Felipe Zimmermann Homma

EPOS no Cell

Florianópolis – SC

2007/2

Felipe Zimmermann Homma

EPOS no Cell

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação

Orientador:
Prof. Dr. Antônio Augusto Medeiros Fröhlich

BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
UNIVERSIDADE FEDERAL DE SANTA CATARINA

Florianópolis – SC

2007/2

“Felipe Zimmermann Homma”

“EPOS no Cell”

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação

---

Prof. Dr. Antônio Augusto Medeiros Fröhlich
Departamento de Informática e Estatística
Orientador

M.Sc. Rafael Cancian
Banca Examinadora

B.Sc. Danillo dos Santos
Banca Examinadora

Aos meus pais, amigos e amantes.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de abreviaturas e siglas

HD-TV	*High-Definition TV* (TV de Alta Definição),	p. 9
HD-DVD	*High-Definition Digital Video Disc*(Disco de Video Digital de Alta-Definição),	p. 9
IBM	*International Business Machines*,	p. 9
DSP	*Digital Signal Processor* (Processador de Sinal Digital),	p. 9
RISC	*Reduced Instruction Set Computer* (Computador com um Conjunto Reduzido de Instruções),	p. 9
POWER	*Power Optimization With Enhanced RISC*,	p. 9
EPOS	*Embedded Parallel Operating System* (Sistema Operacional Paralelo Embarcado),	p. 9
CBE	Cell Broadband Engine,	p. 10
PPE	*POWER Processing Element* (Elemento POWER de Processamento),	p. 10
SPE	*Synergistic Processing Element* (Elemento Sinergético de Processamento),	p. 10
DMA	*Direct Memory Access* (Acesso Direto à Memória),	p. 10
E/S	Entrada/Saída,	p. 10
EIB	*Element Interconnect Bus* (Barramento de Inteconexão de Elementos),	p. 10
PPU	*POWER Processing Unit* (Unidade POWER de Processamento),	p. 10
FPU	*Floating-Point Unit* (Unidade [Aritmética] de Ponto-Flutuante),	p. 10
VXU	*Vector Extension Unit*,	p. 10
SPU	*Synergistic Processing Unit* (Unidade Sinergética de Processamento),	p. 11
MFC	*Memory Flow Controller* (Controlador de Fluxo de Memória),	p. 11
RAM	*Random Access Memory* (Memória de Acesso Aleatório),	p. 11
LS	*Local Storage* (Memória Local),	p. 12
MMU	*Memory Management Unit* (Unidade de Gerência de Memoria),	p. 13
BIC	*Broadband Interface Controller*,	p. 13
IOIF	*Input/Output Interface*,	p. 13
AC	*Address Concentrators* (Concentradores de Endereços),	p. 13

Resumo

Neste trabalho foi realizado uma tentativa de porte do sistema operacional EPOS para a arquitetura Cell BE. As características propostas pela arquitetura são ideais para o desenvolvimento de sistema dedicados para mídia, especialmente reprodutores áudio-visuais e mesmo para aplicações científicas. Além disso, é feito um esboço de uma proposta para o tratamento da complexidade de programação (presente também em outras arquiteturas semelhantes). Infelizmente o porte não foi completado para o fechamento deste documento, mas boa parte das funcionalidades de baixo nível foram implementadas e testadas em um ambiente simplificado.

Introdução

Sistemas para mídia e entretenimento demandam um poder de processamento que vem aumentando com o passar dos anos. Conforme aumentam a quantidade, tipos e qualidade do conteúdo a ser entregue ao consumidor final, mais processamento é exigido dos equipamentos. Requisitos de tempo-real e de alta-disponibilidade, baixo consumo e dissipação de potência são comuns pela natureza dos produtos gerados por esses equipamentos (áudio e vídeo, por exemplo).

Além dos video-games, equipamentos domésticos num futuro próximo, como televisores (HD-TV) e reprodutores (HD-DVD, *Blu-Ray*), também elevarão os requisitos de processamento do *hardware*. Como isso em mente, Sony, Toshiba e IBM criaram um consórcio para estabelecer uma nova arquitetura de processador, que será utilizada em seus equipamentos, compreendendo desde televisores até servidores para computação de alto desempenho.

A arquitetura proposta pela IBM envolve um processador com múltiplos núcleos heterogêneos, ligados por um dispositivo de interconexão de alto desempenho. Com essa abordagem, similar às encontradas em DSPs, é esperado um aumento na relação processamento/-potência (KONGETIRA; AINGARAN; OLUKOTUN, 2005; BARROSO et al., 2000; STANKOVIC, 1996; OLUKOTUN et al., 1996; CONSTANTINOU et al., 2005).

Atualmente, qualquer sistema operacional que funcione em arquitetura POWER (32 e 64 bits) pode executar no Cell sem, no entanto, aproveitar todo o processamento que ele pode oferecer (SONY COMPUTER ENTERTAINMENT INC., 2005). Uma versão funcional de Linux já está disponível para arquitetura mas seu objetivo parece ser entregar uma plataforma para computação de alto-desempenho.

No capítulo 1 é realizada uma breve descrição sobre processadores da arquitetura Cell BE. No capítulo 2 um levantamento um pouco mais profundo da arquitetura, isolando os componentes necessários para o porte do EPOS bem como o relato sobre as atividades sobre o seu desenvolvimento.

A abordagem com núcleos híbridos da arquitetura Cell acarreta em complexidades no desenvolvimento, com algumas soluções propostas para amenizá-las. No capítulo 3 são apresen-

tadas algumas dessas propostas e no capítulo 4 uma sugestão para o tratamento dessa complexidade de programação.

1 *Cell Broadband Engine*

1.1 Componentes Da Arquitetura

Diferente das tecnologias de DSP que são uma coleção de processadores diversos agrupados o Cell se propõe a fazer essa coleção trabalhar de maneira sinergética.

Os paradigmas de operação, os formatos de dados e as semânticas são consistentes e todos os processadores compartilham o modelo de proteção e de resolução endereço da memória.

Desta forma estabeleceu-se um conjunto de um processador PowerPC e alguns processadores vetoriais simples. O PPE destinado a operação do sistema operacional e controle de execução e as SPE otimizadas para o processamento de dados.

Através de um mecanismo de DMA cada SPE compartilha as funções tradução de endereço e proteção de memória do resto do sistema. PPEs, SPEs, interface de memória e controlador de E/S comunicam-se através de uma rede de alto desempenho chamada EIB.

A opção por núcleos heterogêneos serve para atacar a barreira de potência, a arquitetura de memória (tanto do PPE quanto das SPEs) para atacar a barreira de memória e recursos para permitir pipelines maiores (tudo o que costuma ser feito em hardware de execução especulativa deve ser feito pelo compilador, ou pelo programador) tentam minimizar a barreira de freqüência.

1.1.1 *Power Processing Element* (PPE)

O PPE (figura 1.1) é composto de um processador *PowerPC* (PPU), *cache L1* e *L2*, unidade de aritmética em ponto-flutuante (FPU) e unidade vetorial AltiVec (ou VXU).

A PPU é um processador *PowerPC AS* de 64-bit, com capacidade de sustentar até duas *threads* simultâneas. A arquitetura, embora compatível com o padrão POWER, foi desenvolvida para atender as demandas do projeto do Cell.

A arquitetura com dois níveis de *cache* tenta amenizar as latências de memória. A

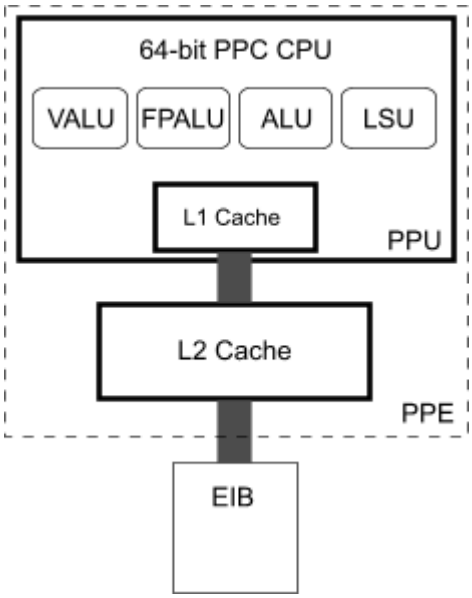

Figura 1.1: Esquemático de um PPE

eliminação da capacidade de execução fora de ordem e mecanismos de "predição" de desvios controlados por *software* tentam diminuir o impacto do tamanho do pipelining, com o objetivo de aumentar a freqüência de operação. No entanto toda a responsabilidade de extrair o paralelismo em nível de instrução fica a cargo do compilador (ou do programador).

A compatibilidade com o padrão POWER permite que qualquer programa escrito para tal padrão possa ser executado sem modificações. No entanto tais aplicações não irão se beneficiar das vantagens de paralelismo oferecidas pela arquitetura.

1.1.2 *Synergistic Processing Element* (SPE)

Um SPE (figura 1.2) é caracterizado pela agregação de uma *Synergistic Processing Unit* (SPU), um bloco de memória privada local , um conjunto de canais de comunicação, bloco de registradores (128 registradores de 128-bits) e um *Memory Flow Controller*(MFC) .

Em termos gerais, a SPU é um processador especializado, o bloco de memória local é uma área de memória dedicada para esse processador e o MFC gerencia as operações de DMA para a troca de dados entre a memória principal e essa memória local (e acesso a dispositivos externos, PPU e outras SPUs).

A super-especialização das SPEs impedem que estas realizem operações normais de um sistema operacional. Elas não tem um mecanismo de virtualização de memória, nem acesso direto a RAM e o suporte a interrupções é restrito.

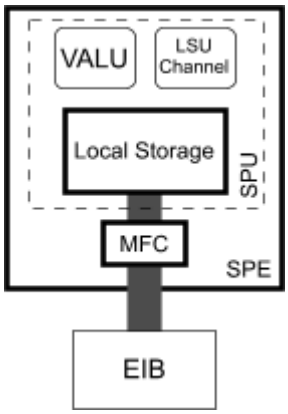

Figura 1.2: Esquemático de um SPE

***Synergistic Processing Unit* (SPU)**

A SPU é um processador vetorial capaz de despachar duas instruções (uma aritmética e uma de memória) por ciclo de execução, sem capacidade de execução fora de ordem. O paralelismo em nível de instrução deve ser planejado e explorado pelo compilador (ou programador) ainda na etapa de compilação.

A atual implementação do *Cell* foi projetada tendo em vista aplicações de multimídia, especificamente um vídeo-game, com grande demanda de cálculos vetoriais de ponto-flutuante de precisão simples. Para diminuir a complexidade do circuito, foram removidos diversos modos de arredondamento não essenciais da unidade aritmética.

Existem alguns registradores de controle mapeados na memória da principal para permitir a comunicação rápida entre as SPU (SPU–SPU) e a PPU (PPU–SPU), que podem ser usados para a construção de mecanismos de sincronização.

Não existem níveis de acesso diferenciados na SPU, no entanto o acesso à SPU só é possível pelo PPU em modo privilegiado.

***Local Storage* (LS)**

Os acessos de memória realizados pela SPU são sempre executadas e tratados relativos a memória local (incluindo a resolução e proteção de endereços). Para acessar a memória principal uma transação DMA deve ser executada, o mesmo ocorrendo com acesso a dispositivos. Uma vantagem desse sistema é o determinismo no tempo de acesso e disponibilidade de dados na memória da SPU.

O bloco de memória local de cada SPU pode ser mapeado e acessado pelo PPU diretamente

na memória principal (através da MMU). O acesso das SPU tanto à memória principal como aos mecanismos de E/S e outros processadores é feito pela EIB através de operações de *DMA* executadas pelo MFC.

Memory Flow Controller **(MFC)**

O MFC é composto por um gerente de *DMA*, responsável pelas transações com a memória principal, e uma MMU, responsável pela tradução e proteção de endereços. Além disso possui uma unidade atômica, que faz atualizações *DMA* atômicas para sincronização entre software executando em varias SPEs e a PPE.

O gerente de *DMA* pode mover fluxo de dados da e para a memória local em paralelo com a execução de programas. possui duas filas de transações uma para as iniciadas pela PPE e outra para as iniciadas pela SPE.

A MMU implementa tradução e proteção usando uma tabela de segmentação e modelo de tabela de página padrão *PowerPC*. Desta forma as transações *DMA* podem englobar todos os endereços de memória do sistema, com tratamento de violações e exceções de lógica sendo aplicados e apresentados como interrupções externas a PPE.

1.1.3 *Element Interconnect Bus* (EIB)

O EIB talvez seja a parte mais importante de todo o projeto do Cell. Ele faz a ligação do PPE, SPEs, E/S e memória, e tem um comportamento que lembra as redes *Token-Ring*.

Possui redes independentes para comandos e para dados. Tem doze portas para elementos, cada um produzindo e consumindo até 16 bytes por ciclo de barramento. Duas dessas portas para o BIC (IOIF0[1] e IOIF1) e uma porta para cada um dos outros componentes.

Os comandos são filtrados através de concentradores de endereço (ACs) que manejam detecção e prevenção de colisão e e certificam-se de que todas as unidades têm um acesso equalitário ao barramento de comando.

O trânsito de dados ocorre em quatro "anéis", cada um conectando todas as portas de dados, em apenas uma direção (dois anéis movem dados no sentido horário e dois no sentido anti-horário). Cada anel pode sustentar até três transferências simultâneas, desde que não ocorra sobreposição.

A largura de banda teórica de cada porta é de 25.6GB/s em cada direção. Os anéis de

[1] ou BIF

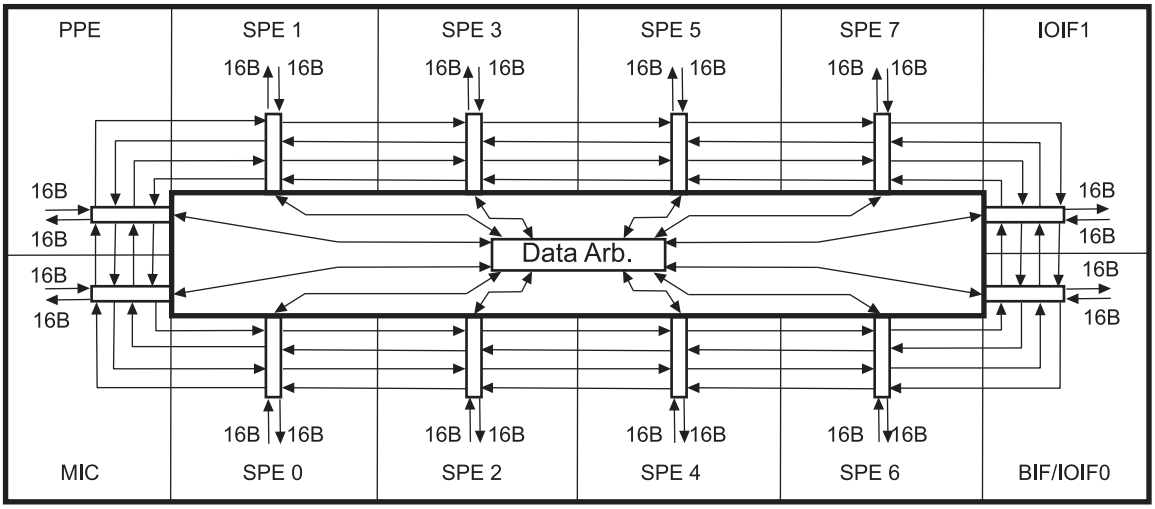

Figura 1.3: EIB - conexões e taxas de transferência

dados podem sustentar 204.8 GB/s para algumas tarefas em um cenário ideal com os quatro anéis executando três transferências concorrentes ao mesmo tempo esse valor pode chegar a 307.2GB/s (teórico).

2 *EPOS no Cell*

2.1 Hardware

A plataforma Cell BE mais acessível no momento é o Playstation 3. Tentativas de aquisição de plataformas de referência foram realizadas perante à IBM sem sucesso.

No Playstation 3 existe uma restrição ao acesso do hardware, que deve ser feito através de uma camada de virtualização (Hypervisor). Diversos registradores tem seu uso permitido somente através de chamadas as rotinas desta camada.

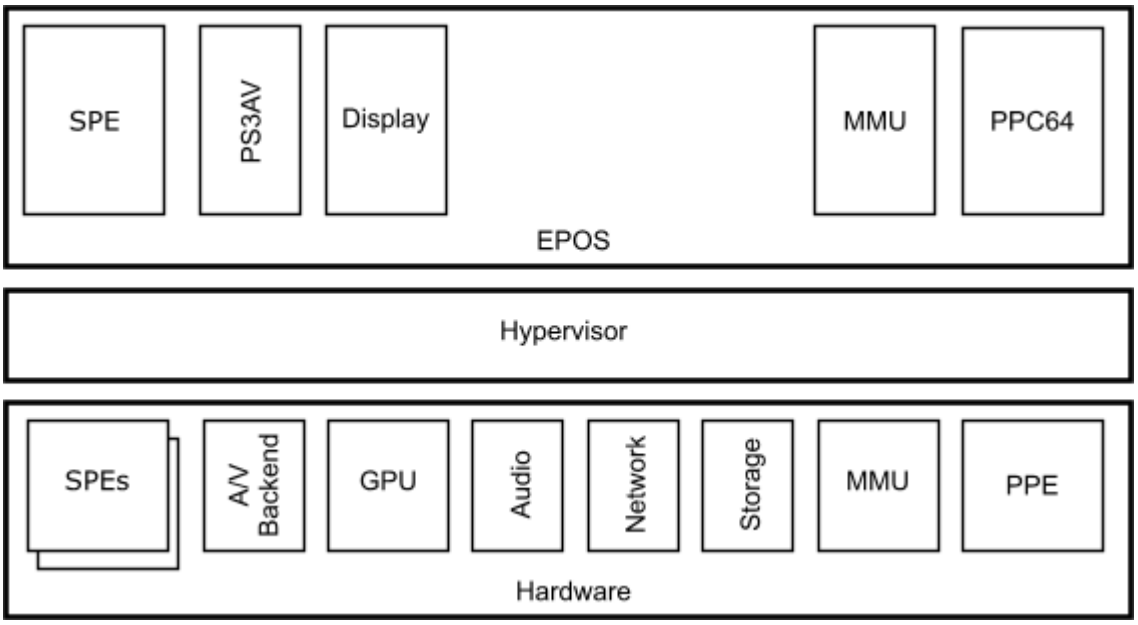

Figura 2.1: Hypervisor e o Sistema Operacional

Além disso, a quantidade de processadores SPE disponíveis para o usuário é menor. Um dos SPE é mantido dedicado ao sistema operacional nativo (*GameOS*) e um outro fica desabilitado, ativado apenas no caso de falha de um dos outros sete.

2.2 Versão para POWER 64 bits

Partindo da versão para POWER32, iniciamos a adaptação das estruturas do sistema operacional para se adequarem as especificações POWER64. Além da diferença do endereçamento de memória maior, vários registradores de controle tem seus mapas de bits completamente diferentes.

Podemos definir as estruturas que foram consideradas importantes nesta versão inicial como sendo:

CPU: que define trocas de contexto.

MMU: controle das facilidades de proteção e virtualização de memória.

PS3AV: um mediador para o sistema de áudio e vídeo do equipamento.

Além destas, outras estruturas importantes como alarmes e relógio de tempo-real serão feitas. Os alarmes utilizam as interrupções do sistema de decrementadores do processador e o relógio é facilmente obtido através de rotinas do Hypervisor.

Rotinas de término do uso dos recursos devem ser executadas. Isso foi observado no kernel do Linux e a razão imaginada, já que nenhuma documentação foi encontrada, é liberar os recursos do sistema de compartilhamento do Hypervisor.

2.2.1 CPU

A abstração de CPU cuida do chaveamento de contexto, início e desligamento do processador e dá acesso aos registradores de controle. Outras responsabilidades incluem facilidades para infraestrutura de sincronização de *threads* e controle de interrupções.

Para as rotinas de sincronização são disponibilizados incrementadores-testadores atômicos, função necessária para construção de estruturas como semáforos e *mutexes*.

Os registradores de propósito geral, bem como qualquer registrador de controle necessário, tiveram sua largura ampliada de 32 para 64 bits, e as funções de salvamento e carregamento do contexto foram alteradas para refletir isso.

Pelo padrão POWER, processos com código objeto em 32-bits pode ser executado em plataformas 64-bits. O suporte a isto não será implementado agora (talvez nunca seja) pois a complexidade envolvida é considerável.

O processador POWER do Cell tem suporte a *multi-threading* em *hardware*, sendo capaz de sustentar até duas *threads* simultaneamente.

Hypervisor

O Hypervisor é uma camada de virtualização presente no Playstation 3. Essa camada também existe em alguns servidores da IBM para permitir múltiplos sistemas operacionais executando simultânea e independentemente na mesma máquina. No caso do console, ele existe para proteger o sistema de tentativas de pirataria de software.

O Hypervisor do Playstation 3 permite que, além do sistema nativo, um segundo sistema operacional seja instalado e, diferente do sistema em servidores IBM, apenas um deles está ativo em determinado instante.

Embora quase todo o contexto da CPU seja transparente, ele isola boa parte do controle da MMU (escrita na tabela de endereço, mapeamento dos segmentos e paginação de memória e de E/S) e virutaliza várias das funcionalidades dos dispositivos (como interfaces de rede e o subsistema de áudio e vídeo) através de *system calls* ao um *kernel* privilegiado.

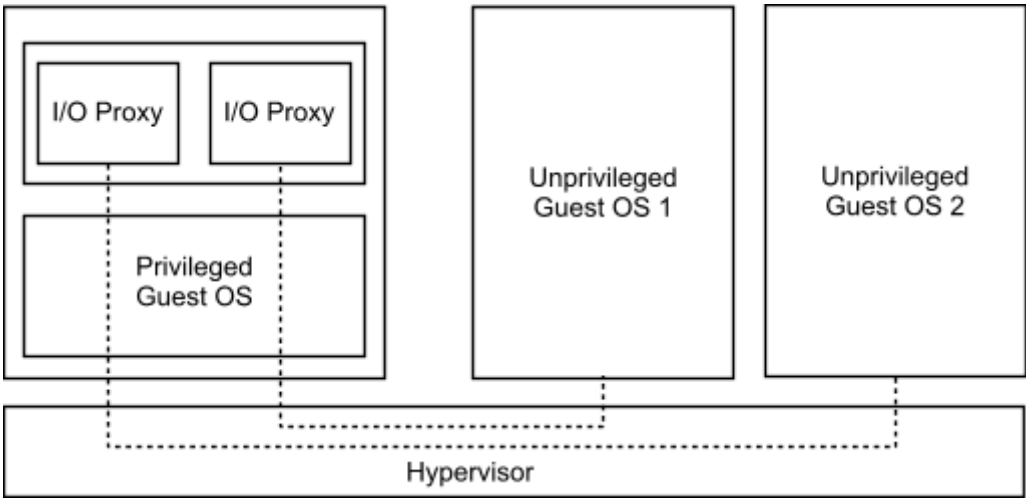

Figura 2.2: Caminho das chamadas ao Hypervisor

A documentação das rotinas do Hypervisor, com suas assinaturas e índice de chamada são escassas, restringindo-se aos comentários (escassos) no código-fonte do *kernel* do Linux.

Então, ao invés de criar um sistema novo para acessar as rotinas do Hypervisor, optou-se por utilizar o que está implementado no *kernel* do Linux para o Playstation 3. As devidas adaptações foram aplicadas para isolar as dependências com o restante do código e para a adequação ao sistema de namespaces do EPOS. No anexo D estão listadas as chamadas conhecidas ao Hypervisor instalado no Playstation 3.

Interrupções

O sistema de interrupções do Cell segue as especificações POWER e compartilha semelhanças com o porte já feito. Algumas modificações se fizeram necessárias apenas para acrescentar ou remover aquelas que fazem parte da implementação especifica do padrão.

O controle de ativação e desativação das interrupções fica a cargo da abstração de CPU (seção 2.2.1). Cada partição tem sua própria tabela de interrupções, não sendo necessário ao sistema operacional ficar gerenciando o chaveamento dela.

2.2.2 MMU

Com a implementação original para POWER32, as facilidades em *hardware* da MMU não são utilizadas, optando-se por uma implementação em *software*. Sem a implementação em hardware, perde-se todas as vantagens da arquitetura de virtualização e proteção de memória oferecidas.

Num *PowerPC* 64 a arquitetura de memória pode ser hierarquizada em três níveis: segmentação, paginação e deslocamento dentro da página. Essa hierarquia só existe se a tradução de endereços (virtualização de memória) estiver ligada. Esse mecanismo pode ser habilitado independentemente para instruções ou dados.

A abstração de MMU no EPOS é feita herdando da classe do sistema MMU_Common que pode ter a quantidade de bits destinado a cada nível de memória configurada por parâmetros de gabarito[1].

Assim como a tabela de interrupções, o Hypervisor disponibiliza o controle da MMU independente para cada partição.

Espaços de endereçamento

Caso o mecanismo de memória virtual esteja ligado, três espaços de endereçamento são criados na arquitetura de memória:

- Espaço de endereçamento real (Real Address Space) que engloba toda a memória física instalada, memórias locais e qualquer memória de dispositivo mapeadas. Até 2^{42} *bytes* podem ser endereçados.

[1] gabarito é como foi traduzido o termo *template* na edição brasileira do livro de referência de C++ (STROUSTRUP, 2000)

- Espaço de endereçamento virtual (Virtual Address Space) espaço que surge pelo mecanismo de segmentação. Até 2^{65} *bytes* podem ser endereçados neste espaço.

- Espaço de endereçamento efetivo (Effective Address Space) esse é o espaço em que todos os endereços de memória de um programa (que utilizam o mecanismo de tradução) se referem. O total de endereços é de 2^{64} entradas (*bytes*).

A tradução de endereços inicia com a requisição de endereço efetivo. A partir do endereço efetivo, através do mecanismo de segmentação obtém-se o endereço virtual, que é submetido a um mecanismo de paginação que através do índice da página e o deslocamento dentro dela obtém o endereço real.

A tradução de endereços virtuais para reais é feita através de uma estrutura mantida em memória pelo sistema operacional, chamada "Tabela de Páginas"[2]. Esta tabela é uma tabela *hash* com mapeamento invertido que mantém entradas de 128 bits codificando diversas informações sobre as páginas. Entre as informações presentes está o tamanho da página que é configurável (64 KB, 1MB ou 16 MB).

A modelagem da abstração para a MMU implementada no EPOS é incompatível com o que foi apresentado anteriormente. Ela trabalha considerando que o endereço real e não uma estrutura de dados que será utilizada para gerá-lo é armazenado na tabela de páginas. Tornar o tipo de dado esperado configurável por parâmetro de gabarito para a classe MMU parece ser a maneira menos intrusiva.

2.2.3 PS3AV

Esta classe foi criada para tornar acessíveis as rotinas de escrita na saída gráfica do equipamento. Ela controla funções primitivas do subsistema de vídeo. As rotinas de desenho são necessárias para poder disponibilizar algum tipo de saída, no caso texto, para os programas.

O sistema de áudio e vídeo fica protegido atrás do Hypervisor e é acessado através de uma porta de comunicação serial virtual (vUART) que pode transmitir para o sistema pacotes padronizados de comandos e receber do sistema pacotes padronizados de informações.

Por enquanto apenas uma inicialização simples, utilizando uma resolução pré-definida é feita. A auto-detecção do tipo de saída (tela) disponível e configurações diversas a partir das opções descobertas são trabalhos futuros.

[2] *Page Table*

Para ativar o subsistema de vídeo, inicialmente chamamos o comando "lv1_gpu_open". Isso disponibiliza o hardware de vídeo para a partição lógica onde estamos executando o sistema operacional. Após isso, criamos uma vUART e então através dela disparamos alguns pacotes de inicialização (ver anexo E), primeiro o subsistema de vídeo, depois o subsistema de áudio e então o sistema de áudio-vídeo (AV).

Neste ponto o sistema AV deve estar apto a receber os comandos de configuração (ver anexo F) que definem a resolução, freqüência e tipo da tela acoplada ao equipamento.

vUART

O wrapper para a vUART foi criado para encapsular o meio que o Hypervisor fornece para comunicação com alguns dispositivos. É através de comando de escrita por esse meio que se inicializa e configura boa parte do subsistema de vídeo.

O Hypervisor fornece rotinas para configuração, leitura e escrita para a porta. Na classe construída as seguintes rotinas foram utilizadas:

- lv1_get_virtual_uart_param
- lv1_write_virtual_uart
- lv1_read_virtual_uart

A primeira rotina permite verificar se o canal está disponível. No caso do sistema de áudio e vídeo é o canal zero. As outras duas escrevem e lêem pela porta e dessa forma pode-se configurar o dispositivo e verificar seu estado.

Os comandos que são transmitidos pela VUART para o sistema AV começam pelo seguinte preâmbulo:

ordem	tamanho (*bytes*)	significado
1	2	versão do pacote
2	2	número de *bytes* que seguem
3	4	a identificação do comando
...	...	os parâmetros do comando

Tabela 2.1: Preâmbulo dos pacotes de comando da vUART

Display

O Display fornece rotinas para a escrita na tela. No futuro espera-se que seja capaz de pintar *pixels* individuais ou ainda fazer operações mais complicadas com o *framebuffer* (como *blitting* por exemplo).

O papel dessa classe é tornar o uso *framebuffer* disponível para os outros componentes do sistema, como consoles de texto. O que ela faz é mapear uma parte da memória de vídeo no espaço de endereçamento real para ser utilizado como *framebuffer* e disponibiliza rotinas em alto nível para o resto do sistema.

Por enquanto a única funcionalidade provida pela classe são as rotinas para escrita de texto na tela, emulando um terminal de texto simples.

2.3 Suporte aos SPEs

Os SPEs funcionam como máquinas independentes. O controle deles é feito através de escrita em registradores mapeados na memória, o que nos leva a mais um recurso acessado pelo Hypervisor.

Ao menos um primeiro programa deve ser carregado por um elemento externo (PPE ou outro SPE). Depois o próprio SPE pode gerenciar o controle de fluxo e carga de novos dados e instruções para si.

Através do Hypervisor é possível criar versões lógicas da SPEs, provavelmente para permitir o compartilhamento delas entre as partições lógicas do processador.

Através da rotina "lv1_construct_logical_spe" é possível obter os uma série endereços relacionados a uma SPE. Depois através de rotinas mapeadas através da abstração de MMU um remapeamento dos endereços para endereços efetivos deve ser feito.

Toda a comunicação com o SPE passa pelo MFC. Em resumo, o MFC é um gerente de transações DMA. Ele ordena os comandos (com um sistema de prioridades, *tags* e execução fora-de-ordem) invocados pelo próprio SPE e também os comandos vindos de outros elementos do sistema, como outros SPEs ou o PPE. Existem canais de mensagens que ligam o SPE diretamente ao PPE, além de canais comunicação mais genéricos e capacidade de transações atômicas.

O MFC possui acesso as MMU compartilhada por todos os elementos no processador, permitindo que ele note excessões em tentativas de escrita ou leitura em memória protegida ou

em páginas que estão em disco (e que permite ao PPE notar o acesso e então carregar a página para a RAM).

2.3.1 SPE

A modelagem da abstração para o SPE (esboço na figura 2.3) segue as informações recebidas pela rotina de criação, adicionando alguns comportamentos desejáveis, como rotinas de criação (ou requisição), controle, interrupções e sincronização.

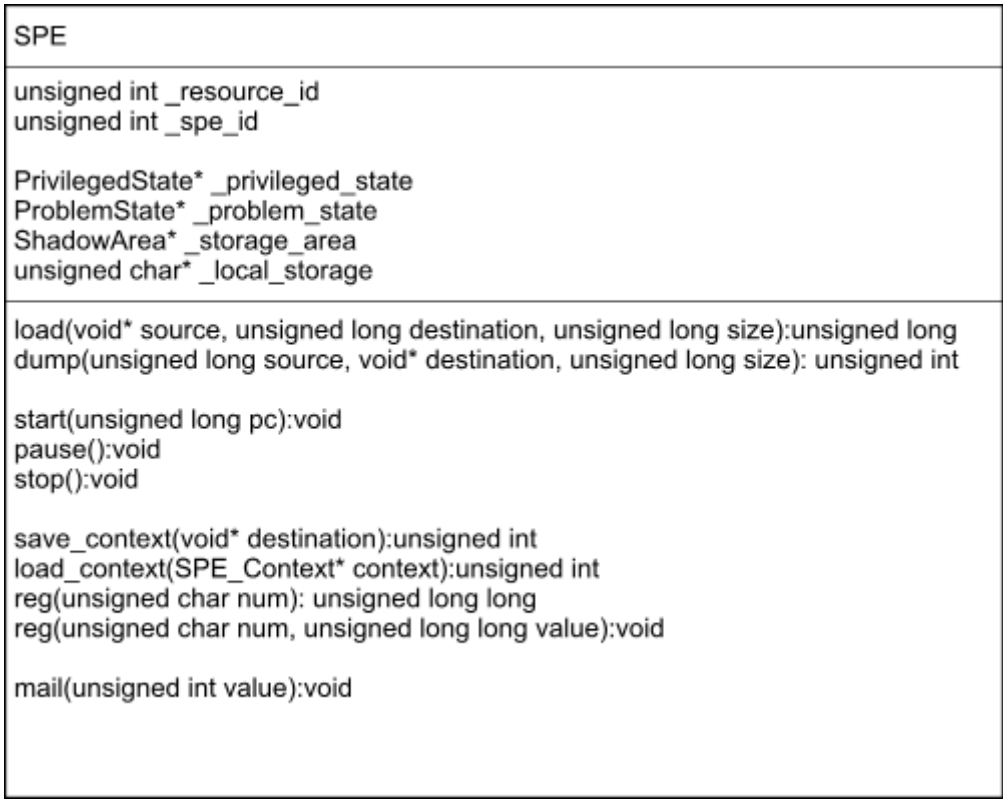

Figura 2.3: Esboço da modelagem da abstração para SPE

Esta estrutura de controle deve estar presente em qualquer processador que se comunique com outro, seja SPE ou PPE. Desta forma muito do processo de criar as transações DMA ficam abstraídas em alguns procedimentos encapsulados na classe.

No Linux os SPEs são tratados como um recurso e não como um processador. A decisão de usar os SPEs como processadores da mesma forma que o PPE ainda precisa ser tomada, mas devido as características dos SPEs é provável que a decisão acabe sendo igual ao projeto do Linux.

A arquitetura diferenciada deles e a dependência do PPE pesam bastante para essa decisão.

Em alguns pontos da documentação é sugerido que os SPEs sejam muito mais co-processadores vetoriais (ou um acelerador) do que um processador independente e completo.

EPOS como uma biblioteca para programas SPE

Analizando a arquitetura de comunicação dos SPE, nota-se que exite uma complexidade considerável, que talvez possa ser melhorada com o uso de uma versão do EPOS para controlar essas operações em um nível um pouco mais alto.

Além de um controle direto sobre as funcionalidades de transação DMA, mecanismos mais elaborados e abstratos poderiam ser construídos, como *Remote Procedure Call* (Chamada Remota de Procedimento, RPC) ou *Remote Object Invocation* (Invocação Remota de Objeto, ROI).

Funções de escrita para a saída padrão, acesso a disco ou qualquer outro periférico, poderiam ser acessíveis através de chamadas diretas ao sistema operacional residente, que se preocuparia com as devidas transações para fora do SPE.

A presença do EPOS também torna possível *multi-threading* nos SPEs, embora o uso de tal recurso possa não trazer beneficio algum. Os SPEs foram projetadas para executar tarefas especificas e aliviar a carga (especialmente de cálculo vetorial) do processador principal.

Mesmo sem as *threads*, o potencial de disponibilizar uma maneira mais amigável de se acessar recursos externos ao SPE e facilitar a comunicação inter-elementos já seria motivo suficiente para a implementação de uma versão, mesmo que simplificada, do EPOS.

2.3.2 Gerenciamento das tarefas dos SPEs

O gerenciamento de tarefas em um SPE pode ser feito em dois níveis. Num deles, o próprio SPE gerencia as suas diversas tarefas, usando a memória local para a troca de contexto. No outro o SPE tem uma tarefa e a PPE é responsável pelo chavemento do contexto, usando a memória principal.

Os SPE possuem 128 registradores de propósito geral de 128 bits, mais uma memória local de 256 Kilobytes e dois conjuntos de registradores de controle.

Em ambos a memória local torna uma política de *time-sharing* pouco aceitável. No primeiro por comprometer uma parte da restrita quantidade de memória, no mínimo 1KB dos registradores de propósito geral, e no segundo o tempo para executar a transferência do contexto , que em grande parte é constituído pelos 256KB da memória local.

2.4 Implementação

O objetivo do trabalho era ter um porte funcional do EPOS para a arquitetura POWER64, com algum suporte primitivo aos SPEs. Isso não foi alcançado.

Os problemas enfrentados foram muito mais causadores de atrasos do que fatores impossibilitadores. O atraso foi suficiente para que o tempo destinado ao término deste trabalho se expirasse e o objetivo inicial não fosse alcançado.

Contudo, alguma coisa foi implementada. Diversas abstrações foram construídas e testadas com o auxílio de um ambiente simplificado, de fácil manutenção, só que sem recursos de mais alto nível (como *threads*).

Os problemas que bloquearam o desenvolvimento estão listados à seguir, seguido de uma descrição do que foi feito no ambiente simplicado e então alguns comentários sobre as tentativas antes de se optar por usar o *hardware* do Playstation 3.

2.4.1 Problemas

Nessa sessão iremos listar alguns dos problemas enfrentados no porte. Não são problemas que impossibilitariam a implementação da versão, no entanto, durante o tempo de desenvolvimento desse trabalho, não foi possivel achar uma solução ideal para ser incorporado definitivamente ao EPOS.

Como vai ser possível notar, os problemas não são impossibilitadores e sim causadores de atraso. Como existe uma data final para o trabalho, esses atrasos impossibilitaram o término da implementação para o EPOS.

Hypervisor

O Hypervisor é um componente importante para o porte, pois todo o hardware é disponibilizado através dele, seja por chamada de sistema, seja através das partes virtualizadas transparentemente. O problema aqui foram as chamadas de sistema.

Não existe um documento central, público detalhando as chamadas do Hypervisor. Isso tornou o desenvolvimento consideravelmente lento, visto que para descobrir o funcionamento de qualquer chamada, uma busca pelo kernel do Linux e pelos diversos fóruns de discussão deveria ser feita.

As chamadas de sistema são apenas um dos problemas. O mecanismo de UART virtual

responsável por algumas das configurações de hardware e comandos para manejo de sistema (desligamento, por exemplo) funciona através de comandos empacotados e serializados. O formato desses pacotes de comando está disponível somente nos fontes do Linux.

MMU

A arquitetura da MMU do Cell possui algumas complexidades no seu manejo. Além disso ela é dependente de algumas chamadas do Hypervisor, que, como visto, tem documentação escassa.

Ao ocorrer um conflito entre dados e estruturas esperados e os existentes na implementação abstrata de MMU do EPOS, a tarefa de implementação foi colocada para depois. No entanto não foi possível completar o trabalho a tempo da confecção deste documento.

Uma versão da MMU em software poderia ter resolvido o problema, mas essa decisão não foi tomada à tempo.

2.4.2 Sandbox

O sistema de construção do EPOS é complexo. Antes de ter um sistema capaz de inicializar e demonstrar de alguma forma que este sistema está funcionando um caminho considerável deve ser transposto.

Como forma de amenizar dificuldades de um início lento e tentar focalizar o esforço na modelagem, implementação e testes dos componentes que mais tarde pudessem ser transportados ao EPOS real, uma árvore simplificada e sem a maior parte do sistema operacional foi construída.

O que sobrou, foram somente as abstrações que tem algum interesse para o porte, normalmente as abstrações mais próximas do hardware, e a única semelhança com o EPOS é que compartilha a mesma estrutura de diretórios e algumas *macros* e, portanto, nem poderia ser considerada uma versão do EPOS.

No entanto isso torna possível o teste individual das funcionalidades das abstrações sem ter a preocupação do ajuste de todo o sistema de construção nem com a solução para o *bootstrap*.

Esse ambiente simplificado foi chamado de *sandbox*[3]. Efetivamente, ele é o resultado desse trabalho.

[3] "caixa de areia" em inglês.

As seguintes abstrações foram implementadas e testadas com o uso deste *sandbox*:

- TSC
- RTC
- Timer
- vUART
- PS3AV
- PS3_Display
- FlashROM

Além dessas, o ambiente possui uma versão simplificada da CPU, para conter as dependências que qualquer uma das outras abstrações exigissem, uma versão improvisada da MMU para permitir alguns testes e estudos (como inicializar, como mapear a tabela de páginas, como alocar as entradas de páginas, etc), e o sistema de *wrapper* para o *Hypervisor* extraído do kernel do Linux e devidamente adaptada. O recurso de saída de texto (abstração *OStream*) do EPOS também foi utilizado.

A implementação do *Sandbox* foi baseada em um programa de demonstração (McLoad) adquirido em um fórum de discussão. Da versão original ainda restam o bootstrap e a MMU, o restante ou foi descartado ou foi substituído pelas abstrações apresentadas aqui.

TSC e RTC

O Hypervisor fornece uma chamada que retorna tanto o timestamp quanto a contagem de segundos desde a *Época*[4]. Como no EPOS existem abstrações independentes para tratar cada um deles a chamada é feita em ambas, retornando apenas o dado pertinente.

Na figura 2.4, pode-se se ver a modelagem para o EPOS. Na versão para o *Sandbox*, não existe derivação de uma classe mais abstrata.

[4] do inglês “Epoch”, no caso é primeiro de janeiro de 2000, segundo comentários no kernel do Linux

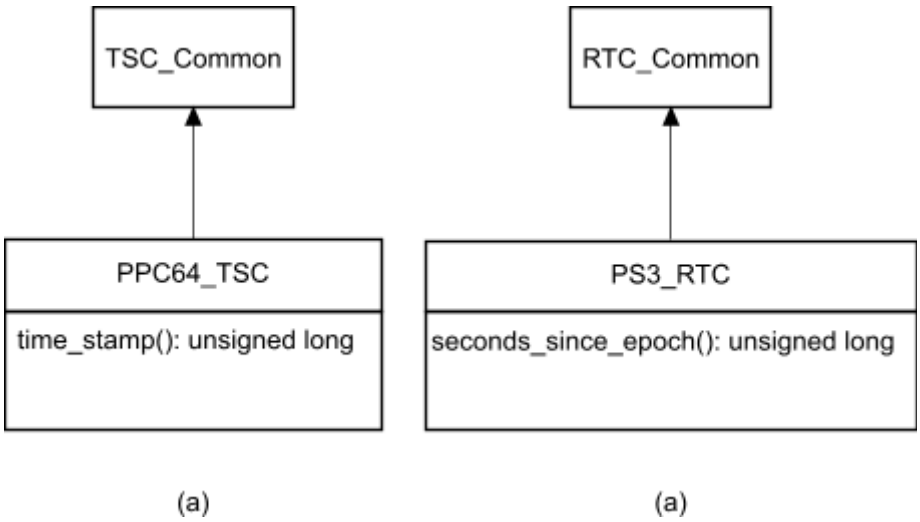

Figura 2.4: (a) TSC e (b) RTC

Timer

O mecanismo fornecido para contagem de tempo com disparo de interrupção é um registrador de contagem (um por thread de hardware). Trata-se de um registrador de 32 bits, incrementado automaticamente e em intervalos constantes, quando o bit mais significativo torna-se "1", ocorre uma interrupção. Esse registrador é chamado de *decrementer*.

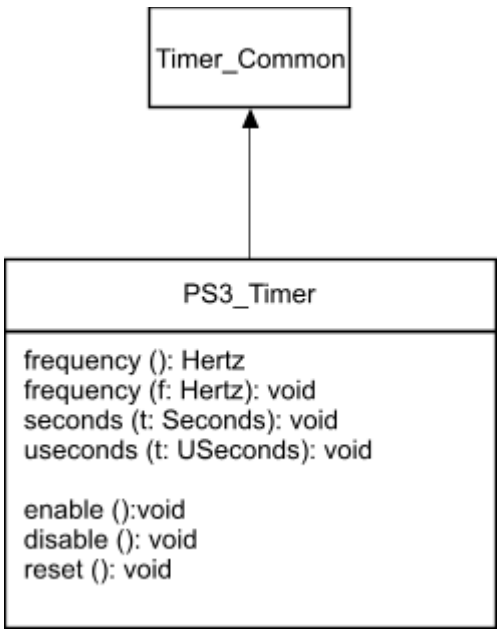

Figura 2.5: Timer

Segundo o manual(IBM SYSTEMS AND TECHNOLOGY GROUP, 2007b), para determinar o período da interrupção, a seguinte fórmula pode ser empregada:

$$dec = \frac{tempo \times TB}{1000000}$$

Onde “tempo” é passado como parâmetro (em microsegundos), a frequência de atualização do registrador (“TB”) pode ser conseguida tanto por chamada ao hypervisor quanto por leitura de um registrador de propósito especial. O valor resultante “dec” é atribuido ao *decrementer*. Para “tempo” em segundos, é só remover a divisão.

Contudo, pela modelagem exigida pelo EPOS (fig. 2.5), uma frequência e não o tempo é especificado para a abstração do Timer. Portanto, aplicando as devidas transformações, a seguinte fórmula é usada dentro do método “Timer::frequency()”:

$$dec = \frac{TB}{frequencia}$$

A “frequencia” é passada como parâmetro em *Hertz*.

A freqüência de atualização pode não ser constante, ela é alterada de acordo com o modo de operação, normalmente associado com a gerência de energia.

O Timer tem uma função de certa importância, visto que o sistema de *threads* do EPOS o utiliza para marcar o intervalo que é disponibilizado para cada *thread*.

vUART, PS3AV e PS3_Display

Estas três abstrações são utilizadas para fornecer ao OStream um meio de saída (no caso o monitor ou televisor acoplado na saída de vídeo).

A vUART é utilizada pelo PS3AV para inicializar o sistema de áudio e vídeo. Ela também é utilizada para fazer o reinicio do equipamento por software. Sua operação consistem em escrever pacotes de dados constituídos de comandos codificados em um fluxo de bytes. Os pacotes necessários foram montados usando as estruturas correspondentes encontradas nos códigos fonte do Linux (podem ser vistos nos anexos E e F).

Como já descrito antes, o PS3AV usa a vUART e configura a saída de vídeo, no momento apenas configurações pré-estabelecidas estão disponíveis e nenhum meio para alterá-las está implementado.

O PS3_Display inicializa a GPU, cria um *framebuffer* na memória dedicada ao vídeo e disponibiliza rotinas para escrever na tela. Essas rotinas seguem a interface estabelecida pelo EPOS e são utilizadas pelo OStream.

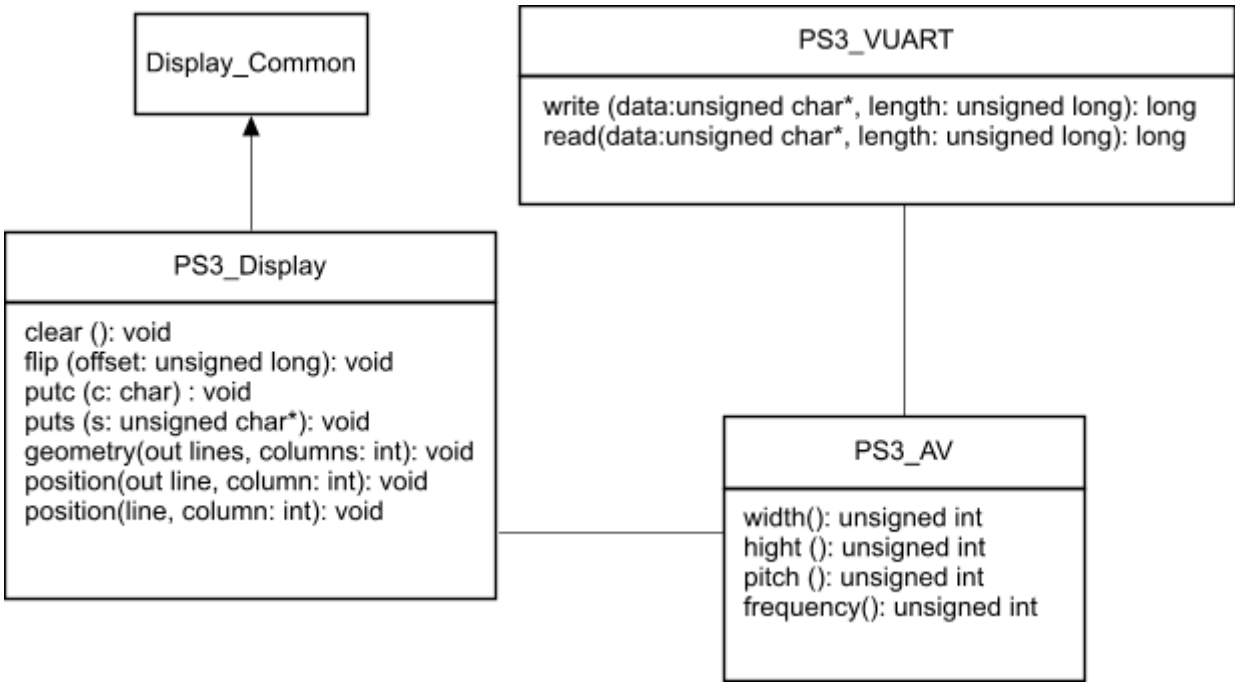

Figura 2.6: vUART, PS3AV e PS3_Display

Resultados

Em resumo, o sistema montado inicia, escreve algumas informações na tela (usando o OStream), pára por alguns instantes (usando o RTC para determinar o tempo), muda o parâmetro de configuração de sistema operacional, desliga as abstrações e reinicia o sistema. Uma variação do programa permitiu verificar se a interrupção do *decrementer* estava configurada corretamente.

Desta forma foi verificado que as abstrações de TSC, RTC e Timer estão funcionando. A FlashROM, que permite definir para o equipamento qual sistema operacional irá executar, também pareceu operar dentro do esperado.

Com esse pequeno protótipo foi possível validar as partes isoladas que em algum momento, com as devidas adaptações, virão a se tornar os mediadores de *hardware* do EPOS.

2.4.3 Tentativas Abandonadas

Antes desta implementação para o Playstation 3 houveram duas outras tentativas. A primeira usando apenas a versão para POWER32 teve um certo sucesso, mas algumas funcionalidades criticas não estavam de acordo e havia a possibilidade dos SPEs estarem além da capacidade de endereçamento.

A segunda foi iniciada por causa da questão do endereçamento. No entanto, estava na iminência da aquisição do hardware e então foi abandonada logo que este chegou.

As duas versões utilizavam o recurso do OpenFirmware para algumas das funcionalidades e uma abstração para acessar essas funcionalidades foi implementado. Saída e entrada de texto e informações interessantes sobre o *hardware* da máquina eram conseguidas através desse sistema.

A versão de 64-bits teria de entrar em modo 32-bits para executar as funções do OpenFirmware (OpenFirmware é um processo 32-bit), mas isso não chegou a ser feito.

Essas duas versões serviram como uma plataforma de estudos para a arquitetura POWER e uma parte do que foi feito nelas pode ser reaproveitado no esforço da versão para o Playstation 3.

3 *Particionamento de código*

3.1 O tradicional: linguagens imperativas

O *Cell* é heterogêneo. O PPE tem arquitetura diferente dos SPEs, portanto o conjunto de instruções que executa no PPE é diferente do que executa nos SPEs. Além disso os SPEs são praticamente máquinas independentes com memória própria e limitada a 256KB. Operações *DMA* devem ser executadas para carregar programa e dados e um agendamento adequado dessas operações precisa ser feita para manter um througput alto.

Utilizando C/C++ com as ferramentas que estão disponíveis publicamente hoje, significa particionar o problema e balancear a comunicação manualmente além de usar compiladores distintos para PPE e SPE.

Existe um esforço da IBM para se produzir uma solução única que gere a partir de um código fonte único e seqüencial, um programa vetorizado (para aproveitar as capacidades SIMD presentes) e distribuído (EICHENBERGER et al., 2006). Essa solução baseia-se no uso das especificações *OpenMP*.

Outras soluções unificadas podem ser encontradas no Sieve C/C++, e talvez no futuro com os compiladores da GNU suportando OpenMP. O Sieve C/C++ e o OpenMP oferecem um tratamento semelhante ao código. Através de anotações (palavras reservadas chamadas “sieves” no Sieve C/C++ e diretrizes de precompilador e macros no OpenMP) o programador isola e define áreas em que o compilador deve tentar paralelizar.

Além desses temos a *Cilk*, que tenta extender a linguagem C para adicionar características de programação paralela. Nem o Cilk nem o OpenMP (do compilador da GNU) tem suporte aos SPEs do Cell.

3.2 As alternativas

O OpenMP (ou variante) não oferece uma melhoria na exposição do paralelismo. Seu papel é permitir que o programador que já saiba onde explorar paralelismo sugira ao compilador que ele tente distribuir o algoritmo. Embora diversas linguagens OO tenham suporte a multi-threading, o paralelismo não emerge naturalmente da implementação.

Em parte, essa dificuldade vem do fato das linguagens convencionais (C/C++ e mesmo Java) serem projetadas para arquiteturas de memória centralizada o que torna difícil a utilização das vantagens de arquiteturas com memória distribuida.

A proposta de alternativas para isso tem surgido, além dos paradigmas mais tradicionais, como abordagens funcionais e máquinas virtuais.

3.2.1 Stream Programming

Deixando as aplicações de propósito geral de lado e reduzindo o dominio para o qual a linguagem pode ser usada, o *Stream Programming* (ou *Stream Processing*) tem se revelado uma alternativa muito promissora.

Dado um conjunto de dados de entrada e saída (*streams*), o paradigma é essencialmente baseado em definir uma série de operações (*kernels*) que devem ser aplicadas para cada elemento do *stream*. Os *kernels* devem operar usando apenas dados locais, isso faz com que cada *kernel* não tenha dependências que comprometeriam o paralelismo de cada um deles.

Quase todas as aplicações de mídia podem ser modeladas como um fluxo de dados que deve ser tratado. É o caso de codificação e decodificação de áudio e vídeo, processamento de imagens e geradores de gráficos tridimensionais.

A seguir uma breve (muito breve) descrição de algumas linguagens de onde foram retiradas as idéias deste trabalho e uma nota sobre GPUs.

Brook C

Brook é um projeto da Universidade de Stanford que propõe uma extensão ao padrão ANSI C e é projetada para incorporar idéias de computação paralela em uma linguagem familiar e eficiente.

Um programa Brook consiste de código C válido somado a algumas extensões sintáticas para a declaração de streams.

StreamIt

StreamIt é uma linguagem de programação e uma infraestrutura de compilação, criada especificamente para sistemas streaming modernos. Foi projetada para facilitar a programação de grandes aplicações streaming, assim como seu mapeamento eficiente e efetivo para uma grande variedade de arquiteturas.

Entre outras coisas que a linguagem propõe está a remoção das responsabilidades de definir granularidade, balanço de carga e decisões de localidade e sincronização do usuário e passá-las ao compilador.

GPUs e Shader Programs

As GPUs sempre implementaram alguma forma de *Stream Processing*. Mesmo quando não eram programáveis, a natureza da aplicação favorecia o uso do paradigma.

Atualmente as GPUs são programáveis e as linguagens para elas são *Turing-Completas*, permitindo que seja possível o uso delas como aceleradores de processamento genéricas.

Cada *shader program* pode ser visto como um *kernel* do *BrookC* ou como um filter do *StreamIt*. Eles são executados repetidas vezes sobre cada um dos *pixels* do *framebuffer* e dessa forma geram imagens.

4 *Framework C++ para Stream Processing*

4.1 Componentes

Os componentes descritos aqui são muito semelhantes ao que lhes é atribuido no *StreamIt*. Diversas outras linguagens/extensões seguem uma modelagem parecida, com alterações pertinente ao domínio de aplicação para o qual ela é projetada.

Quase todos os componentes operarão usando um padrão *proxy*. O objetivo é permitir que um escalonador instalado num PPE ou SPE possa distribuir, transferir ou parar as tarefas (*kernels*) nos SPEs. Dessa forma os *Kernels* já compilados (binários para SPE) são embutidos em suas versões proxy (instalada em um SPE ou PPE) que cuidam do seu carregamento e descarregamento.

Desta forma, as seguintes abstrações foram estabelidas:

- Stream
- Kernel
- Pipeline
- Splitjoin
- Feedback

Streams cuidam do transporte de dados, kernels fazem o trabalho pesado, enquanto Pipelines e Splitjoins se encarregam de descrever as áreas e o tipos de paralelismo possíveis naquele ponto, o Feedback permite um ciclo de realimentação do sistema.

Na figura 4.1 um esboço do relacionamento entre estes componentes.

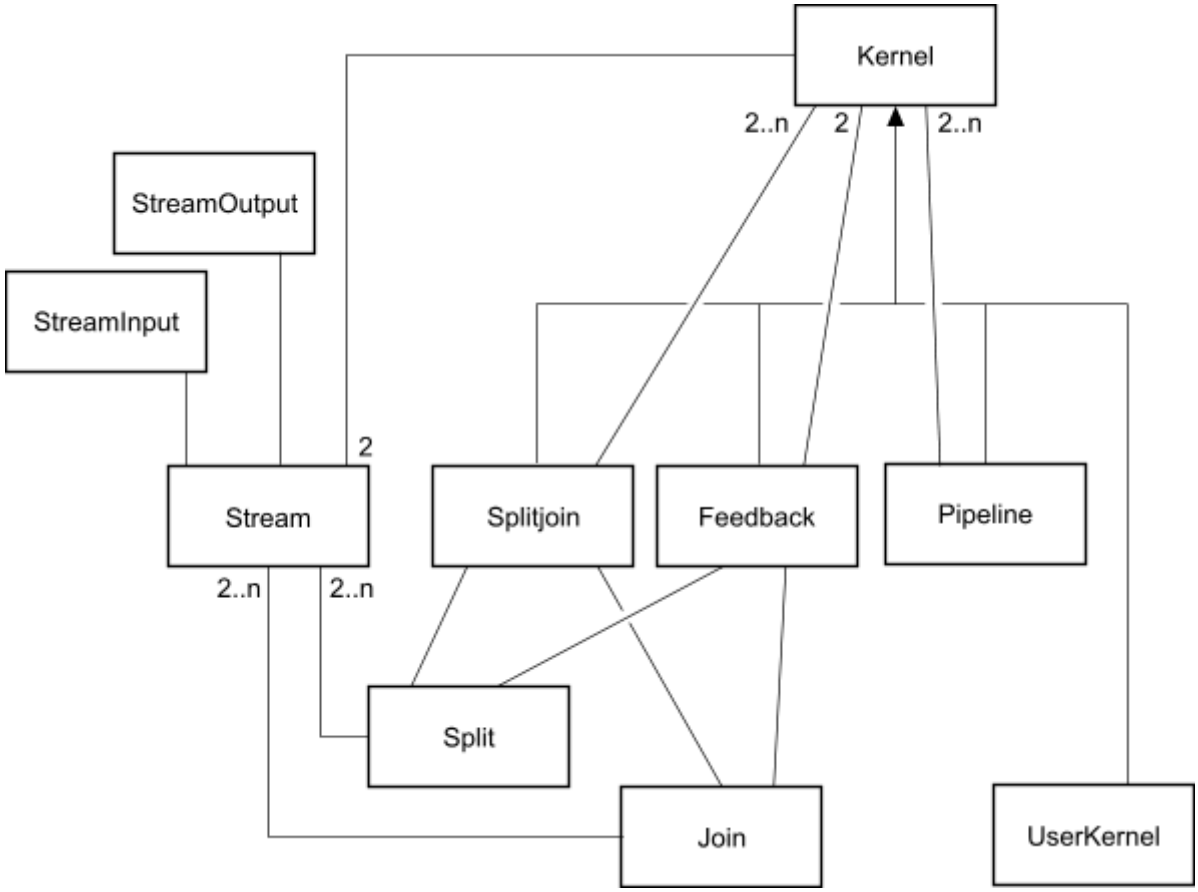

Figura 4.1: Esboço do relacionamento entre os componentes

4.1.1 Stream

Toda a alimentação de dados do sistema vem dos Streams. Embora seja possível a alimentação com dados escalares, isso não é recomendável. A idéia por trás do processamento de Stream é de que o sistema seja alimentado com muitos dados em seqüência que passam por um conjunto de operações e produza na saída um novo fluxo de dados.

Figura 4.2: Stream

Todos os componentes do framework se comunicam através de streams. Desta forma os componentes estabelecem uma relação produtor/consumidor entre si.

Abstratamente o Stream é uma fila FIFO. Para aproveitar melhor os recursos do Cell, o Stream deve ser distribuído de forma a aproveitar as características de memória distribuída e comunicação que a arquitetura oferece.

Operações

Todo o acesso aos Streams (e, portanto aos dados) são feitos usando três operações:

- push: insere dados no final da fila.
- pop: remove dados do inicio da fila
- peek: inspeciona os dados do inicio da fila

Como as operações sobre a fila são executadas através dessas abstrações podemos inserir sistemas de controle e sincronia nelas ao invés de espalhar através das outras estruturas do sistema. Isso torna o uso mais transparente para o usuário.

Stream Distribuído

Como dito na seção 4.1.1, os Streams devem estar distribuídos, para o sistema aproveitar o máximo dos recursos de paralelismo da arquitetura.

Desta forma o Stream pode estar em arranjos (figura 4.3, página 40) em que ele está presente em um SPE (na Memória Local) como o stream de saída e em outro SPE como um stream de entrada ou que ele está com uma parte presente na RAM, utilizando os Local Storage dos SPE e a RAM do sistema, todo no mesmo SPE ou ainda num SPE e na RAM (nesse caso provavelmente uma das extremidades do stream estaria no PPE).

Idealmente a comunicação direta entre SPEs deve ser utilizada, aliviando quase toda a tarefa de gerência do PPE.

Na implementação, as operações descritas na seção anterior se encarregam do agendamento das transações DMA entre os produtores/consumidores.

4.1.2 Kernels

Os Kernels são as unidades mais básicas de processamento. Normalmente um usuário do framework terá que escrever um conjunto de Kernels e/ou usar o conjunto de uma biblioteca.

Como unidades básicas eles idealmente representam operações que podem ser executadas paralelamente sem concorrência, exceto pela relação produtor/consumidor de dados que é estabelecida na criação do grafo do problema.

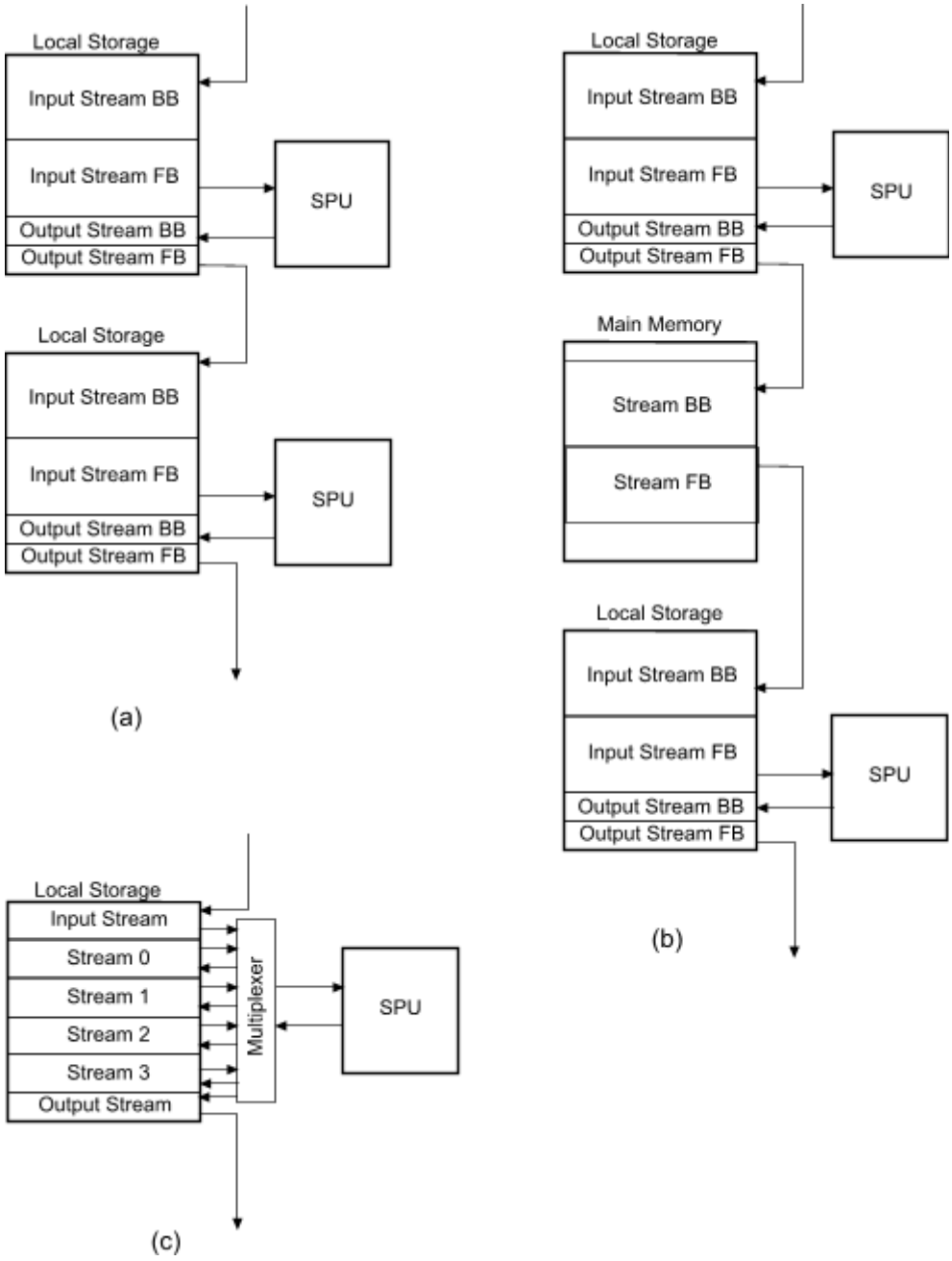

Figura 4.3: Possíveis arranjos de Stream

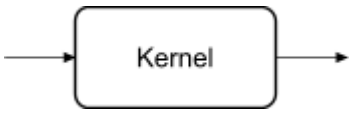

Figura 4.4: Kernel

Cada Kernel tem apenas um Stream de entrada e um Stream de saída, o Stream de entrada deveria fornecer dados apenas-leitura enquanto o de saída deveria fornece-los somente-escrita. Cada umas das operações de *push*, *pop* e *peek* aplicada ao Stream sempre altera a fila de maneira constante.

Do ponto de vista de implementação, existe duas coisas para o usuário precisa se preocupar:

1. o construtor e;
2. void Kernel::work(void).

O *construtor* serve para inicializar o Kernel[1], ele será chamado uma vez durante o ciclo de vida do programa. As operações sobre Stream não estão disponíveis ainda[2]. Também são definidos os valores para push, pop e peek, usando o construtor da classe kernel.

Já no método "`void Kernel::work (void)`" é onde se define a função que o Kernel irá executar. Ele é chamado ciclicamente, sempre que os dados vindos do Stream de entrada estiverem disponíveis.

Como C++ está sendo utilizado, é possível definir mais métodos membros auxiliares como convier ao usuário.

4.1.3 Pipeline

Um Pipeline é um conjunto de Kernels enfileirados e operando em série. A relação produtor/consumidor nesse caso é um-para-um. Portanto o escalonamento do Pipeline deve ser resolvido eficientemente, de forma que não aconteça *data-starvation* no próximo estágio.

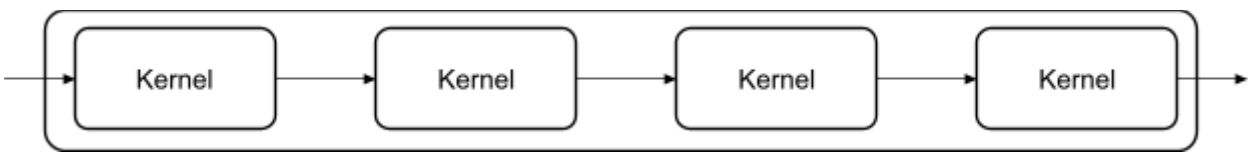

Figura 4.5: Pipeline

[1] Informar sobre algumas variáveis dinâmicas, por exemplo
[2] Porque o programa ainda não foi iniciado, evidentemente

Dentro do Pipeline a coleta e remoção de dados é bem definida e constante para cada Kernel e, portanto, o escalonamento pode ser determinado estaticamente. Este escalonamento estático permitiria a execução de diversos Kernels de um mesmo Pipeline (fusão de kernels) num mesmo processador.

A fusão de kernels deve levar em conta a memória disponível no processador alvo. No caso do Cell, cada SPE tem uma quantidade exclusiva, mas restrita, de memória que é destinada tanto a instruções quanto a dados. Se muitos kernels forem fundidos a ponto da quantidade de memória livre restante para dados não for suficiente para manter um fluxo de execução ótimo, essa fusão não será vantajosa e não deve ser feita.

No StreamIt esse problema é resolvido pelo compilador, que conhece a arquitetura alvo e faz uma análise sobre o grafo do problema, levando em conta o tamanho de cada kernel e a memória disponível para os processadores.

4.1.4 Splitjoin

Essa estrutura indica paralelismo em nível de tarefa. O splitjoin serve pra dividir uma linha de processamento em várias, normalmente instanciando em diversos processadores.

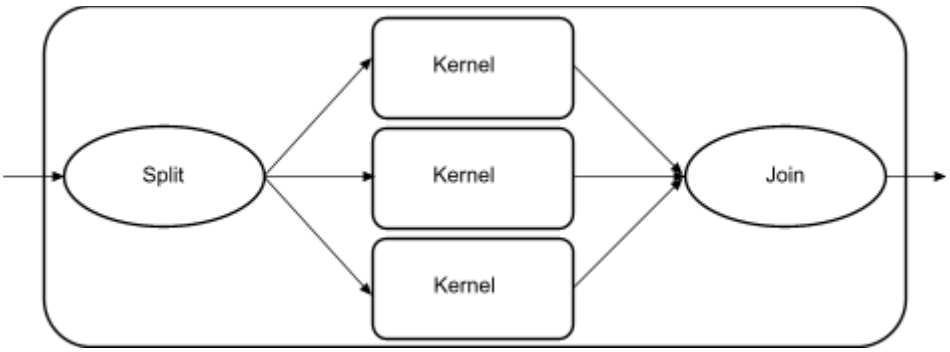

Figura 4.6: SplitJoin

O nome splitjoin vem das estruturas responsáveis pela divisão e reunião dos dados.

Foram mapeados dois tipo de splits e um tipo de join:

- split round-robin
- split duplicate
- join round-robin

Tanto o split quanto o join round-robin distribuem de maneira seqüencial os dados do stream. Já o split duplicate replica os dados para todos os kernels.

4.1.5 Realimentação (Feedback)

Simplificadamente um sistema realimentado pode ser representado da seguinte como na figura 4.7. No caso “f” representa a função, e “s” o fator de escala, que é aplicado sobre a saída de “f”. O resultado de “s” é então usado como uma entrada para “f”.

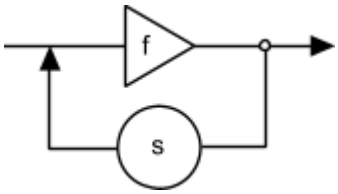

Figura 4.7: Um sistema realimentado simples

O Feedback é modelado (figura 4.8) como um Splitjoin invertido, isto é, com um join na entrada e um split na saída. Diferente do Splitjoin, os splits e joins do Feedback não possuem configuração da quantidade de streams que eles distribuem que é constante em dois.

Apesar de parecer uma excessão ao número de streams que alimentam um kernel, externamente apenas um stream de entrada e outro de saída são disponíveis. O outro conjunto fica para o segundo kernel que representa uma função de escalonamento para a realimentação.

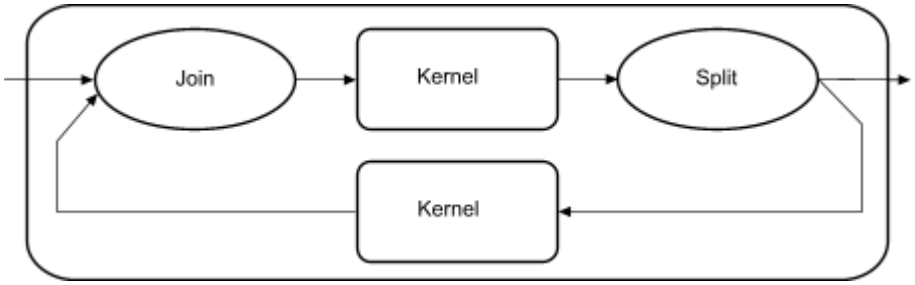

Figura 4.8: Abstração de Feedback

Muitos sistemas apresentam a necessidade de realimentação (sistema de criptografia são bons exemplos), essa necessidade justifica essa pequena desordem no grafo do problema.

4.1.6 Alguns exemplos

Nos anexos A, B e C, existem exemplos do que se espera ser o uso do *framework* a ser implementado. Especificamente:

- anexo A: um filtro passa-baixo;
- anexo B: uma função identidade, que passa os dados sem tratamento e seu uso em um algoritmo de transposição de matriz;

- anexo C: mostra um ordenador que usa diversas estruturas. É como se espera ser um programa feito com o *framework*.

Conclusões

Devido aos próprios limites físicos, que incluem velocidade da luz e o fato de estarmos fabricando trilhas de circuito do tamanho de alguns poucos átomos, chegamos em um ponto em que aumentar a frequência e reduzir os componentes simplesmente não representam ganho de velocidade ou não são mais possíveis. O aquecimento também tornou-se um problema, já que a densidade de calor nos processadores chega a equiparar-se a densidade de calor em um reator nuclear.

A solução encontrada foi reduzir a freqüência de operação e multiplicar o número de núcleos de processamento em um processador, criando assim SMPs em um único *chip*. No caso do Cell, cada processador tem sua própria memória dedicada e a comunicação com a memória principal é feita por um dispositivo de interconexão de alto-desempenho, tornando ele algo como um *grid* em um *chip*.

As linguagens de programação, no entanto, normalmente trabalham com o paradigma de memória centralizada, normalmente com um único processador disponível, o que as torna pouco propícias para gerar programas para essas novas arquiteturas (com multiplos processadores e memória distribuída). Existem algumas propostas de linguagens que são capazes de aproveitar essas qualidades e algumas propostas de extensão de linguagens tipicamente não-paralelas.

Neste trabalho foi feita uma tentativa para o porte do sistema operacional EPOS, infelizmente ela não foi sucedida. Os problemas enfrentados acabaram por atrasar o projeto o suficiente para não ser possível terminar o desenvolvimento a tempo para o fechamento deste documento.

Mesmo assim, um ambiente simplificado foi construído para testar algumas das funcionalidades que estão prontas. Desta forma diversos componentes do sistema estão prontos, com a integração com o EPOS por ser testada, assim que possível.

Além disso, levando em conta a complexidade de desenvolvimento para o tipo de arquitetura escolhida, um esboço para um *framework* baseado na linguagem StreamIt foi feito.

Como trabalhos futuros podemos considerar o término do porte, suporte a operações vetoriais e em ponto-flutuante no contexto da abstração da CPU, *multi-threading* em hardware,

uma versão do EPOS para os SPEs e a melhoria do projeto e implementação do framework para *Stream Processing*.

Referências Bibliográficas

BARROSO, L. A. et al. Piranha: a scalable architecture based on single-chip multiprocessing. In: *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 2000. p. 282–293. ISBN 1-58113-232-8.

CONSTANTINOU, T. et al. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Archit. News*, ACM Press, New York, NY, USA, v. 33, n. 4, p. 80–91, 2005. ISSN 0163-5964.

EICHENBERGER, A. E. et al. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. *IBM Systems Journal*, v. 45, n. 1, p. 59–84, 2006.

FRöHLICH, A. A. M. *Application-Oriented Operating Systems*. Tese (Doutorado) — GMD – Forschungszentrum Informationstechnik GmbH, 2001.

GONZALEZ, J. M. J. Deterministic processor scheduling. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 9, n. 3, p. 173–204, 1977. ISSN 0360-0300.

GORDON, M. I. et al. A stream compiler for communication-exposed architectures. In: *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM Press, 2002. p. 291–303. ISBN 1-58113-574-2.

HA, S.; LEE, E. A. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 40, n. 11, p. 1225–1238, 1991. ISSN 0018-9340.

HA, S.; LEE, E. A. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 46, n. 7, p. 768–778, 1997. ISSN 0018-9340.

IBM. *PowerPC User Instruction Set Architecture: Book 2*. 2.02. ed. USA, January 2005.

IBM. *PowerPC User Instruction Set Architecture: Book 3*. 2.02. ed. USA, January 2005.

IBM. *PowerPC User Instruction Set Architecture: Book I*. 2.02. ed. USA, January 2005.

IBM SYSTEMS AND TECHNOLOGY GROUP. *SPU Application Binary Interface Specification*. 1.4. ed. Hopewell Junction, NY, USA, October 2005.

IBM SYSTEMS AND TECHNOLOGY GROUP. *Cell Broadband Engine: Hardware Initialization Guide*. 1.3. ed. Hopewell Junction, NY, USA, March 2006.

IBM SYSTEMS AND TECHNOLOGY GROUP. *C/C++ Language Extensions for Cell Broadband Engine Architecture*. 2.4. ed. Hopewell Junction, NY, USA, mach 2007.

IBM SYSTEMS AND TECHNOLOGY GROUP. *Cell Broadband Engine: Programming Handbook*. 1.1. ed. Hopewell Junction, NY, USA, April 2007.

IBM SYSTEMS AND TECHNOLOGY GROUP. *Cell Broadband Engine: Registers*. 1.5. ed. Hopewell Junction, NY, USA, April 2007.

KAPASI, U. J. et al. Programmable stream processors. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 36, n. 8, p. 54–62, 2003. ISSN 0018-9162.

KHAILANY, B. et al. Imagine: Media processing with streams. *IEEE Micro*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 21, n. 2, p. 35–46, 2001. ISSN 0272-1732.

KONGETIRA, P.; AINGARAN, K.; OLUKOTUN, K. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 25, n. 2, p. 21–29, 2005. ISSN 0272-1732.

OLUKOTUN, K.; HAMMOND, L. The future of microprocessors. *Queue*, ACM Press, New York, NY, USA, v. 3, n. 7, p. 26–29, 2005. ISSN 1542-7730.

OLUKOTUN, K. et al. The case for a single-chip multiprocessor. In: *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM Press, 1996. p. 2–11. ISBN 0-89791-767-7.

OWENS, J. D. et al. Polygon rendering on a stream architecture. In: *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. New York, NY, USA: ACM Press, 2000. p. 23–32. ISBN 1-58113-257-3.

PHAM, D. et al. The design and implementation of a first-generation cell processor. In: *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*. Washington, DC, USA: IEEE Computer Society, 2005. v. 1, p. 184–185–592. On-line: http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/7FB9EC5D5BBF51ED87256FC000742186.

POLLACK, F. J. New microarchitecture challenges in the coming generations of cmos process technologies (keynote address)(abstract only). In: *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1999. p. 2. ISBN 0-7695-0437-X.

PRASANNA, G. N. S.; AGRAWAL, A.; MUSICUS, B. R. Hierarchical compilation of macro dataflow graphs for multiprocessors with local memory. *IEEE Trans. Parallel Distrib. Syst.*, IEEE Press, Piscataway, NJ, USA, v. 5, n. 7, p. 720–736, 1994. ISSN 1045-9219.

RIXNER, S. et al. A bandwidth-efficient architecture for media processing. In: *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998. p. 3–13. ISBN 1-58113-016-3.

SAKAI, S. Cmp on soc: architect's view. In: *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*. New York, NY, USA: ACM Press, 2002. p. 101–102. ISBN 1-58113-576-9.

SONY COMPUTER ENTERTAINMENT INC. *Cell Broadband Engine Architecture*. 1.0. ed. Tokyo, Japan, August 2005.

STANKOVIC, J. A. Strategic directions in real-time and embedded systems. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 28, n. 4, p. 751–763, 1996. ISSN 0360-0300.

STROUSTRUP, B. *C++ A Linguagem de Programação*. 3. ed. Porto Alegre: Bookman, 2000. Trad. Maria Lúcia Blank Lisbôa e Carlos Artur Lang Lisbôa.

THIES, W. et al. Teleport messaging for distributed stream programs. In: *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM Press, 2005. p. 224–235. ISBN 1-59593-080-9.

WILLIAMS, S. et al. The potential of the cell processor for scientific computing. In: *CF '06: Proceedings of the 3rd conference on Computing frontiers*. New York, NY, USA: ACM Press, 2006. p. 9–20. ISBN 1-59593-302-6.

ANEXO A – Filtro passa baixo

```
class LowPass: virtual public kernel<float, float>
{
private:
  float* weights;
  unsigned int N;
  typedef kernel<float, float> kernel_t;
public:
  LowPass (unsigned int N, float freq)
  : kernel_t (1, 1, N)
  {
    this->N = N;
    weights = calcWeights(freq);
  }

  ~LowPass ()
  {
    if (weights)
          delete weights;
  }

  virtual void work()
  {
    float result = 0;
    for (int i = 0; i < N; ++i)
    {
      result += weights[i] * peek(i);
    }
    push(result);
    pop();
  }
}
```

ANEXO B – Transposição de matrizes

```
template <typename T>
class Identity
: virtual public kernel <T,T>
{
public:
  Identity (unsigned int M) : kernel <T,T> (M,1) {}
  virtual void work ()
  {
    push(pop());
  }
}
```

```
template <int M, int N>
kernel<float , float> transpose()
{
  return (transpose<M, N - 1> (), Identity(M));
}

template <int M, 0>
kernel<float , float> transpose()
{
  return Identity(M);
}
```

ANEXO C – Ordenador

```
template <int N>
kernel<int, int> mergesort()
{
  // operator "+" return pipeline, operator "," returns splitjoin
  return (mergesort(N/2), mergesort(N/2)) + Merge(N);
}

template <2>
kernel<int, int> mergesort()
{
  // operator "+" return pipeline
  return Sort(N) + Merge(N);
}


void main () {
  kernel<int, int> sort = mergesort<10>();
  stream <int> data, result;
  while (data >> sort)
    result << sort;

  return 0;
}
```

ANEXO D – Chamadas conhecidas do Hypervisor

Código	Nome da Função (em C/C++)	entrada	saída
0	lv1_allocate_memory	4	2
1	lv1_write_htab_entry	4	0
2	lv1_construct_virtual_address_space	3	2
3	lv1_invalidate_htab_entries	5	0
4	lv1_get_virtual_address_space_id_of_ppe	1	1
6	lv1_query_logical_partition_address_region_info	1	5
7	lv1_select_virtual_address_space	1	0
9	lv1_pause	1	0
10	lv1_destruct_virtual_address_space	1	0
11	lv1_configure_irq_state_bitmap	3	0
12	lv1_connect_irq_plug_ext	5	0
13	lv1_release_memory	1	0
15	lv1_put_iopte	5	0
17	lv1_disconnect_irq_plug_ext	3	0
18	lv1_construct_event_receive_port	0	1
19	lv1_destruct_event_receive_port	1	0
24	lv1_send_event_locally	1	0
26	lv1_detect_pending_interrupts	1	4
27	lv1_end_of_interrupt	1	0
28	lv1_connect_irq_plug	2	0
29	lv1_disconnect_irq_plug	1	0
30	lv1_end_of_interrupt_ext	3	0
31	lv1_did_update_interrupt_mask	2	0
44	lv1_shutdown_logical_partition	1	0
54	lv1_destruct_logical_spe	1	0
57	lv1_construct_logical_spe	7	6
61	lv1_set_spe_interrupt_mask	3	0
64	lv1_set_spe_transition_notifier	3	0
65	lv1_disable_logical_spe	2	0
66	lv1_clear_spe_interrupt_status	4	0
67	lv1_get_spe_interrupt_status	2	1
69	lv1_get_logical_ppe_id	0	1

Código	Nome da Função (em C/C++)	entrada	saída
73	lv1_set_interrupt_mask	5	0
74	lv1_get_logical_partition_id	0	1
77	lv1_configure_execution_time_variable	1	0
78	lv1_get_spe_irq_outlet	2	1
79	lv1_set_spe_privilege_state_area_1_register	3	0
90	lv1_create_repository_node	6	0
91	lv1_get_repository_node_value	5	2
92	lv1_modify_repository_node_value	6	0
93	lv1_remove_repository_node	4	0
95	lv1_read_htab_entries	2	5
96	lv1_set_dabr	2	0
97	lv1_set_vmx_graphics_mode	1	0
98	lv1_set_thread_switch_control_register	1	0
103	lv1_get_total_execution_time	2	1
116	lv1_allocate_io_segment	3	1
117	lv1_release_io_segment	2	0
118	lv1_allocate_ioid	1	1
119	lv1_release_ioid	2	0
120	lv1_construct_io_irq_outlet	1	1
121	lv1_destruct_io_irq_outlet	1	0
122	lv1_map_htab	1	1
123	lv1_unmap_htab	1	0
127	lv1_get_version_info	0	1
140	lv1_construct_lpm	6	3
141	lv1_destruct_lpm	1	0
142	lv1_start_lpm	1	0
143	lv1_stop_lpm	1	1
144	lv1_copy_lpm_trace_buffer	3	1
145	lv1_add_lpm_event_bookmark	5	0
146	lv1_delete_lpm_event_bookmark	3	0
147	lv1_set_lpm_interrupt_mask	3	1
148	lv1_get_lpm_interrupt_status	1	1
149	lv1_set_lpm_general_control	5	2
150	lv1_set_lpm_interval	3	1
151	lv1_set_lpm_trigger_control	3	1
152	lv1_set_lpm_counter_control	4	1
153	lv1_set_lpm_group_control	3	1
154	lv1_set_lpm_debug_bus_control	3	1
155	lv1_set_lpm_counter	5	2
156	lv1_set_lpm_signal	7	0
157	lv1_set_lpm_spr_trigger	2	0
158	lv1_insert_htab_entry	6	3
162	lv1_read_virtual_uart	3	1
163	lv1_write_virtual_uart	3	1
164	lv1_set_virtual_uart_param	3	0
165	lv1_get_virtual_uart_param	2	1
166	lv1_configure_virtual_uart_irq	1	1
170	lv1_open_device	3	0

Código	Nome da Função (em C/C++)	entrada	saída
171	lv1_close_device	2	0
172	lv1_map_device_mmio_region	5	1
173	lv1_unmap_device_mmio_region	3	0
174	lv1_allocate_device_dma_region	5	1
175	lv1_free_device_dma_region	3	0
176	lv1_map_device_dma_region	6	0
177	lv1_unmap_device_dma_region	4	0
178	lv1_read_pci_config	6	1
179	lv1_write_pci_config	7	0
180	lv1_read_pci_io	4	1
181	lv1_write_pci_io	5	0
185	lv1_net_add_multicast_address	4	0
186	lv1_net_remove_multicast_address	4	0
187	lv1_net_start_tx_dma	4	0
188	lv1_net_stop_tx_dma	3	0
189	lv1_net_start_rx_dma	4	0
190	lv1_net_stop_rx_dma	3	0
191	lv1_net_set_interrupt_status_indicator	4	0
193	lv1_net_set_interrupt_mask	4	0
194	lv1_net_control	6	2
197	lv1_connect_interrupt_event_receive_port	4	0
198	lv1_disconnect_interrupt_event_receive_port	4	0
199	lv1_get_spe_all_interrupt_statuses	1	1
202	lv1_deconfigure_virtual_uart_irq	0	0
207	lv1_enable_logical_spe	2	0
210	lv1_gpu_open	1	0
211	lv1_gpu_close	0	0
212	lv1_gpu_device_map	1	2
213	lv1_gpu_device_unmap	1	0
214	lv1_gpu_memory_allocate	5	2
216	lv1_gpu_memory_free	1	0
217	lv1_gpu_context_allocate	2	5
218	lv1_gpu_context_free	1	0
221	lv1_gpu_context_iomap	5	0
225	lv1_gpu_context_attribute	6	0
227	lv1_gpu_context_intr	1	1
228	lv1_gpu_attribute	5	0
232	lv1_get_rtc	0	2
240	lv1_set_ppe_periodic_tracer_frequency	1	0
241	lv1_start_ppe_periodic_tracer	5	0
242	lv1_stop_ppe_periodic_tracer	1	1
245	lv1_storage_read	6	1
246	lv1_storage_write	6	1
248	lv1_storage_send_device_command	6	1
249	lv1_storage_get_async_status	1	2
254	lv1_storage_check_async_status	2	1
255	lv1_panic	1	0

Tabela D.1: Funções disponibilizadas (conhecidas) pelo Hypervisor

ANEXO E – Pacotes de Inicialização do Sistema AV

ordem	tamanho (bytes)	valor (hex)	significado
1	2	0x0205	versão do pacote
2	2	0x0004	número de bytes que seguem
3	4	0x01000001	Comando: VIDEO_INIT

Tabela E.1: Comando para inicialização subsistema de vídeo

ordem	tamanho (bytes)	valor (hex)	significado
1	2	0x0205	versão do pacote
2	2	0x0004	número de bytes que seguem
3	4	0x02000001	Comando: AUDIO_INIT

Tabela E.2: Comando para inicialização subsistema de audio

ordem	tamanho (bytes)	valor (hex)	significado
1	2	0x0205	versão do pacote
2	2	0x0008	número de bytes que seguem
3	4	0x00000001	Comando: AV_INIT
4	4	0x00000000	Event bit

Tabela E.3: Comando para inicialização sistema de áudio/vídeo

ordem	tamanho (bytes)	valor (hex)	significado
1	2	0x0205	versão do pacote
2	2	0x0004	número de bytes que seguem
3	4	0x00000002	Comando: AV_FIN

Tabela E.4: Comando para desligamento sistema de áudio/vídeo

ANEXO F – Pacotes de Configuração do Sistema AV

ordem	tamanho (bytes)	valor (hex)	significado
1	2	0x0205	versão do pacote
2	2	0x002C	número de *bytes* que seguem
3	4	0x01000002	Comando: VIDEO_MODE
4	4	0x00000000	Parâmetro: tela (0x0=Head A; 0x1=Head B)
5	4	0x00000000	reservado
6	4	0x00000009	Parâmetro: resolução do video (1280x720 px)
7	2	0x0000	reservado
8	2	0x0500	Parâmetro: largura da imagem em *pixels* (1280 px)
9	2	0x0000	reservado
10	2	0x02D0	Parâmetro: altura da imagem em *pixels* (720 px)
11	2	0x0000	reservado
12	2	0x1400	Parâmetro: tamanho da linha em *bytes* (5120 B)
13	4	0x00000000	Parâmetro: Formato de saída
14	4	0x00000007	Parâmetro: Formato de entrada do *framebuffer* (XRGB)
15	1	0x00	reservado
16	1	0x00	reservado
17	2	0x0000	Parâmetro: Ordem de entrada do RGB
18	4	0x00000000	reservado

Tabela F.1: Comando para configuração do sub-sistema de vídeo

ordem	tamanho (bytes)	valor (hex)	significado
1	2	0x0205	versão do pacote
2	2	0x0010	número de *bytes* que seguem
3	4	0x01000002	Comando: AV_MODE
4	2	0x0000	Parâmetro: porta de saída (0x0=HDMI 0;0x1=AVMULTI)
5	2	0x0002	Parâmetro: resolução do video (1280x720 px)
6	2	0x0001	Parâmetro: Espaço de cores de saída (YUV444-8)
7	2	0x0000	Parâmetro: Espaço de cores de entrada (RGB-8)
8	1	0x00	Parâmetro: dithering (desligado)
9	1	0x00	Parâmetro: tamanho do bit (8)
10	1	0x00	Parâmetro: super white (desligado)
11	1	0x01	Parâmetro: aspecto da imagem (16:9)

Tabela F.2: Comando para configuração do sistema av (vídeo)

ordem	tamanho (bytes)	valor (hex)	significado
1	2	0x0205	versão do pacote
2	2	0x0094	número de *bytes* que seguem
3	4	0x04000001	Comando: AVB_PARAM
4	2	0x0002	Parâmetro: número de pacotes de vídeo (2)
5	2	0x0000	Parâmetro: número de pacotes de áudio (0)
6	2	0x0002	Parâmetro: número de pacotes AV de vídeo (2)
7	2	0x0000	Parâmetro: número de pacotes AV de áudio (0)
8	48	ver F.1	Pacote de configuração para a tela A
9	48	ver F.1	Pacote de configuração para a tela B
10	48	ver F.2	Pacote de configuração para a saída HDMI
11	48	ver F.2	Pacote de configuração para a saída AVMultiport

Tabela F.3: Comando para enviar um conjunto de configurações para o sistema AV