmente em memória.

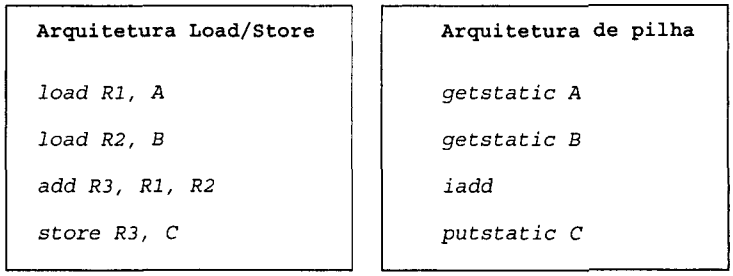



Figura 2.1: Sequência de instruções para a operação `C = A + B` em uma arquitetura *load/store*(a) e uma arquitetura de pilha (b).

Como pode ser observado, na arquitetura *load/store*(Figura 2.1(a)), os valores das variáveis `A` e `B` precisam ser lidos da memória utilizando a instrução `load`, e armazenados em registradores temporários (`R1` e `R2` respectivamente). O resultado da instrução `add R3, R1, R2` precisa ser escrito de volta na memória, através da instrução `store`[1] que utiliza o registrado `R3` como fonte.

Na arquitetura de pilha (Figura 2.1(b)), os valores também precisam ser lidos da memória, porém o destino destes valores já está implícito na própria instrução `getstatic`. Este destino é o topo de uma pilha que pode, inclusive, ser mapeada em um banco de registradores. Em seguida, a instrução `add` desempilha os dois últimos valores empilhados, realiza uma operação de soma e empilha o resultado. Assim, como na arquitetura *load/store*, o resultado empilhado precisa ser escrito de volta na memória, para isso usa-se a instrução `putstatic`.

Esta pilha, na realidade, é chamada de pilha de operandos (do inglês *operand*

[1]Na arquitetura *load/store*, instruções da classe aritméticas e lógicas não podem possuir operandos em memória.

*stack*). É necessário ressaltar a diferença do nome pois, além da pilha de operandos, o processador Java também faz uso da pilha para desempenhar outras funções, entre elas guardar[2] os valores das variáveis automáticas, isto é, das variáveis locais do método, além de guardar informações de retorno para os métodos que foram invocados, como também ocorre em outros processadores que não implementam uma pilha de operandos.

Como pode ser observado na Figura 2.2, que exibe uma pilha típica que pode ser implementada em um processador Java, a passagem de parâmetros de um método A para um método B é facilitada através da intercalação da pilha de operandos do método A com o início do *pool* de variáveis locais do método B. Ou seja, o método A pode passar parâmetros ao método B, simplesmente empilhando estes parâmetros em sua pilha de operandos e informando ao método B quantos parâmetros estão sendo passados. Assim, o método B pode iniciar seu pool de variáveis locais justamente em uma posição da pilha que inclua os valores empilhados pelo método A.

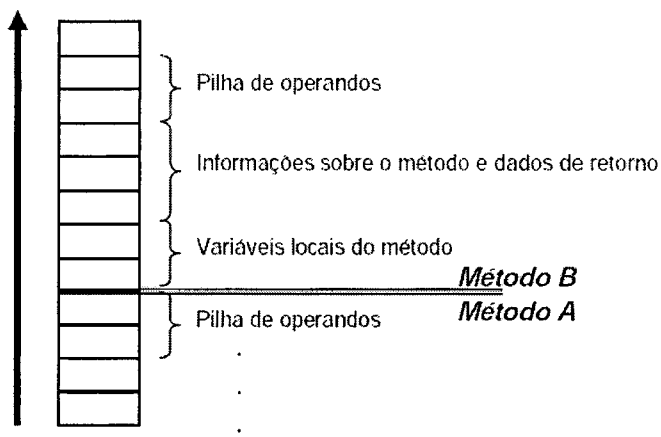



Figura 2.2: Organização da pilha de um processador Java típico.

A versão multiciclo do FemtoJava implementa a pilha na memória de dados, porém a versão *pipeline* do FemtoJava, que será descrita na próxima subseção, faz uso do banco de registrados para a implementação da pilha, mas a fim de reduzir a

[2] A área da pilha, reservada para essa função é chamada de *pool de variáveis locais.*

Tabela 2.1: Instruções suportadas pelo processador FemtoJava.

| Classe | *Bytecodes* |
|---|---|
| Aritméticas e lógicas | `iinc, iadd, isub, iand, ior, ixor, ineg, ishl, ishr, iushr, idiv, irem, imul` |
| Controle de fluxo | `ifeq, ifne, iflt, ifge, ifgt, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, return, ireturn, areturn, invokestatic, if_acmpne, goto` |
| *Load/Store* | `iload, iload_0, iload_1, iload_2, iload_3, istore, istore_0, istore_1, istore_2, istore_3, aload, aload_0, aload_1, aload_2, aload_3` |
| Operações de Pilha | `iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, bipush, sipush, pop, pop2, dup, dup2, dup_x2` |
| *Acesso à Memória* | `iaload, aaload, baload, caload, saload, arraylength, ldc, ldc_w, getstatic, putstatic, bastore, iastore` |
| Nops e operações estendidas | `nop, load_idx, store_idx, sleep` |

área do processador, ele continua guardando na memória de dados as informações de retorno de métodos, pois essas são pouco requisitadas e portanto o acesso a elas não impõem uma redução muito alta do desempenho e um aumento do consumo de potência.

A unidade de controle da versão *pipeline* do FemtoJava implementa um subconjunto de 76 instruções Java. Neste subconjunto encontram-se instruções necessárias para operações lógicas e aritméticas de inteiros, desvios condicionais e incondicionais, execução de métodos estáticos, *load/store* de variáveis locais, operações básicas de pilha, acesso à memória, e operações estendidas (não inclusas na ISA original do Java). Este subconjunto pode ser observado na Tabela 2.1. Os *bytecodes* estendidos são necessários para executar instruções de E/S, programação de interrupções e também para colocar o processador em modo suspenso. O processador FemtoJava só pode executar código de classes (isto é, não pode alocar objetos dinamicamente) porque seu conjunto de instruções apenas suporta os *bytecodes* `invokestatic`, `return`, `areturn` e `ireturn` como instruções para manipulação de métodos.

### Um exemplo

Na Figura 2.3 pode-se observar o código Java de um algoritmo de pesquisa sequencial que verifica se um valor de entrada `x` está presente em um vetor de entrada `v`, retornando a posição deste valor no vetor ou -1 caso `x` não seja encontrado. A Classe `Sequencial` possui 3 campos estáticos: `vetorDados`, `n` e `posicao`, correspondendo respectivamente à um *array* de 10 posições, um inteiro que possui o tamanho do *array* e o último sendo um campo que armazenará o resultado da pesquisa. A classe possui três métodos estáticos:

1. **`void main ()`**: método principal necessário para a execução de um aplicativo Java. Este método instancia um objeto da classe `Sequencial` e realiza uma chamada ao método `void initSystem()`.

2. **`void initSystem ()`**: este método realiza uma pesquisa no vetor através de uma chamada ao método `int busca(int x)`, passando o valor `55` como argumento. O valor de retorno do método `int busca(int x)` será armazenado no campo `posicao`.

3. **`int busca(int x)`**: é neste método que está localizado o algoritmo de pesquisa sequencial que utiliza uma estrutura de repetição para acessar todas as posições do vetor, a fim de verificar se pelo menos um dos valores armazenados neste *array* corresponde ao valor do argumento (valor da variável local `x`). O algoritmo retorna a primeira posição (valor da variável local `i`) do vetor que corresponde ao valor procurado ou retorna `-1` caso não seja encontrada correspondência.

Como resultado da compilação deste código Java, é gerado um arquivo *.class* contendo o código de máquina que pode ser executado por uma JVM. Visto que o processador FemtoJava necessita que este código possua um formato específico, este arquivo compilado precisa ser analisado e adaptado para posteriormente ser

```
1 class Sequencial {
2     static int vetorDados[] = {4, 5, 1, 3, 0, 23, 8, 7, 6, 55};
3     static int n = 10;
4     static int posicao;
5
6     public static int busca (int x) {
7         int i;
8         for (i=0; i<n; i++) {
9             if (x==vetorDados[i]) return i;
10        }
11        return -1;
12    }
13    public static void initSystem() {
14        posicao = busca(55);
15    }
16    public static void main(String[] argv) {
17        Sequencial seq = new Sequencial();
18        Sequencial.initSystem();
19    }
20 }
```

Figura 2.3: Algoritmo de pesquisa sequencial implementado em Java.

utilizado como programa de entrada para o processador FemtoJava. A fim de automatizar este processo, a ferramenta SASHIMI, utilizando como entrada o programa compilado (arquivo *.class*), realiza um análise sobre os *bytecodes* certificando-se que todas as instruções pertencem à ISA do processador. Caso a análise identifique que o programa utiliza apenas as instruções inclusas na ISA, ou seja, não utiliza instruções de alocação dinâmica de memória e nem instruções de ponto flutuante, o SASHIMI fornece como saída, dois arquivos no formato[3] *MIF (Memory Initialization File)*. Estes arquivos possuem respectivamente o conteúdo da memória de instruções (*ROM - Read Only Memory*) e de dados (*RAM - Random Acess Memory*).

A Figura 2.4 exibe o conteúdo do arquivo MIF de memória de instruções, que neste caso possui os *bytecodes* do programa Java da Figura 2.3. Este arquivo utiliza o sistema de numeração hexadecimal e possui basicamente um conjunto de linhas no seguinte formato:

[3]Aceito pelo ambiente *Maxplus-II* da Altera Corporation [1].

```
endereço : opcode; - mnemônico
```

Vale mencionar que a memória de instruções é endereçada por *bytes* e cada palavra de dado possui 1 *byte* no caso do FemtoJava multiciclo 8 *bits* ou 4 *bytes* no caso do FemtoJava *Low Power* 32 *bits*.

```
-- MAX+plus II - Memory Initialization File - generated by SASHIMI CAD Tool

WIDTH = 8;
DEPTH = 128;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT BEGIN
   0 : b8; -- invokestatic
   1 : 00; --
   2 : 49; --
   . . ... ..    ...
   . . ... ..    ...
   . . ... ..    ...
  2b : 01; -- Sequencial.busca.(I)I.2
  2c : 01; --
  2d : 03; -- iconst_0
  2e : 3c; -- istore_1
  2f : 1b; -- iload_1
  30 : b2; -- getstatic
  31 : 00; --
  32 : 10; --
  33 : a2; -- if_icmpge
  34 : 00; --
  35 : 12; --
  36 : 1a; -- iload_0
  37 : b2; -- getstatic
  38 : 00; --
  39 : 11; --
  3a : 1b; -- iload_1
  3b : 2e; -- iaload
  3c : a0; -- if_icmpne
  3d : 00; --
  3e : 03; --
  3f : 1b; -- iload_1
  40 : ac; -- ireturn
  41 : 84; -- iinc
  42 : 01; --
  43 : 01; --
  44 : a7; -- goto
  45 : ff; --
  46 : e9; --
  47 : 02; -- iconst_m1
  48 : ac; -- ireturn
  49 : 00; -- Sequencial.initSystem.()V.0
  4a : 00; --
  4b : 10; -- bipush
  4c : 37; --
  4d : b8; -- invokestatic
  4e : 00; --
  4f : 2b; --
  50 : b3; -- putstatic
  51 : 00; --
  52 : 1e; --
  53 : b1; -- return
```

Figura 2.4: Memória de instruções do FemtoJava multiciclo 8 *bits* contendo os *bytecodes* do algoritmo de pesquisa sequencial mostrado na Figura 2.3.

A Figura 2.5 exibe o conteúdo do arquivo MIF de memória de dados, que neste caso possui os dados estáticos do programa Java da Figura 2.3. Este arquivo também utiliza o sistema de numeração hexadecimal e possui um conjunto de linhas no seguinte formato:

```
endereço : dados; - comentário
```

A memória de dados do FemtoJava é endereçada por palavra, onde cada palavra possui 4 bytes de informação.

```
-- MAX+plus II - Memory Initialization File -
        generated by SASHIMI CAD Tool

WIDTH = 16;
DEPTH = 32;

ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT BEGIN
    0   :   00; -- reserved
    .   .   ..      .....
    f   :   00; -- reserved
    10   :   0a; -- Sequencial.n
    11   :   12; -- Sequencial.vetorDados
    12   :   0a; -- Sequencial.vetorDados.length
    13   :   04; -- Sequencial.vetorDados[0]
    14   :   05; -- Sequencial.vetorDados[1]
    15   :   01; -- Sequencial.vetorDados[2]
    16   :   03; -- Sequencial.vetorDados[3]
    17   :   00; -- Sequencial.vetorDados[4]
    18   :   17; -- Sequencial.vetorDados[5]
    19   :   08; -- Sequencial.vetorDados[6]
    1a   :   07; -- Sequencial.vetorDados[7]
    1b   :   06; -- Sequencial.vetorDados[8]
    1c   :   37; -- Sequencial.vetorDados[9]
    1d   :   00; --
    1e   :   00; -- Sequencial.posicao
    1f   :   00; -- no value
END;
```

Figura 2.5: Memória de dados do FemtoJava contendo os dados estáticos do algoritmo de pesquisa sequencial mostrado na Figura 2.3.

Pode-se observar que o programa da Figura 2.4 possui 21 *bytecodes*, dos quais 16 são distintos. A função de cada um destes *bytecodes* é descrita abaixo:

1. **`invokestatic`** *indexbyte1 indexbyte2*: Realiza uma chamada à um método estático endereçado por *sign_ext16((indexbyte1 « 8) | indexbyte2)*. Onde *sign_ext16* significa extensão de sinal em um número de 16 *bits*.

2. **`iconst_0`**: Empilha a constante 0 na pilha de operandos.

3. **`istore_1`**: Desempilha um valor do topo da pilha de operandos e o armazena na variável local 1.

4. **iload_1**: Carrega o valor da variável local 1 e o empilha na pilha de operandos.

5. **getstatic** *indexbyte1 indexbyte2*: Carrega o valor de uma posição da memória de dados, endereçada por *sign_ext16((indexbyte1 « 8) | indexbyte2)*, e o empilha na pilha de operandos.

6. **if_icmpge** *offsetbyte1 offsetbyte2*: Desempilha dois valores da pilha de operandos. Se o sub-topo é maior ou igual ao topo, faz um desvio relativo ao PC para endereço *sign_ext16((offsetbyte1 « 8) | offsetbyte2)* da memória de instruções.

7. **iload_0**: Carrega o valor da variável local 0 e o empilha na pilha de operandos.

8. **iaload**: Desempilha dois valores da pilha de operandos e carrega o valor de uma posição da memória de dados, endereçada por *sub-topo + topo*, empilhando este valor na pilha de operandos.

9. **if_icmpne** *offsetbyte1 offsetbyte2*: Desempilha dois valores da pilha de operandos. Se o valor do sub-topo é diferente do valor do topo, faz um desvio relativo ao PC para endereço *sign_ext16((offsetbyte1 « 8) | offsetbyte2)* da memória de instruções.

10. **ireturn**: Retorno de método. Desempilha um valor da pilha de operandos do método atual e o empilha na pilha de operandos do método invocador. Sai do contexto do método atual através da recuperação das informações de retorno de métodos, armazenadas na memória de dados.

11. **iinc** *index const*: Incrementa o valor da variável local *index*, somando *const* ao seu valor.

12. **goto** *offsetbyte1 offsetbyte2*: Realiza um desvio incondicional para a instrução endereçada por *PC + sign_ext16((offsetbyte1 « 8) | offsetbyte2)*.

13. **iconst_m1**: Empilha a constante *-1* na pilha de operandos.

14. **bipush** *byte*: Empilha o imediato *byte* na pilha de operandos.

15. **putstatic** *indexbyte1 indexbyte2*: Desempilha um valor da pilha de operandos e o armazena em uma posição, endereçada por *sign_ext16((indexbyte1 « 8) | indexbyte2)*, da memória de dados.

16. **return**: Retorno de método. Sai do contexto do método atual através da recuperação das informações de retorno de métodos armazenadas na memória de dados.

### 2.1.3 FemtoJava *Low Power* 32 *bits*

Como mencionado, para suportar a execução nativa de um subconjunto das instruções da arquitetura Java, Ito *et al.* [11] desenvolveram um processador multiciclo de 8 *bits* chamado FemtoJava. Entretanto, visto que a pilha deste processador é implementada em memória, acessos à pilha são acompanhados de um grande consumo de potência. Visando reduzir este consumo, Beck Filho e Carro [6] propõem uma nova versão do processador FemtoJava, denominada FemtoJava *Low Power*, e Gomes *et al.* [8] apresentaram uma implementação deste processador em VHDL.

FemtoJava *Low Power* é um processador *pipeline* de 32 *bits* que executa *bytecodes* Java nativamente e possui a pilha de operandos e o *pool* de variáveis locais mapeados em um banco de registradores, ao contrário do FemtoJava multiciclo que utiliza a memória de dados como meio de armazenamento dessas estruturas.

A Figura 2.6 exibe a pilha do processador FemtoJava *Low Power* implementada em um banco de 64 registradores, que armazena a pilha de operandos e o *pool* de variáveis locais. Vale mencionar que nenhum cuidado é tomado pelo processador ou compilador em se tratando de um possível estouro de pilha. Entretanto, para os *benchmarks* considerados e suas respectivas entradas utilizadas neste trabalho não houve estouro de pilha.

Com um *pipeline* de cinco estágios, este processador alcança maior desempenho

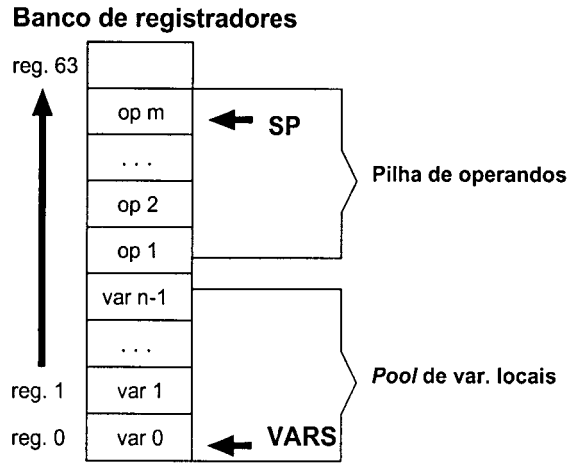



Figura 2.6: Organização da pilha do processador FemtoJava *Low Power*.

e menor consumo de potência, com um pequeno *overheard* de área ocupada, se comparado à versão multiciclo.

Na Figura 2.7 pode-se observar a sequência dos 5 estágios do *pipeline*, que serão detalhados abaixo:

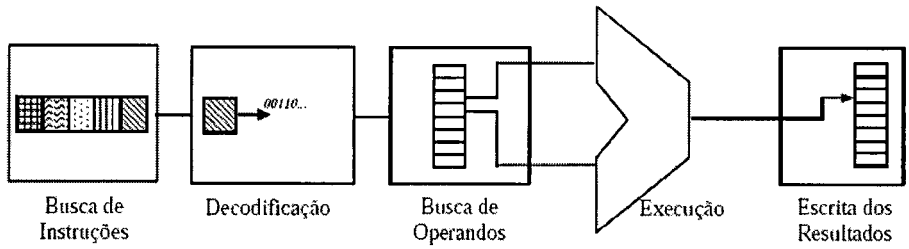



Figura 2.7: Os cinco estágios do *pipeline* do FemtoJava *Low Power*.

1. IF - *Instruction Fetch* ou Busca de Instruções: Este estágio é composto por uma fila de instruções de 9 registradores de 1 *byte* cada (fila de *prefetch*), um registrador para endereçar a memória de instruções (IMAR) e um somador para gerar o endereço da próxima instrução sequencial. Se um endereço não sequencial precisar ser carregado, um multiplexador copia o valor armazenado

no registrador contador de programa (PC) para o registrador IMAR e todas as instruções na fila são desconsideradas. Quando pelo menos 4 posições da fila estiverem livres, uma palavra de 32 *bits* é buscada da memória de instruções[4].

2. **ID** - *Instruction Decode* ou Decodificação de instrução: Neste estágio é realizada a geração da palavra de controle para a instrução que está na primeira posição da fila e informa o tamanho desta instrução ao estágio IF, de forma que a próxima instrução seja deslocada para a primeira posição da fila. Isto é necessário visto que as instruções possuem tamanho variado: elas podem ter 0, 1 ou dois operandos imediatos.

3. **OF** - *Operand Fetch* ou Busca de operandos: Neste estágio é realizada a busca dos valores dos operandos em um banco de registradores de tamanho variável, definido a *priori* nos primeiros estágios do projeto do ASIP. A pilha de operandos e o *pool* de variáveis locais dos métodos estão disponíveis no banco de registradores. Além disso, existem dois registradores reservados: SP e VARS. Eles apontam para o topo da pilha de operandos e para o início do *pool* de variáveis locais respectivamente. Dependendo da instrução, um deles é usado como base para a busca de operandos. Uma vez que os operandos são buscados, eles são enviados ao quarto estágio, onde serão usados pela instrução na devida unidade funcional.

4. **EX** - *Execution* ou Execução: Compreende 5 unidades funcionais a saber: 1 unidade de cálculo de endereços de memória (*load/store*), 1 unidade de multiplicação de inteiros, 1 unidade lógica e aritmética de inteiros, 1 unidade de desvios e 1 unidade de deslocamento. Operações de ponto flutuante são executadas por *software* através de bibliotecas aritméticas, pois o processador FemtoJava não possui unidades de ponto flutuante visto que destina-se à sistemas embarcados. Não existe predição dinâmica de desvios, portanto todos os

[4]O processador FemtoJava não possui *memória cache*.

desvios condicionais são previstos como não realizados. Se é constatado neste estágio que o desvio deveria ter sido realizado, todas as entradas da fila de instruções são removidas e as instruções nos estágios anteriores desconsideradas, resultando em uma penalidade de 3 ciclos de *clock*.

5. **WB** - *Write Back* ou Escrita de resultados: Este estágio armazena no banco de registradores, se necessário, o resultado do estágio de execução usando o registrador SP ou o registrador VARS como base. O banco de registradores não pode ser simultaneamente lido e escrito, logo, uma bolha é inserida no *Pipe* caso uma instrução no quinto estágio tente escrever enquanto outra no terceiro espera ler.

FemtoJava *Low Power* explora o paralelismo temporal no nível de instruções através da técnica de *pipeline*, o que lhe garante um melhor uso de seus recursos de *hardware*. Pois sem a execução em *pipeline*, partes de seu *datapath* não seriam utilizadas em um certo ciclo de *clock*. Entretanto, problemas de dependências verdadeiras (*RAW - Read After Write*) são encontrados quando uma instrução no estágio EX possui o resultado necessário à uma instrução seguinte no estágio OF. FemtoJava trata esse problema através da técnica de *forwarding*, passando diretamente o resultado do estágio EX para o estágio OF. Em processadores baseados em pilha, o uso de tal técnica é mais vantajoso do que em arquiteturas *load-store*, visto que em instruções que manipulam a pilha de operandos, o operando recebido por *forwarding* não será mais necessário. Como consequência, uma vez que o valor foi passado por *forwarding* à instrução dependente, não existe a necessidade da instrução que produziu tal operando escrevê-lo na pilha. A instrução *dup* que duplica o valor do topo da pilha de operandos é uma exceção, pois mesmo que seu operando seja obtido por *forwarding*, a instrução que o produz deve armazená-lo na pilha, visto que a instrução *dup* não destrói o operando por ela utilizado.

Dois tipos de *forwarding* podem ocorrer: quando a instrução no estágio OF

precisa buscar 1 operando do topo da pilha (como *istore* que armazena o topo da pilha em algum lugar do *pool* de variáveis locais); ou quando a instrução necessita de dois operandos da pilha (como operações aritméticas: *iadd, isub, ior*). No primeiro caso, o operando é obtido do estágio EX. No segundo caso, o segundo operando é obtido do estágio WB.

A técnica de *forwarding* também é válida para escritas no *pool* de variáveis locais. Ou seja, se uma instrução no estágio EX produz o valor de uma variável do *pool* e outra no estágio de OF necessita ler a mesma variável, o operando da última é obtido por *forwarding* do estágio EX. Do mesmo modo, se uma instrução em WB irá escrever (no próximo ciclo) em uma variável do *pool* e uma instrução em OF necessita obter o valor dessa mesma variável, tal valor é obtido por *forwarding* do estágio WB.

Devido à técnica de *forwarding* houve uma melhor utilização das unidades funcionais por causa de redução das paradas do *pipeline* que ocorrem em consequência de dependências verdadeiras, além de um redução significativa do número de escritas no banco de registradores. Logo, esta implementação do FemtoJava alcançou redução no consumo de potência por ciclo além de melhor desempenho quando comparada à versão multiciclo. O consumo de potência total gasto na execução de uma aplicação também foi reduzido, visto que o número de ciclos total é menor na execução em *pipeline* [5].

## 2.2 Simulador CACO-PS

A fim de validar a técnica JDTM, este trabalho fez uso do simulador CACO-PS [7, 5] que pode ser instrumentado para fornecer informações quantitativas a respeito da descrição de uma arquitetura alvo. Neste trabalho, foi simulada a arquitetura do processador FemtoJava *Low Power* 32 *bits*, com e sem o mecanismo JDTM.

CACO-PS é um simulador configurável de código compilado, que estima a potência de uma arquitetura baseado em ciclos de *clock*. Apesar de ser um simulador de código compilado, ele oferece a possibilidade da descrição estrutural de qualquer arquitetura em diferentes níveis de abstração.

Basicamente, três arquivos são necessários para uma descrição completa da arquitetura:

1. Descrição da arquitetura em uma sintaxe definida pelo simulador;
2. Descrição em linguagem C dos componentes funcionais;
3. Descrição em linguagem C de um modelo de cálculo de potência para cada componente funcional instanciado na descrição da arquitetura.

A arquitetura é descrita por um conjunto de declarações de componentes, onde cada um contém sinais de entrada, comportamento funcional e sinais de saída. O componente recebe os sinais de entrada, executa seu comportamento funcional sobre esses sinais e retorna o resultado nos sinais de saída.

Os componentes podem ter qualquer granularidade, como por exemplo: multiplexadores, portas lógicas, ULA (Unidade Lógica e Aritmética), transistores, memórias ou até mesmo processadores inteiros. O nível de abstração é definido pelo projetista da arquitetura. Os componentes podem ter seu comportamento funcional e modelo de potência descritos em linguagem C. Vale mencionar que, uma vez especificado o comportamento funcional e o modelo de potência de um componente, estes podem ser reusados na descrição de outra arquitetura.

### 2.2.1 Descrição da arquitetura e simulação

Uma arquitetura deve ser descrita em um arquivo texto, que possua um conjunto de linhas no seguinte formato:

```
Input1,...,InputN -> component id (X,Y,..., Z:control) -> Output1,...,OutputN
```

Onde `Input1,...,InputN` são os sinais que serão usados como entrada para o componente. Para cada componente, um identificador (`id`) é necessário. Considerando que um componente pode ser instanciado mais que uma vez, o `id` é usado para diferenciar uma instância de outra. Finalmente, o componente é controlado pelos sinais de controle `X`, `Y` e `Z`, se necessário. O resultado final retornado pelo componente é propagado para `Output1,...,OutputN`.

A Figura 2.8 exibe 3 componentes (`RegA`, `RegB` e `ULA`). O sinal de controle possui 8 *bits*. Os *bits* 0, 1 e 2 controlam o registrador `RegA`. Os *bits* 3, 4 e 5 controlam o registrador `RegB`. Finalmente, os *bits* 6 e 7 controlam a `ULA` [5].

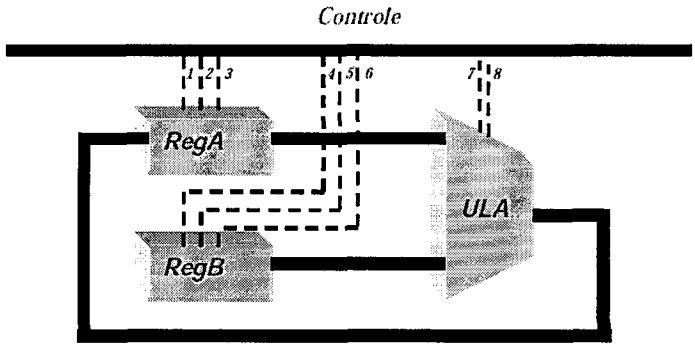



Figura 2.8: Exemplo de uma microarquitetura alvo.

Os *bits* que controlam os registradores são usados para acionar os comandos de *enable*, *set* e *reset* do registrador, respectivamente. Os *bits* de controle da `ULA` indicam as operações: 00 - soma, 01 - subtração, 10 - *and*, 11 - *or*. A Figura 2.9 exibe o exemplo da Figura 2.8 na sintaxe do CACO-PS [5].

Para o exemplo da Figura 2.8 é necessário descrever o comportamento funcional de dois componentes (Registrador e ULA). Pode-se observar na Figura 2.10 a descrição em linguagem C de um registrador de 16 *bits* onde, a cada borda de subida do *clock* (linha 5), é verificado se o sinal de controle *enable* está habilitado (linha 6).

```
regA_out,regB_out        -> ula exemplo (6,7:controle)    -> regA_in
regA_in                  -> reg A (0,1,2:controle)        -> regA_out
regB_in                  -> reg B (3,4,5:controle)        -> regB_out
```

Figura 2.9: Descrição de uma microarquitetura alvo na sintaxe do CACO-PS.

Caso positivo, a função ADICIONA_VALOR grava o valor do sinal de entrada (linha 7). De forma assíncrona, as linhas 10 e 11 verificam se os sinais de controle *set* e *reset* estão desabilitados (lógica inversa) respectivamente. A linha 12 fornece o valor do registrador no ciclo atual.

```
1.  COMPONENTE("reg") {
2.  // enable = controle[0];
3.  // set = controle[1];
4.  // reset = controle[2];

5.  if (clock_event == 1) {
6.          if (controle_atual[0] == 1) {
7.                  ADICIONA_VALOR(input_atual[0]&MASK);
8.          }
9.  }

10. if (controle[1] == 0) ADICIONA_VALOR(0xffff);
11. if (controle[2] == 0) ADICIONA_VALOR(0x0000);
12. output[0] = (PEGA_VALOR & MASK);
13. }
```

Figura 2.10: Descrição do comportamento funcional de um registrador de 16 *bits* (de Beck Filho [5]).

Quando o projetista constrói o comportamento de um componente, existe uma abstração dos sinais de entrada e saída. Quando o componente é instanciado na descrição da arquitetura, o simulador efetua a ligação dos sinais de entrada e saída da instância do componente aos sinais conectados como entrada e saída para aquele componente.

Cada componente pode possuir um modelo de cálculo de potência baseado na frequência de chaveamento do *bits* do sinal de entrada. Para uma descrição razoável desse modelo é necessário uma descrição VHDL no nível de portas do componente em questão. Entretanto, é importante mencionar que este trabalho não incorpora

resultados de estimativas de potência do JDTM.

Para executar a simulação de uma arquitetura qualquer, quatro arquivos são necessários: Descrição do comportamento funcional dos componentes, arquivo de descrição da arquitetura, e o conteúdo das memórias ROM e RAM em formato MIF (opcionais).

## 2.3 DTM - Dynamic Trace Memoization

### 2.3.1 Introdução

O mecanismo de memorização e reuso dinâmico de traços de *bytecodes* Java, que será apresentado no próximo capítulo, baseia-se no mecanismo *DTM (Dynamic Trace Memoization)*, o qual pode ser definido como uma técnica [2] que busca memorizar seqüências de instruções dinâmicas, *i.e.*, traços de instruções e realizar o reuso do conjunto de resultados produzidos por estes traços quando possível.

Este mecanismo faz uso de duas tabelas de memorização, implementadas em *hardware* e indexadas pelo valor do registrador PC (Figura 2.11): *Memo_Table_G - Tabela de Memorização Global* e *Memo_Table_T - Tabela de Memorização de Traços.* O uso de duas tabelas fornece ao mecanismo dois níveis de reuso, o primeiro nível é utilizado no reuso de instruções simples e na construção de traços de instruções, enquanto o outro é responsável pelo reuso de traços de instruções.

A tabela Memo_Table_G basicamente armazena em cada entrada: (i) parte do endereço de memória de uma instrução dinâmica; (ii) os operandos fontes e os valores a eles instanciados e (iii) o resultado produzido pela execução da instrução. A tabela Memo_Table_T basicamente armazena em cada entrada: (i) parte do endereço de memória da primeira instrução de um traço construído dinamicamente, *i.e.*, em tempo de execução; (ii) o endereço de memória da próxima instrução a ser executada

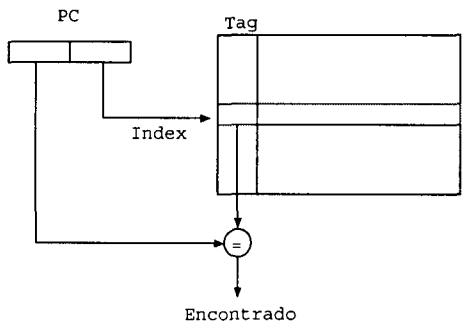



Figura 2.11: Procedimento utilizado para indexar quaisquer das tabelas de memorização.

para o caso em que o traço seja futuramente reusado; (iii) o conjunto de operandos fontes e valores a eles instanciados que foram utilizados como operandos de entrada para as instruções que compõem o traço e (iv) o conjunto de operandos de destino e seus respectivos valores produzidos pelas instruções pertencentes ao traço.

A construção dos traços que serão memorizados é realizada com base na identificação de *instruções redundantes* simples, que são instruções dinâmicas instanciadas com os mesmos valores de seus operandos de entrada observados em uma execução anterior desta instrução, o que portanto faz com que estas instruções, se executadas, produzam o mesmo resultado observado na execução anterior. A identificação de uma instrução redundante é feita por uma consulta à tabela Memo_Table_G, em busca de uma entrada desta tabela que corresponda à instância desta instrução.

Instruções cujas instâncias sejam encontradas na tabela Memo_Table_G, não serão executadas, visto que o resultado destas instruções já está disponível na tabela, bastando apenas *reusá-lo*. Para isso, faz-se necessário atualizar o estado do processador através da escrita deste resultado no banco de registradores, bem como o repasse para possíveis instruções que estão a espera deste resultado (instruções dependentes). A instância de uma instrução que não é encontrada em Memo_Table_G, será memorizada nesta tabela na esperança de que a próxima vez que tal instrução seja executada, sua nova instância corresponda à instância memorizada e conseqüen-

temente seja identificado um resultado redundante.

Os traços são construídos em um *buffer de construção de traços*: Para cada instrução redundante identificada, seus operandos de entrada e o operando de destino serão utilizados para incrementar o conjunto de operandos de entrada e de destino do traço em construção no *buffer*, que após ser finalizado, será memorizado em Memo_Table_T. A construção é finalizada quando uma instrução não redundante é identificada.

### 2.3.2 Definições

Algumas definições se fazem necessárias a fim de melhor compreendimento da rápida explanação do mecanismo que será apresentada adiante.

- O *domínio de instruções válidas* em DTM, representa um subconjunto de instruções da ISA do processador que podem ser incluídas em traços;
- Um *traço redundante* é definido como uma sequência de instruções dinâmicas e redundantes no domínio de instruções válidas;
- Considerando o conjunto de instruções do MIPS III ISA, as instruções de *acesso a memória* (load/store), instruções de *ponto flutuante*, e instruções de *chamadas a rotinas do sistema operacional*, não fazem parte do domínio de instruções válidas em DTM;
- O *contexto de entrada* de um traço é definido como o conjunto de operandos fonte e os valores a eles instanciados. Os operandos fonte são referenciados por instruções que compõem o traço e são produzidos por instruções externas ao traço;
- O *contexto de saída* de um traço é definido como o conjunto de operandos de destino e os valores a eles instanciados. Os operandos de destino são refe-

renciados por instruções que compõe o traço, ou seja, armazenam o resultado decorrente da execução da instrução que o referencia.

### 2.3.3 Memorização e identificação de instruções redundantes

Em DTM, a construção de um traço requer a identificação de redundância com granularidade no nível de instrução dinâmica. Para cada instrução buscada, decodificada, com seus operandos prontos e *pertencente* ao domínio de instruções válidas, é realizada uma pesquisa em Memo_Table_G, onde o endereço e os atuais valores dos operandos de entrada desta instrução são associativamente comparados ao campo `PC`, `sv1` e `sv2` das entradas de Memo_Table_G. Se houver alguma correspondência, tal instrução é rotulada como *redundante*, caso contrário será rotulada como *não redundante* e conseqüentemente inserida em Memo_Table_G. A Figura 2.12 exibe o formato de uma entrada de Memo_Table_G, a qual é composta pelos seguintes campos:

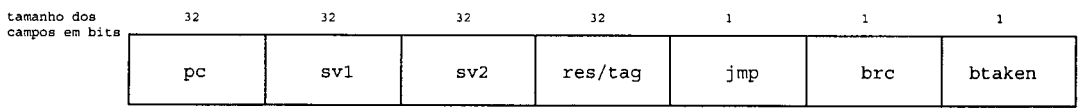



Figura 2.12: Formato de uma entrada de Memo_Table_G.

- `pc` - armazena os bits mais significativos do endereço de memória da instrução;
- `sv1` - valor do primeiro operando de entrada instanciado;
- `sv2` - valor do segundo operando de entrada instanciado;
- `res/targ` - armazena o resultado de uma instrução lógica/aritmética ou o endereço de destino de um desvio incondicional/condicional, dependendo dos campos `jmp` ou `brc`;

- `jmp` - se ativado, indica que a instrução é de desvio incondicional (incluindo chamadas de subrotinas e instruções de retorno);
- `brc` - se ativado, indica que a instrução é de desvio condicional;
- `btaken` - se `btaken` está ativado, indica que o desvio `brc` é realizado, caso contrário `brc` não é realizado.

O mecanismo DTM utiliza uma heurística, baseada na redundância de instruções simples, para a construção e memorização dos melhores traços[5]. Ou seja, quando uma instrução é executada pela primeira vez ela é inserida na tabela de memorização global, e somente as instâncias de instruções que estão presentes nesta tabela e que são reusadas irão construir *traços de instruções redundantes.*

Este tipo de implementação também fornece uma pré-qualificação das instruções (somente instruções constatadamente redundantes) que irão compor um traço, porém não é um indicador de qualidade do traço formado, pois para qualificar um traço precisa-se levar em consideração o número de instruções que estão contidas no traço e que pertencem ao caminho crítico de execução do programa, e a frequência de reuso do traço durante a execução do programa.

### 2.3.4 Construção de traços com base em instruções redundantes

A construção de um traço é realizada em tempo de execução pelo mecanismo DTM implementado em *hardware*, *i.e.*, nenhuma intervenção por parte do compilador é necessária. Tal construção utiliza uma estrutura temporária, chamada *buffer de construção de traços.* A Figura 2.13 esboça o buffer para a construção e armazenamento de informações sobre os contextos (entrada/saída) de traços, além de

[5]os possivelmente mais redundantes.

outras informações. Esta mesma estrutura representa uma entrada da tabela de memorização Memo_Table_T, que irá armazenar (memorizar) os traços construídos.

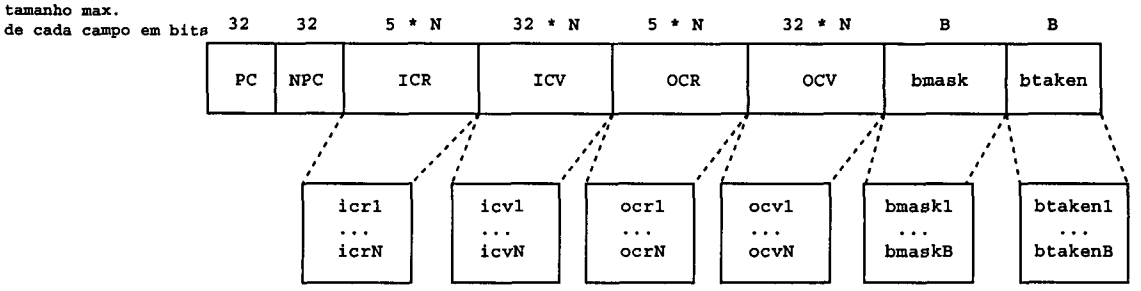



Figura 2.13: Composição do buffer, bem como das entradas de Memo_Table_T.

Os campos da estrutura esboçada na Figura 2.13, possuem a seguinte descrição:

- `PC` - armazena os bits mais significativos do endereço de memória da primeira instrução do traço;
- `NPC` - armazena o endereço da próxima instrução a ser executada para o caso em que o traço seja reusado;
- `icr1,...,icrN` - indicam os registradores que armazenam o contexto de entrada;
- `icv1,...,icvN` - armazena os valores dos registradores do contexto de entrada indicados por `icr1,...,icrN`;
- `ocr1,...,ocrN` - indicam os registradores que armazenam o contexto de saída;
- `ocv1,...,ocvN` - armazena os valores dos registradores do contexto de saída indicados por `ocr1,...,ocrN`;
- `bmask` - cada *bit* ativo deste campo, indica a presença de uma instrução de desvio no traço;

- `btaken` - para cada *bit* ativo em bmask, o *bit* correspondente neste campo, indica se o desvio será realizado ou não realizado.

A construção de um traço é iniciada ao ser identificada uma instrução rotulada como redundante. A finalização e conseqüente memorização do traço em construção é realizada ao ser identificada uma instrução rotulada como não redundante, ao ser alcançado o limite B de instruções de desvio dentro do traço ou quando o número de elementos do contexto de entrada ou de saída tenha alcançado o limite N.

No caso de uma instrução ser rotulada como redundante e já existir um traço em construção, os valores dos operandos fonte e destino desta instrução são utilizados para incrementar os contextos de entrada e de saída do traço. Para essa funcionalidade, são utilizados um mapa de *bits* de contexto de entrada e um mapa de contexto de saída, onde cada *bit* corresponde a um registrador da arquitetura e informa se o mesmo pertence ao contexto de entrada/saída.

A formação dos contextos de entrada e de saída seguem os seguintes passos:

1. um *bit* do contexto de entrada é ativado quando o correspondente registrador é um dos operandos fonte de alguma instrução redundante incluída no traço e o valor de tal registrador foi produzido por uma instrução fora do traço, ou seja, com o correspondente *bit* desativado no mapa de contexto de saída.

2. para cada *bit* ativado no contexto de entrada, o DTM adiciona o identificador e o valor do registrador correspondente ao contexto de entrada no *buffer* de construção de traços.

3. um *bit* do contexto de saída é ativado quando o correspondente registrador é o destino do resultado produzido por uma instrução redundante incluída no traço.

4. para cada *bit* ativado no contexto de saída, o DTM adiciona o identificador

e o valor do registrador correspondente ao contexto de saída no *buffer* de construção de traços.

Outras informações necessárias no momento do reuso de um traço, precisam ser memorizadas durante a construção. Estas são:

1. Parte alta do valor do endereço de memória da primeira instrução que compõe a sequência de instruções inclusas no traço, a qual não será utilizada na indexação da Memo_Table_T. Este valor, armazenado no campo PC do *buffer*, será um identificador do traço e utilizado para pré-selecionar traços candidatos à reuso.

2. O valor do PC da próxima instrução a ser executada para o caso em que o traço seja reusado. Este valor será armazenado no campo NPC do *buffer* e utilizado para redirecionar o fluxo de execução quando o traço for reusado, *i.e.*, ele será utilizado para atualizar o valor corrente do PC de forma a evitar que as instruções pertencentes ao traço sejam re-executadas. Vale mencionar, que o valor do campo NPC reflete todas as transferências de controle determinadas pelas instruções de desvio dentro do traço.

3. Obtenção do padrão de desvios através de informações obtidas das instruções de desvio inclusas no traço. Esse padrão, armazenado nos campos `bmask` e `btaken`, é utilizado para atualizar o previsor de desvios da arquitetura superescalar que incorpora o DTM.

### 2.3.5 Reuso de traços redundantes

A Figura 2.14(a) exibe dois traços redundantes e seus respectivos contextos de entrada e de saída. Estes traços foram construídos a partir de uma sequência de instruções buscadas na *cache* de instruções, em que todas as instruções que fazem parte da

sequência estão com suas respectivas instâncias armazenadas em Memo_Table_G, e portanto foram rotuladas como redundante (círculos em cor cinza). As instruções representadas por círculos em cor negra são instruções rotuladas como não-redundantes (fora do domínio de instruções válidas e/ou não presentes em Memo_Table_G).

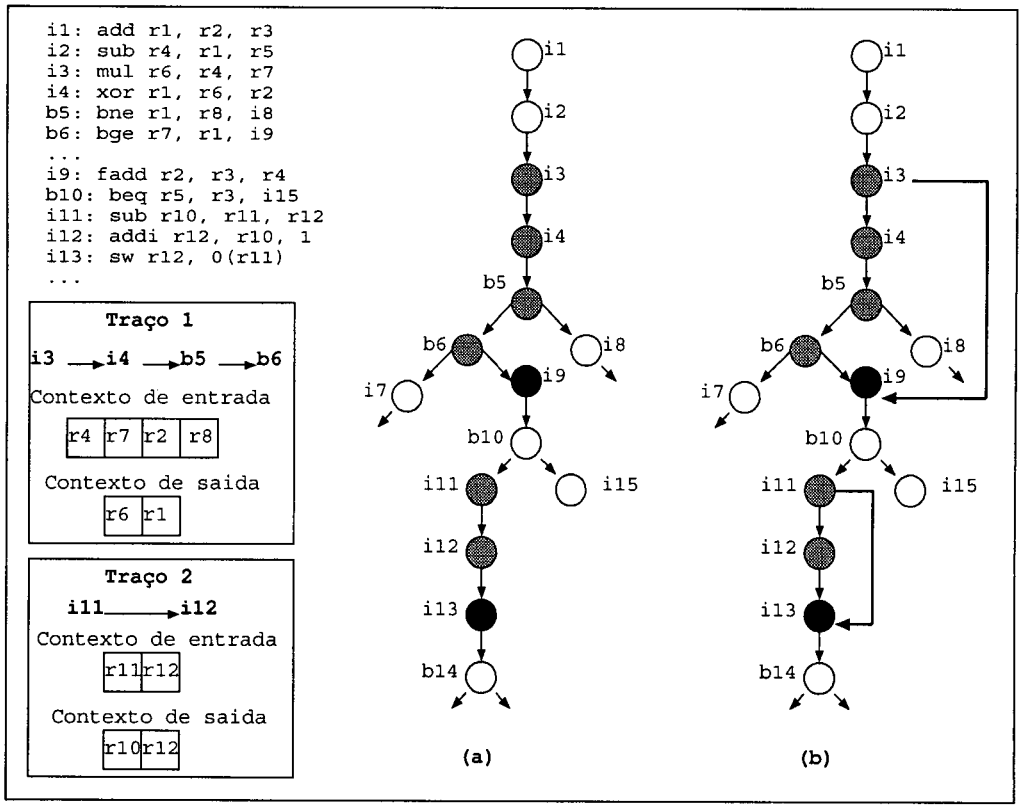



Figura 2.14: Construção e reuso de traços redundantes no DTM.

Reusar um traço equivale a reusar uma sequência de instruções dinâmicas, de modo que todas as instruções representadas no traço serão reusadas de uma só vez. A identificação de possibilidade de reuso dos traços memorizados é realizada em paralelo à identificação de instruções redundantes e conseqüente construção de traços.

A seguir será descrito os procedimentos para identificação e reuso de traços redundantes:

1. Para cada instrução buscada, é verificado se esta pertence ao domínio de instruções válidas

   (a) Se a instrução não pertencer ao domínio de instruções válidas, ela é rotulada como não redundante e executada normalmente.

   (b) Caso contrário, o valor do PC de tal instrução é utilizado para pré-selecionar traços candidatos à reuso através da comparação com o valor do campo PC de cada entrada de Memo_Table_T. Para cada traço pré-selecionado, o conjunto de valores do seu contexto de entrada (`icr1, ..., icrN`) é comparado aos valores dos respectivos registradores do banco de registradores.

      i. Caso, todo o contexto de entrada de algum traço pré-selecionado corresponda aos valores do atual estado do processador, o fluxo de execução da aplicação é redirecionado para o endereço armazenado no campo NPC do traço selecionado. Além disso, o conjunto de valores do contexto de saída é usado para atualizar o estado do processador. Finalmente as informações sobre o padrão de desvios do traço são utilizadas para atualizar o previsor de desvios.

      ii. Caso contrário, nenhum traço redundante foi identificado.

Concorrentemente ao teste de reuso de traços memorizados em Memo_Table_T, o mecanismo também utiliza o valor do PC em conjunto com os valores dos operandos de entrada da instrução como chave de busca para tentar encontrar alguma correspondência em Memo_Table_G. Se uma correspondência for encontrada, tal instrução é reusada e rotulada como redundante, podendo iniciar a construção de um traço ou incrementar um traço em construção. Porém caso o mecanismo identifique ao mesmo tempo tanto um traço quanto uma instrução redundante, o mecanismo faz preferência pelo reuso do traço e não da instrução.

A Figura 2.14(b) exibe o reuso dos dois traços construídos e memorizados em Memo_Table_T. O teste de reuso verifica se os valores dos registradores do contexto

de entrada dos traços memorizados, correspondem ao estado do processador. Se o teste for bem sucedido então os valores do contexto de saída são escritos nos registradores correspondentes, o fluxo de execução é direcionado para a próxima instrução após o traço (neste caso, a instrução `i9` no traço 1 e `i13` no traço 2). Assim, ocorre o colapso de instruções e a dependência de dados dentro do traço, bem como evita-se a execução de possíveis desvios inclusos no traço reusado (`b5` e `b6` no traço 1) que poderiam ser previstos de forma incorreta.

# Capítulo 3

# JDTM - Java Dynamic Trace Memoization

## 3.1 Introdução e definições

JDTM ou *Java Dynamic Trace Memoization* [19] é um mecanismo desenvolvido para uma arquitetura de processador Java[1], que visa memorizar traços de *bytecodes* redundantes e identificar oportunidades de reuso da computação destes traços. Este mecanismo atua na redução (i) do número total de *bytecodes* executados, (ii) das penalidades impostas por desvios realizados, (iii) das penalidades impostas pelas dependências de recursos e dependências de dados e (iv) do número de acessos à memória de instruções.

Este capítulo descreve em detalhes o mecanismo JDTM proposto neste trabalho e para tanto algumas definições são necessárias:

- ***Bytecode* redundante** - Instância de um *bytecode* estático, cujos operandos de entrada e seus valores são os mesmos observados em uma instância anterior do mesmo *bytecode* estático.

[1]Neste trabalho foi utilizado o processador FemtoJava Low Power 32 bits.

- **Traço de *bytecodes*** - Sequência de instâncias de *bytecodes* estáticos;

- **Domínio de *bytecodes* válidos** - Subconjunto do conjunto de *bytecodes* da arquitetura do processador Java que podem ser incluídos em traços redundantes;

- **Traço redundante** - Sequência de *bytecodes* redundantes no domínio de *bytecodes* válidos;

- **Contexto de entrada de um traço** - Conjunto de operandos fonte e valores a eles instanciados, os quais foram produzidos por *bytecodes* externos ao traço. O contexto de entrada pode ser classificado em dois tipos: *contexto de entrada referente ao pool de variáveis locais* e *contexto de entrada referente à pilha de operandos.*

  1. *Contexto de entrada referente ao pool de variáveis locais* - Conjunto de variáveis locais e valores a elas instanciados, que são utilizadas como operandos fontes para os *bytecodes* inclusos no traço, mas que seus valores foram produzidos por *bytecodes* externos ao traço.

  2. *Contexto de entrada referente à pilha de operandos* - Considerando a inclusão do primeiro *bytecode* ao traço, o contexto de entrada referente à pilha de operandos é o conjunto de posições da pilha de operandos (topo, subtopo, etc.) e valores a elas instanciados, os quais são desempilhados por *bytecodes* inclusos no traço, mas que foram empilhados por *bytecodes* externos ao traço.

- **Contexto de saída de um traço** - Conjunto de operandos destino e os valores a eles instanciados, os quais foram produzidos por *bytecodes* que compõem o traço. O contexto de saída pode ser classificado em dois tipos: *contexto de saída referente ao pool de variáveis locais* e *contexto de saída referente à pilha de operandos*:

1. *Contexto de saída referente ao pool de variáveis locais* - Conjunto de variáveis locais e valores a elas instanciados, os quais foram produzidos por *bytecodes* pertencentes ao traço.
2. *Contexto de saída referente à pilha de operandos* - Um único[2] valor empilhado na pilha de operandos por algum *bytecode* pertencente ao traço, mas que não foi utilizado (desempilhado) por *bytecode* incluso no traço.

## 3.2 O domínio de *bytecodes* válidos

Por motivos de economia de recursos de *hardware*, evitar efeitos colaterais e manter a simplicidade de implementação, evitando assim um adicional *overhead* no ciclo de *clock*, nem todos os *bytecodes* da ISA do FemtoJava são utilizados na construção de traços. Portanto, existe um subconjunto ou domínio de *bytecodes* que são válidos para construção. *Bytecodes* que fazem acesso à memória são desconsiderados na construção de traços pois possuem baixa frequência de execução quando comparados, por exemplo, a *bytecodes* da classe *load/store* do *pool* de variáveis locais [23]. Além disso, permitir *bytecodes* de acesso à memória nos traços reusados provocaria possíveis efeitos colaterais durante o reuso. Por exemplo: um *bytecode* de `load` será redundante caso seja instanciado com valores de operandos de entrada que foram observado em uma instância anterior, porém nada garante que o valor carregado da memória seja o mesmo da instância anterior. Portanto, para permitir a inclusão de *bytecodes* de acesso à memória nos traços construídos, seria necessário um mecanismo que garantisse a consistência entre traços memorizados e a memória de dados.

O conjunto de instruções Java consiste de 201 *opcodes* dos quais muitos podem ser sobrepostos. Por exemplo, no caso do opcode `iload`, o qual possui 1 byte como

[2]Essa é uma interessante restrição imposta pelo JDTM, que será melhor explicada na Subseção 3.4.2.2

imediato que funciona como um índice para a variável local, o valor carregado da variável precisa ser um inteiro. Porém existem também *opcodes* separados para carregar valores *long*, *double* e *float* por exemplo. Além disso, a grande maioria das aplicações alvo no ambiente de sistemas embarcados utilizam apenas valores inteiros. Por isso a ISA do FemtoJava e conseqüentemente o mecanismo JDTM cobrem apenas valores inteiros. Além dessa restrição os *bytecodes* que fazem parte do domínio de *bytecodes* válidos possuem as seguintes características:

- 5 ciclos de latência de execução, consequentemente são *bytecodes* simples que utilizam apenas 1 ciclo na fase de execução (EX).
- Obtêm os valores de seus operandos fontes através de acessos ao banco de registradores e/ou por *forwarding* dos estágios posteriores;
- Podem possuir 0, 1 ou 2 operandos de entrada, que são normalmente obtidos do *pool* de variáveis locais ou da pilha de operandos, ambas as estruturas mapeadas em um banco de 64 registradores conforme foi exibido na Figura 2.2.

A Tabela 3.1 exibe em detalhes o conjunto de 57 *bytecodes* que fazem parte do domínio de *bytecodes* válidos. Comparando a Tabela 3.1 com a Tabela 2.1 que descreve todos os *bytecodes* suportados pelo FemtoJava, pode-se observar que o JDTM faz uso de quase todos os *bytecodes* presentes na ISA, excluindo apenas a classe *Acesso à Memória*, os *bytecodes* da classe *Controle de fluxo* que fazem acesso à memória e alguns *bytecodes* da classe *Nops e Outras*.

## 3.3 Identificação de *bytecodes* redundantes

JDTM utiliza uma heurística para a construção de traços, onde *bytecodes* que foram comprovadamente redundantes são boas escolhas para compor traços de *by-*

Tabela 3.1: Domínio de *bytecodes* válidos para a construção de traços.

| Classe | *Bytecodes* |
|---|---|
| Aritméticas e lógicas | `iinc, iadd, isub, iand, ior, ixor, ineg, ishl, ishr, iushr, idiv, irem, imul` |
| Controle de fluxo | `ifeq, ifne, iflt, ifge, ifgt, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, if_acmpne, goto` |
| *Load/Store* | `iload, iload_0, iload_1, iload_2, iload_3, istore, istore_0, istore_1, istore_2, istore_3, aload, aload_0, aload_1, aload_2, aload_3` |
| Operações de Pilha | `iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, bipush, sipush, pop, pop2, dup, dup2, dup_x2` |
| Outras | `nop` |

*tecodes* a serem memorizados e possivelmente reusados durante a execução de uma aplicação alvo. Portanto, faz-se necessário identificar redundância com granularidade no nível de *bytecodes*. Estes *bytecodes* serão agrupados em seqüências e por fim memorizados em uma tabela implementada no *hardware* do processador.

Para identificar redundância no nível de *bytecodes*, ou seja, identificar *bytecodes* redundantes, é utilizada uma tabela associativa de memorização de *bytecodes* chamada *Tabela de Memorização Global* ou *J_Memo_Table_G*, que memoriza as instâncias de todos os *bytecodes* pertencentes ao domínio de *bytecodes* válidos, exceto os *bytecodes* que manipulam valores constantes.

*Bytecodes* que manipulam constantes (`iconst_1`, `bipush` etc.) possuem 1 byte imediato como operando fonte que se destina à pilha de operandos. Com exceção do *bytecode* `goto` que possui 2 bytes imediatos que formam o endereço alvo do desvio incondicional. Observe que estes *bytecodes* não precisam ser memorizados pela J_Memo_Table_G, visto que já são redundantes pelo fato de sempre produzirem o mesmo resultado.

As instâncias memorizadas são compostas: pelos bits mais significativos do endereço do *bytecode* e os valores de seus operandos de entrada. Os campos que compõem

cada entrada de J_Memo_Table_G são descritos abaixo:

- `tag` - Armazena os bits mais significativos do endereço de memória de um *bytecode*, os quais não são utilizados na indexação da tabela;
- `opV1` - Valor do primeiro operando de entrada;
- `opV2` - Valor do segundo operando de entrada, se existir.

Sequências de *bytecodes* podem ser agrupadas em traços redundantes, desde que todos os *bytecodes* das sequências façam parte do domínio de *bytecodes* válidos e sejam *bytecodes* redundantes. Os passos para a identificação de *bytecodes* redundantes são descritos abaixo:

1. Para cada *bytecode* acessado e decodificado, o JDTM verifica se este é um *bytecode* pertencente ao domínio de *bytecodes* válidos. Se o *bytecode* não pertencer, ele é rotulado como *não redundante*;
2. Para um dado *bytecode* que pertença ao domínio de *bytecode* válidos e não é um *bytecode* constante, o JDTM na fase de busca de operandos (OF) irá tentar encontrar 1 instância memorizada em J_Memo_Table_G que corresponda exatamente à instância do *bytecode* em questão. Para isso o JDTM utiliza os bits menos significativos do endereço de memória do *bytecode* para indexar um conjunto de entradas na tabela associativa[3]. Para cada entrada do conjunto, o mecanismo verifica se os bits mais significativos do endereço de memória corresponde ao valor armazenado no campo `tag` e se os valores dos campos `opV1` e `opV2` correspondem ao valor do primeiro operando e ao valor do segundo operando do *bytecode* respectivamente.

[3]Somente com a associatividade maior ou igual à *2-way*, é possível manter memorizada mais de 1 instância do mesmo *bytecode* estático.

(a) Caso não exista em J_Memo_Table_G uma instância idêntica à instância do *bytecode* corrente, então o JDTM aloca uma entrada em Memo_Table_G e armazena as informações obtidas do *bytecode* nos campos da entrada alocada. Além de rotular o *bytecode* como não redundante;

(b) Caso seja encontrada em J_Memo_Table_G uma instância que corresponda à instância do *bytecode* corrente, então o *bytecode* é rotulado como *redundante* e não é inserido na tabela;

3. Caso a instância do *bytecode* manipule apenas valores constantes como operandos de entrada, o *bytecode* é rotulado como redundante e a consulta em J_Memo_Table_G é desnecessária, contribuindo assim para economia no uso de memória e redução de potência dissipada decorrente de acessos à tabela.

A Figura 3.1 apresenta uma entrada genérica da tabela J_Memo_Table_G com os campos `tag`, `opV1`, `opV2` e a quantidade de bits necessários para cada campo. De forma resumida, para cada instância de *bytecode* pertencente ao domínio de *bytecodes* válidos, o mecanismo tenta indexar uma ou mais entradas da tabela utilizando os bits menos significativos do endereço de memória da instância. Para cada entrada indexada, é realizada uma comparação entre os valores dos operandos de entrada da instância, e os valores memorizados nos campos `opV1` e `opV2`. O resultado desta comparação provoca a rotulação da instância em redundante ou não redundante dependendo se a comparação foi respectivamente bem ou mal sucedida.

Os traços construídos a partir de *bytecodes* redundantes serão delimitados por *bytecodes* não redundantes ou não pertencentes ao domínio de *bytecodes* válidos. A Figura 3.2 esboça o procedimento de rotulação de instâncias de *bytecodes* de acordo com sua identificação em J_Memo_Table_G, e delimita uma sequência de instâncias que determinam um *traço redundante*. Nesta figura, o fluxo de execução é representado pelo grafo e os vértices não pontilhados correspondem ao caminho de execução. Os vértices com preenchimento hachurado representam uma sequência

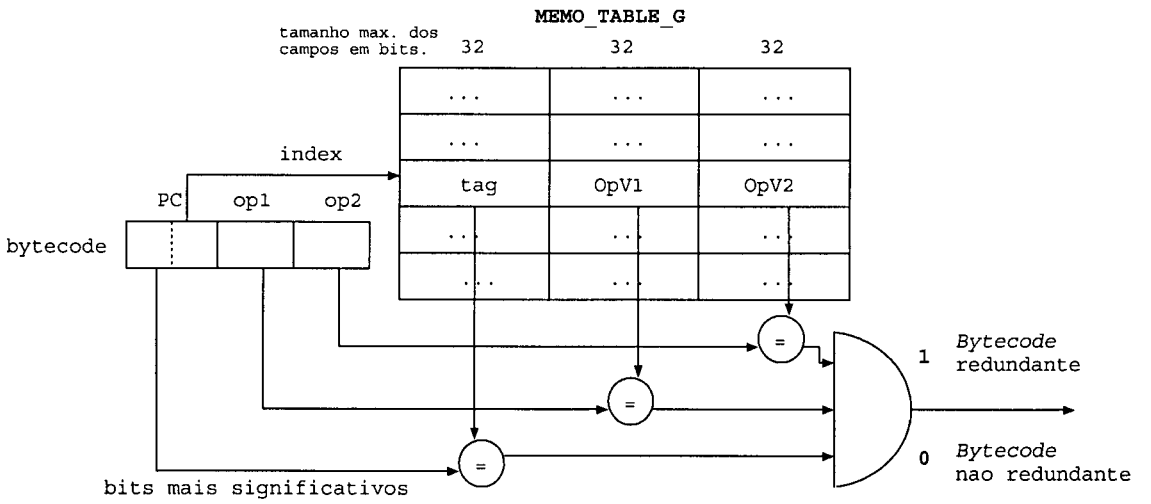



Figura 3.1: Tabela de memorização global (J_Memo_Table_G), utilizada para guardar instâncias de *bytecodes*.

de execução de *bytecodes* redundantes, *i.e.*, presentes em J_Memo_Table_G. Esta sequência é utilizada para formar um traço redundante que será memorizado em outra tabela denominada Tabela de Memorização de Traços ou J_Memo_Table_T.

## 3.4 Construção de traços redundantes

Como mencionado na seção anterior, a construção de traços redundantes é baseada na redundância com granularidade no nível de *bytecodes*. Esta seção visa descrever de forma incremental os passos para a construção destes traços.

Uma estrutura auxiliar denominada *buffer de construção* e esboçada na Figura 3.3, é utilizada para armazenar as informações sobre os contextos de entrada e de saída do traço em construção além de outras informações. Esta mesma estrutura representa uma entrada da tabela de memorização de traços J_Memo_Table_T.

Os campos da Figura 3.3 possuem a seguinte descrição:

- Campo PC - armazena os bits mais significativos do endereço de memória do

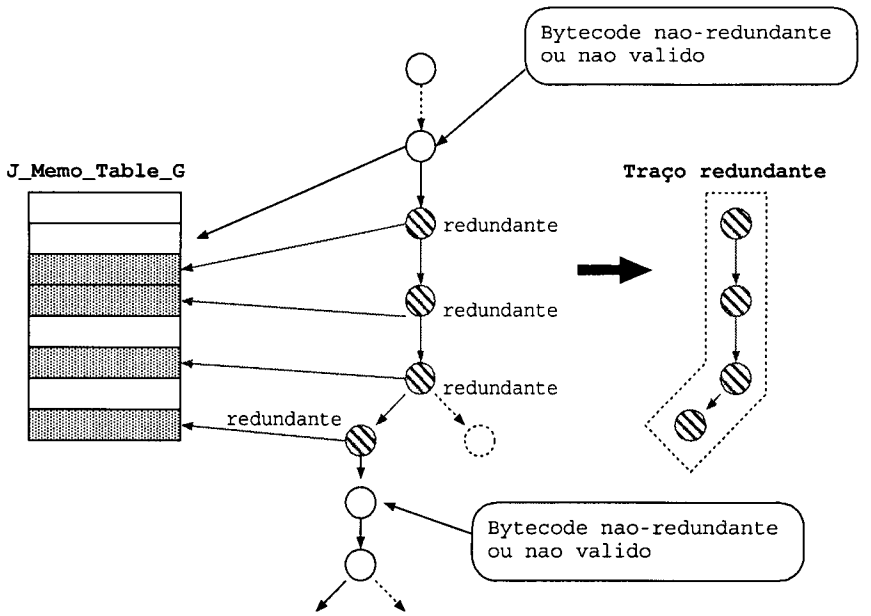



Figura 3.2: Identificando uma sequência de *bytecodes* redundantes.

primeiro *bytecode* incluso no traço;

- Campo NPC (*New PC*) - armazena o valor do endereço de memória do próximo *bytecode* a ser executado para o caso do traço ser reusado;

- Campo ICR (*Input Context Register*)- indica o conjunto de operandos que fazem parte do contexto de entrada do traço. Este campo é dividido em dois

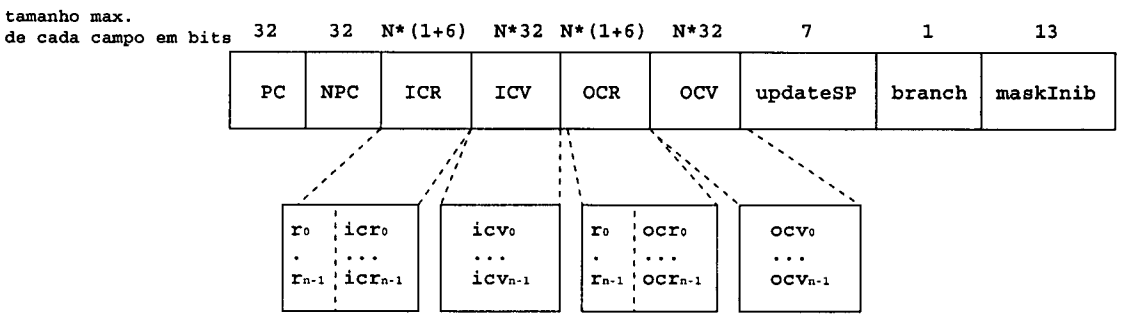



Figura 3.3: Formato de uma entrada da tabela J_Memo_Table_T e do *buffer* de construção.

subcampos: $r_i$ (referência) e $icr_i$ (índice) com $0 \leq i < N$, sendo $N$ o número máximo permitido de operandos de entrada no contexto de entrada;

1. Subcampo $r_i$ - indica se o operando $i$ é referente ao *pool* de variáveis locais ou se é referente à pilha de operandos, sendo ativado ou desativado respectivamente;
2. Subcampo $icr_i$ - se o operando $i$ é referente ao *pool* (indicado pela ativação do subcampo $r_i$) armazena o índice $j$ da variável local $j$ com $0 \leq j < 64$. Caso contrário, armazena a posição (valor 0 se estiver no topo, 1 se estiver no subtopo etc.) do operando na pilha de operandos considerando o estado da pilha imediatamente antes da construção do traço.

- Campo ICV - armazena os valores instanciados ao contexto de entrada indicados pelo campo ICR;
- Campo OCR - indica o conjunto de operandos que fazem parte do contexto de saída do traço. Este campo é dividido em dois subcampos: $r_i$ (referência) e $icr_i$ (índice) com $0 \leq i < N$, sendo $N$ o número máximo permitido de operandos de saída no contexto de saída;

1. Subcampo $r_i$ - indica se o operando $i$ é referente ao *pool* de variáveis locais ou se é referente à pilha de operandos, sendo este ativado ou desativado respectivamente;
2. Subcampo $icr_i$ - se o operando $i$ é referente ao *pool* (indicado pela ativação do subcampo $r_i$) armazena o índice $j$ da variável local $j$ com $0 \leq j < 64$. Caso contrário, armazena o valor 0 indicado que o valor deste operando deve ser escrito no topo da pilha no caso de um reuso. Vale mencionar que em cada traço, só poderá haver um único valor pertencente ao *contexto de saída referente à pilha de operandos.*

- Campo OCV - armazena os valores instanciados ao contexto de saída indicados pelo campo OCR;

- Campo updateSP - indica a redução ou aumento da pilha de operandos considerando a execução dos *bytecodes* inclusos no traço. No caso de um reuso do traço, o valor deste campo deve ser somado ao valor do registrador SP a fim de refletir o estado correto da pilha.

- Campo branch - indica a existência de algum desvio realizado dentro do traço. Esta informação é útil para decidir se a fila de *prefetch* do FemtoJava deve ser esvaziada ou apenas deslocada no caso de um reuso;

- Campo maskInib - armazena uma máscara que informa o quanto a fila de *prefetch* precisa ser deslocada no caso do reuso de um traço que não possui desvio realizado, ou seja, cujo campo branch está desativado.

### 3.4.1 Início e fim da construção dos traços

No 3° estágio do pipeline (Estágio OF) todos os *bytecodes* são rotulados como redundante ou não redundante. Portanto é neste estágio que se dá o início e o fim da construção de cada traço redundante.

Caso seja detectado um *bytecode* rotulado como redundante e caso o *buffer* de construção esteja vazio, a construção do traço é iniciada. Neste caso o endereço de memória do *bytecode* é utilizado para preencher o campo PC do *buffer*, e os operandos de entrada assim como os valores a eles instanciados são utilizados para iniciar a construção do contexto de entrada do traço. A construção do contexto de saída é iniciada utilizando o operando de destino do *bytecode* assim como o valor a ele instanciado, exceto se o operando de destino for a pilha de operandos, que neste caso mantém o contexto de saída vazio, visto que o valor produzido pode ainda ser utilizado e conseqüentemente destruído pelo próximo *bytecode* da sequência de

execução. Exemplo: `iload_0` que carrega o valor da variável 0 e o armazena no topo da pilha, `ifeq` que desempilha o valor do topo. Neste caso o valor produzido foi consumido pelo próprio traço e portanto não necessita ser incluído no contexto de saída.

Caso seja detectado um *bytecode* rotulado como não redundante ou o número máximo $N$ de operandos no contexto de entrada/saída seja atingindo e caso o traço em construção possua pelo menos 3 *bytecodes*, a construção é finalizada e o traço é memorizado na entrada de J_Memo_Table_T indexada pelo campo PC do *buffer*. O número mínimo de 3 *bytecodes* é necessário devido ao fato de que um traço menor não forneceria vantagens em seu reuso visto que o processador substrato possui uma arquitetura *pipeline* escalar (5 estágios), onde cada estágio do *pipeline* possui a latência de apenas 1 ciclo. Isto é, o reuso de traços com apenas 2 *bytecodes* implicaria em desconsiderar para execução os *bytecodes* que estivessem nos estágios ID e OF devido ao redirecionamento do fluxo de execução. Portanto, neste caso, o ganho de 2 ciclos seria perdido. O reuso de traços com apenas 1 *bytecode*, *i.e.*, reuso no nível de *bytecodes* individuais, também não ofereceria vantagem visto que o *bytecode* presente no estágio OF precisaria ser desconsiderado e portanto o ganho de 1 ciclo também seria perdido.

### 3.4.2 Construção dos contextos de entrada e de saída

Basicamente, a construção de traços se dá com a inclusão de valores aos seus contextos de entrada e de saída com base em informações fornecidas por uma sequência de *bytecodes* rotulados como redundantes.

#### 3.4.2.1 Contexto de entrada/saída referente ao pool de variáveis locais

A fim de auxiliar a construção dos contextos *referentes ao pool de variáveis locais*, são utilizadas duas estruturas auxiliares:

1. **Mapa de bits do contexto de entrada:** um registrador de 64 bits, onde cada bit corresponde a uma variável local (o bit $i$ corresponde à variável local $i$). Os procedimentos para a construção do contexto de entrada referente ao *pool* seguem os seguintes passos:

   (a) o bit $i$ é ativado se existe operando fonte no *bytecode* redundante a ser incluído no traço, o qual é a respectiva variável local $i$ que possui o valor a ela instanciado produzido por um *bytecode* externo ao traço, ou seja, com o respectivo bit não ativado no *mapa de bits do contexto de saída.*

   (b) para cada bit ativado no mapa de contexto de entrada, a respectiva variável local é incluída no campo ICR do *buffer*, ativando o subcampo $r$ e atribuindo o índice $i$ da variável ao subcampo *icr*, além de incluir o valor a ela instanciado no campo ICV.

2. **Mapa de bits do contexto de saída:** um registrador de 64 bits, onde cada bit corresponde a uma variável local (o bit $i$ corresponde à variável local $i$). Os procedimentos para a construção do contexto de saída referente ao *pool* seguem os seguintes passos:

   (a) o bit $i$ é ativado se o operando destino de um *bytecode* redundante a ser incluso no traço é a respectiva variável local $i$;

   (b) para cada bit ativado no mapa de contexto de saída, a respectiva variável local é incluída no campo OCR do *buffer*, ativando o subcampo $r$ e atribuindo o índice $i$ da variável ao subcampo *ocr*, além de incluir no campo OCV do *buffer*, o valor a ela instanciado.

#### 3.4.2.2 Contexto de entrada/saída referente à pilha de operandos

Sempre que um *bytecode* a ser incluso no traço precisa desempilhar um valor da pilha de operandos, o qual foi empilhado por um *bytecode* externo ao traço,

*i.e.*, não foi produzido pelo próprio traço, tal valor deve ser incluído no *contexto de entrada referente à pilha de operandos* assim como um identificador deste valor. Da mesma forma, ao final da construção do traço, se houver algum valor empilhado, *i.e.*, produzido pelo próprio traço, mas que não foi utilizado pelo traço, tal valor deve ser incluído no *contexto de saída referente à pilha de operandos.*

A fim de auxiliar a construção dos contextos *referentes à pilha de operandos* é utilizado um registrador de 6 bits denominado *contador de pilha local (CPL)* que monitora todos os valores empilhados e desempilhados pelos *bytecodes* inclusos no traço. Tal contador é incrementado/decrementado sempre que um *bytecode* a ser incluso no traço empilha/desempilha um valor na pilha de operandos.

Desta forma é possível saber, a cada inclusão de *bytecode* no traço, quando um ou mais valores pertencentes à pilha de operandos devem ser incluídos no contexto de entrada referente à pilha, bastando para isso verificar se o contador é menor que zero. Exemplo: registrador CPL = -2, informa que o topo e o subtopo da pilha devem fazer parte do contexto de entrada. Logo estes valores devem ser incluídos no campo ICV assim como seus identificadores *id* que serão numerados de 0 à $N-1$, os quais informam a ordem de leitura destes valores (0 = topo, 1 = subtopo etc.). Os *id's* devem ser incluídos no campo ICR (atribuindo o valor 0 ao subcampo $r$ e o valor *id* ao subcampo *icr*). Após a inclusão dos operandos fonte no contexto de entrada referente à pilha, o CPL é zerado.

Da mesma forma, ao final da construção de um traço é possível saber se existe algum valor empilhado pelo traço, o qual não foi desempilhado pelo próprio traço, bastando verificar se o contador é maior que zero.

Vale mencionar que é permitido somente um único valor no contexto de saída referente à pilha. Tal restrição foi imposta com o objetivo de evitar portas adicionais de leitura no banco de registradores, visto que o contexto de saída referente à pilha pode ser definido somente após a finalização do traço e portanto se faria necessário

obter todos os possíveis valores a partir da pilha de operandos no mesmo ciclo de clock. Assim, traços que, ao serem finalizados, possuam mais do que 1 valor no contexto de saída referente à pilha, serão desconsiderados para memorização. Poderia-se pensar em simplesmente finalizar os traços neste ponto, porém, para isto seria necessário possivelmente remover alguns elementos dos contextos de entrada e/ou de saída que foram inseridos após o momento em que contexto de saída referente à pilha não mais será menor do que 2, resultando em uma complexidade adicional ao JDTM.

### 3.4.3 Informações extras

Além das informações já mencionadas é necessário armazenar outras informações que serão úteis ao se reusar um traço redundante:

- *Informação global sobre o crescimento/decrescimento da pilha de operandos:* Para tanto é utilizado um registrador de 7 bits (-63 à +63) denominado *contador de pilha global (CPG)* que assim como o registrador CPL, monitora os valores empilhados/desempilhados na/da pilha de operandos. O contador CPG é incrementado/decrementado sempre que um *bytecode*, a ser incluso no traço, empilha/desempilha um valor na/da pilha de operandos respectivamente. Porém, este contador será útil apenas na finalização do traço, cujo valor é atribuído ao campo `updateSP` do *buffer* de construção;

- *Informação sobre a existência de desvio realizado incluso no traço:* O processador FemtoJava possui uma fila de *prefetch* que é utilizada para antecipar a busca de *bytecodes* da memória de instruções. Portanto, ao ser reusado um traço redundante que possua um desvio realizado é necessário efetuar um *flush* na fila de *prefetch* e redirecionar a busca de *bytecodes* para o endereço indicado pelo campo NPC. Tal informação é armazenada no campo *branch* do *buffer* ao ser detectado o primeiro *bytecode* de desvio realizado.

- *Informação sobre o número de bytecodes inclusos no traço:* Esta informação, que é armazenada no campo `maskInib` do *buffer*, é útil para evitar o *flush* da fila de *prefetch* caso o traço não possua desvio realizado. O *flush* pode ser evitado, simplesmente deslocando os *bytecodes* da fila pelo número de *bytecode* inclusos no traço.

## 3.5 Um exemplo de construção de traço redundante

Um exemplo de construção de traço redundante é esboçado na Figura 3.4, o qual foi identificado a partir da sequência de 5 *bytecodes* redundantes da Figura 3.4(a). A Figura 3.4(b) exibe o estado da pilha (pilha de operandos e *pool* de variáveis locais) imediatamente antes da execução do primeiro *bytecode* do traço. As Figuras 3.4(c) e 3.4(d) exibem como os mapas de bits mudam dinamicamente durante a construção de um traço, e como os contadores CPL e CPG são incrementados/decrementados durante a construção. O *buffer* com as informações do traço construído, as quais serão memorizadas em uma entrada alocada na tabela J_Memo_Table_T, é esboçado na Figura 3.4(e).

Os detalhes da construção de acordo com a inclusão de cada *bytecode* da sequência são descritos:

- O *bytecode* `0x100` realiza um desvio para o endereço de memória `0x14A` (`0x100` + `0x02` + `0x0048`), caso os valores do topo e subtopo da pilha sejam iguais. Visto que tal *bytecode* consome 2 valores da pilha, o registrador CPG será decrementado em 2 unidades. O registrador CPL (com valor igual à zero), informa que o traço não produziu tais valores e portanto estes devem fazer parte do contexto de entrada, fazendo com que os subcampos $icr_0$ e $icr_1$ recebam os valores 0 e 1 respectivamente e os subcampos $r_0$ e $r_1$ permaneçam desativados

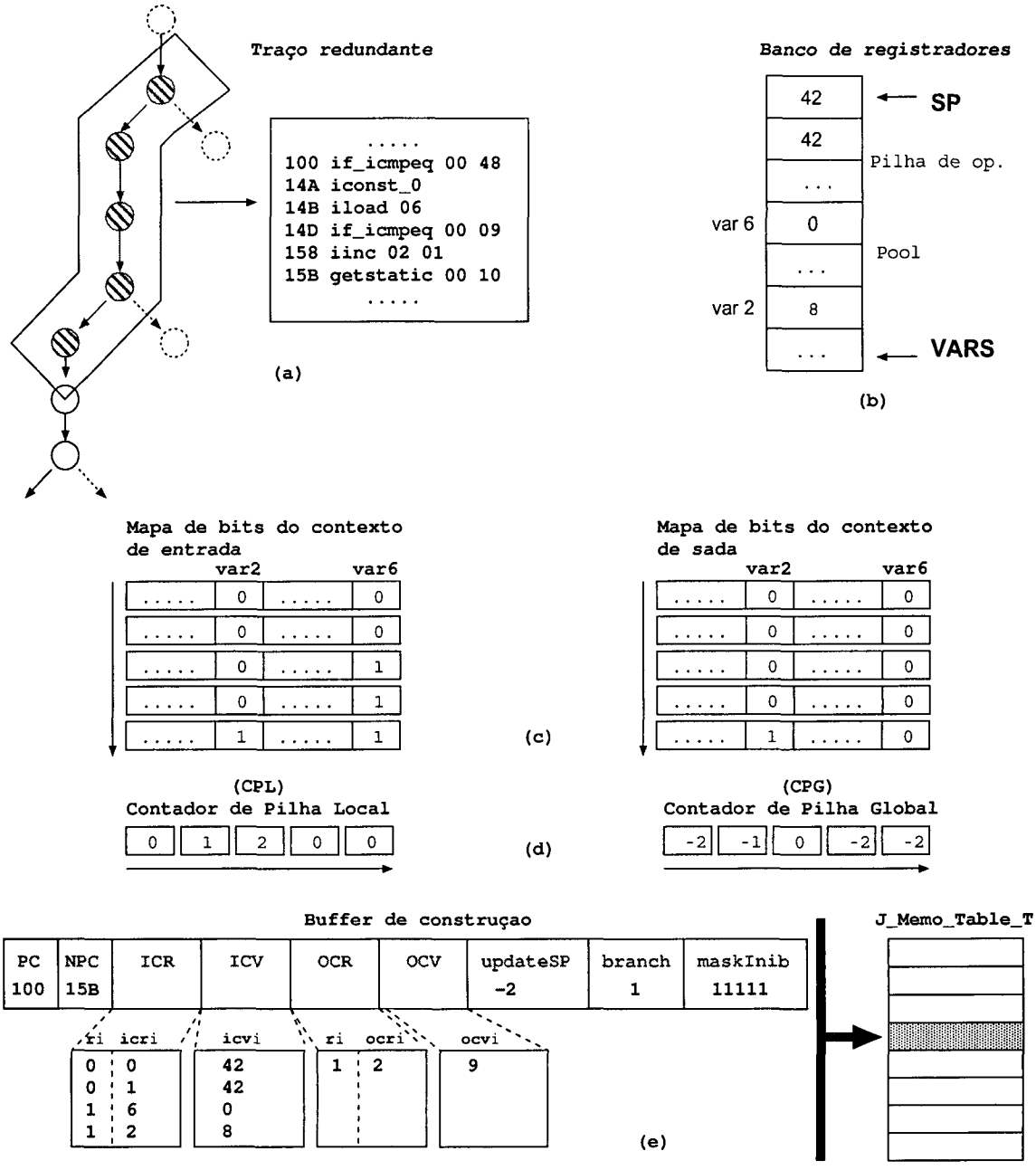



Figura 3.4: Construção de um traço redundante identificado a partir de uma sequência de 5 *bytecodes* redundantes.

indicando que estes contextos de entrada são referente à pilha do operandos. Os campos $icv_0$ e $icv_1$ devem receber os valores 42 e 42 que equivalem ao valor do topo e subtopo da pilha. A execução deste *bytecode* altera a fluxo sequencial do programa e portanto o campo `branch` do *buffer* deve ser ativado. Visto que este *bytecode* não manipula valores do *pool* de variáveis locais, nenhum bit será ativado nos mapas de bits do contexto de entrada/saída. Além disso, como este é o primeiro *bytecode* redundante da sequência (assumindo que o *buffer* de construção estava vazio), o endereço de memória de tal *bytecode* é completamente[4] armazenado no campo PC do *buffer*.

- O *bytecode* `0x14A` empilha a constante 0, fazendo com que ambos os contadores `CPL` e `CPG` sejam incrementados em 1 unidade.

- O *bytecode* `0x14B` carrega o valor da variável local 6 e o empilha. Conseqüentemente o bit 06 do mapa bits do contexto de entrada deve ser ativado e os contadores `CPL` e `CPG` incrementados em 1 unidade. Os subcampos $r_2$ e $icr_2$ devem recebem os valores 1 e 6, indicando que a variável local 6 faz parte do contexto de entrada referente ao *pool*. Por fim o campo $icv_2$ deve receber o valor instanciado à esta variável, que neste caso é 0.

- O *bytecode* `0x14D` desempilha os dois últimos valores empilhados respectivamente pelos *bytecodes* `0x14A` e `0x14B`. Os registradores `CPL` e `CPG` serão decrementados em 2 unidades. Porém neste caso o registrador `CPL` não indicará a necessidade de armazenar estes valores no contexto de entrada referente à pilha, pois os valores desempilhados foram produzidos por *bytecodes* pertencentes ao próprio traço em construção.

- O *bytecode* `0x158`, incrementa a variável local 2 em 1 unidade. Este é um caso especial de *bytecode* que acessa o *pool* de variáveis locais, visto que ele lê e escreve o valor de uma variável local. Ao consultar os Mapas de bits do

[4]considerando que a tabela J_Memo_Table_T é *full-way*

contexto de entrada e de saída, observa-se que o valor da variável local 02 não faz parte do contexto de entrada e não foi produzido por *bytecode* pertencente ao traço. Logo o bit 02 de ambos os mapas são ativados e o valor da variável local (valor 8 neste caso) será utilizado para atualizar o contexto de entrada (subcampos $r_3$ e $icr_3$ e campo $icv_3$), assim como o valor produzido $(8+1)$ será utilizado para atualizar o contexto de saída (campos $ocr_0$ e $ocv_0$).

- O *bytecode* `0x15B` carrega o valor de uma posição de memória e não faz parte do domínio de *bytecodes* válidos, portanto este *bytecode* finaliza a construção do traço, fazendo com que: o valor do campo `NPC` do *buffer* de construção receba o valor `0x15B`; o valor do registrador `CPG` seja transferido para o campo `updateSp` e seja alocada uma entrada, com estrutura idêntica ao *buffer* exibido na Figura 3.4(e), em J_Memo_Table_T para receber o traço construído. É importante observar que o campo `maskInib` é atualizado a cada novo *bytecode* incluído no traço. No caso deste traço equivale à máscara 11111, referente à 5 *bytecodes*.

## 3.6 Reuso de traços

Traços construídos a partir de *bytecodes* redundantes como exposto na seção anterior, possuem grandes possibilidades de reuso. Reusar um traço equivale a executar de uma só vez todos os *bytecodes* inclusos no traço. Essa seção descreve os passos necessários para a identificação de oportunidades de reuso de um traço redundante construído e memorizado em J_Memo_Table_T:

1. Para cada *bytecode* acessado é verificado se este é um *bytecode* pertencente ao domínio de *bytecodes* válidos e se é um *bytecode* inicial de algum traço previamente armazenado em J_Memo_Table_T. Para isso o valor do `PC` do *bytecode*

em questão é utilizado para indexar uma ou mais[5] entradas de J_Memo_Table_T. Para cada entrada indexada, os bits mais significativos do PC são comparados ao valor do campo PC da entrada (que possui os bits mais significativos do PC do primeiro *bytecode* do traço) e portanto uma ou mais instâncias do mesmo traço podem ser pré-selecionadas.

(a) Se nenhuma instância for pré-selecionada, o *bytecode* não representa a primeira instrução de um traço memorizado e portanto, este *bytecode* é executado normalmente.

(b) Em contrapartida, para cada traço pré-selecionado é verificada se o estado do processador representa o mesmo estado observado na última vez em que tal traço foi alcançado. Para tal, os operandos e valores a eles instânciados, presentes no contexto de entrada do traço (campos ICR e ICV), são comparados aos valores presentes no banco de registradores.

   i. Caso exista alguma correspondência entre o contexto de entrada e o estado do processador é identificado um traço redundante e então todo o contexto de saída memorizado (valores dos campos OCR e OCV) e valor do campo updateSP são utilizados para atualizar o estado do processador no estágio WB, *i.e.*, todo o contexto de saída deve ser escrito no banco de registradores em apenas 1 ciclo de *clock*). Além disso, para evitar a reexecução de todas as instruções presentes no traço, o valor do campo NPC é utilizado para atualizar o registrador PC do processador.

   ii. Caso contrário, não foi identificado um traço redundante e portanto o *bytecode* segue sua execução normal.

A Figura 3.5 esboça o reuso de um traço redundante memorizado em J_Memo_Table_T. A partir do endereço de memória do primeiro *bytecode* (0x100) são pré-selecionados

[5]Considerando uma tabela associativa por conjuntos.

dois traços, cujos contextos de entrada são confrontados com o estado atual do processador (valores do banco de registradores). Neste caso, o contexto de entrada do primeiro traço pré-selecionado coincide com o estado, logo, os valores do contexto de saída são escritos em suas respectivas posições no banco de registradores e todos os *bytecodes* inclusos no traço são desconsiderados para execução.

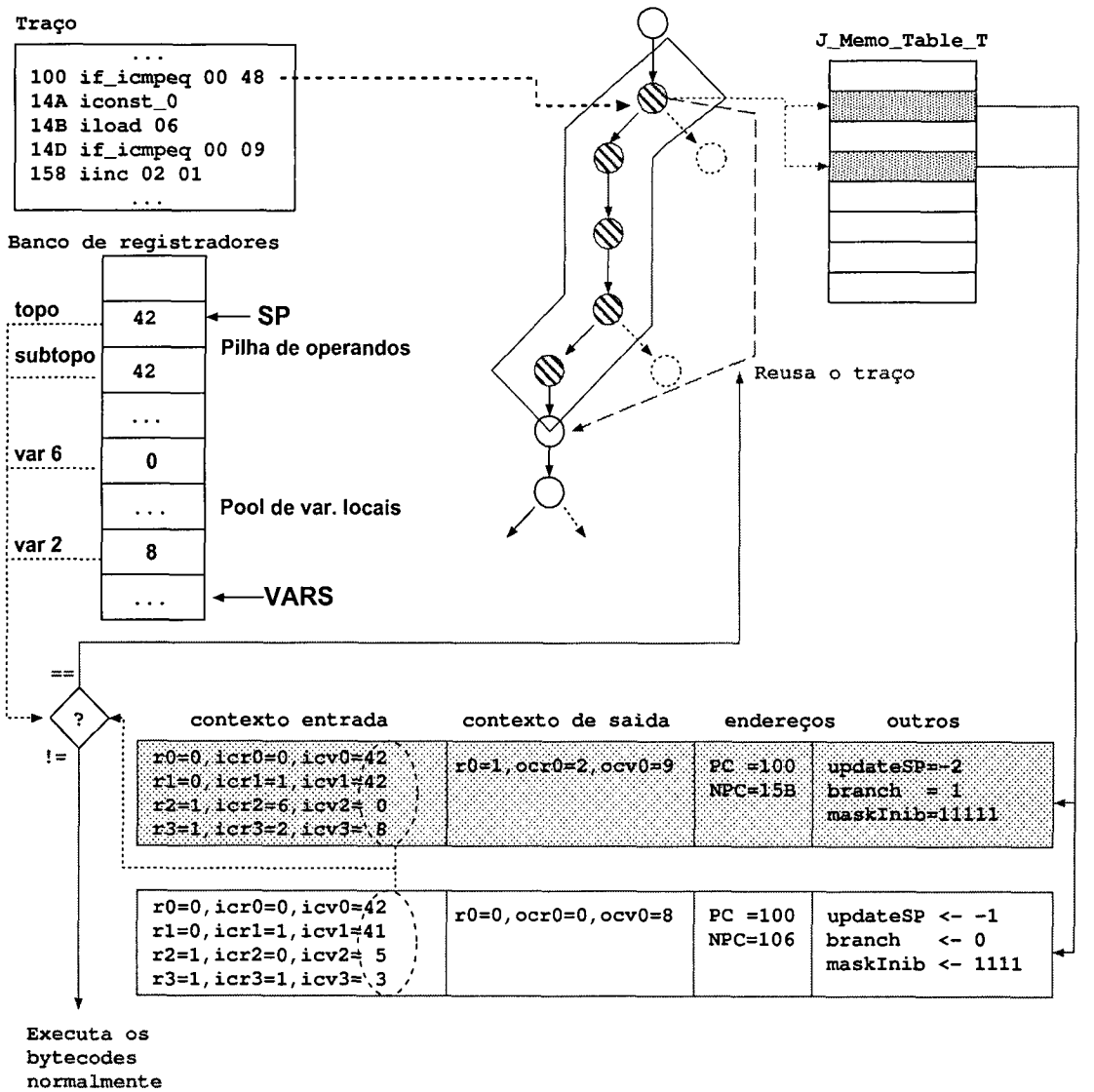



Figura 3.5: Identificação de oportunidade de reuso de um traço redundante.

# Capítulo 4

# Experimentos e Resultados

## 4.1 Metodologia

### 4.1.1 Ambiente de Simulação

A fim de avaliar o mecanismo de memorização e reuso dinâmico de traços de *bytecodes* Java proposto neste trabalho (JDTM), vários experimentos na forma de simulação foram realizados utilizando o simulador CACO-PS [7]. As simulações baseiam-se na arquitetura Java descrita por Beck Filho e Carro [6] e Gomes *et al.* [8], a qual foi modificada para incluir o mecanismo JDTM [19] proposto no Capítulo 3. Foram incluídas basicamente as tabelas de memorização e alguma lógica adicional para a construção e verificação de oportunidades de reuso dos traços. Os recursos computacionais utilizados para as simulações foram estações de trabalho IBM PC (*Cluster Mercury* do NACAD - Núcleo de Atendimento de Computação de Alto Desempenho da COPPE/UFRJ) sob o controle do sistema operacional Linux.

### 4.1.2 *Benchmarks*

Foram utilizados 8 *benchmarks* típicos de aplicações embarcadas. A Tabela 4.1 apresenta o número total de ciclos e de *bytecodes* simulados em cada *benchmark*. O número total de *bytecodes* simulados considera também os *bytecodes* nop - *no operation* impostos por dependências de dados, de controle e de recursos (no FemtoJava, o banco de registradores não pode ser lido e escrito no mesmo ciclo). Vale mencionar que a maioria dos *bytecodes* suportados pelo FemtoJava possuem 5 ciclos de latência (profundidade do *pipeline*), mas alguns *bytecodes* mais complexos tais como os de acesso à memória e os de chamada e retorno de métodos têm maior latência.

**Descrição dos benchmarks e das entradas utilizadas**

1. **CORDIC**: Cálculo das funções seno e coseno de 30 valores (45° à 74°) utilizando o método *CORDIC - Coordinate Rotation Digital Computer* [24]. Tal programa é uma biblioteca aritmética representativa.

2. **Select Sort e Bubble Sort**: Algoritmos de ordenação tipicamente utilizados em escalonadores. Arranjam um conjunto de 50 elementos únicos, aleatoriamente desordenados, colocando-os em ordem crescente.

3. **Sequential Search e Binary Search**: O algoritmo de pesquisa sequencial realiza 5 pesquisas em um *array* com 1K entradas. O algoritmo de pesquisa binária realiza 1 pesquisa em um array ordenado com 500K entradas. As localizações dos valores pesquisados foram estipuladas de forma a levar os algoritmos ao maior tempo de execução, *i.e*, ao pior caso.

4. **IMDCT - Inverse Modified Discrete Cosine Transformation**: Um algoritmo importante na descompactação de arquivos *MP3 - MPEG-1 Audio Layer 3*.

5. **Float operations**: Uma biblioteca para emular soma de números em ponto

Tabela 4.1: Programas, típicos de aplicações embarcadas, utilizados nos experimentos.

| *Benchmark* | Ciclos | Total de *Bytecodes* simulados | CPI (ciclos/bytecodes) |
|---|---|---|---|
| CORDIC | 45967 | 42285 | 1.09 |
| Select Sort | 35505 | 33893 | 1.05 |
| Bubble Sort | 62743 | 42123 | 1.49 |
| Sequential Search | 100084 | 99867 | 1.01 |
| Binary Search | 889 | 844 | 1.05 |
| IMDCT | 40316 | 28154 | 1.43 |
| Float operations | 15354 | 8399 | 1.84 |
| MP3 *player* | 112925 | 58892 | 1.92 |

flutuante, visto que a arquitetura substrato não possui unidade de ponto flutuante. Este programa faz 20 somas de dois números em ponto flutuante e armazena o resultado em um *array* na memória.

6. **MP3 player**: Um *player* de MP3 completo, uma aplicação muito representativa de sistemas embarcados.

Pela Tabela 4.1, é possível notar que os *benchmarks* Bubble Sort, IMDCT, Float Operation e MP3 *player* executam uma grande quantidade de *bytecodes* complexos, o que leva o `CPI` a um valor maior do que o observado nos demais programas.

Para fornecer informações adicionais sobre o conjunto de programas utilizado para avaliar o JDTM, a Tabela 4.2 apresenta a distribuição percentual por classe de *bytecode*: Lógica e Aritmética (operações lógicas e aritméticas de inteiros), Controle de Fluxo (desvios condicionais, incondicionais, chamada e retorno de método), Load/Store (Carga e armazenamento de variáveis locais), Operações de pilha, Acesso à memória e nops/outras (`nop` e os *bytecodes* estendidos: `sleep`, `store_idx` e `load_idx`).

Como pode ser observado na Tabela 4.2, os *benchmarks* Bubble Sort, IMDCT e MP3 *player* possuem grande quantidade de acessos à memória, através do *bytecode*

Tabela 4.2: Distribuição percentual dos *bytecodes* executados (por classe).

| *Benchmark* | Lóg./Aritm. | Contr. Fluxo | Load/Store | Op. pilha | Mem. | nops/outras |
|---|---|---|---|---|---|---|
| CORDIC | 12.27% | 6.19% | 37.89% | 2.98% | 4.76% | 35.91% |
| Select Sort | 4.19% | 7.67% | 24.58% | 4.05% | 15.33% | 44.17% |
| Bubble Sort | 9.22% | 6.17% | 21.50% | 6.20% | 20.80% | 36.12% |
| Sequential Search | 4.99% | 14.99% | 15.00% | 0.03% | 15.00% | 49.99% |
| Binary Search | 6.75% | 9.48% | 27.61% | 4.86% | 9.24% | 42.06% |
| IMDCT | 27.47% | 3.23% | 0.01% | 21.65% | 35.43% | 12.21% |
| Float operations | 11.76% | 4.43% | 25.18% | 8.58% | 6.30% | 43.74% |
| MP3 *player* | 15.48% | 2.23% | 15.34% | 17.48% | 26.27% | 23.20% |
| Média Aritmética | 11.52% | 6.80% | 20.89% | 8.23% | 16.64% | 35.93% |

`iastore` que possui 15 ciclos de latência. Além disso, foi observado que o *benchmark Float operations* executa com alta frequência, os 4 *bytecodes* de maior latência (`invokestatic` e `iastore` com 15 ciclos, `return` com 9 ciclos e `ireturn` com 10 ciclos de latência). Visto que a tabela também contabiliza as instruções `nops` (e portanto o CPI deveria ser próximo à 1 para todos os *benchmarks*) devidas às dependências de dados, controle e de recursos, estas observações justificam o elevado `CPI` de tais *benchmarks*.

Os resultados exibidos na Tabela 4.2 indicam uma grande percentagem de *bytecodes* da classe *nops/outras* na execução de todos os *benchmarks*. Isto é justificado pelo fato que uma leitura e uma escrita no banco de registrados são mutuamente exclusivas. Ou seja, sempre que um *bytecode* no 3° estágio deseja ler e um *bytecode* no 5° estágio deseja escrever no banco, um *bytecode* `nop` é inserido no 3° estágio. Portanto, faz-se extremamente necessário otimizações no nível de compilação a fim de realizar um melhor escalonamento dos *bytecodes* para evitar tais `nops`. Vale mencionar que a exclusividade de escritas sobre leituras no banco de registradores foi mantida na arquitetura substrato incorporando o JDTM, a fim de analisar o impacto do reuso de traços na redução de *bytecodes* `nop`, devido ao colapso de dependências de dados, dependências de controle e principalmente dependências de recursos.

Tabela 4.3: Configuração do processador substrato.

| | |
|---|---|
| Busca de instruções | 1 palavra (4 bytes) por ciclo, com utilização de uma fila de *prefetch*. |
| Número de estágios do pipeline | 5 estágios |
| Latência do pipeline | 5 ciclos |
| Preditor de desvios | estático (desvios são preditos como não seguidos). |
| Cache de dados | — |
| Cache de instruções | — |
| Ciclos p/ acesso à mem. de Dados | 1 ciclo |
| Ciclos p/ acesso à mem. de Instr. | 1 ciclo |
| Mecanismo de Forwarding | Encaminhamento de resultado do 5° e 4° estágio para o 3° estágio. |
| Unidades Funcionais | 1 unidade de resolução de endereços de memória, 1 unidade de multiplicação de inteiros, 1 ALU de inteiros, 1 unidade de desvio e 1 unidade de deslocamento. |
| Latência das Unid. Funcionais | 1 ciclo |
| Registradores arquiteturais | 1 pilha mapeada em 64 registradores. |

Tabela 4.4: Parâmetros arquiteturais do mecanismo JDTM.

| | |
|---|---|
| Contexto de entrada | até 10 valores |
| Contexto de saída | até 10 valores |
| Heurística utilizada na construção de traços | repetição de *bytecodes* simples. |
| Conj. de Seleção | Classes: Lógicas/Aritméticas, Ctrl. de Fluxo com exceção de chamadas e retorno de métodos, Load/Store, Op. de Pilha. |
| Política de atualização das tabelas associativas | FIFO - *First Int First Out* |

### 4.1.3 Parâmetros Arquiteturais

A Tabela 4.3 exibe os principais parâmetros da arquitetura do processador utilizado como substrato para a implementação do JDTM.

Os parâmetros arquiteturais do mecanismo JDTM são descritos na Tabela 4.4, e serão estes os parâmetros utilizados até que sejam explicitamente alterados neste texto.

### 4.1.4 Métricas

A seguintes siglas e equações determinaram os resultados obtidos nas simulações:

- *itot* - Número total de *bytecodes* executados, incluindo *bytecodes* nop;
- *cpi* - Ciclos por instrução;
- *ir* - Número total de *bytecodes* reusados pelo JDTM, incluindo *bytecodes* nop;
- *aceleracao* - Aceleração de desempenho;
- *MH* - Média Harmônica;
- *MA* - Média Aritmética;

O percentual de reuso é dado por:

$$ir/itot \tag{4.1}$$

O percentual de aceleração de desempenho é dado por:

$$aceleracao = ciclos_{base}/ciclosJDTM \tag{4.2}$$

Onde $ciclos_{base}$ e $ciclos_{JDTM}$ são considerados respectivamente para um processador não incluindo e incluindo o mecanismo JDTM. As médias são dadas pelas seguintes expressões:

$$MH = n(\sum_{i=1}^{n}(1/S_i))^{-1} \tag{4.3}$$

$$MA = (\sum_{i=1}^{n} S_i)/n \tag{4.4}$$

$S_i$ indica o elemento a ser considerado pela média.

## 4.2 Resultados

### 4.2.1 Reuso e Aceleração

Esta subseção apresenta e analisa os ganhos em percentual de reuso e aceleração obtidos com a inclusão do mecanismo JDTM na arquitetura substrato.

A primeira análise se refere ao impacto no reuso e aceleração, em vista de alterações no tamanho e na associatividade de ambas as tabelas de memorização. As Figuras 4.1 e 4.2 exibem, respectivamente, em média aritmética e média harmônica, o percentual de reuso e a aceleração obtidos através da utilização do mecanismo JDTM, com variação simultânea do tamanho, *i.e.*, do número de entradas (64 à 4K entradas) em ambas as tabelas. Para cada opção de tamanho, várias opções de associatividade foram usadas (*mapeamento direto* à *full-way*).

Pela Figura 4.1 é fácil observar que o percentual de reuso de *bytecodes* é extremamente dependente da associatividade nas tabelas com mais de 256 entradas . É possível observar também dois conjuntos distintos de percentuais de reuso: para as opções de 64 à 256 entradas e associatividade *ful-way* pouco reuso é observado (máximo de 7.33%). A partir de 512 entradas o percentual máximo de reuso aumenta para 16.11%, 17.30%, 17.38% e 17.41%, nas opções de 512, 1K, 2K e 4K entradas respectivamente.

Pela Figura 4.2 constata-se que as mesmas observações em relação ao percentual de reuso, podem ser aplicadas à aceleração. Ou seja, a aceleração é fortemente influenciada pelo aumento da associatividade e também é possível observar dois conjuntos distintos de aceleração: para opções de 64 à 256 entradas a aceleração máxima alcançada é de 1.03, enquanto que para opções de 512, 1K, 2K e 4K entradas é igualmente alcançada a aceleração de 1.09.

É interessante observar que, na maioria dos casos, os valores absolutos do per-

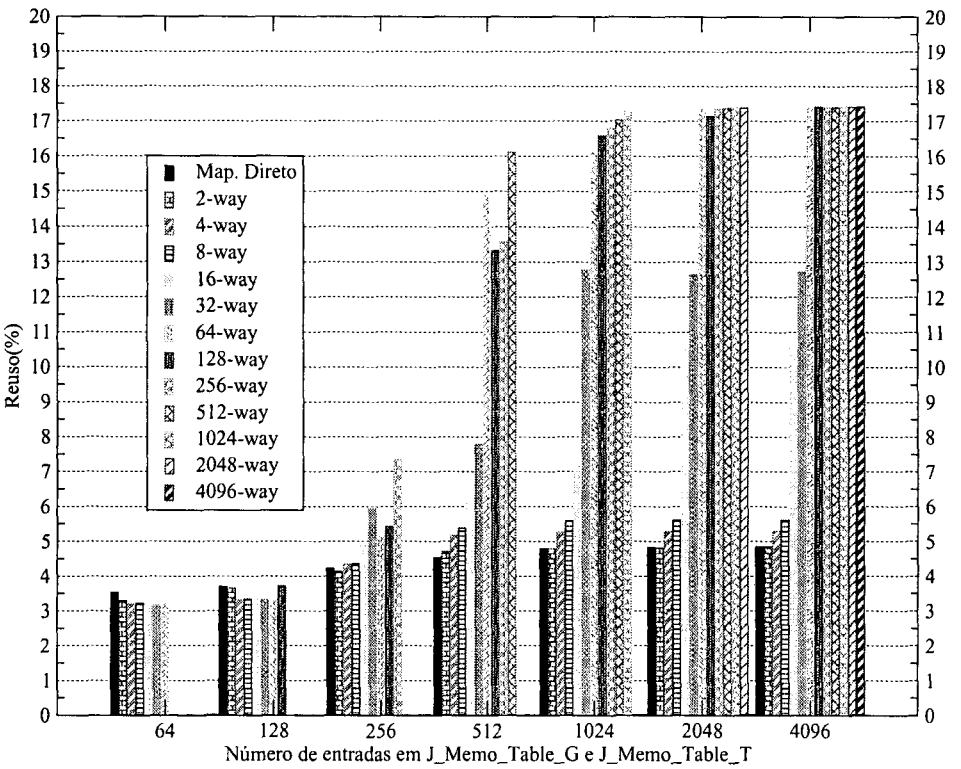



Figura 4.1: Percentual de reuso obtido variando o tamanho e a associatividade de ambas as tabelas.

centual de reuso são pouco maiores que a aceleração obtida. Ou seja, um reuso de 50% não equivale à uma aceleração de 1.5. Isso é primeiramente justificado pela imposição, por parte do mecanismo JDTM, de uma penalidade de 2 ciclos (1 bolha no estágio ID e 1 bolha no estágio OF) para cada traço reusado. Além disso, o reuso de traços que possuam desvio realizado impõe a mesma penalidade de um desvio (3 ciclos) devido a necessidade de efetuar um *flush* na fila de *prefetch*.

A aceleração de desempenho obtida através do mecanismo JDTM é principalmente justificada pela redução do número de *bytecodes* executados, *i.e.*, pelo reuso de computação redundante. Além disso, para o caso de traços reusados, os quais possuam mais de um *bytecode* de desvio seguido, há uma redução global das penalidades impostas por desvios, visto que o reuso destes traços impõem apenas 3 ciclos de penalidade, independente do número de desvios encapsulados no respectivo traço.

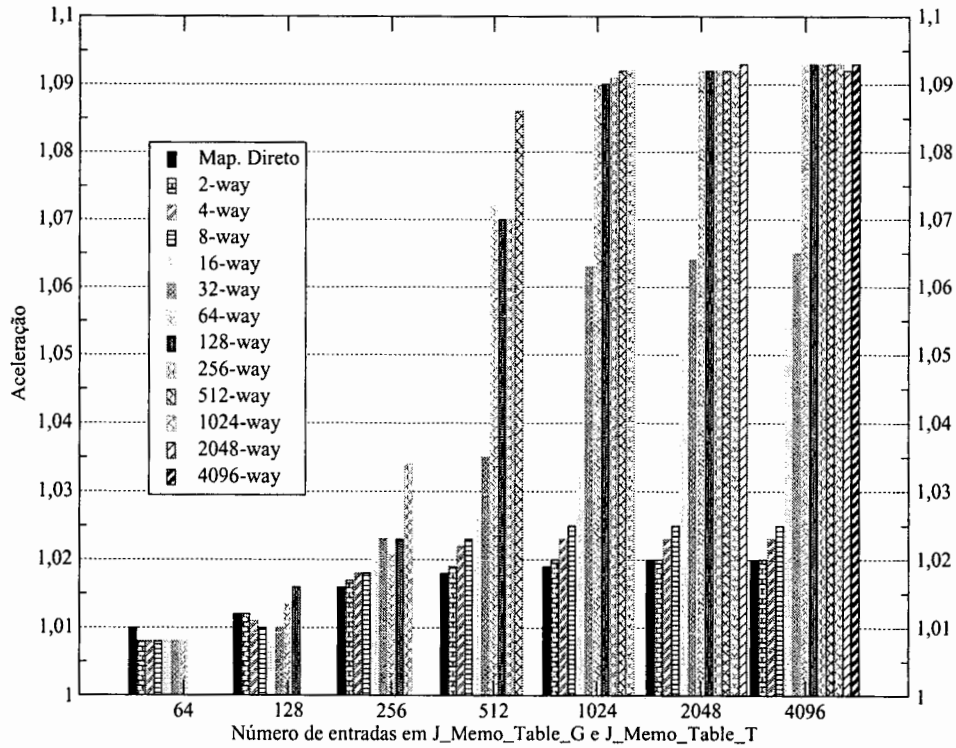



Figura 4.2: Aceleração obtida variando o tamanho e a associatividade de ambas as tabelas.

#### 4.2.1.1 Remarcação dinâmica de *bytecodes* não redundantes

É notável que acertos em J_Memo_Table_G são altamente importantes para a identificação de *bytecodes* redundantes e conseqüente construção dos traços. A fim de reduzir o tamanho e/ou associatividade de J_Memo_Table_G, sem porém resultar em uma perda significativa na aceleração, uma interessante constatação pode ser explorada: Se os valores de todos os operandos de entrada instanciados à um *bytecode*, foram produzidos por *bytecodes* rotulados como redundantes, este *bytecode* também pode ser rotulado como tal independentemente de possuir ou não sua instância memorizada em J_Memo_Table_G. Portanto, o mecanismo de construção de traços foi ligeiramente alterado de forma a remarcar, como redundante, os *bytecodes* que foram rotulados como não redundantes mas que possuem todos os operandos de entrada produzidos por *bytecodes* redundantes.

A alteração consistiu basicamente na adição de um registrador de 64 bits denominado *Registrador de Bits de Monitoração* (*RBM*), onde o bit $i$ do RBM é referente ao registrador $i$ do banco de registradores. Se o bit $i$ estiver ativado, significa que o valor do registrador $i$ foi produzido por um *bytecodee* rotulado como redundante e portanto esse valor é um *resultado redundante.*

A Figura 4.3 esboça o processo de remarcação de *bytecodes* não redundantes:

- A Figura 4.3(a) exibe o estado do banco de registradores e do RBM antes da execução do primeiro *bytecode* do traço a ser construído.
- A Figura 4.3(b) exibe o estado do banco de registradores e do RBM após a execução do primeiro *bytecode*, o qual desempilha o topo e subtopo da pilha. A instância deste *bytecode* está presente em J_Memo_Table_G e portanto ele é rotulado como redundante.
- A Figura 4.3(c) exibe o estado do banco de registradores e do RBM após a execução do *bytecode* `0x14A`. Este *bytecode* empilha a constante 0, e por ser do

tipo constante, é automaticamente rotulado como redundante e o respectivo bit do RBM é ativado.

- A Figura 4.3(d) exibe o estado do banco de registradores e do RBM após a execução do *bytecode* `0x14B`. Este *bytecode*, o qual carrega o valor da variável local 6 e o empilha na pilha de operandos, não possui instância memorizada em J_Memo_Table_G. Portanto ele é, a princípio, rotulado como não redundante. Mas como o respectivo bit de seu registrador fonte (var. 6) foi ativado por algum *bytecode* anterior no fluxo de execução, este *bytecode* é remarcado como redundante e ativado o bit do RBM referente ao seu registrador destino.

- A Figura 4.3(e) exibe o estado do banco de registradores e do RBM após a execução do *bytecode* `0x14D`. Este *bytecode*, o qual desempilha o topo e subtopo da pilha, não possui instância memorizado em J_Memo_Table_G e portanto é rotulado como não redundante. Porém, como os bits de seus operandos fontes estão ativados, o que significa que estes valores foram produzidos por *bytecodes* rotulados como redundantes (*bytecode* `0x14A` e `0x14B`), o *bytecode* `0x14D` é remarcado como redundante.

- A Figura 4.3(f) exibe o estado do banco de registradores e do RBM após a execução do *bytecode* `0x158`. Este *bytecode*, o qual incrementa o valor da variável local 2, possui sua instância memorizada em J_Memo_Table_G e portanto é rotulado como redundante. Além disso, é ativado o respectivo bit no RBM indicando que o resultado é redundante, e se no futuro for unicamente utilizado como operando de entrada por algum *bytecode* com instância não memorizada, será responsável pela remarcação deste *bytecode*.

- O próximo *bytecode* da sequência não faz parte do domínio de *bytecodes* válidos e portanto é responsável pela finalização da construção do traço e sua conseqüente memorização em J_Memo_Table_T.

De forma semelhante, se o valor a ser escrito numa determinada posição da pilha foi produzido por um *bytecode* rotulado como não redundante o respectivo bit do RBM é desativado.

Com o objetivo de analisar os ganhos alcançados com o JDTM através dessa nova versão de construção de traços, os experimentos referentes ao percentual de reuso e aceleração foram re-executados e seus resultados são exibidos nas Figuras 4.4 e 4.5 respectivamente. Vale mencionar que, a partir deste ponto, todos os resultados que serão apresentados foram obtidos com o JDTM utilizando a remarcação de *bytecodes* não redundantes.

Pela Figura 4.4 é possível observar que altos valores de associatividade continuam sendo importante, mesmo com a remarcação. Porém os dois conjuntos distintos de percentuais de reuso não existem mais. Isto porque, com a remarcação, mesmo pequenas tabelas já são suficientes para fornecer identificação de *bytecodes* redundantes, o que permite alcançar percentuais de reuso de 15.62%, 17.16% e 18.81% para tabelas com 64, 128 e 256 entradas respectivamente e associatividade *full-way*. É interessante também observar que o percentual de reuso aumentou no caso das tabelas maiores, atingindo 19.96%, 24.39%, 24.53% e 24.55% para tabelas com 512, 1K, 2K e 4K entradas respectivamente e associatividade *full-way*. O crescimento suave do percentual de reuso a medida que o número de entradas nas tabelas é incrementado é devido em grande parte pela identificação de *bytecodes* redundantes deixar de ser de única e exclusiva responsabilidade de J_Memo_Table_G e passar a ser fornecida também pela monitoração dos valores produzidos através do uso do RBM

Na Figura 4.5 observa-se que a aceleração também passa a contar com apenas um conjunto de aceleração, e mantém a importância dos altos valores de associatividade, pelos mesmo motivos apresentados para os resultados de percentual de reuso. Com associatividade *full-way*, foram alcançadas acelerações de 1.10, 1.10, 1.11, 1.11, 1.13,

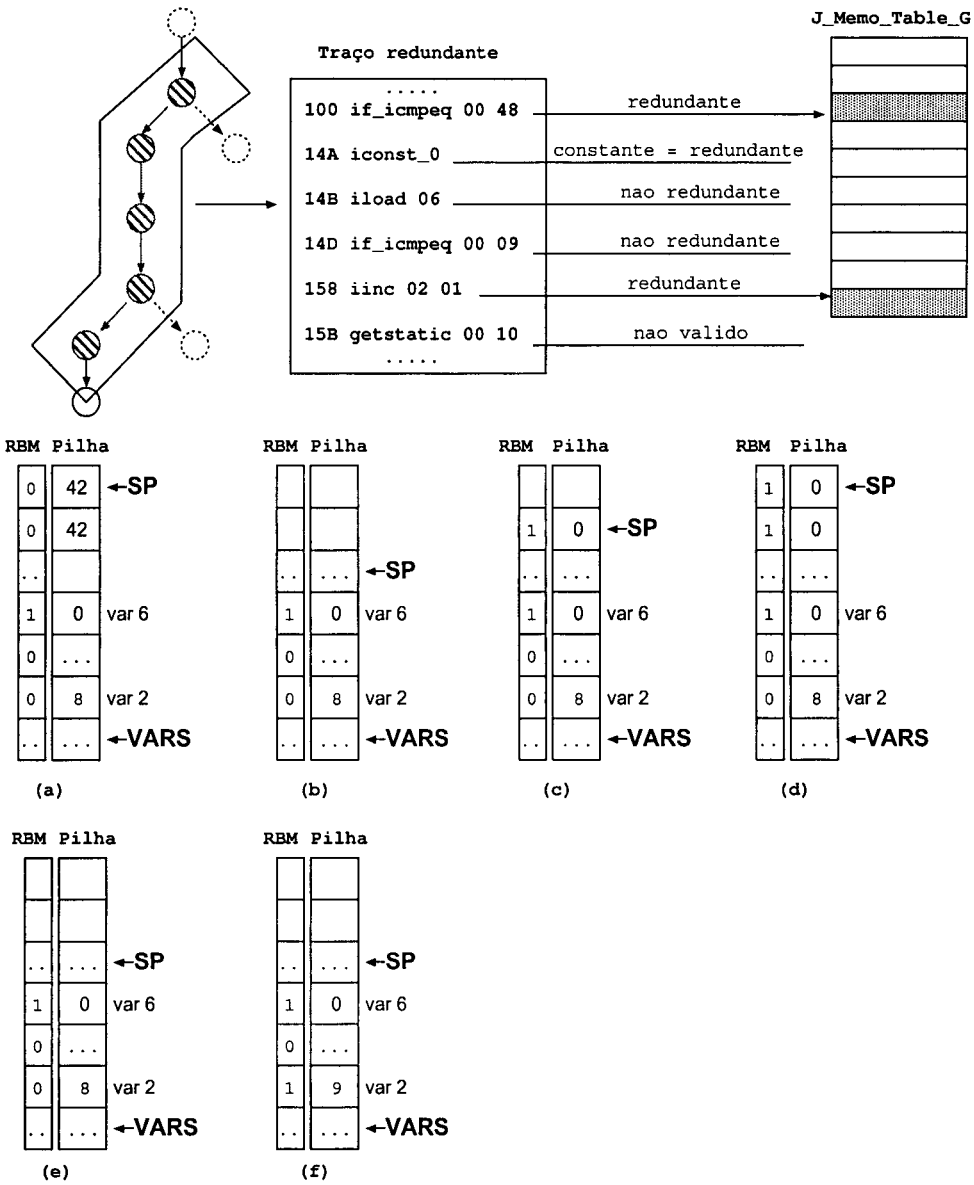



Figura 4.3: Remarcação de *bytecodes* não redundante, com base na monitoração dos valores produzidos.

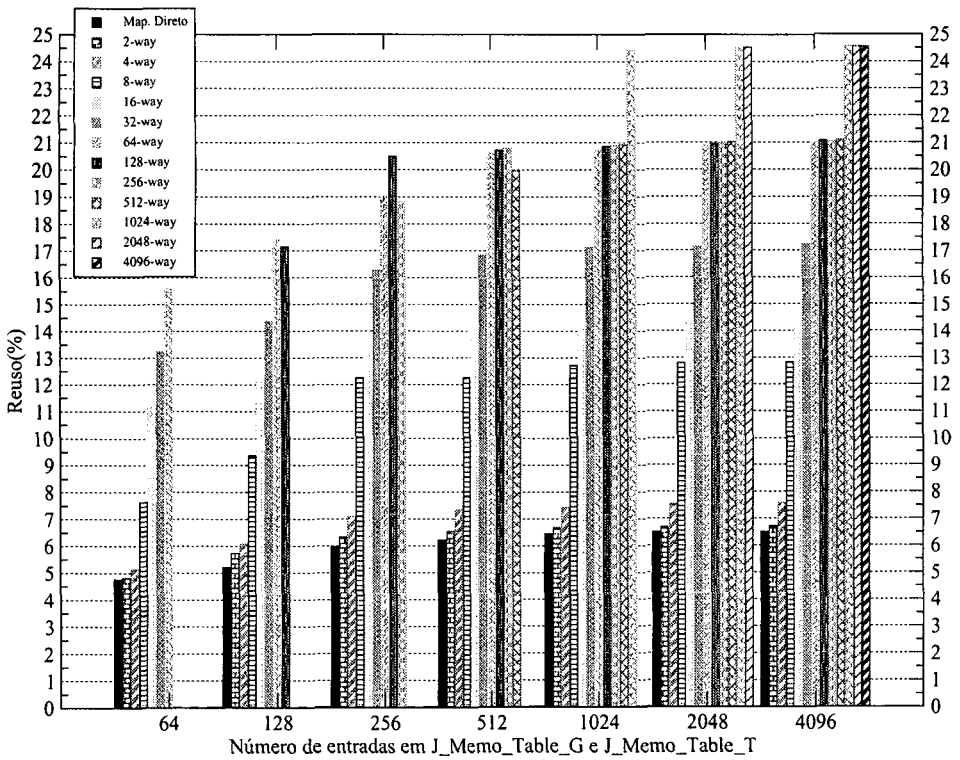



Figura 4.4: Percentual de reuso obtido através do JDTM com remarcação de *bytecodes* não redundantes, variando o tamanho e a associatividade de ambas as tabelas.

1.13, 1.13 para tabelas com 64, 128, 256, 512, 1K, 2K e 4K entradas respectivamente.

### Percentual de reuso obtido em cada *benchmark*

A fim de analisar, em cada *benchmark*, a sensibilidade do JDTM ao aumento da associatividade e do tamanho das tabelas, as Figuras 4.6 e 4.7 exibem respectivamente o percentual de reuso: fixando a associatividade em *full-way* e variando o número de entradas de 64 à 4K, e fixando as tabelas em 4K entradas e variando a associatividade de *mapeamento direto* à *full-way*.

Tanto na Figura 4.6 quanto na Figura 4.7 é possível observar 2 conjuntos distintos de percentuais máximos de reuso: O primeiro conjunto com grande percentual de reuso (máximo de 44.71% na execução do Select Sort e 55.35% na execução do CORDIC) e o segundo com percentual máximo abaixo de 28% (MP3 player com

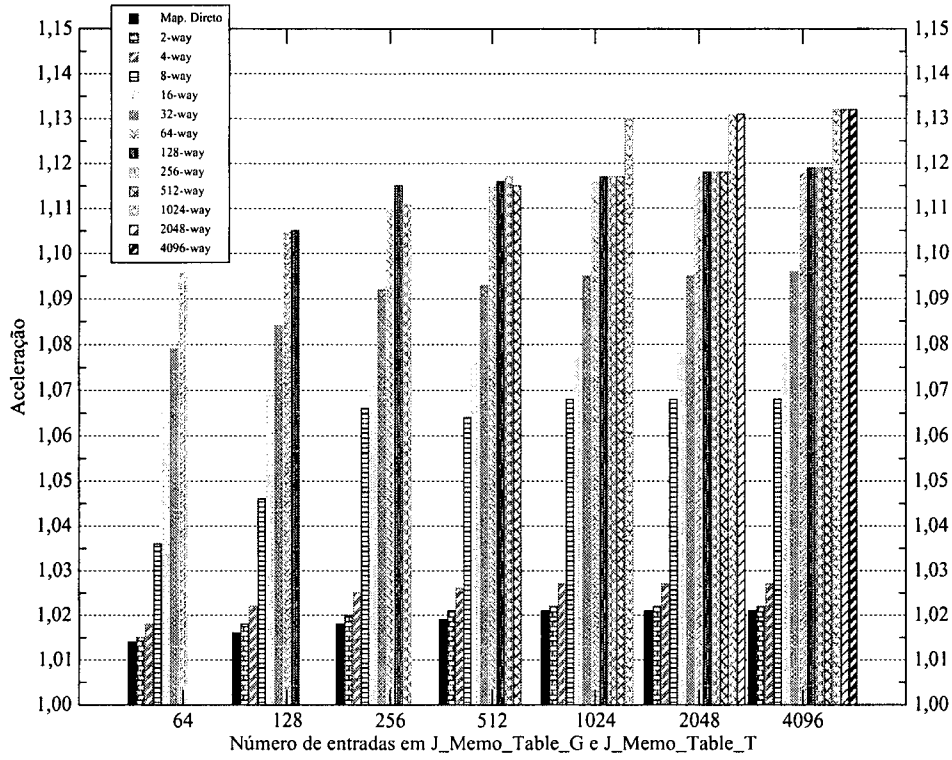



Figura 4.5: Aceleração obtida através do JDTM com remarcação de *bytecodes* não redundantes, variando o tamanho e a associatividade de ambas as tabelas.

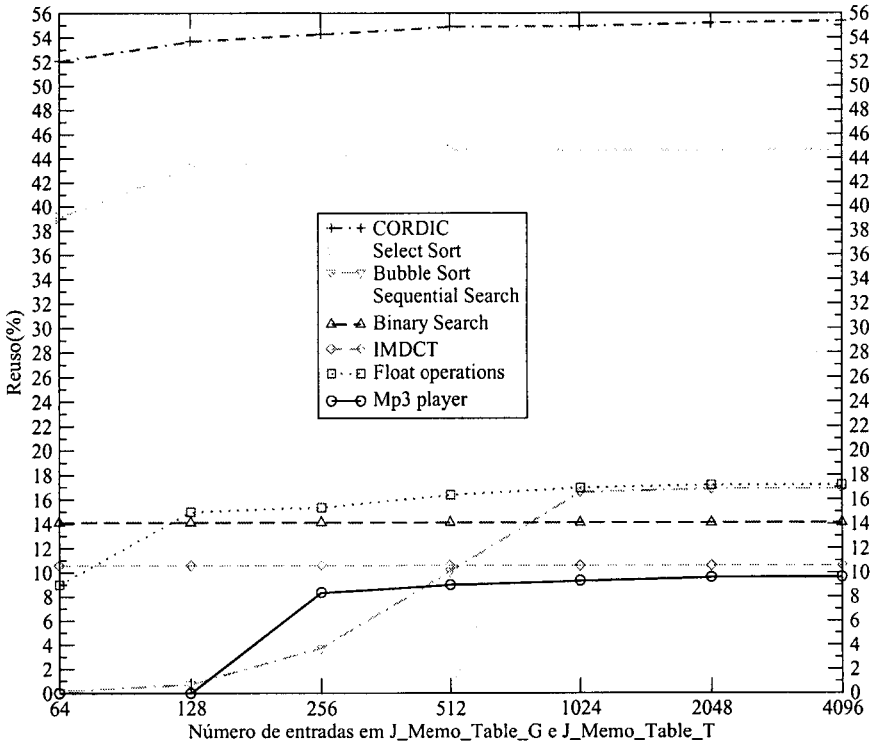



Figura 4.6: Percentual de reuso alcançado por cada *benchmark*, com associatividade fixada em *full-way* e variação do número de entradas de ambas as tabelas.

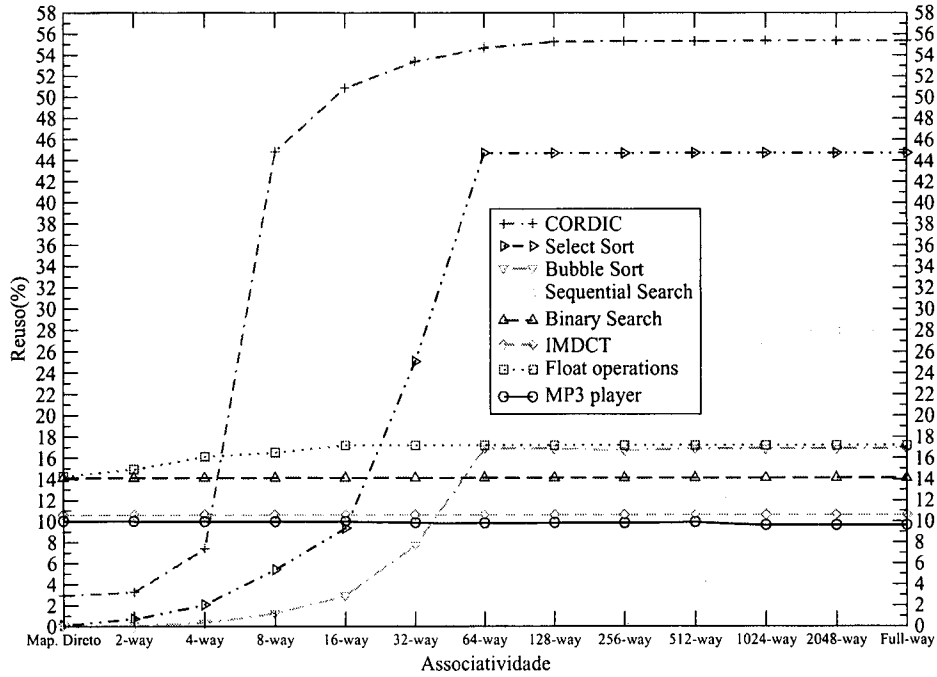



Figura 4.7: Percentual de reuso alcançado por cada *benchmark*, com o número de entradas fixado em 4K e variação da associatividade de ambas as tabelas.

9.61%, IMDCT com 10.57%, Binary Search com 14.10%, Bubble Sort com 16.88%, Float operations com 17.19% e Sequential Search com 27.95% de reuso).

### Aceleração obtida em cada *benchmark*

O quanto a aceleração, obtida na execução de cada *benchmark*, é influenciada pelo aumento do tamanho de ambas as tabelas, é exibido na Figura 4.8. Nesta análise, a associatividade é fixada em *full-way* e varia-se o tamanho das tabelas de 64 à 4K entradas.

Objetivando fazer semelhante análise de sensibilidade da aceleração, a Figura 4.9 exibe a aceleração obtida na execução de cada *benchmark*, fixado o tamanho de ambas as tabelas em 4K entradas e variando a associatividade de *mapeamento direto* à *full-way*.

Pelas Figuras 4.8 e 4.9 é possível ver que existem dois conjuntos distintos de

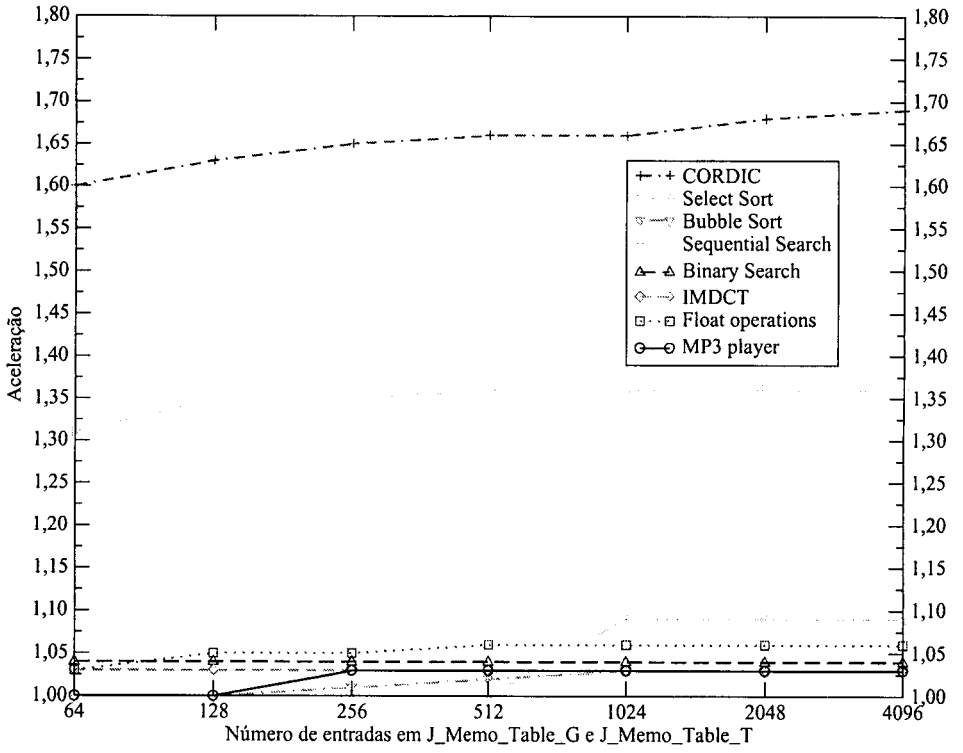



Figura 4.8: Aceleração alcançada por cada *benchmark*, fixando a associatividade em *full-way* e variando o número de entradas de ambas as tabelas.

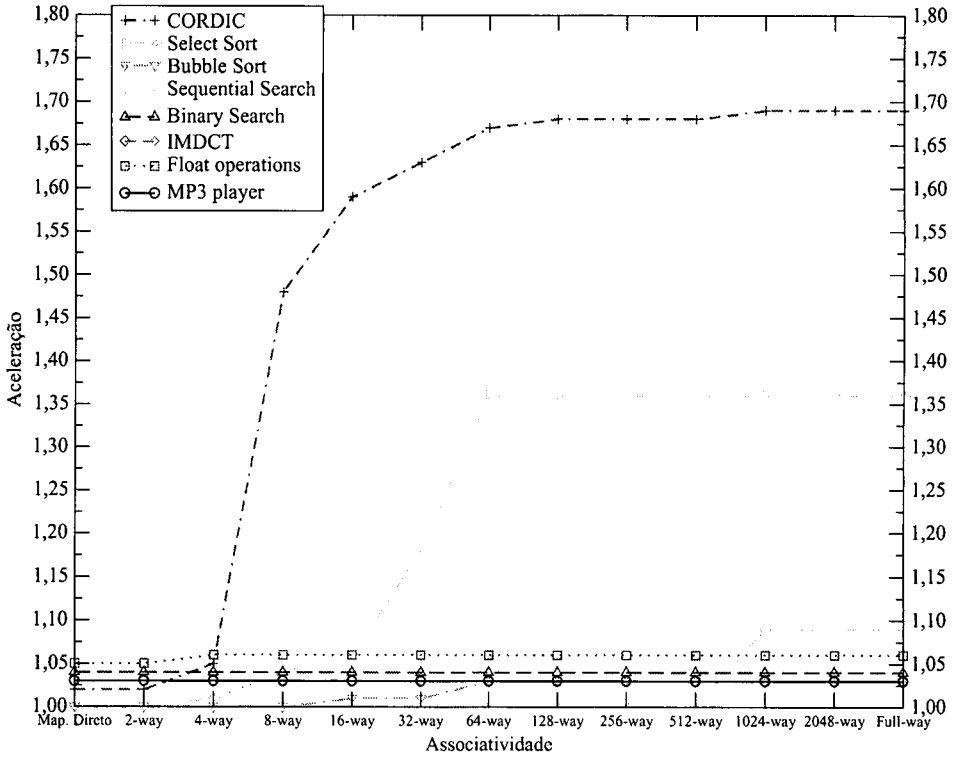



Figura 4.9: Aceleração alcançada por cada *benchmark*, fixado o número de entradas em 4K e variando a associatividade de ambas as tabelas.

aceleração máxima: O primeiro conjunto com grande aceleração (1.36 no Select Sort e 1.69 no CORDIC) e o segundo com aceleração máxima abaixo de 1.09 (IMDCT, MP3 player e Bubble Sort com 1.03, Binary Search com 1.04, Float operations com 1.06 e Sequential Search com 1.09).

O aumento da associatividade é mais importante para se obter grandes melhorias no desempenho do que o aumento do tamanho das tabelas. Para constatar isso, basta notar na Figura 4.9 que, especialmente no caso dos *benchmarks* Select Sort e CORDIC, a alteração da associatividade de *mapeamento direto* para *full-way* faz a aceleração ser incrementada de 1.00 para 1.36 e de 1.02 para 1.69 respectivamente. Já na Figura 4.8 não se observa um incremento tão significativo da aceleração, quando o número de entradas é incrementado de 64 para 4K entradas.

### 4.2.2 Custo e efetividade

Todos resultados apresentados até o momento consideraram o mesmo número de entradas em ambas as tabelas de memorização. Visando identificar a importância de cada uma para a construção e reuso dos traços, as Figuras 4.10 e 4.11 exibem a aceleração obtida com todas as possíveis combinações de tamanho de J_Memo_Table_G e J_Memo_Table_T, com variação de 64 à 4K entradas e associatividade fixada em *full-way*.

A Figura 4.10 exibe tal combinação tomando como base o JDTM *sem* a remarcação de *bytecodes* não redundantes. Neste caso, pode-se observar que o aumento do número de entradas na tabela J_Memo_Table_G é mais importante do que na tabela J_Memo_Table_T. Observa-se também que o aumento do número de entradas em J_Memo_Table_T só influencia na aceleração a partir de J_Memo_Table_G com 512 entradas.

Já a Figura 4.11 exibe a mesma combinação, tomando como base o JDTM *com* a

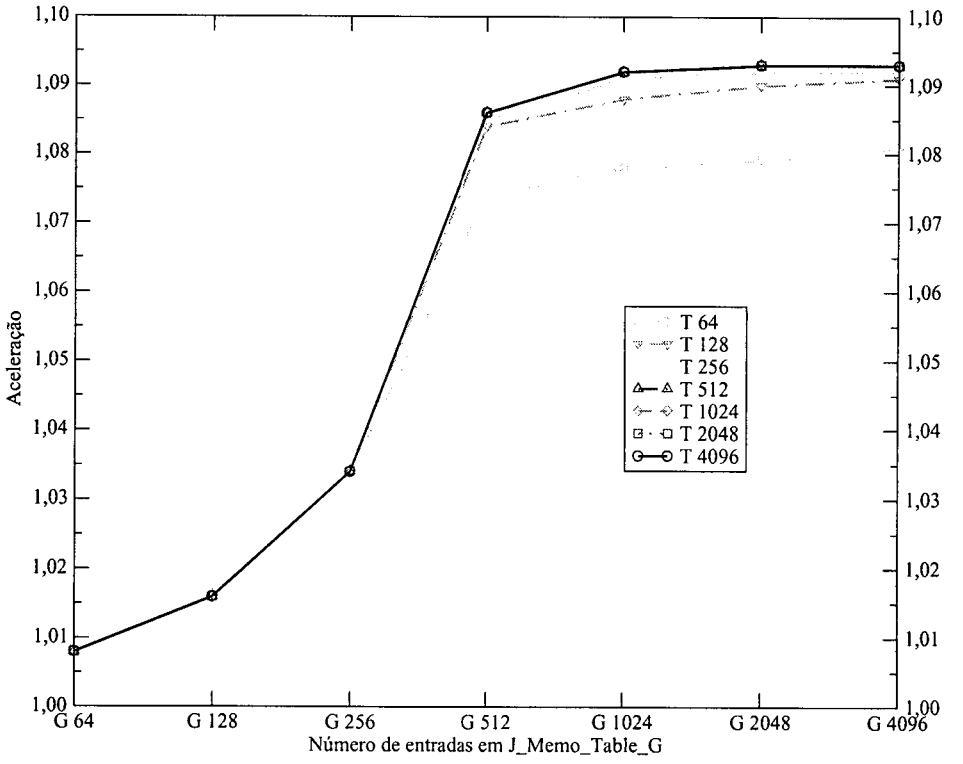



Figura 4.10: Aceleração obtida a partir do JDTM sem a remarcação de *bytecodes* não redundantes e todas as possíveis combinações de número de entradas de J_Memo_Table_G e J_Memo_Table_T.

remarcação de *bytecodes* redundantes. Neste caso observa-se que o aumento do número de entradas da tabela de traços é mais importante do que o aumento na tabela J_Memo_Table_G. Tal constatação, inversa ao observado na Figura 4.10, é devida a remarcação de *bytecodes* redundante que reduz a importância, para a construção de traços, da tabela de memorização global. Observa-se que neste caso, mesmo J_Memo_Table_G com apenas 64 entradas é suficiente para alcançar aceleração considerável em todas as opções de tamanho de J_Memo_Table_T (aceleração de aproximadamente 1.10, 1.10, 1.10, 1.10, 1.11, 1.11 e 1.11 considerando J_Memo_Table_T com 64, 128, 256, 512, 1K, 2K e 4K entradas respectivamente). .

A partir do exposto na Figura 4.11 observa-se semelhante aceleração com 1K, 2K e 4K entradas em J_Memo_Table_T. O que leva a concluir que 1K entradas é a

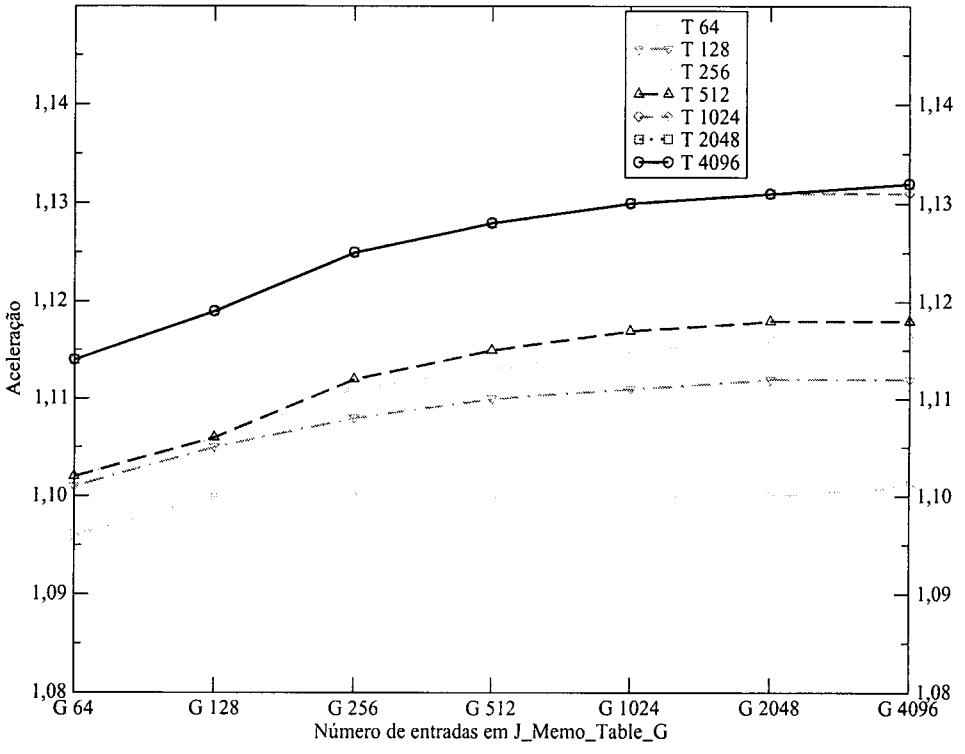



Figura 4.11: Aceleração obtida a partir do JDTM com a remarcação de *byte-codes* redundantes e todas as possíveis combinações de número de entradas de J_Memo_Table_G e J_Memo_Table_T.

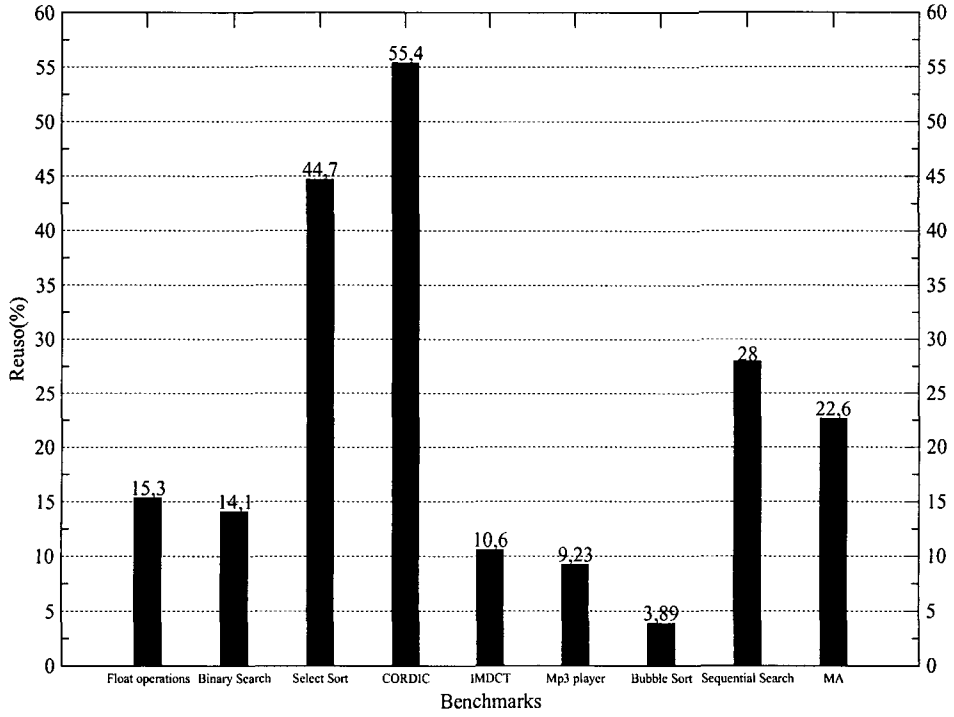



Figura 4.12: Percentual de reuso obtido em cada *benchmark* utilizando J_Memo_Table_G com 256 entradas e J_Memo_Table_T com 1K entradas.

melhor opção para a tabela de memorização de traços.

Com 1K entradas em J_Memo_Table_T, pode-se observar aceleração de aproximadamente 1.11, 1.12, 1.12, 1.12, 1.13, 1.13, 1.13 considerando J_Memo_Table_G com 64, 128, 256, 512, 1K, 2K e 4K entradas respectivamente. A partir do exposto, J_Memo_Table_T com 1K entradas e J_Memo_Table_G com 256 entradas são as melhores opções de tamanho para obter uma considerável aceleração e minimização do custo de implementação das tabelas.

As Figuras 4.12 e 4.13 apresentam respectivamente o percentual de reuso e a aceleração obtidos em cada *benchmark*, utilizando J_Memo_Table_G com 256 entradas e J_Memo_Table_T com 1K entradas, ambas com associatividade *full-way*.

A execução dos benchmarks Bubble Sort e CORDIC obtiveram o menor (3.89%) e o maior (55.4%) percentual de reuso respectivamente. Conseqüentemente, esse re-

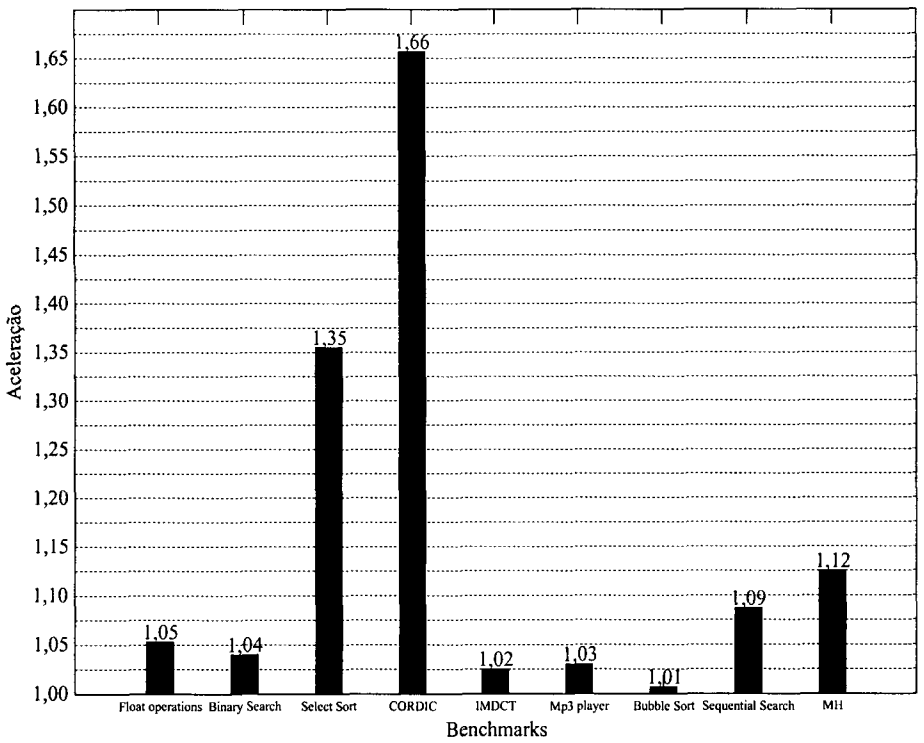



Figura 4.13: Aceleração obtida em cada *benchmark* utilizando J_Memo_Table_G com 256 entradas e J_Memo_Table_T com 1K entradas.

sultado refletiu na menor (1.01) e na maior (1.66) aceleração. Em média aritmética, o JDTM alcançou 22.6% de reuso e em média harmônica alcançou uma aceleração de 1.12.

Portanto todos os próximos resultados foram obtidos a partir de J_Memo_Table_G com 256 entradas e J_Memo_Table_T com 1K entradas e ambas com associatividade *full-way*.

### 4.2.3 Caracterização dos *bytecodes* reusados

Esta subseção caracteriza os *bytecodes* reusados de acordo com as 6 categorias da Tabela 2.1 que classifica todos os *bytecodes* da ISA do processador FemtoJava.

Na Figura 4.14, que caracteriza os *bytecodes* reusados na execução de cada *ben-*

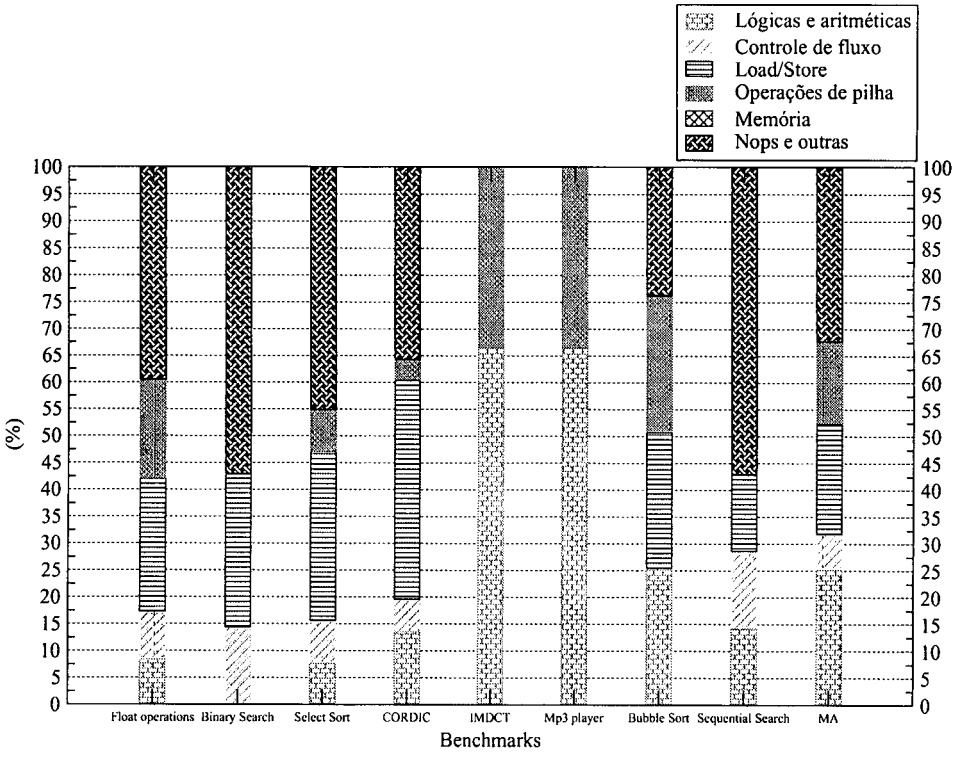



Figura 4.14: Distribuição total de *bytecodes* reusados.

*chmark*, pode ser observado que em média, 25.34% dos *bytecodes* reusados são da classe *Lógica e Aritmética*, 6.40% são do tipo *Controle de Fluxo*, 20.68% pertencem à categoria *Load/Store*, 15.24% são do tipo *Operações de Pilha*, 0% pertencem à categoria *Memória* e 32.33% são de *bytecodes nops*.

Vale mencionar que a categoria *Controle de Fluxo* exclui os *bytecodes* `return`, `ireturn`, `areturn` e `invokestatic` (de acordo com a Tabela 3.1 que apresenta o domínio de *bytecodes* válidos para a construção de traços). Além disso, a inexistência de reuso de *bytecodes* do tipo *Memória* é também devida à não inclusão desta categoria no domínio de *bytecodes* válidos. Finalmente, a categoria *nops/outras* somente inclui os *bytecodes* `nops`, executados devido às dependências de recursos, de dados e de controle. Esta categoria apresenta, em média aritmética, o maior percentual de reuso (32.33%).

### 4.2.4 Tamanho dos contexto de entrada e saída do traços reusados

O número de elementos no contexto de entrada e de saída influenciam diretamente o número de portas de leitura e escrita necessárias no banco de registradores. Tal acréscimo de portas atua negativamente no consumo de potência e área do processador. Visando fazer uma estimativa do impacto do JDTM no custo de uma implementação real do processador substrato, essa subseção descreve informações referentes ao tamanho dos contexto de entrada e de saída dos traços reusados.

A Figura 4.15 apresenta a distribuição percentual do número de elementos no contexto de entrada dos traços reusados. Esta medida determina uma condição arquitetural que define o número de valores acessados e comparados em paralelo (por entrada pré-selecionada em J_Memo_Table_T), para se identificar uma oportunidade de reuso. Pode-se observar que em média 3.45% dos traços reusados possuem 0 elementos no contexto de entrada ou seja, são compostos por *bytecodes* constantes (ex. `bipush`, `iconst_2`). Traços com 1, 2, 3, 4, 5 e 6 elementos no contexto de entrada representam em média 33.33%, 42.48%, 9.57%, 10.45%, 0.46% e 0.28% respectivamente. Portanto, 79.26% dos traços reusados possuem contexto de entrada com menos de 3 elementos.

A Figura 4.16 apresenta a distribuição percentual do número de elementos no contexto de saída dos traços reusados. Essa medida determina uma condição arquitetural que define o número total de portas de escrita necessárias no banco de registradores para permitir a atualização do estado a partir dos valores presentes no contexto de saída do traço reusado.

Observa-se que em média 2.46% dos traços reusados possuem 0 elementos no contexto de saída. Estes traços não possuem *bytecodes* de escrita no *pool* de variáveis locais (ex. `istore_0`) excluindo a existência de elementos no *contexto saída referente ao pool* e são finalizados basicamente por *bytecodes* de desvios, que desempilham

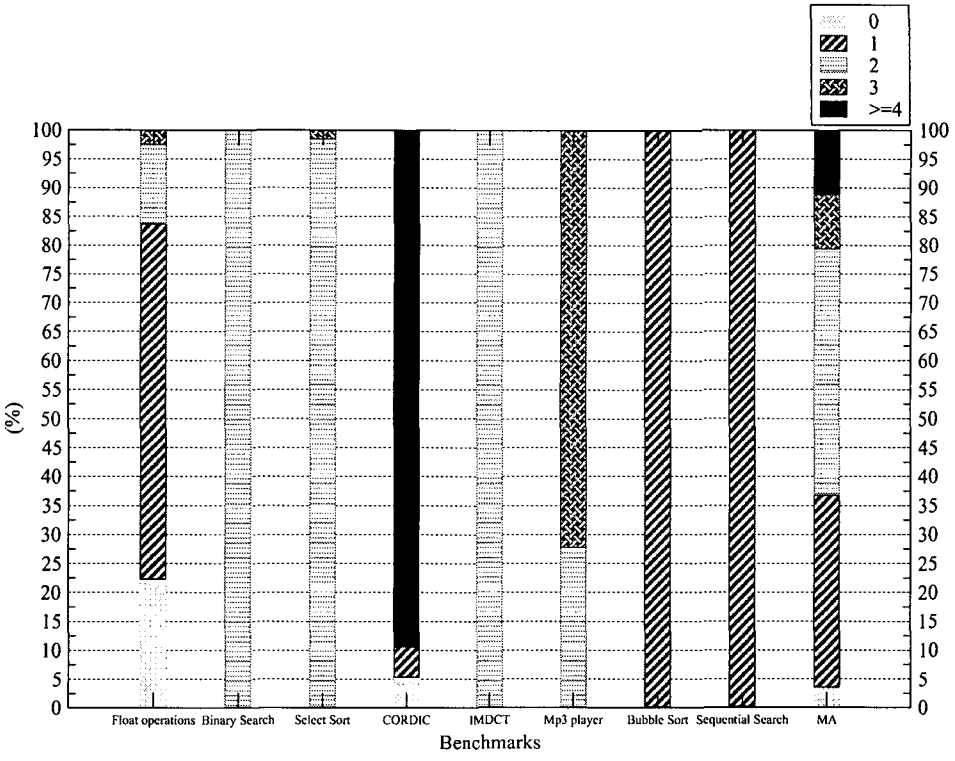



Figura 4.15: Distribuição percentual do número de elementos no contexto de entrada para os traços reusados.

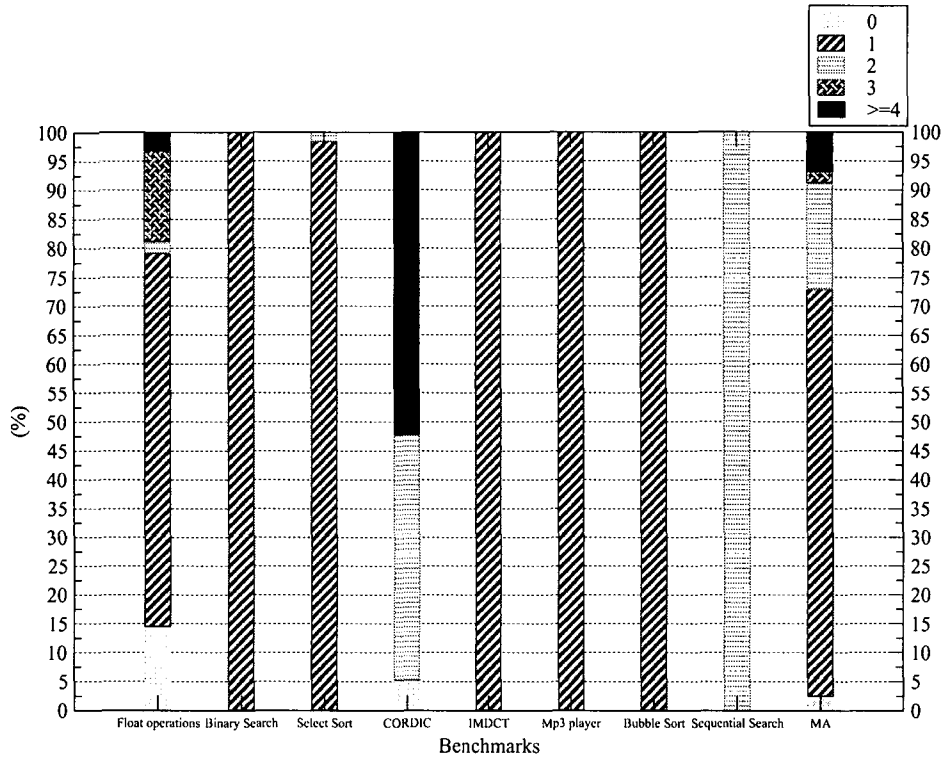



Figura 4.16: Distribuição percentual do número de elementos no contexto de saída para os traços reusados.

valores produzidos ou não pelo traço em questão, excluindo a existência do *contexto de saída referente à pilha.*

Traços com 1, 2, 3, 4, 5, 6, 7 e 8 elementos no contexto de saída, representam em média 70.39%, 18.27%, 1.96%, 5.39%, 0.15%, 0.85%, 0.22% e 0.32% dos traços reusados respectivamente. É importante observar que 72.85% dos traços reusados possuem no máximo 1 elemento no contexto de saída, resultando em um baixo custo de implementação em se tratar de portas de escritas adicionais no banco de registradores.

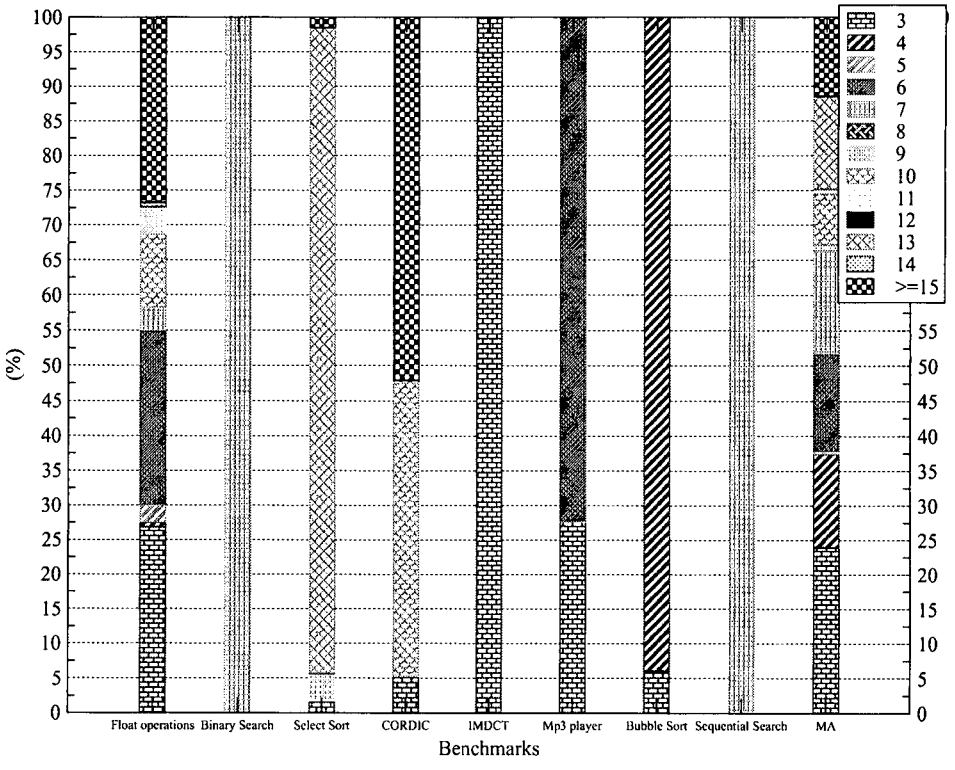



Figura 4.17: Distribuição percentual do número de *bytecodes* nos traços reusados.

### 4.2.5 Número de *bytecodes* inclusos nos traços reusados

O número total de *bytecodes* inclusos nos traços reusados, *i.e.*, o tamanho destes traços, representa o número total de *bytecodes* que deixam de ser executados durante o reuso.

A Figura 4.17 apresenta a distribuição percentual do número de *bytecodes* representados nos traços reusados. Vale mencionar, que os *bytecodes* nop, impostos por perigos estruturais, de dados e de controle também são considerados nesta distribuição. Pelo exposto é possível ver que em média 23.89% dos traços reusados possuem três *bytecodes* (tamanho mínimo exigido pelo JDTM para a memorização de traços). Traços com 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 e 15 *bytecodes* representam respectivamente 13.52%, 0.39%, 13.88%, 14.75%, 0.00%, 0.58%, 7.58%, 0.56%, 0.00%, 13.26%, 0.09% e 11.50% dos traços reusados.

### 4.2.6 Número de desvios realizados inclusos nos traços reusados

Dependências de controle são comumente conhecidos como penalidades impostas por desvios no fluxo sequencial da execução das instruções de uma aplicação alvo. Tais penalidades podem ser reduzidas com uso de um esquema de previsão dinâmica ou estática de desvios. No caso do processador *FemtoJava Low Power 32 bits*, um esquema de previsão estática de desvios é utilizado, onde todos os desvios são considerados a *priori* como *não realizados*. Durante o estágio EX, o resultado da comparação dos operandos do *bytecode* e a decisão por realizar ou não o desvio são conhecidos. Caso o resultado da comparação seja falso, os *bytecodes* nos estágios anteriores do *pipeline* seguem normalmente para os próximos estágios. Caso contrário, estes *bytecodes* são desconsiderados, o fluxo de execução é redirecionado para o endereço alvo do desvio e portanto uma penalidade de 3 ciclos é imposta.

O mecanismo JDTM permite a construção e memorização de traços com fluxo de execução não sequencial, ou seja, permite a inclusão de *bytecodes* de desvio sendo estes seguidos ou não seguidos. O reuso de traços com fluxo de execução sequencial, impõe um penalidade de 2 ciclos devida à desconsideração dos *bytecodes* que estão no estágio ID e OF. Porém, traços que possuam pelo menos 1 *bytecode* de desvio realizado em sua composição, impõem a mesma penalidade de um desvio na arquitetura sem o JDTM, *i.e.*, 3 ciclos de *clock*. Estas penalidade é devida à desconsideração dos *bytecodes* que estão no estágio ID e OF, além do redirecionamento da busca de *bytecodes* do estágio IF.

Traços com mais de 1 *bytecode* de desvio realizado mantêm a penalidade de 3 ciclos. Desta forma, a penalidade global imposta por desvios realizados é reduzida, pois alguns destes são colapsados em traços com mais de 1 *bytecode* de desvio realizado.

A Figura 4.18 apresenta a distribuição percentual de desvios realizados nos traços

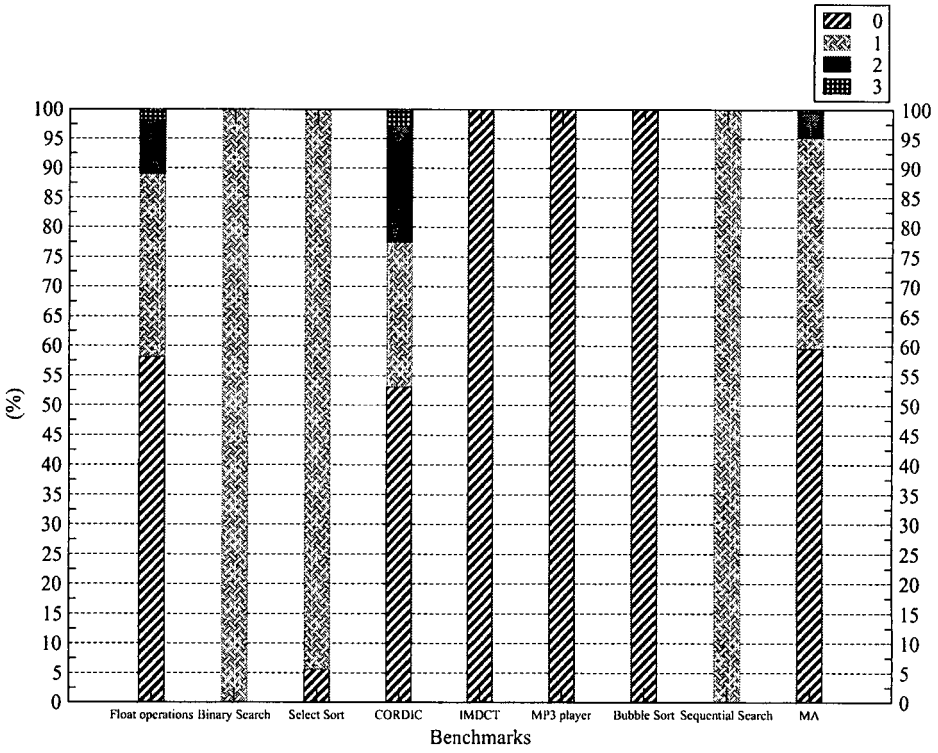



Figura 4.18: Distribuição percentual do número de desvios realizados nos traços reusados.

reusados. Observa-se que em média 59.56% dos traços reusados possuíam fluxo de execução sequencial, 35.63% possuíam 1 desvio realizado, 4.11% possuíam 2 desvios realizados e apenas 0.70% possuíam 3 desvios realizados.

É notável que no caso do *benchmark* CORDIC (maior aceleração observada entre todos os benchmarks), cerca 19.64% dos traços reusados possuíam 2 desvios realizados e 2.95% possuíam 3 desvios realizados, o que favorece a redução das penalidades impostas por esses desvios.

### 4.2.7 Caracterização do fim da construção dos traços memorizados

Como foi mencionado na Subseção 3.4.1, uma das formas de finalização da construção de um traço se dá quando é identificado um *bytecode* não pertencente ao domínio de *bytecodes* válidos . Outra forma é quando o *bytecode* pertence ao domínio de *bytecodes* válidos, porém este não foi rotulado como redundante devido à sua instância não estar memorizada em J_Memo_Table_G e o valor de seus operandos de entrada não terem sido produzidos por *bytecodes* redundantes, o que possibilitaria a remarcação. A fim de caracterizar o fim da construção dos traços memorizados com base nas duas formas de finalização citadas acima, a Figura 4.19 apresenta a distribuição percentual, o que possibilita observar que em média 60% dos traços memorizados foram finalizados devido à identificação de *bytecodes* não pertencentes ao domínio de *bytecodes* válidos, *i.e*, *bytecodes* que fazem acesso à memória. Portanto, conclui-se que ao estender o domínio de *bytecodes* válidos de forma a incorporar acesso à memória na construção dos traços permitirá a construção e consequente reuso de traços maiores.

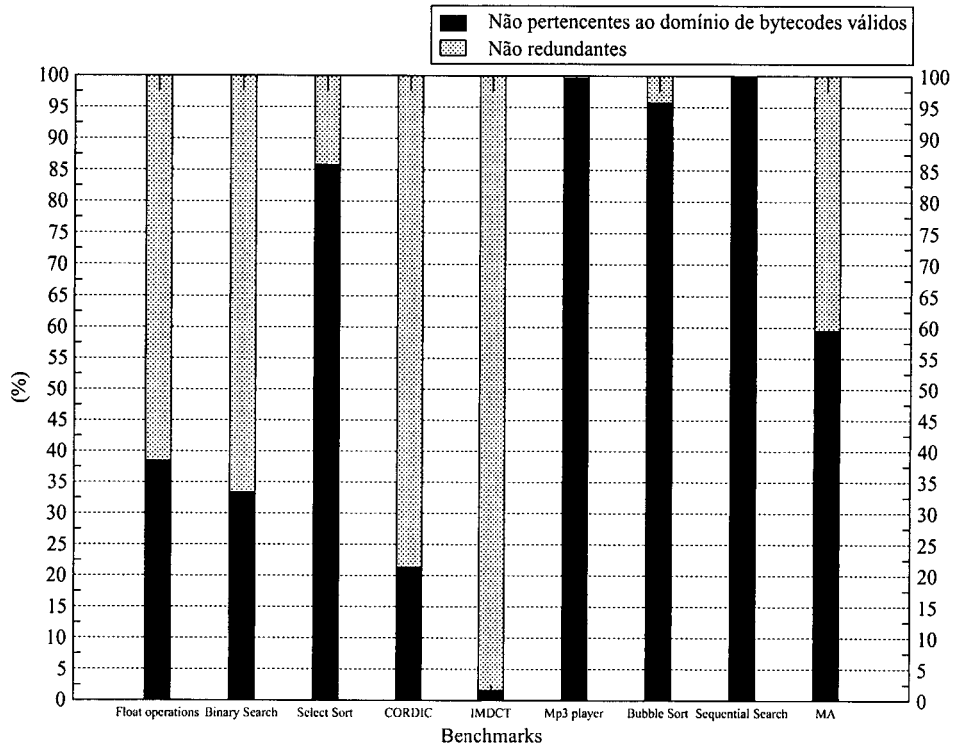



Figura 4.19: Distribuição percentual da forma de finalização da construção dos traços memorizados.

# Capítulo 5

# Conclusões e Trabalhos Futuros

## 5.1 Conclusão

Foi amplamente discutido neste trabalho a exploração de computação redundante como uma alternativa arquitetural a fim de alavancar o desempenho de futuras gerações de processadores. Técnicas como a *Previsão de valores* e o *Reuso de valores* foram apresentadas como formas de exploração de computação redundante.

A técnica de Reuso de valores, sendo objeto de investigação dessa pesquisa, foi mais amplamente discutida neste texto. Em linhas gerais, tal técnica visa memorizar e reusar os valores produzidos pelas instruções de uma aplicação alvo, e com o objetivo de investigar o reuso de computação redundante no contexto de aplicações Java, este trabalho apresentou um novo mecanismo de reuso no nível de traços, denominado *JDTM - Java Dynamic Trace Memoization.* Tal mecanismo foi implementado no simulador da arquitetura do processador FemtoJava Low Power 32 bits, sendo este um processador escalar Java com um pipeline de 5 estágios.

JDTM utiliza basicamente duas tabelas de memorização, implementadas na arquitetura do processador substrato e portanto a maior parte do custo de implementação é devido ao uso memória.

A efetividade do JDTM foi avaliada através da execução dirigida por simulação, de 8 aplicações típicas de sistemas embarcados Java. Pelos resultados, foi constatado que o JDTM provocou uma aceleração média de 1.12 e um percentual médio de reuso de 22.6%, utilizando 256 entradas na tabela J_Memo_Table_G e 1K entradas na tabela J_Memo_Table_T, ambas utilizando associatividade *full-way*.

Entre as principais observações e conclusões obtidas a partir dos experimentos, destacam-se:

- A princípio, os ganhos alcançados com JDTM são extremamente dependentes de tabelas com mais de 512 entradas e foram identificados dois conjuntos de aceleração, com mínimo de 1.01 em tabelas com 64 entradas e máximo de 1.09 em tabelas com 4K entradas. Entretanto, foi proposto e investigado um esquema para remarcação de *bytecodes* não redundantes. Neste caso alcançou-se aceleração de 1.10 mesmo com tabelas de apenas 64 entradas. Isto possibilitou alcançar elevados ganhos de aceleração e ao mesmo tempo reduzir a quantidade de memória que seria, no primeiro caso, necessária para alcançar tal desempenho.

- Com o objetivo de analisar o custo e a efetividade das tabelas de memorização, foram executados experimentos buscando obter a melhor opção em termos de tamanho das tabelas, chegando à conclusão que um maior número de entradas em J_Memo_Table_T (1K entradas) é preferível se comparado à J_Memo_Table_G (256 entradas). Foi observado também que esse efeito é inverso no caso do JDTM sem o esquema de remarcação de *bytecodes* não redundante e que portanto neste caso é preferível ter mais entradas em J_Memo_Table_G do que em J_Memo_Table_T. A explicação para esse efeito é que a ausência do esquema de remarcação impõem toda a responsabilidade de identificação de redundância no nível de *bytecodes*, sobre a J_Memo_Table_G. Ao contrário, com a presença do esquema de remarcação, grande parte da iden-

tificação de *bytecodes* redundantes é realizada pela monitoração dos valores empilhados, através do uso do *RBM (Registrador de Buffer de Monitoração).*

- As tabelas devem possuir associatividade *full-way*, independente do número de entradas que elas possuírem, pois a localidade de valores em aplicações Java não é concentrada em pequenos intervalos de tempo. Portanto, problemas de colisões nas tabelas são altamente prejudiciais ao reuso.

- O número de elementos no contexto de entrada dos traços reusados definem o número de portas de leitura necessárias no banco de registradores e o número de elementos comparados em paralelo no momento de uma busca por oportunidade de reuso. Neste caso, foi observado que 79.26% dos traços reusados possuíam apenas 2 elementos no contexto de entrada.

- O número de elementos no contexto de saída dos traços reusados definem o número de portas de escrita necessárias no banco de registradores e o número de elementos escritos ao mesmo tempo no banco. Neste caso 72.85% dos traços reusados possuíam apenas 1 elemento no contexto de saída.

- As maiores acelerações foram observadas na execução dos *benchmarks* CORDIC e Select Sort. Aproximadamente 95% dos traços reusados na execução do Select Sort possuíam 13 *bytecodes* e 50.3% dos traços reusados na execução do CORDIC possuíam mais de 15 *bytecodes*, o que, portanto, justifica o destacado desempenho.

- O número de desvios realizados incluídos nos traços reusados, são responsáveis pela redução das penalidades globais de desvio. Em média foi observado que aproximadamente 5% dos traços reusados possuíam de 2 a 3 desvios realizados e portanto o reuso destes traços ocultou a penalidade total desses desvios em apenas 3 ciclos.

A operação de identificação de oportunidades de reuso dos traços memorizados

pode impor um retardo no tempo de ciclo de *clock*, em vista da pesquisa associativa na J_Memo_Table_T. Porém, como neste caso o processador substrato utilizado neste trabalho se destina à execução de sistemas embarcados, e portanto possui baixa frequência de operação, tal retardo não se enquadra. Entretanto, para processadores com frequência de 1 GHz ou superior, seria necessário um estágio extra no *pipeline* para os acessos à J_memo_Table_T.

## 5.2 Trabalhos Futuros

Este foi o primeiro trabalho, que se tem conhecimento, objetivando investigar o comportamento de uma técnica de reuso de valores em aplicações Java. Portanto, várias oportunidades de investigação de novas alternativas são visualizadas. Entre elas:

- Implementação do JDTM em uma linguagem de descrição de *hardware* com o objetivo de analisar o impacto do mecanismo no tempo de ciclo de *clock*, na área ocupada no *chip*, bem como desenvolver um modelo de cálculo de potência para estimar o consumo de energia devido aos acessos às tabelas de memorização.

- Significante trabalho de engenharia é necessário para otimizar o JDTM e reduzir seu custo de implementação e potencial impacto no tempo de ciclo do processador.

- Desenvolver uma variação do JDTM, utilizando apenas a tabela de traços. Armazenando e identificando *bytecodes* redundantes pela consideração de traços de apenas 1 *bytecode*.

- Investigar o comportamento do JDTM em uma máquina virtual Java com a execução de aplicativos Java mais gerais.

- Incluir *bytecodes* de acesso à memória no domínio de *bytecodes* válidos para a construção de traços e projetar uma esquema que garanta a consistência entre a memória de dados e as tabelas de memorização.

- Analisar o comportamento do JDTM em um processador Java com preditor de desvios dinâmico, bem como fazer um investigação sobre o compromisso no uso de uma quantidade fixa de *bits*, dividida entre as tabelas de memorização e o preditor. Tal análise pode ser estendida para um processador Java com caches de dados e de instruções, identificando um compromisso na distribuição da quantidade fixa de memória entre as caches, o preditor e as tabelas do JDTM.

- Efetuar melhorias no nível de compilação dos programas Java, a fim de expor uma maior quantidade de redundância para o mecanismo JDTM. Poderia-se por exemplo realizar marcações no código, indicando os trechos com maior probabilidade de serem redundantes, bem como realizar reordenação de código.

# Referências Bibliográficas


[1] Altera Corporation. `http://www.altera.com`. Setembro. 2005.

[2] DA COSTA, A. T. *Explorando Dinamicamente o Reuso de Traces em Nível de Arquitetura de Processador.* PhD thesis, COPPE-UFRJ, Rio de Janeiro, Abril. 2001.

[3] DA COSTA, A. T., FRANÇA, F. M. G., FILHO, E. M. C. "The dynamic trace memoization reuse technique". In: *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pp. 92–99, Philadelphia, October. 2000.

[4] DE AQUINO VIANA, L. M. F. *Memorização Dinâmica de Traces com Reuso de Valores de Instruções de Acesso à Memória.* Tese de Mestrado, COPPE-UFRJ, Rio de Janeiro, Março. 2002.

[5] FILHO, A. C. S. B. *Uso da Técnica VLIW para Aumento de Performance e Redução do Consumo de Potência em Sistemas Embarcados Baseados em Java.* Tese de Mestrado, PPGC da UFRGS, Porto Alegre, Maio. 2004.

[6] FILHO, A. C. S. B., CARRO, L. "Low power Java processor for embedded applications". In: *Proceedings of 12th International Conference on Very Large Scale Integration*, pp. 239–244, Darmsdadt, December. 2003.

[7] FILHO, A. C. S. B., MATTOS, J. C. B., WAGNER, F. R., *et al.* "CACO-PS: A general purpose cycle-accurate configurable power simulator". In: *Proceedings*

*of 16th symposium on Integrated Circuits and Systems Design (SBCCI'03)*, pp. 349–354, São Paulo, September. 2003.

[8] GOMES, V. F., FILHO, A. C. S. B., CARRO, L. "A VHDL implementation of a low power pipelined Java processor for embedded applications". In: *Proceedings X Workshop IBERCHIP*, pp. 102–103, Cartagena de Indias, March. 2004.

[9] GONZALEZ, A., TUBELLA, J., MOLINA, C. "Trace-level reuse". In: *Proceedings of the 1999 International Conference on Parallel Processing*, pp. 30–37, Japan, September. 1999.

[10] HENNESSY, J. L., PATTERSON, D. A. *Computer Architecture: A Quantitative Approach.* 3 ed. San Franscisco, Morgan-Kaufmman, 2003.

[11] ITO, S. A., CARRO, L., JACOBI, R. P. "Designing a Java microcontroller to specific applications". In: *Proceedings of XII Symposium on Integrated Circuits and Systems Design*, pp. 12–15, Natal, September/October. 1999.

[12] ITO, S. A., CARRO, L., JACOBI, R. P. "System design based on single language and single-chip Java ASIP microcontroller". In: *Proceedings of Design Automation and Test in Europe*, pp. 703–707, Paris, 2000.

[13] LAURINO, L. S., DOS SANTOS, T. S. G., NAVAUX, P. O. A., *et al.* "Reuso de traços com loads em arquiteturas superescalares". In: *Anais do WSCAD 2005 - VI Workshop em Sistemas Computacionais de Alto Desempenho*, pp. 49–56, Rio de Janeiro, October. 2005.

[14] LIPASTI, M. H., SHEN, J. P. "Exceeding the dataflow limit via value prediction". In: *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pp. 226–237, Paris, December. 1996.

[15] Intel expands core concept for chips. `http://news.com.com/Intel+expands+core+concept+for+chips/2100-1006_3-54%94714.html`. December. 2004.

[16] PILLA, M. L. *Reuse through Speculation on Traces.* PhD thesis, II-UFRGS, Porto Alegre, Junho. 2004.

[17] PILLA, M. L., NAVAUX, P. O. A., CHILDERS, B. R., *et al.* "Value predictors for reuse through speculation on traces". In: *Proceedings of SBAC-PAD 2004 - 16th Symposium on Computer Architecture and High Performance Computing*, pp. 48–55, Foz do Iguaçu, October. 2004.

[18] RYCHLIK, B., SHEN, J. P. "Characterization of value locality in Java programs", *Workload characterization of emerging computer applications*, pp. 27–51, 2001.

[19] SILVA, B. R., ABREU, E. M., FRANÇA, F. M. G., *et al.* "JDTM - Memorização e reuso dinâmico de traços em uma arquitetura de processador Java". In: *Anais do WSCAD 2005 - VI Workshop em Sistemas Computacionais de Alto Desempenho*, pp. 57–64, Rio de Janeiro, October. 2005.

[20] SODANI, A., SOHI, G. S. "Dynamic instruction reuse". In: *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 194–205, Denver, June. 1997.

[21] SODANI, A., SOHI, G. S. "An empirical analysis of instruction repetition". In: *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pp. 35–45, San Jose, October. 1998.

[22] SODANI, A., SOHI, G. S. "Understanding the differences between value prediction and instruction reuse". In: *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pp. 205–215, Dallas, November. 1998.

[23] STROM, O., KLAUSEIE, A., AAS, E. J. "A study of dynamic instruction frequencies in byte compiled Java programs". In: *Proceedings of 25th EUROMICRO Conference*, pp. 232–235, Milano, September. 1999.

[24] VOLDER, J. "The CORDIC trigonometric computing technique", *IRE Trans. Electron. Comput.* v. EC-8, n. 3, pp. 330–334, 1959.