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

LEANDRO AUGUSTO DE OLIVEIRA

Conjunto de Classes para Aplicações Gráficas 2D em Sistemas Embarcados Baseados no Femtojava

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

Prof. Dr. Luigi Carro
Orientador

Porto Alegre, agosto de 2006.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Oliveira, Leandro Augusto de

Conjunto de Classes para Aplicações Gráficas 2D em Sistemas Embarcados Baseados no Femtojava / Leandro Augusto de Oliveira – Porto Alegre: Programa de Pós-Graduação em Computação, 2006.

62 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2006. Orientador: Luigi Carro.

1. Biblioteca Gráfica. 2. Sistemas Embarcados 3. Java. 4. Femtojava. I. Carro, Luigi. II. Título.

AGRADECIMENTOS

Em primeiro lugar, quero agradecer à minha mãe, que sempre me deu força em todos os momentos e me ensinou muito, às vezes até sem se dar conta.

Também quero agradecer à minha noiva, Gislaine, que me incentiva, apóia, critica (também é preciso) e está sempre comigo, mesmo quando está longe. Sem ela, nada disso teria razão de ser.

Não posso esquecer dos meus amigos, Sandro, Marne, Émerson, Diego, que sempre estão aí, em todos os momentos sei que posso contar com eles. Bom, nem sempre o Sandro pode. Às vezes ele está entregando gás. Outras vezes ele some. Mas quando ele não tem mais nada pra fazer, aí de vez em quando eu posso contar com ele.

Os meus amigos do Xing Yi, Bruno, grande professor, Alex, Douglas e tantos outros. Aprendi muito nos treinos, e quem participa sabe que não estou falando somente da arte, mas também de companheirismo, solidariedade, e tantas coisas que com certeza eu não aprenderia na frente de um computador.

Os colegas e amigos do mestrado, Arnaldo, Juliano, Márcio, Victor e todos os outros com quem pude conversar sobre os temas mais diversos, de microeletrônica a Sherman's Lagoon, tornando os dias menos cansativos e fazendo do ambiente de trabalho um lugar mais agradável. Quero agradecer especialmente ao colega Bruno Neves, que foi de grande ajuda neste trabalho.

Também gostaria de agradecer aos professores do Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande do Sul por seu empenho em transmitir conhecimento e formar profissionais. Um agradecimento especial ao meu orientador, Prof. Dr. Luigi Carro, que tornou este trabalho possível.

Todos contribuíram de alguma forma para este trabalho, seja com críticas, incentivo ou mesmo só conversando para clarear as idéias. A todos que foram citados, e até alguns que não foram, os meus mais sinceros agradecimentos.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 7

LISTA DE FIGURAS 9

LISTA DE TABELAS 11

RESUMO 13

ABSTRACT 15

1 INTRODUÇÃO 17

2 REVISÃO BIBLIOGRÁFICA 21

2.1 Técnicas de hardware para reduzir potência 22
2.2 Consumo de potência da memória 23
2.3 Técnicas em software para reduzir potência 24
2.4 Consumo de energia de sistemas com display gráfico 26
2.5 Compressão do Frame Buffer 28
2.5.1 Compressão RLE 29
2.5.2 Compressão RLE Adaptativa do Frame Buffer 29
2.5.3 Codificação Huffman 31
2.5.4 Compressão Huffman Diferencial do Frame Buffer 32
2.6 Organização otimizada da memória do display 35
2.7 Otimizações no barramento com o display 36
2.8 Manipulação de parâmetros dos displays LCD 37
2.9 Crítica e contextualização do trabalho 39

3 BIBLIOTECA GRÁFICA PARA O FEMTOJAVA 41

3.1 Java em sistemas embarcados 41
3.2 Plataforma Femtojava 42
3.3 Suporte a Objetos no Femtojava 43
3.4 Biblioteca Não-Otimizada 44
3.5 Biblioteca Otimizada para Economia de Energia 46
3.5.1 Compressão RLE 46
3.5.2 Operações com operandos imediatos 47

4 RESULTADOS 49

4.1 Ambiente de simulação 49

4.2 Tamanho do código desenvolvido 51
4.3 Análise dos resultados 52
5 CONCLUSÃO 55
REFERÊNCIAS 57

LISTA DE ABREVIATURAS E SIGLAS

ACPI	Advanced Configuration Power Interface
API	Application Program Interface
CACO-PS	Cycle-Accurate Configurable Power Simulator
CAT	Code Adaptation Tool
CDC	Connected Device Configuration
CLDC	Connected Limited Device Configuration
CRT	Cathode Ray Tube
DMML	Dynamic Memory Management Library
DSP	Digital Signal Processing
DVI	Digital Visual Interface
J2ME	Java 2 Micro Edition
JVM	Java Virtual Machine
LCD	Liquid Crystal Display
LIWT	Limited Intra-Word Transition
LRFU	Least Recently and Frequently Used
MIDP	Mobile Information Device Profile
OLED	Organic Light Emitting Diode
PDA	Personal Digital Assistent
RAM	Random Access Memory
RGB	Red Green Blue
RLE	Run-Length Encoding
ROM	Read-Only Memory
SASHIMI	System As Software and Hardware In Microcontrollers
TDMS	Transition-Minimized Differential Signaling

LISTA DE FIGURAS

Figura 2.1: Estrutura de um gerenciador de energia em nível de sistema..... 22
Figura 2.2: Organização de sistema com display 27
Figura 2.3: Compressão RLE 29
Figura 2.4: *Flag* de compressão..... 30
Figura 2.5: Divisão da tela em regiões, com um dirty flag para cada região 30
Figura 2.6: Exemplo de construção de um código de Huffman 32
Figura 2.7: Exemplo de codificação usando diferenças entre cores adjacentes na tela .. 32
Figura 2.8: Organização da tabela de código 34
Figura 2.9: Reduzindo o tamanho da tabela através da simetria das diferenças 34
Figura 3.1: Modelos de execução de programas Java [NEV 2005]..... 42
Figura 3.2 – Fluxo de projeto com a ferramenta Sashimi e suporte a objetos 43
Figura 3.3 – Fluxo de desenvolvimento com a biblioteca FemtoGraphics..... 45
Figura 3.4 – Imagem com região trasparente representada pela cor preta..... 45
Figura 3.5: Instruções no código original e no código em linha 48
Figura 4.1: Ambiente de simulação 50
Figura 4.2: Topologia do sistema na simulação 51

LISTA DE TABELAS

Tabela 2.1: Comparação entre os métodos RLE, Huffman diferencial e Huffman diferencial com código de tamanho limitado (% do tamanho original) [SHI 2005] 33
Tabela 4.1: Tamanho do código desenvolvido. 51
Tabela 4.2: Tamanho do código objeto (só Resources) 52
Tabela 4.3: Resultados da aplicação Sokoban com o uso da biblioteca de gerência de memória em software 52
Tabela 4.4: Resultados da aplicação Aquário com o uso da biblioteca de gerência de memória em software 52
Tabela 4.5: Resultados da aplicação Sokoban supondo custo zero na gerência dinâmica de memória 53
Tabela 4.6: Resultados da aplicação Aquário supondo custo zero na gerência dinâmica de memória 53

RESUMO

Com o crescimento do mercado de sistemas embarcados, em especial aqueles dispositivos portáteis como PDAs e celulares, observa-se o crescimento de mercados baseados nestas plataformas, como o mercado de entretenimento digital. Devido às características destes dispositivos, novas oportunidades e desafios acompanham estas mudanças. Com cada vez mais recursos incorporados, o projeto de software para estes dispositivos torna-se mais complexo, exigindo soluções que aumentem a produtividade do desenvolvedor.

Este trabalho descreve o estudo de técnicas em software para reduzir o consumo de energia e aumentar o desempenho de sistemas embarcados com recursos gráficos, baseados no microprocessador Femtojava. Como subproduto deste trabalho, foi desenvolvida uma biblioteca gráfica para a plataforma Femtojava. Dois estudos de caso foram desenvolvidos para analisar a biblioteca desenvolvida, caracterizando o seu consumo de energia e desempenho.

Palavras-Chave: Biblioteca gráfica, sistemas embarcados, Femtojava.

Class Library for Femtojava-Based Embedded 2D Graphics Applications

ABSTRACT

With the growth of the embedded systems market, especially PDAs and mobile phones, other markets based on those platforms, like digital entertainment, have experienced growth as well. Due to its characteristics, embedded devices present new opportunities and challenges. With an ever growing number of features, software development for these devices becomes more complex, demanding more powerful tools for increasing developer productivity.

This work presents the study of software techniques to save power and improve performance of graphics capable embedded devices, based on Femtojava microprocessor. As a sub product of this work, a graphical library has been developed for the Femtojava platform. Two case studies were developed in order to analyze the library, characterizing its power consumption and performance.

Keywords: Class Library, Embedded Systems, Java, Femtojava.

1 INTRODUÇÃO

Com a evolução da tecnologia, cresce o uso de sistemas computacionais para auxiliar nas tarefas do nosso cotidiano. Vêem-se dispositivos eletrônicos com diversas finalidades, como entretenimento, comunicação, controle inteligente de dispositivos mecânicos, etc. Estes sistemas são denominados sistemas computacionais embarcados [WOL 2002].

Com o crescimento da demanda por sistemas eletrônicos embarcados e acirrada competição neste mercado, surge a necessidade de reduzir o tempo de projeto desses sistemas. Os fabricantes desses dispositivos precisam entregar um produto de qualidade que atenda as necessidades do consumidor, e devem fazê-lo no menor tempo possível. Existem diversas questões comerciais envolvidas que reduzem consideravelmente o ciclo de vida desses produtos.

No entanto, os sistemas eletrônicos embarcados estão ficando mais e mais complexos. Para que o consumidor adquira um novo produto, substituindo um outro que executa a mesma tarefa, é necessário que este tenha novos recursos que justifiquem sua aquisição. Recursos adicionais tornarão o sistema mais complexo e, portanto, mais difícil de projetar. E isto terá um impacto importante no tempo de desenvolvimento do mesmo.

Buscando amenizar estes problemas, novas metodologias de projeto estão sendo propostas no meio acadêmico, visando reduzir o tempo do projeto. Uma constante nessas propostas é a necessidade de especificar os sistemas em níveis mais altos de abstração. Para que a especificação em alto nível seja transformada em um sistema de hardware, ferramentas de automatização de projeto (Electronic Design Automation) se tornam necessárias. Essas ferramentas estão se tornando cada vez mais complexas, exigindo muito tempo de pesquisa e desenvolvimento.

Outra forma de reduzir o tempo de projeto é a utilização de núcleos de hardware pré-projetados e pré-verificados, que possam ser facilmente integrados ao sistema. A questão da facilidade de integração exige que estes núcleos atendam a uma especificação padronizada para comunicação com outros núcleos. Um exemplo mais prático disto é o projeto baseado em plataforma [SAN 2001], onde a etapa de integração dos núcleos já foi realizada pelos projetistas da plataforma.

Outra questão importante no desenvolvimento de sistemas embarcados é o software que executa nestes sistemas. Com a crescente quantidade de recursos que estes sistemas oferecem, o software aumentou consideravelmente de complexidade, tornando-se o

principal fator nos custos de implementação de sistemas embarcados. O tempo de desenvolvimento do software embarcado em muitos projetos supera o hardware, o que tornou as ferramentas de software para sistemas embarcados ainda mais importantes no projeto.

A plataforma Java [SUN 2005b] apresenta características importantes para o desenvolvimento de sistemas. A biblioteca padrão da linguagem Java, junto com outras bibliotecas, contemplam vários domínios de aplicações, agilizando o desenvolvimento de aplicações. Existem várias ferramentas do estado da arte para o desenvolvimento de aplicações Java, como ferramentas de *refactoring*, automação de compilação, suporte ao paradigma de orientação a aspectos, entre outras. A linguagem Java conta com o recurso de gerência automática de memória, que simplifica o desenvolvimento de software por aliviar a necessidade de gerência manual de recursos.

A questão da portabilidade também é importante. Existem máquinas virtuais Java para as mais diversas plataformas [SUN 2005b], que incluem a biblioteca padrão, o que reduz consideravelmente o custo de desenvolvimento multiplataforma. Embora outras linguagens apresentem esta característica, Java tem a vantagem de apresentar bom desempenho em relação a outras linguagens que também utilizam uma máquina virtual por se tratar de uma linguagem com verificação estática (durante a compilação) de tipos, simplificando o processo de execução dos bytecodes (instruções da máquina virtual). O aumento na produtividade do programador é essencial para o desenvolvimento de sistemas hoje, sejam eles embarcados ou para desktop. Como o código compilado é portável, só é necessário distribuir uma versão da aplicação, que poderá ser utilizada em diversas plataformas de hardware diferentes.

Embora Java seja mais eficiente que linguagens com verificação dinâmica de tipos, o desempenho das aplicações Java é inferior ao desempenho de aplicações compiladas para código nativo. Isto se deve à camada adicional de software, necessária para a execução dos programas Java, a Máquina Virtual Java (JVM). O código Java é compilado para um formato denominado de bytecode, que consiste em instruções de tamanho variável (1 a 3 bytes) para a máquina virtual. Nestas instruções, o primeiro byte corresponde à operação, e os seguintes aos parâmetros, quando estes existirem.

Uma abordagem para aumentar o desempenho do processo de interpretação dos bytecodes é compilar o código Java para o conjunto de instruções da arquitetura onde o programa será executado. Entretanto, isto elimina a vantagem da portabilidade do código compilado na forma de bytecode. Na prática, para manter a portabilidade, as máquinas virtuais implementam um processo de tradução dinâmica de código, transformando bytecodes em código nativo. Este processo é chamado de Just-In-Time [CRA 97]. A compilação JIT aumenta o desempenho do código, mas consome memória adicional para manter sua estrutura de funcionamento, onde se inclui o código traduzido para a arquitetura-alvo.

Outra solução para o problema do desempenho é a criação de processadores Java, que executam nativamente os bytecodes. Existem vários problemas relacionados à sua implementação, pois a especificação da plataforma Java foi criada para execução interpretada, utilizando uma plataforma que já disponha de sistema operacional. Isto complica a implementação destes processadores, pois a linguagem Java não facilita a implementação de software básico.

No intuito de avaliar o uso de Java para sistemas embarcados, [ITO 2001] descreve a implementação de um microcontrolador que executa nativamente bytecodes Java. Trata-

se de uma máquina de pilha com arquitetura Harvard, separando memória de instruções e memória de dados. A implementação original foi posteriormente melhorada em [BEC 2003], tanto em desempenho quanto em consumo de energia.

Outro inconveniente na utilização de Java em sistemas embarcados é a ausência de mecanismos de acesso a endereços arbitrários da memória ou outro mecanismo que permita comunicação com os periféricos do sistema embarcado, fazendo com que a implementação de controladores de dispositivo tenha que contornar esses problemas.

O microcontrolador Femtojava implementa um subconjunto dos bytecodes Java. Recursos como alocação de objetos e chamadas de métodos de objeto não estão presentes na implementação original. Por isso, o microcontrolador Femtojava é acompanhado de uma ferramenta que faz a adaptação dos arquivos de bytecodes (arquivos .class) para que estes utilizem apenas o subconjunto implementado. Esta ferramenta se chama Sashimi (System As Software and Hardware In Microcontrollers) [ITO 2001].

Por implementar um subconjunto da especificação Java, o microcontrolador Femtojava tem algumas limitações, como a falta de suporte a múltiplos fluxos de execução (*threads*). Antes do trabalho realizado em [NEV 2005], não havia suporte a alocação dinâmica de memória, o que dificultava o desenvolvimento de aplicações, visto que um esquema estático deveria ser desenvolvido.

Em [NEV 2005], o suporte a objetos foi implementado como uma biblioteca de software, com um conjunto de ferramentas associado. A implementação permite a alocação de objetos e realiza gerência dinâmica de memória. No entanto, o suporte a objeto ainda é limitado, não sendo possível trabalhar com herança e polimorfismo. E como a implementação é em software, o desempenho sofre um impacto considerável, pois várias instruções disparam um mecanismo caro de gerência de memória, incluindo simples acessos a arrays.

Embora a plataforma Java traga vantagens na sua utilização, somente a sua adoção não resolve os problemas de projeto de sistemas embarcados. Hoje em dia, desenvolver sistemas embarcados implica em desenvolver software [DIA 2000]. O software consome a maior parte do tempo de desenvolvimento destes sistemas. Além de lidar com a complexidade do projeto de software, o projetista ainda precisa considerar questões de desempenho e potência.

A plataforma de um sistema embarcado pode utilizar várias técnicas para economia de energia e para aumentar o desempenho. Porém, isto de nada adianta se as aplicações que rodam sobre esta plataforma não forem projetadas com estas questões em foco. Otimizações no software podem atingir ganhos substanciais em desempenho e potência e devem portanto ser utilizadas no projeto do software embarcado.

Devido à disseminação de sistemas embarcados dos mais diversos tipos, com um número cada vez maior de recursos, muitos sistemas embarcados atuais contam com algum sistema de display gráfico, em geral colorido. Um tipo de display muito utilizado em sistemas embarcados é o LCD (Liquid Crystal Display). Este display, junto à arquitetura utilizada para mantê-lo, é uma grade fonte de consumo de energia. Desta forma, torna-se importante o estudo de formas de economizar energia em sistemas que utilizam displays gráficos.

Existem muitos trabalhos que estudam formas de reduzir o consumo de energia dos sistemas gráficos, desde formas de controlar a emissão de luz do display e sua luz

traseira até formas de reduzir o tráfego nos barramentos do sistema gráfico. Em sua maioria, estes trabalhos tratam de otimizações implementadas em hardware. No entanto, uma abordagem em software também pode trazer ganhos substanciais em termos de desempenho e energia [LEE 95][WUY 96][TIW 94][KOL 96].

O presente trabalho é um estudo da implementação de recursos gráficos para o microcontrolador Femtojava. A abordagem utiliza apenas técnicas de software para aumentar o desempenho e reduzir o consumo de energia de aplicações gráficas, em especial jogos, muito comuns em sistemas embarcados que dispõem de um display. Para tanto, uma biblioteca de software para manipulação de imagens em duas dimensões foi desenvolvida visando execução no microcontrolador Femtojava. Fizeram parte do estudo técnicas para manipulação de imagens que permitem economia de energia e aumento de desempenho.

O objetivo principal deste trabalho é avaliar técnicas em software para reduzir o consumo de energia em sistemas com display gráfico. A biblioteca gráfica desenvolvida é um subproduto deste estudo, com o qual se espera dar mais um passo no sentido de transformar o Femtojava em uma plataforma para sistemas embarcados com recursos gráficos.

Este trabalho está dividido da seguinte forma: o capítulo 2 apresenta a revisão dos trabalhos relacionados ao problema de economia de energia em sistemas embarcados, desde as técnicas mais gerais, até as técnicas que dizem respeito a sistemas com recursos gráficos. O capítulo 3 descreve o ambiente de desenvolvimento do Femtojava, que serviu de base para este trabalho, junto com a descrição da biblioteca de classes para desenvolvimento de aplicações gráficas, que é o produto principal deste trabalho. O capítulo 4 apresenta os resultados obtidos com as técnicas implementadas na biblioteca. Por fim, o capítulo 5 apresenta as conclusões deste trabalho.

2 REVISÃO BIBLIOGRÁFICA

Hoje em dia se vê cada vez mais dispositivos embarcados fazendo parte do nosso dia-a-dia. Com o aumento da quantidade de recursos desses dispositivos ocorre também o aumento do seu consumo de energia. Muitas vezes estes dispositivos utilizam uma bateria como fonte de energia, o que se torna um problema, visto que a bateria tem tempo de carga útil limitado. Para aliviar este problema estudam-se técnicas para reduzir o consumo de energia dos sistemas embarcados.

Vários trabalhos apresentam técnicas para reduzir o consumo de potência. Por exemplo, em [LU 2001] e [BEN 2000] são demonstradas técnicas de economia de energia em nível de sistema. Em [SHI 99a] e [YAO 95] o consumo de energia do processador, considerando o escalonamento de processos, foi estudado. Em [LEE 99] é proposta uma técnica para reduzir o consumo de energia através da manipulação da tensão e freqüência do processador. Em [HIC 97] e [SHI 99b] o problema do consumo de potência nos diferentes níveis de memória foi estudado.

As técnicas de redução de potência podem ser divididas em estáticas e dinâmicas. As técnicas estáticas são aquelas aplicadas em tempo de projeto, como síntese e compilação visando baixo consumo de potência. As técnicas dinâmicas são aquelas que usam o comportamento de tempo de execução do sistema para economizar energia quando as unidades do sistema ficam ociosas ou tem pouco trabalho a fazer [LU 2001].

No momento que um subsistema fica ocioso, ele pode ser desligado para poupar energia. Quando este sistema for novamente requisitado ele será religado. Mas religar o dispositivo implica em um atraso e um custo em energia. Por esse motivo, gerenciar a potência de um sistema não é um problema trivial. É necessário determinar se o desligamento do dispositivo vai realmente economizar energia. As regras que determinam quando um dispositivo pode ser desligado são chamadas de políticas de gerência de energia.

O uso de políticas de gerência de energia afeta o desempenho do sistema. Se um dispositivo é desligado temporariamente, pelo menos uma requisição terá que esperar que ele seja novamente ativado. Uma forma de eliminar essa espera é tentar predizer quando o dispositivo será necessário novamente. No entanto, fazer este tipo de predição é difícil. Ativar o dispositivo muito cedo custa energia, e ativá-lo muito tarde causa perda de desempenho. Tratar estas questões é a responsabilidade de um gerenciador de energia em nível de sistema. A Figura 2.1, adaptada de [BEN 2000], mostra a estrutura de um gerenciador desse tipo.

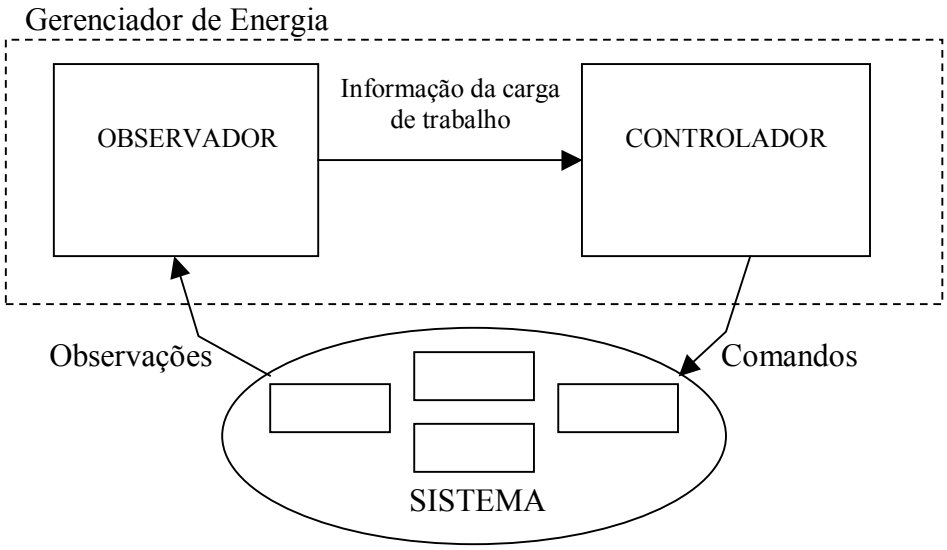

Figura 2.1: Estrutura de um gerenciador de energia em nível de sistema.

As políticas de gerência de energia podem ser divididas em três categorias, de acordo com o método utilizado para prever por quanto tempo o dispositivo pode ser desligado. A primeira categoria é *time-out*. Nesta, se o dispositivo está ocioso por um tempo t, assume-se que o dispositivo ainda continuará ocioso por tempo suficiente para justificar seu desligamento. O problema desta política é a energia perdida durante a espera pelo tempo de *time-out*.

Outra forma, já citada, é a política preditiva. Nesta, tenta-se prever por quanto tempo o dispositivo ficará ocioso, eliminando a espera pelo *time-out*. Se um período ocioso for identificado, através da previsão, como suficiente para justificar o desligamento do dispositivo, então o desligamento será feito. Outra forma de eliminar o *time-out* é modelando o problema com métodos estocásticos, definindo probabilidades para as transições de estado (de ocioso para ocupado e vice-versa). De acordo com as probabilidades o dispositivo será desligado ou não.

Para avaliar políticas de redução do consumo de energia, em [LU 2001] foi feita a implementação de várias políticas utilizando a interface ACPI (Advanced Configuration Power Interface) em um laptop utilizando ambiente Windows 2000. A interface ACPI permite a comunicação entre hardware e software para a gerência de energia a nível de sistema. Lu desenvolveu um tipo de controlador de dispositivo chamado de *filter driver*. Um *filter driver* é um controlador inserido entre um dispositivo e o seu controlador ou entre controladores. Desta forma é possível analisar as requisições ao dispositivo, tornando possível determinar quando este deve ser desligado.

Os experimentos em [LU 2001] mostram que a escolha da política de gerência de energia a nível de sistema é um compromisso entre economia de energia, desempenho, interatividade e utilização de recursos do sistema.

2.1 Técnicas de hardware para reduzir potência

Processadores para dispositivos móveis têm recursos para reduzir o consumo de energia. Por exemplo, no PowerPC 603, estes recursos são: desligamento dinâmico de unidades de execução ociosas, memórias *cache* de baixo consumo e considerações de

potência para standard cells, elementos da parte operativa e distribuição do clock. Além disso, o processador tem três modos de gerência de potência. Estes modos reduzem o consumo de energia quando o processador fica ocioso por um determinado período.

Ao dividir uma máquina de estados em várias sub-máquinas, é possível desligar seletivamente partes de um circuito, reduzindo o chaveamento [CHO 96]. Em [TIW 98] foi estudada uma técnica que consiste em desligar partes de um circuito que não são necessárias para uma determinada função em nível de ciclo. Desligar subsistemas dentro de um circuito é interessante porque a atividade de chaveamento dos transistores é a principal responsável pelo consumo de energia. Mas não é somente no desligamento de um subsistema que se economiza energia. Somente o cuidado de evitar chaveamentos desnecessários, mantendo o estado dos transistores já resulta em ganhos em energia.

Outra forma de economizar energia é manipulando a tensão de alimentação e a freqüência de operação do circuito dinamicamente. Esta abordagem é sugerida em [WEI 94]. Em [GOV 95] são estudadas formas de predição para políticas de ajuste dinâmico de desempenho.

O processador é um dispositivo de computação de propósito geral. Basicamente, troca-se eficiência por flexibilidade, de forma que o processador executa qualquer tipo de computação, porém não da forma mais otimizada. Isto tem um reflexo na potência e no desempenho do sistema. Transferir a computação para módulos otimizados para determinada tarefa traz ganhos em desempenho e potência. Em [HAV 99] é proposta uma técnica de redução de energia baseada na localidade de referência com módulos otimizados dedicados. A idéia consiste em transferir a carga de processamento da CPU para módulos programáveis colocados no próprio canal de fluxo dos dados.

Dispositivos móveis, como celulares, muitas vezes não têm qualquer processamento a fazer, não precisando nestes momentos operar com o máximo de desempenho. O *doze mode* é uma forma de economizar energia nestes sistemas. No *doze mode*, a freqüência de operação é reduzida e nenhum processo do usuário é executado. O sistema fica apenas esperando por mensagens. Quando uma mensagem é recebida, o estado normal é restabelecido. Um estudo feito em [NAI 99] mostra que a economia de energia utilizando este modo pode chegar a 98%.

Em [CHI 99] é demonstrado como explorar o fenômeno de recuperação da carga da bateria para economizar energia, prolongando a vida da bateria. O fenômeno que acontece em condições de descarga em pulso pode ser explorado para melhorar a capacidade de uma célula de energia.

2.2 Consumo de potência da memória

A potência consumida nos microprocessadores é um fator importante no consumo geral de potência de um sistema embarcado. Em especial, os acessos à memória são responsáveis por uma parcela importante deste consumo [SU 95][SHI 95]. Essa constatação evidencia a necessidade de estudo da hierarquia de memória, de forma a reduzir o consumo de energia do sistema também neste aspecto.

Existem vários estudos que consideram o desempenho do sistema como métrica principal da eficiência da hierarquia de memória do sistema. Tradicionalmente as caches são avaliadas pela sua taxa de acerto (hit rate), ignorando a questão do consumo de

energia. Em [HIC 97] diferentes políticas e tamanhos de caches são avaliados para determinar a melhor combinação visando economia de energia.

Hicks argumenta que quanto maior a taxa de acerto da cachê, maior é a economia de energia, pois o acesso à memória principal consome mais energia que o acesso à memória cache. Nos benchmarks utilizados, Hicks descobriu que caches de dados 2-way associative de 8KB e 4-way associative de 16KB apresentam menor consumo de energia, ambas utilizando blocos de 8 palavras. Foi verificado que a cache de 16KB apresenta pouca diferença em consumo de energia, mas tem desempenho melhor. Para a cache de instruções a análise não mostra resultados claros, mas Hicks argumenta que a cache de 16KB com mapeamento direto tem desempenho ótimo.

Em [SHI 99] é proposta uma estratégia para explorar o espaço de projeto na hierarquia de memória, usando como métricas de desempenho o tamanho da cache, o número de ciclos do processador e o consumo de energia. O foco de [SHI 99] é apenas na cache de dados pois em muitas aplicações embarcadas o volume de dados ultrapassa em muito o número de instruções. A estratégia desenvolvida em [SHI 99] é baseada no trabalho de [PAN 97], que trata da otimização da memória para sistemas embarcados utilizando como métricas de desempenho o número de ciclos no processador e o tamanho da cache de dados.

Encontrar o tamanho mínimo para a cache é interessante pois existe uma penalidade em área e potência proporcional ao tamanho da cache. Contrastando com os resultados apresentados em [HIC 97], em [SHI 99] é demonstrado que caches de tamanho reduzido resultam em menor consumo de energia pelo sistema.

Embora o uso de caches de tamanho menor resulte em menor consumo de energia, o mesmo não ocorre na questão da associatividade da cache. Caches com associatividade maior podem utilizar mais recursos, mas as taxas maiores de acerto compensam, reduzindo o consumo de energia. A situação melhora se os programas forem desenvolvidos explorando localidade espacial e temporal, de modo a utilizar melhor a cache.

A configuração de cache visando menor tempo de execução difere de forma significativa da configuração que visa economia de energia. Além disso, a configuração de cache ideal para um programa maior pode ser diferente da configuração para diferentes partes do programa, como foi constatado em [SHI 99], utilizando como aplicação um decodificador MPEG.

Tratar as questões de cache adequadamente não é a única forma de otimizar a utilização da memória. Também é possível reduzir o acesso à memória através de outras técnicas de hardware e software que reduzam o número de acessos à mesma. O restante deste capítulo mostra algumas técnicas que permitem reduzir os acessos e, com isso, reduzir a energia consumida pelo sistema.

2.3 Técnicas em software para reduzir potência

A partir do software é possível reduzir a potência consumida no sistema com diversas técnicas. O controle adequado do funcionamento do hardware pode resultar em economia de energia. Um exemplo disso é o controle do escalonamento de processos.

De forma geral, como visto na seção 2.1, existe dois métodos para reduzir o consumo de energia em processadores. A primeira forma é colocando o processador em

um estado de baixo consumo, onde só algumas partes ficam ligadas, como a geração de clock e os circuitos de temporizadores. Outra forma é reduzindo a tensão e a freqüência de operação do processador.

O escalonamento de processos pode ser feito visando reduzir o consumo de potência, se o escalonador do sistema operacional tiver controle sobre a freqüência de operação e a tensão do processador. Nos momentos em que o sistema estiver com pouco volume de trabalho, o escalonador pode reduzir a tensão e a freqüência, reduzindo assim o consumo de energia. Esta questão torna-se mais complexa em sistemas de tempo real, onde os *deadlines* devem ser atendidos. Se estes *deadlines* não forem atendidos, o sistema é considerado falho.

Um método de escalonamento visando reduzir o consumo de energia por meio do ajuste da freqüência de operação e da tensão de alimentação foi primeiramente proposto em [WEI 94] e posteriormente estendido em [GOV 95]. No método básico, o uso do processador a curto prazo é previsto com base no histórico de utilização. Com a previsão, o desempenho do processador é ajustado para o valor adequado. Mas a existência de atraso no caso da predição falhar impede o uso em sistemas de tempo real.

Em sistemas de tempo real foram propostos métodos de escalonamento estático, como em [YAO 95], [HON 98] e [ISH 98]. A modelo base da abordagem destes métodos é um conjunto de tarefas com um período único. Quando os períodos das tarefas variam, que é o modelo normal em projeto de sistemas de tempo real, pode-se transformar o problema usando o menor múltiplo comum dos períodos das tarefas como o tamanho do período único. As tarefas são escalonadas dentro deste período. Mas este modelo de escalonamento tem o inconveniente de necessitar de grande quantidade de memória para armazenar o escalonamento, que foi calculado estaticamente. Isso é um problema devido à limitação de memória dos sistemas embarcados. Outro problema é que o escalonamento é calculado para um tempo fixo de execução das tarefas, o que reduz a efetividade da redução de energia possível.

Um método de escalonamento dinâmico também é proposto em [YAO 95], chamado de heurística da taxa média, com o mesmo modelo do método estático. Com cada tarefa é associado um valor de taxa média requerida, que é calculada dividindo o número de ciclos requerido pela diferença entre o tempo de chegada da tarefa e o seu *deadline*. O método escolhe as tarefas com uma política EDF (*earliest deadline first* – primeira *deadline* primeiro). Como as taxas médias são calculadas estaticamente com um número fixo de ciclos de execução, o mesmo problema da variação de tempos de execução também ocorre neste método.

Em [SHI 99a] é proposto um método de escalonamento de prioridade fixa visando baixo consumo de energia. O método reduz o consumo de energia aproveitando momentos ociosos do processador oriundos do comportamento normal do sistema e variações no tempo de execução das tarefas. Um mecanismo elaborado para ajustar a tensão do circuito e a freqüência de operação aproveita o tempo ocioso para economizar energia. Para o ajuste da velocidade do processador, Shin propõe duas técnicas. A primeira, uma solução heurística simples, se mostrou segura e precisa o suficiente para ser usada em várias aplicações. No entanto, a solução heurística pode não conseguir aproveitar todo o potencial de economia de energia. Para remediar isto, a segunda solução calcula o escalonamento ótimo, mas consome mais tempo de execução.

Também é possível explorar questões de potência nos compiladores. Por décadas foram pesquisadas formas de otimização do código, visando gerar código mais eficiente do ponto de vista do desempenho. Mas também é possível otimizar o consumo de energia no código gerado. Por exemplo, [LEE 95] explora o problema de alocar memória para variáveis em software DSP embarcado com o objetivo de maximizar transferências de dados simultâneas de diferentes bancos de memória para os registradores. Carregar os registradores com uma operação de transferência dupla exige ligeiramente mais corrente que uma operação de transferência simples, com a vantagem adicional de que ambas as operações executam em um único ciclo. Como a transferência ocorre em um ciclo, consumindo menos energia que duas operações em seqüência, maximizar o uso de operações de transferências duplas aumenta a economia de energia. Os resultados mostram que até 47% da energia pode ser economizada em um processador DSP Fujitsu, trocando-se duas operações de transferência em seqüência por uma operação de transferência dupla equivalente [LEE 95].

Assim como os processadores DSP, os processadores de propósito geral também podem se beneficiar de cuidados especiais para acessar a memória. Como o acesso à memória é mais caro que o acesso a registradores, minimizar o uso de instruções que acessam a memória traz ganhos em potência e desempenho [TIW 94]. Uma forma de reduzir os acessos à memória é assinalando as variáveis ativas de um programa a registradores. Como os processadores têm um número limitado de registradores, algumas variáveis acabam tendo que ser alocadas na memória. Mas a escolha adequada das variáveis que serão assinaladas a registradores reduz o consumo de energia e aumenta o desempenho do programa. Em [KOL 96] é apresentado um algoritmo para assinalamento ótimo de variáveis a registradores visando economia de energia.

Assim como os compiladores podem utilizar técnicas para tornar os programas mais eficientes no acesso à memória, também é possível desenvolver as aplicações visando otimizar este acesso. O uso das estruturas de dados adequadas pode trazer economia de energia, além do melhor desempenho já esperado. Em [WUY 96] é analisada a implementação de conjuntos, um tipo abstrato de dados muito utilizado em aplicações de bancos de dados e comunicações. Um conjunto tem as operações de inserção, remoção e busca. É possível implementar conjuntos com várias estruturas de dados, como listas encadeadas, árvores binárias e arrays. Como esperado, as operações com a memória são as grandes responsáveis pelo consumo de energia. As estruturas são comparadas com base em um fator de preenchimento. Para diferentes fatores de preenchimento, estruturas diferentes têm o menor custo em energia. Por exemplo, para um fator de preenchimento superior a 60%, o array é a estrutura mais adequada, consumindo menor quantidade de energia.

2.4 Consumo de energia de sistemas com display gráfico

Apesar de ser comum que os sistemas embarcados atuais utilizem processadores funcionando em freqüências altas e utilizando grandes quantidades de memória, os sistemas de display são responsáveis por grande parte do consumo de energia [CHO 2002]. Isso ocorre porque o display não tem tempo ocioso esperando por entrada do usuário, como ocorre com o processador e a memória principal.

O sistema de display fica em funcionamento praticamente todo o tempo em que o usuário está utilizando o sistema. Dessa forma, não é possível desligá-lo, e qualquer redução na energia gasta no seu funcionamento deve evitar prejudicar a interação do

usuário com o sistema. Por existirem muitos desafios na implementação de sistemas de display que consumam menos energia, vários trabalhos estudaram este problema, como [CHO 2002], [CHO 2003], [SHI 2005], [SAL 2004] e [HOL 2004].

Técnicas de redução de energia foram aplicadas para componentes individuais de sistemas de display, como em [CHO 2000]. Isto fez com que os componentes, como os painéis LCD e a luz de iluminação traseira do painel, se tornassem mais eficientes em termos de energia. No entanto, abordagens em nível de sistema são mais promissoras [CHO 2002].

A Figura 2.2, retirada de [HOL 2004] mostra a organização e o fluxo de dados de um sistema que inclui uma unidade de display.

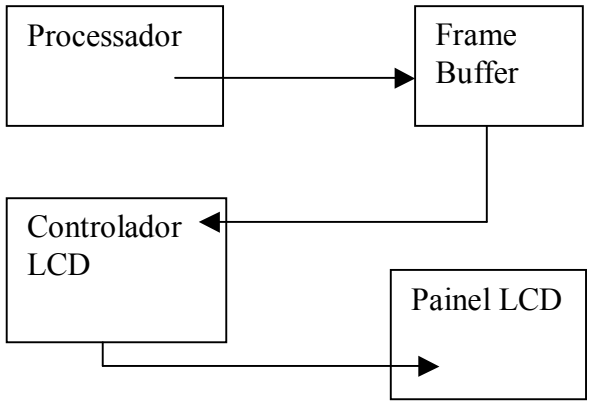

Figura 2.2: Organização de sistema com display

Nos sistemas tradicionais, como descrito na Figura 2.2, os acessos à memória consomem grande quantidade de energia, tanto nas próprias memórias quanto nos barramentos utilizados para transferir as informações entre a memória e o componente que a está lendo, normalmente o processador.

No caso de aplicações gráficas, a aplicação escreve no frame buffer os dados que deverão ser mostrados no display. O frame buffer é a memória responsável por manter a imagem que será mostrada no display. Para mudar a imagem do display, altera-se o conteúdo do frame buffer.

Normalmente, a unidade de display é um componente separado. As informações são passadas para o display através de um intermediário, o frame buffer. Este buffer pode ser localizado na memória principal da plataforma ou em uma memória separada, contida no controlador do LCD. É função do controlador do LCD ler o frame buffer e gerar os sinais para o painel do display. A cada atualização, todo o conteúdo do frame buffer é transmitido pelo controlador para o painel do LCD. Este processo causa grande consumo de energia.

O display é atualizado a uma determinada taxa, chamada de freqüência de varredura vertical, expressa em Hertz (Hz). Esta freqüência costuma variar, sendo comum encontrarmos freqüências de 60Hz ou mais. Isto significa que a cada segundo a imagem contida no frame buffer será transmitida ao circuito que controla o display 60 vezes, fazendo com que este possa consumir mais de 20% da potência total do sistema [HOL 2004].

No caso do frame buffer estar contido na memória principal, o controlador do display realiza a transferência utilizando um controlador de acesso direto à memória (DMA – Direct Memory Access). Como o display deve ser atualizado de acordo com sua freqüência vertical, com valores como 60Hz e 75Hz, o barramento do processador é utilizado, podendo resultar em menor desempenho, caso este precise acessar o barramento. Existe, claro, a vantagem de reduzir o número de chips no sistema, ficando a cargo do projetista determinar a melhor abordagem.

Quando o frame buffer está integrado ao controlador do display, elimina-se o problema de utilizar o barramento do processador para atualizar o display. Um barramento dedicado faz a comunicação do controlador com o painel do display. Desta forma o barramento do sistema fica livre para os outros dispositivos e o consumo de energia para atualizar o display é reduzido. O barramento dedicado entre o controlador e o painel do display é mais curto e menos complexo, e isto se traduz em menor consumo de energia nas atualizações do display.

A utilização dos barramentos e da memória que ocorrem em razão da atualização do display consomem parte importante da potência do sistema. A cada atualização do display, que ocorre a intervalos fixos, todo o conteúdo do *frame buffer* é transferido, através do barramento, para o display. Em um sistema onde o *frame buffer* é localizado na memória principal, o consumo de energia na memória e barramentos relacionados ao *frame buffer* pode chegar a 28% do consumo total [CHO 2002].

2.5 Compressão do Frame Buffer

Como o frame buffer e o barramento entre este e o painel do display são fontes importantes de consumo de energia, redução no número de acessos ao frame buffer e também na quantidade de transferências pelo barramento são uma forma de reduzir o consumo de energia de sistemas com display. Uma forma de se conseguir reduzir estes dois fatores é a utilização de técnicas de compressão de imagens. Este tipo de abordagem é estudado em [SHI 2005] e [SHI 2004].

Existem várias técnicas de compressão de dados [SAL 2003]. Uma técnica simples, a compressão RLE, comprime valores que se repetem em seqüência. Já Huffman, por exemplo, gera uma codificação de tamanho variável bastante eficiente em termos de compressão, mas as freqüências dos símbolos devem ser conhecidas antes da geração da codificação, o que implica em ler duas vezes os dados de entrada, uma para gerar o código e outra para codificar os dados. A codificação aritmética, diferente do codificação de Huffman, garante codificação ótima, mas é um processo iterativo caro. A técnica de compressão Lempel-Ziv-Welch (LZW) é uma técnica que executa apenas uma passada nos dados de entrada e gera saída de tamanho fixo.

No entanto, estas técnicas não são efetivas para um número grande de símbolos, como valores de cores em 16 ou 32 bits. As técnicas são aplicadas a valores de 8 bits, como texto ASCII. A compressão de dados gráficos só é possível quando a quantidade de cores é pequena.

É possível adaptar estas técnicas para utilização em sistemas gráficos. Uma abordagem, descrita nas seções a seguir, consiste em preparar o controlador do LCD para realizar a compressão e descompressão dos dados do frame buffer.

2.5.1 Compressão RLE

A compressão RLE (Run Length Encoding) funciona substituindo repetições de um dado d por um par nd, onde n é o número de repetições e d é o dado que se repete [SAL 2000]. Este tipo de compressão se prova eficiente em imagens simples, ou seja, com poucas cores, que se repetem. A compressão RLE aproveita o fato de que ao escolher-se um pixel de uma imagem, é provável que os pixels adjacentes terão a mesma cor. O algoritmo de compressão varre a imagem linha a linha, procurando por pixels de mesma cor em seqüência.

Para imagens em preto e branco não é necessário indicar qual cor se repete. Basta que se saiba com qual cor a imagem começa. Se, por exemplo, a imagem começar com 10 pixels brancos, depois apresentar 5 pixels pretos e a seguir 5 pixels pretos, a codificação ficaria 10, 5, 5, assumindo que se especifique que a cor branca é a primeira a aparecer. Para o mesmo caso, se o esperado for que a primeira cor seja o preto, basta colocar um primeiro valor 0 (zero), indicando que a imagem não começa com pixels pretos. Isto fica claro na Figura 2.3.

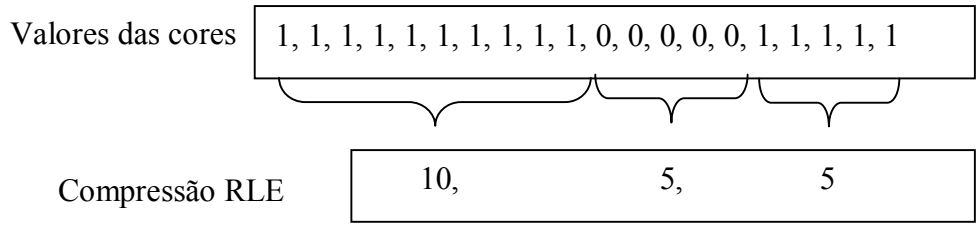

Figura 2.3: Compressão RLE

Em imagens em escala de cinza ou coloridas, onde o valor da cor é representado por apenas um byte, é necessário indicar o número de repetições e a cor que se repete. O problema é identificar se o valor é uma cor ou o número de repetições. Para resolver isso pode-se limitar a quantidade de cores à 128, reservando o primeiro bit para indicar se o valor é uma cor ou o número de repetições do próximo valor. Outra forma é selecionar um valor especial para indicar quando o próximo byte será uma contagem de repetições. Várias outras possibilidades existem para resolver este problema.

Em imagens onde a cor é representada com vários bytes, como na codificação RGB (Red, Green, Blue), a codificação pode ser aplicada independentemente a cada componente.

2.5.2 Compressão RLE Adaptativa do Frame Buffer

Em [SHI 2004], é apresentada uma técnica de compressão do frame buffer baseada na codificação RLE. A compressão é feito sobre valores de 16 bits. Um problema encontrado pelos autores é que a codificação RLE pode fazer com que a imagem ocupe mais espaço na memória, se alguns cuidados não forem tomados. Essa situação é chamada pelos autores de compressão negativa.

Para evitar compressão negativa, os autores utilizam um algoritmo melhorado, chamado por eles de RLE adaptativo. Este algoritmo comprime os dados somente quando pelo menos dois pixels consecutivos têm a mesma cor. Isto garante que o tamanho em memória da imagem compactada não seja maior que o tamanho original.

Para determinar se um valor na imagem compactada é um pixel ou uma quantidade de repetições, o bit de mais baixa ordem do componente verde da cor foi utilizado como *flag*. Isto não tem impacto grande no resultado final da imagem pois a componente verde tem 6 bits, ou seja, um bit a mais que as outras componentes.

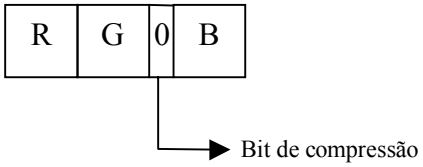

Figura 2.4: *Flag* de compressão

Embora a redução do tamanho em memória da imagem corresponda a uma diminuição do custo em energia, o ganho real é dado pela diferença entre a energia economizada e o trabalho adicional causado pela compressão. O custo da compressão não é grande, mas, se a imagem for alterada frequentemente, o custo pode se revelar proibitivo. Para remediar este problema, os autores propõem uma abordagem incremental, dividindo a tela em regiões e mantendo *dirty flags*, que indicam quando uma região foi alterada.

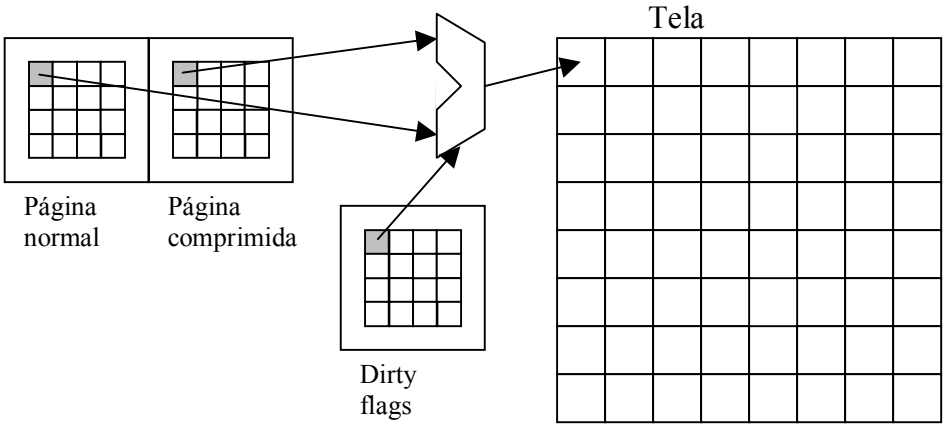

Figura 2.5: Divisão da tela em regiões, com um dirty flag para cada região

Para gerar os resultados, os autores desenvolveram um controlador LCD em plataforma FPGA. O controlador LCD com compressão do frame buffer consome 30mW a mais que o controlador sem compressão, que consome 470mW em média. O protótipo foi feito em uma placa com interface PCI conectada a um PC com sistema Linux.

A compressão incremental se beneficia de blocos com largura pequena. A largura do bloco afeta diretamente a taxa de compressão e o custo de implementação. Em geral, a altura dos blocos não tem muita influência na economia de energia, a não ser quando a altura ultrapassa 128 pixels. Isto ocorre porque a compressão é feita dentro de cada linha, reduzindo o impacto da altura da região no resultado.

Em aplicações que não requerem atualizações parciais, o ganho de energia é afetado somente pela taxa de compressão, que aumenta conforme aumenta a largura dos blocos. Nesses casos, a altura dos blocos não influi na compressão, mas blocos de altura menor implicam em mais registradores para indicar blocos alterados (dirty flags).

Já nas aplicações que requerem atualizações parciais da tela, ocorre a invalidação de todo o conteúdo dos blocos da região afetada, mesmo que uma parte do conteúdo de um bloco não seja afetada. Isso resulta em um ganho menor no consumo de energia, mesmo com uma taxa de compressão alta.

A técnica de compressão do frame buffer apresentada em [SHI 2004] reduz de 50% a 66% o consumo de energia do frame buffer e dos barramentos utilizados na atualização do LCD. Isto resulta em uma economia de 10% a 15% da energia total consumida no protótipo, onde o controlador do LCD é implementado em FPGA. Como o FPGA consome mais energia que uma implementação em silício, o ganho em energia deve ser superior, podendo ficar entre 13% e 17%.

2.5.3 Codificação Huffman

Para fazer a codificação Huffman, o método começa construindo uma lista de todos os símbolos do alfabeto (valores) em ordem decrescente de freqüência [SAL 2003]. Então, uma árvore é construída das folhas para a raiz, com cada símbolo em uma folha. Isto é feito em passos, onde a cada passo os dois símbolos com as menores probabilidades são selecionados, adicionados ao topo da árvore parcial, removidos da lista, e trocados por um símbolo auxiliar representando ambos. Quando a lista for reduzida a apenas um símbolo auxiliar (representando todo o alfabeto), a árvore estará completa. A árvore é então varrida para determinar os códigos dos símbolos.

Para facilitar a compreensão de um exemplo, vamos utilizar as letras de ‘a’ a ‘e’, supondo que o alfabeto (o conjunto dos símbolos da informação que será codificada) tem 5 símbolos. Ao se codificar a cadeia “aabbaaccde”, precisa-se primeiramente definir as probabilidades de cada símbolo. Como tem-se 10 caracteres na cadeia, e destes, 4 são letras ‘a’, então a probabilidade do símbolo ‘a’ é de 0.4. Seguindo a mesma lógica, as probabilidades dos símbolos ‘b’ e ‘c’ são de 0.2 cada, e as probabilidades dos símbolos ‘d’ e ‘e’ são de 0.1 cada.

Na Figura 2.6, que mostra uma árvore completa, o código foi construído da seguinte forma:

- ‘d’ e ‘e’ foram combinados e substituídos pelo símbolo ‘de’, cuja probabilidade é 0.2.
- ‘de’ e ‘c’ foram combinados e substituídos pelo símbolo ‘cde’, cuja probabilidade é 0.4.
- ‘cde’ e ‘b’ foram combinados e substituídos pelo símbolo ‘bcde’, cuja probabilidade é 0.6.
- ‘bcde’ e ‘a’ foram combinados e substuídos pelo símbolo ‘abcde’, cuja probabilidade é 1.

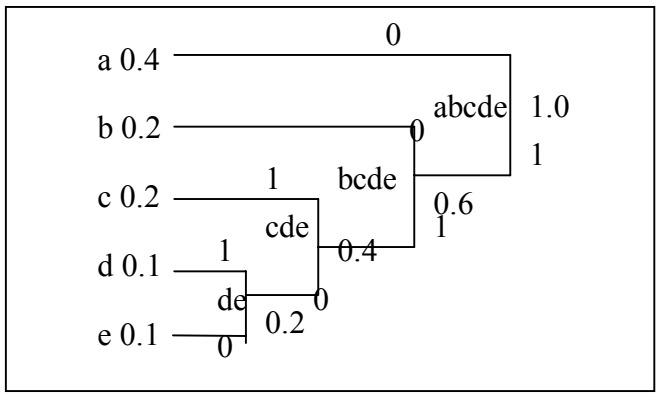

Figura 2.6: Exemplo de construção de um código de Huffman

As decisões de combinar símbolos são arbitrárias. Combinações diferentes gerarão códigos diferentes. A cadeia compactada deve conter, além dos dados codificados, também as freqüências dos símbolos, de forma que o decodificador possa reconstruir o código para decodificar os dados.

Para decodificar os dados percorre-se a árvore a partir da raiz, seguindo o ramo que for indicado na cadeia de bits codificada até chegar a uma folha da árvore. Lá encontra-se o símbolo que deve ser emitido pelo decodificador. Repete-se o processo, a partir da raiz da árvore, até que toda a cadeia seja decodificada.

2.5.4 Compressão Huffman Diferencial do Frame Buffer

Em [SHI 2005] foi observado que o número de cores na tela é significantemente superior ao número de diferenças entre as cores de pixels adjacentes. No artigo é utilizado o conjunto de benchmarks MobileMark 2002 [BUS 2002], onde os autores constatam que as cores dos pixels na maior parte da imagem podem ser representados por um pequeno conjunto de diferenças de cores.

Valores das cores	10, 5, 6, 5, 5, 5, 9, 30, 10, 10, 11, 3, 4
Diferenças entre as cores	10, -5, 1, -1, 0, 0, 4, 21, -20, 0, 1, 2, 1

Figura 2.7: Exemplo de codificação usando diferenças entre cores adjacentes na tela

Shim define como cores críticas o conjunto de cores que é necessário para representar mais de 90% da área da tela, e como diferenças críticas de cores, como o conjunto de diferenças necessárias para o mesmo fim. O número de diferenças críticas não excede 32 diferenças, para a maioria das aplicações. Aplicações que manipulam fotos digitais, no entanto, em geral requerem mais diferenças críticas.

Por trabalhar com diferenças ao invés dos valores de fato das cores, o método de codificação passa a ser chamado de diferencial. Ou seja, a codificação de Huffman aplicada sobre diferenças de cores se torna uma codificação Huffman diferencial. A compressão tem desempenho superior ao método RLE adaptativo descrito na seção 2.5.2, como mostra a Tabela 2.1, retirada de [SHI 2005].

Entretanto, existem problemas para aplicar a codificação Huffman diferencial. A codificação precisa ler os dados de entrada mais de uma vez, pois precisa conhecer as freqüências de todas as diferenças de cores para poder construir a árvore de código. Outro problema é o número elevado de símbolos, que tende a crescer com o aumento do número de cores simultâneas possíveis.

Para resolver esse problema, Shim propõe limitar o tamanho da tabela de código, tendo como resultado uma compressão mais efetiva, levando em consideração o desempenho e a área ocupada. Cada entrada na tabela de código consiste de uma diferença entre cores, sua freqüência e uma palavra de código (a codificação em bits). A última entrada na tabela é a freqüência e palavra de código para os pixels restantes, que não são representados por diferenças críticas.

Tabela 2.1: Comparação entre os métodos RLE, Huffman diferencial e Huffman diferencial com código de tamanho limitado (% do tamanho original) [SHI 2005]

Aplicação	RLE	Huffman differencial	Huffman diferencial com código limitado
Photoshop	71,02	32,06	47,17
Flash	39,66	19,01	38,50
Word	14,95	8,88	9,64
Excel	28,69	10,96	11,63
PowerPoint	44,96	19,40	27,86
Slide show	69,55	21,22	26,94
Outlook	26,36	12,65	14,90
Communicator	24,23	11,80	19,27

Para que o desempenho da compressão seja adequado é necessário escolher um bom conjunto de diferenças críticas. Para fazer esta escolha é construída uma tabela contendo a diferença, sua freqüência e uma etiqueta de tempo. Durante o primeiro período de atualização da tela são selecionadas 32 diferenças críticas, usando uma tabela de 64 entradas. A política de atualização desta tabela é LRFU (menos usado recentemente e freqüentemente), com taxa de troca de 1%, para o caso em que uma nova diferença entre cores deve ser colocada na tabela. Após escolher um conjunto de 32 diferenças críticas, estas são analisadas em relação ao conteúdo do frame buffer mais uma vez, com o objetivo de construir um modelo estatístico preciso durante o segundo período de atualização da tela.

Ao final do segundo período de atualização, os códigos de Huffman são gerados durante o período ocioso que existe entre duas atualizações consecutivas. Com isso, a tabela de código é gerada. Durante um terceiro período de atualização é feita a compressão do frame buffer, utilizando a tabela gerada anteriormente.Como o algoritmo faz a leitura do frame buffer 3 vezes, ele é executado em 3 estágios por um circuito de compressão integrado ao controlador do LCD.

Embora esta técnica utilize um número reduzido de diferenças entre cores na tabela, sua lógica de compressão é mais complexa e exige dois tipos de memórias endereçáveis

pelo conteúdo: uma para a tabela de diferença entre cores e outra para o código de Huffman. Reduzir o número de entradas nestas tabelas é essencial para reduzir o consumo de energia do sistema.

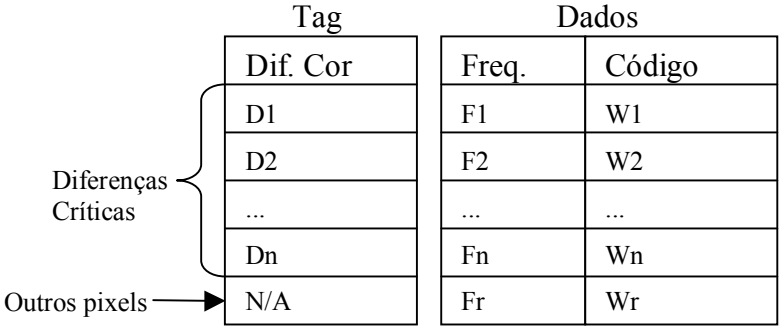

Figura 2.8: Organização da tabela de código

Uma das técnicas utilizadas para reduzir o número de diferenças é trocar diferenças negativas por positivas. Por exemplo, entre os números 31 e 0, que ficam nos extremos da faixa de valores, a diferença é de -31. Mas se observamos que se somarmos 1 ao valor 31, este passa a ser zero (desconsideramos o bit de vai-um), podemos trocar essa diferença pelo número positivo 1. Essa normalização das diferenças faz com que algumas entradas na tabela sejam iguais, podendo ser transformadas em uma só entrada.

Outra forma de reduzir o tamanho da tabela é usar a simetria das diferenças entre cores. Diferenças do tipo (Cr, Cg, Cb) e (32-Cr, 32-Cg, 32-Cb) aparecem com freqüência quase idêntica, de forma que não necessidade de contá-las separadamente. Aplicando esta técnica, ilustrada pela Figura 2.9, aumenta-se o desempenho da compressão.

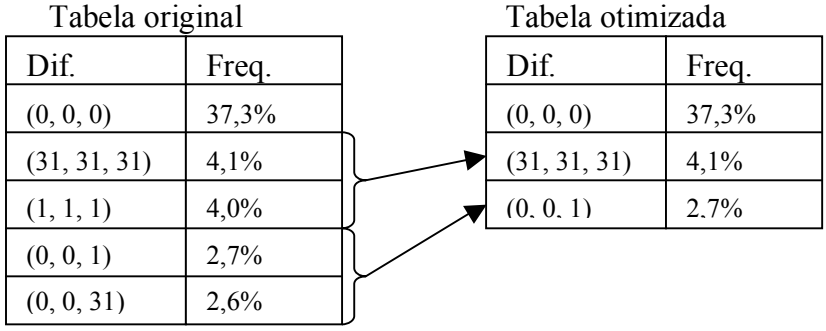

Figura 2.9: Reduzindo o tamanho da tabela através da simetria das diferenças

A lógica de compressão do circuito precisa decidir quando deve substituir o código de Huffman por um novo que corresponda ao estado atual da tela. Se o código for substituído, todo o frame buffer deve ser comprimido novamente, e isto consome energia. Em geral, o conteúdo do frame buffer se mantém relativamente constante enquanto a mesma aplicação está ativa. Quando o usuário muda a aplicação, o código precisa ser atualizado, pois o conjunto de cores utilizado muda, reduzindo a taxa de compressão.

Como a atualização do código e a re-compressão do frame buffer é cara, ela só é feita quando a degradação do desempenho da compressão chega a 3%, ou seja, o código antigo tem diferença de 3% na compressão em relação ao novo código calculado.

A utilização da codificação de Huffman diferencial com tamanho de código limitado, em combinação com as técnicas descritas acima, reduz de 52% a 90% a atividade do frame buffer e dos barramentos de comunicação deste com o controlador do display e o barramento principal do sistema. Com isto, tem-se uma economia de 28% a 55% no sistema de frame buffer [SHI 2005].

2.6 Organização otimizada da memória do display

Em sistemas equipados com displays LCD (Liquid Crystal Display), a luz traseira do display consome em média 30% da energia total do sistema [CHO 2002]. No entanto, estão surgindo displays que operam sem a necessidade de luz traseira, os displays OLED (Organic Light Emitting Diode), nos quais os próprios pixels geram luz [RAJ 2000].

O uso do frame buffer como intermediário na comunicação com o painel do display permite simplificar as aplicações, evitando que estas tenham que lidar diretamente com a sincronização com atualização do display.

Em [HOL 2004] é proposto um método de otimização de duas etapas que reduz acessos redundantes à memória na comunicação com a unidade de display. A primeira etapa trata de otimizar o acesso ao frame buffer, fazendo com que apenas as partes da tela que foram alteradas sejam escritas no frame buffer. Com isto, os acessos à memória feitos pelo processador são reduzidos, reduzindo assim também o consumo de energia do sistema. A segunda etapa envolve uma arquitetura que elimina o frame buffer, utilizando um display OLED, que mantém a informação dos pixels no próprio painel do display, eliminando acessos caros à memória através do barramento. A técnica é avaliada utilizando software de teste dedicado e uma aplicação de decodificação de vídeo.

Situações onde todos os pixels do display diferem de um quadro para outro ocorrem muito raramente. Em uma organização onde é possível apenas atualizar os pixels que foram alterados consegue-se evitar acessos desnecessários à memória. Muitas aplicações podem aproveitar essa abordagem. Aplicações típicas que requerem entrada do usuário podem atualizar apenas a área onde o usuário está fazendo alterações. Aplicações de vídeo ou jogos podem atualizar apenas as partes da imagem onde houve alterações.

A primeira etapa de otimização, proposta em [HOL 2004], é modificar o software que gera os dados para o frame buffer. Ao invés de escrever o frame buffer inteiro, a aplicação deve somente atualizar os pixels que foram alterados. Isto requer que aplicação faça este tipo de controle de alguma forma. Essa abordagem reduz consideravelmente o número de acessos à memória. No entanto, isto só pode ser feito em sistemas que usam apenas um frame buffer. Os sistemas que usam frame buffer duplo (para evitar falhas na imagem quando a atualização é feita durante o retraço) devem ser adaptadas para utilizar apenas um frame buffer. Para tanto, é necessário fazer a sincronização da aplicação com o retraço do LCD. Além disso, a geração de novos dados para o frame buffer não deve ser mais lenta que a varredura do frame buffer.

A segunda etapa da otimização consiste na alteração da organização do hardware. A atualização de displays do tipo LCD gera grande volume de tráfego nos barramentos bem como grande volume de acessos à memória. Mas para displays que não precisam

de atualização consegue-se uma economia substancial de energia. Uma tecnologia recente, o display do tipo OLED, apresenta esta característica.

Nos displays OLED a memória é integrada a nível de pixel, ou seja, a imagem do display é contida nos próprios pixels do painel do display. Desta forma não há tráfego nos barramentos durante a atualização da imagem, a menos que esta tenha sido mudada. Com isso, a utilização de um frame buffer pode ser evitada, e a atualização (agora desnecessária) dos pixels é eliminada. Com isso consegue-se reduzir consideravelmente o consumo de energia.

A combinação das otimizações na aplicação descritas na primeira etapa com um display com memória a nível de pixel resulta em uma economia ainda maior em termos de energia. Utilizando esta técnica em duas etapas, [HOL 2004] indica que é possível obter uma economia de até 72% na energia do sistema.

2.7 Otimizações no barramento com o display

Dentro do subsistema que controla o LCD, o barramento com o display responde por uma parcela significativa da potência consumida, em geral em torno de 10%. Uma forma de economizar energia na transmissão dos dados para o display é reduzir a atividade de chaveamento no barramento. Muitos displays LCD utilizam comunicação serial para reduzir os efeitos elétricos que ocorrem durante a transmissão dos dados a uma freqüência na casa das centenas de MHz. Portanto, soluções que reduzam o chaveamento em comunicações seriais são importantes.

A interface digital entre o controlador o display LCD e seu controlador não tem um padrão definido. Em [BOC 2004] e [SAL 2004] foram propostas técnicas baseadas na interface DVI [DDW 99]. Para introduzir estes trabalhos esta interface será brevemente descrita aqui. A interface DVI foi proposta pelo Digital Display Working Group (DDGW), dirigido pela Intel e que inclui companhias como HP, Fujitsu, IBM e NEC.

A interface DVI, como suas competidoras, é baseada em uma tecnologia de transmissão de sinais denominada TMDS (*Transition-Minimized Differential Signaling*). Esta interface é implementada usando um codificador e um serializador para transmitir os pixels no formato RGB. A transmissão é feita em três pares trançados, com um par adicional para o sinal de *clock*. Cada valor de 8 bits é codificado e serializado na forma de um valor de 10 bits. O uso de TDMS resulta em poucas transições no fio por usar codificação de transições, onde as transições são codificadas, não os valores. Uma característica importante é que o *clock* é usado apenas como referência de freqüência, evitando problemas de *clock skew*.

Apesar do nome, TDMS não foi concebido visando reduzir consumo de energia. A minimização de transições foi pensada com o intuito de evitar interferência eletromagnética excessiva no cabo. Mas o fato de codificar transições e não valores torna esta interface bastante atraente para a implementação de técnicas de redução de consumo de energia.

Em [CHE 2003], um esquema de codificação chamado de codificação cromática foi proposto para reduzir a atividade de chaveamento em cada canal de cor, baseando-se no princípio de localidade tonal, ou seja, no fato de que pixels adjacentes têm grande semelhança nas suas componentes RGB.

A técnica descrita em [BOC 2004] também se baseia na localidade tonal e utiliza diferenças entre pixels, ao invés dos seus valores. A codificação por transições não garante redução no número de transições nos canais. Para isso, uma codificação adequada das diferenças é necessária. Pelo fato de ser uma conexão serial, as alternativas para reduzir transições são explorar a redundância temporal e a seqüência temporal dos bits transmitidos. Como a primeira forma não pode ser explorada na interface DVI, pois esta utiliza dois bits de redundância por símbolo, resta apenas explorar a seqüência temporal.

O problema a ser resolvido é minimizar o número de transições na transmissão de uma palavra. Para isso, basta agrupar os bits iguais no início ou fim de uma palavra. Com isso a transmissão de um valor pode ser feita com apenas uma transição. A solução adotada em [BOC 2004] é codificar as 18 diferenças mais comuns como uma palavra com uma transição apenas. Outras diferenças não são utilizadas. Para transmitir outros valores, o próprio valor do pixel é enviado. Com esta técnica, Bocca conseguiu uma redução média de 55% no número de transições na interface DVI.

Em [SAL 2004] é proposto uma codificação otimizada para utilização com displays LCD, combinando as técnicas utilizadas em [CHE 2003] e [BOC 2004]. A codificação é denominada LIWT (Limited Intra-Word Transition). Diferente de [BOC 2004], a codificação LIWT utiliza códigos com mais de uma transição. O mapeamento é feito em ordem inversa: os códigos com menos transições são os que tem maior probabilidade. Como são construídos códigos com mais de uma transição, é possível representar uma faixa maior de valores.

Outra diferença em relação aos trabalhos anteriores é a consideração das transições entre pixels consecutivos. Isto é importante pois a técnica básica minimiza as transições dentro de uma mesma palavra, o que torna as transições de uma palavra para outra uma fração importante das transições totais. Com isto, Salerno conseguiu uma média de 60% de redução nas transições na trasmissão dos dados pela interface DVI.

2.8 Manipulação de parâmetros dos displays LCD

Em [CHO 2002] são descritas formas de reduzir o consumo dos displays LCD através da manipulação de parâmetros destes dispositivos. São propostas três técnicas que podem ser usadas em conjunto: manipulação da taxa de atualização, controle da profundidade de cores e redução da luz traseira acompanhada de compensação de brilho e melhoria no contraste.

Em displays do tipo CRT (Cathode Ray Tube), a diminuição da taxa de atualização da imagem implica na redução da qualidade da mesma. Já nos displays LCD isso não ocorre. Nestes, quando a taxa de atualização passa de um determinado patamar, não se nota diferença na qualidade da imagem. Por essa razão, os displays LCD em geral apresentam taxa de atualização fixa, diferente dos displays CRT, onde esta taxa é configurável. Reduzir a taxa de atualização economiza energia no frame buffer e nos barramentos, porque menos operações são feitas nestes componentes do sistema.

A maioria dos sistemas de display coloridos comerciais permitem o controle da profundidade de cores (número de cores simultâneas no display). Quando esta profundidade é reduzida, o frame buffer passa a comportar resoluções maiores ou mais páginas de memória. No entanto, também é possível utilizar a redução do número de

cores para reduzir o consumo de energia. Em [CHO 2002] é proposto que as aplicações continuem operando com o número normal de cores (o máximo), mas que o controlador do LCD ajuste a quantidade de cores para reduzir energia. Dessa forma, as aplicações não perdem compatibilidade.

Como o desligamento das memórias SDRAM é feito a nível de dispositivo, é possível desligar um dos componentes de memória se a profundidade de cores for reduzida de 16 bits para 8 bits.

Os displays LCD em dispositivos como os celulares utilizam uma fonte de luz atrás do display para iluminá-lo. Esta luz é uma das fontes consumidoras de potência do sistema, podendo consumir cerca de 30% da potência total do sistema [CHO 2002]. Para reduzir o consumo de energia pela luz do display foi proposta em [CHO 2002] e [GAT 2002] o idéia de ajustar a iluminação do display, mas aumentar concorrentemente a transmissividade do painel LCD, para compensar a perda de fidelidade. Esta técnica foi chamada de *Backlight Scaling*.

Se a imagem alterada por *Backlight Scaling* é idêntica à original, então não houve perda de fidelidade. No entanto, admitindo-se alguma diminuição na qualidade da imagem, é possível conseguir maior economia de energia. Com isso, a compensação com o ajuste do brilho não é mais possível, fazendo com que a fidelidade da imagem seja afetada. Entretanto, a simples métrica do brilho restringe muito a política de *backlight scaling*, tornando a técnica ineficaz. Como a diminuição da luz do display limita a faixa de brilho da imagem, uma política de *backlight scaling* que não cause variações no brilho é muito conservadora para conseguir economizar energia de forma substancial. Para permitir uma variação menos conservadora, em [CHE 2004] é proposto que se utilize o contraste da imagem como métrica para determinar a fidelidade da imagem após a aplicação da técnica de *backlight scaling*.

O termo contraste descreve a diferença entre pixels claros e escuros. Em displays LCD com luz traseira, o controle de brilho ajusta a iluminação proporcionada pela luz traseira enquanto o contraste ajusta a transmissividade do display LCD. A fidelidade do contraste é definida sem quantificar o contraste em si, que não tem definição universal. Adotando o contraste da imagem como métrica para determinar a fidelidade da imagem, Cheng consenguiu uma economia de 3,7 vezes na energia, com apenas 10% de distorção do contraste.

Em [PAS 2003] a técnica de redução da luz traseira do display com ajuste do contraste foi estudada para vídeos. A adaptação do vídeo não é feita pelo dispositivo móvel, mas por um servidor *proxy*, que transmite o vídeo já adaptado para o dispositivo, tendo conhecimento dos parâmetros adequados para cada tipo de dispositivo. Pasricha identificou que a compensação do brilho pode ser feita de forma mais agressiva em imagens em movimento, pois a distorção é menos perceptível.

O servidor *proxy* mantém um banco de dados de valores de luminosidade para todos os vídeos armazenados no servidor de vídeos, parâmetros para os dispositivos e uma base de regras para determinar valores de compensação. Toda a comunicação entre o dispositivo móvel e o servidor de vídeos passa pelo *proxy*, de forma que este pode alterar os vídeos em tempo real.

O cliente utiliza uma camada de software que trata toda a comunicação com o servidor. Um componente de gerência de comunicação é responsável por enviar requisições de vídeo para o servidor *proxy*, junto com informações sobre os recursos do

dispositivo, como número de níveis de luz traseira, nível atual da mesma, e o tipo do dispositivo. Essas informações são coletadas do dispositivo por um componente de monitoramento.

Para medir a qualidade da adaptação dos vídeos, Pasricha utilizou um método subejtivo, selecionando 30 pessoas para avaliar a qualidade dos vídeos após a adaptação. As respostas dadas pelas pessoas selecionadas foram analisadas de forma a determinar os parâmetros do algoritmo de compensação. Após analisar os resultados da pesquisa, Pasricha chegou a um conjunto de regras, valores de luminosidade e parâmetros no algoritmo de compensação que resultam em economia de energia de até 60% com diminuição aceitável da qualidade dos vídeos na avaliação subjetiva das pessoas selecionadas para o estudo.

2.9 Crítica e contextualização do trabalho

A idéia deste trabalho é analisar a possibilidade de reduzir o consumo de energia em aplicações gráficas a partir do software, sem manipular parâmetros de hardware como freqüência de operação e tensão de alimentação. Muitas das técnicas propostas na literatura são adequadas apenas para a implementação em hardware.

Uma das abordagens escolhidas é a codificação (compressão) das imagens, de forma a reduzir os acessos à memória. No entanto, várias técnicas de compressão, como Huffman, por exemplo, utilizam uma tabela de códigos. Embora o uso destas técnicas resulte em taxas de compressão maiores, os acessos à memória não serão reduzidos. O caso típico implica em acessar a cadeia compactada e acessar a tabela para escrever um pixel na imagem final, ou seja, são duas leituras e uma escrita, contra uma leitura e uma escrita no caso de não haver compactação.

Outro problema é que a codificação é feita no nível de bit, o que resulta em operações extras para retirar o valor da cadeia compactada. Em hardware isto não é um problema. Uma operação de deslocamento em hardware não tem custo. Mas em software a mesma operação deve ser feita por uma instrução, que irá consumir tempo e ocupar o pipeline do processador. Pelos motivos expostos, a codificação RLE foi escolhida para implementação em software, por ter a característica de não aumentar o número de acessos à memória e ter uma lógica facilmente implementada em software, com poucas instruções.

Mesmo apresentando estas características, não era esperado da codificação RLE um desempenho atraente do ponto de vista de consumo de energia e tempo de execução. Então, uma outra técnica é proposta no capítulo 3. Esta técnica consiste em transformar os dados da imagem em operandos imediatos, eliminando um dos acessos à memória, e otimizando o uso do pipeline, já que se eliminam os laços de execução necessários para copiar as imagens para o *frame buffer*.

3 BIBLIOTECA GRÁFICA PARA O FEMTOJAVA

A biblioteca gráfica desenvolvida para este trabalho é baseada no microcontrolador Femtojava e em sua biblioteca de gerência dinâmica de memória. Este capítulo trata da plataforma Femtojava, que visa execução embarcada, suas ferramentas de desenvolvimento de software e, por fim, a implementação da biblioteca gráfica FemtoGraphics, em suas duas versões, original e otimizada.

3.1 Java em sistemas embarcados

A plataforma Java prevê recursos para sistemas embarcados. Java 2 Micro Edition (J2ME) é uma plataforma para ambientes com restrições de recursos. A especificação J2ME contém máquinas virtuais, configurações e perfis para vários ambientes com necessidades variadas. Com a configuração e perfil adequando, as aplicações J2ME podem executar em diversos dispositivos, como celulares, PDAs, *set-top boxes*, etc.

Uma máquina virtual Java com bibliotecas principais, classes e API constitui uma configuração J2ME. Exitem duas configurações: Connected Device Configuration (CDC)[SUN 2001] e Connected Limited Device Configuration (CLDC)[SUN 2002d][SUN 2002e].

Os perfis definem ainda mais o ambiente J2ME, especificando a plataforma adequada para cada dispositivo. Personal Basis [SUN 2002a] e Personal [SUN 2002b] são perfis para a configuração CDC. O perfil Personal é um super-conjunto do perfil Personal Basis. Ambos são baseados no perfil Foundation e ambos permitem o desenvolvimento de aplicações Java para dispositivos de eletrônica de consumo. O perfil MIDP [SUN 2005] é a base para a configuração CLDC. O perfil PDA é destinado a dispositivos PDA e inclui o perfil MIDP, introduzindo a API Personal Information Management [SUN 2002c].

Há ainda outras soluções para sistemas embarcados que não são baseadas em Java, como o *Binary Runtime Environment for Wireless* (BREW), da Qualcomm [QUA 2005] e o *.Net Compact Framework*, da Microsoft [MIC 2005]. A existência destes sistemas é uma grande evidência da necessidade de promover o reuso de componentes através de bibliotecas, para reduzir o tempo de desenvolvimento de sistemas embarcados. Estas bibliotecas facilitam o acesso a periféricos do sistema de forma independente de dispositivo, aumentando a portabilidade do código.

No entanto, não é nosso objetivo neste trabalho atender às especificações da plataforma J2ME. O microcontrolador Femtojava ainda não suporta o suficiente para atender a estas especificações. Este trabalho tem o objetivo de prover algum suporte gráfico ao Femtojava, de forma que novas aplicações sejam possíveis. As seções a seguir descrevem a plataforma Femtojava e a biblioteca gráfica desenvolvida.

3.2 Plataforma Femtojava

Devido à adoção de Java em celulares e dispositivos PDA [LAW 2002], surge a necessidade de melhorar a eficiência do software desenvolvido para a plataforma Java. O microcontrolador Femtojava [ITO 2001] executa bytecodes Java diretamente, sem uma máquina virtual. O Femtojava tem um conjunto de instruções reduzido, arquitetura Harvard, baixo custo em área e a possibilidade de sintetizar um processador personalizado para a aplicação, removendo algumas instruções. Este processo é feito através de uma ferramenta chamada Sashimi, que analisa os bytecodes Java da aplicação e gera uma parte de controle que suporta apenas as instruções que são de fato usadas. Em [BEC 2003] uma versão de baixo consumo de energia do Femtojava foi desenvolvida. Existem outras implementações em hardware de máquinas Java, como Picojava-I [OCO 97], Picojava-II [SUN 99], Komodo [KRE 99] e TRAJA [SHI 99].

O microprocessador Femtojava executa um conjunto limitado dos bytecodes Java. Por ser limitado, existem algumas restrições que devem ser respeitadas pelo software para que este execute no Femtojava. Estas restrições são resultado da execução em hardware dos bytecodes Java. A Figura 3.1, originalmente demonstrada em [AAS 2001] e expandida em [NEV 2005] mostra as formas de execução de programas Java, destacando a execução com um processador Java, como é o caso do Femtojava.

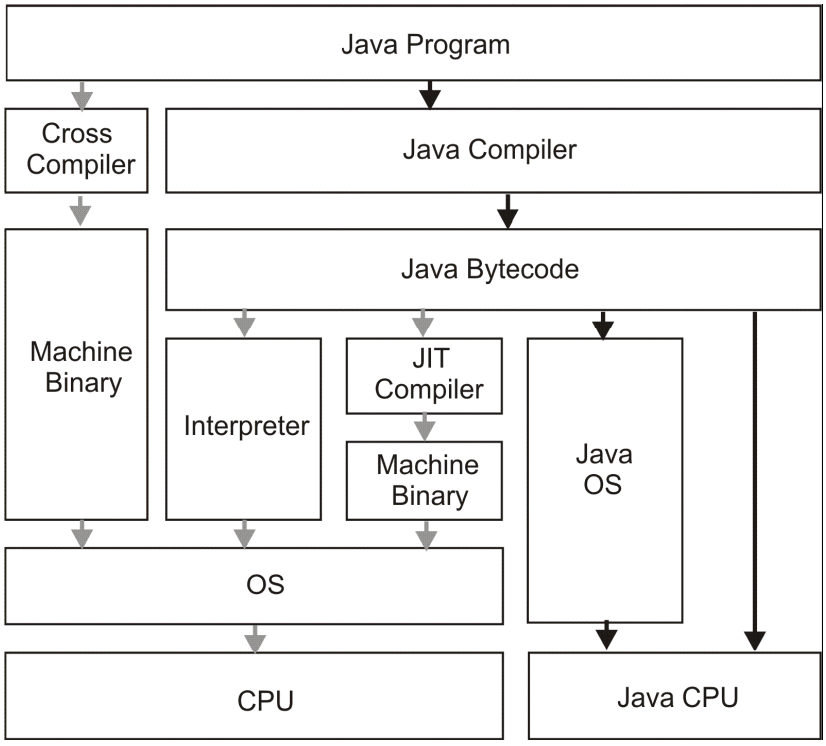

Figura 3.1: Modelos de execução de programas Java [NEV 2005]

Recursos como alocação de objetos e chamadas de métodos de objeto não estão presentes na implementação original. Por isso, o microcontrolador Femtojava é acompanhado de uma ferramenta que faz a adaptação dos arquivos de bytecodes (arquivos .class) para que estes utilizem apenas o subconjunto implementado. Esta ferramenta se chama Sashimi (System As Software and Hardware In Microcontrollers) [ITO 2001].

Por implementar um subconjunto da especificação Java, o microcontrolador Femtojava tem algumas limitações, como a falta de suporte a vários fluxos de execução (threads). Antes do trabalho realizado em [NEV 2005], não havia suporte a alocação dinâmica de memória, o que dificultava o desenvolvimento de aplicações, visto que um esquema estático deveria ser desenvolvido.

3.3 Suporte a Objetos no Femtojava

Alguns recursos são mais facilmente implementados em software, como é o caso da gerência dinâmica de memória. O sistema de gerência de memória do Femtojava, descrito em [NEV 2005], é implementado através de uma biblioteca de software que é adicionada às aplicações através de uma ferramenta. Esta ferramenta determina quando o sistema de gerência deve executar e insere as devidas chamadas a este sistema na aplicação. O fluxo de projeto com a ferramenta Sashimi e o suporte a objetos é exposto na Figura 3.2.

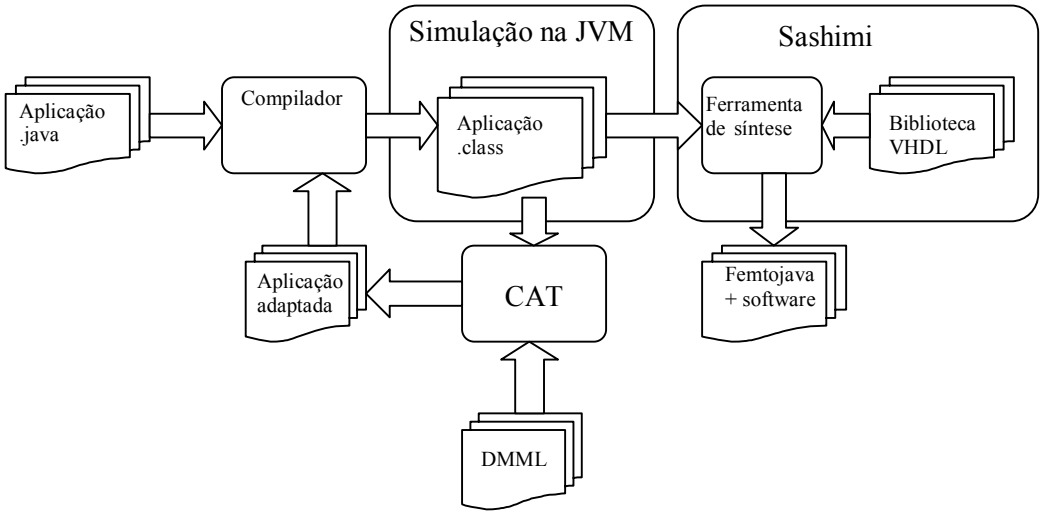

Figura 3.2 – Fluxo de projeto com a ferramenta Sashimi e suporte a objetos

A ferramenta CAT (*Code Adaptation Tool*) é responsável por adaptar a aplicação do usuário, substituindo as operações de alocação de objetos por chamadas à biblioteca DMML (*Dynamic Memory Management Library*). Após adaptada, a aplicação é novamente compilada e simulada. Após, a ferramenta Sashimi faz a adaptação final necessária para a execução no Femtojava.

A implementação permite a alocação de objetos e realiza gerência dinâmica de memória. No entanto, o suporte a objeto ainda é limitado, não sendo possível trabalhar com herança e polimorfismo. E como a implementação é em software, o desempenho

sofre um impacto considerável, pois várias instruções disparam um mecanismo caro de gerência de memória, incluindo simples acessos a arrays.

A biblioteca de gerência de memória foi projetada para conter apenas métodos que respeitam as restrições do microcontrolador Femtojava. A biblioteca permite apenas a criação e manipulação de objetos cujo código contém apenas código suportado pelo Femtojava. Isto, de fato, coloca um problema importante, já que o conjunto de recursos suportados fica restrito, tornando mais difícil validar extensões como a que é proposta neste trabalho.

Há esforços atualmente em desenvolvimento para contornar as restrições existentes no Femtojava e melhorar o suporte aos recursos da especificação da máquina virtual. A biblioteca de gerência dinâmica de memória foi a primeira extensão do conjunto de instruções do Femtojava completamente implementada em software. Ela utiliza uma ferramenta de adaptação de código que troca automaticamente instruções que ativam execução da gerência de memória por chamadas a métodos de classe da biblioteca.

3.4 Biblioteca Não-Otimizada

A biblioteca FemtoGraphics foi concebida para ser uma forma simples de criar aplicações gráficas para dispositivos embarcados baseados no microprocessador Femtojava. Seu projeto teve como objetivo um custo baixo em memória, implementando apenas primitivas sobre as quais uma aplicação mais complexa pode ser desenvolvida. Esta seção apresenta a versão original da biblioteca, sem otimizações no consumo de energia e desempenho.

O desenvolvimento de aplicações com a biblioteca FemtoGraphics começa com o uso do *Femtojava Resource Compiler* (FemtoRC), que gera código Java a partir de arquivos de imagem. Além de imagens, futuramente esta ferramenta suportará outros tipos de recursos. Esta ferramenta não apenas transforma imagens em código Java, como também é responsável por parte da implementação das técnicas descritas na próxima seção para reduzir o consumo de energia.

A ferramenta recebe como entrada um conjunto de arquivos de imagens e devolve como saída um arquivo para imagem contendo a imagem codificada em código Java, já convertida para o formato de cor suportado pela biblioteca de tempo de execução, de acordo com alguns parâmetros que controlam como as imagens são armazenadas e manipuladas.

Depois de gerar os arquivos-fonte a partir das imagens, estes arquivos devem ser compilados junto com a aplicação. Este processo é descrito na Figura 3.3. A aplicação deve ser desenvolvida utilizando um conjunto restrito da funcionalidade da plataforma Java, de acordo com as restrições do nosso ambiente. Esta biblioteca foi projetada com estas restrições em mente.

A classe principal da biblioteca é FemtoSystem. Ela é responsável por prover acesso aos recursos do sistema, como o display gráfico. Futuramente outros recursos serão acessados através desta classe.

A classe mais importante da biblioteca, responsável pela maior parte do processamento, é a que mantém os dados relativos às imagens, e se chama FemtoImage. FemtoImage é responsável pela manipulação de imagens. A operação de maior custo é a transferência de uma imagem para outra, ou seja, a cópia da imagem para o frame

buffer. Estas operações de transferência são executadas a cada atualização da imagem, pois os elementos gráficos que compõem a imagem poderão estar em movimento, como ocorre em aplicações como jogos.

As imagens nem sempre são retangulares. Muitas vezes formas diferentes são representadas. Para isso, alguma cor é escolhida para representar um pixel transparente, isto é, que não aparece na imagem final na tela. A operação de transferência de uma imagem para a tela testa cada pixel para determinar se ele deve ser transferido ou não. Por exemplo, na imagem da Figura 3.4, a região na cor preta não será transferida para a imagem final. A cor definida como transparente é chamada de *chroma key*. A técnica de fundo azul utilizada na televisão é um exemplo de utilização de *chroma key*.

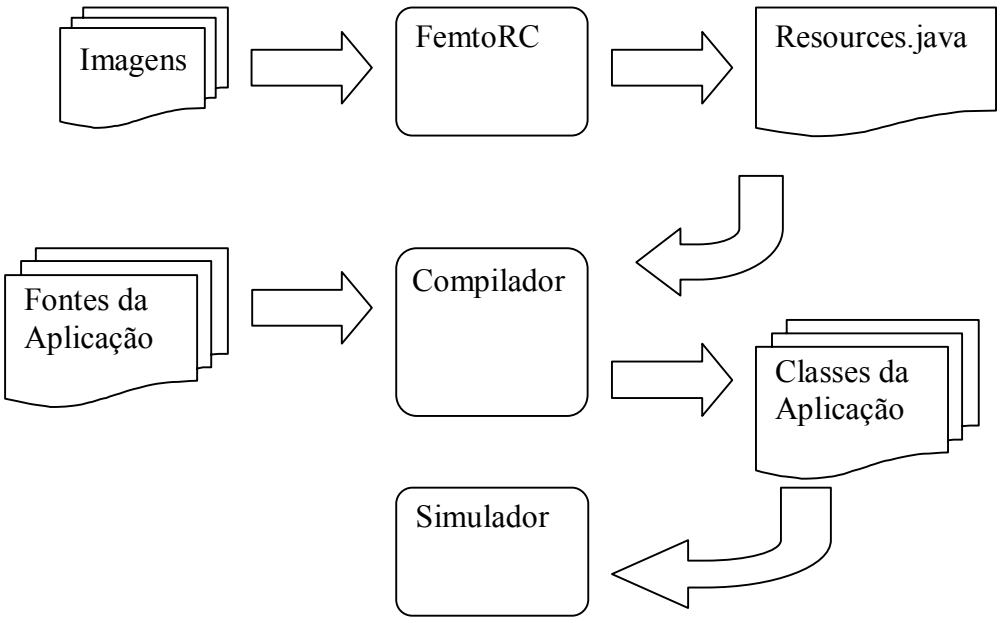

Figura 3.3 – Fluxo de desenvolvimento com a biblioteca FemtoGraphics

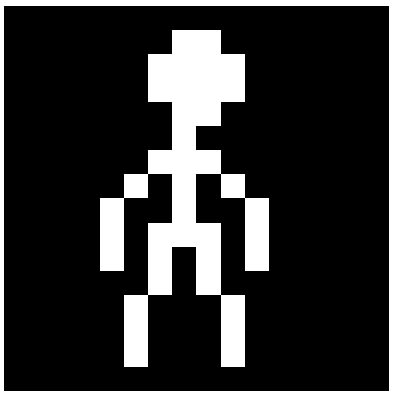

Figura 3.4 – Imagem com região trasparente representada pela cor preta

A classe Screen é composta por um objeto FemtoImage, que representa o frame buffer em si. Só deve haver um objeto Screen no sistema, e este objeto é acessado através do método FemtoSystem.screen(). Um sistema embarcado pode ter dois displays gráficos, mas a implementação da biblioteca não sofreria grande impacto para suportar

mais de um display, então esta possibilidade foi ignorada por não trazer qualquer valor adicional ao trabalho.

3.5 Biblioteca Otimizada para Economia de Energia

Para reduzir o consumo de energia foram estudadas duas técnicas, decodificação RLE e compilação das imagens na forma de operandos imediatos de instruções *sipush*. Estas técnicas foram utilizadas visando reduzir o número de acessos à memória. Como a quantidade de acessos à memória é um fator importante no consumo de energia, ao reduzi-los é possível reduzir também o consumo de energia, contanto que o procedimento adotado para tanto não implique em maior consumo de energia que os acessos à memória que foram eliminados.

A razão para o uso somente da codificação RLE, em detrimento de outras técnicas de codificação que atingem taxas bem maiores de compressão, é justamente a necessidade de reduzir o número de acessos à memória. Por exemplo, a codificação de Huffman utiliza uma tabela auxiliar, onde é mantido o código. Desta forma, para decodificar um valor são necessários dois acessos à memória. Ou seja, um acesso é feito à cadeia de bits codificada e um acesso é feito ao código. Além disso, a codificação é feita no nível de bit, o que implica em diversas operações de deslocamento que têm custo significativo em software.

3.5.1 Compressão RLE

A primeira técnica consiste na utilização da compressão RLE [SAL 2005]. Este tipo de compressão reduz os acessos à memória, desde que as imagens tenham pixels de cores repetidas em seqüência. Como resultado tem-se menos leituras da memória através do *opcode* iaload. Na implementação feita neste trabalho a compressão é aplicada em cada linha separadamente. Isto foi feito para simplificar a implementação do recorte das imagens, no caso de uma imagem ultrapassar os limites da tela. Atualmente a biblioteca não implementa recorte das imagens, mas a forma como a biblioteca foi escrita permite facilmente incorporar este recurso de forma eficiente. Este recurso não tem relação com consumo de energia, mas é necessário em interfaces gráficas com o usuário e em jogos.

A compressão das imagens da aplicação é feita pelo FemtoRC, que é responsável por tranformar as imagens em arrays bidimensionais em forma de código Java. No caso da compressão RLE, os arrays são gerados com tamanhos de linha diferentes, dependendo da compressão que foi atingida para a dada linha. Uma variável no arquivo gerado indica se as imagens estão comprimidas.

Na implementação da biblioteca FemtoGraphics, a compressão RLE usa valores negativos para representar contagens de repetições, limitando a quantidade de cores representáveis a 15 bits, ou 32768 cores. Entretanto, a técnica pode ser usada sem esta limitação e com maior número de cores. A escolha de usar valores negativos é devida ao fato de que o tipo de dados *short* foi usado, e números inteiros em Java têm sinal.

O FemtoRC, descrito na seção anterior, quando executado com a opção de compressão, produz como saída um arquivo fonte Java com todas as imagens comprimidas em RLE. Este arquivo deve ser compilado junto com a aplicação. Após isso, a biblioteca de tempo de execução irá descomprimir as imagens no momento de transferi-las para a tela. O custo em tempo de execução desta operação é baixo, uma vez

que a compressão RLE é bastante simples. Além disso, o número de acessos à memória é reduzido, tornando esse método um pouco mais eficiente que a transferência de imagens sem compressão.

Um benefício extra da compressão RLE ocorre no uso de *chroma keys*. O teste que determina se um pixel deve ser copiado para a imagem de destino pode ser feito apenas uma vez, caso o pixel se repita, pois o resultado do teste será o mesmo. Isto significa menos comparações e menos desvios na execução do código, aumentando a eficiência e ajudando a manter o pipeline tão cheio quanto possível.

Entretanto, a compressão RLE pode resultar em menor desempenho algumas vezes, dependendo da imagem sobre a qual ela opera. Imagens nas quais as cores não se repetem em seqüência freqüentemente terão uma taxa de compressão muito baixa, podendo inclusive gerar uma versão "comprimida" maior que a imagem original. Isto ocorre, por exemplo, se os valores de cor dos pixels estiverem acima do limiar definido e os mesmos não se repetirem com a freqüência necessária.

3.5.2 Operações com operandos imediatos

A segunda técnica estudada para diminuir a potência reduz os acessos à memória colocando os valores dos pixels da imagem diretamente como operando imediatos de operações sipush. A operação sipush coloca um valor do tipo short no topo da pilha. Uma técnica similar é utilizada em [HAP 2001] para aumentar o desempenho em aplicações gráficas. Hapgood utiliza geração dinâmica de código para eliminar acessos a arrays, aumentando o desempenho nas operações com imagens muito utilizadas. De forma diferente, na biblioteca desenvolvida neste trabalho, a geração do código é feita estaticamente, isto é, como parte da compilação da aplicação.

Na versão não otimizada da biblioteca FemtoGraphics, os pixels da imagem são carregados da memória através de operações iaload. Somente a operação iaload implica em duas leituras da memória RAM, uma para a referência do array e uma para o deslocamento, e uma escrita na RAM, do valor lido. Na ROM haverá a leitura do bytecode da operação getstatic, para buscar o endereço de início do array, operações iadd para calcular o deslocamento nas duas dimensões, e por fim o próprio iaload, para buscar o elemento. Ao colocar os pixels como operandos imediatos, tudo isto, incluindo as operações com a RAM, é substituído por uma única operação sipush. Claramente, o número de acessos à memória é reduzido substancialmente.

O processo original de transferir uma imagem para a tela consiste em, para cada pixel, calcular um índice para os dados da imagem fonte, acessar o array para ler o pixel, testar se o mesmo é transparente, se não for transparente calcular um índice para a imagem de destino (o frame buffer da tela), e escrever o pixel nesta imagem. Muitas destas operações podem ser simplificadas ou eliminadas utilizando a segunda técnica. A Figura 3.5 mostra o código original e o código transformado por esta técnica.

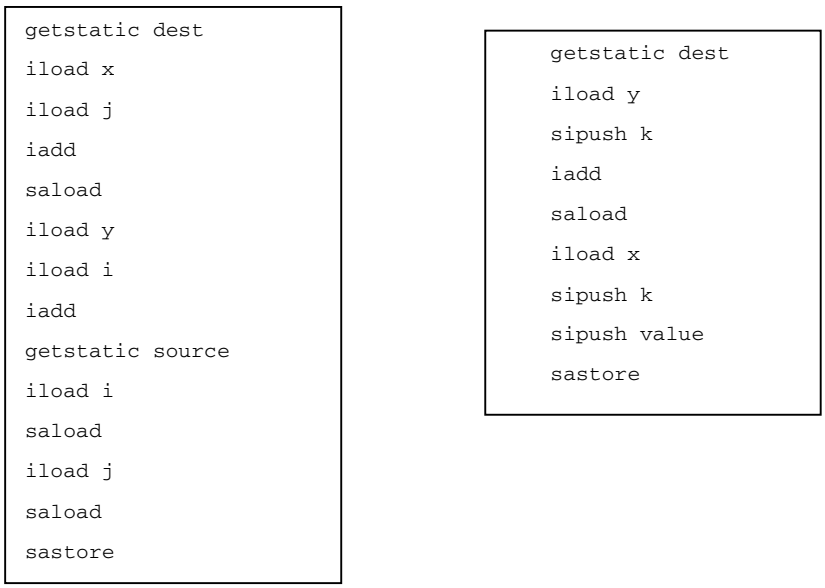

Figura 3.5: Instruções no código original e no código em linha

A primeira simplificação no processo é que ao utilizar operando imediatos não há calculo do indice para a imagem fonte, uma vez que cada pixel corresponde a uma operação sipush. Por exemplo, a seguir tem-se o código original que copia um pixel da imagem fonte para a imagem destino:

```
Dest[y+i][x+j]=source[i][j];
```

Onde x, y, i e j são variáveis. Com operando imediatos, isto se torna:

```
Dest[y+K][x+L]=value;
```

Onde x e y são variáveis e K, L e value são constantes. É importante ver como os bytecodes para essas operações serão gerados pelo compilador. O primeiro resultaria em uma operação sload (short load) para as variáveis, duas operações sadd (short add) para as adições, uma operação getstatic para cada array, duas operações saload (os arrays são bidimencionais) para o array de origem e uma operação saload e uma sastore para o array de destino.

A segunda expressão usa apenas um getstatic, três operações sipush, um para cada constante, duas operações sadd e um sastore (short array store). O número reduzido de acessos à memória tem um grande impacto no consume de energia. São reduzidos os acessos à memória ROM (apenas uma operação sipush contra uma getstatic, duas sload, e uma saload), bem como os acessos à memória RAM (apenas uma operação sastore, para escrever o pixel no frame buffer).

Uma vantagem extra desta técnica é que nenhum código é gerado para pixels transparentes (onde a cor é igual a *chroma key*). Isso resulta em ainda menos uso de memória e código mais eficiente, já que nenhum teste é feito em tempo de execução, e nenhum desvio de fluxo acontece.

4 RESULTADOS

Para avaliar a biblioteca desenvolvida, bem como as técnicas propostas para redução do consumo de energia, duas aplicações foram desenvolvidas, uma versão do jogo Sokoban, adaptada de [NEV 2005] para incluir suporte gráfico, e uma demonstração não-interativa de um aquário, que utiliza em maior escala o recurso de transparência, com o intuito de melhor avaliar o método RLE.

4.1 Ambiente de simulação

A ferramenta CACO-PS (Cycle-Accurate COnfigurable Power Simulator) [BEC 2003a] é um simulador de potência configurável, de código compilado, que executa o circuito ciclo a ciclo. A atividade do circuito é medida pela quantidade de capacitâncias de *gate* que chaveiam durante a simulação do circuito. A quantidade de chaveamentos pode ser multiplicada pela estimativa do custo em energia para realizar um chaveamento, obtendo-se assim o custo em energia de um programa simulado. Além de medir o chaveamento no circuito, também é possível medir o consumo de energia nas memórias ROM e RAM do sistema.

O simulador CACO-PS recebe como entrada uma arquitetura descrita como componentes e suas conexões. Os componentes são descritos em código C, dentro da estrutura do simulador, sendo compilados junto a ele. A descrição dos componentes é feita em duas partes. A primeira descreve sua funcionalidade, ou seja, código C que transforma os sinais de entrada em sinais de saída, realizando algum processamento. A segunda parte descreve o consumo de energia, verificando o número de chaveamentos ocorridos na execução da primeira parte.

Embora o simulador CACO-PS tenha sido concebido para simular qualquer circuito, existem funcionalidades implementadas nele para trabalhar especificamente com a família de processadores Femtojava. Alguns parâmetros do simulador simplificam a simulação de programas feitos para o Femtojava.

Embora o simulador CACO-PS simule um circuito com desempenho superior ao de um simulador VHDL, seu desempenho não é adequado para a simulação de aplicações. Em [NEV 2005] esse problema foi encontrado para simular aplicações mais complexas em termos de tempo de execução. Neves também encontrou limites de memória na versão 16 bits do Femtojava, mas estes foram resolvidos em uma versão para 32 bits.

A dificuldade de simular programas maiores com o CACO-PS levou a uma série de ferramentas para vencer esta limitação. Assim como o simulador CACO-PS, essas ferramentas foram desenvolvidas pelo grupo LSE em outros trabalhos não relacionados ao trabalho desenvolvido nesta dissertação. A descrição aqui visa elucidar o funcionamento do ambiente de simulação.

O ambiente de simulação utilizado para obter o tempo em ciclos e a energia consumida na execução das aplicações desenvolvidas consiste de várias ferramentas. A Figura 4.1 mostra o fluxo de dados do sistema, bem como as ferramentas utilizadas.

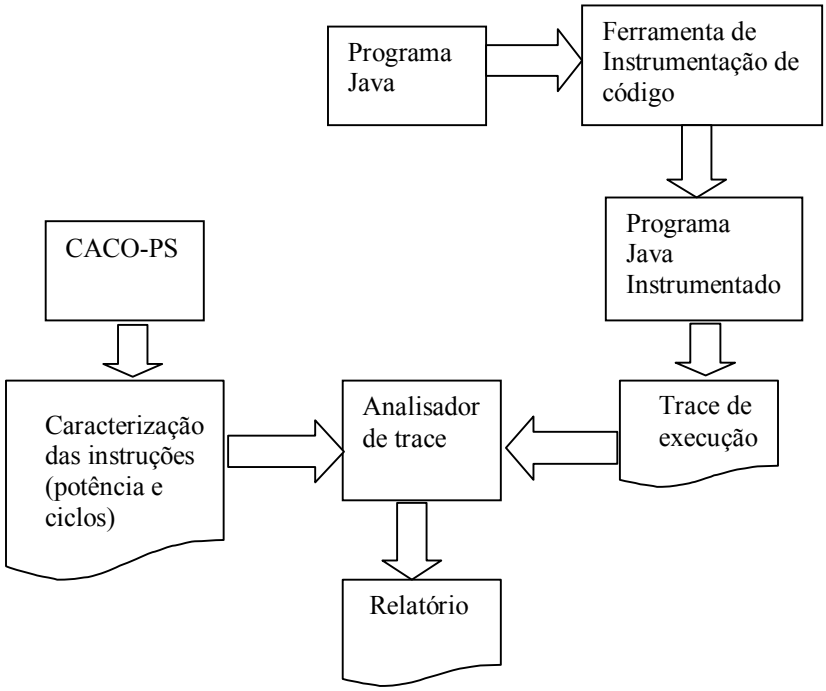

Figura 4.1: Ambiente de simulação

O conjunto de ferramentas conta com um relatório de caracterização das instruções do Femtojava quanto ao número de ciclos e o consumo de energia, gerado a partir de simulações no CACO-PS. Uma outra ferramenta faz a instrumentação do código da aplicação do usuário, gerando uma aplicação instrumentada, capaz de gerar um trace de sua execução. Utilizando a caracterização das instruções, uma ferramenta analisa o trace de execução, determinando o tempo em ciclos e o custo em energia da execução do programa.

A partir do número de capacitâncias de gate chaveadas, o analisador calcula a potência em miliwatts(mW) e a energia em Joules(J). Para tanto, o analisador assume que o Femtojava opera a uma freqüência de 50MHz e utilizando Vdd de 3,3V. O cálculo do consumo de energia leva em conta somente a potência dinâmica. Ou seja, no caso da implementação da técnica de operandos imediatos, onde há um aumento substancial da memória de dados, a potência estática não é considerada.

A simulação pressupõe um sistema organizado como na Figura 4.2. Neste sistema, o *frame buffer* é acessado através do mesmo barramento que comunica o Femtojava com a memória de dados. Já a comunicação entre o *frame buffer* e o controlador do *display* é feita através de um barramento dedicado, evitando que as atualizações do *display* interfiram na execução do programa no Femtojava.

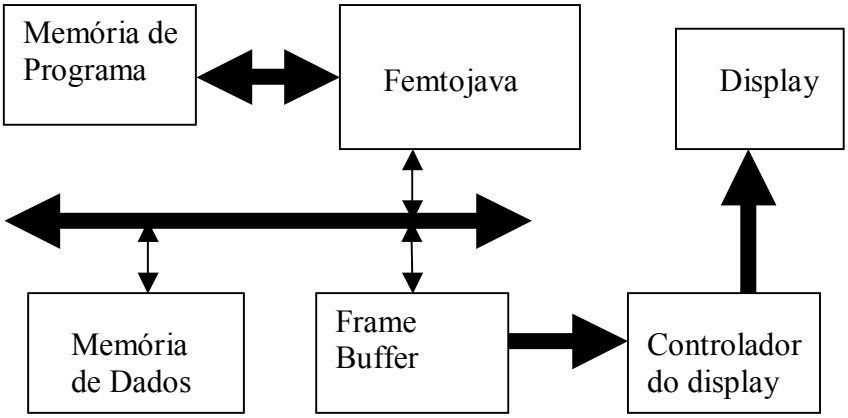

Figura 4.2: Topologia do sistema na simulação

4.2 Tamanho do código desenvolvido

A Tabela 4.1 mostra a complexidade do trabalho desenvolvido na forma de linhas de código fonte Java e tamanho do código objeto. Uma grande parte do código em execução nas aplicações é gerado pela ferramenta FemtoRC. Esta ferramenta é o maior componente desenvolvido, justamente por ser responsável pela geração de código. O código gerado simplificou o código das aplicações, fazendo com que as versões normal, RLE e código imediato pudessem ser testadas sem alterar o código da aplicação a cada teste, e sem que fosse necessário manter versões diferentes da aplicação. Em outras palavras, o código gerado controla o comportamento da biblioteca.

Tabela 4.1: Tamanho do código desenvolvido.

Componente	**Linhas de código Java**	**Código Objeto (em KB)**
FemtoRC	513	9
FemtoGraphics	416	4
Aquário	158	3
Sokoban	473	9
Total	1560	25

A aplicação Sokoban foi adaptada de uma versão sem gráficos, que continha apenas a lógica da aplicação. Algumas partes da aplicação original foram cortadas e cerca de metade do código foi adaptado ou reescrito. A aplicação Aquário foi completamente desenvolvida neste trabalho.

A Tabela 4.2 mostra do tamanho do código objeto da classe Resources, que contém as imagens da aplicação compiladas, para as três implementações da biblioteca. Como se pode ver na tabela, a maior parte do código objeto das aplicações é resultado da compilação das imagens na forma de código Java. Pela tabela é possível ver que a codificação RLE resultou em uma compressão de 47% do tamanho original na aplicação Sokoban e 67% do tamanho original para a aplicação Aquário. O código imediato causou um aumento de 81% na aplicação Sokoban e 171% na aplicação Aquário.

Tabela 4.2: Tamanho do código objeto (só Resources)

Aplicação	**Original**	**RLE**	**Imediato**
Sokoban	21	10	38
Aquario	21	14	57

4.3 Análise dos resultados

Duas aplicações foram utilizadas para avaliar as técnicas propostas. A Tabela 4.3 mostra o número de ciclos e a energia consumida na execução da aplicação Sokoban. A Tabela 4.4 mostra o número de ciclos, instruções e energia consumida na aplicação Aquário.

Na aplicação Sokoban, a implementação que usa compressão RLE não produziu bons resultados. A explicação para este resultado é que a implementação da descompressão utilize um laço de controle adicional, tornando o uso do pipeline do Femtojava menos eficiente.

Tabela 4.3: Resultados da aplicação Sokoban com o uso da biblioteca de gerência de memória em software

	Original	RLE	Imediato
Ciclos (x1000)	2.766.422	2.603.266	1.012.032
Instruções (x1000)	2.272.001	2.137.813	846.081
Energia (J)	1,488	1,403	0,564

Tabela 4.4: Resultados da aplicação Aquário com o uso da biblioteca de gerência de memória em software

	Original	RLE	Imediato
Ciclos (x1000)	468333	546815	212815
Instruções (x1000)	387018	453205	184230
Energia (J)	0,226	0,265	0,109

O método de operandos imediatos produziu resultados excelentes devido ao fato de que sua utilização removeu algumas operações que requerem gerência de memória (o acesso a arrays). Como o sistema de gerência de memória do Femtojava é implementado em software, operações que dependem deste tornam-se bastante caras em termos de desempenho e consumo de energia. Então, qualquer redução nesta area resulta em ganhos substanciais em termos de energia e tempo de execução. Por isso, o tempo de execução ficou em 36% do tempo original e o custo em energia foi reduzido para 38% do custo original.

Na aplicação Aquário os ganhos foram um pouco menores, mas ainda assim atraentes. A versão com operandos imediatos utiliza 45% do tempo de processador utilizado pela versão original, e 48% da energia original. Na versão RLE houve um

resultado inesperado. A versão RLE tem desempenho levemente inferior, tanto em potência quanto em energia.

Como a utilização do sistema de gerência de memória mascarou os resultados obtidos, por ser responsável por grande parte do consumo de energia das aplicações, foram realizados testes supondo custo zero na gerência de memória, de forma a determinar o real ganho das técnicas propostas. Também assim é possível analisar qual seria o ganho destas técnicas se o sistema de gerência de memória tivesse custo baixo, como por exemplo, se fosse implementado em hardware, reduzindo o impacto na aplicação. A Tabela 4.5 mostra os resultados neste cenário para aplicação Sokoban, enquanto a Tabela 4.6 mostra os resultados para a aplicação Aquário.

Os ganhos com a técnica de operandos imediatos se confirmam na aplicação Sokoban quando se retira o custo do sistema de gerência de memória. Porém, na aplicação Aquário, os resultados não são tão bons. A versão de operandos imediatos utiliza 77% do tempo do tempo da versão original, e 69% da energia da versão original. Já a versão RLE foi inferior à versão original na aplicação Aquário, e teve praticamente o mesmo desempenho da aplicação Sokoban original.

Tabela 4.5: Resultados da aplicação Sokoban supondo custo zero na gerência dinâmica de memória

	Original	RLE	Imediato
Ciclos (x1000)	869233	868628	265148
Instruções (x1000)	713991	712514	221722
Energia (J)	0,542	0,514	0,167

Tabela 4.6: Resultados da aplicação Aquário supondo custo zero na gerência dinâmica de memória

	Original	RLE	Imediato
Ciclos (x1000)	11531	12006	8963
Instruções (x1000)	6914	7387	4337
Energia (J)	0,049	0,052	0,034

Os resultados indicam que as técnicas propostas apresentam resultados melhores quando são aplicadas para reduzir o custo de operações caras, como as operações da biblioteca de gerência de memória. Embora os resultados ainda sejam bons na ausência do sistema de gerência de memória, a combinação com outras técnicas é desejável.

As técnicas propostas podem ser aplicadas, da forma descrita neste trabalho (estaticamente), principalmente a aplicações como jogos, onde as imagens são definidas antes da execução. Em aplicações onde as imagens são construídas em tempo de execução, como em interfaces gráficas com o usuário, estas técnicas podem ser adaptadas para execução dinâmica, por exemplo, na forma de geração de código em tempo de execução.

Os resultados deste trabalho não estão confinados ao microprocessador Femtojava. Nada do que foi desenvolvido é conceitualmente específico para o Femtojava, podendo ser facilmente adaptado para outros processadores. Também cabe lembrar que o ganho em desempenho e potência pode ser reproduzido em outras aplicações.

Acredita-se que os resultados apresentados neste capítulo seriam mais expressivos se combinados com técnicas em hardware. Porém, este trabalho visou explorar apenas técnicas em software, tendo relativo sucesso neste sentido.

5 CONCLUSÃO

Este trabalho demonstrou duas técnicas de software para reduzir o consumo de energia em sistemas embarcados com interface para um display gráfico. A técnica de compressão RLE não mostrou bons resultados, mesmo em aplicações onde se esperava resultados satisfatórios. Isto se deve principalmente ao fato do algoritmo depender de um laço de execução extra, que associado à utilização de uma versão do microcontrolador Femtojava sem predição de saltos, causa péssimo desempenho do pipeline, tornando a técnica pouco efetiva.

Já a segunda técnica se provou mais atraente, pois além de desenrolar laços estaticamente, ainda evita acessos extras à memória, colocando os dados como operandos imediatos de operações sipush. Porém, esta técnica tem suas limitações. Ela não permite recortes na imagem, que são necessários quando uma imagem está parcialmente escondida por uma das bordas da tela. Além disso, nem todas as imagens podem ser transformadas em código efetivamente, pois o tamanho do código pode ficar proibitivo. Cabe salientar que o espaço necessário em memória é maior utilizando esta técnica, se contabilizarmos memória de programa e memória de dados juntas.

Além da análise das técnicas, uma contribuição importante deste trabalho é a biblioteca de classes desenvolvida, FemtoGraphics, junto com sua ferramenta de compilação de imagens para código, o FemtoRC. Esta biblioteca não tem nenhuma dependência na biblioteca padrão da linguagem Java, o que permite que seja usada com um mínimo de suporte. Isto é especialmente importante no contexto de sistemas embarcados.

Entre as possibilidades de trabalhos está a implementação da técnica de geração de código de forma dinâmica, determinando em tempo de execução quais imagens devem ser transformadas em código, de forma a deixar o processo transparente para o usuário da biblioteca. Como o Femtojava apresenta uma arquitetura de Harvard, este tipo de técnica não seria tão facilmente implementada. Outro trabalho futuro importante é a implementação de um sistema de interface com o usuário, com primitivas como janelas e menus.

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…** Washington: IEEE Computer Society, 2001. p. 396-399.

[BEC 2003] BECK FILHO, A.C.S.; CARRO, L. Low Power Java Processor for Embedded Applications. In: IFIP INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI-SOC, 12., 2003, Darmstadt. **Proceedings…** Darmstadt: Technische Universit, 2003.

[BEC 2003a] BECK FILHO, A.C.S.; MATTOS, J.C.B.; 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, São Paulo. **Proceedings…** Los Alamitos: IEEE Computer Society, 2003. p. 349-354.

[BEN 2000] BENINI, L. et al. A survey of design techniques for system-level dynamic power management. **IEEE Transactions on VLSI Systems**, New York, v. 8, n. 3, p. 299-316, June, 2000.

[BOC 2004] BOCCA, A. et al. Energy-Efficient Bus Encoding for LCD Displays. In: ACM/IEEE GREAT LAKES SYMPOSIUM ON VLSI, GLSVLSI, 14., 2004, Boston. **Proceedings…** New York: ACM Press, 2004. p. 240-243.

[CHE 2003] CHENG, W. C.; PEDRAM, M. Chromatic Encoding: a Low Power Encoding Technique for Digital Visual Interface. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2003. **Proceedings…** Los Alamitos: IEEE Computer Society, 2003. p. 694-699.

[CHI 99] CHIASSERINI, C. F.; RAO, R. R. Pulsed battery discharge in communication devices. In: INTERNATIONAL CONFERENCE ON MOBILE COMPUTING AND NETWORKING, MOBICOM, 5., 1999, Seattle. **Proceedings…** [S.l.:s.n.], 1999. p. 101-106.

[CHO 96] CHOW, S.-H. et al. Low-power realization of finite state machines – a decomposition approach. **ACM Transactions on Design Automation of Electronic Systems**, New York, p. 315-340, July 1996.

[CHO 2000] CHOI, B.-D.; KWON, O.-K. Stepwise data driving method and circuits for low-power tft-lcds. **IEEE Transactions on Consumer Electronics**, [S.l.], v. 46, p. 1155-1160, Nov. 2000.

[CHO 2002] CHOI, I.; SHIM, H.; CHANG, N. Low-power color TFT LCD display for hand-held embedded systems. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ISLPED, 2002, Monterey. **Proceedings…** New York: ACM Press, 2002. p. 112-117.

[CHO 2003] CHOI, I. et al. LPBP: Low-Power Basis Profile of the Java 2 Micro Edition. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ISLPED, 2003, Seoul. **Proceedings…** New York: ACM Press, 2003. p. 36-39.

[CRA 97] CRAMER, T. et al. Compiling Java just in time. **IEEE Micro**, [S.l.], v. 17, n. 3, p. 36-43, May/June, 1997.

[DDW 99] DIGITAL DISPLAY WORKING GROUP. **Digital Visual Interface**: versão 1.0. 1999. Disponível em: <http://www.ddwg.org>. Acesso em: 15 jun. 2006.

[DIA 2000] DIAZ-HERRERA, J. L.; MADISETTI, V. K. Embedded systems product lines. In: CSE WORKSHOP, 2000, Limerick. **Proceedings...** [S.l.:s.n.], 2000.

[FUT 2002] FUTUREMARK CORPORATION. **MobileMark**. 2002. Disponível em: <http://www.futuremark.com/products/mobilemark2002/>. Acesso em: 15 jun. 2006.

[GAT 2002] GATTI, F. et al. Low Power Control Techniques for TFT LCD Displays. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURES AND SYNTHESIS FOR EMBEDDED SYSTEMS, CASES, 2002, Grenoble, France. **Proceedings…** [S.l.:s.n.], 2002. p. 218-224.

[GOV 95] GOVIL, K. et al. Comparing algorithms for dynamic speed-setting of a low-power CPU. In: INTERNATIONAL CONFERENCE ON MOBILE COMPUTING AND NETWORKING, MOBICOM, 1995, Berkeley. **Proceedings…** New York: ACM Press, 1995. p. 13-25.

[HAP 2001] HAPGOOD, B. Self-modifying Code. In: **Game Programming Gems 2**. Hingham: Charles River Media, 2001.

[HAV 99] HAVINGA, P. J. M.; SMIT, G. J. M. Octopus embracing the energy efficiency of handheld multimedia computers. In: INTERNATIONAL CONFERENCE ON MOBILE COMPUTING AND NETWORKING,

MOBICOM, 5., 1999, Seattle. **Proceedings…** New York: ACM Press, 1999. p. 77-87.

[HIC 97] HICKS, P. et al. Analysis of power consumption in memory hierarchies. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ILSPED, 1997, Monterey. **Proceedings…** New York: ACM Press, 1997. p. 239-242.

[HOL 2004] HOLLEVOET, L. et al. A Power Optimized Display Memory Organization for Handheld User Terminals. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2004, Paris. **Proceedings…** Washington: IEEE Computer Society, 2004. v.3, p. 294.

[HON 98] HONG, I. et al. Power optimization of variable voltage core-based processors. In: DESIGN AUTOMATION CONFERENCE, DAC, 35., 1998, San Francisco. **Proceedings…** New York: ACM Press, 1998. p. 176-181.

[ISH 98] ISHIHARA, T.; YASUURA, H. Voltage scheduling problem for dynamically variable voltage processors. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ISLPED, 1998, Monterey. **Proceedings…** New York: ACM Press, 1998. p. 197-202.

[ITO 2001] ITO, S.A.; CARRO, L.; JACOBI, R.P. "Making Java work for microcontroller applications", **IEEE Design and Test of Computers**, Los Alamitos, California, v. 18, n. 5, p. 100-110. 2001.

[KOL 96] KOLSON, D. J. et al. Optimal register assignment to loops for embedded code generation. **ACM Transactions on Design Automation of Electronic Systems**, New York, v. 1, n. 2, p 251-279, Apr. 1996.

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

[LAW 2002] LAWTON, G. Moving Java into Mobile Phones. **IEEE Computer**, Los Alamitos, v.35, n.6, p. 17-20, June 2002.

[LEE 95] LEE, M. T.; TIWARI, V. A memory allocation technique for low-energy embedded DSP software. In: SYMPOSIUM ON LOW POWER ELECTRONICS, 1995. **Proceedings…** [S.l.:s.n.], 1995. p. 24-25.

[LEE 99] LEE, Y.; KRISHNA, C. M. Voltage-clock scaling for low energy consumption in real-time embedded systems. In: INTERNATIONAL CONFERENCE ON REAL-TIME COMPUTING SYSTEMS AND APPLICATIONS, RTCSA, 6., 1999. **Proceedings…** [S.l.:s.n.], 1999. p. 272-279.

[LU 2001] LU, Y.-H.; MICHELI, G. D. Comparing system level power management policies. **IEEE Design and Test of Computers**, Los Alamitos, v. 18, n.2, p. 10-19, Mar./Apr., 2001.

[MIC 2005] MICROSOFT CORPORATION. **.Net Compact Framework**. 2006. Disponível em: <http://msdn.microsoft.com/mobility/netcf/>. Acesso em: 15 jun. 2006.

[NAI 99] NAIK, K.; WEI, D. S. L. Energy-conserving software design for mobile computers. In: DIMAC WORKSHOP ON MOBILE NETWORKS AND COMPUTING, 1999, Piscataway. [S.l.:s.n.], 1999. p. 237-258.

[NEV 2005] NEVES, B. S. **Gerência Dinâmica de Memória em Aplicações Java Embarcadas**. 2005. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[OCO 97] O'CONNOR, J.M.; TREMBLAT, M. Picojava-I the Java Virtual Machine in Hardware. **IEEE Micro**, [S.l.], v.17, n.2, p. 45-53, Mar./Apr. 1997.

[PAN 97] PANDA, P. R. et al. Architectural exploration and optimization of local memory in embedded systems. In: INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, ISSS, 10., 1997, Antwerp. **Proceedings…** Washington: IEEE Computer Society, 1997. p. 90-97.

[PAS 2003] PASRICHA, S. et al. Reducing backlight power consumption for streaming video applications on mobile handheld devices. In: WORKSHOP ON EMBEDDED SYSTEMS FOR REAL-TIME MULTIMEDIA, ESTIMedia, 2003, Newport. **Proceedings…** [S.l.]: ACM SIGDA, 2003.

[QUA 2005] QUALCOMM. **Binary Runtime Environment for Wireless (BREW)**. Disponível em: <http://brew.qualcomm.com/brew/en/>. Acesso em: 15 jun. 2006.

[RAJ 2000] RAJESWARAN, G. et al. Active Matrix Low Temperature Poly-Si TFT/OLED Full Color Displays: Development Status. **SID Symposium Digest of Technical Papers**, San Jose, v.31, n.1, p. 974-977, May 2000.

[SAL 2003] SALOMON, D. **Data Compression:** The Complete Reference. 3rd ed. New York: Springer, 2003.

[SAL 2004] SALERNO, S. et al. Limited intra-word transition codes: an energy-efficient bus encoding for LCD display interfaces. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ISLPED, 2004, Newport. **Proceedings…** New York: ACM Press, 2004. p. 206-211.

[SAN 2001] SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-Based Design and Software Design Metodology for Embedded Systems. **IEEE Design and Test**, [S.l.], v.18, n.6, p.23-33, Nov./Dec. 2001.

[SHI 95] SHIMAZAKI, Y. et al. An automatic-power-save cache memory for low-power RISC processors. In: IEEE SYMPOSIUM ON LOW POWER ELECTRONICS, 1995. [S.l.:s.n.], 1995. p. 58-59.

[SHI 99] SHIMIZU, N.; NAITO, M. A Dual Issue Queued Pipelined Java Processor TRAJA – Toward na Open Source Processor Project. In: ASIA PACIFIC CONFERENCE ON ASIC, AP-ASIC, 1999, Seoul. **Proceedings…** [S.l.:s.n.], 1999. p. 213-216.

[SHI 99a] SHIN, Y.; CHOI, K. Power conscious fixed priority scheduling for hard real-time systems. In: DESIGN AUTOMATION CONFERENCE, DAC, 36., 1999, New Orleans. **Proceedings…** New York: ACM Press, 1999. p. 134-139.

[SHI 99b] SHIUE, W.-T.; CHAKRABARTI, C. Memory exploration for low-power embedded systems. In: DESIGN AUTOMATION CONFERENCE, DAC, 36., 1999, New Orleans. **Proceedings…** New York: ACM Press, 1999. p. 140-145.

[SHI 2004] SHIM, H.; CHANG, N.; PEDRAM, M. A compressed frame buffer to reduce display power consumption in mobile systems. In: ASIA SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2004, Yokohama. **Proceedings…** Piscataway: IEEE Press, 2004. p. 818-823.

[SHI 2005] SHIM, H.; CHO, Y.; CHANG, N. Frame Buffer Compression Using a Limited-Size Code Book for Low-Power Display Systems. In: WORKSHOP ON EMBEDDED SYSTEMS FOR REAL-TIME MULTIMEDIA, 2005. **Proceedings…** [S.l.:s.n.], 2005. p. 7-12.

[SU 95] SU, C.; DESPAIN, A. Cache design trade-offs for power and performance optimization: a case study. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ISLPED, 1995, Dana Point. **Proceedings…** New York: ACM Press, 1995. p. 63-68.

[SUN 99] SUN MICROSYSTEMS. **Picojava-II Microarchitecture guide.** Santa Clara, 1999.

[SUN 2001] SUN MICROSYSTEMS. **Connected Device Configuration (CDC) and the Foudation Profile**. 2001. Disponível em: <http://java.sun.com/products/cdc/wp/CDCwp.pdf>. Acesso em: 15 jun. 2006.

[SUN 2002a] SUN MICROSYSTEMS. **Programmer's Guide - J2ME Personal Basis Profile**. 2002. Disponível em: <http://java.sun.com/j2me/docs/pdf/PP_Programmer_Guide.pdf>. Acesso em: 15 jun. 2006.

[SUN 2002b] SUN MICROSYSTEMS. **Programmer's Guide - J2ME Personal Profile**. 2002. Disponível em:

<http://java.sun.com/j2me/docs/pdf/PP_Programmer_Guide.pdf>. Acesso em: 15 jun. 2006.

[SUN 2002c] SUN MICROSYSTEMS. **PDA Profile for the J2ME Platform (JSR-75)**. 2002. Disponível em: <http://www.jcp.org/jsr/detail/75.jsp>. Acesso em: 15 jun. 2006.

[SUN 2002d] SUN MICROSYSTEMS. **The CLDC HotSpot Implementation Virtual Machine.** 2002. Disponível em: <http://java.sun.com/products/cldc/wp/CLDC HI WhitePaper.pdf>. Acesso em: 15 jun. 2006.

[SUN 2002e] SUN MICROSYSTEMS. **J2ME Connected, Limited Device Configuration.** 2002. Disponível em <http://jcp.org/aboutJava/communityprocess/.nal/jsr030/>. Acesso em: 15 jun. 2006.

[SUN 2005] SUN MICROSYSTEMS. **Mobile Information Device Profile (MIDP)**. Disponível em: <http://java.sun.com/products/midp/>. Acesso em: 15 jun. 2006.

[SUN 2005b] SUN MICROSYSTEMS. **Java Technology**. Disponível em: <http://java.sun.com/>. Acesso em: 15 jun. 2006.

[TIW 94] TIWARI, V. et al. Power analysis of embedded software: A first step towards software power estimation. **IEEE Transactions on VLSI Systems**, Los Alamitos, p. 437-445, Dec. 1994.

[TIW 98] TIWARI, V. et al. Guarded evaluation : Pushing power management to logic systhesis/design. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, Los Alamitos, p. 1051-1060, Oct. 1998.

[WEI 94] WEISER, M. et al. Scheduling for reduced CPU energy. In: USENIX SYMPOSIUM ON OPERATING SYSTEM DESIGN AND IMPLEMENTATION, 1., 1994. **Proceedings…** [S.l.:s.n.], 1994. p. 13-23.

[WOL 2002] WOLF, W. What is embedded computing? **IEEE Computer**, Los Alamitos, v.35, n.1, p. 136-137, Jan. 2002.

[WUY 96] WUYTACK, S. et al. Transforming set data types to power optimal data structures**. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], p. 619-629, June 1996.

[YAO 95] YAO, F. et al. A scheduling model for reduced CPU energy. In: SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, FOCS, 36., 1995, Milwaukee. **Proceedings…** Washington: IEEE Computer Society, 1995. p. 374-382.