UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

BRUNO SILVEIRA NEVES

Gerência Dinâmica de Memória em Aplicações Java Embarcadas

Dissertação apresentada como requisito parcial para a obtenção de grau de Mestre em Ciência da Computação

Prof. Dr. Luigi Carro
Orientador

Porto Alegre, junho de 2005

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Neves, Bruno Silveira
Gerência Dinâmica de Memória em Aplicações Java Embarcadas / Bruno Silveira Neves - Porto Alegre: PPGC da UFRGS, 2005.
109 f.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR - RS, 2005. Orientador: Luigi Carro.

1. Gerência Dinâmica de Memória. 2. Garbage Collection. 3. Java. 4. Sistemas Embarcados. 5. Sistemas de Tempo Real. I. Carro, Luigi. II. Título

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitor de Pós-Graduação: Prof[a]. Valquiria Linck Bassani
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Flávio Rech Wagner
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço primeiramente ao povo brasileiro que através da UFRGS disponibiliza uma completa infra-estrutura de recursos para que todo o esforço por mim realizado tivesse o amparo tecnológico adequado ao alcance.

Agradeço ao meu orientador Luigi Carro pela oportunidade fornecida e pela confiança depositada em mim para o desenvolvimento deste trabalho. Sua atitude certamente me manteve mais uma vez motivado para exercer um trabalho com alegria e perspectivas para um crescimento contínuo.

Agradeço ao colega Júlio Mattos por dividir o conhecimento com humildade e consciência de que não deveríamos ajudar somente aqueles que podem nos retribuir, mas sim aqueles que de fato precisavam de ajuda.

Agradeço aos amigos Gustavo Pereira, André Bastos, Rubinei Ângelo, Daniel Ferrão e Márcio Oliveira, pelo companheirismo e incentivo, pois foram companheiros sinceros durante todo o longo caminho percorrido até que esta dissertação estivesse hoje à disposição de você leitor.

Agradeço ao professor e amigo Luciano Agostini, meu orientador durante o curso de graduação, por me apoiar sempre com respeito e humildade única e por transmitir com experiência e honestidade as circunstâncias positivas e negativas que seriam por mim encontradas durante o curso de mestrado.

Agradeço a minha noiva Mariléia Rosa pelo apoio valioso para que eu fosse capaz de me reerguer sempre que algum obstáculo mais difícil me confrontasse durante este caminho.

Agradeço a minha irmã Cheron Neves pela presença importante em minha vida, através da qual pude compartilhar muitos momentos de amor, possibilitando um desenvolvimento ainda mais feliz.

Agradeço em especial aos meus pais Antônio e Sandra Neves pelo amor, sustento e educação fornecidos para que eu pudesse me tornar com orgulho o homem que sou hoje. Sem dúvida alguma, toda a disciplina e motivação que tenho para aprender e ensinar foram em mim plantados através destas duas pessoas.

Agradeço da mesma forma aos meus avós Margarida e Natálio Silveira pelo amor e educação concedidos durante meu crescimento. De forma especial, agradeço a ti meu avô que fisicamente já não esta presente desde fevereiro de 2004, mas cujo amor e respeito por mim ficarão guardados eternamente em meu coração.

Dedico este trabalho a todos os meus familiares e em especial ao meu avô que possibilitou durante toda sua vida que nossa família fosse unida e feliz

O caminho

Fui! Vim!

Subi! Desci!

Às vezes com força. Às vezes cansado.
Mas o importante é que eu queria:
Ir...! Vir...!
Assim passava o tempo...
Na subida encontrava alguém descendo
Na descida encontrava alguém subindo
Mas alegrava-me sempre, porque em geral não subia sozinho, muito menos descia sozinho!
Sempre tinha alguém indo, alguém vindo...

Teve momentos em que pude até conversar
Outros...
Conversei comigo mesmo!
Entre os dois prefiro momentos que conversei.
Teve alguém que sorriu!
Teve alguém que nem me olhou!
Foram poucos talvez...

Aprendi a observar melhor quem passa por mim
Não tenho nenhuma garantia
Que voltarei a encontrá-lo
Por isso, embora, mesmo não conversando quero olhar seu rosto
Pois tenho certeza...
Pode estar refletindo a mesma minha angústia. Melhor ainda, pode ser a pessoa com quem poderei colaborar para torná-la mais feliz.
Pode ser a pessoa que me trará felicidade
Ou então...
Posso também mostrar a ela, minha felicidade.
Isso é tão bom de dividir! Compartilhar!

Sandra Mara Silveira Neves

SUMÁRIO

LISTA DE FIGURAS **15**

LISTA DE TABELAS **17**

RESUMO **19**

ABSTRACT **21**

1 INTRODUÇÃO **23**

1.1 A linguagem Java **23**

1.2 Sistemas Embarcados e Tempo Real **25**

1.3 Java em Sistemas Embarcados **26**

1.4 A Arquitetura FemtoJava e as Metas para este Trabalho **28**

2 GERÊNCIA DINÂMICA DE MEMÓRIA **31**

2.1 *Garbage Collection* **31**
2.1.1 Mark-sweep 32
2.1.2 Copying 32
2.1.3 Reference Counting 32

2.2 Variações dos Algoritmos **33**

2.3 Comparação dos Algoritmos Clássicos **33**

2.4 Propostas Existentes **34**
2.4.1 Propostas Clássicas 34
2.4.2 Propostas Modernas e Atuais 37

3 DESCRIÇÃO DAS SOLUÇÕES PROPOSTAS **53**

3.1 As arquiteturas de Base **53**

3.2 Implementação 1 - Baseada no Algoritmo de Lin **55**

3.2.1 Resolução da *Constant Pool*.....58
3.2.2 Comentários sobre a Implementação.....60

3.3 Implementação 2 - Baseada no Algoritmo de Ritzau 60

3.4 Integração à Arquitetura Alvo 62
3.4.1 A Ferramenta de Adaptação de Código (CAT).....64

4 RESULTADOS..... 71

4.1 Estimativas Quanto ao Uso da Memória 72
4.1.1 Address Book 73
4.1.2 Mp3 Player.....77
4.1.3 Sokoban Game 81

4.2 Estimativas em desempenho, energia e potência..... 85
4.2.1 Consumos para as Operações Básicas.....86
4.2.2 Address Book 89
4.2.3 Mp3 Player.....91
4.2.4 Sokoban Game 96

4.3 Estimativas Finais sobre Este Trabalho..... 97
4.3.1 Impactos sobre o Conjunto de Instruções da Arquitetura 97
4.3.2 Código Desenvolvido 98
4.3.3 Nota sobre Utilização das Implementações 99

5 CONCLUSÕES..... 101

5.1 Propostas de Trabalhos Futuros 101

REFERÊNCIAS..... 105

ANEXO A DETALHAMENTO DE CUSTOS MP3 PLAYER 113

ANEXO B CÓDIGO DESENVOLVIDO..... 115

LISTA DE ABREVIATURAS

AMP	*Active Memory Processor*
BIT	*Bytecode Instrumenting Tool*
CACO-PS	*Cycle Accurate Configurable-Power Simulator*
CAT	*Code Adapting Tool*
CBT	*Complete Binary Tree*
CC	*Class and Counter*
CPU	*Central Processing Unit*
DJ	*Decompiler Java*
DMML	*Dynamic Memory Management Library*
DSP	*Digital Signal Processing*
OAA	*Object Allocation Address*
FPGA	*Field Programmable Gate Array*
GC[1]	*Garbage Collection*
HP	*Header Position*
JDK	*Java Development Kit*
JIT	*Just In Time*
JVM	*Java Virtual Machine*
KVM	*K-Virtual Machine*
RAM	*Random Acess Memory*
RISC	*Reduced Instruction Set Computer*
ROM	*Read Only Memory*
RTSJ	*Real Time Specification for Java*
SASHIMI	*System As Software and Hardware In Microcontrollers*
SGDM	*Sistema de Gerenciamento Dinâmico de Memória*
SOC	*System On A Chip*
SPEC	*Standard Performance Evaluation Corporation*
OST	*Object Status Table*
UL	*Used List*
VLIW	*Very Long Instruction Word*
WCET	*Wort Case Execution Time*
WCMR	*Wort Case Memory Requirements*
ZCT	*Zero Count Table*

[1] Dependendo do contexto GC pode ainda significar Garbage Collector

LISTA DE FIGURAS

Figura 1.1: Diferentes formas para execução de programas Java......................... 25
Figura 2.1: Etapas do algoritmo *Mark-sweep* 32
Figura 2.2: Funcionamento do algoritmo de contagem de referências 33
Figura 2.3: Segregated *Free List*s [DON 2001]... 34
Figure 2.4: O algoritmo de Baker.. 36
Figura 2.5: Arquitetura de [SRI 2003] para gerência dinâmica de memória........... 39
Figura 2.6: Aplicação do sistema de [SRI 2003] 40
Figura 2.7: Arquitetura proposta por [LIN 2000].. 41
Figura 2.8: Estruturas adicionais do método de [LIN 2000] 42
Figura 2.9: Estrutura em árvore de [SIE 2000] para armazenamento de *arrays*.... 45
Figura 3.1: Implementação Baseada no Algoritmo de Lin. 56
Figura 3.2: Estrutura do frame na arquitetura FemtoJava 58
Figura 3.3: Estrutura de frame complementar para o Femtojava........................... 58
Figura 3.4: Estrutura do frame do picoJava 2 59
Figura 3.5: Implementação baseada no Algoritmo de Ritzau................................ 60
Figura 3.6: Modificações no cabeçalho de objeto 61
Figura 3.6: Fluxo de projeto SASHIMI modificado. 62
Figura 3.7: Ferramentas existentes para instrumentação de código 64
Figura 3.8: Arquitetura geral da CAT 65
Figura 3.9: Arquitetura geral do módulo de substituição de código 65
Figura 3.10: Formato de arquivo .jad e o formato da aplicação modificada produzida pela CAT 66
Figura 3.11: Vinculação de códigos para as classes da aplicação 67
Figura 3.12: Acesso a *Constant pool* de uma classe qualquer 68
Figura 3.13: Acesso à estrutura de definição dos campos de classe 68
Figura 4.1: *Address Book*: Número de objetos na memória durante a execução.. 73
Figura 4.2: *Address Book*: Utilização de memória através da primeira implementação... 75
Figura 4.3: *Address Book*: Utilização de memória através da segunda implementação... 76
Figura 4.4: *Address Book*: Utilização de memória: implementação 1 x implementação 2 77
Figura 4.5: *MP3 Player*: Número de objetos na memória durante a execução 77
Figura 4.6: *MP3 Player*: Utilização de memória através da primeira implementação... 79
Figura 4.7: *MP3 Player*: Utilização de memória através da segunda implementação... 80
Figura 4.8: *MP3 Player*: Utilização de memória: implementação 1 x implementação 2 81

Figura 4.9: *Sokoban Game*: Número de objetos na memória durante a execução 82
Figura 4.10: *Sokoban Game*: Utilização de memória através da primeira implementação 83
Figura 4.11: *Sokoban Game*: Utilização de memória através da segunda implementação 84
Figura 4.12: *Sokoban Game*: Utilização de memória: implementação 1 x implementação 2 84
Figura 4.12: Decodificação dos frames pelo MP3 Player 92
Figura 4.13: Impactos sobre o conjunto de instruções 98
Figura 4.14: Comparação conjunto de Instruções FemtoJava x JVM 98

LISTA DE TABELAS

Tabela 2.1: Dados gerais das aplicações contidas no *benchmark* SPECjvm98 48
Tabela 2.2: Dados produzidos por [DYK 2002].. 50
Tabela 3.1: Resumo dos dados disponíveis na DMML.. 63
Tabela 3.2: 11 métodos implementados na classe DmmsStr. 63
Tabela 4.1: Estimativas para o conjunto de aplicações usadas neste trabalho 72
Tabela 4.2: detalhamento dos dados envolvidos para o cálculo da taxa de amostragem utilizada para cada aplicação 73
Tabela 4.3: *Address Book*: Desempenho da primeira implementação quanto ao uso de memória... 74
Tabela 4.4: *Address Book*: Desempenho da segunda implementação quanto ao uso de memória... 75
Tabela 4.5: *MP3 Player*: Desempenho da primeira implementação quanto ao uso de memória... 78
Tabela 4.6: *MP3 Player*: Desempenho da segunda implementação quanto ao uso de memória... 80
Tabela 4.7: *Sokoban Game*: Desempenho da primeira implementação quanto ao uso de memória... 82
Tabela 4.8: *Sokoban Game*: Desempenho da segunda implementação quanto ao uso de memória... 83
Tabela 4.9: Custos para as operações básicas com a primeira implementação.. 87
Tabela 4.10: Custos para as operações básicas com a segunda implementação.. 87
Tabela 4.11: Custos para os métodos da Classe DmmsStr com a primeira implementação.. 87
Tabela 4.12: Ciclos para os métodos da Classe DmmsStr com a segunda implementação.. 88
Tabela 4.13: Energia (nJ) para os métodos da Classe DmmsStr com a segunda implementação 89
Tabela 4.14: Potência (mW) para os métodos da classe DmmsStr com a segunda implementação 89
Tabela 4.15: *Address Book*: Ciclos e energia 90
Tabela 4.16: *Address Book*: Comparações de desempenho e energia 91
Tabela 4.17: *Address Book*: Potência média dissipada 91
Tabela 4.18: *MP3 Player*: Diferenças máximas para os 3 frames considerados.. 92
Tabela 4.19: *MP3 Player*: Média dos resultados para os três frames analisados.. 93
Tabela 4.20: *MP3 Player*: Percentual de diferenças em relação à média para os 3 frames centrais... 93
Tabela 4.21: *MP3 Player*: Ciclos e energia para todos os frames 94

Tabela 4.22: *MP3 Player*: Comparação do desempenho e energia 94
Tabela 4.23: *MP3 Player*: Consumo total produzido pela compactação 95
Tabela 4.24: *MP3 Player*: Influência da compactação para cada frame decodificado 95
Tabela 4.25: *MP3 Player*: Potência média dissipada 95
Tabela 4.26: *Sokoban Game*: Ciclos e energia 96
Tabela 4.27: *Sokoban Game*: Comparação do desempenho e energia 97
Tabela 4.28: *Sokoban Game*: Potência média dissipada 97
Tabela 4.29: Modificações sobre o conjunto de instruções da arquitetura 98
Tabela A1.1: Ciclos e energia para o 2º frame 113
Tabela A1.2: Ciclos e energia para o 3º frame 114
Tabela A1.3: Ciclos e energia para o 4º frame 114
Tabela A2.1: Arquivos de código desenvolvidos por este trabalho 115

RESUMO

Esta dissertação apresenta duas implementações de algoritmos para gerência dinâmica de memória em *software*, as quais foram desenvolvidas utilizando como alvo uma plataforma embarcada Java. Uma vez que a plataforma utilizada pertence a uma metodologia para geração semi-automática de *hardware* e *software* para sistemas embarcados, os dois algoritmos implementados foram projetados para serem integrados ao contexto desta mesma metodologia.

Como forma de estabelecer comparações detalhadas entre as duas implementações desenvolvidas, foram realizadas diversas estimativas em desempenho, uso de memória, potência e energia para cada implementação, utilizando para isto duas versões existentes da plataforma adotada. Através da análise dos resultados obtidos, observou-se que um dos algoritmos desenvolvidos obteve um desempenho melhor para realização da gerência dinâmica da memória. Em contrapartida, o outro algoritmo possui características de projeto que possibilitam sua utilização com aplicações de tempo-real.

De um modo geral, os custos adicionais resultantes da utilização do algoritmo de tempo-real, em relação ao outro algoritmo também implementado, são de aproximadamente 2% para a potência média dissipada, 16% para o número de ciclos executados, 18% para a energia consumida e 10% sobre a quantidade de total memória utilizada. Isto mostra que o custo extra necessário para utilização do algoritmo de tempo real é razoavelmente baixo se comparado aos benefícios proporcionados pela sua utilização.

Como impactos finais produzidos por este trabalho, obteve-se um acréscimo de 35% sobre o número total de instruções suportadas pela arquitetura utilizada. Adicionalmente, 12% das instruções que já existiam no conjunto desta arquitetura foram modificadas para se adaptarem aos novos mecanismos implementados. Com isto, o conjunto atual da arquitetura passa a corresponder a 44% do total de instruções existentes na arquitetura da máquina virtual Java.

Por último, além das estimativas desenvolvidas, foram também realizadas algumas sugestões para melhoria global dos algoritmos implementados. Em síntese, alguns pontos cobertos por estas sugestões incluem: a migração de elementos do processamento do escopo dinâmico para o estático, o desenvolvimento de mecanismos escaláveis para compactação de memória em tempo-real, a integração de escalonadores ao processo de gerência de memória e a extensão do processo de geração semi-automática de software e hardware para sistemas embarcados.

Palavras-Chave: Gerência Dinâmica de Memória, Garbage Collection, Linguagem Java, Sistemas Embarcados, Sistemas de Tempo Real.

Dynamic Memory Management in Embedded Java Applications

ABSTRACT

This dissertation presents two implementations of algorithms for dynamic memory management in software, developed using as target an embedded Java platform. Since the used platform belongs to a methodology of semi-automatic generation of hardware and software for embedded systems, the two implemented algorithms were designed to be integrated to the context of this methodology.

To the realization of detailed comparisons between the two implementations, several estimates in performance, memory use, power and energy for each implementation were developed, using for this two existing versions of the adopted architecture. Through the analysis of the obtained results, it was observed that one of the algorithms had a better performance to the realization of the dynamic memory management. On the other hand, the other algorithm holds design characteristics that allow it to be used with real-time applications.

In general, the additional costs of the utilization of the real-time algorithm, with regard to the other implemented algorithm, are of approximately 2% for the average dissipated power, 16% for the number of executed cycles, 18% for the consumed energy and 10% over the total quantity of utilized memory. This reveals that the extra cost necessary for the utilization of the real-time algorithm is reasonably small if compared to the benefits obtained by its utilization.

As final impacts produced for this work, was obtained an increase of 35% over the total number of instructions supported for the used architecture. Additionally, 12% of the instructions that already existed in the set of this architecture were modified to adapt to the new implemented mechanisms. In this way, the current set of the architecture pass to correspond to 44% of the total number of instructions existing in the architecture of the Java virtual machine.

Beyond the developed estimates, were also attained some suggestions to the global improvement of the implemented algorithms. Some points covered by these suggestions include: the migration of elements from the dynamic scope to the static scope, the development of scalable mechanisms for real-time memory compaction, the integration of schedulers to the process of memory management and the extension of the process of semi-automatic generation of software and hardware for embedded systems.

Keywords: Dynamic Memory Management, Garbage Collection, Java Language, Embedded Systems, Real-time Systems.

1 INTRODUÇÃO

Uma área de grande interesse, tanto no meio acadêmico quanto no industrial, diz respeito à concepção de dispositivos eletrônicos embarcados para as mais diversas aplicações, tais como processamento de áudio, vídeo, telefonia, transmissão e processamento de dados. Estes dispositivos encontram-se presentes na composição de grande parte dos utensílios eletrônicos usados diariamente pela humanidade, sendo por este motivo os principais responsáveis pelo desenvolvimento e comercialização de circuitos integrados no mundo.

Considerando todos os custos envolvidos na produção destes dispositivos e a atual concorrência existente entre os fabricantes, torna-se cada vez maior a necessidade de uma maior automatização do processo de desenvolvimento de *hardware* e *software* para aplicações embarcadas.

Para isto, foi criada no ambiente da UFRGS a metodologia SASHIMI (*System As Software and hardware In MIcrocontrollers*) [ITO 2001]. Atualmente a metodologia SASHIMI consiste de uma ferramenta para geração automática de *hardware* para sistemas embarcados, com base em uma especificação da aplicação a ser suportada, descrita em linguagem Java. Existem atualmente estudos no sentido de prover também a geração automática de *software* para sistemas embarcados, com base em uma especificação em mais alto nível da aplicação a ser suportada [MAT 2004].

A base para a síntese realizada pela ferramenta SASHIMI é o processador FemtoJava [ITO 2000] [BEC 2004] [BEC 2004a] [KRA 2002], uma vez que dele são extraídos os componentes de *hardware* utilizados para compor o sistema embarcado gerado, ou seja, o suporte em *hardware* gerado pela ferramenta é um subconjunto dos itens de *hardware* que formam a versão completa deste processador.

Atualmente, devido a algumas restrições em sua arquitetura, o FemtoJava não realiza suporte a uma parcela significativa do conjunto completo de *bytecodes* Java. Neste sentido, uma restrição muito importante está relacionada à ausência de um mecanismo de gerenciamento dinâmico de memória, uma vez que, sem este mecanismo, não é possível estabelecer suporte a elementos dinâmicos que compõem uma importante vantagem relacionada à orientação a objetos proporcionada por Java.

1.1 A linguagem Java

Java [YEL 99] é uma linguagem de programação orientada a objetos desenvolvida originalmente para programação em alto nível de componentes eletrônicos, mas sua primeira grande utilização foi em uma aplicação *desktop* voltada à navegação na Internet [IVA 99]. Java foi criada pela Sun Microsystems [SUN 2005] para oferecer diversas características que a tornariam ideal para projetos complexos de *software*.

Entre as principais vantagens disponíveis em Java podem-se citar: maior facilidade para desenvolvimento de código, portabilidade, simplicidade e um completo ambiente de desenvolvimento. A maior facilidade para desenvolvimento está relacionada antes de tudo ao fato da linguagem ser orientada a objetos e em segundo plano devido à linguagem não possuir recursos complexos como múltipla herança, existente por exemplo em C++ [SOU 2005].

Java é baseada e especificada pela *Java Virtual Machine (JVM) Specification* [YEL 99]. Tradicionalmente, a forma de execução de um programa Java em sistemas do tipo *desktop* consiste na compilação de um código fonte Java resultando em um código intermediário (os *bytecodes* Java), que são interpretados por uma arquitetura em *software* (JVM) capaz de rodar sobre diferentes plataformas existentes (Windows, Linux, Macintosh, OS/2 entre outras). Isto é o que garante a portabilidade de Java.

Entretanto, com o intuito de solucionar algumas deficiências mantidas pelo uso de uma camada adicional de *software* (JVM), surgiram outras alternativas para a execução de aplicações descritas em Java.

Uma das principais deficiências de Java é o elevado tempo de execução dos seus programas, se comparados a programas descritos em outras linguagens como C. Este baixo desempenho está relacionado ao custo de interpretação mantido pelo uso de uma arquitetura virtual.

Algumas soluções para este problema baseiam-se no uso de compiladores Java que compilam o código fonte diretamente para o código de máquina da arquitetura alvo. A principal conseqüência disto é a perda da portabilidade mantida pelo uso de um código intermediário.

Outras soluções baseiam-se no uso de compiladores JIT (*Just In Time*), que realizam a compilação do código intermediário (*bytecodes*) para o código da máquina alvo em tempo de execução. Esta solução gera um agravamento de outro problema da linguagem relacionado ao alto custo em memória para execução de um programa.

Por fim, outra possibilidade é a construção de máquinas Java para execução nativa de *bytecodes* Java. Esta solução eleva o desempenho da execução, ao mesmo tempo em que diminui drasticamente a quantidade de recursos de memória necessários, uma vez que se elimina a camada adicional de *software* necessária para realizar a interpretação. Além disso, a portabilidade da linguagem neste caso fica conservada. A fig. 1.1, originalmente extraída de [AAS 2001], exibe algumas das diferentes formas de se executar um programa Java, tendo sido adaptada para mostrar uma opção não considerada. Na fig. 1.1, o caminho em setas negras mostra a forma de execução utilizada neste trabalho.

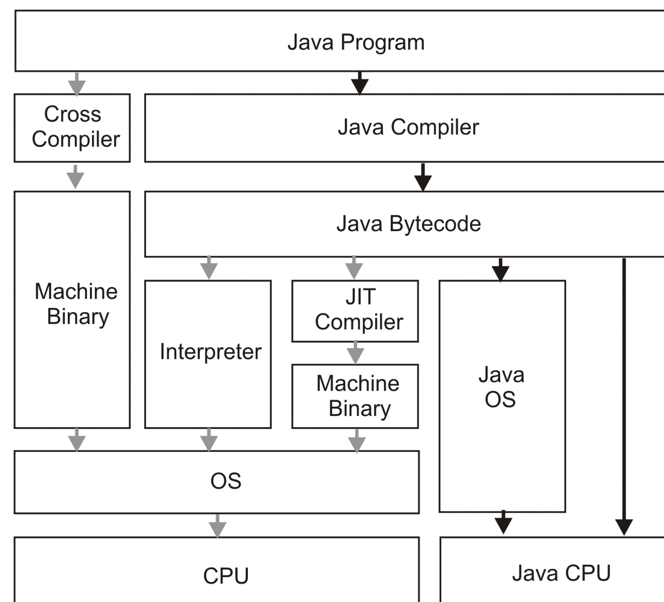

Figura 1.1: Diferentes formas para execução de programas Java

1.2 Sistemas Embarcados e Tempo Real

Sistemas embarcados são sistemas projetados e desenvolvidos para serem alocados a um propósito bem específico [LEW 2000]. Nestes sistemas, tanto o *software* quanto o *hardware* são otimizados segundo um ou mais parâmetros de projeto, tais como: área, potência, desempenho, previsibilidade, segurança e consumo de energia.

O objetivo primário para o qual surgiram sistemas embarcados foi o controle de dispositivos externos, tais como freios, bombas, controle de vôo de aeronaves, sistemas de segurança, dispositivos médicos, máquinas de lavar e microondas. Atualmente, algumas tecnologias tais como computação distribuída, sistemas inteligentes, computação móvel, redes de comunicação e a Internet têm expandido a aplicação dos sistemas embarcados.

A falta de envolvimento direto com o usuário em muitos casos exige que tanto o *hardware* quanto o *software* sejam altamente confiáveis, o que em outras palavras significa que estes componentes do sistema devem, em muitos casos, ser tolerantes a falhas e trabalhar em tempo real, uma vez que o propósito do sistema é responder e controlar eventos do mundo real [LEA 99].

Originalmente, sistemas embarcados eram baseados em processadores de 8 bits e eram programados em linguagem assembly. Atualmente, os sistemas embarcados usam processadores de 32, 64 e até 128 bits e, devido a fatores como tempo de projeto e fabricação, são programados prioritariamente em linguagens de alto nível tais como: Ada, C++ e Java [COR 2003].

Tipicamente, o *software* para sistemas embarcados é responsável por 80 a 90% das funcionalidades globais e custos de desenvolvimento de um sistema embarcado [DIA 2000].

Muitos sistemas embarcados seguem severas restrições de tempo real e segurança. Uma restrição de tempo real pode ser definida segundo dois pontos de vista:

- sobre o conhecimento preciso de quais suboperações são necessárias à realização de uma dada operação;

- sobre o conhecimento preciso do tempo limite dentro do qual o sistema deve realizar uma dada operação.

Alguns exemplos de restrições de tempo real podem ser:

- um sistema deve levar não mais que 10 segundos para executar uma operação aritmética complexa;
- cada ciclo de aquisição não deve exceder 2 segundos;
- a execução de uma tarefa consome sempre 387 instruções de uma máquina e as instruções executadas são sempre as mesmas para a execução desta tarefa.

Diferentes sistemas podem ter diferentes requisitos para alcançar seus prazos finais de execução (*deadlines*). Sistemas de tempo real podem ser de dois tipos: hard e soft.

Sistemas do tipo hard são aqueles onde a perda de um *deadline* pode causar um verdadeiro desastre (com possibilidade de perda de vidas). Alguns exemplos destes sistemas podem ser encontrados em estações de energia nuclear, equipamentos médicos e em controle de aeronaves.

Por outro lado, sistemas do tipo soft são aqueles onde a perda de um *deadline* pode ser ocasionalmente aceitável. Por exemplo, em sistemas telefônicos, quando uma pessoa deseja realizar uma chamada, o tom de discagem deve estar disponível dentro de um certo período de tempo. Uma falha ao produzir um tom neste caso não é catastrófica, desde que a mesma ocorra apenas com uma baixa periodicidade.

O grau exato de tolerância à perda de *deadlines* por um sistema de tempo real é informado segundo o nível de segurança exigida para este sistema. Segurança é uma função do tipo de trabalho exigido para um dado sistema [LEW 2000].

Freqüentemente, o conceito de sistemas de tempo real é confundido com o conceito de sistemas interativos. Como o próprio nome diz, sistemas interativos são aqueles que realizam iteração de alguma forma com seus usuários, mas não necessariamente devem ter *deadlines* rígidos. Algumas aplicações para os sistemas interativos são agendas, alguns tipos de jogos, chats, etc.

Outras restrições importantes para sistemas embarcados são o espaço físico e o custo de desenvolvimento que limitam o tipo e a quantidade de *hardware* utilizados. Um exemplo típico é um sistema de satélite onde o tamanho e o custo máximo definem o poder de computação do sistema. Neste sentido, um problema que vem sendo bastante investigado diz respeito à quantidade de memória disponível em sistemas, já que o tamanho máximo da memória influencia outros parâmetros do sistema como tempo de execução, previsibilidade, consumo de potência e energia, como se vê por exemplo em [CHE 2002] [LEE 2003] [MIL 2005].

Por fim, o consumo de potência e energia também é outra restrição relevante em sistema embarcados já que grande parte dos sistemas embarcados existentes operam com o uso de baterias (Palms, celulares, etc).

1.3 Java em Sistemas Embarcados

O mercado de sistemas embarcados e de tempo real está explodindo na era pós-PC, especialmente conforme cada vez mais dispositivos são capazes de se comunicar via Internet [HAR 2000].

Uma vez que a produção de *software* para tais sistemas torna-se cada vez mais difícil devido à crescente complexidade e intensa concorrência entre os fabricantes, surge a necessidade de se buscar novas alternativas para o desenvolvimento do *software* para estes sistemas.

Desenvolvedores e consumidores se defrontam, a cada dia, com produtos cada vez mais heterogêneos, com uma pletora de processadores, sistemas operacionais e dispositivos periféricos.

Neste meio, a tecnologia Java vem crescendo devido a maior produtividade e portabilidade que são fundamentais para o desenvolvimento de sistemas embarcados.

Entretanto, a baixa performance e o comportamento não determinístico de muitas implementações Java têm limitado seu uso em sistemas embarcados de tempo real. A natureza dinâmica do ambiente de execução Java é uma das principais vantagens nos sistemas *desktop*s. Contudo, algumas áreas de Java, como a realização de uma gerência dinâmica de memória, são problemas importantes a serem mapeados para o contexto de sistemas de tempo real.

Visando encontrar soluções para este e outros problemas inerentes a Java, em 1999 foi criada a Real Time Specification For Java (RTSJ) que proveu um conjunto de diretivas para sete áreas críticas de Java, com relação a tempo real [BOL 2000]. Como exemplo destas diretivas foram criadas novas áreas de memória para a alocação de dados dinâmicos (objetos), além da já existente Heap.

Além dos problemas relacionados à restrição de tempo real, Java possui também outros fatores que dificultam a sua migração para sistemas embarcados:

- Ausência de mecanismos para acesso direto à memória e tratamento de interrupção, os quais são fundamentais para desenvolvimento de *drivers* de controle para dispositivos.
- Grande consumo de memória para armazenamento da JVM e da aplicação.
- Execução lenta devido ao código ser interpretado.
- Dependência de um sistema de arquivos para armazenamento de classes obtidas via Internet.
- Alto consumo de potência e energia devido à execução em uma arquitetura em *software*.

Conforme mostrado na fig. 1.1, a forma de execução de Java, usada neste trabalho, elimina grande parte dos problemas acima mencionados. Como existe disponibilidade de uma CPU para execução nativa de *bytecodes* Java, criaram-se também mecanismos para tratamento de interrupções. Pelo mesmo motivo deve-se subtrair dos custos em memória e energia uma parcela significativa correspondente à JVM, que uma vez em *hardware* elimina-se o custo de interpretação. Esta forma de execução de Java é melhor segundo [IVA 99], considerada a necessidade de conservação da portabilidade e da carga dinâmica de classes, necessárias para adequada integração de sistemas embarcados à Internet.

Outro fator crítico na migração para sistemas embarcados é que Java por padrão inclui uma larga biblioteca de classes não escalável. Em sistemas embarcados onde é altamente desejável que apenas os dados realmente necessários residam em uma memória extremamente limitada, surge a necessidade de adaptação destas bibliotecas para que possam ser aplicadas neste novo contexto. A tarefa de analisar e remover todas as classes e métodos

não necessários para minimizar os custos finais em memória para aplicações embarcadas pode ser difícil [PER 2000].

Finalmente, todos os fatores anteriormente mencionados precisam ser adequadamente ajustados para que o custo final do produto não seja a última e principal barreira para a consolidação de Java no mercado de sistema embarcados.

1.4 A Arquitetura FemtoJava e as Metas para este Trabalho

O FemtoJava [ITO 2000] foi criado em 1999 com base na especificação da JVM (*Java Virtual Machine*) [YEL 99] com o propósito de que se obtivesse uma máquina capaz de executar nativamente *bytecodes* Java.

Tal como a estrutura da JVM, o conjunto de instruções suportado por este processador pressupõe uma arquitetura de pilha para sua execução.

As principais características relevantes deste processador são:

- Apenas uma única *thread* pode ser suportada.
- Todos os dados são do tipo inteiro.
- Sua arquitetura é gerada automaticamente através do ambiente SASHIMI de acordo com os requisitos da aplicação, existindo em função disto a possibilidade de escolha do tamanho da palavra de dados a ser utilizada.
- Sua síntese, através do ambiente SASHIMI, produz um conjunto de componentes descritos em linguagem VHDL que podem ser validados em dispositivos do tipo FPGA.
- Possui memórias de programa e de dados internas e fisicamente distintas, utilizando espaços de endereçamento separados.
- Os tamanhos das memórias são configuráveis de acordo com as características da aplicação.
- Atualmente os dados da aplicação que residem em memória, precisam ser alocados em tempo de projeto, ou seja, não existe alocação de memória em tempo de execução.
- A versão Multiciclo foi a primeira a ser desenvolvida, mas atualmente já existem as versões Pipeline [BEC 2004], VLIW [BEC 2004a] e DSP [KRA 2002].

Dentre as principais deficiências presentes atualmente na arquitetura do processador, estão a ausência de suporte para mais de um fluxo de execução (*thread*) e a inexistência de suporte à alocação dinâmica de memória. Com o aumento da complexidade das aplicações devido ao avanço dos requisitos exigidos para os diferentes tipos de sistemas embarcados existentes, a carência por estas duas características passou a restringir o emprego deste processador para tais aplicações, ou pelo menos exige que uma série de adaptações sejam realizadas no código destas aplicações, para que estas possam ser suportadas pelo conjunto atual de instruções existente.

Este trabalho pretende solucionar o problema relativo à gerência de memória, provendo ao FemtoJava a capacidade de alocar objetos Java dinamicamente, assim como também de realizar a desalocação automática destes objetos através de mecanismos de *garbage collector* [JON 96].

O restante de texto está organizado da seguinte forma: o capítulo 2, a seguir, apresenta os principais trabalhos realizados com relação a gerência dinâmica de memória. O capítulo 3 apresenta as arquiteturas produzidas através deste trabalho. O capítulo 4 apresenta os resultados obtidos através de cada

arquitetura desenvolvida. Por fim, o capítulo 5 apresenta as conclusões finais alcançadas assim como algumas propostas de atividades futuras relacionadas a este trabalho.

2 GERÊNCIA DINÂMICA DE MEMÓRIA

Em Java toda a gerência de memória é realizada automaticamente, ou seja, o desenvolvedor não precisa se preocupar com a alocação nem com a desalocação de memória. A forma de alocação dinâmica de memória é através da criação de objetos que são desalocados automaticamente por um mecanismo de *Garbage collection* [JON 96], sempre que não são mais úteis ao programa.

O suporte à gerência dinâmica automática de memória é um dos maiores responsáveis pela robustez, simplicidade e facilidade de desenvolvimento de Java. Uma vez que as tarefas de alocação e desalocação de memória não são responsabilidade do programador, Java não sofre com os erros de memória causados pela manipulação incorreta de ponteiros, manipulação esta comumente praticada por muitos desenvolvedores que utilizam linguagens como C++.

Entretanto, a gerência dinâmica de memória torna-se um grave problema quando Java é usada em sistemas embarcados, uma vez que, conforme dito anteriormente, estes sistemas em geral possuem requisitos de tempo real, consumo de potência e tamanho de memória.

Existe uma vasta quantidade de pesquisas já realizadas em busca por soluções para as deficiências de Java, muitas das quais são pensadas especificamente tendo como alvo os sistemas embarcados.

Devido à grande quantidade de publicações sobre gerência dinâmica de memória, pode-se pensar que o problema já foi plenamente resolvido. De fato, muitas soluções existem para uma grande parte das aplicações, contudo a situação para aplicações de tempo real embarcadas é completamente diferente, sendo significantemente pequeno o número de artigos publicados sobre gerência dinâmica de memória para sistemas embarcados nos eventos mais relevantes sobre tempo-real [MIL 2005].

Nestes trabalhos, a expressão *garbage collection* é utilizada com grande freqüência para se referenciar ao processo de gerência dinâmica de memória, por isto ambas as expressões serão utilizadas sem restrições neste trabalho, para fazer referência ao conjunto de técnicas para alocação e desalocação dinâmica de memória.

2.1 *Garbage Collection*

Um *garbage collector* distingue os objetos em memória que não são mais necessários (garbage) dos objetos ainda em uso e reclama (libera) o espaço ocupado por objetos inúteis para futuro uso [JON 96]. Um objeto é considerado em uso ou vivo quando ele é alcançável a partir de alguma variável na pilha ou a partir de algum objeto sabidamente vivo [KIM 99].

É comum na literatura a categorização de algoritmos clássicos de *garbage collection* em duas classes: *Reference Counting* e *Tracing*. Os algoritmos de *tracing* mais simples são conhecidos como *stop-the-world* (pare o mundo), uma

vez que exigem que a aplicação pare de executar e aguarde até que o *garbage collector* identifique e colete todos os objetos inacessíveis na memória. Estes tipos de algoritmos, por sua vez, podem ser classificados em *Mark-sweep* e *Copying*.

2.1.1 Mark-sweep

Este algoritmo atua em duas etapas, das quais a primeira consiste em uma marcação (*trace*) de todos os objetos alcançáveis a partir de algum nodo raiz. Na segunda etapa, é realizada uma varredura sobre a memória para reclamar todos os objetos não marcados pela primeira etapa. A fig. 2.1 abaixo mostra as etapas do algoritmo *Mark-sweep*.

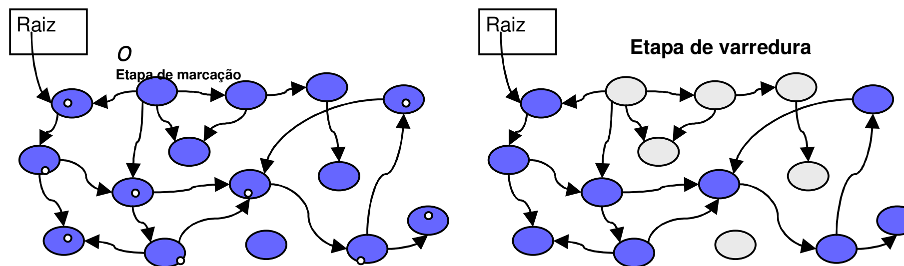

Figura 2.1: Etapas do algoritmo *Mark-sweep*

2.1.2 Copying

Consiste em dividir a memória em duas áreas: *From-Space* e *To-Space*, apenas uma das áreas permanece disponível para alocação durante toda a execução. Quando a quantidade de espaço ocupado no *To-Space* atinge um certo limite, o *To-Space* passa a ser o *From-Space* e o *From-Space* se torna o *To-Space*. Neste momento, o *From-Space* começa a ser vasculhado em busca de objeto alcançáveis que são copiados (evacuados) para o novo *To-Space*. Após todo o *From-Space* ter sido vasculhado este espaço é inteiramente reclamado.

2.1.3 Reference Counting

O método de Contagem de Referências [JON 96] baseia-se na presença de um contador para cada objeto alocado na memória. Este contador é incrementado sempre que uma nova referência é feita ao objeto pela aplicação e decrementado toda vez que uma referência ao objeto é extinta. Sempre que o valor de um contador atinge o valor zero o objeto correspondente deve ser desalocado, uma vez que a aplicação não exige sua presença na memória. A fig. 2.2 ilustra como ocorre o funcionamento do algoritmo de contagem de referências.

Na fig. 2.2, a referência *p* que apontava para o objeto *o1* passa a apontar o objeto *o2.* Como o objeto *o2* obtém a referência p, seu contador de referências (*o2.rc*) deve ser incrementado. Em contrapartida, a perda da referência *p* faz com que o contador de referências do objeto *o1* (*o1.rc*) seja decrementado. Como o novo valor do contador é zero, o espaço ocupado pelo objeto *o1* na memória deve ser então liberado. Antes, contudo, que o espaço ocupado por *o1* seja liberado é necessário que todas as referências que ele mantém a outros objetos (descendentes de *o1*) sejam excluídas. Este processo se repete até que nenhum descendente (direto ou indireto) de *o1* possua referência a qualquer outro objeto.

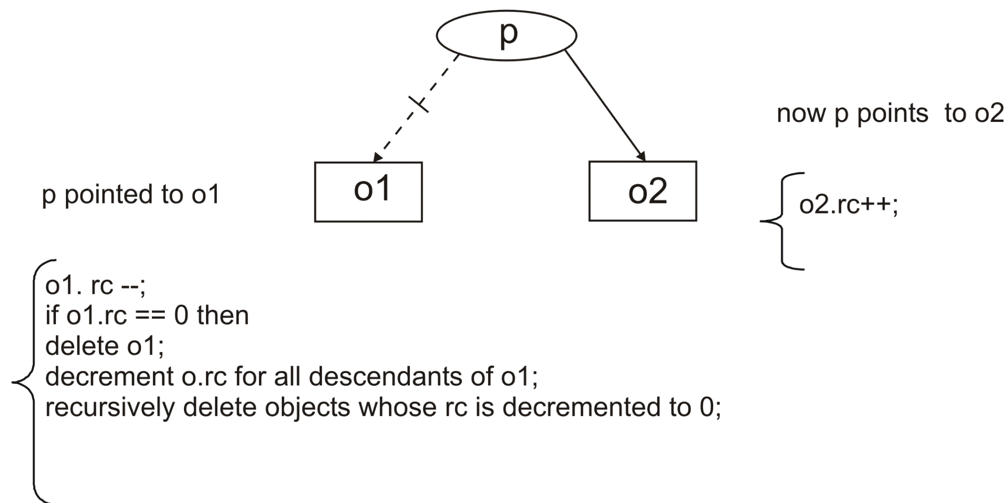

Figura 2.2: Funcionamento do algoritmo de contagem de referências

2.2 Variações dos Algoritmos

Os algoritmos acima descritos podem ainda sofrer algumas variações conforme os requisitos pretendidos para o sistema alvo. Abaixo estão listadas diferentes formas de implementação de um mesmo algoritmo clássico, segundo [JON 96]:

Incremental: um algoritmo incremental é capaz de ser interrompido sempre que a aplicação estiver em condições de execução e retornar a execução quando a aplicação não estiver executando ou requisitar liberação de memória.

Paralelo: o algoritmo executa em um processador diferente do utilizado para a aplicação. Exige que estratégias de sincronização controlem os acessos à memória compartilhada.

Geracional: divide logicamente a memória em áreas chamadas gerações. O algoritmo concentra a maior parte de seus esforços sobre as gerações mais novas onde há uma maior taxa de mortalidade de objetos. Objetos que sobrevivem ao *garbage collector* são promovidos a gerações mais antigas onde o índice de mortalidade é menor exigindo menos atividade de coleta.

Postergado: é capaz de adiar uma ação até que ela seja realmente necessária, poupando assim execução com tarefas muitas vezes caras e inúteis.

2.3 Comparação dos Algoritmos Clássicos

Considerando todos os possíveis campos de aplicação para algoritmos acima descritos, é possível encontrar vantagens e desvantagens em cada algoritmo. Como um simples exemplo, o algoritmo de *Mark-sweep* é o algoritmo que oferece o melhor desempenho, sendo por isso o algoritmo utilizado para a gerência de memória na JVM [HAN 2002]. Entretanto, este algoritmo possui um tempo de resposta variável, o que é uma desvantagem vital em sistemas embarcados de tempo real que necessitam de previsibilidade. O algoritmo de cópia mantém implicitamente uma organização linear dos dados, a um custo de se utilizar apenas metade dos recursos disponíveis em memória.

Nestas condições, o algoritmo de contagem de referências oferece um comportamento no nível de granularidade de operações desejável para sistemas

embarcados. Entretanto, o *overhead* em tempo de execução torna-se um problema a ser amortizado através de alguma estratégia.

2.4 Propostas Existentes

Esta seção apresenta algumas das principais propostas existentes para a gerência de memória em sistemas de tempo real. Uma subdivisão desta seção foi realizada para separar as propostas antigas mais importantes, que até hoje formam a base de muitas propostas atuais, das propostas mais recentes entre as quais estão as bases para este trabalho.

2.4.1 Propostas Clássicas

2.4.1.1 Segregated Free Lists

Uma das políticas mais simples de alocação utiliza um conjunto de listas de blocos livres de memória, onde cada lista armazena blocos de um tamanho fixo particular [DON 2001]. Quando um bloco de memória é liberado ele é simplesmente empilhado na lista correspondente ao seu tamanho. Quando a aplicação requisita um espaço de memória, a lista com blocos do tamanho correspondente ao tamanho do espaço desejado é acessada para satisfazer a requisição.

Existem diversas e importantes variações de listas segregadas. Uma variação comumente usada é criar blocos de tamanho iguais aos de cada classe da aplicação. Caso duas classes possuam o mesmo tamanho seus blocos podem ser agrupados em uma única lista de objetos similares.

Outra variação também bastante utilizada consiste em manter listas cujo tamanho dos blocos aumentam em uma potência de dois (fig. 2.3) e utilizar um arredondamento do tamanho necessário para satisfazer a requisição para o tamanho de bloco imediatamente superior quando necessário.

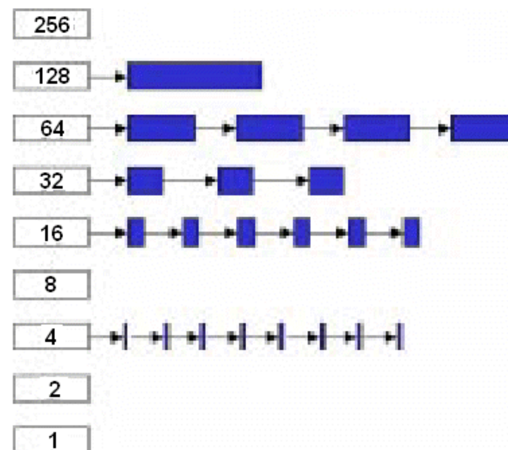

Figura 2.3: Segregated *Free List*s [DON 2001]

2.4.1.2 Buddy Systems

Buddy Systems [PET 1977] são uma variação de listas segregadas suportando um limitado mas eficiente método de quebra e aglutinação. Em uma versão bem simples, a heap é conceitualmente quebrada em duas grandes porções chamada *Buddies*. Estas áreas são repetidamente quebradas em dois pequenos *buddies*, até que uma suficientemente pequena parte é alcançada. Esta divisão hierárquica é utilizada para limitar onde objetos devem ser alocados e como eles podem ser aglutinados em grande áreas livres.

Uma área livre pode somente ser unida com seu *buddy*, ou seja, o primeiro bloco de um par de *buddies* pode somente ser unido com o bloco seguinte. O bloco livre resultante é, portanto, uma das áreas livres pertencentes ao nível imediatamente superior na hierarquia de divisão de memória. Esta restrição no processo de aglutinação de blocos garante que o bloco resultante sempre estará alinhado em um dos limites da divisão hierárquica de memória.

Quando um bloco é liberado seu *buddy* pode ser encontrado por um simples calculo de endereço e seu *buddy* sempre será uma lacuna (uma porção livre de memória) ou uma porção não disponível de memória (por ter sido ocupada em uma alocação ou por ter sido quebrada e alguns ou todas as suas subpartes terem sido alocadas).

O processo de aglutinação é relativamente rápido, mas a maior vantagem em alguns contextos é que ele requer um pequeno *overhead* por objeto (somente um bit é requerido por *buddy*) para indicar se o *buddy* é uma área livre contígua. Donahue [DON 2001] diz que o tempo $O(log(M))$ (onde M é o tamanho da heap), necessário para se obter um bloco a partir de blocos maiores, quando não existem blocos de um tamanho desejado disponíveis, é eficiente para sistemas embarcados de tempo real.

2.4.1.3 O algoritmo do Trem

Seligman [SEL 95] apresentou o Algoritmo do Trem que combina *traces* incrementais (*incremental tracing*) com coleta geracional. Neste esquema, a tarefa de coleta ocorre em pequenos passos. Seu algoritmo quebra a geração mais antiga em blocos de tamanho fixo chamados carros. Trens são feitos de grupos de carros que são encadeados. O algoritmo tenta fazer com que os objetos relacionados (que se referenciam) fiquem agrupados em um mesmo trem de forma a facilitar a liberação de memória. A principal estratégia para redução do tempo de resposta do algoritmo consistiu em coletar apenas uma pequena porção da geração mais antiga durante cada execução do GC.

2.4.1.4 Cache Collector

Em [GEH 93] foi desenvolvido um *garbage collector* que reside em uma memória cache. Este coletor realiza a alocação de objetos primariamente na cache sem se importar com a busca por espaço livre na memória principal. Um algoritmo de contagem de referências é utilizado para remover muitos objetos antes de serem escritos na memória principal. Em simulação, esta estratégia conseguiu remover entre 50 a 70% dos objetos antes que eles saíssem da cache. A estratégia utilizada para realizar a alocação é um *buddy system* modificado para ser executado por um *hardware* combinacional.

2.4.1.5 Deferred Reference Counting

O método de Deustch-Brobow [DEU 1976] para contagem de referências explora o fato de que muitas atualizações de referências são geradas sobre referências armazenadas em variáveis locais. Sua estratégia consiste em fazer com que a atualização de referências mantidas por variáveis locais não gere incrementos do contador do objeto, mas ao invés disto, estes incrementos são adiados até serem necessários. Assim, se uma referência é destruída rapidamente então o respectivo contador do objeto não é atualizado. Isto elimina um grande número de atualizações associadas com objetos de vida curta. Contudo, se uma referência é atualizada dentro de um objeto, então o incremento do contador deve ser realizado naquele mesmo momento.

O princípio básico de funcionamento do algoritmo consiste em fazer com que antes de um objeto com contador igual a zero seja deletado, sua referência deva ser empilhada em uma *Zero Count Table* (ZCT). Em períodos específicos o sistema deve verificar através de uma busca sobre a pilha e registradores se existem referências para os objetos contidos na ZCT, caso não existam, estes objetos são então desalocados conforme a necessidade de liberação de memória. Especificamente, segundo dados atuais [PET 2004], o uso desta técnica permite uma redução de até 80% do tempo de execução em relação a um algoritmo de contagem de referências tradicional.

2.4.1.6 Algoritmo de Baker

Em outra técnica desenvolvida por Henry Baker [BAK 93], todos os objetos livres e alocados são mantidos em uma lista circular duplamente encadeada, conforme mostra a fig. 2.4. Todas as operações são realizadas atualizando ponteiros dentro desta lista. Alguns bits por objeto são mantidos por um sistema de cores no qual cada cor possui o seguinte significado:

- *Branco*: objeto livre.
- *Cinza*: objeto vivo mas ainda não escaneado.
- *Preto*: objeto vivo já escaneado.
- *Cor de linho (ecru)*: usada para denotar objetos vivos ou alocados desde a última coleta, mas que ainda não foram marcados ou escaneados.

Uma vez que todos os objetos precisam ter o mesmo tamanho, a alocação é feita em tempo constante simplesmente transferindo nodos da lista de cor branca para a lista cor de linho. A coleta é realizada transferindo os objetos da lista cor de linho para a lista de cor branca até que a lista cor de linho contenha apenas objetos livres.

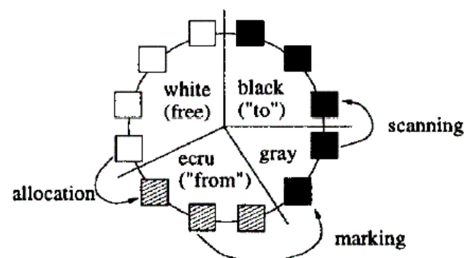

Figure 2.4: O algoritmo de Baker

2.4.1.7 *Lazy Freeing*

Weizenbaum [WEI 1963] inventou a *Lazy Freeing* (liberação preguiçosa) visando solucionar o problema do não determinismo para se decrementar o contador dos filhos de um objeto morto. O processo desenvolvido por Weizenbaum consiste das seguintes etapas:

- Quando um objeto tem seu contador decrementado para zero, o objeto deve ser empilhado em uma pilha dedicada especificamente para estes objetos.
- Durante a alocação o objeto no topo da pilha deve ser desempilhado e liberado e o espaço ocupado pelo objeto deve ser utilizado para realizar a alocação.
- A liberação do objeto desempilhado consiste em decrementar todos os filhos do objeto e para todo o filho cujo contador atingir zero empilhar este filho.
- Se mais espaço for requerido para a alocação um outro objeto deve ser desempilhado.

A principal vantagem propiciada pelo método de Weizenbaum é a quebra do custo da desalocação ao longo da computação, ou seja, a eficiência do *Reference counting* praticamente não é modificada enquanto que a desalocação é espalhada por diversas operações de alocação.

2.4.2 Propostas Modernas e Atuais

Atualmente, muitos algoritmos para *garbage collection* requerem uma quantidade substancial e variável de memória. Idealmente, isto só poderia ser eficientemente alcançado através da integração de uma lógica de *garbage collection* com dispositivos de memória [LIN 2001] [WIS 97]. Entretanto, como o mercado de dispositivos de memória é um mercado muito sensível ao custo, memórias com *garbage collection* integrado não poderiam ser produzidas em um número tal que as tornassem comercialmente viáveis. Neste sentido, ocorre a necessidade de se projetar sistemas dedicados que gerenciem a memória dinamicamente.

2.4.2.1 *Garbage Collectors Paralelos e Controlados por Escalonadores*

Uma vez que em geral *Garbage Collectors* são intrusivos, por causarem longas pausas na execução da aplicação, alguns projetistas têm concentrado seus esforços em reduzir o tempo de pausa. Blelloch [BLE 2001] desenvolveu um *garbage collector* paralelo, onde o objetivo era eliminar o tempo de pausa, uma vez que a memória era liberada enquanto a aplicação continuava sua execução concorrentemente. O principal obstáculo encontrado por Blelloch foi estabelecer um mecanismo de sincronização, entre o GC (*garbage collector*) e a aplicação, que não resultasse em um significativo *overhead* em execução.

Outra alternativa bastante utilizada hoje em dia, em sistemas de tempo real, é a implementação de *Garbage Collectors* em conjunto com escalonadores [KIM 99] [HEN 2003]. Nestas implementações, o algoritmo de *garbage collection* utilizado deve ser capaz de ser executado de forma incremental. Os escalonadores devem executar tarefas de alta prioridade do GC, que são responsáveis por reservar uma certa quantidade de memória livre para alocação evitando assim a perda de *deadlines* por tarefas críticas da aplicação.

Kim [KIM 99] apresentou um algoritmo de escalonamento para múltiplas aplicações e um *garbage collector* rodando em um sistema embarcado de tempo real, com um único processador. O algoritmo de Kim é baseado em uma técnica de escalonamento aperiódico, onde todas as aplicações devem ter prioridades concordantes com um *rate monotonic scheduling*, no qual a tarefa mais curta tem a maior prioridade.

2.4.2.2 Sistemas Híbridos

Muitas vezes a diminuição do tempo de pausa para *garbage collection* pode ocasionar um prejuízo para a performance global da aplicação. Um exemplo disto seria uma implementação tradicional de um algoritmo de contagem de referências. Com o intuito de diminuir o tempo de pausa para *garbage collection* sem prejuízos para a performance, Blackburn [BLA 2003] desenvolveu um *garbage collector* híbrido, que combina um algoritmo de cópia geracional (seção 2.2) para coleta de objetos de vida curta e um *Reference Counting* para coleta de objetos de vida longa. Blackburn baseou sua estratégia no fato que *Reference Counting* produz menos esforços para a coleta de objetos antigos, isto porque os objetos antigos são objetos que tendem a permanecerem por mais tempo vivos. Neste sentido, *tracing* resultaria em um grande custo de execução para liberar uma pequena quantidade de objetos que morreriam na *Old Generation.* Em contrapartida, *Reference Counting* resultaria analogamente em um custo desnecessário para atualização de contadores de objetos contidos em uma *Young Generation*. Neste caso, um algoritmo de *tracing*, que escaneia apenas objetos acessíveis, deveria ser utilizado para um maior desempenho do sistema. Entretanto, para que o algoritmo de *tracing* não possua um longo tempo de pausa para verificar todos os objetos acessíveis na *Young Generation*, foi necessário que o tamanho desta área fosse relativamente pequeno em relação ao *Old Generation*, dado que um ciclo de execução de um algoritmo de *tracing* é uma função direta do número de objetos alcançáveis a partir da pilha o que por sua vez depende do tamanho da área a ser coletada. Considerando que o algoritmo de *tracing* utilizado por Blackburn é um algoritmo de cópia, todos os objetos vivos encontrados na *Young Generation* durante a execução do algoritmo de coleta são copiados para a *Old Generation*.

2.4.2.3 Gerência de Memória em Hardware

2.4.2.3.1 Srisa-an Co-processor

Srisa-an [SRI 2003] desenvolveu um co-processador para realizar *garbage collection* chamado *Active Memory Processor* (*AMP*). Este co-processador é uma estrutura combinacional baseada em um sistema de vetores de bits para realização do controle dinâmico da memória. A estratégia utiliza *Reference counting* com um contador de tamanho bastante limitado como algoritmo principal e um *Mark-sweep* para eliminação de lixo cíclico quando necessário. A fig. 2.5 mostra a arquitetura AMP desenvolvida para gerência de memória.

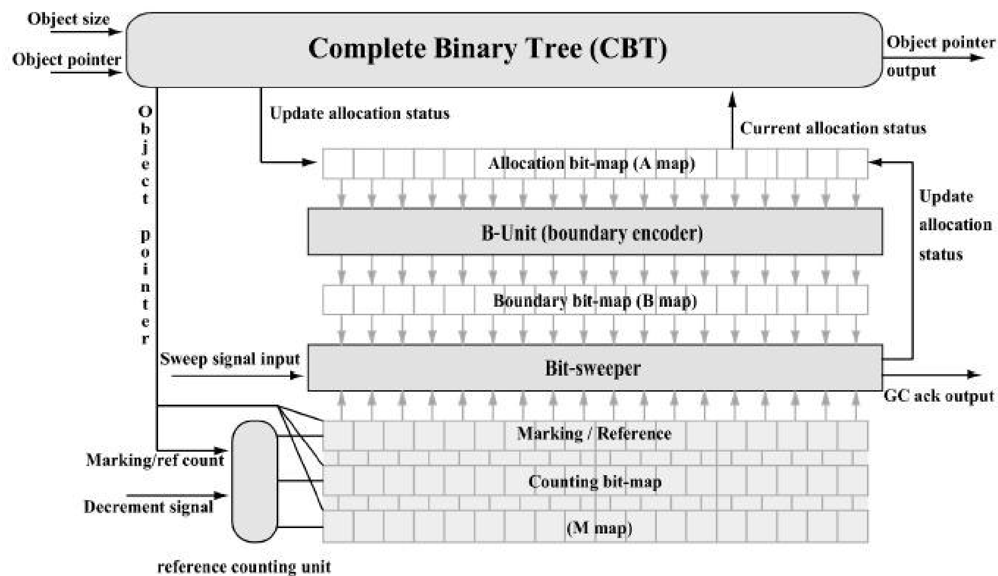

Figura 2.5: Arquitetura de [SRI 2003] para
gerência dinâmica de memória

Na arquitetura exibida na fig. 2.5, a cada ciclo os vetores precisam estar atualizados para refletirem o atual estado de alocação da memória. Cada bit, em cada um dos vetores, representa o estado de ocupação de um endereço físico de memória. A nomenclatura e a função de cada vetor estão apresentadas a seguir:

A-map: representa o *status* de alocação para toda a *heap.* Cada bit com valor 1 representa um bloco de memória ocupado por objeto. Em uma alocação, a CBT (*Complete Binary Tree*) é utilizada para identificar a área livre mais adequada para a alocação de um objeto. Sua ação baseia-se na análise dos dados em *A-map* para verificar se existe espaço suficiente para comportar o objeto.

B-map: indica o tamanho de cada objeto alocado na *heap.* As bordas de cada objeto recebem zeros e o corpo uns, exceto pelo primeiro objeto, que não possui borda inicial igual a zero. Este processo é realizado pelo *hardware B-unit* que realiza a inversão de bits em *B-map* de acordo com o status momentâneo de *A-map*. O vetor abaixo ilustra a alocação de três objetos, dos quais os dois primeiros utilizam 4 blocos e o terceiro 7 blocos.

111011101111110

M-map: indica o número de referências aos objetos permitindo definir quais objetos são válidos na *heap*. Em uma abordagem *Mark-sweep*, um dos três vetores que compõem *M-map* é utilizado com elemento de marcação de objetos alcançáveis. Neste vetor, um valor igual a 1 é inserido no bit correspondente ao bloco inicial do objeto válido na *heap*, os demais bits devem ser iguais a 0. Para o exemplo acima o valor de M-map seria:

100010001000000

Em uma abordagem *Reference counting* é necessária a utilização dos três vetores de M-map. Cada coluna em M-map corresponde a um contador de referências para um objeto. Mantendo o exemplo dado, assim ficaria M-map se os três objetos tivessem os valores 2, 3 e 4 respectivamente para seus contadores.

000000001000000
100010000000000
000010000000000

Sempre que um ciclo de *Mark-sweep* ocorre ou que o contador de um objeto atinge o valor zero, o status resultante de M-map é utilizado por outro componente, *Bit-Sweeper*, que realiza a atualização do vetor *A-map*, gerando posteriormente uma atualização também de *B-map* conforme foi explicado.

A fig. 2.6 mostra uma visão da aplicação do sistema proposto por Srisa-an, onde é possível perceber que devido ao tempo de propagação do circuito combinacional o co-processador possui um limite ótimo para tamanho da memória a ser gerenciada. Srisa-an informa que em sistemas com grandes requisitos de memória é necessário, porém viável, utilizar a instanciação de co-processadores conforme o número de módulos de memória a serem gerenciados.

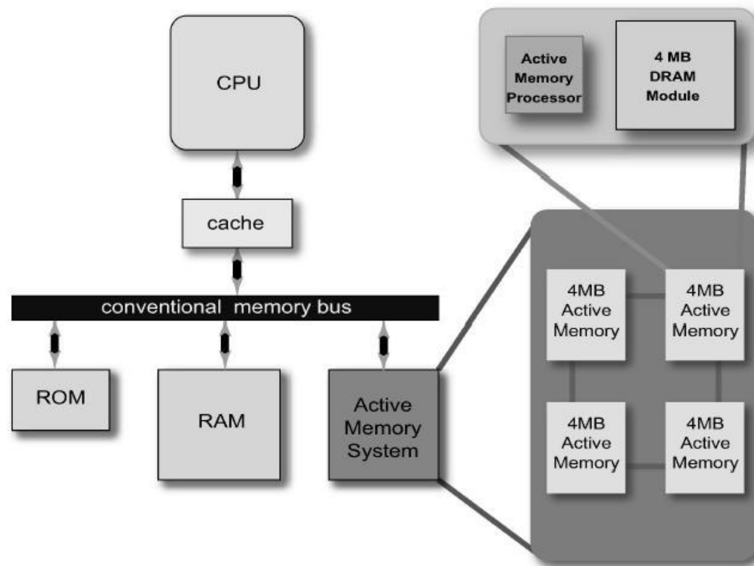

Figura 2.6: Aplicação do sistema de [SRI 2003]

O *hardware* combinacional *Active Memory Processor* (AMP) possui um alocador rápido e determinístico, capaz de realizar uma alocação em menos de 20 ciclos. Além disso, seu *Mark-sweep* executa em tempo constante (153.000 ciclos para uma memória de 4MB).

2.4.2.3.2 Lin Co-processor

Lin [LIN 2000] implementou outro *garbage collector* em *hardware*, baseado também em *Reference counting*, que se comunica com o processador principal através de um barramento. Quando o processador emite uma operação de acesso à memória por este barramento o módulo de gerência recebe a requisição e processa a operação. As operações de acesso à memória são executadas com prioridade máxima pelo módulo de gerência. Com uma prioridade menor o sistema executa em *background* a tarefa de *garbage collection*. A fig. 2.7 mostra em alto nível de abstração a arquitetura desenvolvida por [LIN 2000].

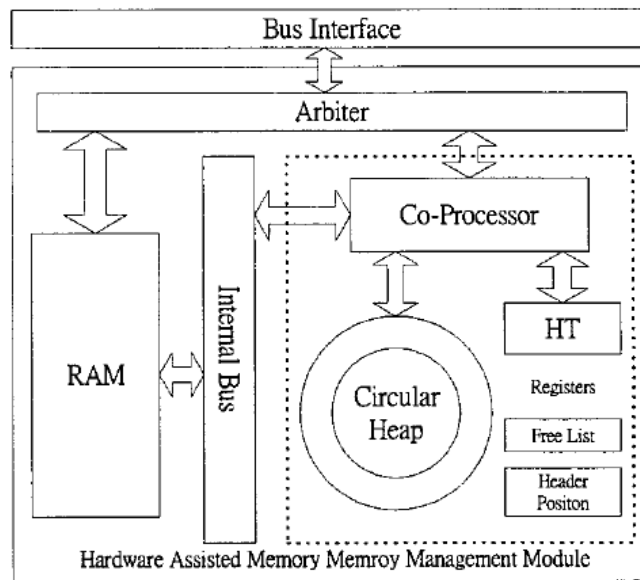

Figura 2.7: Arquitetura proposta por [LIN 2000]

Na fig. 2.7, o árbitro é a porta de entrada e saída do módulo de gerência de memória. Quando o processador envia uma requisição o árbitro a decodifica e extraí o bytecode nela contido. Se o bytecode corresponder a uma instrução de manipulação da heap, o árbitro a transfere para o co-processador que realiza a operação correspondente e se houver algum valor de retorno o devolve ao árbitro. Caso o bytecode seja uma instrução que não exija acesso à heap, então ela é executada diretamente através do acesso a uma memória RAM separada, onde são armazenadas as informações estáticas do programa. Um barramento interno é usado para conectar a memória principal à heap que foi organizada em uma estrutura circular para satisfazer requisitos de tempo real. Como o algoritmo utiliza uma estratégia de compactação da heap para eliminação de fragmentação externa, a implementação da heap em forma circular permite que qualquer operação seja realizada sobre a heap, mesmo quando a compactação estiver em curso. Isto permite principalmente que uma alocação não necessite aguardar pelo término da compactação, uma vez que os dados sempre podem ser armazenados na outra extremidade da heap alocada. Para gerenciar a heap circular e o processo de compactação são necessárias algumas estruturas adicionais, as quais estão detalhadas na fig. 2.8.

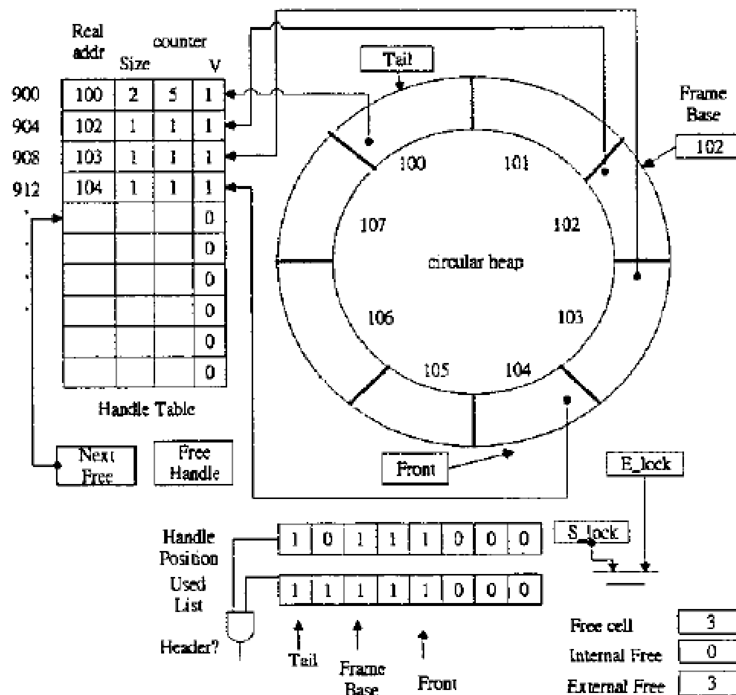

Figura 2.8: Estruturas adicionais do método de [LIN 2000]

Mais à esquerda na fig. 2.8 está a *Handle Table* onde são armazenadas todas as informações sobre os objetos localizados na heap. Cada endereço de uma entrada da *Handle Table* corresponde à referência ao objeto cujas informações foram postas naquela entrada. Logo, o endereço desta entrada é o valor atribuído a todas as variáveis do programa vinculadas a este objeto. As demais informações mantidas pela *Handle Table* são o endereço real de alocação do objeto (na heap), o tamanho do objeto, o contador de referências e um bit de validade referente à própria entrada. Segundo Lin, a principal vantagem do uso de uma *Handle Table* é a não necessidade de ter que vasculhar a pilha para atualizar todas as referências a objetos que foram movidos na heap devido à compactação.

O vetor *Used List* é usado para armazenar o estado de cada célula na heap circular. Cada bit indica se a célula está ou não livre. O vetor *Header Position* informa que células correspondem ao endereço inicial de um objeto. Em cada primeira palavra de um objeto é armazenado um ponteiro que indica o endereço da entrada correspondente ao objeto na *Handle Table*. Quando a heap é compactada, este ponteiro é utilizado para realizar a atualização do endereço real do objeto na *Handle Table*. Os vetores *Used List* e *Header Position* são combinados para informar à compactação o tamanho do objeto e se a célula é o endereço inicial de um objeto. Existem também no módulo alguns registradores especiais:

- *S_lock* e *E_lock*: são usados para indicar qual objeto está sendo movido correntemente pela compactação.
- *Free Handle*: armazena o número de entradas livres na *Handle Table*.
- *Next Free*: informa a próxima entrada livre da *Handle Table*.
- *Free Cell*: informa o número total de células livres.
- *Frame Base*: informa o endereço base na heap do *working set* corrente.
- *Front*: informa o início da heap alocada.
- *Tail*: informa o fim da heap alocada.

- *Internal Free*: informa o número total de células livres dentro da heap alocada.

Neste sistema a compactação é disparada sempre que o valor de *internal free* for maior ou igual ao valor de *free cell*.

2.4.2.4 Gerência de Memória em Software

A origem do desenvolvimento de sistemas para gerência dinâmica de memória para sistemas embarcados está relacionada ao desenvolvimento de algoritmos em *software*. A gerência por *software* apresentava indícios de maior flexibilidade para o projeto de tais sistemas, já que as plataformas em *hardware* eram consideradas suficientemente rápidas para satisfazer questões de desempenho [HAN 2002]. Entretanto, com o avanço no campo das aplicações embarcadas alguns parâmetros de projetos sofreram modificações para obtenção de *software*s mais robustos, capazes de gerar impactos bastante reduzidos sobre os custos finais para suporte às novas aplicações.

Neste sentido, em alguns casos, pequenos detalhes passaram a ser incorporados a propostas antigas, recriando-as para que pudessem atingir resultados significativos na era das modernas aplicações. A exemplo disto, são citados alguns projetistas:

- Bacon [BAC 2003] desenvolveu um *copying* collector baseado no algoritmo de Baker [BAK 93], cuja principal vantagem está na incrementalização do processo de evacuação dos objetos. Para isto, Bacon fez com que as barreiras de leitura (necessárias para impedir a aplicação de acessar um objeto em transição do *From-Space* para o *To-Space*) fossem incorporadas junto ao código da aplicação através de algumas modificações sobre o compilador, eliminando esta tarefa de seu *garbage collector*;
- Fuhrmann [FUH 2004], desenvolveu o *garbage collector* para o seu processador Java, Komodo [KRE 99], baseando-se também no algoritmo de Baker. O *garbage collector* foi implementado em *software* e executa sobre um *slot* de thread dedicado, controlado por um escalonador para atender requisitos de tempo real;
- Por fim, Donahue [DON 2001] apresentou um algoritmo para sistemas embarcados de tempo real que utiliza um sistema de desfragmentação postergada da heap, para aumentar a performance de seu *buddy system*. A sua estratégia consiste em postergar a recombinação de blocos, que segundo ele é responsável por uma significativa queda de performance em programas Java que criam objetos com tempo de vida curto. Assim, a recombinação só ocorre quando efetivamente uma grande área contígua de memória é necessária.

2.4.2.4.1 Reciclagem de Objetos

Deters [DET 2004] apresentou uma técnica inteiramente inédita de reciclagem de objetos com intuito de reduzir os recursos necessários em memória. Esta técnica consiste em fazer com que todos os objetos mortos de um determinado tipo *t* sejam armazenados em uma lista *T* exclusiva para objetos do tipo *t*. Posteriormente, quando um objeto do mesmo tipo *t* é criado, um dos objetos na lista *T* é reciclado para se tornar o novo objeto criado do tipo *t*. Neste contexto, uma rotina padrão de alocação de memória só necessita ser invocada quando a lista T encontra-se vazia. Entretanto, aparentemente, identificou-se um importante problema com esta estratégia que está relacionado à alocação de objetos que

possuem *arrays* como variáveis de instância. Como o *array* pode ser de tamanhos diferentes é possível que dois objetos de um mesmo tipo possuam tamanhos diferentes, dificultando assim o processo de reciclagem de objetos. Este problema restringe, portanto, a aplicação desta estratégia a uma parcela de aplicações bem específica. Outro problema menos importante desta técnica é o *overhead* para se manter um ponteiro para cada objeto, necessário para realizar a formação das listas utilizadas pelo algoritmo.

2.4.2.4.2 Algoritmo de Ritzau

Ritzau [RIT 2001] desenvolveu em linguagem C um algoritmo *Reference counting* tempo real para gerência dinâmica de memória em *software*. Seu algoritmo é capaz de lidar coerentemente com a recursividade necessária para a atualização dos contadores de objetos filhos de um objeto desalocado, baseando sua solução na estratégia de *Lazy freeing* proposta por Weizenbaum [WEI 1963]. Além disso, em seu algoritmo não implementa qualquer estratégia de compactação da heap. Isto elimina boa parte do *overhead* em memória e tempo de execução com a manutenção de *handles*. Para que a heap não sofresse fragmentação externa, foi utilizado um sistema de blocos encadeados de memória, o que fez com que existisse fragmentação interna dentro do bloco. Segundo Ritzau, a vantagem da fragmentação interna em relação à fragmentação externa é que a interna é previsível. Em seus experimentos Ritzau identificou uma fragmentação interna de meio bloco por objeto.

Conforme um objeto pode necessitar mais de um bloco para seu armazenamento, Ritzau utilizou uma palavra por bloco para realizar o encadeamento de blocos de um mesmo objeto. Em seu sistema o tamanho do bloco foi projetado para ser parametrizável. Cada objeto alocado possui em seu primeiro bloco um cabeçalho que consome duas palavras deste bloco, além da palavra utilizada para realizar o encadeamento de blocos. Estas palavras informam o contador de referências e o tipo do objeto.

Os blocos utilizados para realizar a alocação de um objeto podem ser obtidos de duas listas encadeadas mantidas pelo sistema: a *Free List* e a *To-Be-Free List*. Na inicialização do sistema uma rotina realiza o encadeamento inicial de todos os blocos disponíveis na memória, gerando assim a *Free List*. Após a inicialização a *To-Be-Free List* está vazia. Sempre que um objeto deve ser alocado o sistema realiza primeiro a busca na *To-Be-Free List* e não havendo blocos nesta lista então os blocos são obtidos na *Free List*. Seguindo uma estratégia similar a de Deustch-Brobow [DEU 1976] (*Deferred Reference counting*), quando o objeto é desalocado seus blocos são adicionados à *To-Be-Free List.* Quando o alocador deseja utilizar os blocos contidos nesta lista ele precisa chamar a *decChildren* que recebe como parâmetro de entrada sempre o primeiro bloco da *To-Be-Free List.* Esta função foi projetada para identificar o tipo do objeto ao qual o bloco pertence e as palavras contidas naquele bloco que mantém referências para objetos filhos. Assim, todos os objetos filhos referenciados pelo bloco passado como parâmetro são também adicionados à *To-Be-Free List.* Segundo Ritzau, este processo é determinístico, uma vez que o número máximo de objetos filhos a serem adicionados à *To-Be-Free List* é no pior caso dependente do tamanho do bloco. Ritzau diz também que, em processos de alta prioridade, caso seja necessário aumentar a velocidade da alocação, é possível garantir que o número de blocos necessários para realizar a alocação fique disponível em *Free List*. Para isto, é necessário que um escalonador execute com um nível intermediário de prioridade uma função chamada *rc_prealloc*, cujo propósito é transferir blocos da *To-Be-Free List* para a *Free List*. O número de blocos a serem transferidos da *To-Be-Free List*

para a *Free List* é calculado através de equações tais como as apresentadas por Henriksson [HEN 98]. Estas equações levam em conta diversas características específicas das aplicações, entre as quais a mais importante e mais difícil de ser obtida é a mínima quantidade de memória necessária para a heap.

Um dos problemas a serem encontrados na implementação de um algoritmo como o de Ritzau e em sistemas de tempo real de um modo amplo é o tratamento de *arrays*. Arrays podem ser extremamente grandes, por isso seria ideal que seu armazenamento fosse sempre realizado de forma contígua, uma vez que de outro modo o custo para acessar uma posição do *array* poderia ser bastante elevado. Entretanto, somente sistemas que utilizam compactação de memória permitem que *arrays* fiquem sempre dispostos de maneira contígua. Em sistemas que não utilizam compactação de memória, tais como o de Ritzau, que utiliza uma memória dividida em blocos, torna-se necessária alguma estratégia eficiente para armazenamento de *arrays*, uma vez que a implementação de uma lista encadeada exigiria o trilhamento de diversos blocos para realizar acesso a um dado, gerando assim uma significativa queda de performance para o sistema.

Pensando neste problema, Siebert [SIE 2000] implementou uma estratégia eficiente para armazenamento e acesso a *arrays* em seu sistema baseado em blocos de memória. Sua técnica consiste em armazenar *arrays* em uma estrutura de árvore, onde os dados do *array* (conteúdo propriamente dito) são armazenados apenas nas folhas da árvore, conforme o *array* apresentado na fig. 2.9 abaixo, que contém 11 elementos (um por palavra) distribuídos em 5 blocos de 16 bytes.

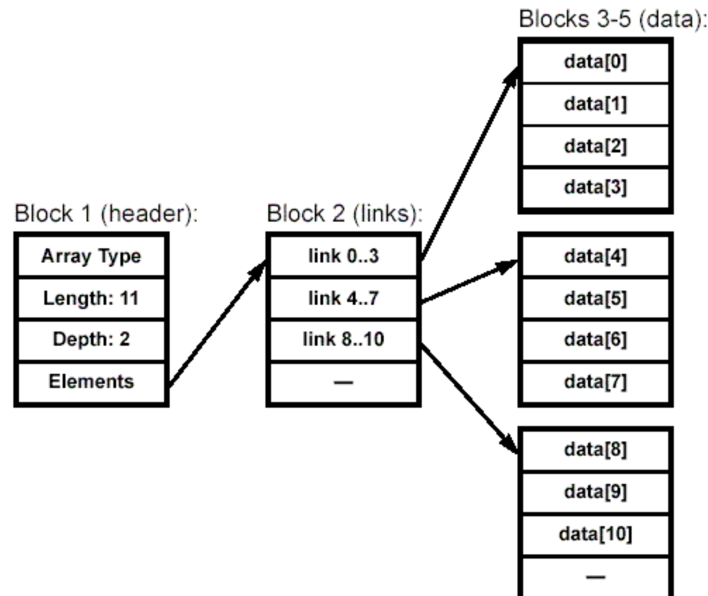

Figura 2.9: Estrutura em árvore de [SIE 2000]
para armazenamento de *arrays*

O uso da estrutura em árvore resulta em um desempenho $O(\ln(size))$ para acesso a qualquer elemento de um *array*, onde *size* é o tamanho do *array*. Este custo ainda é alto comparado com o custo constante para realizar acesso a um elemento quando o *array* está disposto contiguamente. Entretanto, Siebert diz que em um sistema com blocos de 32 bytes e uma heap de 16 MB, a profundidade da árvore jamais ultrapassa 7.

2.4.2.5 *Garbage Collectors* de Baixa Energia

Em [CHE 2002a] e [DEL 2002] são feitos alguns experimentos visando reduzir a quantidade de energia consumida em ambiente embarcado Java. O principal alvo desta proposta é a redução de energia consumida por correntes de fuga assim como por correntes dinâmicas em dispositivos de memória. Para isto, é utilizado um algoritmo de compactação juntamente com um algoritmo de desalocação (*Mark-sweep*) visando manter todos os dados nas porções iniciais da memória. Como a memória é dividida em vários bancos é possível desligar os bancos que não contêm dados válidos para a aplicação, reduzindo assim a energia global do sistema simplesmente pela eliminação da energia consumida apenas para manter os bancos de memória ligados. Os experimentos são realizados sobre a KVM, uma JVM desenvolvida pela Sun Microsystems especificamente para sistemas embarcados [SUN 2005a]. A KVM implementa dois algoritmos de *Mark-sweep*, um com compactação e o outro sem compactação. O algoritmo que realiza compactação faz uma diferenciação entre objetos permanentes e objetos ditos dinâmicos. Para isto, uma certa quantidade de memória é reservada no fim da heap para alocação dos objetos permanentes, esta área é chamada *permanent space*. Os principais benefícios da utilização desta área é que ela não é marcada (*mark* fase), varrida (*sweep* fase), ou compactada. O algoritmo de compactação usado pela KVM é um *Break Table-based algorithm* [HAD 1967]. As vantagens deste algoritmo segundo [CHE 2002a] são:

- Não há necessidade de espaço extra para manter a informação de relocação.
- Objetos de todos os tamanhos podem ser tratados.
- A ordem dos objetos é mantida.

Entretanto, a principal desvantagem do uso deste algoritmo é que tanto a ordenação da tabela *(break table)* quanto a atualização das referências são operações custosas tanto em termos de tempo de execução quanto em energia.

Segundo o modelo de tratamento da memória, qualquer dos bancos do sistema pode estar em um dos três modos de operação:

- Read/write: quando uma operação de leitura ou escrita está sendo realizada sobre o banco. Há consumo de energia dinâmica para a pré-carga das *bitlines* e também para a leitura através do *sense amplifier* [RAB 96]. Também há consumo de energia relativa à corrente reversa das memórias ligadas.
- Active: quando o banco está ativo (armazenando dados), mas não está realizando nenhuma leitura ou escrita. Neste modo é consumida energia dinâmica apenas para a pré-carga. Durante o restante do tempo a única energia consumida é a energia relativa à corrente reversa.
- Inactive: quando o banco não contém dados necessários ao sistema. Devido ao mecanismo de redução da corrente de fuga, quase não há consumo de energia (a corrente de fuga é extremamente baixa) e também não há ocorrência de consumo de energia dinâmica.

O custo para trazer qualquer banco de memória (para bancos de 16KB) do estado *inactive* para o estado *active* é de 350 ciclos na arquitetura alvo, que é um SoC com um processador embarcado microSPARC-Iiep 100MHz de 32 bits com um pipeline de cinco estágios estilo RISC, que implementa a especificação da arquitetura SPARC v8. Este acréscimo em execução para realizar a realimentação do bloco resultou ainda em um aumento de 5.2% em energia. A fig. 2.10, mostra a

disposição física e lógica dos componentes da arquitetura SoC utilizada em [CHE 2002a].

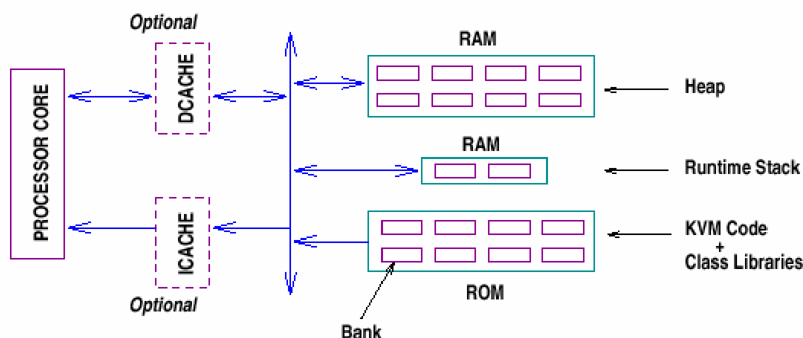

Figura 2.10: Arquitetura de [CHE 2002a]

Na fig. 2.10, o código da KVM e bibliotecas de classe são armazenados em um ROM (com 128KB de tamanho) numa configuração bastante típica, sendo que esta ROM é dividida em oito bancos de memória. Seletivamente, são ativados apenas os bancos que contêm bibliotecas acessadas pela aplicação.

A heap é mantida em uma RAM (de 128KB) também dividida em oito bancos de memória que além dos objetos armazenam também o código da aplicação. Uma diferença da arquitetura de [CHE 2002a] para a arquitetura utilizada neste trabalho é a utilização de uma outra RAM exclusivamente para manter a pilha principal do programa. De acordo com o crescimento da pilha principal também é possível desligar um dos bancos utilizados para seu armazenamento.

Para realizar a liberação de memória e a compactação da heap é utilizado um sistema de *work-scheduling*[2] chamado *K-allocator collector*, onde o GC é invocado a cada k alocações realizadas. O valor de k depende basicamente das aplicações a serem utilizadas, ou ainda mais especificamente da taxa com que a aplicação gera objetos inacessíveis. Neste ponto, [CHE 2002a] reforçou que o valor de k deve ser exatamente calculado para um melhor desempenho da técnica, já que muitos fatores positivos do sistema podem ser invertidos quando k é mal definido. Por exemplo, uma freqüência de execução desnecessariamente elevada do GC poderia resultar em um aumento na energia consumida pela ROM, RAM e processador. Uma alternativa a este problema seria a utilização de um algoritmo *Reference counting* para realizar a *garbage collection*, já que este algoritmo realiza um esforço proporcional ao número de objetos que vão se tornando desnecessários.

---

[2] Working scheduling, na terminologia usada pela comunidade que estuda garbage Collection, significa o escalonamento do GC segundo uma determinada quantidade de trabalho realizado, ou seja, o fator que determina quando o GC é executado é o volume de atividades realizadas pela aplicação.

2.4.2.6 Análise Comportamental e Requisitos para as Aplicações Orientadas a Objetos

Persson [PER 2000a] realizou um estudo para definir o WCET (*Wort Case Execution Time*) e o *Worst Case Memory Requirements* (WCMR) de uma aplicação orientada a objetos. Como produto deste estudo, ele definiu uma técnica que permite prever a quantidade de memória necessária para a execução de qualquer programa orientado a objetos, sendo esta informação fundamental para escalonadores que executam algoritmos de *garbage collection* como uma tarefa do sistema, reservando memória suficiente para que a aplicação não sofra nenhuma perda de *deadline*. Além disto, tais informações permitem também antecipar alguns requisitos de projeto de *Garbage Collectors*. Devido à dificuldade para prever estaticamente a quantidade de memória necessária para uma aplicação, como conseqüência principalmente da variação ocorrida sobre os dados de entrada, a técnica de Persson depende que algumas anotações sejam realizadas no código fonte do programa. Estas anotações são inseridas manualmente como comentários no código pelo programador. Persson desenvolveu algumas regras e uma ferramenta que utiliza tais regras para estimar o WCMR.

Uma possível implementação identificada pelo presente trabalho consistiria em empregar o uso da técnica de Persson para fornecer a quantidade de memória ao escalonador usado por Ritzau, isto permitiria executar com precisão a função *rc_prealloc* realizando a reserva de uma quantidade específica de memória livre para aplicações mais críticas. Neste sentido, considerando a arquitetura alvo deste trabalho, uma alternativa para a realização da gerência dinâmica de memória no FemtoJava seria uma descrição Java do algoritmo proposto por Ritzau.

Por fim, outro fator importante para o projeto de *Garbage Collectors* é a caracterização das aplicações a serem suportadas. Kim [KIM 2000] realizou um estudo do comportamento de programas Java com relação à utilização de memória. A análise foi feita através de *traces* de referências realizadas pelas aplicações disponíveis no *benchmark* SPECjvm98 [HUD 2005], rodando com um compilador JIT. De particular interesse para este trabalho, Kim apresentou características gerais para as aplicações analisadas, as quais estão apresentadas na tab. 2.1 abaixo.

Tabela 2.1: Dados gerais levantados por [KIM 2000] para as aplicações contidas no *benchmark* SPECjvm98

	Compress	**Jess**	**DB**	**Javac**	**MTRT**	**Jack**
Classes carregadas (usuário /total)	27 / 87	154 / 219	18 / 81	156/220	40/102	67/131
Métodos invocados ($x10^3$)	225.961	101.884	114.282	89.887	280.340	43.795
Bytecodes executados ($x10^3$)	12.474.021	1.820.852	3.700.062	1.953.961	2.122.522	2.996.618
Número de objetos alocados	7.446	8.131.609	3.262.899	6.244.896	6.695.116	6.955.528
Tamanho médio de objeto (byte)	14.823	40	31	36	25	31
Heap mínima (MB)	15	2	12	12	9	2
Heap máxima (MB)	106	308	98	217	161	203

Embora o benchmark SPECjvm98 não tenha sido projetado para sistemas embarcados, considera-se as informações apresentadas na tab. 2.1 relevantes para este trabalho, uma vez que foram poucas as estimativas sobre comportamento de aplicações Java encontradas na literatura. Isto demonstra uma escassez deste tipo de informação, principalmente no que diz respeito a aplicações embarcadas.

Uma constatação importante sobre os dados apresentados na tab. 2.1 é que as aplicações Java de um modo geral não possuem requisitos constantes para sua execução. Mesmo assim, alguns fatores tais como o tamanho médio dos objetos alocados serviram como referência para o desenvolvimento das soluções propostas neste trabalho. Especificamente, o tamanho médio dos objetos alocados propiciou uma base importante para a realização de estimativas tanto para a definição dos requisitos mínimos de memória esperados, quanto para o cálculo do *overhead* em memória produzido por cada solução investigada.

Uma tentativa de realizar uma comparação entre o desempenho das aplicações apresentadas na tab. 2.1 e o desempenho obtido por algumas aplicações embarcadas utilizadas neste trabalho (apresentadas no capítulo 4) foi desenvolvida como forma de estabelecer uma relação entre cada tipo de aplicação. Contudo, como as aplicações embarcadas utilizadas neste trabalho geram seus resultados em função do tempo consumido para execução, esta comparação foi realizada apenas para as estimativas independentes do tempo de execução existentes para cada tipo de aplicação.

Neste sentido, identificou-se que o número máximo de objetos presentes na memória durante a execução das aplicações apresentadas na tab. 2.1, que é resultado da relação entre a heap mínima necessária para execução (linha 7 tab. 2.1) e o tamanho médio dos objetos alocados (linha 6 tab. 2.1), é em média 187 vezes maior que o obtido para as aplicações embarcadas analisadas.

Outra diferença, também neste sentido, está relacionada à quantidade mínima de heap para execução. Em média, as aplicações embarcadas utilizadas neste trabalho requerem um volume de heap aproximadamente 48 vezes menor que o utilizado pelas aplicações não embarcadas apresentadas na tab 2.1. Em geral, muitos projetistas de sistemas para gerência de memória, com foco para sistemas embarcados, tais como [DEL 2002] utilizam em seus sistemas aplicações que requerem volumes de memória considerados pelo ponto de vista deste trabalho inaceitáveis para sistemas embarcados.

Levando em consideração a relação entre a heap máxima (linha 8 tab. 2.1), que corresponde à quantidade total de memória alocada por cada aplicação, e número de bytecodes executados (linha 4 ab. 2.1) que pode ser concebido como um parâmetro de tempo para esta análise, observou-se que as aplicações não embarcadas alocam seus objetos aproximadamente 15000 vezes mais intensamente que as aplicações embarcadas utilizadas pelo presente trabalho. Isto revela que as aplicações embarcadas analisadas requerem menos atividade por parte do mecanismo de gerência dinâmica de memória.

Por último, considerando o tamanho médio dos objetos alocados pelos dois tipos de aplicações, que representa a relação entre a heap mínima necessária para execução e o número máximo de objetos presentes na heap em tempo de execução (demonstrado ser bem superior para as aplicações não embarcadas), são bastante semelhantes. Isto permite concluir que o esforço realizado para manipulação de um objeto é independente do tipo de aplicação utilizada. Dessa forma, conclui-se que, de um modo geral, o projeto de sistemas para gerência de

memória deve permanecer focado apenas nos requisitos já discutidos com relação a sistemas embarcados (seção 1.2 deste texto).

Em [FON 2002] é feita uma análise quantitativa de padrões presentes nas alocações e desalocações de memória em programas Java. A análise foi realizada através da modificação do código fonte do JDK, obtendo assim um *trace* da execução dos programas. Os *benchmark*s utilizados foram o Java2D demo e o Forte4j ambos da SUN, um jogo chamado Othello e o SPECjvm98. Java2D é uma aplicação demo inclusa na versão 1.3 do JDK. Forte4j é um ambiente de desenvolvimento de Java, onde os dados foram obtidos através da execução (uso) normal do *software*. As informações coletadas foram: tipo da alocação (classe, objeto ou *array*), tamanho em bytes do espaço alocado e o valor do *handle* (ponteiro) retornado. Suas principais conclusões foram:

- Embora *arrays* possam atingir uma larga variação de tamanho, estes em geral não são maiores que 1024 bytes.
- 90% das alocações são de tamanhos menores que 256 bytes.
- Uma das causas principais da baixa performance de Java é o extensivo uso de alocações dinâmicas de memória durante a criação de objetos e *arrays*.

Em [DYK 2002], um estudo mais detalhado do comportamento da memória é realizado considerando também o *benchmark* SPECjvm98. Alguns dados considerados mais relevantes para este trabalho são apresentados na tab. 2.2.

Tabela 2.2: Dados produzidos por [DYK 2002]

	JVM clássica					
Program	Total Objetos Alocados	Total Memória Alocada (MB)	Máximo tamanho objeto (MB)	% objetos maiores 4096 bytes	Alocação latência mínima (ciclos)	% alocações < 1000 ciclos
Compress	11624	105.33	3.000	1.66	410	92.25
Db	3215782	60.205	1.116	0.01	407	98.99
Jack	6871646	134.76	0.034	Negligível	387	94.46
Javac	5943930	139.82	0.034	Negligível	-	-
Jess	7939856	210.99	0.034	Negligível	388	90.20
Mpegaudio	15182	0.924	0.034	0.20	413	93.83
Mtrt	6644193	84.16	0.1526	Negligível	407	97.43

A análise dos resultados apresentados também fornece um noção importante dos custos relativos à gerência dinâmica de memória em programas Java. Em especial, algumas conclusões obtidas através desta análise são:

- O número de objetos alocados demonstra uma significativa atividade exercida pela gerência de memória. Entretanto, os resultados seriam ainda mais relevantes se o número total de objetos desalocados também tivesse sido apresentado.
- A quarta e quinta coluna informam que embora existam objetos grandes eles compõem uma pequena minoria em relação aos demais objetos também existentes.
- Com base na observação da duas últimas colunas, o custo em ciclos necessário para a alocação de grande parte dos objetos permite deduzir o *overhead* produzido pela alocação de memória.

Na tab. 2.2 é possível perceber que uma das aplicações do SPECjvm98 não foi utilizada ao longo de toda a tabela. Da mesma forma, os dados para a aplicação *javac* não são gerados para alguns itens da tabela. O motivo pelo qual estes dados não foram apresentados são desconhecidos segundo a análise feita pelo presente trabalho.

3 DESCRIÇÃO DAS SOLUÇÕES PROPOSTAS

Este capítulo apresenta o estudo realizado sobre duas implementações de algoritmos para gerência dinâmica de memória em *software*, desenvolvidas para uso para sistemas embarcados baseados em Java. Cada uma destas implementações engloba um alocador e um garbage collector. A primeira implementação foi desenvolvida objetivando custos menores para realização da gerência dinâmica de memória e a segunda implementação foi desenvolvida objetivando primariamente o alcance de um comportamento adequado a tempo real.

Considera-se que o ideal seria poder disponibilizar um número maior de soluções para o problema, uma vez que isto possibilitaria identificar com maior clareza, através de comparações entre as implementações, as melhores estratégias para realizar a gerência de memória, segundo requisitos típicos para sistemas embarcados, tais como desempenho, tempo-real, potência, energia e tamanho da memória utilizada. Sendo assim, não há garantias de que qualquer uma das soluções propostas neste trabalho seja ótima para algum requisito embarcado típico, uma vez que as duas soluções já alcançadas, foram desenvolvidas tendo como meta, em geral, um nível satisfatório para cada um destes requisitos.

O restante do capítulo apresenta a arquitetura das duas implementações obtidas, assim como a estratégia utilizada para integração dos módulos de gerência ao processador Femtojava e à metodologia SASHIMI. Por simplicidade, a sigla SGDM (Sistema de Gerenciamento Dinâmico de Memória), também será utilizada para se referir, em algumas situações, às implementações realizadas.

3.1 As arquiteturas de Base

Dos algoritmos clássicos (apresentados na seção 2.1), optou-se por construir apenas abordagens baseadas em *Reference counting*, uma vez que este algoritmo possui características desejadas para sistemas embarcados tais como a arquitetura alvo. Destas características as principais são:

- O algoritmo é incremental.
- Suas operações são simples e possuem um baixo tempo de resposta.
- O tempo de resposta de suas operações não depende do tamanho da memória.

É importante, no entanto, ressaltar que o fato de terem sido utilizados apenas algoritmos baseados em *Reference Counting* não significa que qualquer um dos algoritmos clássicos para gerência dinâmica de memória não possa apresentar um desempenho melhor para algum parâmetro típico de projeto de sistemas embarcados.

Da mesma forma, é importante observar que o uso de algoritmos do tipo Reference Counting como base para realização da gerência dinâmica da memória incorre na herança de algumas deficiências relevantes relacionas a este algoritmo. Entretanto, estas deficiências requerem soluções simples, sendo por isto adequadas para os objetivos perseguidos por este trabalho. Tais deficiências são:

- O algoritmo não suporta aplicações que implementem referências cíclicas. Contudo, algumas pesquisas já realizadas demonstram que, em se tratando de aplicações embarcadas, este tipo de referência de um modo geral nunca ocorre [RIT 2001].
- Dependendo da forma como é implementada a desalocação de um objeto, o decremento do contador de referências de todos os descendentes pode levar à desalocação de um número desconhecido de objetos na heap. Entretanto, conforme descrito no capítulo 2 deste texto (seção 2.4.1.7), Weizenbaum desenvolveu a técnica de *Lazy Freeing* para solucionar este problema.

Dentre as estratégias baseadas em *Reference counting* apresentadas no capítulo 2, o algoritmo de Lin (seção 2.4.2.3.2) foi o escolhido para formar a base para uma primeira versão do sistema. Os motivos da escolha deste algoritmo foram:

- Os dados sobre a alocação dos objetos ficam organizados em uma tabela central, cuja implementação foi considerada mais intuitiva para uma primeira versão do sistema. Além disso, o uso de tal tabela reduz significativamente o *overhead* gerado por constantes atualizações de referências a objetos (mantidas por elementos da aplicação) comuns em algoritmos que utilizam compactação.
- A organização dos objetos na memória é feita de forma simples diretamente em palavras e todas as palavras que contêm o corpo de um mesmo objeto alocado ficam sempre mantidas contiguamente na heap, maximizando o desempenho para a alocação e o acesso a qualquer parte do objeto. Outro benefício desta forma de alocação dos objetos é a economia em memória obtida para aplicações que lidam com objetos razoavelmente grandes. Isto porque o *overhead* em memória mantido para alocação de qualquer objeto é sempre o mesmo e corresponde a uma entrada da tabela para armazenamento dos dados sobre a alocação do objeto.
- Embora proposto originalmente em *hardware*, este algoritmo é o único encontrado que utiliza contagem de referências integrado a um mecanismo de compactação para reduzir os custos como a manipulação de objetos na memória.
- O algoritmo de compactação da memória é simples e produz um acréscimo ao custo total de execução relativamente baixo [LIN 2000].

Entretanto, a migração da proposta originalmente em *hardware* de Lin para uma versão em *software* requer algumas adaptações que produzem impactos significativos sobre a previsibilidade global do sistema. Ainda assim, o algoritmo permanece simples, sendo esperados custos para execução relativamente menores que os obtidos com outros sistemas baseados em *Reference counting*. Os impactos sobre a previsibilidade global do sistema estão relacionados sobretudo a dois fatores:

- Ao mecanismo de compactação, que na proposta original em *hardware* era executado paralelamente a outras operações com a memória por um

módulo dedicado. Em *software* esta compactação passa agora a executar de forma exclusiva, ocupando inteiramente a unidade de processamento e obrigando que todas as demais operações tenham que aguardar que a compactação termine para que a CPU possa ser liberada. De acordo com a estratégia utilizada nesta primeira versão, a cada execução do mecanismo de compactação de memória é possível que uma parcela significativa da memória tenha que ser acessada, o que resulta em uma complexidade vinculada ao tempo de resposta para a compactação próxima *n*, onde *n* é o tamanho da memória utilizada. Isto evidencia mais claramente o abalo causado pela compactação sobre a previsibilidade desta implementação.
- Ao mecanismo de desalocação, especificamente com relação ao cascateamento necessário para desalocação de todos os descendentes do objeto desalocado, pelos mesmos motivos descritos para o item anterior.

Com relação à segunda implementação, buscou-se priorizar o atendimento a restrições de tempo real, e entre as propostas existentes optou-se desta vez por construir uma base sobre um algoritmo desenvolvido originalmente para *software*. Assim, o algoritmo que mais influenciou na construção da base para a segunda implementação foi o de Ritzau (seção 2.4.2.4.2). Este algoritmo foi escolhido porque:

- Combina com simplicidade as principais técnicas para obtenção de tempo-real em algoritmos para gerência dinâmica de memória:
  - Lazy freeing
  - Deferred *Reference counting*
  - Permite o uso opcional de escalonadores para maior desempenho do algoritmo.
- Possui todas as suas operações construídas sob um nível de granularidade mínimo tanto para obtenção de tempo real quanto para aumento de desempenho.
- Não requer recursos sofisticados nem grandes volumes de memória.

3.2 Implementação 1 - Baseada no Algoritmo de Lin

Como a arquitetura de Lin foi implementada em *hardware*, foi necessário realizar algumas adaptações para que fosse possível migrá-la para *software*. A fig. 3.1 mostra a arquitetura alcançada através desta adaptação.

A fig. 3.1 apresenta uma *Handle Table* modificada que, em sua versão em *software*, passar a residir juntamente com outros dados na RAM. Esta nova tabela passou a ser chamada de *OST* (*Object Status Table*). Cada entrada da *OST* contém os seguintes campos, onde cada campo consome uma palavra de memória:

- *OAA*: Informa o endereço real de alocação do objeto na heap.
- *CC*: Armazena simultaneamente a classe (estaticamente é realizada a vinculação de um código numérico exclusivo a cada classe da aplicação) e o contador de referências do objeto.
- *Next*: Um ponteiro para a próxima entrada livre da *OST*.
- *Prev*: Um ponteiro para a última entrada da *OST* ocupada antes da entrada corrente.

Diferentemente da *Handle Table* de Lin, que utiliza um campo para armazenamento do tamanho do objeto, optou-se por armazenar apenas a classe do objeto e dela extrair o tamanho do objeto através de um mapeamento realizado dinamicamente por meio de uma tabela auxiliar que consiste de um vetor onde o índice de cada posição do vetor corresponde a uma classe da aplicação (conforme o código numérico atribuído a cada classe estaticamente) e o valor de cada elemento corresponde ao tamanho de uma instância desta classe. Este mapeamento reduz a quantidade de memória necessária já que elimina a necessidade de se manter uma palavra por objeto apenas para armazenamento do seu tamanho.

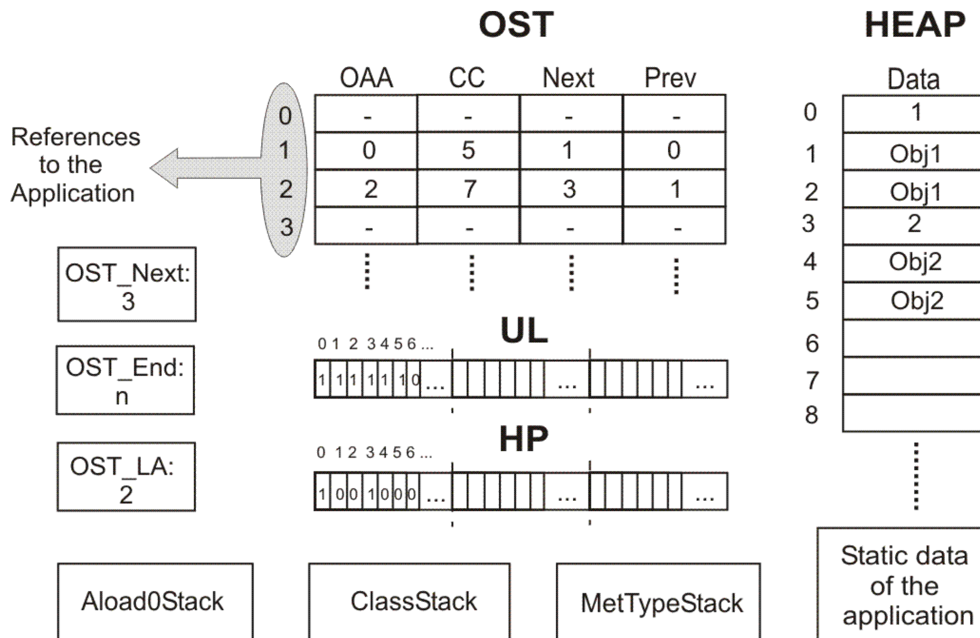

Figura 3.1: Implementação Baseada no Algoritmo de Lin.

Na fig. 3.1, os campos *Next* e *Prev* permitem a composição de duas listas dentro da *OST*, uma para as entradas livres e outra para as entradas ocupadas. Esta modificação foi necessária porque, na implementação de Lin, a cada objeto alocado, a posição da *Handle Table* utilizada corresponde sempre à posição seguinte (fisicamente) à última posição ocupada. Assim, Lin mantinha sua *Handle Table* compactada juntamente com a heap, sendo ambas compactadas paralelamente.

Quando os dados sobre um objeto são armazenados em uma entrada da *OST*, o endereço do campo *Next* da entrada alocada é atribuído à variável *OST_Next*, cujo valor será utilizado para definir a entrada da *OST* a ser utilizada durante a próxima alocação.

A variável *OST_End* informa a última entrada livre da *OST*. Quando um objeto é desalocado, sua entrada correspondente na *OST* é inserida no fim da lista de entradas livres e o endereço desta entrada é atribuído à variável *OST_End*.

A variável *OST_LA* informa a entrada da *OST* correspondente ao último objeto alocado na heap. Seu propósito é permitir o acesso ao endereço de alocação deste objeto, sua classe do objeto e conseqüentemente o seu tamanho. Com estes dados o algoritmo de alocação é capaz de realizar o cálculo do endereço inicial para a nova alocação, assim como determinar se existe espaço disponível na memória para satisfazer esta alocação. Existindo espaço na heap

para a alocação do objeto, a primeira palavra deve sempre ser reservada para armazenamento de um ponteiro para a entrada da *OST* onde estão as informações sobre a alocação do objeto. Este procedimento tem por objetivo auxiliar o processo de compactação da heap, permitindo que seja realizada a atualização do endereço de alocação do objeto.

Os vetores *Used List* (*UL*) e *Header Position* (*HP*) foram utilizados com os mesmos propósitos criados por Lin. Para que sua implementação em *software* fosse independente dos parâmetros de configuração utilizados para RAM, optou-se por implementá-los através de dois vetores de inteiros de 32 bits. Dessa forma, o acesso às posições de *UL* e *HP* precisou ser definido através de funções para mapeamento e mascaramento de bits. Estas funções são:

- *FMap*: define a palavra do vetor de inteiros onde está o bit alvo, correspondente à posição de *UL* ou *HP* a ser acessada.

$$FMap = pos / 32$$

- *FMsk*: define a máscara para isolamento do bit alvo.

$$FMsk = pos \% 32 \quad \text{(\% é a função módulo)}$$

Assim, para realizar o acesso à posição 65 de *UL* , Fmap informa que deve ser acessada a posição 2 do vetor de inteiros que representa *UL* e *FMsk* informa que deve ser utilizada uma máscara para isolamento do segundo bit (bit 1) desta posição.

O elemento *Aload0Stack* foi criado para fornecer suporte especial à instrução *aload_0*. Esta instrução funciona diferentemente de acordo com o contexto de execução. Se esta instrução for executada a partir de um método estático, sua função é carregar uma referência a objeto contida na variável local de número zero do frame do método corrente, para o topo da pilha principal do processador. No entanto, se esta instrução for executada a partir de um método dinâmico, sua função é carregar para o topo da pilha principal do processador o valor da referência para o objeto a partir do qual foi realizada a chamada ao método corrente. Esta dupla funcionalidade é reconhecida e tratada internamente pela JVM, mas ainda não tinha sido implementada na arquitetura do processador FemtoJava. Para criar suporte adequado a esta instrução, foi projetado um algoritmo que monitora as chamadas a métodos de instância e captura as referências aos objetos a partir dos quais foram realizadas tais chamadas. As referências capturadas são então empilhadas em *Aload0Stack*, antes do desvio para o método de instância. Durante a execução do método de instância, todas as instruções *aload_0* resultam na obtenção de uma cópia do topo de *Aload0Stack*, permitindo o acesso ao objeto que gerou a chamada de método. Este acesso normalmente é feito para manipulação de variáveis de instância do objeto.

Quando o método de instância retorna a sua chamada o valor no topo de *Aload0Stack* é desempilhado, devolvendo o contexto para o objeto que havia anteriormente invocado um de seus métodos.

Um suporte especial também foi necessário para determinar se o método corrente é estático ou dinâmico, uma vez que a estrutura do frame de método na arquitetura Femtojava não disponibiliza esta informação, conforme mostra a fig. 3.2.

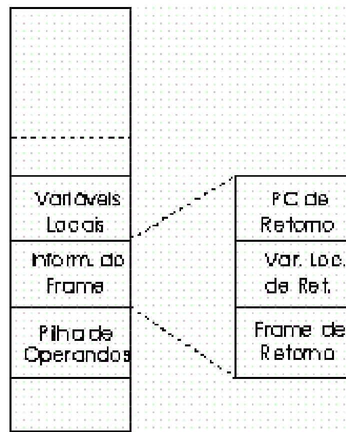

Figura 3.2: Estrutura do frame na arquitetura FemtoJava

Desse modo, optou-se em construir também uma pilha (*MetTypeStack*) para armazenamento da informação de tipo (estático ou dinâmico) de cada método invocado. Esta estratégia soluciona a duplicidade existente para a instrução *aload_0*, já que o valor correto a ser carregado para o topo da pilha do processador é definido pelo tipo do método corrente, armazenado no topo da pilha *MetTypeStack*.

Sob um ponto de vista menos abstrato, os elementos *metType* e *Aload0Stack*, fazem parte na realidade de uma estrutura de frame em *software* complementar para o processador Femtojava, que possui ainda um terceiro item para resolução de informações necessárias dinamicamente, conforme mostra a fig. 3.3.

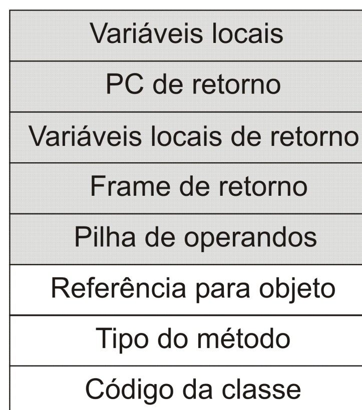

Figura 3.3: Estrutura de frame complementar para o Femtojava

3.2.1 Resolução da *Constant Pool*

Ao todo a JVM estabelece quatro principais operações para completa realização da gerência dinâmica de memória. Destas operações, três requerem informações geradas em tempo de compilação, armazenadas nos arquivos executáveis da aplicação. Entre estas informações a mais relevante para este trabalho é a *Constant pool*, que consiste em uma tabela de símbolos dividida em diversas entradas que são acessadas de acordo com um índice que nada mais é do que o operando de algumas instruções. Cada classe da aplicação possui sua própria *constant pool,* com informações básicas a execução do código Java. A

seleção da *constant pool* a ser acessada por requisição de alguma instrução é feita com base no método em execução. Logo, o acesso deve ser feito sempre à *constant pool* da classe a qual pertence o método corrente. Uma alternativa para resolver dinamicamente o acesso à *constant pool* é manter junto ao frame do método um ponteiro para o endereço de memória para onde foi carregada a *constant pool* relacionada ao método. Esta solução foi empregada na estrutura do frame projetada para o processador picoJava 2 [SUN 99], apresentada na fig. 3.4.

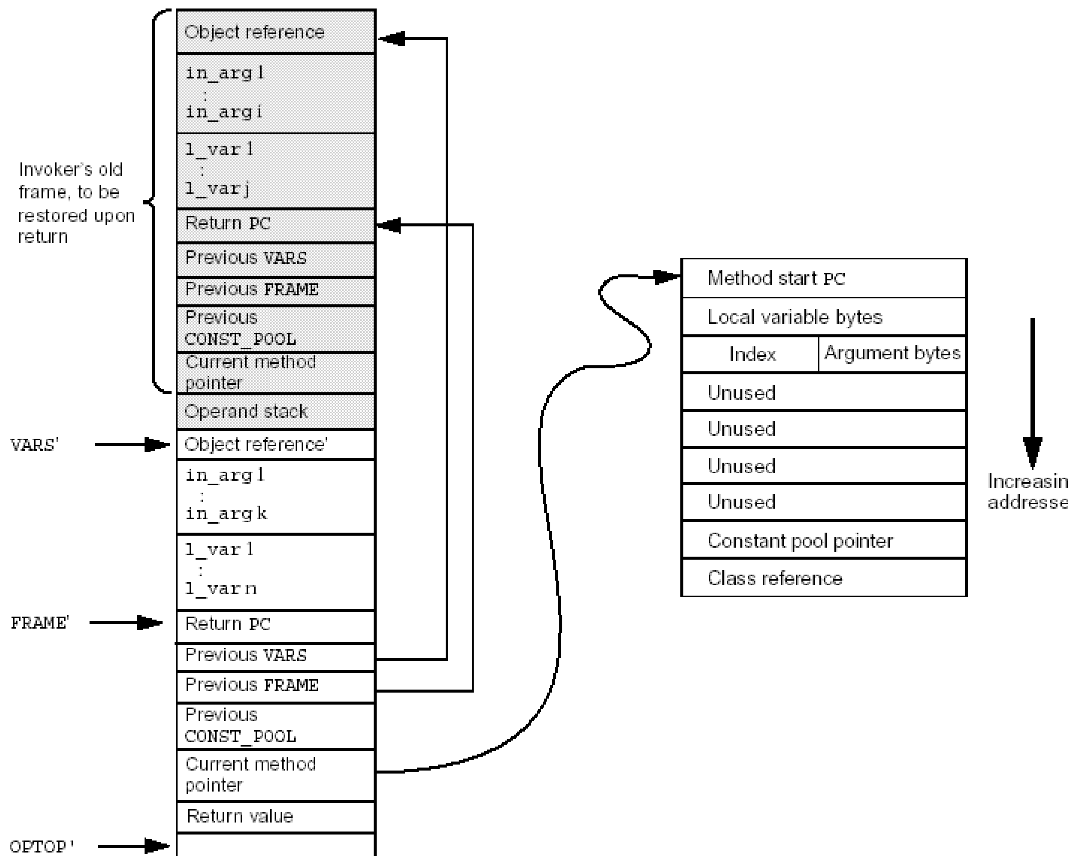

Figura 3.4: Estrutura do frame do picoJava 2

A fig. 3.4 torna possível algumas analogias com a fig. 3.3. A primeira semelhança é o uso de um campo do frame para armazenamento da referência ao objeto que criou a chamada ao método, caso o método não seja estático. Assim, se o valor deste campo não é nulo o frame corresponde a um método de instância ou objeto. Esta solução é um pouco mais simples do que projetada para o Femtojava, contudo optou-se por utilizar campos separados para informações de referência a objeto e tipo do método, em benefício de uma maior modularização do algoritmo de gerência de memória.

Outra analogia está relacionada à estratégia para acesso à *constant pool* nas duas arquiteturas. A cada frame empilhado o picoJava armazena endereços de contexto da *constant pool* do frame anterior e permite o acesso indireto à *constant pool* do frame corrente via acesso intermediário ao conjunto de informações do método. No modelo de acesso à *constant pool* criado para o FemtoJava, a parte complementar do frame contém o código numérico da classe

que é usado para acessar a localização da *constant pool* correspondente através da consulta a uma tabela que mapeia código da classe em endereço de *constant pool*.

3.2.2 Comentários sobre a Implementação

Uma dificuldade para migração da proposta original em *hardware* de Lin para a atual em *software* foi a preservação do fator tempo-real, já que nesta implementação não foi possível utilizar os benefícios da *heap* circular, uma vez que o processamento seqüencial do *software* impediria que alocações pudessem ocorrer enquanto a compactação estivesse em execução. Outra dificuldade foi o tratamento da desalocação de um objeto, onde o decremento do contador dos objetos descendentes foi implementado recursivamente. A recursividade e conseqüentemente o não-determinísmo existente para a desalocação poderiam ainda ser tratados com estratégias simples como a já desenvolvida por [WEI 1963]. Entretanto, como ainda permaneceria o indeterminísmo para a tarefa de compactação, optou-se em realizar a desalocação da forma tradicional, uma vez que esta é considerada a forma mais simples [WIK 2005].

3.3 Implementação 2 - Baseada no Algoritmo de Ritzau

Com base no algoritmo de Ritzau, desenvolvido em linguagem C, foi desenvolvida uma versão Java que também possui algumas adaptações para que se tornasse mais adequada à plataforma alvo. A fig. 3.5 mostra em alto nível a arquitetura da solução 2.

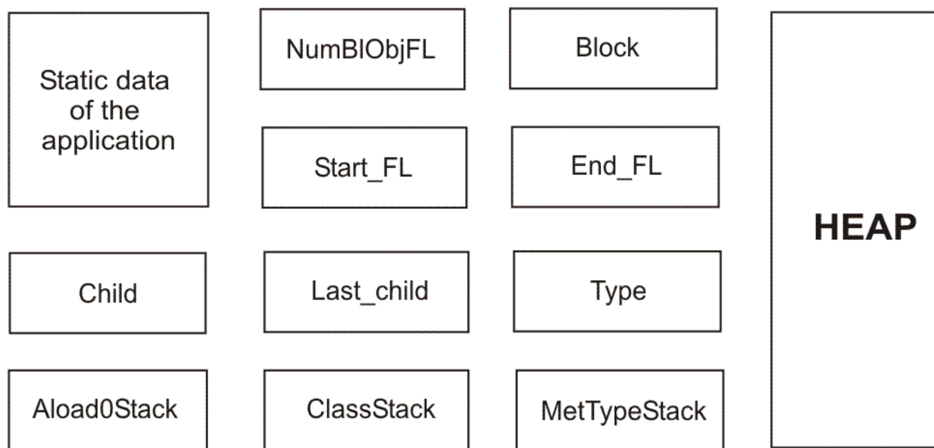

Figura 3.5: Implementação baseada no Algoritmo de Ritzau

Com base na fig. 3.5 é possível perceber que muitos elementos da primeira implementação estão sendo reutilizados na segunda. Entretanto, algumas das principais características que diferenciam esta implementação da primeira são:

- A memória (heap) é dividida logicamente em diversos blocos que são entregues à aplicação na forma de listas encadeadas. O tamanho do bloco é configurável, através de um parâmetro de compilação.
- Os blocos que compõem um objeto não precisam estar dispostos contiguamente na memória.
- Todas as operações do sistema sem exceção são determinísticas.

- Não há compactação de memória de forma que os objetos nunca mudam seu endereço de alocação durante todo o seu ciclo de vida.

Entre as alterações realizadas em relação à proposta inicial de Ritzau, buscou-se primeiramente modificar o cabeçalho de objeto, com o intuito de reduzir o *overhead* em memória para aplicações que alocam objetos em maioria pequenos, tipicamente menores que 32 bytes, como as analisadas por [DYK 2002]. Como resultado, foi realizada a união do contador de referências e classe do objeto em uma única palavra de memória no bloco de cabeçalho. Com isto, o *overhead* no bloco de cabeçalho é de duas palavras (levando-se em consideração uma palavra para armazenamento do ponteiro para encadeamento dos blocos). Para uma memória com uma largura de 32 bits e o uso de no máximo 2 blocos por objeto e 32 bytes/bloco, esta simples modificação resulta em uma economia de memória de 12%. A fig. 3.6 mostra à esquerda o cabeçalho de objeto usado por Ritzau e à direita o novo cabeçalho proposto.

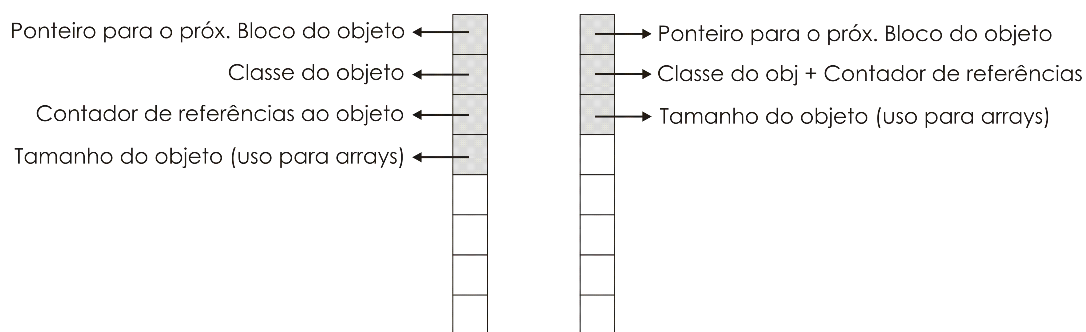

Figura 3.6: Modificações no cabeçalho de objeto

Como uma segunda adaptação, decidiu-se eliminar uma das listas disponíveis para obtenção de blocos usados para alocação de objetos, com base na observação de que a retirada de blocos da *Free list* deveria ocorrer apenas até que esta lista não contivesse mais nenhum bloco. Esgotados os blocos na *Free list*, a obtenção de blocos passaria a ser realizada exclusivamente na *To-be-Free List*. A diferença entre obter blocos na *Free list* ou na *To-be-Free List* é que no segundo caso existe necessidade de se chamar a função *DecChildren* (seção 2.4.2.4.2) sobre o bloco obtido nesta lista, o que certamente dada a complexidade desta função geraria uma diferença significativa entre os custos para obtenção de um bloco em cada uma das listas. Uma vez que não há controle sobre o ponto exato durante a execução da aplicação em que se esgotam os blocos na *Free list*, não é possível determinar o custo exato de cada alocação realizada pelo sistema.

Sendo assim, a adaptação seguinte consistiu em criar uma classe de objeto denominada de *NoRef* que não possui qualquer vínculo com a aplicação original, possuindo esta classe as seguintes características:

- O número de blocos necessários para armazenar uma instância desta classe é sempre igual a 1.
- Uma instância desta classe não possui referências para quaisquer outras instâncias de qualquer outra classe.

Por fim, o processo de inicialização foi modificado para que todos os blocos fossem inicializados para conter instâncias da classe *NoRef*. Deste modo, até que todos os blocos sejam alocados pelo menos uma vez, a função *DecChildren* receberá apenas blocos sem referências para outros objetos. Isto faz com que a

função seja sempre invocada, sem que contudo produza qualquer efeito sobre os demais objetos alocados na heap.

Dadas as modificações apresentadas nesta seção e considerando a fig. 3.5, o propósito de cada um dos elementos ilustrados é:

- *Start_FL*: endereço do primeiro bloco da lista de blocos livres.
- *End_FL*: endereço do último bloco da lista de blocos livres.
- *Type*: tipo ou classe do objeto que esta sendo reciclado a partir da *Free List*.
- *Block*: informa qual o bloco do objeto em processo de reciclagem está sendo processado pelo sistema de gerência.
- *Child*: informa a ordem do próximo descendente do objeto em processo de reciclagem a ter seu contador de referências decrementado.
- *LastChild*: informa a ordem do último descendente do objeto em processo de reciclagem a ter seu contador de referências decrementado.
- *NumBlObFL*: número total de blocos do objeto em processo de reciclagem.

3.4 Integração à Arquitetura Alvo

Uma vez que o processador FemtoJava está inserido em uma metodologia de geração semi-automática de *hardware* e *software* para sistemas embarcados, na qual *hardware* e *software* são gerados com recursos precisos para suporte a cada aplicação, a integração dos dois sistemas desenvolvidos à plataforma alvo foi projetada para seguir também esta estratégia.

A fig. 3.6 mostra como cada implementação foi integrada ao fluxo de projeto do ambiente SASHIMI, cuja síntese inicia com a descrição Java original da aplicação que é compilada para obtenção dos arquivos *class* onde estão contidos todos os *bytecodes* necessários para a criação e manipulação de objetos. Estes arquivos (.*class*) são enviados à ferramenta CAT (*Code Adapting Tool*) que realiza substituição dos *bytecodes* dinâmicos, sem suporte pelo processador FemtoJava, por *bytecodes* já suportados, que estão disponíveis na forma de métodos de gerenciamento da memória armazenados na DMML (*Dynamic Memory Management Library*). A CAT é responsável por substituir cada bytecode dinâmico por uma chamada ao método apropriado que está armazenado na DMML.

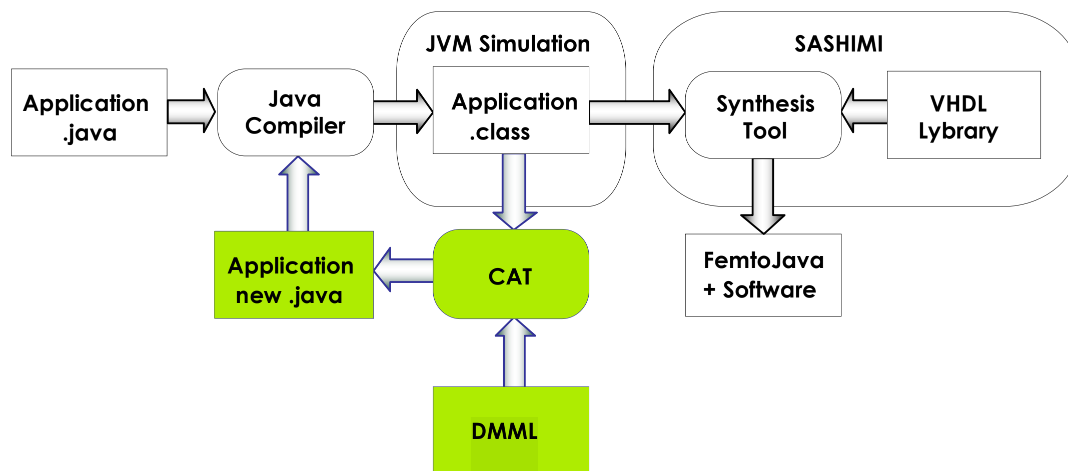

Figura 3.6: Fluxo de projeto SASHIMI modificado.

Assim os arquivos *class* modificados são então utilizados para simulação na máquina virtual tradicional que acessa as classes e os métodos obtidas da DMML que contém ainda algumas variáveis também utilizadas pelo Alocador e o Garbage Collector. O restante do processo de síntese realizado pelo ambiente SASHIMI permanece não modificado até a geração do *hardware* e do *software*.

A DMML é formada por dois pacotes cada um contendo duas classes: Dmms.*class* e DmmsStr.*class*. Cada um dos pacotes especifica um dos sistemas desenvolvidos. Desse modo, o pacote dmms1 contém as classes relativas à primeira implementação que se baseia na proposta de Lin e o pacote dmms2 contém as classes relativas à segunda implementação que se baseia na proposta de Ritzau.

Após modificados pela ferramenta de adaptação, todos os arquivos executáveis da aplicação que realizam manipulação dinâmica de memória contêm chamadas para a classe Dmms que armazena todos os métodos necessários para realização da gerência dinâmica de memória. A classe DmmsStr foi criada para dar suporte à manipulação de Strings, sendo carregada para a memória do processador apenas se a aplicação exigir suporte a Strings. A tab. 3.1 exibe um resumo de dados relativos a estas duas classes, conforme a implementação considerada.

Tabela 3.1: Resumo dos dados disponíveis na DMML

Implementação	Classe	Nº de métodos	Nº de linhas total
1	Dmms	35	888
2	Dmms	34	891
1	DmmsStr	12	396
2	DmmsStr	11	296

A classe DmmsStr foi projetada para suportar um subconjunto de 11 métodos da classe String Java. A implementação destes métodos foi realizada utilizando-se algumas primitivas do próprio sistema (métodos da classe Dmms), como forma de obter um melhor desempenho para o processamento de Strings. A tab. 3.2 lista e descreve os 11 métodos implementados na classe DmmsStr.

Tabela 3.2: 11 métodos implementados na classe DmmsStr.

Método	Descrição
Aloc	Um construtor de Strings.
charAt	Retorna um caractere de uma String de acordo com a posição especificada.
concat	Concatena duas Strings.
substring	Retorna uma substring de uma String com base em uma posição inicial e uma final.
equalsIgnoreCase	Compara duas Strings, ignorando diferenças entre caracteres maiúsculos e minúsculos.
replace	Substitui todas as ocorrências de um caractere por um outro.
lastIndexOf	Retorna a posição da última ocorrência de um caractere em uma String
indexOf	Retorna a posição da primeira ocorrência de um caractere em uma String
equals	Compara duas Strings, considerando diferenças entre caracteres maiúsculos e minúsculos.

Método	Descrição
compareTo	Compara duas Strings lexicograficamente, retornando: 0 se as Strings forem iguais < 0 se a primeira String for menor que a segunda > 0 se a primeira String for maior que a segunda
length	Retorna o comprimento de uma String

3.4.1 A Ferramenta de Adaptação de Código (CAT)

A CAT pode ser concebida como uma aplicação pertencente a um conjunto de ferramentas existentes para instrumentação de código, tal como [LEE 2004] [VAL 2005] [DAH 2005], assim como mostra a fig. 3.7. Estas ferramentas permitem que o código de qualquer aplicação possa ser modificado de forma a possibilitar que novas classes e métodos possam ser incorporados a esta aplicação de maneira automatizada.

Uma tentativa de utilização da ferramenta BIT (Bytecode Instrumenting Tool) [LEE 2004] foi realizada sendo, no entanto, mal sucedida. Esta ferramenta produz modificações exclusivamente sobre o formato class de uma aplicação. Logo, a revalidação de cada código modificado precisaria ser feita inserindo-se pontos de visualização em tela (*prints*) com o intuito de observar os diferentes valores de cada variável ao longo do fluxo de execução da aplicação. Neste sentido, principais problemas encontrados foram:

- O processo de revalidação da aplicação se torna excessivamente lento devido ao trabalho extra de inserção de pontos de observação.
- A análise do código modificado pela ferramenta torna-se difícil devido a quantidade normalmente grande de pontos de observação inseridos.

Outro problema que impediu o uso do BIT foi o fato de apenas um parâmetro poder ser utilizado para as chamadas aos métodos adicionais criados para aplicação. Com isto, uma chamada de método com *n* parâmetros precisaria ser decomposta em *n* chamadas de métodos, o que poderia introduzir um overhead em processamento significativo apenas para adaptar a aplicação.

Desse modo, sob o risco de reincidir sobre os mesmos problemas com as demais ferramentas, optou-se em construir uma nova ferramenta especifica para realizar a adaptação da aplicação, sendo assim criada a CAT.

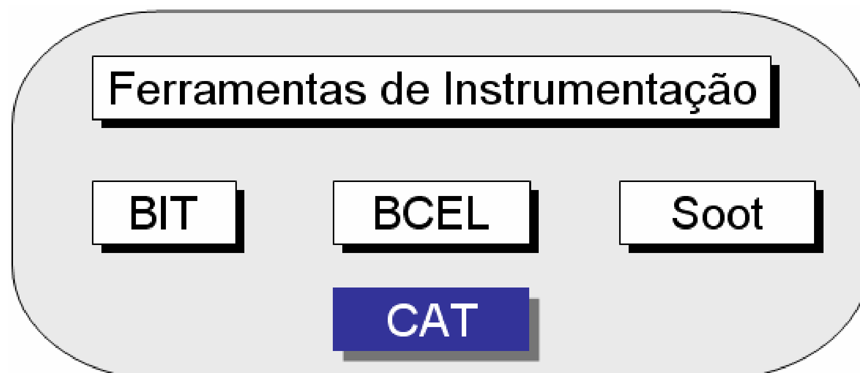

Figura 3.7: Ferramentas existentes para instrumentação de código

A arquitetura geral da CAT pode ser dividida em dois módulos como mostra a fig. 3.7. No primeiro módulo à esquerda, é feita sobre o código da aplicação a substituição das instruções não suportadas pelo processador por rotinas da

DMML. No segundo módulo, é gerada com base nos arquivos .*class* da aplicação, a classe MemDatApp que contém todas as *constant pools* da aplicação, além de informações sobre a estrutura de cada um dos objetos desta aplicação.

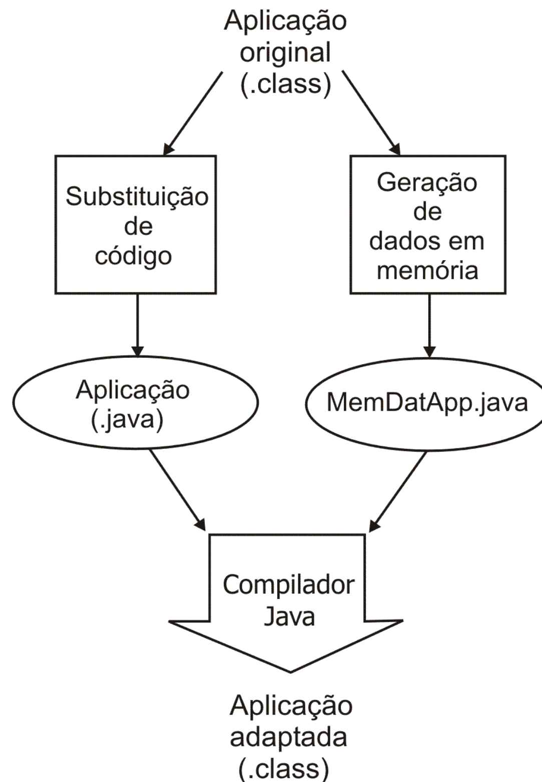

Figura 3.8: Arquitetura geral da CAT

3.4.1.1 Módulo de Substituição de Código

Conforme mostra a fig. 3.8, o estágio inicial de execução do módulo de substituição de código consiste em fazer com que os arquivos *class* sejam enviados para ferramenta DJ [DOW 2005] que entre suas funções é capaz de extrair do .*class* um código Java equivalente ao original da aplicação com uma extensão padrão para seus arquivos *.jad*. Este formato de arquivo pode ter sua extensão modificada manualmente (ou automaticamente através do utilitário de configuração da ferramenta) para .java e logo após compilados e executados sobre uma máquina virtual Java.

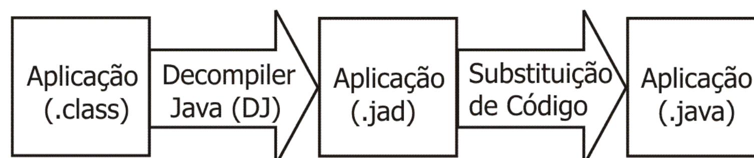

Figura 3.9: Arquitetura geral do módulo de substituição de código

Abaixo de cada linha de código Java contido em um arquivo .jad, são exibidos na forma de comentários os *bytecodes* que correspondem à linha Java, conforme mostrado na fig. 3.9 (a). Este formato de código é então enviado para o núcleo da adaptação de código, onde cada bytecode comentado é utilizado para

produzir uma adaptação sobre a linha Java (não comentada) imediatamente acima. Cada linha Java adaptada (fig. 3.9 (b)) é capaz de ser compilada produzindo apenas *bytecodes* pertencentes ao conjunto pré-existente de instruções da arquitetura alvo.

```
If (j <= sizeAg - 1)
//* 5 12: iload      5
//* 6 14: getstatic #2 <Field int sizeAg>
//* 7 17: iconst_1
//* 8 18: isub
//* 9 19: icmpgt    50
{
 Agend[j] = new AgendEl();
//* 10 22: getstatic            #4 <Field AgendEl[] agend>                    (a)
//* 11 25: iload    5
//* 12 27: new                  #3 <Class AgendEl>
//* 13 30: dup
//* 14 31: invokespecial  l#9 <Method void AgendEl()>
//* 15 34: aastore
 Agend[j].constrAgendEl(s, s1, i, s2);
//* 16 35: getstatic            #4 <Field AgendEl[] agend>
//* 17 38: iload    5
//* 18 40: aaload
//* 19 41: aload_0
//* 20 42: aload_1
//* 21 43: aload_2
//* 22 44: iload_3 5
//* 23 45: aload   4
//* 24 47: invokevirtual   #10 <>Method void AgendEl constrAgendEl(String, String, int , String)>
}

                                                                              (b)
If (j <= sizeAg - 1){
 Dmms.SetElemArraySimp(agend, j, 1, 0, Dmms.mgNew(3, 0, 83));
 AgendEl.constrAgendEl(10, Dmms.loadElemarraySimp(agend, j, 1), Dmms.mgAload_0(s), s1, s2, i, s3);
}
```

Figura 3.10: Formato de arquivo .jad e o formato da aplicação modificada produzida pela CAT

3.4.1.2 Módulo de Geração de Dados em Memória

Uma vez que a arquitetura do processador FemtoJava dispõe de memórias RAM e ROM para alocação de dados, a alternativa encontrada para armazenar os dados necessários à gerência dinâmica de memória foi a geração destes dados na forma de uma outra classe extra para a aplicação *MemDatApp*. Especificamente, esta classe contém, além de uma *constant pool* para cada classe da aplicação, uma estrutura para definição do tipo e posicionamento de cada campo (variável de instância) da classe. Esta estrutura é necessária porque o SGDM deve ser capaz de determinar o número de campos em cada objeto e tamanho de cada campo, o correto posicionamento de cada campo (para acesso ao campo) e se cada campo contém ou não uma referência a outro objeto [APP 97].

O mecanismo para acesso à *constant pool* e à estrutura de campos de cada classe segue o formato para acesso a dados proposto para JVM. A JVM determina a *constant pool* a ser acessada a partir do nome da classe correspondente. Este nome em geral é obtido também de uma *constant pool* da aplicação, já que muitas informações contidas nesta tabela são disponíveis em formato texto (veja utf8 [YEL 99]). Outras informações e formato texto também obtidas através do acesso à *constant pool* são nomes de métodos e campos de objetos. Com isso, o mecanismo utilizado pela JVM para leitura de informações

contidas nas classes da aplicação é baseado em diversas comparações de Strings.

A diferença entre forma de acesso às informações dinâmicas utilizada por este trabalho e a forma utilizada pela JVM consiste na vinculação de um código numérico a cada classe da aplicação, que tem por objetivo aumentar o desempenho e reduzir o número de comparações de Strings realizadas. A fig. 3.10 exibe como ocorre a vinculação de código a cada classe.

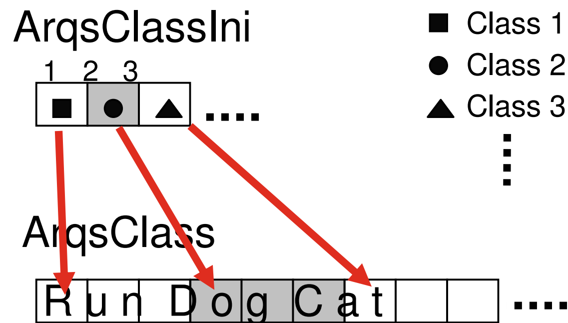

Figura 3.11: Vinculação de códigos para as classes da aplicação

Na fig. 3.10, cada índice que identifica uma posição do vetor *ArqsClassIni* corresponde a um código para uma classe da aplicação. O valor contido na posição indicada por um índice informa a posição em *ArqsClass* onde se inicia o nome da classe correspondente a este índice. Para identificar o código de uma determinada classe o SGDM deve percorrer cada uma das posições de *ArqsClassIni* e realizar uma comparação entre a String apontada por esta posição e a String com o nome da classe desejada.

Após obtenção do código numérico correspondente a uma determinada classe, a consulta à *constant pool* desta classe consiste inicialmente em utilizar o código da classe para indexar uma das posições o vetor *CpsIndexIni*, conforme mostra a fig. 3.11. O valor da posição indexada pelo código informa o valor de outro índice (*i*) que será utilizado para acessar uma das posições de *CpsIndex*. Cada posição de *CpsIndex* contém o endereço de uma das entradas da *constant pool* desejada, que está armazenada junto as demais *constant pools* da aplicação dentro do vetor *Cps*. É importante observar que o vetor *CpsIndex* contém os endereços das entradas de cada *constant pool* da aplicação.

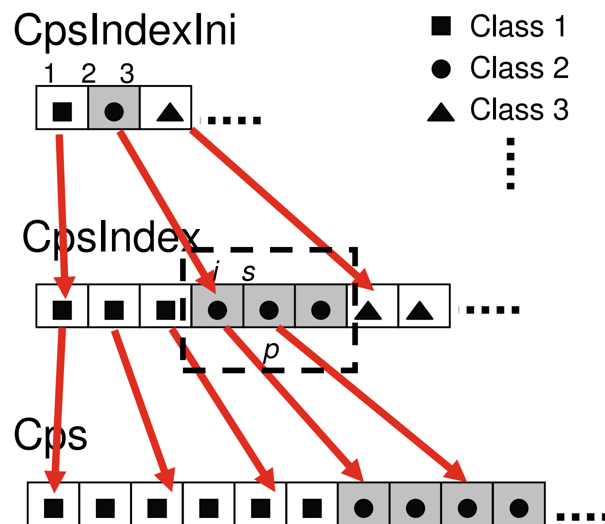

Figura 3.12: Acesso a *Constant pool* de uma classe qualquer

Sendo assim, o último índice calculado (*i*) corresponde ao endereço inicial da parcela (*p*) em *CpsIndex* correspondente a *constant pool* pretendida. Neste momento, o número da entrada (*e*) da *constant pool*, ao qual deseja-se fazer acesso deve ser utilizado como um deslocamento a ser somado sobre o endereço inicial *(i*) levantado. Em seguida, o valor resultante desta soma (*s*) indicará a posição para acesso no vetor *CpsIndex*. Por fim, o valor contido na posição acessada de *CpsIndex* corresponderá à posição (*o*) em *Cps* onde se inicia a entrada desejada da *constant pool* alvo.

Este processo precisou ser realizado através de uma hierarquia de ponteiros, uma vez que as entradas de cada constant pool não possuem um tamanho fixo. Ao todo, cada entrada de uma constant pool Java pode assumir 11 diferentes tipos [YEL 99].

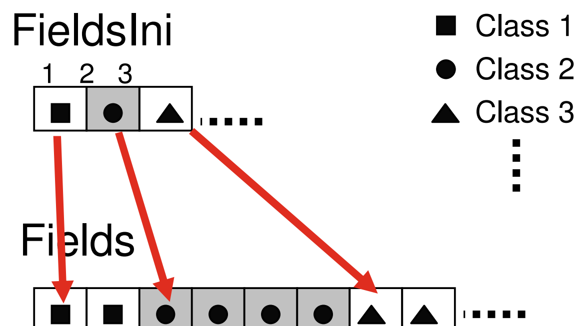

Figura 3.13: Acesso à estrutura de definição dos campos de classe

O acesso à estrutura de campos de cada classe é feito de forma similar ao acesso à constant pool, possuindo entretanto um nível a menos em sua hierarquia, uma vez que todos os itens da estrutura de campos Java, utilizados neste trabalho possuem um tamanho fixo. Na fig. 3.12, o acesso à estrutura de campos de uma classe também inicia com uma indexação a partir do código da classe. Este código informa a posição do vetor *FieldsIni* correspondente à classe a qual o campo pertence. O valor contido nesta posição informa o índice para acesso ao endereço inicial da parcela (*p*) em *Fields* correspondente à classe pretendida. Neste momento o acesso a informações sobre qualquer campo da classe é feito através

de deslocamentos cujos tamanhos são múltiplos do tamanho usado para armazenar cada informação sobre os campos.

4 RESULTADOS

Este capítulo apresenta os principais resultados obtidos através deste trabalho. As estimativas mais expressivas são com relação ao uso de memória, desempenho, energia consumida e potência média para as duas implementações desenvolvidas, além dos impactos percentuais sobre o conjunto de instruções da arquitetura alvo.

Para possibilitar grande parte destas estimativas, um conjunto de aplicações tipicamente embarcado foi utilizado, sendo formado por um livro de endereços (*Address Book*), um *Mp3 Player* [TYS 2005] e um jogo mundialmente conhecido (*Sokoban Game*) [SOK 2005]. As principais características gerais relacionadas a estas aplicações são:

- *Address Book*
    - Aplicação de pequeno porte (com 244 linhas de código)
    - Intensa manipulação de Strings
    - Armazena até cinco informações por registro
    - Todas as informações em formato texto possuem um tamanho fixo de 13 caracteres
- *Mp3 Player*
    - Aplicação de grande porte (com 7907 linhas de código)
    - Aplicação com características voltadas ao fluxo de dados
    - Intensa realização de operações aritméticas
    - Intensa manipulação de *arrays* simples, bi-dimensionais e tri-dimensionais
- *Sokoban Game*
    - Aplicação de pequeno porte (com 865 linhas de código)
    - Intensa manipulação de *arrays* simples, bi-dimensionais
    - Utilização de um único cenário para todo o jogo
    - A manipulação dinâmica da memória está vinculada à mudança de estado de pequenos elementos do cenário durante a execução da aplicação

Outras informações mais específicas sobre estas aplicações estão apresentadas na tab. 4.1. Algumas destas informações foram obtidas em tempo de execução, através da inserção de pontos de observação no código da aplicação já devidamente configurado para execução sobre a plataforma FemtoJava.

Tabela 4.1: Estimativas para o conjunto de aplicações usadas neste trabalho

Estimativa	*Address Book*	*Mp3 Player*	*Sokoban*
Número total de objetos alocados	48	20605	143
Número total de objetos desalocados	32	17657	138
Tamanho médio de objeto (bytes)	52	125	20
Tamanho máximo de objeto (bytes)	124	131072	400
Número máximo de objetos presentes na heap em tempo de execução	20	3281	48
Número máximo de chamadas de métodos aninhadas	6	10	8
Número máximo de chamadas de métodos de instância aninhadas	2	8	7
Número de arquivos *class*	2	25	7
Número de classes que podem ser instanciadas	1	10	6
Memória para armazenamento de todas as *constant pools* (bytes)	9176	205908	20060

Para as estimativas correspondentes ao Mp3 player, foi realizada uma pequena modificação sobre esta aplicação para que fosse possível coletar informações sobre sua execução. Como esta aplicação realiza uma execução baseada na decodificação de frames de dados, a modificação consistiu em fazer a aplicação decodificar apenas o número de frames desejado. Isto foi necessário, porque a decodificação de um conjunto de frames completo tornaria inviável o levantamento de informações sobre a aplicação.

4.1 Estimativas Quanto ao Uso da Memória

Esta seção apresenta o comportamento quanto ao uso de memória para o conjunto de aplicações utilizado neste trabalho. Os dados apresentados foram obtidos através de uma ferramenta de perfilamento construída especificamente para este fim. Basicamente, esta ferramenta consiste em uma biblioteca de rotinas para geração de informações, as quais são acionadas a partir dos principais pontos de manipulação da memória contidos em cada implementação. O acionamento a partir destes pontos é feito através de chamadas para as rotinas da biblioteca, as quais são inseridas com o uso da ferramenta BIT [LEE 2004].

Desse modo, a cada operação de alocação ou desalocação de um objeto, o estado de ocupação da memória é atualizado em uma tabela mantida pela ferramenta de perfilamento. Entretanto, para que a geração de gráficos de ocupação da memória pudesse ser mais simples, utilizou-se também uma taxa de amostragem baseada no número máximo de instruções executadas, obtido entre as duas implementações desenvolvidas. Com isto, após a execução de um determinado número de instruções o estado corrente de ocupação da memória é recuperado da tabela armazenada pela ferramenta de perfilamento e estes dois parâmetros são agrupados para formar um dos pontos dos gráficos de ocupação apresentados.

Um detalhamento dos dados envolvidos para o cálculo da taxa de amostragem utilizada para cada aplicação está apresentado na tab. 4.2. A variação entre as taxas utilizadas para cada aplicação ocorre em função do arredondamento aplicado ao intervalo de amostragem utilizado em cada caso.

Tabela 4.2: detalhamento dos dados envolvidos para o cálculo da taxa de amostragem utilizada para cada aplicação

Aplicação	Número máximo de instruções executadas	Intervalo de instruções para amostragem	Taxa de amostragem
Address Book	351412	2300	0,7%
Mp3 Player	1082087354	4800×10^3	0,4%
Sokoban	36913709	290×10^3	0,8%

Com relação ao levantamento de dados para a segunda implementação desenvolvida, optou-se em utilizar três diferentes tamanhos de bloco como forma de expor a variação de performance ocorrida em conseqüência da mudança do tamanho de bloco utilizado. Logo, por simplicidade os três tamanhos de bloco utilizados foram extraídos de uma seqüência de valores obtidos com base em uma função de potência de base dois, resultando assim em tamanhos iguais a 16, 32 e 64 bytes.

4.1.1 Address Book

A fig. 4.1 apresenta a variação do número de objetos presentes na memória durante toda a execução da aplicação. Neste caso, uma vez que se pretendia apenas demonstrar a variabilidade dos objetos alocados, decidiu-se substituir o número de instruções executadas (eixo horizontal) pelo número de pontos de amostragem utilizados. Para isto, a amostragem foi realizada sobre os pontos de manipulação da memória e não sobre o número de instruções executadas. Desse modo foi possível obter um gráfico independente da implementação utilizada, ocorrendo entretanto a perda da relação temporal entre cada ponto amostrado.

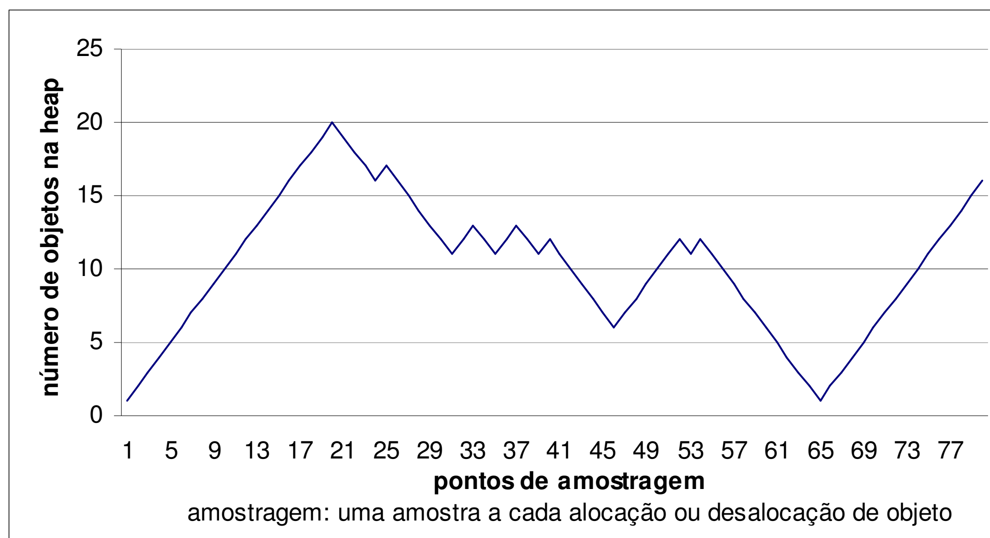

Figura 4.1: *Address Book*: Número de objetos na memória durante a execução

Na fig 4.1 a desalocação acentuada verificada no gráfico é resultado da rotina *EraseAllNames* disponível entre funções realizadas pela aplicação.

A tab. 4.3 apresenta um resumo do desempenho em memória obtido pela primeira implementação. O motivo da diferença entre os valores para o tamanho médio e máximo de objeto apresentados na tab. 4.3, em relação aos apresentados

na tab. 4.1, é que no primeiro caso são apresentados os custos totais em memória para armazenamento de cada objeto, enquanto que no segundo os dados apresentados correspondem apenas aos bytes necessários para armazenamento dos campos do objeto. Os itens 3 e 4 apresentam informações sobre o volume total de heap movimentado pelo sistema. Os itens 5 e 6 apresentam informações sobre o volume total de memória movimentado. Os itens 7 e 8 exibem informações que permitem definir, respectivamente, os requisitos mínimos de memória para armazenamento da heap e da *OST*. Os itens 9 e 10 apresentam informações referentes à *OST* semelhantes às apresentadas nos itens 3 e 4 para a heap. No item 11 é apresentada a relação percentual entre os requisitos mínimos em memória para armazenamento de todas as estruturas de dados do sistema (*overhead* estático do sistema) e os requisitos mínimos em memória para armazenamento da heap.

Tabela 4.3: *Address Book*: Desempenho da primeira implementação quanto ao uso de memória

	Estimativa	**Valor**
1	Tamanho médio de objeto (heap+*OST*+*HP*+*UL*) (bytes)	79
2	Tamanho máximo de objeto (heap+*OST*+*HP*+*UL*) (bytes)	157
3	Total de heap alocada (bytes)	2816
4	Total de heap desalocada (bytes)	2004
5	Total de memória alocada (heap+*OST*+*HP*+*UL*) (bytes)	3760
6	Total de memória desalocada (heap+*OST*+*HP*+*UL*) (bytes)	2642
7	Máxima alocação de memória para heap (bytes)	1052
8	Máxima alocação de memória para a *OST* (bytes)	320
9	Total de memória alocada para *OST* (bytes)	768
10	Total de memória desalocada para *OST* (bytes)	512
11	*OST*+*UL*+*HP* em relação à heap (%)	36

A fig. 4.2 mostra o comportamento quanto ao consumo de memória para cada componente da primeira implementação. Na fig. 4.2 é possível perceber que a heap e a *OST* são os componentes que mais consomem memória e também os que possuem a maior variação de consumo na primeira implementação. A curva de consumo correspondente aos vetores *UL* e *HP* demonstra que estes componentes desempenham um consumo relativamente baixo e aproximadamente constante se comparado respectivamente aos consumos de memória e a variação obtida para os demais componentes desta implementação. Em geral, para cada ponto de amostragem que compõe o gráfico apresentado na fig. 4.2, o consumo produzido conjuntamente pelos vetores *UL* e *HP* equivale à 6% do consumo produzido pela heap. De um modo geral, é possível dizer que o consumo produzido por estes dois vetores é praticamente constante, influindo muito pouco sobre os requisitos de memória obtidos através desta implementação.

Ainda com relação à fig. 4.2, é possível também perceber novamente a desalocação acentuada no gráfico proporcionada pela rotina *EraseAllNames* existente na aplicação.

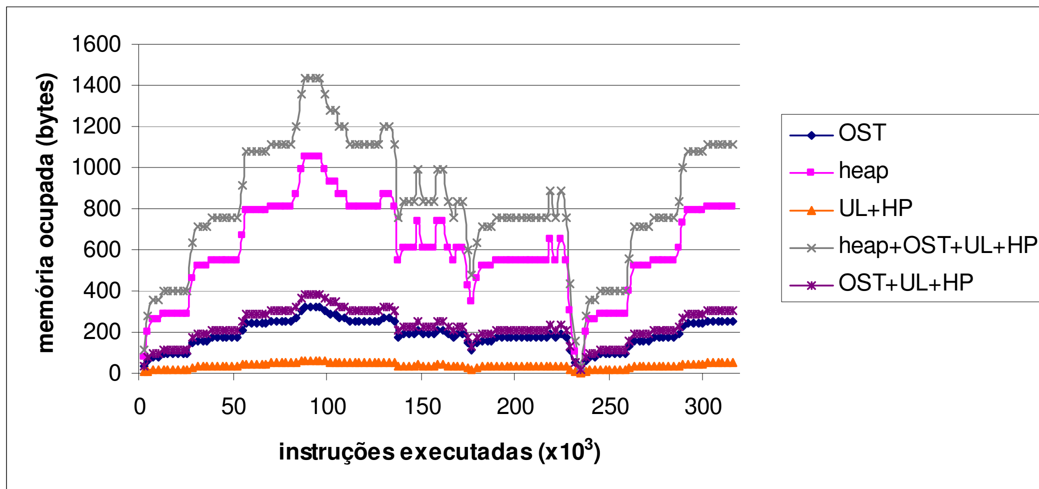

Figura 4.2: *Address Book*: Utilização de memória através da primeira implementação

A tab. 4.4 apresenta um resumo do desempenho em memória obtido pela segunda implementação. De um modo geral, os melhores resultados são favoráveis à utilização de um tamanho de bloco de 64 bytes. Este resultado é esperado, uma vez que a aplicação manipula exclusivamente Strings de 52 bytes.

Para sistemas de gerência dinâmica de memória que utilizam uma estrutura de heap organizada em blocos, um tamanho de bloco maior normalmente possibilita um melhor desempenho quando as aplicações usadas manipulam Strings. Entretanto, nem sempre um tamanho de bloco maior produz um melhor rendimento da memória quando a média de tamanho dos objetos alocados é elevada. A exemplo disto, considerando a tab. 4.4, para um tamanho de bloco igual a 16 bytes (menor tamanho utilizado) os custos em memória são bastante elevados, mas são no entanto menores do que os custos obtidos quando o tamanho do bloco é dobrado. Isto porque um tamanho de bloco menor neste caso permite uma alocação de memória mais precisa por parte do sistema de gerência. Dessa forma, isto mostra que é conveniente sempre fazer uma avaliação caso a caso, já que além do problema explicitado, muitas aplicações podem ainda alocar Strings de tamanhos bastante diferenciados [DYK 2002].

Tabela 4.4: *Address Book*: Desempenho da segunda implementação quanto ao uso de memória

	Estimativa	**Tamanho do bloco (bytes)**		
		16	**32**	**64**
1	Tamanho médio de objeto (bytes)	79	90	72
2	Tamanho máximo de objeto (bytes)	176	160	192
3	Total de memória alocada (heap) (bytes)	3776	4288	3456
4	Total de memória desalocada (heap) (bytes)	2688	3008	2432
5	Máxima alocação de memória para heap (bytes)	1408	1664	1280
6	Bytes gastos para encadeamento dos blocos	352	208	80
7	Custos para encadeamento dos blocos em relação ao tamanho total da heap (%)	25	12,5	6,25

A fig. 4.3 mostra o comportamento quanto ao consumo de memória para a segunda implementação. Um detalhe importante a ser observado no gráfico exibido é a inexistência da curva acentuada normalmente ocasionada pela grande desalocação produzida pela rotina *EraseAllNames* da aplicação. Isto ocorre porque que o algoritmo de desalocação utiliza a estratégia de *Lazy Freeing* (Liberação Preguiçosa) [WEI 1963], para atendimento a requisitos de tempo real. Com isto, apenas a memória utilizada pelo objeto desalocado diretamente pela aplicação é liberada. O restante da memória a ser liberada, correspondente aos descendentes deste objeto, só é liberada quando a memória ocupada pelo objeto é reciclada para uso da aplicação. Assim, de forma condizente com a estratégia utilizada, a curva gerada por este algoritmo mostra que apenas a quantidade de memória realmente necessária é produzida pelo *garbage collector*.

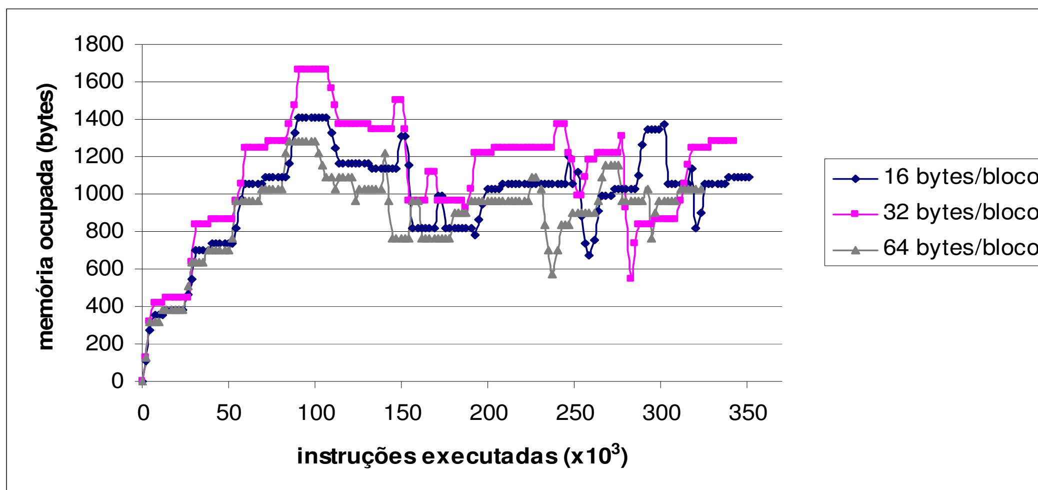

Figura 4.3: *Address Book*: Utilização de memória através da segunda implementação

A fig. 4.4 apresenta uma comparação entre os consumos de memória obtidos com a primeira e a segunda implementação. O tamanho fixo de 52 bytes para alocação de todas as Strings da aplicação, e a possibilidade de configuração do tamanho de bloco utilizado possibilitam que a segunda implementação desempenhe uma melhor utilização de memória. Entretanto, é importante novamente salientar que o uso de uma aplicação com alta variabilidade do tamanho dos objetos alocados pode produzir um resultado favorável à primeira implementação. Isto porque esta implementação possui um *overhead* em memória aproximadamente constante[3] para alocação de qualquer objeto. Se a aplicação alocar bem distribuidamente objetos de vários tamanhos, mesmo a melhor configuração do tamanho do bloco para a segunda implementação poderá resultar em um *overhead* superior ao obtido com a primeira implementação, pelos mesmos motivos já explicados anteriormente.

---

[3] Apenas uma parcela muito pequena deste overhead é variável. Esta parcela corresponde ao consumo de alguns poucos bits para armazenamento das informações relativas aos vetores *UL* e *HP*.

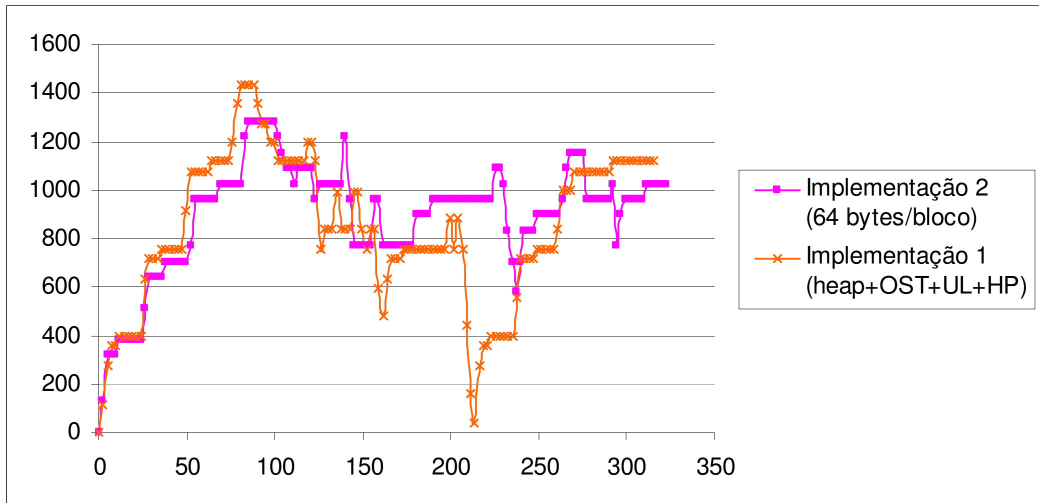

Figura 4.4: *Address Book*: Utilização de memória:
implementação 1 x implementação 2

4.1.2 Mp3 Player

O levantamento de estimativas quanto à utilização de memória realizada por esta aplicação foi desenvolvido considerando a decodificação de 5 frames de um arquivo em formato mp3.

A fig. 4.5 apresenta a variação do número de objetos presentes na memória durante toda a execução da aplicação. O fato de um grande número de objetos permanecer na memória durante a execução ocorre porque, de acordo com as características de projeto da aplicação, estes objetos são necessários durante a maior parte da execução, não sendo possível desalocá-los.

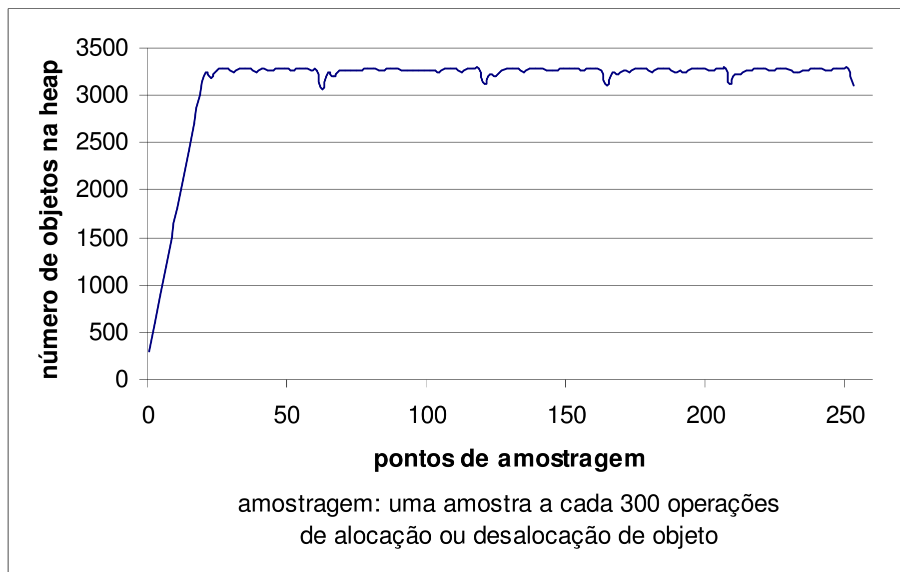

Figura 4.5: *MP3 Player*: Número de objetos na memória durante a execução

Entretanto, o gerenciamento dinâmico da memória possibilita uma economia significativa de memória, correspondente ao grande número de alocações e desalocações que ainda ocorrem mesmo após a alocação dos objetos cuja permanência na memória é constante. A tab. 4.5 apresenta o resumo do desempenho em memória obtido pela primeira implementação

Tabela 4.5: *MP3 Player*: Desempenho da primeira implementação quanto ao uso de memória

	Estimativa	**Valor**
1	Tamanho médio de objeto (heap+*OST*+*HP*+*UL*) (bytes)	157
2	Tamanho máximo de objeto (heap+*OST*+*HP*+*UL*) (bytes)	139289
3	Total de heap alocada (bytes)	2724624
4	Total de heap desalocada (bytes)	2595028
5	Total de memória alocada (heap+*OST*+*HP*+*UL*) (bytes)	3224593
6	Total de memória desalocada (heap+*OST*+*HP*+*UL*) (bytes)	3039730
7	Máxima alocação de memória para heap (bytes)	363728
8	Máxima alocação de memória para a *OST* (bytes)	52496
9	Total de memória alocada para *OST* (bytes)	329680
10	Total de memória desalocada para *OST* (bytes)	282512
11	*OST*+*UL*+*HP* em relação à heap (%)	20

A fig. 4.6 mostra o comportamento quanto ao consumo de memória para cada componente da primeira implementação ao longo da execução, onde o pico máximo de ocupação total de memória encontrado é de 430966 bytes. Algumas informações importantes derivadas deste valor relacionadas a alguns valores apresentados na tab. 4.5 são:

- Somente o maior objeto alocado durante a execução corresponde a aproximadamente 32% da quantidade mínima total de memória para execução através desta implementação.
- O volume mínimo de memória necessário para execução com esta implementação corresponde aproximadamente a apenas 12,9% do somatório total de memória alocada, o que demonstra claramente que, embora muitos objetos possam permanecer constantemente na memória, o gerenciamento dinâmico possibilita de fato um impacto positivo sobre os recursos necessários em memória.

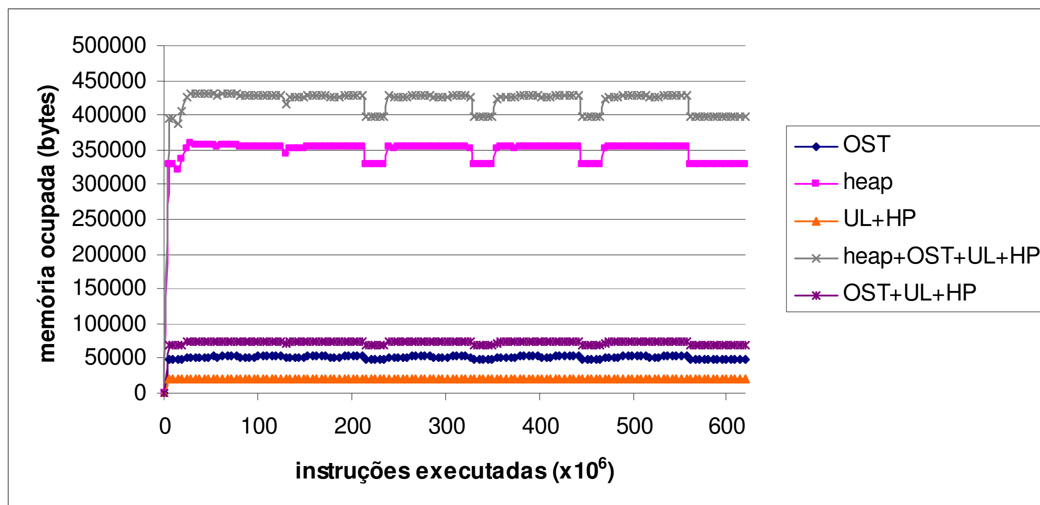

Figura 4.6: *MP3 Player*: Utilização de memória através da primeira implementação

Ainda com base nos dados apresentados na fig. 4.6 e na tab. 4.5, outra importante constatação é de que o overhead estático mantido pela primeira implementação (correspondente à curva *OST* + *UL* + *HP*) não ultrapassa 20% (item 11 tab. 4.5). Este percentual é significativamente menor que o obtido para o Address Book, sendo esta diminuição atribuída principalmente ao aumento do tamanho médio dos objetos alocados, causando um aumento do tamanho mínimo necessário para heap no caso do Mp3 Player. Logo, a proporção entre o tamanho de heap necessário e o número de entradas requerido para a *OST*, durante o pico máximo de ocupação da memória, é maior no caso do Mp3 Player.

A tab. 4.6 apresenta um resumo do desempenho em memória obtido pela segunda implementação para os três tamanhos de bloco adotados para medição neste trabalho. De um modo geral, os melhores resultados são favoráveis à utilização de um tamanho de bloco de 32 bytes para esta aplicação e isto se deve aos seguintes fatores:

- Considerando que a média de tamanho dos objetos alocados pela aplicação é de 125 bytes (tab. 4.1), o uso de blocos de 16 bytes desperdiça um maior volume de memória para encadeamento dos blocos.
- O uso de blocos de 64 bytes produz desperdício de memória de 22,9% por objeto devido à fragmentação interna .
- Para alguns objetos menores que 32 bytes o tamanho de bloco intermediário causa um menor desperdício de memória.

Tabela 4.6: *MP3 Player*: Desempenho da segunda implementação quanto ao uso de memória

	Estimativa	Tamanho do bloco (bytes)		
		16	32	64
1	Tamanho médio de objeto (bytes)	180	165	187
2	Tamanho máximo de objeto (bytes)	174784	149824	139840
3	Total de memória alocada (heap) (bytes)	3698320	3383200	3841024
4	Total de memória desalocada (heap) (bytes)	3199216	2939520	3318784
5	Máxima alocação de memória para heap (bytes)	517472	458656	540544
6	Bytes gastos para encadeamento dos blocos	129368	57332	33784
7	Custos para encadeamento dos blocos em relação ao tamanho total da heap (%)	25	12,5	6,25

A fig. 4.7 ilustra o comportamento quanto ao consumo de memória para o segundo sistema. Através da observação dos dados apresentados na fig. 4.7 como única medida de desempenho para a aplicação é possível perceber que, tal como ocorreu com o Address Book, há uma aparente ausência de relação lógica entre o tamanho de bloco utilizado e o desempenho alcançado pela segunda implementação. Após a obtenção do desempenho utilizando blocos de 16 bytes, o tamanho do bloco foi estendido para 32 bytes, obtendo-se com isso, uma melhora do desempenho para a aplicação. Entretanto, estendendo-se novamente o tamanho de bloco utilizado para 64 bytes, o desempenho torna-se pior que o desempenho inicial utilizando-se blocos de 16 bytes. Isto demonstra que a escolha do tamanho de bloco a ser utilizado requer um conhecimento mais profundo da aplicação, especialmente no que diz respeito ao tipo de objeto alocado. Este conhecimento auxilia a identificação e minimização dos desperdícios provocados pelo uso de um tamanho de bloco inadequado.

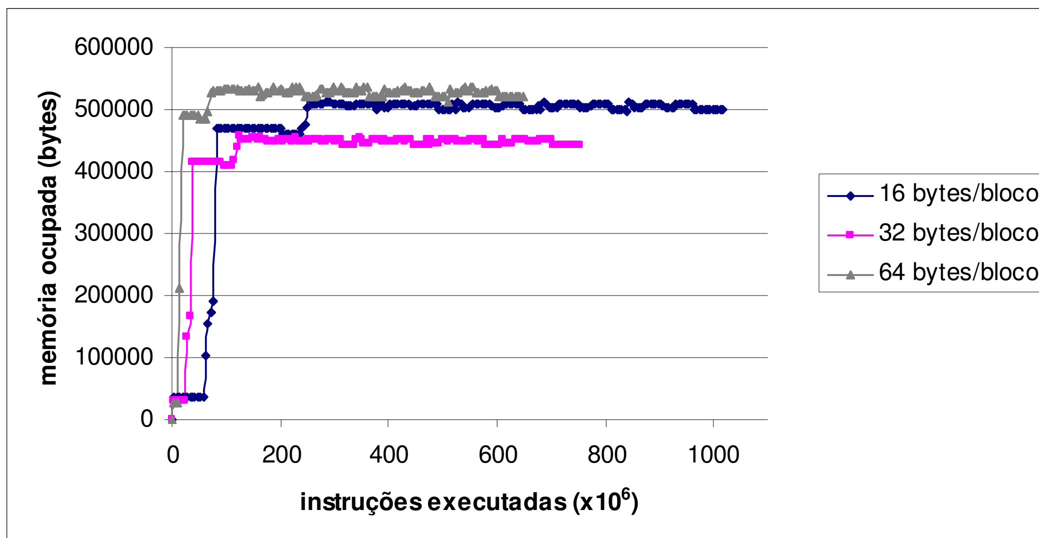

Figura 4.7: *MP3 Player*: Utilização de memória através da segunda implementação

A fig. 4.8 apresenta uma comparação entre os consumos de memória obtidos com a primeira e a segunda implementação, onde é possível perceber que

a curva de consumo gerada pela primeira implementação em geral é menor do que o consumo gerado pela segunda implementação.

Uma primeira justificativa para isto pode ser realizada com base na comparação dos custos para armazenamento de um objeto de tamanho médio em cada um dos sistemas. Na primeira implementação são necessários 157 bytes para armazenamento deste objeto, enquanto que na segunda são necessários no mínimo 165 bytes para armazenamento do mesmo objeto.

Outro fator que justifica a vantagem da primeira implementação em relação à segunda está no custo para armazenamento de um objeto de tamanho máximo. Isto porque através da análise da execução da aplicação, observou-se que este objeto permanece constantemente na memória. Logo, considerando o custo necessário em cada sistema para armazenamento deste objeto, concluiu-se que a diferença resultante contribui com aproximadamente 25,6% para diferença entre as curvas de consumo geradas.

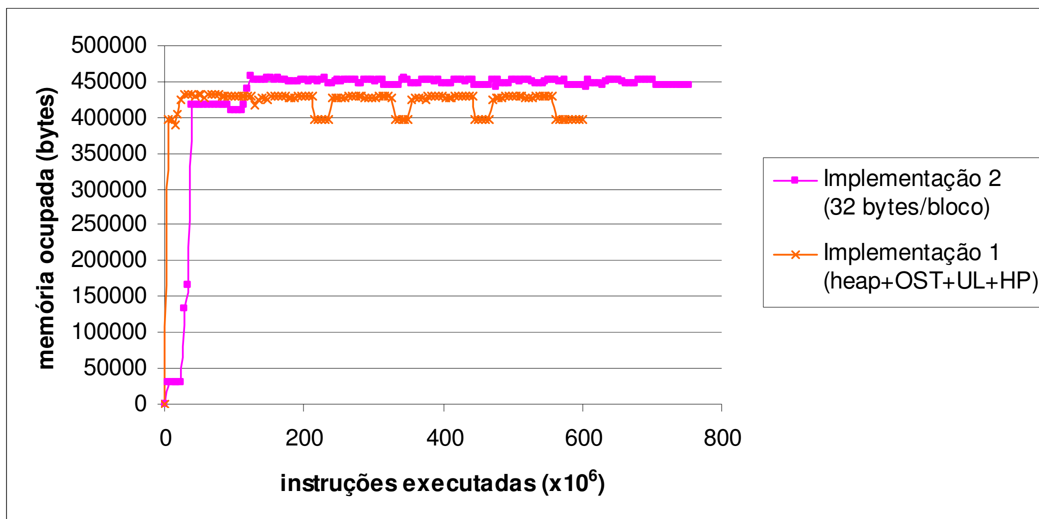

Figura 4.8: *MP3 Player*: Utilização de memória: implementação 1 x implementação 2

4.1.3 Sokoban Game

Na fig. 4.9 é apresentado o número de objetos presentes na memória ao longo da execução da aplicação. Tal como ocorrido para o *Mp3 Player*, esta aplicação também possui muitos objetos alocados permanentemente na memória. Estes objetos correspondem em grande maioria à estrutura do cenário da aplicação.

Após criados os objetos correspondentes a este cenário, a aplicação começa sua interação com o usuário, iniciando uma intensa seqüência de sucessivas alocações e desalocações na memória. Estas operações correspondem à movimentação produzida com base nos dados de entrada informados pelo usuário. Cada posição dentro do cenário é mantida por um objeto da aplicação, logo cada mudança de posição implica na desalocação do objeto relativo à antiga posição com imediata alocação de um objeto com informações correspondentes à nova posição.

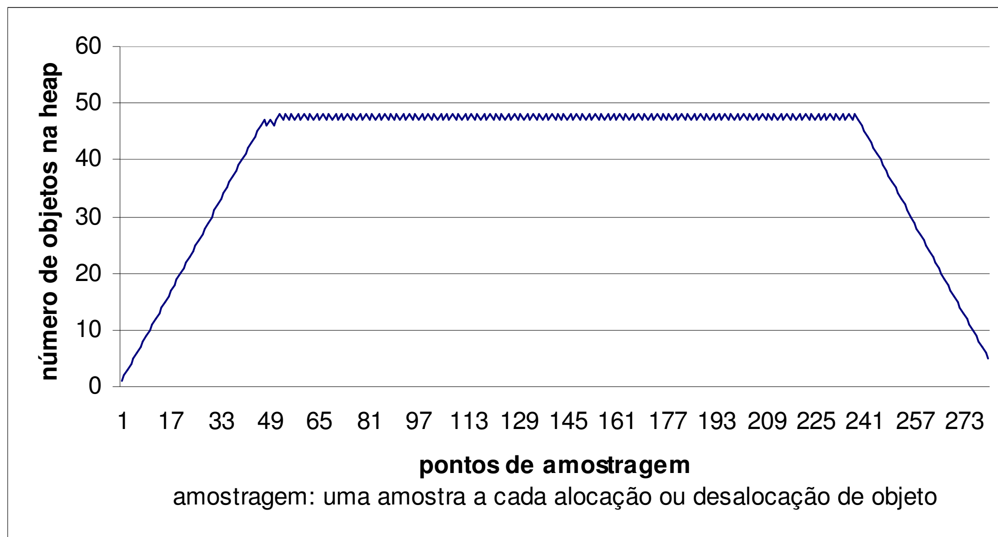

Figura 4.9: *Sokoban Game*: Número de objetos na memória durante a execução

A tab. 4.7 apresenta as estimativas quanto ao uso de memória obtido através desta aplicação rodando sobre a primeira implementação. Com a diminuição do tamanho médio dos objetos alocados, o overhead estático (item 11) novamente aumenta e atinge o maior percentual dentre as aplicações utilizadas neste trabalho.

Tabela 4.7: *Sokoban Game*: Desempenho da primeira implementação quanto ao uso de memória

	Estimativa	**Valor**
1	Tamanho médio de objeto (heap+*OST*+*HP*+*UL*) (bytes)	46
2	Tamanho máximo de objeto (heap+*OST*+*HP*+*UL*) (bytes)	450
3	Total de heap alocada (bytes)	3968
4	Total de heap desalocada (bytes)	3752
5	Total de memória alocada (heap+*OST*+*HP*+*UL*) (bytes)	6504
6	Total de memória desalocada (heap+*OST*+*HP*+*UL*) (bytes)	6195
7	Máxima alocação de memória para heap (bytes)	2448
8	Máxima alocação de memória para a *OST* (bytes)	768
9	Total de memória alocada para *OST* (bytes)	2288
10	Total de memória desalocada para *OST* (bytes)	2208
11	*OST*+*UL*+*HP* em relação à heap (%)	37

A fig. 4.10 mostra o comportamento quanto ao consumo de memória para cada componente do primeiro sistema. Na fig. 4.10 é possível observar que ao longo de toda a interação com o usuário a aplicação exerce um baixo impacto sobre a utilização da memória. Isto ocorre porque o objeto que armazena as informações sobre o posicionamento dentro do cenário da aplicação possui um tamanho fixo e relativamente pequeno. Logo, como todas as alocações e desalocações são para este tipo de objeto, a curva de utilização de memória se assemelha a uma linha de estabilização do processo de gerência.

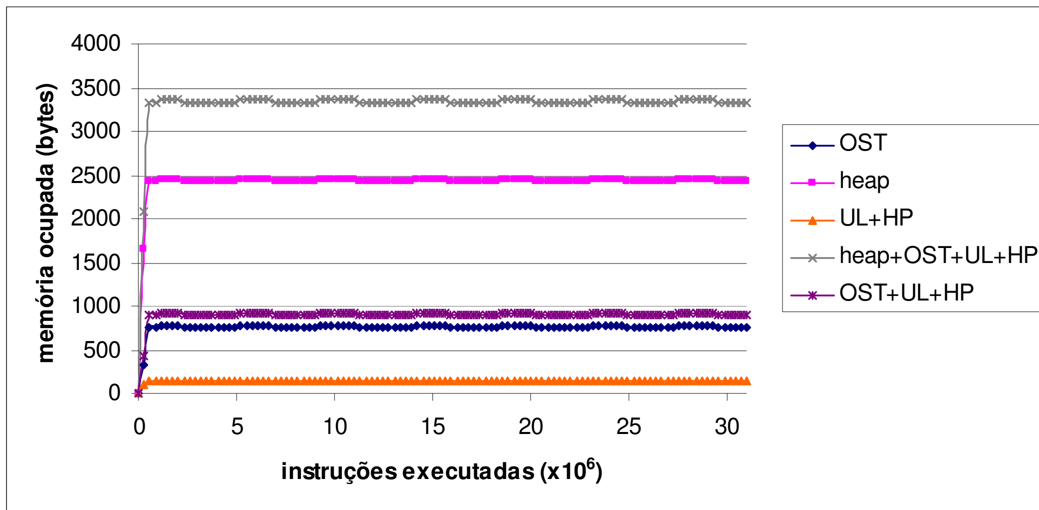

Figura 4.10: *Sokoban Game*: Utilização de memória através da primeira implementação

A tab. 4.8 apresenta um resumo do desempenho em memória obtido pela segunda implementação. Embora a média de tamanho dos objetos alocados seja baixa, alguns poucos objetos com tamanho significativamente maior que a média torna o desempenho do sistema para 32 bytes/bloco melhor.

Tabela 4.8: *Sokoban Game*: Desempenho da segunda implementação quanto ao uso de memória

	Estimativa	**Tamanho do bloco (bytes)**		
		16	**32**	**64**
1	Tamanho médio de objeto (bytes)	46	46	70
2	Tamanho máximo de objeto (bytes)	544	480	448
3	Total de memória alocada (heap) (bytes)	6480	6464	9984
4	Total de memória desalocada (heap) (bytes)	3152	3168	6272
5	Máxima alocação de memória para heap (bytes)	3440	3424	3904
6	Bytes gastos para encadeamento dos blocos	860	428	244
7	Custos para encadeamento dos blocos em relação ao tamanho total da heap (%)	25	12,5	6,25

A fig. 4.11 ilustra o comportamento quanto ao consumo de memória para o segundo sistema, onde é possível perceber uma considerável semelhança entre os desempenhos obtidos com o uso de 16 e 32 bytes/bloco. Isto ocorre porque o os objetos alocados possuem tamanho muito próximos a estes dois tamanhos de bloco. Isto se reflete no tamanho médio obtido para cada configuração de tamanho de bloco utilizado, onde os valores obtidos são idênticos desprezando-se os dígitos decimais[4].

[4] Os dígitos decimais são resultantes do cálculo da média dos tamanhos dos objetos alocados. Os tamanhos envolvidos na média são múltiplos do tamanho de bloco utilizado em cada configuração escolhida para o uso com a segunda implementação.

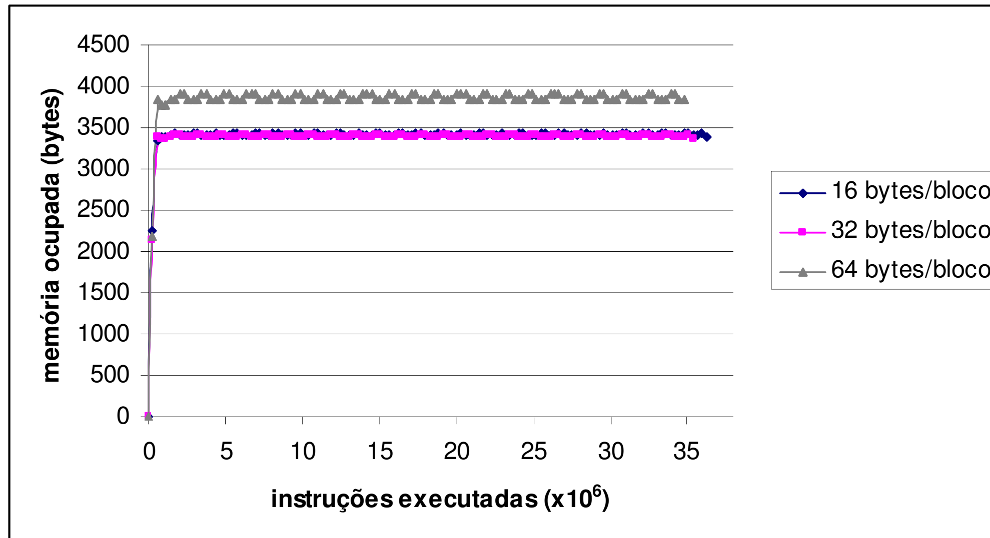

Figura 4.11: *Sokoban Game*: Utilização de memória através da segunda implementação

A fig. 4.12 apresenta uma comparação entre os consumos de memória obtidos com a primeira e a segunda implementação. Na fig. 4.12, o consumo obtido com a primeira implementação é bastante semelhante ao consumo obtido com a segunda implementação utilizando blocos de tamanho igual a 32 bytes. Acredita-se que alguns objetos permanentes de tamanho intermediário tenham causado o favorecimento à primeira implementação.

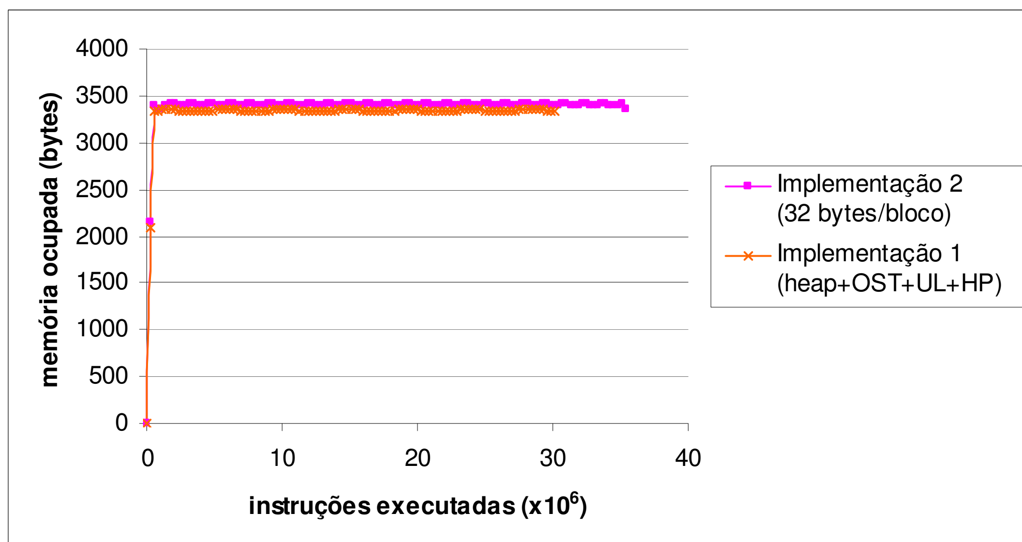

Figura 4.12: *Sokoban Game*: Utilização de memória: implementação 1 x implementação 2

4.2 Estimativas em desempenho, energia e potência

Nesta seção os resultados são apresentados para duas versões do processador FemtoJava: multiciclo [ITO 2001] e pipeline [BEC 2004]. As principais características arquiteturais destas versões são:

- **Multiciclo**: consiste em um microcontrolador com suporte a um reduzido conjunto de *bytecodes*, utiliza o modelo arquitetural de *Harvard* e suas instruções podem levar de 3 até 14 ciclos para serem completadas.
- **Pipeline**: esta versão, também conhecida como FemtoJava *Low-Power*, foi projetada visando a obtenção de um baixo consumo de energia, utilizando para isso um pipeline de cinco estágios e um mecanismo para detecção de dependências, além de um sistema para execução de *forwarding*. Os cinco estágios desta arquitetura são:
    - busca de instrução
    - decodificação
    - busca de operandos
    - execução
    - escrita de resultados

A idéia original para este trabalho era que todas as estimativas de desempenho, energia consumida e potência fossem realizadas através do simulador CACO-PS (*Cycle Accurate COnfigurable Power Simulator*) [BEC 2003]. Contudo, apenas algumas estimativas mais simples e as estimativas correspondentes ao *Address Book* puderam ser obtidas através deste simulador. Os resultados correspondentes às demais aplicações precisaram ser obtidos através de traces da execução dos programas e do cálculo baseado nos valores em ciclos, potência e energia utilizados por cada instrução em cada uma das arquiteturas.

Antes de explicar os motivos que levaram ao uso desta forma alternativa para obtenção dos resultados para estas aplicações, considera-se importante explicar primeiramente como ocorre o funcionamento do simulador. Sendo assim, os dados de entrada para o CACO-PS são:

- Uma descrição arquitetural (executável) da plataforma FemtoJava em linguagem C.
- Dois arquivos em formato *.mif* para inicialização das memórias RAM e ROM que são gerados pelo ambiente SASHIMI.

Com base nestas informações o procedimento realizado pelo simulador consiste em ler o conteúdo da ROM e processar os dados sobre a descrição de entrada. A potência dissipada pela arquitetura é calculada segundo o número de capacitâncias de *gate* que chaveiam durante a execução. Entretanto, os resultados em energia e potência foram convertidos e apresentados como medidas, respectivamente em nJ (nano Joules) e mW (mili Watts).

Considerando a forma como são calculados os dados pelo CACO-PS, os seguintes motivos impediram a coleta de resultados para o *MP3 Player* e o *Sokoban* através deste simulador:

- Como todas as palavras utilizadas para realizar o endereçamento das memórias RAM geradas pelo SASHIMI são de 16 bits, esta ferramenta não consegue endereçar corretamente memórias com tamanho maior do que 64K palavras. No caso do mp3, a menor RAM gerada teria um tamanho aproximado de 104K palavras. Desse modo a RAM para esta aplicação não pode ser gerada corretamente e conseqüentemente ocorre a perda de um dos dados de entrada para o simulador.

- A simulação do *Sokoban* no CACO-PS excedeu o tempo considerado suficiente para sua conclusão. Como o simulador não dispõe de recursos que possibilitem determinar o percentual já simulado nem o percentual ainda restante da simulação, acredita-se que sua execução tenha entrado em um laço infinito devido a algumas falhas na geração da ROM produzidas ocasionalmente durante algumas sínteses geradas através do SASHIMI.

Ainda com relação ao uso de traces para a obtenção de resultados para algumas aplicações, foi necessário utilizar um mecanismo especial para cálculo da energia consumida com a arquitetura multiciclo. Isto porque os únicos valores de energia consumida por instrução disponíveis correspondiam à arquitetura pipeline do FemtoJava.

Desse modo, com o intuito de estabelecer uma correlação entre os resultados apresentados, a solução adotada foi utilizar conjuntos de valores derivados a partir do conjunto existente para a arquitetura pipeline. Para isso, os valores de energia por instrução existentes para esta arquitetura, foram multiplicados pelas constantes 1,5, 1,75 e 2. Estima-se que a energia final obtida para as aplicações com o uso destas constantes possa ser um valor representativo.

Além disso, a análise da relação entre os dados levantados através do CACO-PS para o Address Book (seção 4.2.2) revelou um fator igual a 1,5 entre os valores das duas versões do processador, o que fornece bons indícios sobre a escolha das constantes utilizadas.

Por último, para todos os resultados apresentados utilizou-se uma freqüência de 50 MHz e Vdd igual a 3.3v, para ambas versões da arquitetura alvo. Todos os resultados mostram que as duas implementações possuem melhor desempenho e energia na versão pipeline do processador FemtoJava, porém com uma dissipação maior de potência para esta arquitetura.

4.2.1 Consumos para as Operações Básicas

As tabelas 4.9 e 4.10 apresentam os resultados para as operações básicas realizadas pelos dois SGDMs desenvolvidos: alocação, desalocação e operações de acesso aos objetos. As operações de acesso aos objetos são:

- *Putfield*: atribui um valor para determinado campo de um objeto.
- *Getfield*: captura o valor de um determinado campo de um objeto.

Comparando as duas versões implementadas é possível observar que os resultados de desempenho, potência e energia não possuem grande diferença entre eles. Isto se deve principalmente ao custo da manipulação de Strings e outras tarefas comuns aos dois algoritmos que consomem um grande número de ciclos.

Tabela 4.9: Custos para as operações básicas com a primeira implementação

	Multiciclo			Pipeline		
Operação	**Ciclos**	**Energia (nJ)**	**Potência (mW)**	**Ciclos**	**Energia (nJ)**	**Potência (mW)**
Alocação	1933	167,4216	4,3306	707	107,6006	7,6097
Desalocação	536	46,6002	4,3479	170	26,9002	7,9118
Putfield	2436	211,9316	4,3500	694	110,3441	7,9499
Getfield	2421	209,9509	4,3360	680	107,5050	7,9048

Tabela 4.10: Custos para as operações básicas com a segunda implementação

	Multiciclo			Pipeline		
Operação	**Ciclos**	**Energia (nJ)**	**Potência (mW)**	**Ciclos**	**Energia (nJ)**	**Potência (mW)**
Alocação	2151	186,5268	4,3358	701	111,1983	7,9314
Desalocação	597	51,6054	4,3221	239	36,8475	7,7087
Putfield	2718	236,8609	4,3573	804	130,2235	8,0985
Getfield	2717	235,2975	4,3301	796	128,3223	8,0604

A tab. 4.11 apresenta os custos correspondentes à execução do conjunto de métodos para manipulação de Strings da classe DmmsStr da primeira implementação. Assim como foi feito para a aplicação *Address Book*, optou-se em utilizar um tamanho fixo de 52 bytes para todas as Strings utilizadas como forma de simplificar a compreensão dos dados apresentados.

Tabela 4.11: Custos para os métodos da Classe DmmsStr com a primeira implementação

	Multiciclo			Pipeline		
Método	**Ciclos**	**Energia (nJ)**	**Potência (mW)**	**Ciclos**	**Energia (nJ)**	**Potência (mW)**
Aloc	2646	227,01	4,2898	1129	167,10	7,4004
charAt	162	13,48	4,1630	51	9,22	9,0471
compareTo	1817	155,66	4,2836	530	87,03	8,2104
Concat	3766	322,02	4,2754	1649	249,07	7,5522
Equals	1696	144,01	4,2457	454	74,06	8,1565
equalsignoreCase	2675	231,97	4,3360	696	113,66	8,1652
Indexof	582	50,64	4,3508	189	31,19	8,2535
lastIndexOf	1182	103,15	4,3636	373	60,48	8,1076
Length	53	4,30	4,0653	21	3,97	9,4756
Replace	2345	201,30	4,2921	979	150,02	7,6621
Substring	896	76,44	4,2661	397	59,23	7,4603

A tab. 4.12 apresenta o desempenho em ciclos para a manipulação de Strings através da segunda implementação. De um modo geral, a maioria dos métodos para esta implementação possuem um consumo maior que os obtidos para os mesmos métodos com a primeira implementação (tab. 4.11). Adicionalmente, alguns métodos nesta tabela possuem um consumo bastante superior se comparado aos dos métodos correspondentes no primeiro sistema, independentemente do tamanho de bloco utilizado. O motivo desta discrepância é

que a primeira implementação possui um melhor desempenho para estes métodos em virtude da característica de linearidade vinculada à forma como realiza o acesso aos dados em memória.

Para dar uma idéia mais clara da vantagem da primeira implementação sobre a segunda, caso uma aplicação utilizasse uma única vez cada um dos métodos disponibilizados, a diferença em ciclos existente apenas para a manipulação de Strings entre as duas implementações seria de 82%[5].

Restringindo-se a comparação apenas para os resultados da tab. 4.12, percebe-se que o uso de blocos maiores aumenta o desempenho para a maioria dos métodos apresentados. Diferentemente do que ocorre quanto ao consumo de memória, o consumo de energia apresenta um desempenho proporcional ao tamanho de bloco utilizado. Isto porque, em conseqüência do tamanho fixo estipulado para todas as Strings, o aumento do tamanho do bloco diminui o esforço realizado pelo sistema para realização de todas as suas operações.

Tabela 4.12: Ciclos para os métodos da Classe DmmsStr com a segunda implementação

Método	Multiciclo			Pipeline		
	Tamanho do bloco de memória (bytes)					
	16	32	64	16	32	64
Aloc	3671	4149	3063	1268	1470	1124
charAt	770	658	495	236	248	206
compareTo	18262	15618	14118	6395	5749	5295
Concat	41078	33495	29847	14268	12472	11473
Equals	2369	2231	2231	894	844	844
equalsignoreCase	18343	15699	14199	6418	5772	5318
Indexof	3905	3445	3376	1434	1302	1277
lastIndexOf	9327	8005	7255	3282	2959	2732
Length	46	46	46	19	19	19
Replace	19266	16390	14630	6944	6234	5720
Substring	9354	7768	7148	3328	2943	2755

A tab. 4.13 apresenta os consumos em energia para a segunda implementação. De um modo geral, a energia consumida também foi bastante elevada para alguns métodos, respeitando portanto a proporção com o acentuado número de ciclos apresentados na tab. 4.12.

---

[5] Valor alcançado considerando a comparação apenas com a melhor performance obtida pela segunda implementação (utilizando blocos de 64 bytes).

Tabela 4.13: Energia (nJ) para os métodos da Classe DmmsStr com a segunda implementação

Método	Multiciclo			Pipeline		
	Tamanho do bloco de memória (bytes)					
	16	32	64	16	32	64
Aloc	318,35	359,23	264,92	202,85	234,36	177,87
charAt	67,88	58,13	43,89	38,04	39,79	32,51
compareTo	1579,52	1350,96	1220,40	1050,41	938,22	864,10
Concat	3575,57	2914,25	2594,66	2348,51	2036,15	1866,61
Equals	206,01	193,40	193,76	139,99	132,71	134,64
equalsignoreCase	1601,70	1371,49	1239,88	1053,28	941,56	866,79
Indexof	340,32	300,07	293,59	233,29	210,31	206,16
lastIndexOf	810,40	695,93	630,18	535,20	479,50	442,84
Length	4,60	4,72	4,50	2,90	2,21	2,08
Replace	1682,58	1430,94	1275,82	1135,69	1014,27	929,11
Substring	816,22	677,62	622,95	546,30	478,65	448,40

A tab. 4.14 apresenta a potência média dissipada pelos métodos de manipulação de Strings da segunda implementação. Em média, o primeiro sistema consumiu menos potência que o segundo sistema. De um modo geral, a variação do tamanho do bloco de memória não influenciou significativamente na potência consumida por cada método.

Tabela 4.14: Potência (mW) para os métodos da classe DmmsStr com a segunda implementação

Método	Multiciclo			Pipeline		
	Tamanho do bloco de memória (bytes)					
	16	32	64	16	32	64
Aloc	4,3361	4,3292	4,3245	7,9989	7,9718	7,9125
charAt	4,4083	4,4172	4,4340	8,0601	8,0236	7,8923
compareTo	4,3246	4,3250	4,3222	8,2128	8,1599	8,1597
Concat	4,3522	4,3503	4,3466	8,2300	8,1629	8,1348
Equals	4,3480	4,3345	4,3426	7,8298	7,8623	7,9764
equalsignoreCase	4,3660	4,3681	4,3661	8,2057	8,1563	8,1497
Indexof	4,3576	4,3552	4,3482	8,1343	8,0765	8,0724
lastIndexOf	4,3444	4,3469	4,3431	8,1536	8,1025	8,1047
Length	5,0050	5,1340	4,8964	7,6316	5,8293	5,4771
Replace	4,3667	4,3653	4,3603	8,1775	8,1350	8,1216
Substring	4,3630	4,3616	4,3576	8,2077	8,1321	8,1380

4.2.2 Address Book

A tab. 4.15 apresenta as estimativas em ciclos e energia para a aplicação. A primeira coluna da tab. 4.15 indica o SGDM utilizado. A segunda coluna informa o tamanho do bloco de memória utilizado para a segunda implementação. A terceira coluna informa a arquitetura do processador FemtoJava utilizada. Na quarta e quinta coluna são apresentados, respectivamente, o número total de ciclos e a energia consumida para execução de toda a aplicação. A sexta e sétima coluna apresentam, respectivamente, o número de ciclos e a energia consumida apenas pela gerência dinâmica de memória. A oitava coluna apresenta a relação

percentual entre os dados apresentados na quarta e sexta coluna. Desta forma, a oitava coluna apresenta o percentual do consumo produzido por cada sistema em relação ao consumo total, resultante da execução completa da aplicação.

Tabela 4.15: *Address Book*: Ciclos e energia

SGDM	Tam. Bloco (bytes)	FemtoJava Arq.	Total Ciclos	Total Energia (nJ)	SGDM Ciclos	SGDM Energia (nJ)	SGDM x Total (%)
1	-	Multiciclo	830712	71740	648328*1	59982*1	78,04
1	-	Pipeline	316029	49530	226104*2	38882*2	71,55
2	16	Multiciclo	976444	85131	853607	74589	87,42
2	16	Pipeline	351412	56532	286810	46999	81,62
2	32	Multiciclo	958890	83564	831609	72621	86,73
2	32	Pipeline	348587	56006	283069	46276	81,20
2	64	Multiciclo	896666	78117	778201	67949	86,79
2	64	Pipeline	330239	52831	267441	43579	80,98

*1 - deste valor 6,40% são relativos ao consumo com a compactação (equivalente a 44366 ciclos e 3806,68 nJ)

*2 - deste valor 7,07% são relativos ao consumo com a compactação (equivalente a 17206 ciclos e 2688,77 nJ)

Na tab. 4.15 o primeiro sistema novamente leva vantagem no consumo de ciclos e energia. Isto ocorre devido a três principais motivos:

- O uso de compactação no sistema 1 permite que suas operações básicas e métodos da classe DmmsStr desempenhem um consumo menor em conseqüência da disposição simplificada dos dados na memória.
- Devido à sua simplicidade, o processo de compactação mantém um baixo consumo em ciclos e energia se comparado ao consumo total obtido para toda a gerência de memória.
- O sistema 2 consome mais ciclos e energia para atender requisitos de tempo real. O algoritmo é projetado para sempre manipular a quantidade de memória estritamente necessária e como conseqüência disto suas operações são mais custosas.

Entretanto, a não utilização de compactação pelo segundo sistema ameniza um pouco a maior complexidade deste algoritmo. A diferença máxima de desempenho entre os resultados obtidos com o primeiro e o segundo sistema, considerando o custo total para execução da aplicação, é 14,9% para a arquitetura multiciclo e 10,1% para a arquitetura pipeline. Considerando apenas os custos relativos à gerência dinâmica de memória, esta diferença é de 24% para a arquitetura multiciclo e 21,2% para a arquitetura pipeline.

A tab. 4.16 mostra outras comparações para os resultados apresentados na tab. 4.15. Considera-se os dados apresentados na tab. 4.16 válidos tanto para a arquitetura multiciclo quanto para a pipeline, já que existe uma proximidade significativa entre os valores obtidos com cada uma dessas arquiteturas.

Tabela 4.16: *Address Book*: Comparações de desempenho e energia

Comparação	Diferença Total aplicação (%)	Diferença SGDM (%)
SGDM1 x SGDM2 (16 *bb)	10,1	21,2
SGDM1 x SGDM2 (32 *bb)	9,3	20,1
SGDM1 x SGDM2 (64 *bb)	4,3	15,5
SGDM2 (16 *bb) x SGDM2 (32 *bb)	0,8	1,3
SGDM2 (16 *bb) x SGDM2 (64 *bb)	6,0	6,8
SGDM2 (32 *bb) x SGDM2 (64 *bb)	5,3	5,5
bb: bytes/bloco		

Na tab. 4.17 está apresentada a potência média dissipada durante a execução da aplicação. A quarta coluna informa a potência média consumida para toda a aplicação. A quinta coluna apresenta a potência dissipada exclusivamente pela gerência de memória. Por último, a sexta coluna apresenta a potência dissipada exclusivamente pela compactação de memória.

Em média, a relação entre a potência consumida através da arquitetura multiciclo e a potência dissipada pela arquitetura pipeline foi de 67,1%.

Tabela 4.17: *Address Book*: Potência média dissipada

SGDM	Tam. Bloco (bytes)	FemtoJava Arq.	Total Potência Média (mW)	SGDM Potência Média (mW)	SGDM Compact. (mW)
1	-	Multiciclo	4,3180	4,3297	4,2901
1	-	Pipeline	7,8364	7,9904	7,8135
2	16	Multiciclo	4,3593	4,3690	-
2	16	Pipeline	8,0436	8,1935	-
2	32	Multiciclo	4,3573	4,3663	-
2	32	Pipeline	8,0333	8,1740	-
2	64	Multiciclo	4,3560	4,3658	-
2	64	Pipeline	7,9990	8,1475	-

4.2.3 Mp3 Player

Conforme mencionado anteriormente, os resultados relativos a esta aplicação foram obtidos considerando a decodificação de 5 frames de um arquivo de áudio no formato mp3. Entretanto, um levantamento dos custos correspondentes à decodificação de cada um desses frames foi realizado como forma de estabelecer uma relação entre o consumo da aplicação e o número de frames decodificados.

Para isto foi necessário realizar uma análise do comportamento da aplicação para determinar o ponto inicial da decodificação de cada frame. Desta análise identificou-se que a aplicação possui dois estágios para decodificação de um frame: uma decodificação inicial mais complexa e uma decodificação final efetuada antes do envio dos dados para a saída.

Uma peculiaridade deste mecanismo de decodificação reside no fato de que estes dois estágios não são executados continuamente durante a decodificação de um mesmo frame. A fig. 4.12 mostra como ocorre a decodificação de cada frame pela aplicação, onde em cada coluna os blocos de decodificação superiores são executados antes dos inferiores.

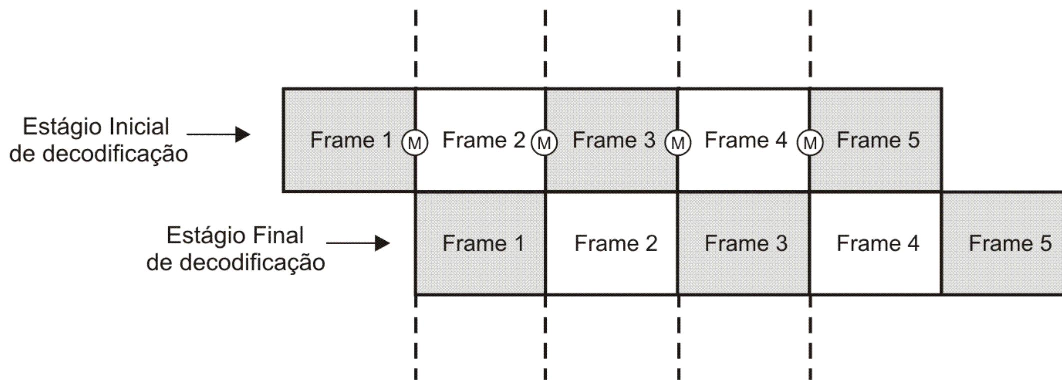

Figura 4.12: Decodificação dos frames pelo MP3 Player

Sendo assim, optou-se por simplicidade em realizar a análise individual apenas para os frames centrais (2, 3 e 4) na seqüência de decodificação. Dessa forma, foi possível estabelecer grupos compostos de um único estágio inicial para cada estágio final e vice-versa. Na fig. 4.12 os círculos indicam o ponto onde inicia cada medição e as colunas limitam os conjuntos utilizados para coleta de informações.

Embora estes conjuntos sejam formados por um estágio de decodificação inicial e final relativos a frames diferentes, considerou-se o resultado obtido através de qualquer um dos conjuntos equivalente ao resultado de um frame ou uma unidade de execução da aplicação. O anexo 1 apresenta um detalhamento dos custos em ciclos e energia para o segundo, terceiro e quarto frame, considerando as duas implementações realizadas.

Na tab. 4.18, cada valor apresentado corresponde à diferença máxima obtida entre os frames analisados, para a correspondente estimativa. A tab. 4.19 apresenta a média dos resultados obtidos para estes mesmos frames.

Tabela 4.18: *MP3 Player*: Diferenças máximas para os 3 frames considerados

SGDM	Tam. Bloco (bytes)	Arq.	Fator	Total Ciclos	Total Energia (nJ)	SGDM Ciclos	SGDM Energia (nJ)
1	-	Mult.	1,5	27400327	6044651	25535118	5599756
			1,75		7052048		6533014
			2		8059228		7466051
1	-	Pipe	-	5566895	4029614	5205714	3733025
2	16	Mult.	1,5	92982370	17624498	91118397	17178603
			1,75		20561634		20041434
			2		23498594		22904084
2	16	Pipe	-	17238616	11749297	16879844	11452042
2	32	Mult.	1,5	88117344	16800398	86253371	16354504
			1,75		19600196		19079995
			2		22399821		21805312
2	32	Pipe	-	16389309	11199911	16030537	10902656
2	64	Mult.	1,5	85486527	16341986	83622554	15896091
			1,75		19065388		18545187
			2		21788621		21194111
2	64	Pipe	-	15920003	10894310	15561231	10597056

Tabela 4.19: *MP3 Player*: Média dos resultados para os três frames analisados

SGDM	Tam. Bloco (bytes)	Arq.	Fator	Total Ciclos	Total Energia (nJ)	SGDM Ciclos	SGDM Energia (nJ)
1	-	Mult.	1,5	634604562	118367345	618845299	114519581
			1,75		138093327		133604275
			2		157818337		152688069
1	-	Pipe	-	116396171	78909168	113665860	76344034
2	16	Mult.	1,5	884468203	165300729	868708941	161452964
			1,75		192848285		188359232
			2		220394245		215263977
2	16	Pipe	-	161170207	110197122	158439896	107631988
2	32	Mult.	1,5	700431467	133515087	684672204	129667322
			1,75		155765497		151276444
			2		178014600		172884332
2	32	Pipe	-	129405454	89007300	126675143	86442166
2	64	Mult.	1,5	628493115	121112781	612733853	117265017
			1,75		141296326		136807274
			2		161478674		156348406
2	64	Pipe	-	116892565	80739337	114162253	78174203

A tab. 4.20 apresenta a relação percentual entre as diferenças máximas obtidas com os três frames analisados e a média dos valores obtidos para estes frames. Os resultados apresentados na tab. 4.20 mostram que a variação do percentual de diferenças entre as estimativas obtidas para cada frame não ultrapassa 13,65%. Este valor fornece uma noção significativa do percentual de erro para realização de inferências quanto aos custos obtidos para a decodificação de qualquer número de frames através desta aplicação.

Tabela 4.20: *MP3 Player*: Percentual de diferenças em relação à média para os 3 frames centrais

SGDM	Tam. Bloco (bytes)	Arq.	Total Ciclos	Total Energia (nJ)	SGDM Ciclos	SGDM Energia (nJ)
1	-	Multiciclo	4,32	5,11	4,13	4,89
1	-	Pipeline	4,78	5,11	4,58	4,89
2	16	Multiciclo	10,51	10,66	10,49	10,64
2	16	Pipeline	10,70	10,66	10,65	10,64
2	32	Multiciclo	12,58	12,58	12,60	12,61
2	32	Pipeline	12,67	12,58	12,65	12,61
2	64	Multiciclo	13,60	13,49	13,65	13,56
2	64	Pipeline	13,62	13,49	13,63	13,56

A tab. 4.21 apresenta os custos totais para decodificação de todos os 5 frames do arquivo mp3 utilizado para esta aplicação. A diferença máxima de desempenho obtida entre as duas implementações, considerando o custo total para execução da aplicação chega a 42,7% para a arquitetura multiciclo. Considerando apenas os custos relativos à gerência dinâmica de memória, esta diferença é de 43,3% para a arquitetura multiciclo. A tab. 4.22 mostra outras comparações para os resultados apresentados na tab. 4.21.

Tabela 4.21: *MP3 Player*: Ciclos e energia para todos os frames

SGDM	Tam. Bloco (bytes)	Arq.	Fator	Total Ciclos	Total Energia (nJ)	SGDM Ciclos	SGDM Energia (nJ)
1	-	Mult.	1,5	3379738836	631141269	3298275697	611248698
			1,75		736321346		713113382
			2		841496168		814973182
1	-	Pipe	-	619784518	420748084	605677213	407486591
2	16	Mult.	1,5	6021415588	1107649238	5939952449	1087756668
			1,75		1292239198		1269031235
			2		1476819963		1450296977
2	16	Pipe	-	1082087354	738409982	1067980049	725148489
2	32	Mult.	1,5	4290463111	807771035	4208999972	787878465
			1,75		942386355		919178391
			2		1076994442		1050471456
2	32	Pipe	-	784772248	538497221	770664943	525235728
2	64	Mult.	1,5	3606083194	689308156	3524620055	669415585
			1,75		804181697		780973734
			2		919048757		892525772
2	64	Pipe	-	666744040	459524379	652636735	446262886

A relação média aproximada entre o consumo realizado pela gerência dinâmica de memória em relação ao consumo total da aplicação é de 98%. A razão disto é que a maior parte das operações realizadas por esta aplicação envolvem a manipulação de arrays. Estes elementos já eram suportados pela arquitetura FemtoJava antes da realização deste trabalho, entretanto tais elementos não eram tratados como elementos dinâmicos da aplicação. Através deste trabalho estes elementos passaram a ser reconhecidos como estruturas dinâmicas na memória o que resultou no envolvimento de todos os custos relacionados a estas estruturas.

Tabela 4.22: *MP3 Player*: Comparação do desempenho e energia

Comparação	Diferença Total aplicação (%)	Diferença SGDM (%)
SGDM1 x SGDM2 (16 *bb)	42,7	43,3
SGDM1 x SGDM2 (32 *bb)	21,0	21,4
SGDM1 x SGDM2 (64 *bb)	7,0	7,2
SGDM2 (16 *bb) x SGDM2 (32 *bb)	27,5	27,8
SGDM2 (16 *bb) x SGDM2 (64 *bb)	38,4	38,9
SGDM2 (32 *bb) x SGDM2 (64 *bb)	15,0	15,3
bb: bytes/bloco		

A tab. 4.23 mostra o consumo produzido pela compactação de memória durante a execução dos 5 frames através primeira implementação. A última coluna da tab. 4.23 informa o percentual consumido pela compactação em relação ao consumo total produzido pela gerência dinâmica de memória.

Tabela 4.23: *MP3 Player*: Consumo total produzido pela compactação

Arq.	Fator	Ciclos	Energia (nJ)	Compact. x SGDM (%)
Multiciclo	1,5	744565846	128020763	20,94
	1,75		149354896	
	2		170689690	
Pipeline	-	130607276	85344845	

A tab. 4.24 mostra para a arquitetura pipeline a influência da compactação sobre os custos com a gerência de memória em cada um dos três frames centrais. Os resultados apresentados na tab. 4.24 mostram que a compactação também manteve uma razoável regularidade durante a decodificação destes frames.

Tabela 4.24: *MP3 Player*: Influência da compactação para cada frame decodificado

Frame	Ciclos Compact.	Energia (nJ) Compact.	Ciclos SGDM	Energia (nJ) SGDM	Compact. x SGDM (%)
2	125575828	19961336	110243183	73905484	27,93
3	22056195	14327847	115448897	77638509	19,10
4	22056195	14327847	115305499	77488110	19,13
Média	24966275	16205677	113665860	76344034	22,05

A tab. 4.25 exibe a potência dissipada ao longo da execução dos 5 frames utilizados para esta aplicação. Considerando o conjunto de constantes utilizadas para produzir as aproximações correspondentes à arquitetura multiciclo, a relação entre a potência consumida por esta arquitetura em relação à versão pipeline foi em média de 67,9%. Este valor é bastante próximo do valor correspondente obtido para o Address Book, cujas estimativas foram todas realizadas através de simulação ciclo a ciclo, de modo que considera-se que isto seja um bom indício sobres os valores das constantes de aproximação utilizadas para obter os resultados relativos à arquitetura multiciclo.

Tabela 4.25: *MP3 Player*: Potência média dissipada

SGDM	Tam. Bloco (bytes)	FemtoJava Arq.	Fator	Total Potência Média (mW)	SGDM Potência Média (mW)	SGDM Compact. (mW)
1	-	Multiciclo	1,5	9,3371	9,2662	8,5970
			1,75	10,8932	10,8104	10,0297
			2	12,4491	12,3545	11,4624
1	-	Pipeline	-	33,9431	33,6389	32,6723
2	16	Multiciclo	1,5	9,1976	9,1563	-
			1,75	10,7304	10,6822	-
			2	12,2631	12,2080	-
2	16	Pipeline	-	34,1197	33,9495	-
2	32	Multiciclo	1,5	9,4136	9,3594	-
			1,75	10,9823	10,9192	-
			2	12,5510	12,4789	-
2	32	Pipeline	-	34,3091	34,0768	-
2	64	Multiciclo	1,5	9,5576	9,4963	-
			1,75	11,1503	11,0788	-

SGDM	Tam. Bloco (bytes)	FemtoJava Arq.	Fator	Total Potência Média (mW)	SGDM Potência Média (mW)	SGDM Compact. (mW)
			2	12,7430	12,6613	-
2	64	Pipeline	-	34,4603	34,1892	-

4.2.4 Sokoban Game

A tab. 4.26 apresenta o desempenho quanto ao volume de ciclos e a energia consumidos por esta aplicação. A diferença máxima entre os resultados obtidos através de cada implementação, considerando o custo total para execução da aplicação, é de 16,3% para a arquitetura multiciclo. Considerando apenas os custos relativos à gerência dinâmica de memória, esta diferença é de 16,5% para a mesma arquitetura. A tab. 4.27 mostra outras comparações para os resultados apresentados na tab. 4.26.

Tabela 4.26: *Sokoban Game*: Ciclos e energia

SGDM	Tam. Bloco (bytes)	Arq.	Fator	Total Ciclos	Total Energia (nJ)	SGDM Ciclos	SGDM Energia (nJ)
1	-	Mult.	1,5	164447000	30363330	160753623	29361423
			1,75		35423376		34254479
			2		40483091		39147228
1	-	Pipe	-	30909713	20241546	30382247	19573614
2	16	Mult.	1,5	195334577	36894138	191641200	35892230
			1,75		43042524		41873627
			2		49190563		47854699
2	16	Pipe	-	36913709	24595282	36386243	23927350
2	32	Mult.	1,5	190229400	35948508	186536023	34946600
			1,75		41939300		40770403
			2		47929760		46593897
2	32	Pipe	-	35963099	23964880	35435633	23296948
2	64	Mult.	1,5	187884894	35517097	184191517	34515189
			1,75		41435993		40267096
			2		47354564		46018701
2	64	Pipe	-	35515601	23677282	34988135	23009350

Uma característica diferenciada observada através da execução desta aplicação foi a não necessidade de ocorrência de compactação. Conforme ilustrado pela fig. 4.9 (seção 4.1.3), isto ocorre porque durante a mudança de posição dentro do cenário (único momento em que ocorre a desalocação de um objeto) o objeto desalocado que contém as informações sobre a posição abandonada tem imediatamente seu espaço ocupado por um objeto do mesmo tamanho. Desse modo, como após realizar a última mudança de posição o sistema não necessita alocar nenhum outro objeto, nenhuma quantidade de memória precisa ser disponibilizada pela compactação.

Tabela 4.27: *Sokoban Game*: Comparação do desempenho e energia

Comparação	Diferença Total aplicação (%)	Diferença SGDM (%)
SGDM1 x SGDM2 (16 *bb)	16,3	16,5
SGDM1 x SGDM2 (32 *bb)	14,1	14,3
SGDM1 x SGDM2 (64 *bb)	13,0	13,2
SGDM2 (16 *bb) x SGDM2 (32 *bb)	2,6	2,6
SGDM2 (16 *bb) x SGDM2 (64 *bb)	3,8	3,8
SGDM2 (32 *bb) x SGDM2 (64 *bb)	1,2	1,3
bb: bytes/bloco		

A tab. 4.28 exibe a potência média dissipada pela aplicação. Considerando o conjunto de constantes utilizadas para produzir as aproximações correspondentes à arquitetura multiciclo, a relação entre a potência consumida por esta arquitetura em relação à versão pipeline foi em média de 67,1%.

Tabela 4.28: *Sokoban Game*: Potência média dissipada

SGDM	Tam. Bloco (bytes)	FemtoJava Arq.	Fator	Total Potência Média (mW)	SGDM Potência Média (mW)
1	-	Multiciclo	1,5	9,2320	9,1324
			1,75	10,7705	10,6543
			2	12,3089	12,1762
1	-	Pipeline	-	32,7430	32,2123
2	16	Multiciclo	1,5	9,4438	9,3644
			1,75	11,0176	10,9250
			2	12,5914	12,4855
2	16	Pipeline	-	33,3146	32,8797
2	32	Multiciclo	1,5	9,4487	9,3673
			1,75	11,0233	10,9283
			2	12,5979	12,4892
2	32	Pipeline	-	33,3187	32,8722
2	64	Multiciclo	1,5	9,4518	9,3694
			1,75	11,0270	10,9308
			2	12,6020	12,4921
2	64	Pipeline	-	33,3336	32,8816

4.3 Estimativas Finais sobre Este Trabalho

4.3.1 Impactos sobre o Conjunto de Instruções da Arquitetura

A fig. 4.13 apresenta os impactos sobre o conjunto de instruções da arquitetura alvo. Uma vez que este trabalho tem por objetivo estender a capacidade da ferramenta Sashimi para suporte às aplicações, foram consideradas instruções do conjunto pré-existente da arquitetura FemtoJava somente as instruções suportadas pela síntese através desta ferramenta.

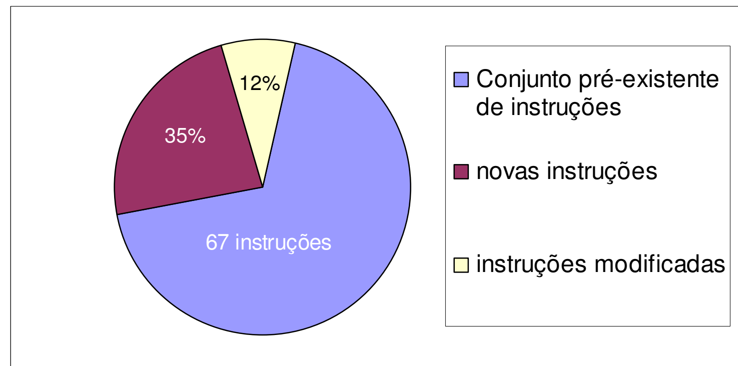

Figura 4.13: Impactos sobre o conjunto de instruções

A tab. 4.29 apresenta as novas instruções suportadas pela arquitetura assim como as instruções, já existentes, modificadas por este trabalho.

Tabela 4.29: Modificações sobre o conjunto de instruções da arquitetura

Novas Instruções	aaload, aastore, aconst_null, aload_0, aload_1, aload_2, aload_3, aload_<n>, anew*array*, astore_0, astore_1, astore_2, astore_3, astore_<n>, getfield, invokevirtual, areturn, ldc_w, ldc_2w, multianew*array*, new, new*array*, putfield
Instruções Modificadas	saload, sastore, iastore, iaload, caload, castore, baload, bastore

A Fig. 4.14 mostra o percentual atingido pelo novo conjunto de instruções do FemtoJava em relação ao conjunto de instruções da JVM. Com as novas instruções o conjunto atual da arquitetura passa a corresponder a 44% do total de instruções existentes na máquina virtual.

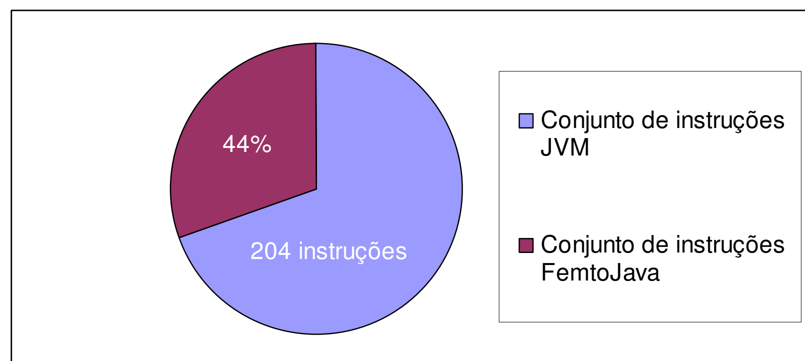

Figura 4.14: Comparação conjunto de Instruções FemtoJava x JVM

4.3.2 Código Desenvolvido

Ao todo foram escritas para este trabalho 16.091 linhas de código, das quais 2.471 correspondem as duas implementações realizadas, 7.594 correspondem à ferramenta de adaptação de código e 6.026 correspondem a outros utilitários de uso geral também desenvolvidos. Uma descrição mais detalhada sobre cada código desenvolvido pode ser encontrada no segundo anexo ao final deste texto.

4.3.3 Nota sobre Utilização das Implementações

Uma vez que atualmente a escolha do algoritmo para gerência de memória deve ser feita pelo usuário, o mesmo deve conhecer claramente as vantagens e desvantagens de cada um dos algoritmos aqui propostos para que possa optar corretamente por aquele mais adequado aos seus requisitos.

Neste sentido, a escolha do algoritmo a ser utilizado deve considerar primeiramente a existência de restrições de tempo-real entre os requisitos a serem respeitados. Caso a aplicação não exija previsibilidade ou ainda se aplicação não possuir *deadlines* rigorosos a serem alcançados, é possível utilizar-se o primeiro algoritmo proposto neste trabalho. Conforme demonstrado ao longo deste capítulo, este algoritmo apresenta em geral um desempenho melhor para todos os principais requisitos relacionados à sistemas embarcados, tais como utilização de memória, dissipação de potência, consumo de ciclos e energia. Entretanto, se o principal requisito a ser considerado estiver relacionado à utilização de memória, torna-se interessante a realização de uma análise de custos para cada caso, já que, embora o primeiro algoritmo não desperdice memória através de fragmentação interna ou externa e possa ainda manter um *overhead* aproximadamente constante para a alocação de qualquer objeto, a possibilidade de configuração do tamanho de bloco utilizado pelo segundo algoritmo é capaz de resultar em um desempenho melhor para alguma determinada aplicação.

Por outro lado, caso a aplicação exija que seus *deadlines* sejam rigorosamente alcançados, a escolha deve ser feita pelo segundo algoritmo proposto. Neste caso, deve-se buscar um ajuste adequado do tamanho de bloco a ser utilizado, como forma de maximizar a performance obtida para os demais requisitos.

Conforme demonstrado em seções anteriores deste capítulo, a escolha de um tamanho de bloco grande (não excessivamente), em geral, resulta em um bom desempenho com relação ao consumo de ciclos e conseqüentemente também energia[6]. Entretanto, esta escolha pode levar a um desperdício desnecessário dos recursos de memória. Logo, a escolha do tamanho de bloco ideal deve partir da identificação do melhor tamanho de bloco com relação ao uso de memória, uma vez que, conforme demonstrado este tamanho de bloco tende a oferecer um desempenho igual ou bastante próximo do desempenho capaz de ser obtido com o uso de blocos de tamanho maior.

Para isto, a estratégia a ser utilizada consiste em primeiramente identificar o tamanho médio dos objetos alocados pela a aplicação. Assim, deve-se iniciar a busca pelo tamanho de bloco ideal utilizando-se um tamanho de bloco grande apenas o suficiente para armazenar um objeto de tamanho médio. Um cuidado especial deve ser tomado com relação ao *overhead* necessário para armazenamento do cabeçalho do objeto (seção 3.3), já que a desconsideração deste custo pode fazer com que o sistema aloque um bloco extra apenas para armazenamento destas informações. Em seguida, é aconselhável realizar a experimentação de outros tamanhos de bloco, como forma de possibilitar uma identificação mais precisa do tamanho do bloco a ser utilizado.

---

[6] Cabe aqui ressaltar que a potência dissipada, que é outro importante parâmetro de projeto para sistemas embarcados, demonstrou pouca variabilidade com relação ao tamanho de bloco utilizado.

Ainda levando em consideração os resultados apresentados neste capítulo, caso a escolha seja feita pelo segundo algoritmo desenvolvido, o custo adicional em relação à utilização do primeiro algoritmo será, em média, de aproximadamente 2% com relação à potência média dissipada, 16% para o número de ciclos executados, 18% para a energia consumida e 10% para a quantidade total de memória utilizada. Conforme demonstrado, este custo pode ser bem superior caso o usuário não defina adequadamente o tamanho de bloco a ser utilizado com o segundo algoritmo.

Por fim, uma vez que a solução disponibilizada através deste trabalho se aplica a todas as versões existentes da arquitetura FemtoJava, o usuário dispõe ainda da possibilidade de escolha da arquitetura mais adequada a seus objetivos. A exemplo disto, neste capítulo demonstrou-se que o usuário é capaz de realizar a substituição de uma arquitetura Pipeline por outra Multiciclo, obtendo uma redução em potência de aproximadamente 2,5 vezes a um custo extra em ciclos e energia, respectivamente de 4,3 e 1,7 vezes.

5 CONCLUSÕES

Este trabalho apresenta duas implementações de algoritmos para gerência dinâmica de memória em *software*, as quais tiveram como alvo uma plataforma embarcada Java. Uma vez que a plataforma utilizada pertence a uma metodologia de geração semi-automática de *hardware* e *software* para sistemas embarcados, os dois sistemas implementados foram projetados para também se inserirem no contexto desta mesma metodologia.

Como forma de estabelecer comparações detalhadas entre as duas implementações desenvolvidas foram realizadas diversas estimativas em desempenho, potência e energia para cada implementação, utilizando para isto duas versões existentes da arquitetura adotada. Através da análise dos resultados obtidos, observou-se que um dos algoritmos desenvolvidos obteve um desempenho melhor para realização da gerência dinâmica da memória. Em contrapartida, o outro algoritmo possui características de projeto que possibilitam sua utilização com aplicações de tempo-real.

De um modo geral, os custos adicionais obtidos para a utilização do algoritmo de tempo-real são, em média, de aproximadamente de 2% em potência, 16% em desempenho (ciclos), 18% em energia e 10% sobre a quantidade de memória total utilizada. Isto mostra que o custo extra necessário para utilização do algoritmo de tempo real é razoavelmente baixo se comparado aos benefícios proporcionados pela utilização deste algoritmo.

Por último, considerando o conjunto de instruções da arquitetura utilizada, obteve-se um acréscimo de 35% sobre o número de instruções suportadas por esta arquitetura. Adicionalmente, 12% das instruções que já existiam no conjunto desta arquitetura foram modificadas para se adaptarem aos novos mecanismos implementados. Com as novas instruções o conjunto atual da arquitetura passa a corresponder a 44% do total de instruções existentes na arquitetura da máquina virtual Java.

5.1 Propostas de Trabalhos Futuros

A JVM estabelece o uso de uma tabela de símbolos denominada *constant pool* para resolução de informações necessárias à execução de programas Java. Entre as informações mantidas por esta tabela, estão dados vitais para a realização do gerenciamento dinâmico de memória. Uma vez que a arquitetura alvo não estabelecia o uso de uma constant pool para suporte à execução de suas aplicações, esta tabela precisou ser incorporada ao conjunto de informações mantidas na memória da arquitetura. Com isto, foi mantida a compatibilidade entre estrutura global da plataforma utilizada e especificação da máquina Java.

Entretanto, considerando todas as restrições inerentes ao projeto de sistemas embarcados, uma alternativa para aumentar o desempenho global de ambos os sistemas desenvolvidos consistiria em uma análise de possíveis

elementos a serem calculados ainda durante a compilação de cada sistema. Isto permitiria o aumento do desempenho do sistema com proporcional redução da energia consumida para realização das mesmas tarefas, uma vez que tais elementos deixariam de ser calculados em tempo de execução. Dentre o conjunto de elementos a serem cobertos por esta análise, estima-se que a constant pool contenha um forte potencial para redução dos custos de processamento apresentados neste trabalho. Nesse sentido, outro benefício trazido por esta adaptação seria uma redução significativa dos recursos em memória necessários para suporte a aplicações com alocação dinâmica de memória.

Especificamente com relação ao primeiro sistema desenvolvido, outra possibilidade para obtenção de melhores resultados está relacionada à otimização e à exploração do uso da compactação de memória. Conforme algumas tendências apresentadas neste trabalho ([DEL 2002] e [CHE 2002]), a compactação de memória pode ser utilizada como forma de reduzir o consumo de energia, através do desligamento de uma parcela da memória que tende a não ser utilizada durante um determinado período da execução da aplicação. Neste sentido, alguns experimentos poderiam ser conduzidos com relação à aplicação de uma estratégia semelhante ao trabalho desenvolvido como forma de verificar os impactos sobre o consumo de energia da primeira implementação.

Alternativamente, outra estratégia potencialmente válida a ser estudada ainda com relação à primeira implementação é o desenvolvimento de um algoritmo de compactação escalável para tempo-real. Uma possibilidade neste sentido está vinculada a uma estratégia para reuso de objetos, tal como a desenvolvida por [DET 2004]. Neste caso, uma proposta consiste em fazer com que cada objeto, tão logo se torne inútil para a aplicação, seja imediatamente reciclado. Para isto, deve ser movimentado para o espaço ocupado por este objeto, com o mesmo custo de uma alocação, aquele objeto (casa exista) da mesma classe armazenado o mais próximo possível do fim da heap. Com isto, estima-se que seja possível obter uma compactação incremental dos dados e uma liberação natural de uma grande parcela de memória para desligamento.

Considerando ainda a disponibilidade para realização de otimizações no nível da arquitetura em *hardware*, outra possível forma de continuação deste trabalho consiste na análise e mapeamento de alguns elementos de cada implementação para *hardware*, como forma de obtenção de desempenhos e consumos ainda melhores. Neste sentido, a redução do consumo de energia através do desligamento de bancos de memória poderia ser implementada também para o segundo sistema desenvolvido neste trabalho. Para isto, haveria a necessidade de se analisar a viabilidade para realização do desligamento de porções pequenas de memória, o que poderia levar a uma exploração parcial do espaço de projeto de memórias.

Ainda no contexto de exploração do espaço de projeto com foco para redução da fragmentação de memória e melhoria do consumo de energia, outro possível experimento a ser realizado é a construção de um sistema que realize dinamicamente a modificação do tamanho de bloco utilizado conforme os requisitos da aplicação. Entretanto, os ganhos relacionados ao uso desta estratégia dependem principalmente da identificação de aplicações com algumas características específicas, tais como:

- A aplicação deveria variar seus requisitos de tamanho de bloco durante sua execução.
- A cada mudança do tamanho de bloco a aplicação deveria executar por um tempo suficientemente longo com o novo tamanho de tal modo que

os custos resultantes desta modificação fossem justificados pelos ganhos obtidos.

Estas características são atípicas para a maioria das aplicações, entretanto, devido ao avanço tecnológico torna-se cada vez mais comum o surgimento de dispositivos capazes de executar múltiplas aplicações, o que tende conseqüentemente a proporcionar espaço para o projeto de sistemas de gerência dinâmica de memória reconfiguráveis durante a execução.

Outra proposta interessante para continuação deste trabalho seria a realização da integração de um escalonador de tarefas a ambos mecanismos para gerência de memória. Para este trabalho seria necessário a utilização de um escalonador também adequado às características da arquitetura FemtoJava, tal como o já proposto por [DAR 2003]. O escalonador utilizado deveria ainda levar em consideração alguns requisitos das aplicações utilizadas, tal como a quantidade máxima de ciclos para execução e o volume mínimo de memória necessária, apresentados neste trabalho. Algumas das contribuições de um escalonador de tarefas para as implementações realizadas por exemplo seriam:

- Como auxílio para o desenvolvimento de um mecanismo de compactação incremental para o primeiro sistema, através da diminuição do tempo de pausa para compactação. Para isto, o escalonador deveria controlar a execução do processo de compactação para que a mesma pudesse ocorrer concorrentemente com a execução da aplicação.
- Como uma forma de reservar blocos plenamente disponíveis para a alocação realizada pelo segundo sistema. Deste modo, a tarefa realizada pelo escalonador consistiria em executar periodicamente uma análise dos blocos armazenados na lista de blocos livres. Aqueles blocos que mantivessem referências ativas para outros objetos alocados na memória deveriam ser desvinculados destes objetos, devendo para isto serem decrementados os contadores de referências de cada objeto referenciado por um destes blocos.

Por fim, uma atividade que vem diretamente ao encontro dos interesses e trabalhos já desenvolvidos pelo grupo consiste no desenvolvimento de uma ferramenta para geração automática e otimizada de sistemas para gerência dinâmica de memória. Para isto, o passo inicial deve ser a construção de uma biblioteca de componentes em *software* e *hardware* que devem ser caracterizados quanto a área, desempenho, potência e energia. Em seguida, o núcleo inteligente da ferramenta deve ser capaz de selecionar os componentes a serem integrados com base nos requisitos da aplicação alvo. Considerando todos os custos envolvidos com a gerência dinâmica de memória, discutidos neste trabalho, o desenvolvimento desta ferramenta pode ser entendido como uma extensão importante para o ambiente Sashimi, já que permitiria a geração de subsistemas de gerência dinâmica de memória adequados aos requisitos específicos de cada aplicação.

REFERÊNCIAS

[AAS 2001] AAS, E. J. et al. An Implementation of an Embedded Microprocessor Core With support for Executing Byte Compiled Java Code. In: EUROMICRO SYMPOSIUM ON DIGITAL SYSTEMS DESIGN, DSD, 2001, Warsaw. **Proceedings…** [S.l.]: IEEE Computer Society, 2001. p. 396-399

[APP 97] APPEL, A. W. **Modern Compiler Implementation in Java:** Basic Techniques. Cambridge: Cambridge University Press, 1997. 538 p.

[BAC 2003] BACON, D. F.; RAJAN, V. T.; CHENG, P. A Real-time Garbage Collector with Low Overhead and Consistent Utilization. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 30., 2003, New Orleans . **Proceedings…** New York: ACM Press, 2003. p. 285-298.

[BAK 93] BAKER, H. G. Minimizing Reference Count Updating with Deferred and Anchored Pointers for Functional Data Structures. **ACM SIGPLAN Notices**, New York, v. 29, n. 9, 1994.

[BEC 2004] BECK, A. C. S.; GOMES, V. F.; CARRO, L. A VHDL Implementation of a Low Power Pipelined Java Processor for Embedded Applications. In: IBERCHIP WORKSHOP, 10., 2004. **Proceedings…** [S.l.: s.n.], 2004.

[BEC 2004a] BECK, A. C. S.; CARRO, L. A VLIW Low Power Java Processor for Embedded Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 17., 2004, Pernambuco. **Proceedings…** New York: ACM Press, 2004. p. 157-162.

[BEC 2003] BECK, A. C. S.; WAGNER, F. R.; CARRO, L. CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 16., 2003. **Proceedings…** Washington: IEEE Computer Society, 2003. p. 349.

[BLA 2003] Blackburn, S. M.; McKinley, K. S. Ulterior Reference counting: Fast Garbage Collection without a Long Wait. **ACM SIGPLAN Notices**, New York, v.38, n.11, p. 344-358, Nov. 2003.

[BLE 2001] BLELLOCH, G. E; CHENG, P. A Parallel, Real-Time Garbage Collector. **SIGPLAN Notices**, New York, v.36, n.5, p. 125-136, May 2001.Trabalho apresentado na ACM SIGPLAN Conference on Programming Language Design and Implementation, 2001.

[BOL 2000] BOLLELLA, G. et al. **The Real-Time Specification for Java.** The Java Series. Addison-Welsey, 2000.

[DAH 2005] DAHM, M.; ZYL, J. V.; HAASE, E. **The Byte Code Engineering Library (BCEL)**. Disponível em: <http://jakarta.apache.org/bcel> Acesso em:14 jun. 2005.

[DAR 2003] DA ROSA, L. S. Scheduling policy costs on a Java microcontroller. In: WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, JTRES, 2003, Catania. **Proceedings…** [S.l.: s.n.], 2003. p 520-533.

[CHE 2002] CHEN, G. et al. Tuning Garbage Collection for Reducing Memory System Energy in an Embedded Java Environment. **ACM Transactions on Embedded Computing Systems (TECS)**, New York, v. 1, n. 1, p. 27–55, Nov. 2002.

[CHE 2002a] CHEN, G.; et al. Tuning Garbage Collection in an Embedded Java Environment. In: INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, HPCA, 8., 2002. **Proceedings…** Washington: IEEE Computer Society, 2002. p. 92.

[COR 2003] CORSARO, A.; Cytron, R. K. Efficient Memory-Reference Checks for Real-time Java. In: ACM SIGPLAN CONFERENCE ON LANGUAGE, COMPILER, AND TOOL FOR EMBEDDED SYSTEMS, 5., San Diego. **Proceedings...** New York: ACM Press, 2003. p. 51 - 58.

[DOW 2005] DOWNLOAD.COM. **DJ Java Decompiler**. Disponível em: <http://members.fortunecity.com/neshkov/dj.html>. Acesso em: 14 jun. 2005.

[DEL 2002] DE LA LUZ, V.; KANDEMIR, M.; KOLCU, I. Automatic Data Migration for Reducing Energy Consumption in Multi-Bank Memory Systems. In: DESIGN AUTOMATION CONFERENCE, DAC, 39., **Proceedings…** New York: ACM, 2002. p. 213-218.

[DEU 76] DEUTCH, L. P.; BOBROW, D. G. An Efficient, Incremental**,** Automatic Garbage Collector. **Communications of the ACM**, New York, v. 19 , n. 9, p. 522-526, Sept. 1976.

[DET 2004] DETERS, M. et al. Automated Reference Counted Object Recycling for Real-Time Java. In: IEEE REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM, RTAS, 10., 2004. **Proceedings…** Washington: IEEE Computer Society, 2004. p. 424.

[DIA 2000] DIAZ-HERRERA, J. L.; MADISETTI, V. K. Embedded Systems Product Lines. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 22., 2000., **Proceedings…** [S.l.: s.n.], 2004. p. 90-97.

[DON 2001] DONAHUE, S. M. et al. Storage Allocation for Real-Time, Embedded Systems. In: INTERNATIONAL WORKSHOP ON EMBEDDED SOFTWARE, 1., 2001, Berkeley. **Proceedings…** London: Springer-Verlag, 2001. p. 131-147.

[DYK 2002] DYKSTRA, L.; SRISA-AN, W.; CHANG J. M. An analysis of the Garbage Collection Performance in Sun's HotSpot™ Java Virtual Machine. In: IEEE INTERNATIONAL PERFORMANCE, COMPUTING, AND COMMUNICATIONS CONFERENCE, IPCCC, 21., 2002, Phoenix. **Proceedings…** [S.l.: s.n.], 2002. p. 335-39.

[FON 2002] FONG, A., LI, R. C. L. Dynamic Memory Allocation/Deallocation Behavior in Java Programs. In: IEEE TENCON, 2002, Beijing. **Proceedings…** [S.l.: s.n.], 2002. p. 314-317.

[FUH 2004] FUHRMANN, S.; et al. Real-time garbage collection for a multithreaded Java microcontroller. **Real-Time Systems**, Norwell, v. 26, n. 1, p. 89-106, Jan. 2004.

[GEH 93] GEHRINGER, E. F.;CHANG, Ellis. **Hardware-Assisted Memory Management**. In: OOPSLA: WORKSHOP ON MEMORY MANAGEMENT AND GARBAGE COLLECTION, 1993. Disponível em:<http://citeseer.ist.psu.edu/339009.html> Acesso em: 24 jun. 2005.

[HAD 67] HADDON, B. K.; WAITE, W. M. A compaction procedure for variable length storage elements. **Computer Journal**, [S.l.], v. 10, p. 162-165, Aug. 1967.

[HAN 2002] HANSEN, L. T.; CLINGER, W. D. An Experimental Study of Renewal-Older-First Garbage Collection. **SIGPLAN Notices**, New York, v.37, n.9, p.247-258, Sept.2002. Trabalho apresentado na 7. International Conference on Functional Programming, 2002.

[HAR 2000] HARDIN, D. S. The Virtual Machinist. **IEEE Instrumentation and Measurement Magazine**, New York, June 2000.

[HEN 98] HENRIKSSON, R. **Scheduling Garbage Collection in Embedded Systems**. 1998. PhD thesis, Lund Institute of Technology.

[HEN 2003] HENRIKSSON, R.; ROBERTZ, S. G. Time-Triggered Garbage Collection Robust and Adaptive Real-Time GC Scheduling for Embedded Systems. In: CONFERENCE ON LANGUAGE, COMPILER, AND TOOL FOR EMBEDDED SYSTEMS, 2003, San Diego. **Proceedings…** New York: ACM Press, 2003. p. 93-102.

[HUD 2005] Huddle, C. V. **Standard Performance Evaluation Corporation**. Dísponível em: <http://www.spec.org.osg/jvm98>. Acesso em: 20 jun. 2005.

[ITO 2001] ITO, S. A.; CARRO, L.; JACOBI R. Making Java Work for Microcontroller Applications. ***IEEE Design & Test***, [S.l.], v. 18, n. 5, p. 100-110, Sept.-Oct. 2001.

[ITO 2000] ITO, S. A. **Projeto de Aplicações Específicas com Microcontroladores Java Dedicados**. 2000. 84f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[ITO 99] ITO, S. A. et al. Implementação de Uma Máquina Java. In: WORKSHOP IBERCHIP, 5., 1999, Lima. **Memórias...** Lima: Hozlo S.R.L., 1999. p. 252-260.

[IVA 99] IVANOVIC, V.; MAHAR, M. **Using Java in Embedded Systems**. Disponível em: <http://img.cmpnet.com/edtn/ccellar/e034pdf1.pdf> Acesso em: 20 jun. 2005.

[JON 96] JONES, R.; LINS, R. D. **Garbage Collection:** algorithms for automatic dynamic memory management. Chichester: John Wiley, 1996. 377 p.

[KIM 99] KIM, T.et al. Scheduling Garbage Collector for Embedded Real-Time Systems. **ACM SIGPLAN Notices**, New York, v. 34, n. 7, p. 55 - 64, July 1999.

[KIM 2000] KIM, J.; HSU, Y. Memory System Behavior of Java Programs: Methodology and Analysis. In: INTERNATIONAL CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS, 2000, Santa Clara. **Proceedings...** New York: ACM Press, 2000. p. 264-274.

[KRA 2002] KRAPF, R. C.; CARRO, L. Signal processing applications for embedded java systems In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 15., 2002. **Proceedings…** [S.l.: s.n.], 2002. p. 209-213.

[KRE 99] KREUZINGER, J. The Komodo Project: Thread-based Event Handling Supported by a Multithreaded Java Microcontroller. In: EUROMICRO CONFERENCE, 25., 1999, Milano. **Proceedings…** [S.l.: s.n.], 1999. p. 122-128.

[LEE 2003] LEEMAN, M. et al. Methodology for refinement and optimization of dynamic memory management for embedded systems in multimedia applications. In: IEEE WORKSHOP ON SIGNAL PROCESSING SYSTEMS, SIPS, 2003. **Proceedings…** [S.l.: s.n.], 2003. p. 369-374.

[LEE 2004] LEE, H. B.; ZORN, B. G. **BIT:** Bytecode Instrumenting Tool. Disponível em: < http://www.cs.colorado.edu/~hanlee/BIT/ >. Acesso em: 03 mar. 2004.

[LEA 99] LEAR, A. C. Shedding Light on Embedded Systems. **IEEE Software**, [S.l.], v. 16, n. 1, p. 122-125, Jan. /Feb. 1999.

[LEW 2000] LEWIS, O.; MANNION, M.; BUCHANAN, W. J. Performance Issues of Variability Design for Embedded System Product Lines. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 22., 2002, Limerick. **Proceedings…** [S.l.: s.n.], 2000.

[LIN 2001] LINDWER, M. **Java in Embedded Systems.** Disponível em: <http://home.iae.nl/users/lindwer/XOOTIC/Java_in_embedded.XOOTIC.final.pdf.> Acesso em: 20 jun. 2005.

[LIN 2000] LIN, C.; CHEN, T. Dynamic memory management for real-time embedded Java chips. In: INTERNATIONAL CONFERENCE ON REAL-TIME COMPUTING SYSTEMS AND APPLICATIONS, 7., 2000. **Proceedings…** [S.l.: s.n.], 2000. p. 49-56.

[MAT 2004] MATTOS, J. C. B. et al. Design Space Exploration with Automatic Selection of SW and HW for Embedded Applications. In: INTERNATIONAL WORKSHOP ON SYSTEMS, ARCHITECTURES, MODELING, AND SIMULATION, SAMOS, 2004, Samos. **Proceedings…** [S.I.]: Springer-Verlag, 2004.

[MIL 2005] MILLBERG, M. **An Efficient Dynamic Memory Manager for Embedded Systems**. Disponível em: <http://www.ifip.or.at/con2000/icda2000/icda-14-2.pdf.> Acesso em: 20 jun. 2005.

[PER 2000] PERRIER, V. Adapting Java for Embedded Development. **IEEE Instrumentation and Measurement Magazine**. New York, May 2000.

[PER 2000a] PERSSON, P. **Predicting Time and Memory Demands of Object-Oriented Programs**. Disponível em: <www.control.lth.se/articles/article.pike?action=fulltext&artkey=per00lic> Acesso em: 20 jun. 2005.

[PET 77] PETERSON, J. L.; NORMAN, T. A. Buddy systems. **Communications of the ACM**, New York, v. 20, n. 6, p. 421-431, June 1977.

[RAB 96] RABAEY, J. M. **Digital Integrated Circuits:** *A Design Perspective*. Upper Saddle River: Prentice Hall, 1996. 702 p.

[RIT 2001] RITZAU, T. Hard Real-Time *Reference counting* without External Fragmentation. In: JAVA OPTIMIZATION STRATEGIES FOR EMBEDDED SYSTEMS WORKSHOP AT ETAPS, 2001. **Proceedings…** [S.I.: s.n.], 2001.

[SEL 95] SELIGMAN, J.; GRARUP, S. Incremental Mature Garbage Collection Using the Train Algorithm. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 9., 1995. **Proceedings…** London: Springer-Verlag, 1995. p. 235-252.

[SIE 2000] SIEBERT, F. Eliminating External Fragmentation in a Non-Moving Garbage Collector for Java. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURE, AND SYNTHESIS FOR EMBEDDED SYSTEMS, 2000, San Jose. **Proceedings...** New York: ACM Press, 2000. p. 9-17.

[SOK 2005] SOKOBAN Homepage. Disponível em: <http://www.cs.ualberta.ca/ ~games/> Acesso em: 18 mar. 2005.

[Sou 2005] SOULIÉ, J. **Complete C++ language tutorial**. Disponível em: <www.cplusplus.com/doc/tutorial/> Acesso em: 21 jun. 2005.

[SRI 2003] SRISA-AN, W.; DAN LO, C.; CHANG, M. Active Memory Processor: A Hardware Garbage Collector for Real-Time Java Embedded Devices. **IEEE Transactions on Mobile Computing**, [S.l.], v. 2, n. 2, p. 89-101, 2003.

[SUN 2005] SUN MICROSYSTEMS**. Sun Developer Network**. Disponível em: <http://java.sun.com/> Acesso em: 20 jun. 2005.

[SUN 2005a] SUN MICROSYSTEMS. **J2ME Building Blocks for Mobile Devices.** Disponível em: <http://java.sun.com/products/cldc/wp/ KVMwp.pdf>. Acesso em: 20 jun. 2005.

[SUN 99] SUN MICROSYSTEMS**. PicoJava-II Programmer's Reference Manual**. [S.l.], 1999. 512 p.

[TYS 2005] TYSON, J.; FREUDENRICH, C. **How MP3 Players Work**. Disponível em: <http://www.howstuffworks.com/mp3-player.htm> Acesso em: 18 mar. 2005.

[YAH 2005] YaHOO GEOCITIES. **Avajii - a Java bytecode viewer**. Disponível em: < http://www.geocities.com/nilzone/ .> Acesso em: 18 mar. 2005.

[YEL 99] YELLIN, F. ; LINDHOLM, T. **The Java™ Language Specification (Java Series)**. 2nd ed. [S.l.]: Addison-Wesley, 1999.

[VAL 2005] VALLEE-RAI R. et al. **Soot: a Java Optimization Framework**. Disponível em: <http://www.sable.mcgill.ca/soot/#credits> Acesso em: 14 jun. 2005.

[WEI 63] WEIZENBAUM, J. Programming Languages: Symmetric List Processor. **Communications of the ACM**, New York, v. 9 , n. 2, p. 524- 536, Feb. 1963.

[WIK 2005] WIKIPEDIA. **Reference counting**. Disponível em: <http://en.wikipedia.org/wiki/Reference_counting> Acesso em: 21 jun. 2005.

[WIS 97] WISE, D. et al. Research Demonstration of a Hardware Reference-Counting Heap. **Lisp and Symbolic Computation**, Hingham, v. 10, n. 2, p.151-181, July 1997.

ANEXO A DETALHAMENTO DE CUSTOS MP3 PLAYER

Tabela A1.1: Ciclos e energia para o 2º frame

SGDM	Tam. Bloco (bytes)	Arq.	Fator	Total Ciclos	Total Energia (nJ)	SGDM Ciclos	SGDM Energia (nJ)
1	-	Mult.	1,5	616640318	114412446	602124116	110861611
			1,75		133479307		129336666
			2		152545340		147810968
1	-	Pipe	-	112733510	76272670	110243183	73905484
2	16	Mult.	1,5	822832882	153696969	808316680	150146134
			1,75		179310712		175168071
			2		204923011		200188638
2	16	Pipe	-	149696402	102461505	147206075	100094319
2	32	Mult.	1,5	641894534	122388191	627378332	118837356
			1,75		142784279		138641638
			2		163179191		158444819
2	32	Pipe	-	118501293	81589596	116010966	79222410
2	64	Mult.	1,5	571657880	110263118	557141678	106712283
			1,75		128638552		124495911
			2		147012908		142278535
2	64	Pipe	-	106303600	73506454	103813273	71139268

Tabela A1.2: Ciclos e energia para o 3º frame

SGDM	Tam. Bloco (bytes)	Arq.	Fator	Total Ciclos	Total Energia (nJ)	SGDM Ciclos	SGDM Energia (nJ)
1	-	Mult.	1,5	644040645	120457097	627659234	116461367
			1,75		140531356		135869680
			2		160604568		155277019
1	-	Pipe	-	118300405	80302284	115448897	77638509
2	16	Mult.	1,5	914756476	170883750	898375065	166888020
			1,75		199361797		194700121
			2		227838118		222510569
2	16	Pipe	-	166879202	113919059	164027694	111255284
2	32	Mult.	1,5	729387989	138968481	713006578	134972751
			1,75		162127736		157466060
			2		185285597		179958047
2	32	Pipe	-	134824467	92642798	131972959	89979024
2	64	Mult.	1,5	656677059	126470123	640295648	122474393
			1,75		147546489		142884813
			2		168621586		163294037
2	64	Pipe	-	122150491	84310793	119298983	81647018

Tabela A1.3: Ciclos e energia para o 4º frame

SGDM	Tam. Bloco (bytes)	Arq.	Fator	Total Ciclos	Total Energia (nJ)	SGDM Ciclos	SGDM Energia (nJ)
1	-	Mult.	1,5	643132723	120232494	626752548	116235764
			1,75		140269320		135606478
			2		160305101		154976219
1	-	Pipe	-	118154598	80152551	115305499	77488110
2	16	Mult.	1,5	915815252	171321467	899435077	167324738
			1,75		199872346		195209505
			2		228421604		223092722
2	16	Pipe	-	166935018	114210802	164085919	111546361
2	32	Mult.	1,5	730011878	139188589	713631703	135191860
			1,75		162384475		157721634
			2		185579013		180250131
2	32	Pipe	-	134890602	92789506	132041503	90125065
2	64	Mult.	1,5	657144407	126605103	640764232	122608374
			1,75		147703939		143041098
			2		168801529		163472647
2	64	Pipe	-	122223603	84400764	119374504	81736323

ANEXO B CÓDIGO DESENVOLVIDO

A tab. A.1 apresenta o conjunto de arquivos de código desenvolvidos para este trabalho. A primeira coluna da tab. 1, contém um código que identifica a ferramenta a qual o arquivo está incorporado.

Tabela A2.1: Arquivos de código desenvolvidos por este trabalho

Id	Arquivo/Código	Nº de linhas	Descrição
01	AjustaFor.java	275	Conjunto de arquivos que formam a ferramenta de adaptação de código descrita neste trabalho.
01	AjustaIf.java	187	
01	AjustaMet.java	188	
01	AjustaWhile.java	214	
01	Debug.java	135	
01	DoAsBIT2.java	2521	
01	MakeTable.java	647	
01	PassoAPasso.java	1422	
01	TabFieldCt.java	74	
01	TabFieldEl.java	32	
01	TabMetCt.java	81	
01	TabMetEl.java	31	
01	TabVarCt.java	273	
01	TabVarEl.java	42	
01	MyClass2.java	1242	
02	CompFiles.java	95	Ferramenta para comparação de arquivos em formato texto. Fornece como dado de saída uma listagem das linhas onde foram encontradas diferenças entre os arquivos comparados.
03	FormatArray2.java	240	Ferramenta para adaptação de *arrays* de formato de inicialização estática para dinâmica.
04	CalcCodSize.java	96	Ferramenta que permite calcular o tamanho em bytes de um código java
05	JadToJava.java	45	Ferramenta que permite modificar a extensão de todos os arquivos em um diretório e subdiretórios de .jad para .java
06	HexToDec.java	70	Ferramenta que gera para todos os métodos de uma aplicação os endereços iniciais e finais do PC (*Program Counter*)

Id	Arquivo/Código	Nº de linhas	Descrição
06	MakeCacoDat.java	406	criados durante a geração do arquivo rom.mif através da ferramenta SASHIMI. A ferramenta ainda realiza a formatação dos intervalos gerados e os incorpora ao código C do simulador CACO-PS.
07	Profile.java	309	Biblioteca de métodos para geração das estimativas quanto ao uso de memória apresentadas neste trabalho.
08	Statistics.java	800	Ferramenta para geração estimativas gerais sobre uma aplicação. Entre as estimativas geradas, estão: número e nome dos métodos executados pela aplicação, total de instruções executadas, total de instruções executadas por método e número de chamadas a métodos aninhadas.
08	ToStatistics.java	102	
08	ToCountMetAninh.java	362	
08	ToSuport.java	706	
09	RemPackMp3.bat	114	Conjunto de arquivos para processamento em lotes, desenvolvidos para auxiliar o processo de geração de estimativas em ciclos, energia e potência apresentados neste trabalho.
09	compile.bat	825	
09	copyJavaToS2.bat	88	
09	copyMEMs.bat	240	
09	copy_fjas.bat	231	
09	copy_read_arq.bat	231	
09	copyBats.bat	231	
09	copyMakCaco.bat	231	
09	openSimuls.bat	240	
10	CallThem.c	136	Ferramenta que realiza a remoção de referências a pacotes nos arquivos .java da aplicação, permitindo executá-la a partir de qualquer diretório do sistema de arquivos
10	RemPack.java	112	
11	GetClassFiles.java	116	Ferramenta que atua de forma integrada ao decompilador *avajii* [YAH 2005] permitindo a visualização e gravação em disco do conteúdo em formato texto da constant pool correspondente a cada *.class* da aplicação