UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO
MESTRADO EM SISTEMAS E COMPUTAÇÃO

SISTEMA OPERACIONAL E BIBLIOTECA DE FUNÇÕES PARA PLATAFORMAS MPSOC: UM ESTUDO DE CASO PARA SIMULADORES DE RESERVATÓRIOS

TADEU FERREIRA OLIVEIRA

Agosto de 2010
NATAL/RN

TADEU FERREIRA OLIVEIRA

SISTEMA OPERACIONAL E BIBLIOTECA DE FUNÇÕES PARA PLATAFORMAS MPSOC: UM ESTUDO DE CASO PARA SIMULADORES DE RESERVATÓRIOS

Dissertação submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como parte dos requisitos para obtenção do título de Mestre em Sistemas e Computação (MSc.).

Orientador: Prof. Dr. Ivan Saraiva Silva

Agosto, 2010
Natal/RN

RESUMO

O aumento da demanda por poder de processamento nos últimos anos forçou a indústria de circuitos integrados a buscar formas de prover maior poder de processamento com menor dissipação de calor, menor consumo de potência e área em chip. Isso vinha sendo feito com o aumento do *clock* dos circuitos. Porém, com a proximidade dos limites físicos dessa abordagem, surgem como solução alternativa as arquiteturas com múltiplos processadores em um único chip: os MPSoC (Multi-Processor System on a Chip). Essa abordagem exige que novas ferramentas e novos softwares sejam desenvolvidos buscando aproveitar ao máximo o aspecto paralelo destas arquiteturas. A indústria de exploração de petróleo tem como uma de suas atividades iniciais a decisão de projetos de exploração de campos de petróleo. Essas decisões são tomadas baseando-se em simulações computacionalmente intensivas, situação em que os MPSoCs podem oferecer aumento de performance através de paralelismo. Este trabalho apresenta a proposta de implementação de um micro-kernel de sistema operacional e bibliotecas auxiliares para a plataforma MPSoC STORM analisando a influência na simulação de reservatórios.

ABSTRACT

The increasingly request for processing power during last years has pushed integrated circuit industry to look for ways of providing even more processing power with less heat dissipation, power consumption, and chip area. This goal has been achieved increasing the circuit clock, but since there are physical limits of this approach a new solution emerged as the multiprocessor system on chip (MPSoC). This approach demands new tools and basic software infrastructure to take advantage of the inherent parallelism of these architectures. The oil exploration industry has one of its firsts activities the project decision on exploring oil fields, those decisions are aided by reservoir simulations demanding high processing power, the MPSoC may offer greater performance if its parallelism can be well used. This work presents a proposal of a micro-kernel operating system and auxiliary libraries aimed to the STORM MPSoC platform analyzing its influence on the problem of reservoir simulation.

SUMÁRIO

1. Introdução 10

1.1. Motivação 11

1.2. Objetivos 12

1.3. Estrutura do trabalho 13

2. Plataforma STORM 14

2.1. Processador SPARC V8 15

2.2. Memória da STORM 16

2.3. Cache 20

2.4. Mecanismo de interconexão 20

2.5. Sistema de Entrada e Saída 21

3. Sistemas operacionais para MP-SoC 25

3.1. Estado da arte em sistemas operacionais para MPSoC 29

3.2. Migração de sistemas operacionais single-core para MPSoC 31

4. O sistema operacional MicroC/OS-II 33

4.1. Visão Geral 33

4.2. Processos no MicroC/OS-II 35

4.3. A troca de Contexto no MicroC/OS-II 35

4.4. A troca de Contexto no Sparc 36

4.5. Escalonamento 37

4.6. Semáforo e Mutex 38

4.7. Gerenciamento de Memória 39

4.8. Portando o MicroC/OS-II 40

5. Projeto do Sistema Operacional para STORM 41

5.1. Escalonamento 41

5.2. Gerenciamento de Entrada e Saída 45

5.3. Gerenciamento de memória 50

5.4. Comunicação Inter-processos 52

5.5. Gerenciamento de Interrupções 54
6. Testes e Avaliação de desempenho 58
6.1. A simulação de reservatórios 58
6.2. Testes com a simulação de reservatórios 58
6.3. Inicialização do sistema operacional 62
7. Conclusões e trabalhos futuros 65
8. Referências 67

LISTA DE FIGURAS

Figura 2.1 - Conexão dos Módulos da STORM 15
Figura 2.2 - Janelas de Registradores do SPARC 16
Figura 2.3 - Leitura na memória compartilhada 18
Figura 2.4 - Módulos no modelo de memória distribuída 19
Figura 2.5 - Topologia NoCX4 21
Figura 2.6 - Nó de E/S IONode 22
Figura 2.7 - Formato dos pacotes de E/S 23
Figura 2.8 - Camadas de E/S 23
Figura 3.1 - Cópia do sistema operacional para cada processador 26
Figura 3.2 - Sistema operacional Mestre-Escravo 27
Figura 3.3 - Sistema Operacional SMP 28
Figura 4.1 - Estrutura de TaskControlBlock 36
Figura 5.1 - Arquitetura de vários Cores 42
Figura 5.2 - Abertura de um arquivo (a) Pedido de abertura (b) Autorização 48
Figura 5.3 – Operaçõe do IONode (a) pedido de leitura/escrita (b) resposta do IONode 49
Figura 5.4 - Memória em partições 50
Figura 5.5 - Alocação de memória dinâmica 52
Figura 6.1 - Instâncias de teste da STORM 61
Figura 6.2 - Total de ciclos com e sem o Sistema Operacional 61
Figura 6.3 - Carga total da NoC 62
Figura 6.4 - Tempo das atividades do boot 63
Figura 6.5 - Ciclos para o boot do MPI por número de processos 64

LISTA DE TABELAS

Tabela 3.1 - Operações da SoCDMMU 31
Tabela 4.1 - Operações de E/S..... 49
Tabela 4.2 - Operações suportadas pelo MPI..... 53
Tabela 4.3 - Tipos de pacote de E/S..... 55

LISTA DE SIGLAS

MPI	Message Passing Interface
MPSoC	Multiprocessor System-on-Chip
NoC	Network-on-Chip
RISC	Reduced Instruction Set Computer
SoC	System-on-Chip
SPARC	Scalable Processor Architecture
STORM	MPSoC Directory-Based Platform
RMS	Rate Monotonic Scheduler
EDF	Earliest Deadline First
MCAPI	Multicore Communication Application Program Interface
IPC	Inter Process Communication
SMP	Symetric Multiprocessor
EPDF	Earliest Pseudo Deadline First
PFAIR	Proportionate-Fair
DMA	Direct Memory Access
SoCDMMU	System On Chip Dynamic Memory Management Unit
CWP	CurrentWindow Pointer

1. Introdução

Com o aumento da demanda por poder de processamento nos últimos anos, seja para aplicações de alto requisito como processamento de vídeos ou gerência de redes de alta velocidade seja para dispositivos embarcados, a indústria de circuitos integrados vem buscando formas de prover maior poder de processamento com menor dissipação de calor, menor consumo de potência e menor área em chip. Até pouco tempo atrás este esforço vinha concentrando-se no aumento da velocidade de *clock* dos elementos processadores e no aumento do número de transistores por área, sustentado pela Lei de Moore.

Além do aumento na demanda de processamento para equipamentos de uso geral e de aplicações de grande porte, o mercado de sistemas embarcados portáteis atraiu a atenção dos desenvolvedores de software e hardware para a busca de soluções cada vez mais eficientes. Estes dispositivos, como os celulares e câmeras, são caracterizados pelas restrições de recursos, tal como área e de consumo de potência, já que estes são alimentados por baterias. O mercado de sistemas embarcados, então, forçou a indústria a utilizar uma solução diferente da que vinha usando nos últimos anos, que exigia cada vez mais consumo de energia e área em chip.

Nos últimos anos grandes empresas perceberam que, ao utilizar o paralelismo, poderiam obter maior ganho de desempenho com menor custo em área e menor consumo de potência. Alternativas ao acúmulo de transistores e ao aumento de *clock* estão tornando-se comuns hoje no mercado, como os processadores de 4 núcleos paralelos em um único chip.

Em (Borkar, Mulder, *et al.*, 2005) é apresentada uma projeção para 2015: a chamada *Many-core Era*, com processadores na casa das centenas de cores, Aponta-se ainda que o aumento no *clock* e a diminuição do tamanho dos componentes usados na fabricação dos atuais processadores está alcançando limites físicos. Estes limites se referem ao aumento de consumo de potência e geração de calor. Além disto, o aumento no *clock* dos processadores não tem sido acompanhado pelo aumento na velocidade de acesso às memórias (Borkar, Mulder, *et al.*, 2005), o que diminui o ganho desta abordagem.

Uma das tarefas mais importantes da área de engenharia de petróleo é a previsão de produção de hidrocarbonetos de reservatórios. Através dessa tarefa é possível, por exemplo, quantificar reservas, avaliar e priorizar projetos de exploração e dimensionar sistemas de produção. Sendo assim, a simulação computacional de reservatórios de petróleo tem se tornado cada dia mais importante para a indústria de exploração de petróleo e gás natural.

A simulação de reservatórios é a ferramenta mais comum utilizada na previsão de produção de hidrocarbonetos. Esta simulação é feita utilizando-se métodos numéricos já que a complexidade impede que métodos analíticos modelem o problema.

A simulação numérica é complexa apenas por sua natureza. Assim, o desenvolvedor do simulador precisa criar aplicações extremamente otimizadas, mas evitando investir tempo em questões específicas do hardware que está sendo utilizado. Para que esse objetivo seja alcançado é necessária uma camada que abstraia do programador os detalhes do hardware, oferecendo uma visão simplificada do gerenciamento de recursos de uma plataforma multi-processada.

A introdução de um conjunto de funções de Sistemas operacionais é capaz de acelerar o desenvolvimento de aplicações e melhorar o aproveitamento dos recursos da plataforma. Entretanto, estes benefícios acarretam em um *overhead* de processamento que deve ser analisado, especialmente em sistemas especialistas como os aplicados à simulação de reservatórios.

Com o surgimento dos processadores *multi-core* muitos sistemas operacionais vêm propondo técnicas para utilizar melhor o paralelismo inerente a essas arquiteturas. No entanto, a mudança do paradigma *single-core* para *many-core* não é simples, pois exige que as aplicações e o sistema operacional estejam prontos a paralelizar suas ações. Um dos grandes desafios dos sistemas operacionais *multi-core* é aproveitar ao máximo este paralelismo.

1.1. Motivação

O desenvolvimento de aplicações atuais para sistemas MP-SoC está altamente atrelado ao hardware, o que diminui a capacidade de reutilização dessas aplicações em plataformas diferentes. Para que seja possível desenvolver aplicações capazes de executar em vários ambientes MP-SoC diferentes com pouca ou nenhuma alteração, é necessário oferecer uma abstração do hardware.

O projeto de sistemas operacionais como máquinas abstratas pode oferecer a abstração necessária não só para aumentar a capacidade de executar aplicações em ambientes diferentes, mas também oferecer uma considerável diminuição da complexidade das aplicações, já que recursos compartilhados podem ser agora gerenciados externamente à aplicação.

Como apontado em (Bridges, Vachharajani, *et al.*, 2008) desenvolver uma aplicação paralela corretamente e que supere em desempenho uma implementação seqüencial é uma tarefa complexa, especialmente por ter de lidar com questões como: *deadlocks*, condições de corrida entre outras. Ao desenvolver uma biblioteca de funções que gerencie estes recursos é possível diminuir a probabilidade de que erros de implementação causem falhas graves nos sistemas.

Estes conjuntos de argumentos também são usados para defender a construção de sistemas operacionais *micro-kernel*, onde apenas um conjunto mínimo de funções está efetivamente executando em modo supervisor. Este trabalho segue esta linha, buscando implementar conjuntos de bibliotecas para gerenciamento recursos que não sejam diretamente ligadas ao *kernel* do sistema operacional. Este conjunto de bibliotecas permite uma maior adaptação do sistema operacional além de possibilitar que estudos e medições possam ser feitas nos módulos independentemente.

1.2. Objetivos

Este trabalho objetiva desenvolver um *micro-kernel* de sistema operacional capaz de gerenciar recursos em um ambiente MP-SoC que utilize NoCs como mecanismo de interconexão. Em conjunto com uma série de bibliotecas de software de baixo nível, para interface com o hardware, tais como: gerenciamento de entrada e saída, gerenciamento de alocação de memória dinâmica e comunicação entre processos.

Este sistema será testado e validado utilizando uma aplicação conhecida da engenharia de petróleo que é a simulação de reservatórios. Não se objetiva porém, melhorar os processos de simulação de reservatório e sim a estrutura sobre a qual este é executado em sitemas MP-SoC. Esta aplicação já foi avaliada do ponto de vista do hardware e dos modelos de memória em (Oliveira, 2009). O trabalho atual estende o primeiro oferecendo o suporte de software necessário ao desenvolvimento de aplicações cada vez mais complexas. Será utilizada a plataforma STORM (REGO, 2006) para efetuar as simulações de execução do sistema e das aplicações que o utilizarão. Como a plataforma STORM ainda não contempla um módulo em hardware para controle de entrada e saída este trabalho deverá também propor e implementar uma arquitetura para este módulo.

Como objetivos específicos este trabalho apresentará:

- Um módulo de entrada e saída em hardware para a plataforma STORM

- A implementação de um escalonador de tempo-real para sistemas MP-SoC
- Uma biblioteca de entrada e saída para abstrair a interação das aplicações com o módulo de entrada e saída
- Um módulo de alocação de memória dinâmica que suporte as instruções *malloc* e *free* do C
- Um mecanismo de comunicação inter-processos baseado no padrão MPI integrado ao sistema operacional

1.3. Estrutura do trabalho

O presente trabalho está dividido da seguinte forma: no capítulo 2 é apresentada a plataforma STORM que será usada como alvo dos testes do sistema operacional a ser implementado por este trabalho, bem como o módulo de E/S desenvolvido para esta. O capítulo 3 exibe o estado da arte em sistemas operacionais para MP-SoC, quais os objetivos dos que existem atualmente e os detalhes de implementação, assim como as características e decisões de projeto de cada um. Já no capítulo 4 é feita a descrição da implementação proposta por esse trabalho. No capítulo 5 temos a apresentação e discussão dos resultados de testes com os vários módulos do sistema operacional. Finalmente no capítulo 6 temos uma discussão dos resultados e propostas de trabalhos futuros.

2. Plataforma STORM

A plataforma STORM (MPSoC Directory-Based Plaftorm) (REGO, 2006) foi desenvolvida com o objetivo de estudar a viabilidade e uso de um projeto baseado em MP-SoC utilizando NoC como mecanismo de interconexão. Esta plataforma vem sendo usada também como base para diversos experimentos, especialmente na exploração de espaço de projeto.

A plataforma STORM foi totalmente implementada usando SYSTEMC (IEEE Standard Society, 2006), uma biblioteca de templates para C++, que permite a construção de simulações de hardware obedecendo a um relógio. Esta ferramenta permite uma descrição funcional do sistema e mantém a habilidade de manter a simulação com precisão de ciclo.

A metodologia usada para o projeto de STORM é conhecida como projeto baseado em plataforma (Sangiovanni-Vicentelli e Martin, 2001). Essa metodologia permite que se desenvolvam módulos integráveis cujas combinações e configurações formam a plataforma. Assim, é possível explorar várias possíveis arquiteturas alterando apenas um conjunto de configurações. Exemplos destas variações de arquiteturas podem ser vistos em (Oliveira, 2009).

Diversos módulos estão disponíveis na plataforma. Exemplos desses módulos são: processador SPARC, memórias cache e o mecanismo de interconexão. Alguns dos módulos existentes serão comentados em maior detalhe nas próximas seções. Além destes módulos já existentes, este trabalho visa propor e desenvolver novos que serão agregados a esta plataforma.

A plataforma STORM pode ser caracterizada como uma plataforma CC-NUMA (*Cache-Coherent Nonuniform Memory Access*) (Tanenbaum, 2008), cada processador tem uma cache de dados e uma cache de instrução controladas por uma estrutura de diretório.

A Figura 2.1 exibe a maneira como os diversos módulos da STORM se interconectam. Há dois mecanismos de interconexão atualmente disponíveis: a NoCX4 e a Árvore Obesa. Cada módulo (processador, ou memória, por exemplo) se conecta a esse mecanismo enviando e recebendo dados através da NoC.

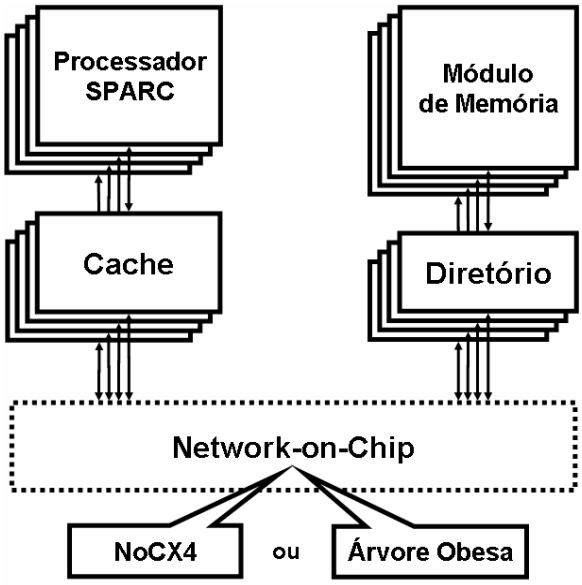

Figura 2.1 - Conexão dos Módulos da STORM

2.1. Processador SPARC V8

O elemento processante da STORM é uma implementação em SYSTEMC do processador SPARC V8 (*Scalable Processor Architecture*) (SPARC International, Inc., 1992). Este processador, de arquitetura RISC (*Reduced Instruction Set Computer*) foi desenvolvido pela *Sun Microsystems* e foi escolhido como elemento processante por ter sua especificação aberta e contar com uma vasta gama de ferramentas disponíveis. Uma das ferramentas disponíveis é o compilador GCC (GNU *Compiler Collection*). A disponibilidade de um compilador para este processador permite que se escreva código em linguagem de alto nível C para a plataforma. O GCC é, então, utilizado para gerar código binário que é carregado na memória da plataforma.

O processador SPARC utiliza uma arquitetura com *pipeline* de cinco estágios: busca, decodificação, execução, memória e escrita. No estágio de busca, a instrução é movida da memória para os registradores do processador; na decodificação os operandos são obtidos dos registradores e a instrução é decodificada; no terceiro estágio (execução) os operandos são transportados para a ULA onde a operação é efetivamente executada. Operações de LOAD/STORE obtêm dados da memória no estágio chamado memória. No último estágio o processador escreve o resultado da operação no conjunto de registradores do processador.

Uma característica importante da arquitetura do SPARC é a janela de registradores. Esse conceito oferece um conjunto de 32 registradores sendo 8 globais e 24 registradores da janela. Quando uma há uma chamada de função no código do SPARC, a janela é mudada.

A janela é composta por três tipos de registradores: registradores de entrada, que contém os valores passados como parâmetro para a função atual; registradores locais, que armazenarão as variáveis locais; e os registradores de saída, usados para a passagem de parâmetros para uma chamada a outra função. Os registradores de saída de uma janela sobrepõem os registradores de entrada da janela seguinte. A Figura 2.2 exibe como os registradores de duas janelas se relacionam.

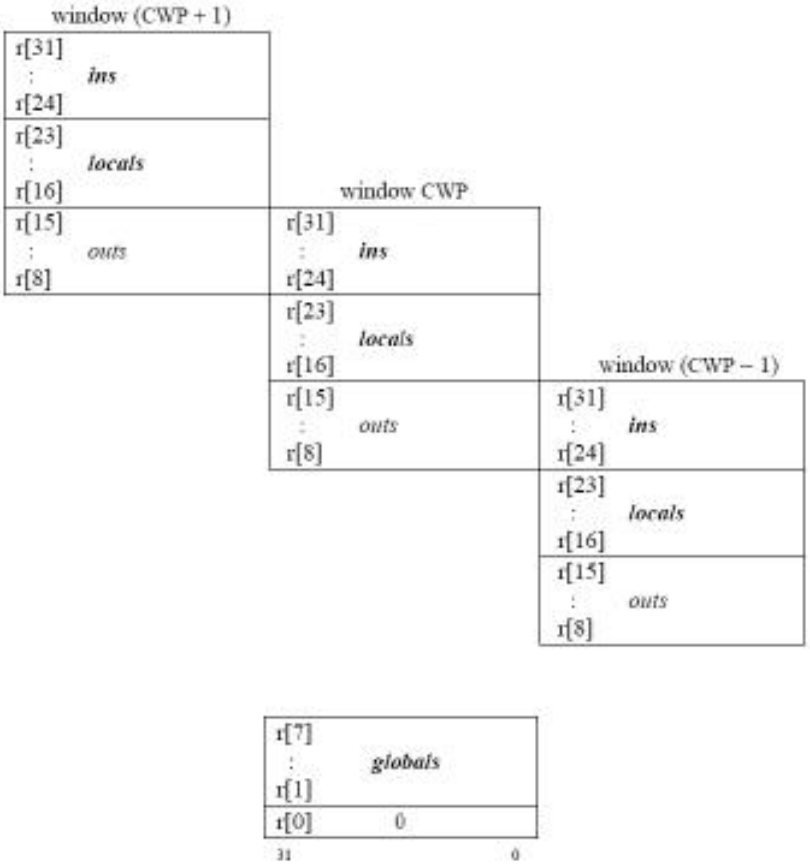

Figura 2.2 - Janelas de Registradores do SPARC

O processador tem um registrador que informa o número da janela atual CWP (*Current Window Pointer*). A implementação atual do SPARC em STORM contém 32 janelas, oito registradores globais e tratamento de interrupções.

2.2.Memória da STORM

Um módulo importante da plataforma STORM é a memória. A localização, o endereçamento e a forma de acesso são características importantes desse módulo. Sendo uma plataforma CC-NUMA, as características citadas da memória afetam diretamente o projeto de software básico para esta plataforma.

Em linhas gerais, pode-se dividir os modelos de memória da STORM em dois: memória compartilhada, onde há um ou vários módulos de memória compartilhando um endereçamento único e contínuo; e memória distribuída, onde cada processador tem um módulo de memória agregado ao qual só este tem acesso. A memória é divida em blocos e acessada usando bloco por bloco. A seguir apresenta-se em mais detalhes os dois modelos de memória atualmente presentes na STORM.

2.2.1. Modelo de memória compartilhada

Este foi o primeiro modelo proposto ainda em (REGO, 2006). Neste modelo, toda a memória é vista pelos processadores em um único espaço de endereçamento e todos processadores podem acessá-la independente do endereço. Restrições de acesso precisam ser feitas em software por um gerenciador de memória (possivelmente em um sistema operacional).

Um programador não precisará conhecer a localização de um módulo de memória para acessá-lo, já que todo o acesso é feito usando instruções LOAD e STORE comuns, do próprio processador SPARC V8. Cada módulo de memória pode ter um tamanho diferente, definido no momento da configuração de uma instância da plataforma, além de poder ser instanciado em qualquer local da NoC, exceto no endereço (0,0). Este endereço é reservado para o processador de boot. O tamanho mínimo de um módulo de memória é um bloco. Bloco é também a unidade de transferência de dados na plataforma, e o tamanho do bloco é configurável.

O acesso a um dado no modelo de memória compartilhada é executado como na Figura 2.3. Inicialmente, o processador faz o pedido à uma das caches podendo ocorrer um cache *hit* (quando o dado já estiver disponível na cache) ou cache *miss* (quando o dado não estiver na cache). No caso de cache *miss*, o CaCoMa (*Cache Communication Manager*) deverá efetuar a tradução do endereço lógico de memória para um endereço de NoC e enviar um pacote solicitando o bloco em questão.

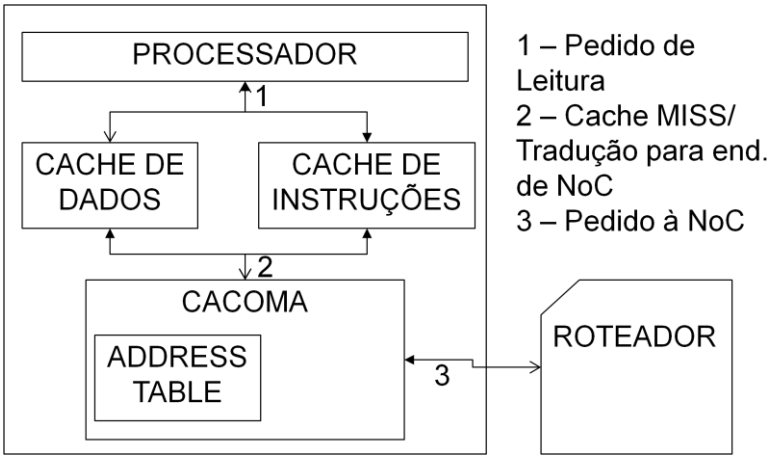

Figura 2.3 - Leitura na memória compartilhada

2.2.2. Modelo de memória distribuída

Outro modelo de memória, proposto e implementado por (Oliveira, 2009), é o modelo de memória distribuída. Neste modelo, cada processador tem um espaço de endereçamento próprio e uma memória dedicada, a qual apenas este processador tem acesso. Esse modelo foi implementado e testado em conjunto com uma biblioteca seguindo o padrão MPI (*Message Passing Interface*)(MPI Forum, 2008), que permite que os processadores se comuniquem através de mensagens enviadas pela rede em chip.

A conexão dos módulos no modelo de memória distribuída pode ser vista na Figura 2.4. A cache é separada em cache de dados e cache de instruções. Para este modelo, dois novos módulos foram adcionados: o DAMa (*Data Access Manager*) e o CoMa (*Communication Manager*).

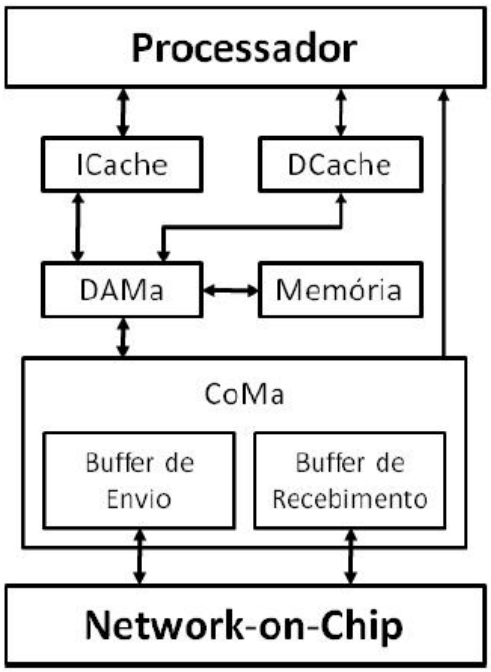

Figura 2.4 - Módulos no modelo de memória distribuída

Todo acesso a dados externos às caches é feito pelo DAMa. Cada módulo ligado ao DAMa está mapeado em uma faixa de endereço permitindo, assim, que o processador e as caches enxerguem os módulos externos como memórias. Dessa forma, as operações utilizadas serão sempre do tipo LOAD/STORE, e o DAMa pode ser utilizado para conectar os mais diversos módulos, desde blocos de memória até dispositivos de entrada e saída. A comunicação com outros processadores também é tratada como um espaço de memória e é gerenciada pelo CoMa.

O processador utiliza o DAMa para realizar comunicação com os módulos externos, isso inclui o CoMa. Esta comunicação, do processador com o CoMa, é possível pois os registradores do CoMa estão mapeados em um endereço de memória conhecido pelo DAMa, e estão descritos a seguir:

- AvailableSendBuffer: informa o espaço disponível no buffer de envio;
- AvailableRecBuffer: informa a quantidade de dados disponíveis no buffer de recebimento;
- SendBuffer Data: utilizado para escrever dados no buffer de envio;
- RecBuffer Data: utilizado para acessar os dados armazenados no buffer de recebimento;

2.3.Cache

Na plataforma STORM a cache é dividida em dois módulos: a cache de dados, chamada DCache; e cache de instruções, chamada ICache. Considerando que o processador utilizado (SPARC V8) suporta *pipeline,* esta decisão de projeto oferece grandes vantagens pois permite acesso simultâneo a instruções e dados. Essas caches podem ser configuradas separadamente com informações como tamanho total, tamanho de um bloco (linha de cache) e algoritmo de substituição de blocos.

Dentro do *pipeline* do SPARC V8, a cache de instruções é acessada apenas pelo estágio de busca de instrução. Como apenas instruções de leitura são permitidas, essa cache tem uma implementação mais simples em relação à cache de dados. A cache de dados, porém deve oferecer total suporte ao protocolo de diretório usado no sistema.

No modelo de memória distribuída, a cache se comunica com o DAMa que, por sua vez,se comunica com a memória para obter o dado requisitado ou para gravar a informação. em caso de uma operação de STORE. Para acesso externo, o DAMa programa o CoMa através de seus registradores para que este envie os dados em forma de pacote para o nó destino usando a NoC.

2.4.Mecanismo de interconexão

A STORM utiliza como mecanismo de interconexão uma NoC (*Network on Chip*). Em (REGO, 2006) é apresentada a NoCX4 que tem uma topologia em grelha de roteadores, como exibido na Figura 2.5. Nesta topologia, cada módulo é ligado a um roteador de modo que o endereço de um módulo é a localização cartesiana do roteador ao qual este está ligado.

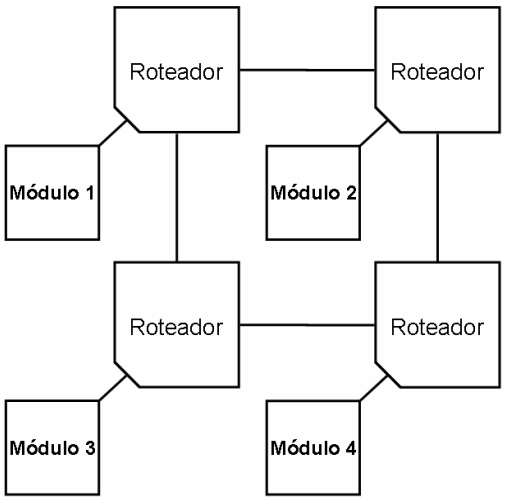

Figura 2.5 - Topologia NoCX4

Os pacotes da NoCX4 são compostos de palavras de 32 bits, com uma palavra inicial de cabeçalho e várias palavras de *payload* (carga de dados efetiva). Este conjunto de palavras pode ter tamanho variável sendo, inclusive, zero (pacotes apenas com cabeçalho).

2.5. Sistema de Entrada e Saída

A plataforma STORM conta hoje com um mecanismo de entrada e saída, já fruto deste trabalho, baseado em interrupções. Este dispositivo é agregado a um dos nós da NoC, não havendo restrição espacial. A qualquer nó da NoC pode ser agregado este mecanismo de entrada e saída. Este novo módulo pode ser tratado como um nó comum da NoC tendo capacidade de se ligar a um dispositivo de Entrada e Saída, como por exemplo, uma porta serial.

A Figura 2.6 exibe os principais elementos deste módulo. A interface com a NoC é feita através de dois elementos: o IONode Sending Buffer e o IONode Receiving Buffer. Os dois elementos são necessários à comunicação com a NoC, já que esta usa o controle de fluxo baseado em créditos. Este controle de fluxo é baseado no espaço disponível nestes buffers.

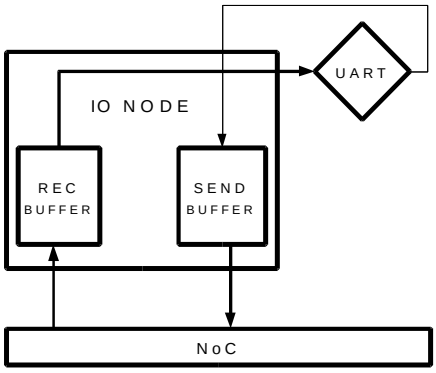

Figura 2.6 - Nó de E/S IONode

Quando um pacote chega ao IONode, inicialmente ele é armazenado no IONode Receveiving Buffer, numa fila para que possa ser atendido no momento adequado. Ao ser atendido, o pacote é decodificado pelo IONode que envia os comandos necessários ao dispositivo de E/S. Ao fim da operação de E/S no dispositivo, este envia os dados ao IONode que armazena no IONode Sending Buffer preparando-o assim para a transmissão pela NoC para o nó de destino.

O módulo UART é uma simulador de um dispositivo de E/S, onde é possível anexar um dispositivo externo. Na simulação implementada em SystemC este dispositivo é representado por um arquivo em disco, do qual o simulador pode ler e escrever dados a medida que o módulo UART recebe os comandos do IONode.

Os buffers estão ligados ao IONode, que é responsável por gerenciar a fila de pedidos de E/S vindos dos nós processadores. Quando um pedido é enviado de um nó processador para o nó de entrada e saída IONode, este armazena o pedido em uma fila interna e responde os pedidos seguindo a ordem de chegada (FIFO). No pedido, são informadas: (i) a operação a ser realizada; (ii) o endereço a ser acessado; (iii) o nó de origem do pedido.

Ligado ao IONode está um ou mais dispositivos de entrada e saída. Estes dispositivos podem ser de bloco ou de caractere. O IONode identifica para qual dispositivo deve ser enviado o pedido através do endereço contido no pacote de pedido de E/S. Assim, os pacotes de E/S trafegam na NoC com o cabeçalho, como definido na figura Figura 2.7. Isto permite que se anexe outros tipos de dispositivos, em pontos diferentes da NoC.

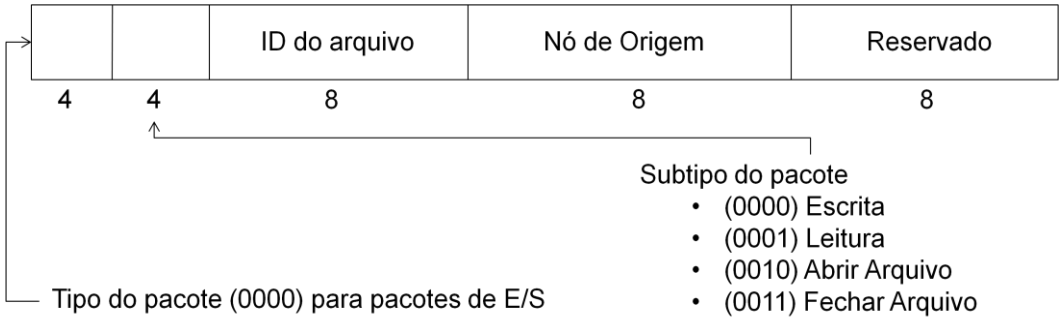

Figura 2.7 - Formato dos pacotes de E/S

Os pacotes de E/S são formados por 3 campos: Operação, Endereço e Nó de Origem. A camada de software que trata esses pacotes, presente nos nós processadores, deve conhecer este formato para gerar os pacotes e lê-los no momento da resposta. Um programador deve então fazer as leituras e escritas usando este padrão de pacotes. Com o uso do sistema operacional, o programador fica livre de conhecer estes formatos, usando apenas chamadas de alto nível através das bibliotecas de E/S disponibilizadas.

Uma vez que um sistema operacional seja usado para gerenciar os recursos do MPSoC, deverá também oferecer recursos que facilitem a interação com os dispositivos de E/S. Essa interação é feita pelas bibliotecas de E/S. Essencialmente a interação do usuário com o sistema de entrada e saída, usando um sistema operacional, pode ser dividida em 4 camadas, como exibido na Figura 2.8: Software de nível de usuário, Software Independente de dispositivo, *Drivers* e o *Firmware* do hardware efetivo.

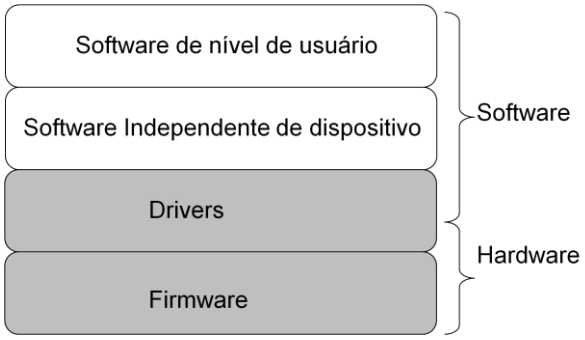

Figura 2.8 - Camadas de E/S

A plataforma STORM implementa os 2 níveis mais baixos dessa abordagem: o *Firmware* e o *Driver* de dispositivo, implementados em hardware. O *Firmware* está implementado no próprio dispositivo de entrada e saída, sendo responsável por interpretar os comandos enviados pelo *driver* para o dispositivo. Já o *driver* é dependente do sistema operacional. Na STORM, o *driver* é responsável por adicionar o cabeçalho do sistema operacional aos pacotes que serão enviados pela NoC. Para tanto, este deve estar distribuído no nó processador e no IONode.

Uma das aplicações que podem ser encontradas para as NoCs e os MPSoC são os roteadores de redes de alta velocidade (TILERA, 2009). Esses equipamentos tem uma demanda de E/S muito grande, portanto a comunicação com os nós de E/S deve ser otimizada para alcançar os altos níveis de exigência destes dispositivos.

3. Sistemas operacionais para MP-SoC

Muitos trabalhos, no campo acadêmico e da indústria, (Nollet, Verkest e Kapeldreef, 2007) vêm apresentando propostas de implementação de sistemas operacionais e de bibliotecas de gerência de recursos para MP-SoC. Os trabalhos apresentados concentram-se em gerenciar as tarefas dos processadores, estando em diferentes níveis de integração com o hardware da plataforma de execução.

Recentemente um conjunto de empresas uniu-se para formar a *MulticoreAssociation*, cujo objetivo é oferecer um padrão para o desenvolvimento de hardware MP-SoC. Os focos principais são nas questões de gerência de recursos, comunicação entre os diversos processadores, virtualização e práticas de programação. Esta associação entregou recentemente o primeiro desses padrões, a MCAPI(Moyer, 2008) (*Multicore Communication Application Program Interface*) que é um padrão de IPC (*Inter Process Communication*). Essa especificação é aberta e está disponível para download no site da associação.

Para usar a MCAPI é necessário definir a topologia do MP-SoC em tempo de compilação, dispensando então a necessidade de um protocolo de descoberta de nomes. Na prática, um nó pode ser então uma *thread*, um processador, ou um ASIC por exemplo. Com essa abstração é possível então usar a MCAPI em variados MP-SoCs.

O projeto de um sistema operacional para MP-SoC deve balancear uma série de características desejáveis, entre eles (Nollet, Verkest e Kapeldreef, 2007) destaca: baixo consumo de potência, alto desempenho, facilidade de programação, previsibilidade, flexibilidade e escalabilidade.

Um aspecto importante no desenvolvimento de sistemas operacionais para múltiplos processadores é a centralização de serviços do sistema operacional em um único núcleo. Em (Tanenbaum, 2008) são apontadas 3 categorias para a divisão da responsabilidade de execução dos serviços de sistema operacional. Estas categorias podem ser resumidas como: (i) Uma Cópia do sistema operacional para cada processador; (ii) Mestre Escravos; (iii) SMP (*Symetric Multiprocessor*).

No modelo de uma cópia do sistema operacional por processador, sistemas independentes executam suas tarefas e toda a sincronização e paralelismo ficam por conta de uma biblioteca de comunicação externa ao sistema operacional. Como os processos em cada núcleo são separados e coordenados por rotinas de sistema operacional diferentes, não há compartilhamento de cache ou memória, como exibido na Figura 3.1.

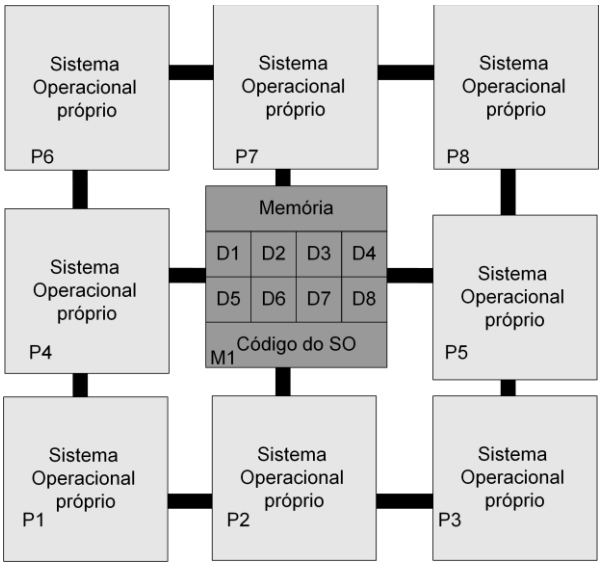

Figura 3.1 - Cópia do sistema operacional para cada processador

Na prática todas as estruturas internas do sistema operacional precisarão ser copiadas para cada processador. Sendo assim, cada processador tem uma instância completa do sistema operacional e a biblioteca de comunicação precisará prover mecanismos de coerência de cache. A implementação deste tipo de sistema operacional é simples já que pode ser implementado como um sistema operacional para uma arquitetura *single-core* e apenas replicado entre os processadores. Como o sistema operacional conhece apenas a CPU em que está executando, não poderá migrar uma tarefa para outra CPU. Este fato pode ocasionar um desbalanceamento da carga entre as CPUs, fazendo com que uma CPU esteja sobrecarregada enquanto outra se encontra sem atividade.

No modelo Mestre Escravos, exibido na Figura 3.2, o sistema operacional estará centralizado executando sempre na mesma CPU . Este modelo permite grande compartilhamento de recursos entre os processos executando no momento, mesmo que estes estejam em CPUs diferentes. Isto porque o sistema operacional central tem conhecimento de quais processos estão executando em cada CPU.

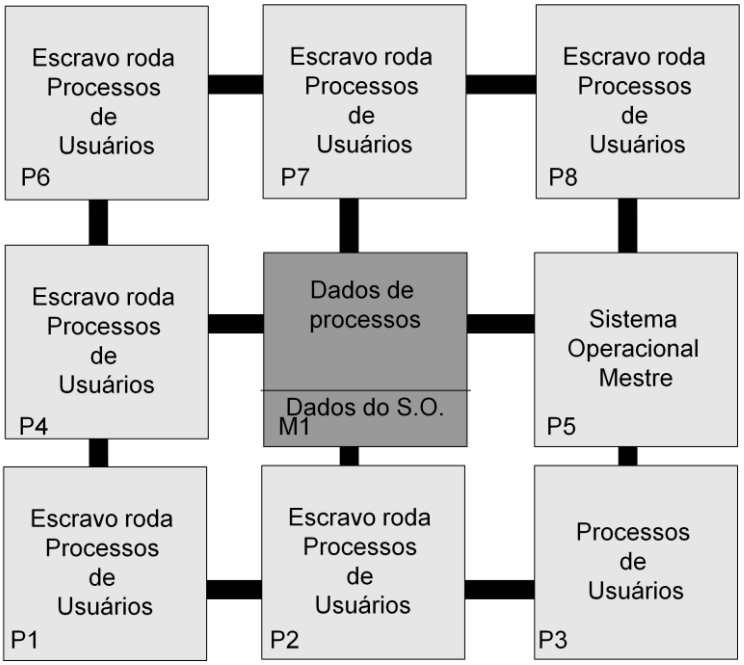

Figura 3.2 - Sistema operacional Mestre-Escravo

Sempre que um processo efetuar um pedido de serviço do sistema operacional, a CPU mestre irá executar a rotina pedida e devolver o resultado à CPU que solicitou. Há concorrência mínima para acesso às estruturas de dados do sistema operacional, porém, alta concorrência para a comunicação com a CPU Mestre, o que pode gerar um gargalo para este modelo. Como o sistema operacional conhece todas as CPUs é possível balancear a carga migrando tarefas entre as diversas CPUs.

O modelo SMP (*Symetric Multiprocessor*) mantém as estruturas de dados do sistema operacional em uma memória compartilhada como pode ser visto na Figura 3.3. Estas estruturas são acessíveis por todos os núcleos processantes e as rotinas de serviço do sistema operacional podem ser executadas por qualquer CPU. Quando um pedido é feito por um processo ao sistema operacional, a própria CPU, onde este processo está rodando, pode fazer uma troca de contexto do processo para o *kernel*, executando a rotina que for necessária. Este modelo ajuda a balancear a carga entre as CPUs, evitando também o problema de centralizar pedidos do sistema operacional a um único núcleo.

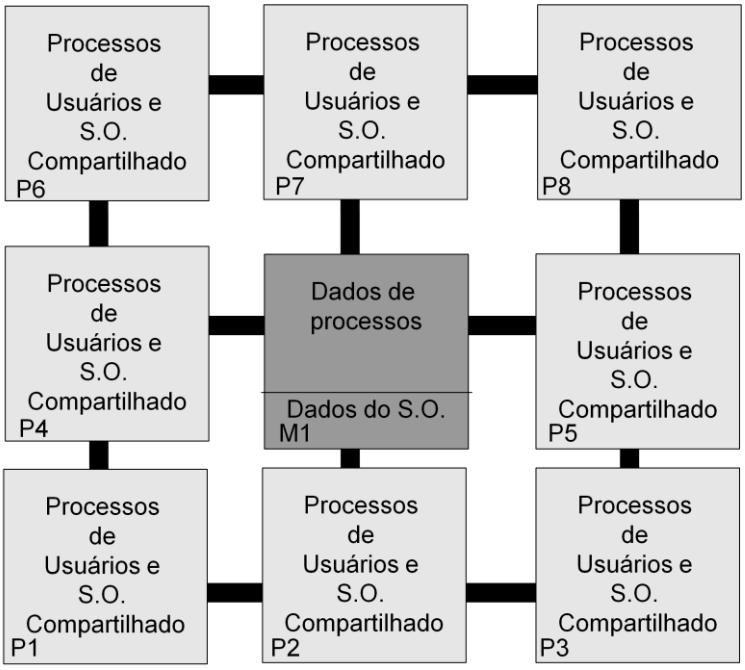

Figura 3.3 - Sistema Operacional SMP

Como o código do sistema operacional pode estar sendo executado simultaneamente em uma ou mais CPUs, é preciso proteger, através de áreas de exclusão mútua, as estruturas de dados do sistema operacional. Para implementar esta exclusão, um projetista poderia, sempre que uma CPU estiver executando código de *kernel*, não permitir que outras CPUs o fizessem. Porém, isto penalizaria muito o paralelismo das CPUs (aproximando-se do modelo mestre-escravos), O projetista deverá, então, buscar usar estruturas de exclusão mútua apenas nos trechos minimamente necessários.

O desenvolvimento deste tipo de sistema torna-se então mais complexo, já que acessos às estruturas de sistema operacional podem levar a inconsistências se não forem tratadas as questões de acesso concorrente. Condições de deadlock também podem ocorrer devido ao uso mais freqüente de estruturas de alocação e liberação de recursos. De acordo com (Tanenbaum, 2008), esta estratégia é bastante usada hoje já que permite um maior aproveitamento do paralelismo dos núcleos.

Na implementação efetuada por este trabalho, uma abordagem híbrida é utilizada. Nesta abordagem, recursos locais como a alocação de memória e escalonamento de processos são gerenciados pelo sistema operacional local, enquanto recursos globais como mecanismos de entrada e saída e comunicação inter-processos são gerenciados em um ou mais nós mestre.

3.1. Estado da arte em sistemas operacionais para MPSoC

O projeto Tesselation(Liu, Klues, *et al.*, 2009) propõe um sistema operacional que gerencie recursos de *multi-core,* considerando partições e garantindo disponibilidade mínima para estas partições. Uma aplicação poderá, então, solicitar um conjunto de processadores, assim como hoje solicita uma área de memória. Esse conjunto de processadores será garantido, podendo o sistema operacional, então, alocar uma série de processadores a uma única tarefa. Com a crescente capacidade de virtualização, o próprio hardware poderá oferecer um conjunto de processadores virtuais.

O sistema operacional ENEA OSE (Strömblad, 2009), desenvolvido para aplicações de gerenciamento de redes de alta velocidade, oferece um *micro-kenel* centralizado em uma única CPU (modelo mestre-escravo). As tarefas são distribuídas entre as CPUs, podendo disponibilizar serviços em CPUs diferentes de onde o kernel está localizado. Como exemplo desses serviços, podem ser citados o gerenciador de sistemas de arquivos e o gerenciamento da pilha TCP/IP). Entretanto cada CPU terá um conjunto de tarefas atribuídas a ela e um escalonador próprio.

A comunicação entre as CPUs é feita por uma biblioteca de IPC (*Inter Process Communication*) especificamente desenvolvida para trabalhar em conjunto com o sistema ENEA. Essa estratégia diminui (mas não elimina) a quantidade de condições de corrida e o gargalo do processador mestre, já que distribui o serviço do escalonador. Contudo, penaliza a capacidade de balanceamento de carga entre diferentes CPUs.

Uma abordagem SMP pode ser encontrada no Linux que inclui em seu escalonador a capacidade de lidar com configurações SMP(Bovet e Cesati, 2000). As chamadas do sistema operacional poderão ocorrer em qualquer CPU e serão executadas na CPU que fez a chamada (modelo SMP). Como uma forma de melhorar o número de cache *hits,* o escalonador usa o conceito de afinidade do processador. A idéia é que, ao substituir um processo, o novo processo deve sempre tentar ser alocado à CPU que o executou na última vez. Para obter este resultado, o escalonador mantém o identificador da última CPU na qual o processo foi executado. Sempre que este processo estiver na fila para ser executado, ganhará um bônus em sua prioridade se for alocado para a CPU que o executou por último.

Em (Paulin, Pilkington, *et al.*, 2004) é apresentado o sistema ST Multiflex, uma proposta de implementação de um módulo de hardware para gerenciamento de recursos do MP-SoC. Esta proposta dá suporte a uma biblioteca de troca de mensagem em uma configuração SMP. O sistema Multiflex define ainda uma interface com um compilador próprio capaz de integrar módulos heterogêneos no MP-SoC. Esta interface possibilita programação em linguagens de alto nível (C/C++) para a plataforma. Os processadores de propósito geral executam um sistema Linux e o MP-SoC conta também com elementos DSP e ASIC. Esses elementos são controlados pelos processadores de propósito geral, combinando então os modelos SMP e Mestre-Escravo.

Outra proposta de implementação em hardware chamada RealFast RTU (*Real Time Unit*) é apresentada em (Klevin, 2003). Esta proposta oferece um bloco IP contendo as funções de gerenciamento de recurso, onde a RTU é ligada a um sistema operacional qualquer. A erencia os processadores do MP-SoC, a comunicação entre a RTU e o MP-SoC é feita através de registradores mapeados em memória e interrupções.

O sistema AsyMOS(Nollet, Verkest e Kapeldreef, 2007) separa tarefas específicas para cada elemento processante do MP-SoC, caracterizando-se portanto como um sistema operacional do tipo mestre escravo. Assim, alguns processadores ficarão responsáveis por tarefas de gerenciamento de recursos, enquanto outros ficarão responsáveis apenas por executar código de aplicações. Essa abordagem permite que se aproveite melhor o conteúdo das caches e que se divida o gargalo inerente ao modelo mestre-escravo entre vários processadores.

Os sistemas K42 e MITExokernel(Nollet, Verkest e Kapeldreef, 2007) oferecem *micro-kernels* e um conjunto de módulos adicionais que podem ser adicionados e retirados do *kernel*, oferecendo alta adaptabilidade do sistema operacional a diferentes projetos. Além disto, os gerenciadores de recursos dispõem de estruturas de dados separadas, evitando a concorrência e aumentando a flexibilidade do sistema operacional.

Uma unidade de gerenciamento de memória implementada em hardware, chamada SoCDMMU (*System on chip Dynamic Memory Management Unit*), é proposta por (Shalan e Mooney, 2000). Esta unidade interage com o sistema operacional criando um gerenciamento de memória em dois níveis: o primeiro nível em hardware e o segundo em software, feito pelo sistema operacional.

A SoCDMMU oferece um conjunto de operações implementadas em hardware para alocação de memória em blocos, Estas operações são relativas a: acesso exclusivo, somente leitura e leitura escrita.As operações são como apresentado na Tabela 3.1.

Tabela 3.1 - Operações da SoCDMMU

Operação	Descrição
G_alloc_ex	Aloca n blocos como acesso exclusivo
G_alloc_rw	Aloca n blocos como acesso de leitura e escrita
G_alloc_ro	Aloca n blocos como acesso de somente leitura
G_dealloc	Desaloca um conjunto de blocos

Em (Shalan e Mooney III, 2002) é apresentada a adaptação do sistema Atalanta para usar a SoCDMMU. Este sistema operacional utiliza alocação estática. A memória é dividida em partições de tamanho fixo no momento do boot do sistema de forma que uma aplicação pode requisitar uma ou mais dessas partições.

3.2.Migração de sistemas operacionais single-core para MPSoC

A migração da base de sistemas operacionais *single-core* para um ambiente *multi-core* é possível utilizando-se uma das três abordagens de sistema operacional distribuído, exibidas na seção Sistemas operacionais para MP-SoC. Para cada abordagem é necessário um grau diferente de adaptação. À medida que o sistema operacional se torna mais distribuído, as possíveis condições de corrida entre os recursos compartilhados do sistema operacional se tornam mais comuns.

Os mecanismos para evitar estas condições de corrida, devem ser adaptados, buscando diminuir a granularidade dos *locks* envolvidos. Situações que exijam exclusão mútua em todo o MPSoC são, então, gerenciadas por um mecanismo diferente do que se tem para um único núcleo.

Os sistemas operacionais *single-core* atuais podem ser transportados com poucas alterações para o mundo *multi-core* se usado o modelo de uma cópia por processador. Porém, como já demonstrado anteriormente, esse modelo dificulta o compartilhamento de informações entre os núcleos e o compartilhamento de recursos do MP-SoC como um todo.

O modelo mestre-escravo permitiria um controle centralizado, melhor balanceamento de carga e compartilhamento de recursos. O número de alterações no núcleo de um sistema operacional *single-core* para suportar este tipo de adaptação não deverá ser muito grande e se concentraria essencialmente nos algoritmos de escalonamento de processos e métodos de IPC.

Já para o modelo SMP, que oferece grandes vantagens em relação a os modelos anteriores, o núcleo do sistema operacional teria que ser reescrito. Isto para garantir que as estruturas de dados compartilhadas por diversos serviços do sistema operacional estejam protegidas. E ainda, garantir que as funções possam ser reentrantes, evitando problemas de condição de corrida entre os diversos processadores executando código em nível de *kernel* simultaneamente. Essa reescrita do núcleo poderia claramente inviabilizar o uso desses sistemas *single-core* em MP-SoCs.

4. O sistema operacional MicroC/OS-II

Historicamente, o desenvolvimento de um sistema operacional começa a partir de uma base simples. Exemplos desse processo são o desenvolvimento de sistemas como o Linux, baseado no Minix; do próprio Minix baseado no UNIX; ou do Windows baseado no DOS. Partindo deste mesmo princípio, o sistema operacional para a plataforma STORM foi iniciado usando como base o sistema MicroC/OS-II.

4.1. Visão Geral

O MicroC/OS-II é um sistema operacional de tempo real escrito usando a linguagem C seu principal obejtivo é ser um kernel de sistema operacional, leve, de tempo real e facilmente portável para novas arquiteturas, apesar de não ser um sistema operacional open-source o seu autor oferece o código fonte do sistema para fins de estudo e acadêmicos, sem custo. O código está disponível também para uso comercial sob um licença definida em(Labrosse, 2002).

O autor do kernel Jean Labrosse tendo desenvolvido-o quase totalmente em C publicou a primeira versão na revista Embedded System Programming enviando junto com o artigo todo o código fonte do kernel, em 1992 um livro do mesmo autor exibia em detalhes a implementação e o código do kernel comentado. Desde então MicroC/OS-II já foi portado para mais de 40 plataformas diferentes no mundo todo. Uma empresa foi construída em torno deste sistema a Micrium que hoje oferece a licença de uso comercial do MicroC/OS-II e suporte aos desenvolvedores.

O código do MicroC/OS-II tem aproximadamente 5500 linhas de código e é disponibilizado junto com o livro(Labrosse, 2002) a vantagem desta maneira de distribuição é que é possível oferecer de maneira consistente um manual completo e bem descritivo do funcionamento interno do kernel. As principais características do kernel estão detalhadas no manual para fácil consulta.

A maior parte do código do MicroC/OS-II foi escrita em ANSI-C, onde apenas partes muito específicas foram escritas em assembly de um processador. O requisito necessário para portar o MicroC/OS-II é apenas que o processador em questão tenha um registrador apontador da pilha e que os outros registradores possam ser salvos e carregados nesta para possibilitar a troca de contexto. O processador usado neste trabalho, o SPARCV8, tem estas características e pode, portanto ser usado para tal. Deve ser necessário também um compilador capaz de

executar partes assembly inline, para permitir ativar e desativar interrupções do processador.

O MicroC/OS-II pode ser executado em processadores de 8, 16, 32 e 64 bits em microcontroladores e em DSPs. A empresa desenvolvedora do MicroC/OS-II a Micrium oferece vários ports para vários processadores, atualmente existem cerca de 40 ports do MicroC/OS-II. O kernel é capaz de rodar a partir de um dispositivo ROM o que permite portanto usá-lo como sistema operacional embarcado, exigindo apenas que as estruturas de controle do sistema operacional e a área da pilha estejam em uma memória volátil.

Outra característica importante deste kernel é que ele é escalável, no sentido que vários serviços podem ser ativados e desativados permitindo que se adapte a plataformas com necessidades e poder de processamento diferentes. Assim, retirando serviços, é possível diminuir os requisitos de memória tanto ROM quanto RAM para uma determinada instância do MicroC/OS-II.

O escalonador do MicroC/OS-II é preemptivo o que significa que se uma tarefa de maior prioridade está pronta para ser executada, as tarefas de mais baixa prioridade esperarão o término da primeira, isso garante que tarefas de alta prioridade sempre atenderão ao deadline que é imposto, é dever do desenvolvedor definir as prioridades a partir do deadline de cada tarefa. O limite atual do número de tarefas é de 64, no caso da implementação atual teremos até 64 tarefas por núcleo processador na NoC o que nos oferece então um limite teórico de 64 X 256 = 16384 processos simultâneos em STORM.

Sendo um kernel de tempo real suas tarefas internas tem tempo determinístico, todas as chamadas de sistema dos kernel têm um tempo de resposta fixo, o qual não depende do número de tarefas sendo executadas no momento. Isso permite que o programador execute quantas tarefas forem necessárias sem se preocupar com a implicação da carga na velocidade de resposta do kernel. Evidentemente há uma preocupação com os deadlines de cada tarefa e a ocupação do processador por estas, se o número de tarefas supera a capacidade do processador de dar vazão a estas o sistema perderá os deadlines das tarefas de tempo real.

As informações de uma tarefa são armazenadas na pilha de execução desta o kernel permite que se tenha pilhas de tamanhos diferentes para tarefas diferentes o que economiza memória RAM e permite um gerenciamento melhor das necessidade de RAM de cada tarefa.

Vários serviços já estão implementados pelo MicroC/OS-II outros foram adicionados neste trabalho, alguns dos serviços pré-existentes são os mutex e semáforos além dos serviços de gerenciamento de tempo como *OSTimeDly* que permite que uma tarefa durma por uma

quantidade de tempo especificada em mili-segundos.

O tratamento de interrupção do MicroC/OS-II permite que uma interrupção de hardware desbloqueie uma tarefa trazendo essa à execução, além disto até 255 interrupções podem ser aninhadas. O kernel foi certificado pela *Federal Aviation Administration* (FAA) para uso em aviões comerciais, tendo atendido ao padrão RTCA DO-178B para software de equipamentos de aviação. Este certificado é importante pois garante que o código foi testado e validado para uso em aplicações de tempo real críticas que incluam questões de segurança para a vida de seres humanos.

4.2. Processos no MicroC/OS-II

Processos ou tarefas, como são conhecidos no MicroC/OS-II, são programas em execução no sistema, compostos de varáveis locais e estruturas de dados que armazenam informações de controle do próprio sistema operacional. Neste kernel uma tarefa pode estar em um dos 5 estados diferentes: *Waiting, Ready, Running, ISR, Dormant*.

Waiting: Uma task que se encontre no estado Waiting está bloqueada aguardando que algum evento ocorra, seja a liberação de um recurso ou uma interrupção do timer por exemplo.

Ready: Uma task Ready está pronta para a execução esperando ser alocada ao processador.

Running: é atingido quando uma tarefa está atualmente alocada ao processador sendo, portanto capaz de executar suas instruções.

ISR Running: Uma tarefa pode ter sido chamada a execução para o tratamento de uma interrupção neste momento diz-se que a tarefa está em estado de ISR Running.

Dormant: O último estado possível para uma tarefa é Dormant que significa que a tarefa não está pronta para executar nem está aguardando um evento, a tarefa terminou seu código es está prestes a ser eliminada do sistema.

4.3. A troca de Contexto no MicroC/OS-II

Como em qualquer outro sistema operacional o MicroC/OS-II exige, na troca de contexto, que todos os dados da tarefa atualmente em execução sejam salvos para que outra possa ser chamada à execução. O kernel oferece uma estrutura de dados para armazenar os dados básicos de uma tarefa, esta estrutura é chamada de TCB (Task Control Block) e pode

ser vista na Figura 4.1 sendo constituída basicamente de uma lista ligada de estruturas que armazenam informações como: Ponteiro para a pilha, Tamanho da pilha, Lista de eventos aguardados pela tarefa, Prioridade, Área de extensão entre outros.

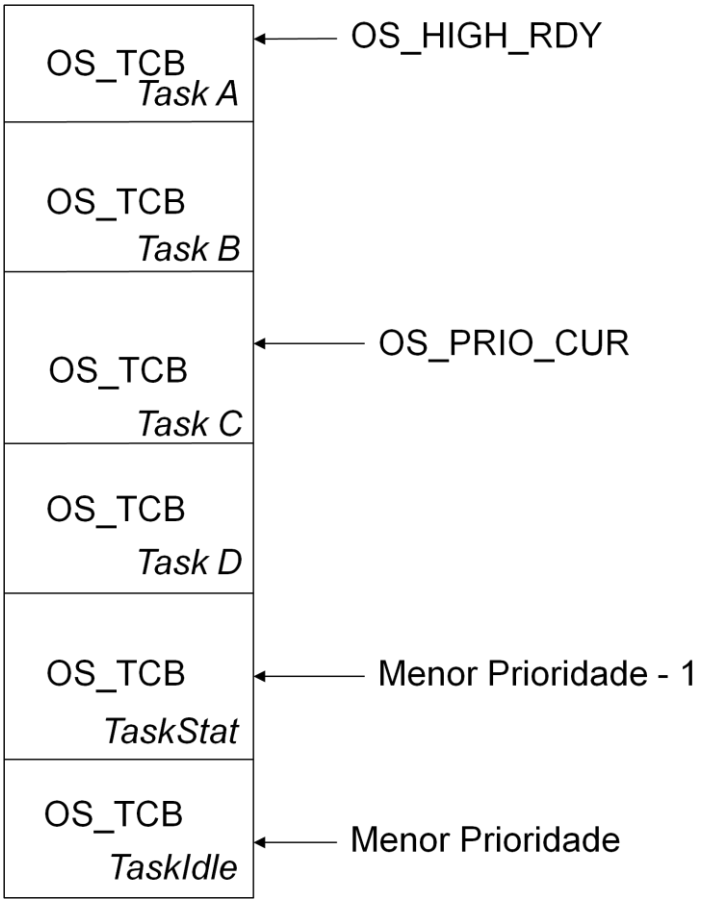

Figura 4.1 - Estrutura de TaskControlBlock

Além do TCB uma pilha contendo as variáveis locais e os registradores é usada, o kernel pode ser configurado para arquiteturas onde a pilha cresce do endereço maior para o menor ou o inverso, esta é uma configuraçào simples feita através de instruções #DEFINE do próprio kernel.

4.4.A troca de Contexto no Sparc

Uma peculiaridade importante dos processadores Sparc é a maneira como os registradores são organizados. Há no Sparc 3 conjuntos de registradores o registradores Locais, Globais e os registradores In e Out. Em um dado momento o processador vê um conjunto desses registradores que formam uma janela, sempre que uma chamada de função é feita, ao invés de salvar os registradores na pilha, com uma instrução específica como um push, a janela de registradores visível é movimentada (usando uma função SAVE) de modo que os registradores do tipo Out sejam agora enxergados como registradores do tipo

permitindo que parâmetros sejam passados à próxima função através deste mecanismo. Quando a função retorna, basta efetuar a instrução RESTORE, os dados que haviam sido passados pelos registradores IN serão então retornados à função chamadora.

Este mecanismo é ótimo para linguagens de programação onde a chamada de pequenos procedimentos é comum já que, uma chamada de função não exige que se salve todos os registradores na pilha. Porém para salvar totalmente o contexto de uma tarefa seria necessário salvar todas as janelas existentes no SPARC, a implementação atual do SPARCV8 de STORM tem 32 janelas.

A decisão então para evitar esse overhead excessivo na troca de contexto foi utilizar da técnica do *lazy restore*. Para isso é necessário o tratamento de especial de duas interrupções do processador SPARC as interrupções são *windowUnderflow* e *windowOverflow* e um tratamento especial ao flag do processador WIM (*Window Invalid Mask*). O WIM é um registrador de 32 bits que utiliza uma máscara de bits indicando quais janelas são válidas para uso a janela cujo bit em *WIM* é 1 está inválida para o uso.

O mecaninsmo de *lazy restore* usa dessa característica para marcar janelas que ainda não foram salvas na troca de contexto, desse modo é necesário apenas salvar uma janela na troca de contexto e marcar as outras janelas como inválidas, quando uma operação do tipo SAVE é efetuada e a janela está marcada como inválida o tratador de interrupção deverá salvar esta janela e daí continuar a execução da tarefa.

Da mesma forma, quando uma operação do tipo RESTORE for executada e a janela estiver marcada como inválida pelo *WIM* é necessário carregar esta da pilha de volta ao processador. Esse mecanismo permitiu uma troca de contexto mais ágil para processos que utilizam muitas chamadas de função.

4.5. Escalonamento

Atualmente o sistema MicroC/OS-II utiliza um escalonador preemptivo baseado em prioridades, como as prioridades são definidas pelo usuário sugere-se que este considere o deadline de cada tarefa para definir os níveis de prioridade, assim, quanto maior a taxa de execução de uma tarefa (quantas vezes por segundo esta deve ser executada) maior deve ser sua prioridade.

Um algoritmo sugerido para a seleção das prioridades é o RMS (Rate Monotonic Scheduling) o teorema do RMS afirma que se a inequação abaixo for atendida todos os

deadlines das tarefas serão atendidos.

$$\sum_i \frac{E_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

Onde n é o número total de tarefas no sistema e E_i é o tempo máximo de execução da task i e Ti corresponde ao período de execução da task i. Assim E_i/T_i corresponde à fração de tempo da CPU necessária para executar a tarefa i. O valor limite do segundo termo $n(2^{\frac{1}{n}} - 1)$ para um número infinitamente grande de tarefas é $ln(2) = 0.693$ o que significa que para atender a todas tarefas de tempo real a CPU deve estar a menos de 70% de uso(Labrosse, 2002).

O escalonador do MicroC/OS-II pode ser travado de modo que uma tarefa execute sem a possibilidade de ser interrompida por outra tarefa de prioridade maior, esse recurso deve ser usado com atenção de modo a evitar que se quebre a capacidade de escalonamento de tempo real do sistema. Com o scheduler travado uma tarefa irá executar até que ela própria destrave o escalonador. É importante notar que mesmo com o escalonador travado, uma interrupção pode retirar a tarefa de execução para permitir o tratamento daquela, porém neste caso mesmo que a interrupção deixe outra tarefa em estado de pronto e que esta tenha uma prioridade maior que a tarefa anterior, a tarefa anterior é que será alocada ao processador, já que o escalonador não será chamado neste caso.

4.6. Semáforo e Mutex

Uma forma comum de tratar a concorrência em sistemas multi-tarefa é o uso de semáforos estes permitem que tarefas diferentes compartilhem recursos, como dispositivos de E/S e áreas de memória compartilhada. O funcionamento de um semáforo foi definido por (Dijkstra, 1965) e é baseado em duas primitivas indivisíveis: uma para obter acesso, decrementando o semáforo, e outra liberar o acesso incrementando o semáforo.

Este tipo de semáforo é conhecido como semáforo contador e pode gerenciar acesso a um recurso onde se tem mais de uma instância, já para recursos únicos, como uma área de memória compartilhada utiliza-se semáforos binários que podem assumir apenas dois valores 0 ou 1 esses semáforos também são conhecidos como mutex pois permitem a exclusão mútua entre tarefas concorrentes.

O MicroC/OS-II implementa os dois tipos de controle de acesso concorrente, na implementação deste trabalho um outro tipo de semáforo foi implementado, onde vários

processadores podem sincronizar-se através de um nó central. A descrição detalhada deste mecanismo está na seção 5.2.

Na implementação do MicroC/OS-II quando uma tarefa é bloqueada por um semáforo, esta é incluída em uma fila de modo que quando o semáforo for liberado pela outra tarefa as tarefas que dependiam deste sinal possam ser acordadas na ordem em que fizeram o pedido.

Esse mecanismo foi especialmente necessário no controle de concorrência ao dispositivo de E/S implementado neste trabalho, para cada dispositivo de E/S existente na NoC há um semáforo que permite ou não o uso deste, quando um processo tenta o acesso ao dispositivo de E/S é necessário obter acesso ao semáforo correspondente a aquele dispositivo. O sistema operacional no momento do boot conhece todos os dispositivos de E/S e, portanto cria em memória a estrutura necessária para gerenciar esses semáforos.

O MicroC/OS-II é capaz de perceber problemas como a inversão de prioridades no uso de mutex e semáforos, o problema da inversão de prioridades ocorre quando uma tarefa de prioridade mais baixa obtém acesso a um recurso que é esperado por uma tarefa de mais alta prioridade. Para resolver esse problema o escalonador utiliza a herança de prioridade, uma tarefa dona de um recurso receberá um bônus em sua prioridade sempre que outra tarefa de prioridade maior solicite acesso a um recurso em posse da primeira.

4.7. Gerenciamento de Memória

O mecanismo de gerenciamento de memória do MicroC/OS-II é bem simplificado, já que este é direcionado aos sistemas embarcados a alocação de memória é estática, ou seja feita em blocos de tamanhos predefinidos. A memória de trabalho ou memória RAM é dividida em partições e cada partição é por sua vez dividida em blocos de mesmo tamanho, portanto se o programador precisa alocar N objetos de tamanho K é necessário criar uma partição que contenha N blocos com tamanho K.

Este modelo é funcional e bastante rápido em tempo de alocação, já que a localização dos blocos é pré-determinada. Porém este mecanismo sofre de vários problemas quando a quantidade ou o tamanho dos objetos a serem alocados são desconhecidos a priori (o que é mais comum), não é possível, em tempo de execução, redimensionar uma partição para colocar mais objetos caso esta esteja cheia, a solução seria criar outra partição com mais blocos do mesmo tamanho da primeira, o que começa a comprometer a velocidade do algoritmo, e exige-se mais memória para o controle desta nova partição.

Outro problema dessa abordagem é que ela é bastante suscetível à fragmentação interna, já que se um objeto menor que o tamanho do bloco é alocado a um bloco a área não usada neste não poderá ser alocada a outro objeto, ficando perdida.

Uma das necessidades urgentes no desenvolvimento de aplicações para a plataforma STORM era a capacidade de usar as funções *malloc()* e *free()* para gerenciar memória, permitindo que mais programas possam ser facilmente portados para STORM. O esquema de alocação de memória estático oferecido pelo MicroC/OS-II não permite o uso dessas funções de modo satisfatório.

Para atacar estes problemas uma nova biblioteca de gerência de memória foi implementada. Utilizando-se do conceito de listas ligadas de blocos livres é possível alocar e liberar memória em tamanhos quaisquer, com pequeno overhead para o controle. Esta implementação é descrita em detalhes na seção 5.3.

4.8. Portando o MicroC/OS-II

Como o kernel está quase inteiramente escrito em C e a plataforma STORM tem um compilador ANSI-C, grande parte do código pode ser reutilizada sem muitas alterações, exceções para questão de bibliotecas básicas que precisaram ser implementadas, e para o uso da biblioteca de ponto flutuante do GCC, já que na versão atual o processador SPARC da plataforma STORM não conta com uma unidade de ponto flutuante.

Há três arquivos originais do MicroC/OS-II que precisam ser refeitos para o processador em questão, os arquivos são: os_cpu.h os_cpu_c.c e os_cpu_a.s. Nestes três arquivos estão as funções do MicroC/OS-II que são dependentes da arquitetura do processador.

Além disto, uma série de bibliotecas precisou ser criada para que o sistema operacional pudesse ser executado no SPARC e na NoC. Essas bibliotecas estão distribuídas nos arquivos: *crt0.s, os_io_manager.c, os_mpi_manager.c, stormIO.c, stormLib.c, stormMem.c, stormUtil.c e stormString.c* além de seus respectivos arquivos de cabeçalho (arquivos .h).

5. Projeto do Sistema Operacional para STORM

O MicroC/OSII é um RTOS (*Real Time Operating System*) certificado para o uso em aplicações de *hard-real-time* (Labrosse, 2002). A maior parte do sistema está escrita em C, o que permite uma grande portabilidade. O escalonador deste sistema é bastante documentado e oferece a possibilidade de fácil adaptação a várias plataformas. A gerência de memória do MicroC/OSII é estática, sendo feita através de partições. Não há nenhuma biblioteca de controle de entrada e saída. A implementação realizada neste trabalho utilizou as estruturas de dados do MicroC/OSII e o escalonador *Round Robin*, que garantem os requisitos de tempo-real exigidos pelas aplicações *hard-real-time*.

Foi implementado um mecanismo de alocação dinâmica de memória, semelhante ao *malloc/free* comum na linguagem C. Esse mecanismo é baseado numa lista de blocos livres, utilizando o algoritmo do *first-fit* (Kernighan e Ritchie). Este algoritmo oferece bom desempenho no tempo de alocação.

Uma biblioteca completa de entrada e saída foi implementada seguindo os modelos das bibliotecas de tratamento de E/S da linguagem C. Esta biblioteca disponibiliza uma interface simples para o acesso aos dispositivos de E/S oferecendo as chamadas para tratamento de arquivos como: *fopen( ), fclose( ), fread( )* e *fwrite( )*.

A comunicação entre processos é feita através de chamadas do sistema operacional que implementam o padrão MPI. Dessa forma, uma aplicação pode tratar o MPSoC como um conjunto de núcleos processadores, facilitando o transporte de aplicações do mundo dos *clusters* para o mundo dos MPSoC.

5.1. Escalonamento

Um dos principais elementos de um sistema operacional é o escalonador. Este módulo é responsável por selecionar quais processos usarão o a CPU e quanto tempo será destinado para a execução de cada um. Sempre que um processo muda de estado, o escalonador precisa ser acionado para trazer um novo processo para execução. Assim, o algoritmo de seleção deve ser bastante otimizado, buscando velocidade na resposta.

O projeto de um algoritmo de escalonamento deve levar em consideração o tipo de aplicação que será executada na plataforma alvo. Mas em linhas gerais, (Tanenbaum, 2008) oferece questões importantes a serem consideradas pelos sistemas operacionais na implementação de seus escalonadores. São elas: (i) Justiça – considerados dois processos semelhantes ambos devem ter o mesmo tempo da CPU; (ii) Garantia da Política – um sistema pode definir políticas que são específicas para seu propósito, como por exemplo: garantir um tempo mínimo de CPU para todo processo; (iii) Balanceamento – é importante utilizar todos os recursos disponíveis no equipamento mantendo, sempre que possível, todos os recursos utilizados, evitando que um processador fique parado.

O conjunto de operações de salvamento de um processo em uma memória auxiliar e carregamento de outro processo é chamado de troca de contexto. Esse conjunto de operações costuma ser bastante oneroso, já que envolve salvar todo o estado atual em uma memória que, comumente, é uma ou mais ordens de grandeza mais lenta que o processador.

Em (Anderson, Calandrino e Devi, 2005) os autores consideram um sistema multiprocessado como o mostrado na Figura 5.1. Os autores observam que um cache miss na cache L2 é significativamente mais prejudicial ao desempenho que um cache miss da L1. Desse modo propõem evitar escalonar simultaneamente *threads* que causem alta concorrência pela cache L2. O escalonador proposto, então, utiliza grupos de *threads* chamadas *megatasks*, com o objetivo de evitar que tarefas que causem alta competição pela cache L2 sejam escalonadas ao mesmo tempo.

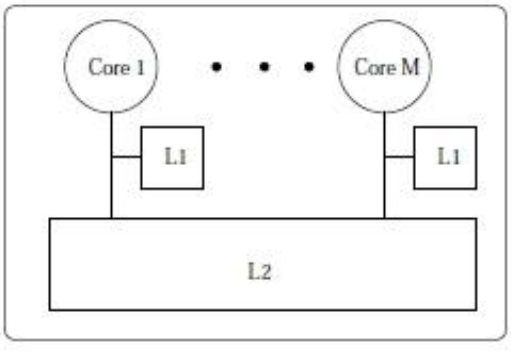

Figura 5.1 - Arquitetura de vários Cores

5.1.1. Escalonamento em RTOS

Uma das principais aplicações de MP-SoC está nos sistemas embarcados. A maioria destes sistemas executam tarefas críticas que demandam por respostas em tempo real. (Tanenbaum, 2008) apresenta uma definição para sistemas de tempo real como sendo sistemas em que o tempo é parâmetro chave, e divide-se estes sistemas em duas categorias: os *hard-real-time* e *soft-real-time*. O primeiro deve prover garantias que certas tarefas devem ser executadas em determinado tempo (alcançar o deadline). No segundo, pode ocorrer alguma variação no tempo de resposta esperado desde que esta não seja freqüente. Exemplos de sistemas *hard-real-time* são: controladores de equipamentos em fábricas e sistemas de automação. Exemplos de sistemas *soft-real-time* são: reprodutores de mídia digital.

Diferentemente do escalonador de um sistema convencional, os projetos de escalonadores de tempo real apresentam algumas novas questões que devem ser consideradas: alcançar deadlines – garantir que um processo seja executado em um intervalo de tempo definido; previsibilidade – garantir que o comportamento do sistema seja o esperado, não importando a carga atual.

Dois algoritmos clássicos de escalonamento de tempo real são: RMS (*Rate Monotonic Scheduler*) (Liu e Layland, 1973) e EDF (*Earliest Deadline First*). Esses algoritmos são amplamente utilizados em sistemas de tempo real na indústria. Como exemplo, o sistema de tempo real da Micrium chamado MicroC/OS-II, usado como base para este trabalho, utiliza um escalonador cuja prioridade das *tasks* é dada pelo programador à medida que as *tasks* são criadas. Entretanto, no guia de referência deste sistema (Labrosse, 2002) é sugerido que se utilize o algoritmo RMS para determinar as prioridades.

As estratégias de implementação de algoritmos de escalonamento em sistemas de tempo real variam juntamente com o número de sistemas operacionais de tempo real. Algumas propostas comuns em sistemas de tempo real, amplamente utilizados na indústria, são apresentadas a seguir.

O sistema QNX é um sistema de tempo real direcionado a plataformas x86 e ARM que utilizam um conjunto grande de algoritmos de escalonamento(QNX Software Systems GmbH & Co. KG., 2006). Com isso, o usuário tem a possibilidade de escolher o algoritmo a ser utilizado que mais se adéque a aplicação,

Os algoritmos são:

- FIFO – Primeiro a entrar primeiro a sair. Adequado a sistemas batch. Um processo fica de posse da CPU até que: (i) termine sua execução e libere espontaneamente; (ii) fique bloqueado; (iii) um processo de maior prioridade esteja pronto;
- RoudRobin – Fila circular com um quantum configurável. Cada processo executará por um tempo fixo (quantum) ou até que um processo de maior prioridade chegue;
- Sporadic – Define 2 níveis de prioridade apenas: alto e baixo. Quando uma *thread* precisa ser executada é oferecido um saldo; a medida que a *thread* usa tempo de CPU, seu saldo é diminuído; quando a *thread* zera o saldo, cai de prioridade; após um tempo T configurável a *thread* recupera o saldo e a prioridade;
- Adaptative Partitioning Scheduler – Utiliza o conceito de partição (QNX Software Systems, 2009) para alocar a CPU aos processos. Assim, é possível garantir uma porcentagem da CPU a determinada aplicação, evitando que uma única aplicação monopolize o uso de CPU, e garantindo que aplicações críticas sempre tenham o mínimo necessário.

O eCos é um sistema operacional *open-source* que trabalha com dois escalonadores diferentes. No primeiro, chamado *Multilevel Queue,*, um conjunto de *threads* está alocada a uma fila de prioridades. Dentro de uma fila de *threads* com a mesma prioridade, o algoritmo utilizado é um *Round-Robin*. O eCos suporta arquiteturas SMP apenas com o escalonamento do tipo Multilevel Queue(Massa, 2002). O segundo algoritmo de escalonamento é chamado Bitmap. Neste algoritmo pode haver apenas uma *thread* por prioridade e a *thread* de maior prioridade será executada sempre até o final ou até ficar bloqueada.

O escalonamento do tipo *Multilevel Queu*e também é usado pelo NetBSD(NetBSD Foundation, 2008), um sistema *open-source* direcionado a sistemas embarcados com fortes restrições de recursos. Porém, um processo tem sua prioridade recalculada sempre que ocupa muito tempo de CPU. O tempo necessário para diminuir a prioridade de um processo é configurado no sistema, em tempo de compilação.

O uso de escalonadores projetados para arquiteturas *single-core* pode ser útil, considerando-se um escalonamento hierárquico como proposto em (Srinivasan e Anderson, 2002) e (Liu, Klues, *et al.*, 2009). Neste modelo há um escalonador que se preocupa com o sistema como um todo, e instâncias do escalonador mais simples que se preocupam com grupos de *threads* relacionadas, seja por compartilharem recursos ou páginas de memória.

Um algoritmo proposto por (Srinivasan e Anderson, 2002) oferece escalonamento de tarefas em tempo real para sistemas de multiprocessadores. Este algoritmo, chamado PD2, é uma implementação do algoritmo PFAIR. A metodologia adotada pelo algoritmo PFAIR considera que uma *subtask* com um *deadline* menor tem sempre prioridade maior que uma *subtask* com deadline maior. O algoritmo PD2 inclui regras para o desempate para o caso em que o deadline seja igual. Outra implementação do algoritmo PFAIR é EPDF (*Earliest Pseudo Deadline First*)(Anderson e Srinivasan, 2004). Nesta implementação, a regra para desempate é uma escolha arbitrária, o que simplifica o algoritmo, porém torna-o um algoritmo sub-ótimo.

Na implementação atual o algoritmo escolhido foi o EDF por ser capaz de tratar tarefas aperiódicas, e sua implementação ser possível usando o algoritmo *Round Robin*. Neste caso, a prioridade dos processos do sistema operacional é dada no momento da criação do processo, baseando-se no *deadline* para a execução.

Uma questão importante na implementação de processos em STORM é que, como seu processador é um SPARC, o modelo de registradores utilizando janelas adiciona uma complexidade extra ao salvar e carregar o contexto de um processo. Enquanto o modelo de janelas oferece ótimo desempenho para aplicações que tem muitas chamadas de funções, como é comum em aplicações escritas em C, este mesmo modelo oferece um grande *overhead* para o salvamento do contexto, já que várias janelas precisam ser salvas e carregadas a cada operação.

5.2. Gerenciamento de Entrada e Saída

Outra tarefa importante do sistema operacional é controlar o acesso concorrente aos dispositivos de entrada e saída. Em (Tanenbaum, 2008) são exibidas três maneiras de tratar entrada e saída do ponto de vista do software: E/S programada, E/S dirigida por interrupções e E/S dirigida por DMA (*Direct Memory Acess*).

Na E/S programada, a CPU executa todo o trabalho e fica dedicada à tarefa de gerenciar o dispositivo, checar seu status e enviar os dados na velocidade adequada ao dispositivo em questão. Isto pode desperdiçar muitos ciclos de CPU se o dispositivo de E/S for muito mais lento que a CPU (o que é bastante comum).

Na E/S dirigida por interrupção, a CPU programa o dispositivo para executar determinada tarefa e está livre para executar outros processos. Quando o dispositivo termina a execução do que foi programado, este avisa a CPU através de uma interrupção. Ocorre então uma troca de contexto para executar a rotina de tratamento de interrupção correspondente. Esta técnica libera o processador para executar outras tarefas enquanto o dispositivo efetua o que foi pedido, porém se o número de interrupções for grande, isto pode comprometer a efetividade da CPU com o tempo entre trocas de contexto.

A última técnica exposta é E/S dirigida por DMA. Nessa técnica a CPU envia uma série de comandos a um dispositivo de DMA que se encarrega de programar o dispositivo de entrada e saída. É também responsabilidade do DMA armazenar o resultado em um endereço pré-estabelecido, interrompendo a CPU apenas quando todo o trabalho foi executado. Esta técnica libera a CPU para executar outros processos. Há de se ressaltar que o módulo de DMA deve ser rápido o suficiente para acompanhar a velocidade do dispositivo de E/S ou este será sub-utilizado.

O trabalho de (Medardoni, Ruggiero, *et al.*, 2007) sugere um conjunto de diretivas para o desenvolvimento de sistema de entrada e saída para arquiteturas MP-SoC que utilizam barramentos, especificamente arquiteturas que utilizarem hierarquias de barramentos.

Poucos trabalhos, no entanto, tem se esforçado para oferecer um modelo adequado de gerenciamento de I/O em plataformas MP-SoC baseadas em NoC. Neste trabalho, uma implementação de bibliotecas de Entrada e Saída completa foi feita, oferecendo ao programador chamadas de alto nível para tratar a entrada e saída do MPSoC.

Para a implementação do gerenciamento de entrada e saída da plataforma STORM, foi escolhido o modelo baseado em interrupções. Este modelo permite um bom aproveitamento da CPU. já que a CPU fica liberada para executar outros processos enquanto aguarda o pedido de E/S acontecer. O modelo com suporte a DMA não foi escolhido pois sua implementação seria mais complexa demandando muito tempo em uma tarefa que não era o foco principal do trabalho.

A implementação aqui apresentada, é capaz de oferecer acesso concorrente a dispositivos de E/S localizados em qualquer ponto da NoC. O dispositivo deve ser conectado à NoC utilizando o IONode descrito na seção 2.5 Sistema de Entrada e Saída. A camada de software necessária para gerenciar essas operações está distribuída entre os nós processadores. Existe um nó mestre capaz de gerenciar a concorrência entre os vários processos espalhados na NoC.

O nó mestre oferece um mecanismo de exclusão mútua distribuído para acesso aos dispositivos de entrada e saída. Este nó pode também estar executando aplicações como os outros nós. Um módulo especial de entrada e saída do sistema operacional está disponível apenas no nó mestre. Este nó é definido em tempo de compilação e sua escolha é dependente do projeto do MPSoC. Esta escolha deve buscar diminuir o tráfego, evitando o gargalo inerente ao nó mestre.

Seguindo o modelo POSIX (*Portable Operating System Interface*)(The Open Group, 2008), o sistema operacional implementado oferece ao usuário a abstração de um arquivo para cada dispositivo de E/S disponível no sistema. Toda operação de E/S é iniciada fazendo um pedido de abertura de arquivo ao sistema operacional. O processo usará a função *fopen( )* da biblioteca do sistema operacional, que, por sua vez, irá identificar que essa é uma operação de E/S bloqueando o processo até que a operação esteja concluída.

Quando um pacote é recebido pelo CoMa, este gera uma interrupção que informa ao sistema operacional que há dados a serem lidos para a memória local do processador em seu buffer. Esse mecanismo permite que o sistema operacional possa enviar dados pela NoC e escalonar processos diferentes enquanto aguarda a resposta chegar.

O mecanismo de entrada e saída pode ser dividido em 3 fases: (i) Abertura do arquivo; (ii) Operações de E/S; (iii) Fechamento do arquivo. Em cada uma dessas fases o trabalho é divido entre o nó que fez o pedido, o nó mestre e o nó de entrada e saída IONode.

A Figura 5.2 apresenta o processo de abertura de um arquivo. Na primeira fase, Figura 5.2, o processo faz o pedido ao sistema operacional local. Este bloqueia o processo e cria um pacote de requisição de abertura para o arquivo em questão. Este pacote é composto por: um identificador do arquivo a ser aberto, o modo de acesso (somente leitura, leitura e escrita), e o endereço de NoC do nó de origem do pedido. O sistema operacional local inclui então este processo na fila de processos bloqueados esperando o acesso do arquivo e envia o pacote. A partir deste momento outro processo pode ser escalonado para executar no processador.

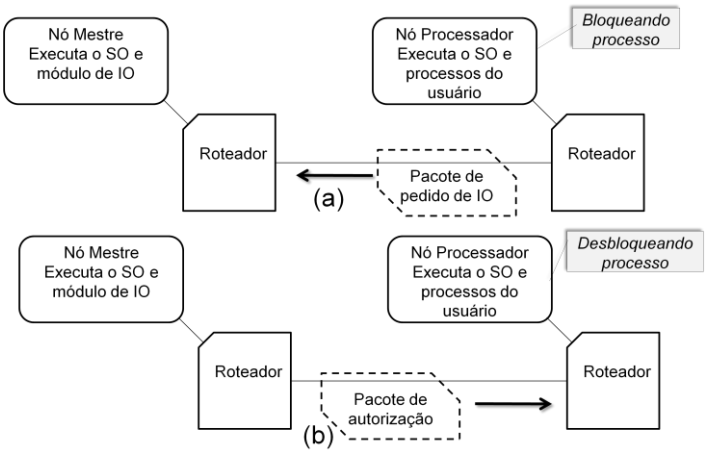

Figura 5.2 - Abertura de um arquivo (a) Pedido de abertura (b) Autorização

Quando o pacote chega ao nó mestre é necessário verificar se o arquivo em questão não já está sendo usado por outro processo do MPSoC. Para isso, o nó mestre mantém uma tabela com o estado de todos os dispositivos de E/S do MPSoC. Após consultar esta tabela, o nó mestre pode colocar o pedido em uma fila de espera ou pode liberar o acesso imediatamente, caso o dispositivo esteja livre.

Uma vez que tenha recebido o pacote de resposta do nó mestre, o sistema operacional local desbloqueia o processo, que entrará agora na fase de realizar as operações de entrada de saída (Figura 5.3). Nesta fase, o sistema operacional comunica-se diretamente com o IONode, desonerando o nó mestre. O processo do usuário irá então usar as funções *fread( )* e *fwrite( )* para ler e escrever dados no dispositivo de E/S. Da mesma maneira que o *fopen( )*, o sistema operacional bloqueará o processo e criará o pacote de E/S que será enviado diretamente ao IONode.

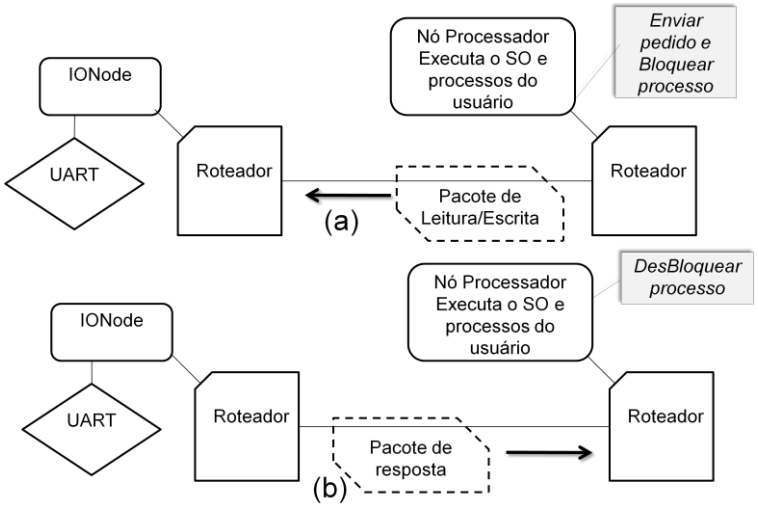

Figura 5.3 – Operaçõe do IONode (a) pedido de leitura/escrita (b) resposta do IONode

Após todas as operações terem sido realizadas, o processo do usuário deve chamar a função *fclose( )* para liberar o arquivo. Novamente, o sistema operacional criará o pacote a ser enviado, agora para o nó mestre, informando da liberação do arquivo. A Tabela 5.1 resume as operações de entrada e saída suportadas pelo módulo.

Tabela 5.1 - Operações de E/S

Operação	**Descrição**	**Origem**	**Destino**
fopen( )	Solicita a abertura de um arquivo ao nó mestre	Nó processador	Nó-Mestre de E/S
fread( )	Faz uma leitura de um dado de um dispositivo de E/S	Nó processador	IONode
fwrite( )	Faz uma escrita de um dado em um dispositivo de E/S	Nó processador	IONode
fclose( )	Libera um arquivo que estava sendo usado por um process	Nó processador	Nó-Mestre de E/S

Esse mecanismo garante o acesso concorrente e consistente de um ou mais dispositivos de E/S existentes no MPSoC.

5.3.Gerenciamento de memória

Em um sistema operacional multitarefa, o gerenciamento da memória disponível é tarefa importante para garantir o acesso seguro e otimizado entre os vários processos concorrentes. Em sistemas com múltiplos módulos de memória, duas estratégias podem ser adotadas para o endereçamento da memória. Pode-se trabalhar com memória compartilhada, onde todos os blocos de memória estão mapeados no espaço de endereçamento de todos os elementos processadores do sistema; ou trabalhar com a memória distribuída, onde cada processador tem alocado para si uma parte da memória em seu espaço de endereçamento específico.

O sistema operacional proposto por este trabalho baseia-se na versão de memória distribuída da plataforma STORM. Nesta versão, a cada nó da rede está conectado um processador SPARCV8 junto com um módulo de memória. O processador endereça a memória local totalmente e o acesso é feito através de um componente DAMa (*Data Access Manager*). Este componente conhece todos os endereços de memória, sendo capaz de decidir para onde e de onde obter o dado endereçado pelo processador.

O sistema operacional MicroC/OSII gerencia memória estaticamente, ou seja, a alocação é feita em blocos de tamanho fixo criados dentro de uma partição, como mostra a Figura 5.4. Um usuário poderá, então, obter espaços de memória apenas que sejam múltiplos do tamanho do bloco definido. Ao definir a aplicação que irá executar, o usuário cria N partições na memória, cada uma com um tamanho de bloco diferente. É responsabilidade do usuário obter o bloco da partição adequada buscando evitar a fragmentação da memória.

É possível criar partições com blocos de memória de tamanho diferente. Assim um usuário poderá alocar blocos de memória de tamanho diferente de acordo com a sua necessidade, mas estará restrito aos tamanhos definidos nas partições.

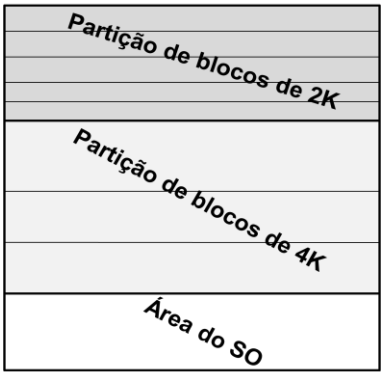

Figura 5.4 - Memória em partições

Na implementação do sistema operacional discutida neste trabalho, um mecanismo de alocação de memória local foi implementado buscando aumentar a performance do sistema. Este mecanismo é baseado em listas ligadas mantidas pelo sistema operacional e permite que um usuário obtenha um bloco de memória de qualquer tamanho, limitado apenas ao tamanho da memória disponível para uso.

Este controle é feito utilizando-se uma lista ligada de blocos livres. Inicialmente a lista aponta para um grande bloco de memória livre. Quando um pedido de alocação é feito, o algoritmo irá buscar na lista um bloco livre que caiba o tamanho requisitado pelo usuário. Será tomado, então, o primeiro bloco que possa abrigar o pedido feito. Este método de escolha é conhecido como *first-fit*. Uma vez encontrado, o bloco é dividido em duas partes: uma alocada para o usuário e outra formando um novo bloco não alocado.

O método *first-fit* é um algoritmo rápido já que faz a menor busca possível. Há outros algoritmos conhecidos da literatura para a busca de um bloco livre de memória para alocação. Estes algoritmos aumentam o tempo de busca por um bloco, tentando diminuir a fragmentação gerada por situações em que o bloco selecionado não é do tamanho exato pedido pelo usuário.

O algoritmo *best-fit* tenta buscar blocos em que sobre o mínimo de espaço possível. Para isso, é necessário percorrer toda a lista de blocos livres, o que pode ser muito custoso. Já o algoritmo *worst-fit* busca achar o maior bloco possível de modo que o espaço restante em um bloco possa ser mais facilmente usado posteriormente. Estudos mostram que não é vantajoso o uso desses algoritmos, já que a fragmentação continua a ocorrer e o tempo de alocação cresce (Tanenbaum, 2008). A implementação discutida neste trabalho utiliza, então, o algoritmo *first-fit* para buscar o bloco de memória a ser alocado.

A memória de um nó é dividida pelo sistema operacional no momento do boot em duas áreas: uma área reservada para o sistema operacional; e outra alocável pelo usuário. Dentro da área alocável pelo usuário, os pedidos de *malloc* e *free* são atendidos pelo sistema operacional. Além dos próprios dados do usuário, esta área também guarda a estrutura de dados necessária para o controle da própria região de memória.

Cada bloco alocado consome uma área de até 2 palavras para armazenar o ponteiro para a próxima área livre e o tamanho do bloco alocado. Assim, quando um bloco é alocado, é sempre alocado o tamanho pedido mais 2 palavras de controle do sistema operacional.

Em linhas gerais a estrutura de dados que permite este gerenciamento é exibida na Figura 5.5. O cabeçalho aponta para a próxima área livre da memória e indica quanto, a partir daquele endereço, está alocado a um processo.

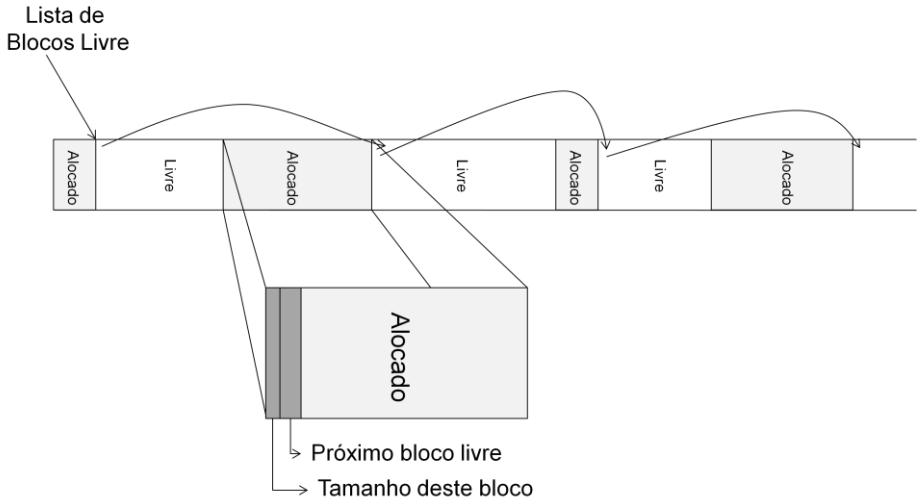

Figura 5.5 - Alocação de memória dinâmica

A liberação da memória pode ser feita, então, facilmente, bastando consultar o cabeçalho que indica o tamanho alocado e mover o ponteiro da lista de blocos livres para o próximo endereço livre, somando o endereço que acabou de ser liberado.

5.4.Comunicação Inter-processos

O mecanismo de comunicação Inter-processo utilizado é baseado no padrão MPI. Esta biblioteca foi implementada para a plataforma STORM em uma versão sem o sistema operacional por (Oliveira, 2009). Neste trabalho a biblioteca MPI foi integrada ao *kernel* do sistema operacional e é a base da comunicação inter-processos. Na prática, todas as chamadas do MPI foram transferidas para dentro do sistema operacional e uma interface para estas chamadas é oferecida ao programador.

Sempre que o programador usa uma das chamadas do MPI, uma chamada ao sistema operacional é feita. Esta chamada é transparente ao programador. O sistema operacional trata, então, de decodificar o pedido do usuário e traduzi-lo para um nível mais baixo onde, o próprio *kernel* se encarrega de criar um pacote no formato adequado para o transporte pela NoC.

Esta integração com o sistema operacional oferece ao usuário uma maneira fácil de lidar com a comunicação inter-processos dentro do ambiente distribuído do MPSoC.

O modelo MPI trata processos em execução através de um número único no MPI. Este número é chamado *rank* e quando um processo deseja enviar uma mensagem para outro é necessário conhecer o número de *rank* do destino. Este número é atribuído na inicialização do MPI. No modelo atual, isto é feito no momento do *boot* do sistema operacional, permitindo que o usuário crie tantos processos participantes do MPI quantos forem necessários. Este número é limitado apenas pelo número máximo de processos suportados pelo sistema operacional.

O processo de inicialização do MPI consiste em: cada nó enviar o PID (*process ID*) dos processos que desejam participar do MPI ao nó mestre do MPI. Este, após receber os PIDs de todos os nós, atribui um *rank* único a cada processo e envia a todos os nós a tabela contendo o *rank* de cada processo. Com esta tabela o sistema operacional poderá traduzir os pedidos de envio e recebimento de mensagem de entre *ranks* para o envio e recebimento de mensagens na NoC.

O sistema operacional é responsável por receber o pedido do processo, criar um pacote de NoC que contenha o cabeçalho que identifica o MPI, bloquear o processo em questão, traduzir o rank para um endereço de NoC e enviar o pacote pela NoC para o nó de destino. No nó de destino, quando um pacote é recebido, uma interrupção é gerada e o sistema operacional verifica se o pacote é um de MPI. Caso seja, o pacote é lido e armazenado em um buffer interno do sistema operacional, de acordo com o processo a que se destina a mensagem.

A especificação MPI define que deve haver um nó central que também é responsável por algumas tarefas de gerência do MPI. Este nó pode ser definido em tempo de compilação do sistema operacional e pode estar, ou não, no mesmo nó mestre de E/S. Esta característica torna o sistema operacional bastante flexível já que os módulos podem ser divididos e alocados a nós específicos com maior ou menor tamanho de memória, localização na NoC entre outras características.

As operações atualmente suportadas pelo MPI são descritas na Tabela 5.2.

Tabela 5.2 - Operações suportadas pelo MPI

Operação	**Descrição**
MPI_Comm_size	Retorna o número de processadores em um grupo
MPI_Comm_rank	Retorna o rank de um processador em um grupo
MPI_Init	Inicializa o ambiente MPI
MPI_Finalize	Finaliza o ambiente MPI

MPI_Recv	Recebe uma mensagem de outro nó da rede
MPI_Send	Envia uma mensagem a outro nó da rede
MPI_Bcast	Envia uma mensagem de um processador raiz para todos os outros processadores em um grupo
MPI_Reduce	Combina os dados fornecidos por cada processador de um grupo em um dado único
MPI_Pack	Empacota dados em uma região contígua de memória
MPI_Unpack	Desempacota dados contidos em uma região de memória contigua de acordo com o tipo de dado fornecido
MPI_Pack_size	Retorna a quantidade de bytes necessários para empacotar um determinado dado
MPI_Op_create	Cria uma função de combinação definida pelo usuário

O padrão MPI define um cabeçalho a ser enviado em cada pacote. Além deste cabeçalho, uma palavra do sistema operacional é adicionada ao início do pacote, identificando-o como um pacote MPI. Isso permite que o gerenciador de interrupções identifique a origem e o destino do pacote realizando o tratamento necessário para este.

5.5. Gerenciamento de Interrupções

Para garantir a capacidade do sistema operacional de gerenciar melhor o uso do processador, foi implementada uma interrupção no processador SPARC que indica quando há dados a serem lidos vindos da NoC. Desta maneira o processador fica livre para executar outros processos, enquanto aguarda a chegada de um dado vindo da NoC.

Este mecanismo de interrupções é fundamental para o funcionamento do escalonador, pois permite que o sistema operacional escalone processos diferentes enquanto aguarda que um dado chegue da NoC. Uma vez que um processo tenha sido bloqueado por estar efetuando uma operação de E/S, por exemplo, outros processos podem rodar. Quando a operação de E/S for completada, o mecanismo de interrupção acordará o sistema operacional que, por sua vez, poderá transferir o processo da fila de bloqueados para a fila de prontos.

A versão anterior da STORM utilizava um mecanismo de espera ocupada para efetuar leituras e escritas na NoC. Essencialmente, uma vez que uma leitura era requisitada por um processo, este ficava ocupando a CPU testando um registrador do CoMa em *loop* até que algum dado estivesse pronto para ser lido. A mudança do modelo espera-ocupada, existente na versão anterior da STORM, para o modelo dirigido por interrupções possibilita, portanto, uma melhor utilização do tempo de CPU compartilhada entre os vários processos em execução.

O CoMa é a interface de hardware com a NoC e é responsável por identificar o pacote. Esta identificação é feita através da leitura do cabeçalho de NoC, que informa, entre outras coisas, o tamanho total do pacote. Este primeiro cabeçalho tratado ainda no CoMa é retirado e o *payload* do pacote é armazenado no buffer do CoMa. Neste momento, a interrupção é gerada, avisando ao sistema operacional que dados vindos da NoC estão disponíveis para a leitura do buffer.

A interrupção em questão é gerada pelo CoMa (*Communication Manager*) que é a interface entre o núcleo do processador e a NoC. Quando a rotina de tratamento desta interrupção é chamada, o sistema operacional lerá o pacote no buffer do CoMa. A primeira palavra é o cabeçalho do sistema operacional e identifica que tipo de pacote está chegando: um pacote de MPI, um pacote de E/S, um pacote de controle do sistema operacional, etc. Essa primeira palavra é adicionada no momento do envio do pacote.

Uma vez que tenha sido lido o cabeçalho do sistema operacional do início do pacote, a decisão de como tratar o pacote é passada ao módulo específico, que trata aquela categoria de pacote. Este módulo irá passar o resto dos dados para uma área interna de memória de onde serão copiados para o espaço de memória do usuário no momento que o processo for restaurado.

Para os pacotes de E/S, o módulo de gerência de E/S irá analisar o tipo do pacote. Há oito tipos de pacote de E/S. A Tabela 5.3 exibe cada um desses tipos.

Tabela 5.3 - Tipos de pacote de E/S

Cabeçalhos de pedidos de E/S	
IO_REQUEST_ACCESS	Identificador de um pedido de acesso ao no mestre
IO_WRITING	Identificador de um pedido de escrita ao IONode
IO_READING	Identificador de um pedido de leitura ao IONode
IO_FREE_ACCESS	Identificador de um aviso de fechamento de um arquivo

Cabeçalhos de resposta de E/S	
IO_RESP_REQUEST_ACCESS	Resposta do nó mestre a um pedido de acesso a arquivo
IO_RESP_WRITING	Resposta do IONode a um pedido de escrita
IO_RESP_READING	Resposta do IONode a um pedido de leitura
IO_RESP_FREE_ACCESS	Resposta do nó mestre a um pedido fechamento de arquivo

No momento que se descobre o tipo do pacote, algumas ações devem ser tomadas pelo sistema operacional. Caso este seja a resposta de um pedido de escrita bem sucedido (IO_RESP_WRITING), é necessário desbloquear o processo que solicitou a escrita. Caso o pacote seja a resposta de um pedido de leitura (IO_RESP_READING), o próprio módulo de gerência de E/S irá copiar os dados para o espaço do usuário. Após feita a cópia, o processo poderá ser retirado da fila de bloqueados, sendo movido para a fila de prontos. Em qualquer dessas situações, o escalonador será chamado a verificar qual o processo deve ser escalonado depois da interrupção ser tratada.

Os pacotes IO_REQUEST_ACCESS e IO_FREE_ACCESS são tratados apenas pelo nó mestre E/S. Estes pacotes são enviados pelos nós processadores para obter acesso exclusivo a um dispositivo de E/S. O nó mestre mantém uma fila de pedidos de acesso a cada dispositivo do MPSoC. Quando um pedido para acesso é feito (chamando *fopen( )*, por exemplo), o sistema operacional local bloqueia o processo até receber um pacote de liberação de acesso (IO_RESP_REQUEST_ACCESS) vindo do nó mestre.

De maneira análoga o processo é bloqueado ao executar um *fclose( )* até que tenha recebido o aviso de que o arquivo foi fechado com sucesso. Esses pacotes podem, então, ser usados como um mecanismo de exclusão mútua distribuída.

Uma maneira de se obter tratamento de acesso mutuamente exclusivo a recursos por processos é baseado em um arquivo de *lock*. Este mecanismo funciona da seguinte maneira, um nome de arquivo é definido entre os processos que desejam concorrer por um recurso. Sempre que um dos processos concorrentes quiser utilizar do recurso ou quiser entrar em sua região crítica, para garantir que nenhum outro processo dentro do ambiente distribuído possa ter acesso, o processo irá tentar obter acesso exclusivo ao arquivo de *lock*.

Outro tipo de pacote existente é o pacote que contem operações do MPI. Atualmente, do cabeçalho de 1 palavra, adicionado pelo sistema operacional, apenas os primeiros 4 bits são usados. Estes bits contem o valor 0001, que indica que este é um pacote de MPI. Os outros bits podem ser usados em implementações futuras para armazenar informações como: o grupo MPI, *flags*, entre outras informações pertinentes aos pacotes do tipo MPI.

Para esses pacotes, o módulo de MPI irá copiar os dados para uma buffer reservado para estas mensagens. Uma vez copiado, os processos continuam em execução. Caso algum processo esteja bloqueado esperando esta mensagem (com um *mpi_receive( )*, por exemplo), este será desbloqueado e poderá executar.

O módulo de MPI do *kernel* irá, então, ler o cabeçalho do pacote e, ao identificar o *rank* do processo, poderá copiar para um espaço de armazenamento do sistema operacional que serve como buffer temporário para as mensagens.

Quando um processo executa um *mpi_receive( )*, o sistema operacional busca neste buffer por possíveis mensagens armazenadas. Se não há mensagens, o processo é colocado em estado bloqueado. Da mesma forma, quando um pacote chega, o *kernel* deverá enviar um sinal aos processos que estejam bloqueados esperando por este pacote. O processo que estiver na fila de espera poderá sair do estado bloqueado e passar ao estado pronto. Neste momento, o dado pode ser copiado para o espaço do usuário, onde estará disponível para o processo.

6. Testes e Avaliação de desempenho

6.1.A simulação de reservatórios

O Projeto Simulação de Reservatórios em Ambiente de MP-SoC desenvolvido no Programa de Pós-Graduação em Sistemas e Computação da Universidade Federal do Rio Grande do Norte visa avaliar a complexidade da simulação de reservatórios sob a nova ótica do desenvolvimento de sistemas computacionais integrados em um único chip, dotado de dezenas a centenas de unidades funcionais dedicadas ou de propósito geral. Neste contexto um trabalho anterior (Oliveira, 2009) já avaliou e desenvolveu parte do simulador de reservatórios baseado no modelo *Black Oil*.

Esse modelo não é o mais completo existente, mas sua complexidade é suficiente para obter resultados satisfatórios em um espaço de tempo aceitável e. Por isso, é um dos modelos mais utilizados em simuladores comerciais. Outra característica importante desse modelo é seu bom desempenho na simulação de reservatórios que possuem óleos pesados e com baixa volatilidade, tipo mais comum de reservatório encontrado no Brasil.

6.2.Testes com a simulação de reservatórios

A simulação de reservatórios permite que, dadas as condições iniciais de um poço, possa-se realizar previsões de extração para aquele poço. A simulação de reservatórios pode ser divida então em três fases essenciais.

A primeira fase é a normalização dos dados. Inicialmente os dados colhidos em campo e em laboratório estão em escalas diferentes e precisarão passar por uma normalização para que seja utilizados no simulador.

A segunda fase é a discretização, as equações diferenciais que representam um sistema óleo-gás-água de um reservatório são muito complexas para ser resolvidas de forma analítica em tempo hábil. É necessário utilizar técnicas de discretização numérica para obter resultados aproximados.

O processo de discretização de equações diferenciais leva a um sistema de equações lineares. O que leva à última fase do simulador de reservatórios, que é a resolução dos sistemas lineares que descrevem o reservatório.

Entre as três fases citadas, a que consome mais tempo é a resolução dos sistemas lineares, como apontado em (SILVA, 2003). Chegando a ocupar 80% do tempo total do simulador esta fase é o foco dos testes realizados neste trabalho.

Há vários métodos de resolução de sistemas lineares, como apresentado em (Oliveira, 2009), a escolha feita por este trabalho, do método SOR (*Successive Over-Relaxation*), deve-se a este ser um método capaz de resolver as equações lineares utilizadas em simuladores e este método já ter sido testado em trabalhos anteriores, permitindo uma comparação com as características adicionadas por este trabalho à plataforma STORM.

Os vários métodos para solução de sistemas de equações lineares são divididos, basicamente, em dois grupos: métodos diretos e métodos iterativos. Os métodos diretos utilizam algoritmos de fatoração para encontrar a solução exata do sistema em um número fixo de passos. Já os métodos iterativos tentam aproximar, a cada iteração, uma solução inicial (pouco precisa) à solução real do sistema. As iterações são realizadas até que a solução aproximada seja suficientemente precisa.

Para a resolução de sistemas lineares de grande porte é sugerido o uso dos métodos iterativos já que estes consomem menos memória e podem ser mais facilmente paralelizados(FANCHI, HARPOLE e BUJNOWSKI, 1982.). Um sistemas de equações lineares pode ser representado como segue:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\ \vdots & \quad \vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

Podendo ser também representado como $A\bar{x} = \bar{b}$ onde A é a matriz dos coeficientes, x é um vetor coluna das variáveis e b um vetor coluna dos elementos independentes. Como mostrado a seguir:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \quad \bar{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \bar{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

No método iterativo de Jacobi, uma solução aproximada para o componente x_i^{k+1} na iteração $k+1$ é encontrada em função dos demais elementos da iteração anterior, k, conforme a Equação abaixo. Dessa forma, o vetor $\bar{x}^k$ não pode ser sobrescrito até o final da iteração $k+1$.

$$x_i^{k+1} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j=1\ j\neq i}^{n} a_{i,j}x_j^{(k)}\right) \qquad i = 1, 2, \ldots, n$$

a) Método de Gauss-Seidel

O método de Gauss-Seidel é obtido a partir do método de Jacobi, diferenciando apenas que os componentes do vetor solução que já foram calculados na iteração $k+1$ são imediatamente utilizados no cálculo dos demais componentes desta mesma iteração, resultando na seguinte expressão:

$$x_i^{k+1} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j<i}^{n} a_{i,j}x_j^{(k+1)} - \sum_{j>i}^{n} a_{i,j}x_j^{(k)}\right) \qquad i = 1, 2, \ldots, n$$

No método de Jacobi, observa-se que os elementos x_i^{k+1} podem ser calculados independentes da ordem em que cada elemento é calculado. No método de Gauss-Seidel a ordem em que o elemento x_i^{k+1} é calculado é importante, pois seu valor depende dos elementos já calculados nesta mesma iteração. Normalmente o número de iterações no método Gauss-Seidel é menor que no método de Jacobi.

b) Método SOR (*Successive Over-Relaxation*)

O método SOR é obtido a partir de uma sobre-relaxação do método de Gauss-Seidel. Seu funcionamento consiste na obtenção de uma média ponderada entre a iteração de Gauss-Seidel e a iteração anterior para cada componente, resultado em:

$$x_i^{k+1} = \omega x_i^{k+1} + (1-\omega)x_i^{k}$$

Onde x_i^{k+1} é o componente da iteração de Gauss-Seidel, obtido pela equação acima, e ω é o fator de sobre-relaxação. O uso de fator ω adequado acelera a convergência do método, reduzindo o número de iterações necessárias para resolver o sistema. Quando ω é igual a 1, o método SOR é igual ao método de Gauss-Seidel.

Para os testes a seguir foram utilizadas cinco instâncias diferentes da plataforma como exibidas na Figura 6.1, e uma comparação com o modelo sem sistema operacional foi realizada.

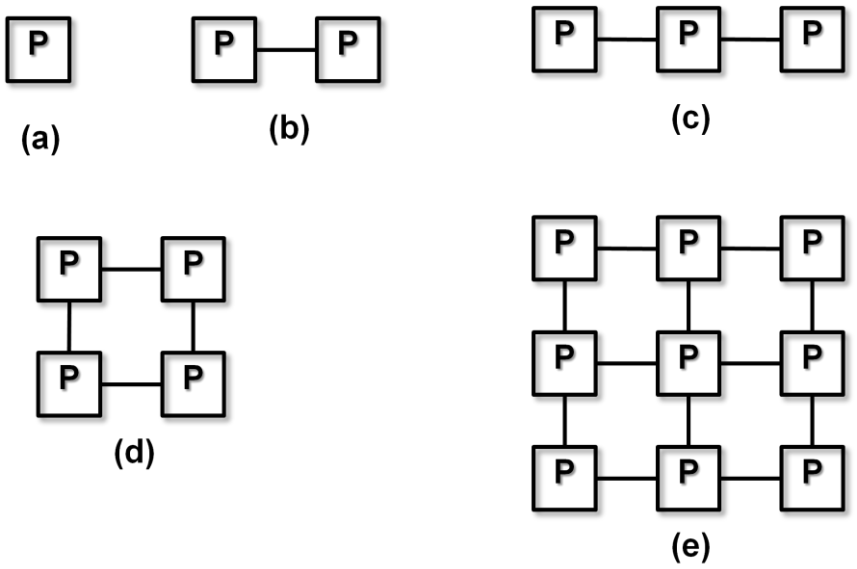

Figura 6.1 - Instâncias de teste da STORM

O gráfico da Figura 6.2 exibe a quantidade ciclos executados pela simulação de reservatórios para as instâncias da plataforma exibidas na Figura 6.1. Há de se notar um aumento no número de ciclos executados. Devido essencialmente ao overhead adicionado pelo sistema operacional.

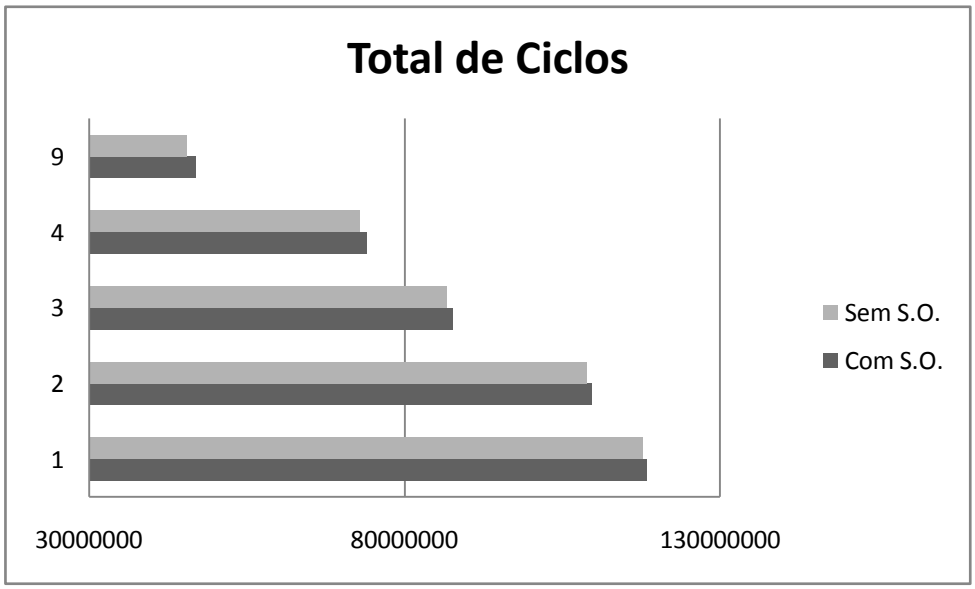

Figura 6.2 - Total de ciclos com e sem o Sistema Operacional

Outra medida importante na avaliação do sistema operacional é a carga adicional que este injeta na NoC devido aos dados de controle que trafegam entres os nós a Figura 6.3 exibe uma comparação entre a carga injetada pelo algoritmo SOR com e sem o sistema operacional.

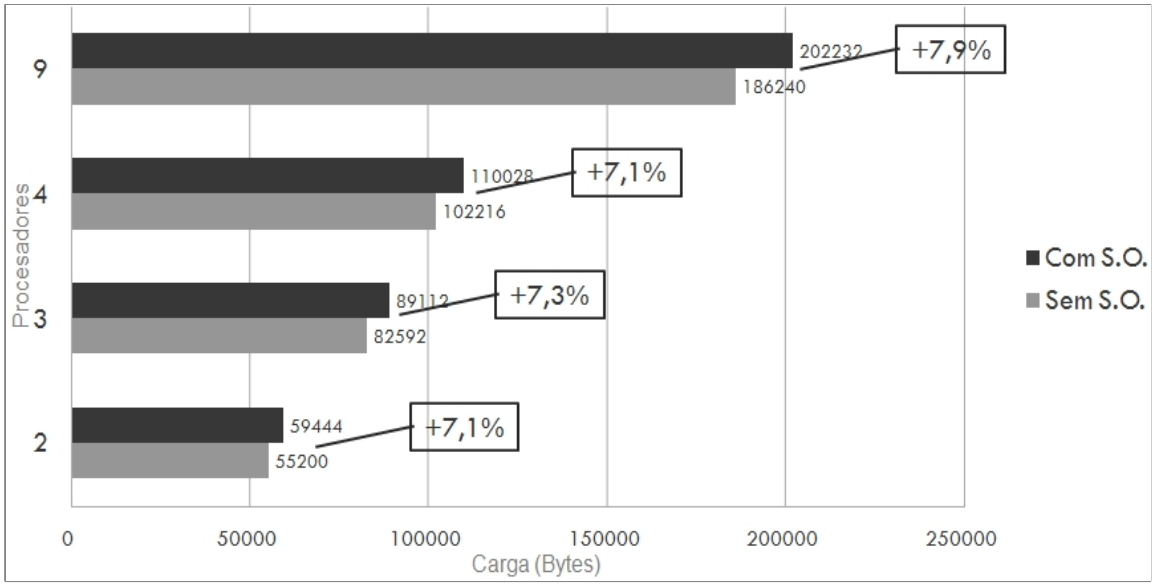

Figura 6.3 - Carga total da NoC

O aumento médio de 7% na carga injetada na NoC dá-se essencialmente pelo incremento dos cabeçalhos de Sistema Operacional, para o algortimo SOR são usados essencialmente pacotes de tamanho grande, pois é necessário copiar grandes volumes de dados, as matrizes de coeficientes exibidas na seção 6.1 são dividas entre os vários nós e copiadas, entre eles.

Considerando a limitação da NoC usada na STORM de tamanho máximo do pacote de 32 palavras e que a adição do sistema operacional incluiu 2 palavras de cabeçalho a mais em cada pacote de MPI, apenas esse acréscimo já adicionaria 6,25% de carga na NoC. Além do cabeçalho adicional, informações de controle do MPI e dos semáforos distribuídos, comentados na seção 5.2 adicionam mais carga estando então na média de 7% de acréscimo de overhead do Sistema Operacional, em termos de carga.

6.3. Inicialização do sistema operacional

O processo de inicialização do sistema operacional inclui algumas tarefas que preparam o ambiente para a execução de tarefas. Entre estas tarefas estão: (i)Inicialização de estruturas de controle do sistema, como: tabelas de processos, tabelas de arquivos abertos, gerenciadores de semáforo e mutex; (ii) Inicialização do MPI; (iii) Criação dos processos iniciais internos do sistema operacional.

A inicialização das estruturas de controle do sistema operacional é um passo de tempo constante, dado que estas estruturas têm sempre o mesmo tamanho. Esta etapa consome 600065 ciclos. Sendo que os primeiros 60 ciclos são consumidos pela plataforma para carregar um loader do endereço 0 da memória e definir uma pilha de execução inicial.

O gráfico da Figura 6.4 mostra o tempo de inicialização dos vários elementos do sistema operacional, há de se perceber que o tempo total é dominado pela criação das estruturas de gerenciamento de semáforos e mutex e gerência de I/O isso acontece pois essas duas fases da inicialização necessitam alocar blocos de memória para armazenar informações de controle, como a posse de cada semáforo e inicializar a lista de dispositivos de I/O.

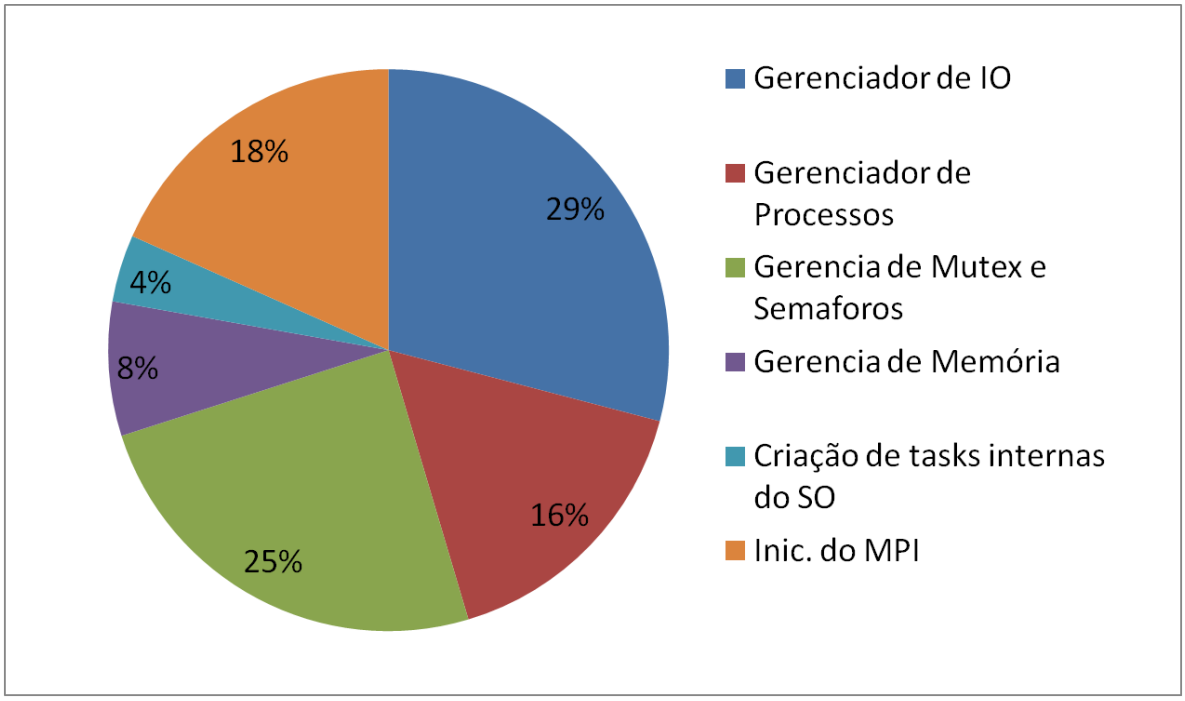

Figura 6.4 - Tempo das atividades do boot

Outra observação importante é que entre as diversas atividades do boot a única atividade que não tem tempo constante, sendo dependente da aplicação, é a Inicialização do MPI, o fator mais importante nesta fase do boot é a quantidade de processos em cada nó, já que na inicialização do MPI é necessário enviar ao nó mestre o número de processos em cada nó e só então enviar a cada nó os ranks dos processos dos vários nós.

Na fase de inicialização do MPI todos os processos participantes do MPI enviam ao nó mestre do MPI seus PIDs após receber todos os PIDs o nós mestre atribui um rank a cada processo e devolve aos nós estes dados. Assim o tempo despendido nesta tarefa é dependente do número de processos participantes do MPI. Para uma instância de 2 nós temos 8945 ciclos com 2 processos em cada nó, já para a mesma instância com 3 processos em cada nó temos um custo de 9402 ciclos, o gráfico da Figura 6.5 exibe a variação do tempo em ciclos.

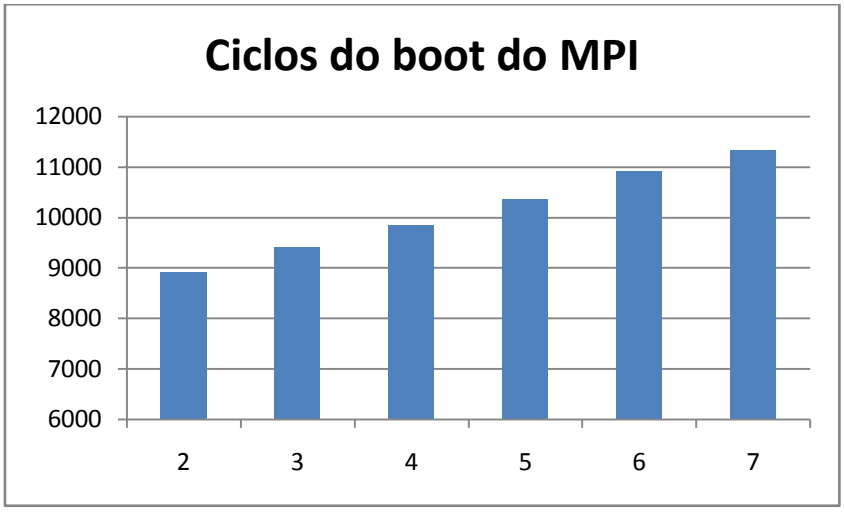

Figura 6.5 - Ciclos para o boot do MPI por número de processos

A criação dos processos iniciais do sistema operacional é simples, já que apenas duas tarefas são criadas a Task Idle e a Task Stat. A task Idle é chamada à execução sempre que não há um processo pronto para executar. Seu corpo é bastante simples e implementa apenas um contador. Já a task Stat gera estatísticas sobre o sistema esta é chamada a cada intervalo de tempo para fazer a contabilidade do que ocorreu naquele intervalo. Com a taskStat é possível obter informações de uso do processador, tempo idle, tempo que cada processo utilizou a CPU entre outras.

7. Conclusões e trabalhos futuros

Este trabalho buscou implementar um *micro-kernel* para a plataforma STORM capaz de gerenciar processos e alocação de memória. oferecendo ao usuário da plataforma a capacidade de utilizar melhor os recursos disponíveis, em um nível de abstração adequado.

Além disto, um conjunto de bibliotecas de software básico foi criado para este micro-kernel bibliotecas de alocação de memória, de entrada e saída, e um conjunto de chamadas do sistema operacional para comunicação inter-processo baseado no padrão MPI.

Como extensão deste trabalho há muitas frentes a serem tratadas, muitas das quais podem oferecer novos resultados, melhores e mais precisos do uso de sistemas operacionais em ambientes MPSoC.

A gerência de memória atualmente é feita internamente em um único nó, assim quando um pedido de alocação de memória é feito o espaço é buscado apenas na memória do nó local. Há uma hierarquia de memória em STORM hoje representada exclusivamente pelas caches e pela memória local, uma adição a esta hierarquia pode ser um módulo de memória externo. Outra abordagem pode ser, considerar que a memória de todos os nós está disponível para cada um dos outros, sendo que a memória localizada em um nó distante está em um nível abaixo na hierarquia de memória.

O sistema operacional não é capaz de oferecer nenhum mecanismo de proteção de acesso a memória, já que hoje a plataforma STORM não conta com nenhum mecanismo, em hardware, para controle de acesso a memória. Um avanço importante na segurança das aplicações implementadas seria a criação de uma MMU (*Memory managemente Unit*) que poderia ser acoplada ao DAMa e controlaria o acesso a memória. Este módulo poderia também ser usado para implementação de memória virtual.

Não foi o foco deste trabalho a implementação do módulo de entrada e saída em STORM, daí este módulo estar ainda muito simplificado necessitando de mais atenção no que diz respeito ao suporte a diferentes dispositivos e análise de melhores estratégias para a integração de dispositivos externos à NoC.

O protocolo definido para acesso aos dispositivos de entrada e saída garante acesso exclusivo e concorrente entre os vários nós do MPSoC, mas isso depende do nó mestre, o que gera um gargalo claro para aplicações que exijam acesso de E/S freqüente, outros modelo que diminuam este gargalo precisam ser analisados, um modelo capaz de distribuir o controle de acesso entre os nós processadores seria o mais adequado.

8. Referências

ANDERSON, J. H.; CALANDRINO, J. M.; DEVI, U. C. **Real-Time Scheduling on Multicore Platforms**. The University of North Carolina at Chapel Hill. [S.l.]. 2005.

ANDERSON, J.; SRINIVASAN, A. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. **Journal of Comp. and Sys. Sciences**, 2004. 157-204.

BORKAR, S. Y. et al. **Platform 2015: Intel Processor and Platform Evolution for the Next Decade**. Intel Corporation. [S.l.]. 2005.

BOVET, D. P.; CESATI, M. **Undesrtanding linux kernel**. 1. ed. [S.l.]: O'Reilly, 2000.

BRIDGES, M. J. et al. Revisiting the Sequential Programming Model for the Multicore Era. **IEEE Micro**, p. 12-20, Jan/Fev 2008.

CAM, H.; ABD-EL-BARR, M.; SAIT, S. M. **A high-performance hardware-efficient memory allocation technique and design**. Proceedings of International Conference on Computer Design (ICCD'99). [S.l.]: [s.n.]. 1999. p. 274-276.

DIJKSTRA, E. Cooperating sequential processes. **Technological University**, Eindhoven, Setembro 1965.

FANCHI, J. R.; HARPOLE, K. J.; BUJNOWSKI, S. W. **BOAST:** A three-dimensional, three-phase black oil applied simulation tool. Oklahoma: s.e., 1982.

IEEE STANDARD SOCIETY. **IEEE Open SystemC Language Reference Manual**. [S.l.], p. 441. 2006.

KERNIGHAN, B. W.; RITCHIE, D. M. **The C programming language**. 2. ed. [S.l.]: Prentice Hall.

KLEVIN, T. Get RealFast RTOS with Xilinx FPGAs. **Xcell Journal**, Fevereiro 2003.

LABROSSE, J. J. **MicroC/OS-II The Real-Time Kernel Second Edition**. 2. ed. Berkeley: CMP Books, 2002.

LIU, C.; LAYLAND, J. Scheduling algorithms for multiprogramming in a hard real-time environment. **Journal of ACM**, v. 20, p. 46-61, 1973.

LIU, R. et al. **Tessellation:** Space-Time Partitioning in a Manycore Client OS. HotPar09. Berkeley: [s.n.]. 2009.

MASSA, A. J. **Embedded Software development with eCos**. 1. ed. New Jersey: PRENTICE HALL, 2002.

MEDARDONI, S. et al. **Capturing the interaction of the communication, memory and I_O subsystems in memory-centric industrial MPSoC platforms**. DATE07. [S.l.]: [s.n.]. 2007. p. 660-665.

MOYER, B. Getting Back to a Simpler Life A New Multicore Communications API is Ratified. **Embedded Technology Journal**, 2008.

MPI FORUM. **A Message-Passing Interface Standard, Version 2.1**. Disponível em:< http://www.mpi-forum.org/docs/mpi21-report.pdf>. 2008.

NETBSD FOUNDATION. NetBSD Internals. **NetBSD Internals**, 2008. Disponivel em: <http://www.netbsd.org/docs/internals/en/index.html>. Acesso em: 21 Março 2008.

NOLLET, V.; VERKEST, D.; KAPELDREEF, L. **A Quick Safari Through the MPSoC Run-Time Management Jungle**. Embedded Systems for Real-Time Multimedia, 2007. ESTIMedia 2007. IEEE/ACM/IFIP Workshop on. New York: [s.n.]. 2007. p. 41-46.

OLIVEIRA, B. C. **Simulação de Reservatórios de Petróleo em Ambientes MPSoC**. Dissertação (Mestrado em Sistemas e Computação) – Programa de Pós-Graduação em Sistemas e Computação, UFRN. Natal, p. 66. 2009.

PAULIN, P. G. et al. **Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management**. CODES+ISSS'04. Stockholm: [s.n.]. 2004.

QNX SOFTWARE SYSTEMS. Community portal for QNX software developers. **Community portal for QNX**, 2009. Disponivel em: <http://community.qnx.com/sf/docman/do/listDocuments/projects.core_os/docman.root.adaptive_partitioning>. Acesso em: 1 Maio 2009.

QNX SOFTWARE SYSTEMS GMBH & CO. KG. **QNX 4 to QNX Neutrino Migration Guide**. QNX Software Systems. Ontario, p. 135. 2006.

REGO, R. S. L. S. **Projeto e implementação de uma plataforma MPSoC usando SystemC**. Dissertação (Mestrado em Sistemas e Computação) – Programa de Pós-Graduação em Sistemas e Computação, UFRN. Natal. 2006.

SANGIOVANNI-VICENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. **IEEE Design & Test of Computers**, Los Alamitos, 18, Nov/Dez 2001. 23-33.

SHALAN, M. A.; MOONEY, V. **A Dynamic Memory Management Unit for Embedded Real-Time System-on-a-Chip**. International Conference on Compilers, Architecture and Synthesis for Embedded Systems. [S.l.]: [s.n.]. 2000. p. 180-186.

SHALAN, M.; MOONEY III, V. J. **Hardware support for real-time embedded multiprocessor system-on-a-chip memory management**. CODES'02. Colorado: [s.n.]. 2002. p. 79-84.

SILVA, F. A. E. A. Parallelizing Black Oil Reservoir Simulation Systems for SMP Machines. **Proceedings of the 36th annual symposium on Simulation**, Washington, 2003. 224.

SPARC INTERNATIONAL, INC. **The SPARC architecture manual, version 8**. Prentice Hall. [S.l.]. 1992.

SRINIVASAN, A.; ANDERSON, J. **Optimal rate-based scheduling on multiprocessors**. Proc. of the 34th ACM Symp. on Theory of Comp. [S.l.]: [s.n.]. 2002. p. 189-198.

STRÖMBLAD, P. **ENEA Multicore: High performance packet processing enabled with a hybrid SMP/AMP OS technology**. Enea AB. [S.l.]. 2009.

TANENBAUM, A. S. **Modern Operating Systems**. 3. ed. Upper Saddle River: Prentice Hall, 2008.

THE OPEN GROUP. **The System Interfaces volume of POSIX.1-2008**. Disponível em: http://www.opengroup.org/onlinepubs/9699919799/. 2008.

TILERA. **Tile-GX Processor Family**. Disponível em: http://www.tilera.com/products/processors/TILE-Gx_Family. 2009.