JDTM - Memorização e Reuso Dinâmico de Traços em uma Arquitetura de Processador Java

Bruno R. Silva, Eduardo M. Abreu
e Felipe M. G. França
Universidade Federal do Rio de Janeiro
COPPE - Sistemas e Computação
{brunors, melione, felipe}@cos.ufrj.br

Antônio C. S. Beck e Luigi Carro
Universidade Federal do Rio Grande do Sul
Instituto de Informática
{caco, carro}@inf.ufrgs.br

Resumo

JDTM - Java Dynamic Trace Memoization é um mecanismo implementado em uma arquitetura de processador Java, que realiza memorização e reuso dinâmico de traços de bytecodes redundantes. Para um conjunto de 8 programas típicos de sistemas embarcados, foi alcançada uma aceleração de 11% (média harmônica). Esta aceleração é justificada pela redução (i) do número de instruções executadas, (ii) dos caminhos críticos determinados por dependências verdadeiras e (iii) do número de penalidades devido aos desvios realizados.

1. Introdução

Bytecodes redundantes, ou seja, instâncias de *bytecodes* estáticos cujos operandos de entrada se repetem e portanto produzem o mesmo resultado, representam uma significativa porção dos *bytecodes* executados em aplicações Java [10]. Visando reusar os resultados produzidos por estes *bytecodes*, evitando sua re-execução, foi implementado uma versão do mecanismo *DTM - Dynamic Trace Memoization* [5] [4], que memoriza sequências dinâmicas de instruções[1] redundantes, *i.e.*, *traços redundantes*, e posteriormente reusa seus resultados atualizando o estado do processador de forma a evitar a re-execução destes traços. Como arquitetura substrato para esta pesquisa, foi utilizado o simulador *CACO-PS* do processador *FemtoJava Low Power 32 bits* [1]. Em um conjunto de 8 programas típicos de sistemas embarcados, foram identificados dois conjuntos distintos de acelerações: (i) 6 programas apresentaram acelerações variando de 0% à 6% e (ii) dois programas apresentaram acelerações de 35% e 66%.

1 Os termos *instrução* e *bytecode* serão usados indistintamente nas descrições de DTM e JDTM.

Este trabalho está organizado da seguinte forma. A Seção 2 descreve sucintamente o mecanismo DTM, originalmente concebido para uma arquitetura de três operandos. A arquitetura do processador FemtoJava Low Power 32 bits é apresentada na Seção 3. A Seção 4 introduz o mecanismo JDTM e suas principais diferenças em relação ao DTM original. A Seção 5 apresenta uma descrição dos experimentos realizados e resultados. Finalmente, a última seção apresenta as principais conclusões e trabalhos futuros.

2. DTM - Dynamic Trace Memoization

DTM é um mecanismo que busca construir, memorizar e reusar traços redundantes. Um traço é uma sequência dinâmica de instruções emitidas durante a execução de um programa. Um traço é redundante se e somente se for constituído de *instruções redundantes*.

DTM pode ser visto como um mecanismo de 3 fases implementando em *hardware*, que ocorrem paralelamente à execução de um programa alvo:

1. **Identificação construtiva de traços:** Para cada instrução buscada e decodificada, esta é pesquisada em uma tabela de instruções chamada *Memo Table G* (Tabela de Memorização Global), utilizando seu endereço de memória e valores dos operandos de entrada (conteúdo dos registradores fonte) como chave de pesquisa. Caso seja encontrada alguma correspondência, tal instrução é rotulada como *redundante*, caso contrário será *não-redundante* e inserida na tabela.
2. **Construção e Memorização de traços:** O *contexto de entrada* de um traço é definido como o conjunto de valores de operandos utilizados por instruções encapsuladas nos traços, mas que são produzidos por instruções fora do traço. O *contexto de saída* de um traço é o conjunto de resultados produzidos por instruções encapsuladas no traço (conteúdo dos registradores destino dessas instruções). A construção de um traço con-

siste em adicionar valores ao seu contexto de entrada e saída para cada instrução redundante encontrada. Toda instrução rotulada como redundante pelo passo anterior será usada para incrementar um traço em construção ou iniciar a construção de um novo traço. Uma instrução rotulada como não-redundante, finaliza a construção de um traço. Desta forma o traço construído e pronto para ser memorizado em uma outra tabela denominada *Memo Table T* (Tabela de Memorização de Traços), é constituído de uma sequência de instruções redundantes, rotuladas com base nas informações armazenadas em Memo Table G.

3. **Reuso de traços.** Paralelamente aos passos anteriores, o mecanismo faz uma busca por oportunidades de reuso de algum traço construído e memorizado em Memo Table T. Para isso, utiliza-se o endereço de memória de cada instrução buscada e decodificada a fim de realizar uma pré-seleção de traços candidatos à reuso. Para cada traço pré-selecionado, o mecanismo compara o contexto de entrada ao estado atual do processador. Em caso de correspondência, todas as instruções do traço selecionado não serão executadas. Assim, na fase de escrita de resultados, todo o contexto de saída do traço reusado é utilizado para atualizar o estado do processador, fazendo com que a semântica do fluxo de execução seja mantida.

3. FemtoJava Low Power 32 bits

O FemtoJava Low Power é um processador baseado em pilha, escalar *pipelined* com 5 estágios de execução e arquitetura *Harvard*, que executa *bytecodes* Java nativamente. Possui a pilha e conjunto de variáveis locais mapeados em um banco de 64 registradores,

Este processador pode ser visto como uma versão em silício [7] de uma máquina virtual Java com algumas limitações[2] [6]. O FemtoJava foi concebido originalmente como um processador multiciclo [8] e posteriormente sua versão *pipeline* [2] foi desenvolvida. O núcleo do processador FemtoJava faz parte de um plataforma para desenvolvimento de aplicações embarcadas baseadas em Java [9].

Os 5 estágios do *pipeline* são descritos abaixo:

1. **IF** - *Instruction Fetch*: Composto por uma fila de instruções de 9 registradores de 1 byte, denominada fila de *prefetch*, que ao possuir pelo menos 4 posições livres recebe uma palavra de 4 bytes da memória de instruções, indexada pelo contador de programas.
2. **ID** - *Instruction Decoder*: Responsável por gerar a palavra de controle e informar à fila de prefetch o tamanho da instrução atual. Isto é necessário devido ao tamanho das instruções ser variável.
3. **OF** - *Operand Fetch*: A busca de operandos é realizada em um banco de registradores de tamanho variável, definido a *priori* nos estágios anteriores do projeto do sistema embarcado. Dois registradores de uso restrito são utilizados: VARS, que referencia o início do conjunto de variáveis locais de um método e SP, que referencia o topo da pilha.
4. **EX** - *Execution*: Após o término da busca de operandos, os mesmos são enviados ao estágio de execução que seleciona a unidade funcional adequada para a execução do *bytecode* atual. Como o processador FemtoJava não possui mecanismo de previsão dinâmica de desvios, é assumido um esquema de previsão estática, onde desvios condicionais são considerados, por *default*, como não realizados. Caso a previsão esteja incorreta, há uma penalidade de 3 ciclos de clock.
5. **WB** - *Write Back*: Neste estágio o resultado obtido na fase de execução é escrito no banco de registradores, utilizando VARS ou SP como endereço base, somado ao índice da variável local ou índice da pilha respectivamente. O processador FemtoJava não permite uma leitura e uma escrita simultânea no banco de registradores, logo, uma bolha é inserida no Pipe caso uma instrução no quinto estágio tente escrever enquanto outra no terceiro estágio espera ler.

O processador FemtoJava possui um esquema de *forwarding* de resultados do estágio de execução e *write back* para o estágio de busca de operandos. Desta forma, dependências verdadeiras são resolvidas mais eficientemente. Sem este mecanismo, seria necessário inserir bolhas no *pipeline* até que a instrução que produz o resultado termine sua escrita no banco de registradores.

4. JDTM

Denominamos aqui como JDTM - *Java Dynamic Trace Memoization*, um mecanismo para memorização e reuso dinâmico de traços de *bytecodes* Java redundantes. O modelo conceitual Java corresponde a uma arquitetura de zero operandos[3], fazendo com que algumas características do DTM original fossem modificadas para atender à sintaxe e semântica das aplicações Java.

Os principais aspectos para a operação do JDTM são: (i) como identificar *bytecodes* redundantes; (ii) como utilizar a redundância de *bytecodes* para a construção de tra-

2 Não possui alocação dinâmica de memória e cobre um conjunto de instruções reduzido salvando área e potência, visto que destina-se ao mercado de sistemas embarcados.

3 Também chamado de máquina de pilha.

Mnemônico	Classe
iadd, imul, ineg, isub, iand, ior, ixor, ishl, ishr, iushr	Lógica/Aritm.
goto, if_icmpeq, if_icmpge, if_icmpgt, if_icmle, if_icmlt, if_icmpne, ifeq, ifge, ifgt, ifle, iflt, ifne	Contr. de Fluxo
istore, istore_0, istore_1, istore_2, istore_3, iload, iload_0, iload_1, iload_2, iload_3	Load/Store
bipush, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, iconst_m1, dup, pop, sipush	Op. de pilha
iinc	Outras

Tabela 1. *Bytecodes* válidos para o JDTM.

ços redundantes e (iii) como identificar traços redundantes e reusá-los. Estes aspectos serão detalhados a seguir.

4.1. Identificação de *bytecodes* redundantes

Para cada *bytecode* buscado e decodificado, JDTM verifica se ele faz parte do conjunto de *bytecodes válidos*[4] para a construção de traços redundantes (Tabela 1). Um *bytecode* válido, na fase de busca de operandos (*Operand Fetch*), é pesquisado em uma tabela de memorização de *bytecodes* (Memo Table G), utilizando o PC e o(s) valor(es) de seu(s) operando(s) como chave de busca. Em caso de sucesso, este *bytecode* é rotulado como redundante e caso contrário será rotulado como não-redundante. *bytecodes* não-redundantes são inseridos na Memo Table G na fase de execução, onde cada entrada desta tabela é composta pelos valores do: PC; operando(s) de entrada (máximo 2 operandos) e o resultado produzido. Um fato notável é que os *bytecodes bipush, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, iconst_m1* e *goto* serão sempre redundantes, visto que manipulam constantes. Logo estes *bytecodes* serão automaticamente rotulados como tal e nunca serão inseridos em Memo Table G.

4.2. Construção de traços redundantes

Como definido na Seção 2, JDTM considera um traço redundante como sendo uma sequência de *bytecodes* redundantes e pertencentes ao conjunto de *bytecodes* válidos para o JDTM.

Assim como no mecanismo DTM, JDTM possui um contexto de entrada e um contexto de saída. O contexto de entrada é definido como o conjunto de valores utilizados por *bytecodes* pertencentes ao traço, mas que foram produzidos por *bytecodes* fora do traço. O contexto de saída é o conjunto de valores produzidos pelos *bytecodes* pertencentes ao traço. Tanto o contexto de entrada quanto o contexto de saída podem ser classificados em 2 tipos:

1. **Contexto referente ao *pool* de variáveis locais:** Considerando o contexto de entrada, será o conjunto de valores carregados pelos *bytecodes iload, iload_0, iload_1, iload_2, iload_3* e *iinc* do *pool* de variáveis locais. Considerando o contexto de saída será o conjunto de valores armazenados pelos *bytecodes istore, istore_0, istore_1, istore_2, istore_3* e *iinc* no *pool* de variáveis locais.
2. **Contexto referente à pilha:** Considerando o contexto de entrada será o conjunto de valores consumidos da pilha, mas que foram empilhados por *bytecodes* que estão fora do traço. Considerando o contexto de saída será o valor (se houver) produzido e empilhado por algum bytecode pertencente ao traço, mas que não foi consumido por nenhum *bytecode* incluso no traço em questão.

4.2.1. Construção dos contextos de entrada e saída Para a inserção de valores no contexto de entrada e saída durante a construção de um traço, são utilizadas 4 estruturas auxiliares:

1. **Mapa de bits do Contexto de Entrada:** Possui B bits (parâmetro arquitetural indicando o número máximo de valores que podem ser armazenados no contexto), onde o bit x é relacionado à variável local x ($0 \geq x < B$). Ex. O valor 1 no bit 0 significa que a variável local 0 faz parte do contexto de entrada. Logo, se um próximo *bytecode* a ser inserido no mesmo traço utilizar o valor da variável local 0, isto não implicará na inclusão deste valor no contexto de entrada, visto que o mesmo já faz parte de tal contexto.
2. **Mapa de bits do Contexto de Saída:** Possui B bits, onde o bit x é relacionado à variável local x ($0 \geq x < B$). Ex. O valor 1 no bit 3 significa que a variável local 3 faz parte do contexto de saída. Portanto, se um próximo *bytecode* a ser inserido no mesmo traço utilizar o valor da variável local 3, isto não implicará na inclusão deste valor no contexto de entrada, visto que o mesmo foi produzido por um *bytecode* pertencente ao traço.
3. **Contador de Pilha Local:** É um contador inicializado com 0 sempre que a construção de um novo traço é iniciada. O contador é incrementado e decrementado conforme os valores são empilhados e desempilhados na/da pilha. Sempre que houver a necessidade de decrementar este contador quando o mesmo é igual 0, significa que o topo e/ou subtopo atual deve ser incluído no contexto de entrada referente à pilha. Por exemplo: Se o contador é igual 1, pode-se dizer que

4 Instruções de acesso a memória não são válidas, visto a necessidade de um mecanismo para garantir a consistência da memória.

até o momento foi empilhado um valor que ainda não foi consumido. Neste caso, se a próxima instrução a ser incluída no traço necessitar de 2 operandos presentes na pilha (ex. *if_icmpeq, iadd*), um destes operandos (o subtopo) fará parte do contexto de entrada do traço em questão e o contador será decrementado uma vez, visto que ele nunca atinge valores negativos.

4. **Contador de Pilha Global:** Segue o mesmo comportamento da estrutura anterior, porém pode armazenar valores negativos. Ao fim da construção de um traço o valor desse contador é transferido para o campo *updateSp* do *Buffer de construção* (Subseção 4.2.2) que indica o valor à ser somado ao registrador SP do processador, quando o traço em questão é reusado.

4.2.2. *Buffer* de construção: É constituído pelos campos pc e npc, que possuem respectivamente o endereço de memória do primeiro *bytecode* do traço e o endereço da próxima instrução a ser executada para o caso em que o traço seja reusado. Além destes, o *Buffer* de construção armazena também os valores do contexto de entrada nos campos $icv_0, ..., icv_{B-1}$, enquanto os correspondentes índice das variáveis locais ou posições da pilha são armazenadas nos campos $icr_0, ..., icr_{B-1}$. Seguindo o mesmo esquema, o *Buffer* de construção armazena os valores do contexto de saída nos campos $ocv_0, ..., ocv_{B-1}$ e os correspondentes índices das variáveis locais ou posição da pilha nos campos $ocr_0, ..., ocr_{B-1}$. A fim de diferenciar quando um determinado valor é referente à pilha ou ao conjunto de variáveis locais, foi adicionado, tanto no contexto de entrada como no contexto de saída, um bit[5] (campos rachurados do *Buffer* de construção exibido na Figura 3) ao início de cada campo icr_i e ocr_i, $i = 0, ..., B-1$. O *Buffer* ainda possui um campo chamado *updateSp*, que informa o valor que deve ser somado ao registrador SP, refletindo o estado da pilha (após a construção, o tamanho da pilha pode ter sido reduzido, aumentado ou inalterado) após o reuso do traço; o campo $branch$ informa se o traço incorpora pelo menos 1 desvio tomando, e é utilizado para determinar a necessidade ou não de esvaziar a fila de *prefetch* do processador em caso de reuso. E por fim, o *Buffer* de construção possui o campo $maskInib$ sendo uma máscara, que informa o deslocamento que deve ser realizado na fila de prefetch em caso de reuso de um traço que não encapsula bytecode de desvio realizado.

4.2.3. Exemplo de construção de um traço A Figura 1 apresenta o exemplo de um traço que pode ser identificado como redundante, pois os *bytecodes* que o compõe são redundantes. Traços redundantes no JDTM são delimitados

5 Ao receber o valor 1, o bit informa que o valor em questão é referente ao conjunto de variáveis locais. Por outro lado, ao receber 0 indica que é referente à pilha.

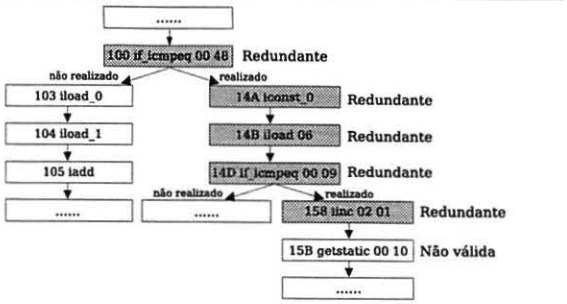

Figura 1. Exemplo de um traço redundante.

por *bytecodes* não redundantes ou não pertencentes ao conjunto de *bytecodes* válidos.

Para cada *bytecode* redundante da Figura 1, a Figura 2 mostra como os mapas de bits do contexto de entrada/saída e os contadores de pilha Local e Global mudam dinamicamente durante a construção de um traço:

- O *bytecode if_icmpeq* realiza um desvio para o endereço de memória 0x14A (0x100 + 0x02 + 0x0048), caso o topo e subtopo da pilha sejam iguais. Visto que tal *bytecode* consome 2 valores da pilha, o Contador de Pilha Global será decrementado em 2 unidades, e o Contador de Pilha Local não será decrementado, pois o mesmo já possui o valor 0. Ambos os valores do topo e subtopo foram produzidos por *bytecodes* não pertencentes ao traço em questão, pois o Contador de Pilha Local, se decrementado de 2 unidades, seria igual a -2. Portanto o topo e sub-topo atual devem fazer parte do contexto de entrada referente à Pilha, e os campos icr_0 e icr_1 devem receber os valores 0 e 1 respectivamente. Os campos icv_0 e icv_1 receberão os valores 42 e 42 que equivalem ao valor do topo e subtopo da pilha para este caso. A execução desse *bytecode* altera a fluxo sequencial do programa e portanto o campo *branch* do *Buffer* de construção recebe o valor 1. Visto que o *bytecode* não manipula valores do *pool* de variáveis locais, os mapas de bits do contexto de entrada/saída não serão atualizados.
- O *bytecode iconst_0* empilha a constante 0, fazendo com que o ambos os contadores de pilha (Local e Global) sejam incrementados em 1 unidade.
- O *bytecode iload 06* carrega o valor da variável local 6 e o empilha na posição SP+1 da pilha. Como já dito, o *Mapa de bits do contexto de entrada* tem a função de informar quais variáveis locais já fazem parte do contexto de entrada do traço em construção. O *Mapa de bits do contexto de saída* informa quais variáveis locais foram produzidas por *bytecodes* pertencentes ao traço, ou seja, que fazem parte do contexto de saída. Portanto após a execução desse *bytecode*, o bit 06 do

Mapa bits do contexto de entrada recebe o valor 1 e os contadores de pilha Local e Global são novamente incrementados em 1 unidade. Os campos icr_2 e icv_2 do *buffer* de construção recebem os valores 6 (variável local 6) e 0 (valor da variável local 6) respectivamente.

- O *bytecode if_icmpeq* desempilha os dois últimos valores empilhados respectivamente pelos *bytecodes iconst_0* e *iload 06*. Os contadores de pilha Local e Global serão decrementados em 2 unidades. Porém neste caso o Contador de Pilha Local 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 iinc 02 01*, 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 da variável local 02. Ao consultar os Mapas de bits do contexto de entrada e saída, observa-se que o valor da variável local 02 ainda não faz parte do contexto de entrada e não foi produzido por *bytecode* pertencente ao traço. Logo o bit 02 de ambos os mapas recebem o valor 1 e o valor da variável local (valor 8 neste caso) será utilizado para atualizar o Contexto de entrada (campos icr_3 e 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 getstatic 00 10* carrega o valor de uma posição de memória e não faz parte do conjunto de *bytecodes* válidos para o JDTM, portanto este *bytecode* finaliza a construção do traço, fazendo com que: o valor do campo *npc* do *Buffer* de construção seja igual ao seu endereço de memória; o valor do Contador de Pilha Global seja transferido para o campo *updateSp* e seja alocada uma entrada em Memo Table T (com estrutura idêntica ao *Buffer* de Construção exibido na Figura 3) para receber o traço construído. É importante dizer que o campo *maskInib* é atualizado a cada novo *bytecode* incluído no traço. E nesse caso possuirá a máscara 11111 referente à 5 *bytecodes*.

4.3. Identificação e Reuso de traços redundantes

Paralelamente à construção dos traços, o mecanismo JDTM busca por oportunidades de reuso dos traços já construídos e memorizados. A seguir, descrevem-se os passos necessários para identificar uma possibilidade e realizar um reuso:

1. Para cada *bytecode* buscado, decodificado e com seu(s) operando(s) disponível(s), o endereço de memória deste *bytecode* é comparando ao campo *pc* de cada entrada da Memo Table T (assumindo, neste caso, uma tabela completamente associativa). Desta forma, alguns traços, podem ser pré-selecionados.
2. Para cada traço pré-selecionado, o seu contexto de entrada é comparado ao estado atual do processador.
   (a) Em caso de alguma correspondência, o *bytecode* não será executado e no estágio *write back* do *pipeline*, o estado do processador é atualizado conforme o contexto de saída do traço selecionado para reuso. Também são atualizados o valor do registrador SP, o valor do contador de programas (PC) e a fila de prefetch é deslocada ou esvaziada.
   (b) Caso contrário, o *bytecode* segue seu fluxo normal de execução.

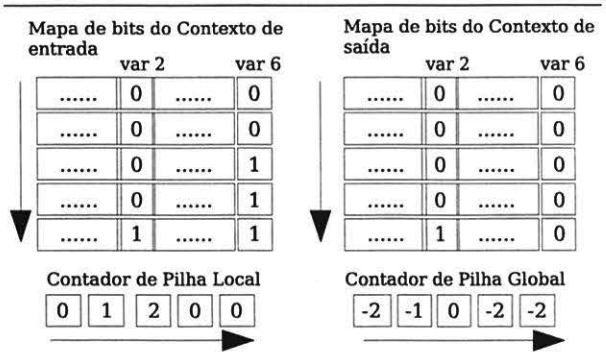

Figura 2. Mapa de bits do contexto de entrada/saída.

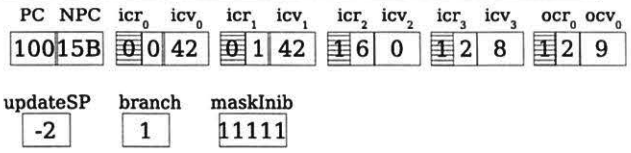

Figura 3. *Buffer* de Construção contendo o traço da Figura 1.

5. Experimentos e Resultados

5.1. Ambiente de simulação

Para efetuar os experimentos de avaliação do mecanismo proposto, foi utilizado o simulador CACO-PS [3]. Este é um simulador que possui um princípio semelhante ao SystemC

Benchmark	Descrição
Senos e Cosenos	Cálculo de senos e cosenos
Select Sort	Algoritmo de ordenação
Bubble Sort	Algoritmo de ordenação
Busca Binária	Busca binária em um array
Busca Sequencial	Busca sequencial em um array
IMDCT	Algoritmo de descompressão mp3
Float point	Várias operações de ponto flutuante
Mp3 player	Um mp3 player

Tabela 2. Oito *Benchmarks* típicos de aplicações embarcadas.

[11], podendo fornecer informações quantitativas da arquitetura descrita.

5.2. Métrica

As seguintes equações determinaram os valores obtidos nas simulações:

O percentual de aceleração de desempenho é dado por:

$$aceleracao = ciclos_{base}/ciclos_{JDTM} \quad (1)$$

Onde $ciclos_{base}$ e $ciclos_{JDTM}$ são considerados respectivamente para um processador não incluindo e incluindo o mecanismo JDTM.

A média é dada pela seguinte expressão:

$$HM = n(\sum_{i=1}^{n}(1/S_i))^{-1} \quad (2)$$

S_i indica o elemento a ser considerado pela média.

5.3. *Benchmarks*

Foram utilizados 8 *benchmarks* típicos de aplicações embarcadas. A Tabela 2 apresenta a descrição de cada *benchmark*: Cálculo de senos e cosenos, como uma representativa biblioteca aritmética; Ordenação e Pesquisa, usados em escalonadores; IMDCT (*Inverse Modified Discrete Cosine Transformation*), uma parte importante do algoritmo de descompactação MP3; *Float Operation*, uma biblioteca para emular soma de números em ponto flutuante, visto que o processador FemtoJava não possui unidade de ponto flutuante e um MP3 player completo como uma aplicação muito representativa de sistemas embarcados.

5.4. Resultados

A Figura 4 exibe a aceleração média alcançada com JDTM, variando o tamanho e a associatividade[6] das tabelas de memorização. Observa-se que o mecanismo é sensível à associatividade e ao tamanho das tabelas. Com essa primeira implementação, foi alcançada aceleração máxima em média harmônica de 9%, utilizando 512 entradas e associatividade full-way em ambas as tabelas. Tabelas maiores que 512 entradas não oferecem um aumento significativo do ganho.

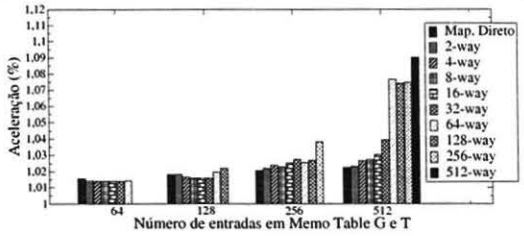

Figura 4. Aceleração média alcançada com o JDTM.

5.4.1. Esquema de re-rotulação de *bytecode* não redundantes O esquema de re-rotulação consiste em otimizar a construção de traços, de forma a reduzir a associatividade e/ou o tamanho das tabelas, sem com isso perder um porção significativa de aceleração. Tal otimização se baseia no fato que, instruções cujo operandos foram produzidos por instruções rotuladas como redundantes, são também instruções redundantes. Ou seja, mesmo que tais instruções não estejam presentes na Memo Table G, elas são rotuladas como redundantes e podem iniciar a construção de um novo traço ou incrementar um traço em construção. Para isso o mecanismo foi ligeiramente modificado de forma a monitorar todos os valores empilhados e armazenados na pilha e no *pool* de variáveis locais (rotulando *todos* os *valores* como redundante ou não redundante, se estes foram produzidos por *bytecodes* redundantes ou não redundantes respectivamente). A Figura 5 apresenta os ganhos alcançados com esta otimização.

A construção de traços é extremamente dependente de acertos na Memo Table G. A otimização proporcionou uma melhor forma de construir traços, mesmo com tabelas extremamente pequenas. Com apenas 64 entradas, conseguiu-se alcançar a mesma aceleração (9% em média harmônica) da versão anterior do JDTM com tabelas de 512 entradas, além de uma aceleração máxima ligeiramente maior (perto

6 Foi utilizado a política de substituição FIFO (First In, First Out) nas tabelas associativas.

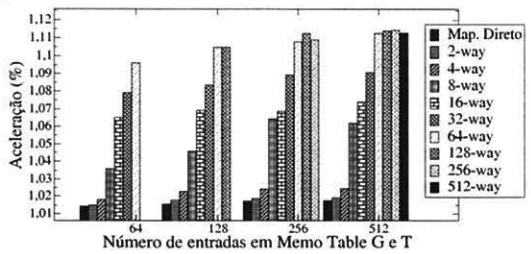

Figura 5. Aceleração alcançada com a versão otimizada do JDTM.

de 11%). Ou seja, com a otimização e apenas 1/8 do tamanho das tabelas da versão anterior do JDTM, atingiu-se a mesma aceleração.

As Figuras 6 e 7 exibem a aceleração alcançada por cada *benchmark*, utilizando tabelas completamente associativas (full-way) e variando de 64 à 512 entradas.

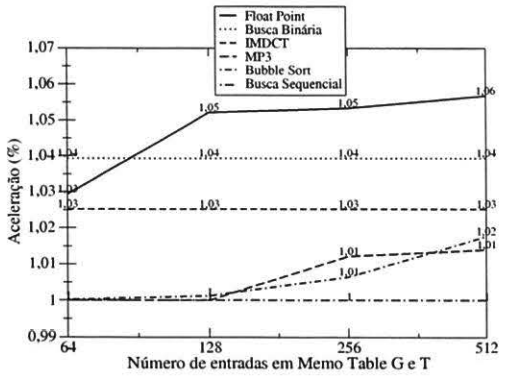

Figura 6. Aceleração com associatividade full-way.

A aceleração alcançada pelo JDTM nos *benchmarks* IMDCT (3%) e Busca binária (4%) se mantém constante mesmo com o aumento do número de entradas. Isso leva a crer que a redundância de tais *benchmarks* é totalmente explorada com apenas 64 entradas.

É notável que a execução do *benchmark Busca Sequencial* não tenha alcançado aceleração. Isso é um *efeito colateral* do esquema de re-rotulação de *bytecodes* não redundantes (Subseção 5.4.1) somado ao pressuposto de que *bytecodes* que manipulam constantes serão sempre rotulados como redundantes. O trecho de código abaixo, que compõem o *benchmark* de busca sequencial, exemplifica tal efeito: assumindo JDTM sem o esquema de re-rotulação, o único traço redundante é composto pelos *bytecodes 57 a 59*:

```
43 : iconst_0
44 : istore_1
45 : goto 00 0f
.........
53 : iinc 01 01
56 : iload_1
57 : iload_2
58 : iconst_1
59 : isub
5a : if_icmple ff ec
```

Assumindo agora JDTM com esquema de re-rotulação e considerando a primeira execução deste trecho de código: a instância do *bytecode 43* é constante, portanto ele é rotulado como redundante. O *bytecode 44* será redundante também (usando o esquema de re-rotulação), pois opera sobre o valor produzido pelo *bytecode 43*. O próximo *bytecode* também é rotulado como redundante (desvio constante) e realiza um desvio para o *bytecode 56*, que também será redundante (novamente utilizando o esquema de re-rotulação), porque opera sobre o valor produzido pelo *bytecode 44*. O *bytecode 57* finaliza a construção do traço, visto que não está presente em Memo Table G (é a primeira execução) e não opera sobre valor produzido por *bytecode* redundante (não será re-rotulado). Porém, analisando esse código estaticamente, verifica-se que o traço construído jamais será reusado, porque o *bytecode 43* não é alcançado por um desvio, logo só foi executado 1 vez e iniciou a construção de um traço devido ao fato de ser assumido como sempre redundante.

Continuando a execução, o *bytecode 5a* efetua um desvio para o *bytecode 48*, e seguindo o fluxo de execução possível até que o *bytecode 53* seja reexecutado, o valor da variável local 1 não é alterado (mantendo-se como um valor produzido pelo *bytecode* redundante *44*). Logo o *bytecode 53*, ao ler este valor, inicia a construção de um traço, produzindo um novo valor para a variável local 1, que será marcado como redundante. Este traço irá encapsular o traço *57 a 59*, impedindo o seu reuso. Como resultado, sempre que o *bytecode 5a* realizar o desvio, um novo traço será criado e nunca será identificada a redundância do traço *57 a 59*.

As Figuras 8 e 9 exibem a aceleração alcançada por cada *benchmark*, utilizando 512 entradas em ambas as tabelas (Memo Table G e Memo Table T) e variando a associatividade (mapeamento direto à full-way).

6. Conclusão e Trabalhos Futuros

Várias pesquisas têm se concentrado no aumento da frequência de operação e/ou despacho de múltiplas instruções explorando o Paralelimos no nível de instruções. *Dynamic Trace Memoization* concentra todo esforço em reaproveitar resultados já calculados, evitando assim a reexecução de longos traços de instruções redundantes. Desta forma, JDTM ao fazer uso da redundância implícita de programas Java no nível de traços de *bytecode*, consegue reduzir as penalidades de desvios realizados, visto que vá-

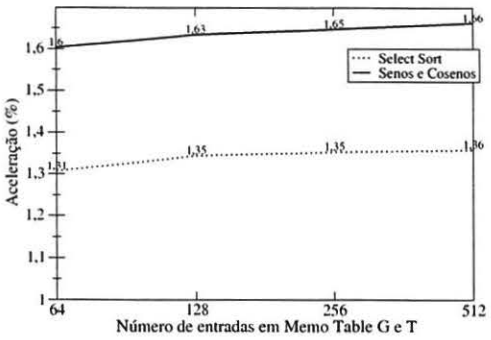

Figura 7. Aceleração com associatividade full-way (Continuação).

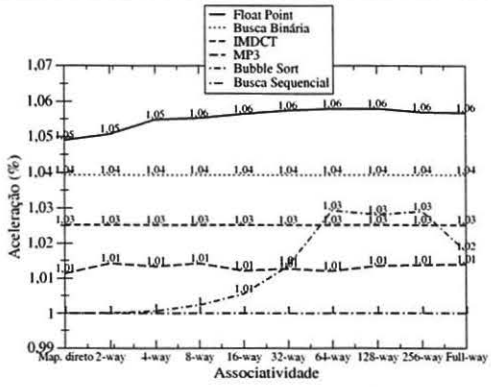

Figura 8. Aceleração com 512 entradas.

rias instruções de desvio podem estar encapsuladas nos traços memorizados; quebrar dependências verdadeiras dentro dos traços memorizados, pois estas são eliminadas ao efetuar um reuso e por fim a redução do número de instruções executadas. Está sendo implementado a versão em VHDL do mecanismo JDTM no processador FemtoJava, com a finalidade de se verificar alterações no velocidade de Clock e área ocupada, bem como compromissos no uso de memória das tabelas de memorização versus introdução de caches de instruções e dados.

Agradecimentos

Os autores agradecem o suporte financeiro do CNPq e CAPES. Esta pesquisa foi desenvolvida junto ao Núcleo de Atendimento em Computação de Alto Desempenho da COPPE/UFRJ.

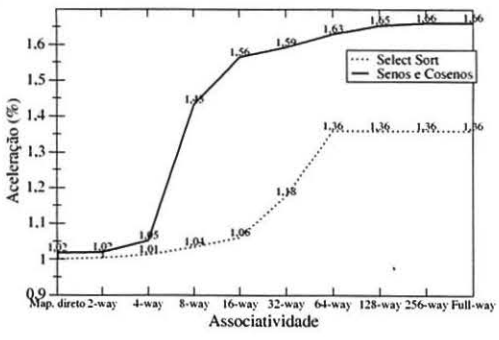

Figura 9. Aceleração com 512 entradas (Continuação).

Referências

[1] A. C. S. Beck. Uso da Técnica VLIW para aumento de Performance e Redução do consumo de potência em Sistemas Embarcados Baseados em Java. Master's thesis, PPGC da UFRGS, Porto Alegre, 2004.

[2] A. C. S. Beck and L. Carro. Low Power Java Processor for Embedded Applications. In *IFIP 12th International Conference on Very Large Scale Integration, Germany*, December 2003.

[3] A. C. S. Beck, J. C. B. Mattos, F. R. Wagner, and L. Carro. CACO-PS: a general purpose cycle-accurate configurable power simulator. In *16th Symposium on Integrated Circuits and Systems Design*, pages 349–354, 8-11 Sept. 2003.

[4] A. T. da Costa. *Explorando dinamicamente o reuso de Traces em nível de arquitetura de processador*. PhD thesis, (Engenharia de Sistemas e Computação) - Universidade Federal do Rio de Janeiro, 2001.

[5] A. T. da Costa, F. M. G. França, and E. M. C. Filho. The Dynamic Trace Memoization Reuse Technique. In *The International Conference on Parallel Architectures and Compilation Techniques - PACT 2000*.

[6] M. W. El-Kharashi and F. Elguibaly. Java microprocessors: Computer architecture implications. In *1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM'97)*, pages 277–280, Victoria, BC, Canada, Aug. 20-22 1997.

[7] D. S. Hardin. Crafting a Java virtual machine in silicon. *IEEE Instrumentation & Measurement Magazine*, 4:54–56, Mar 2001.

[8] S. A. Ito, L. Carro, and R. P. Jacobi. Designing a Java microcontroller to specific applications. In *XII Symposium on Integrated Circuits and Systems Design, 1999. Proceedings.*, pages 12–15, 29 Sept.-2 Oct. 1999.

[9] S. A. Ito, L. Carro, and R. P. Jacobi. Making Java Work for Microcontroller Applications. *IEEE Design and Test of Computers*, 18(5):100–110, September/October 2001.

[10] B. Rychlik and J. P. Shen. Characterization of value locality in Java programs. pages 27–51, 2001.

[11] SystemC.org. Systemc. http://www.systemc.org/.