UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE
UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALEXANDRO LIMA DAMASCENO

O IMPACTO DA HIERARQUIA DE MEMÓRIA SOBRE A ARQUITETURA IPNOSYS

MOSSORÓ - RN
2016

ALEXANDRO LIMA DAMASCENO

O IMPACTO DA HIERARQUIA DE MEMÓRIA SOBRE A ARQUITETURA IPNOSYS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação - associação ampla entre a Universidade do Estado do Rio Grande do Norte e a Universidade Federal Rural do Semi-Árido, para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Profº Sílvio Roberto Fernandes de Araújo, D.Sc.
Coorientador: Profº Gustavo Girão Barreto da Silva, D.Sc.

MOSSORÓ - RN
2016

D155i Damasceno, Alexandro Lima.
Impacto da Hierarquia de Memória sobre a Arquitetura IPNoSys / Alexandro Lima Damasceno. - 2016.
84 f. : il.

Orientador: Sílvio Roberto Fernandes.
Coorientador: Gustavo Girão Barreto da Silva.
Dissertação (Mestrado) - Universidade Federal Rural do Semi-árido, Programa de Pós-graduação em Ciência da Computação, 2016.

1. Hierarquia de Memória. 2. Arquitetura IPNoSys. 3. Redes em chip. I. Fernandes, Sílvio Roberto, orient. II. da Silva, Gustavo Girão Barreto, co-orient. III. Título.

ALEXANDRO LIMA DAMASCENO

O IMPACTO DA HIERARQUIA DE MEMÓRIA SOBRE A ARQUITETURA IPNOSYS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de Mestre em Ciência da Computação.

APROVADA EM: 27 / 07 / 2016.

BANCA EXAMINADORA

Sílvio Roberto Fernandes de Araújo, D.Sc.
Orientador

Gustavo Girão Barreto da Silva, D.Sc.
Co-orientador

Karla Darlene Nepomuceno Ramos, D.sc.
Avaliadora do Programa

Ivan Saraiva Silva, D.sc.
Avaliador Externo

Ao quarto homem da fornalha.

Agradecimentos

Primeiramente agradeço a Deus que sempre me guiou em todas minhas decisões e me mostrou que às vezes o que nos parece uma derrota é apenas a prorrogação de uma vitória muito maior.

Aos meus pais, Aurissandro e Irani, os melhores pais do mundo que mesmo com toda dificuldade, sempre se esforçaram para me dar tudo que precisei, me influenciaram e apoiaram em cada momento difícil. Nos momentos de solidão e angústia eram meu porto seguro, o ombro amigo que me consolava.

Aos meus orientadores, Silvio Fernandes e Gustavo Girão, que com muito empenho, bom humor e experiência profissional fizeram nossas reuniões mais produtivas, utilizando os meios certos para me fazer superar minhas limitações.

Aos amigos e familiares que sempre acreditam na minha capacidade, obrigado por compartilharem e comemorarem comigo cada etapa concluída.

À minha esposa, Larissa Damasceno, pelo apoio e confiança depositados em mim e por todo amor, carinho e compreensão nos momentos mais difíceis.

Agradeço à CAPES, pelo financiamento deste trabalho.

″Que os nossos defeitos reguem as sementes.″
Autor Desconhecido

Resumo

Ao longo dos anos, com a ascensão das tecnologias, a busca por melhorias no desempenho dos sistemas computacionais é algo notável. Os sistemas computacionais evoluíram tanto em capacidade de processamento como em complexidade das arquiteturas implementadas. Nesses sistemas é crucial a utilização de memórias uma vez que elas são responsáveis pelo armazenamento de dados que serão processados. Considerando um ambiente ideal, as memórias deveriam ter uma capacidade de armazenamento ilimitado, o acesso de dados imediato e o custo por bit extremamente baixo. Porém nos sistemas reais as memórias não apresentam essas características. Capacidade de armazenamento, velocidade e custo por bit são fatores que crescem proporcionalmente entre si. Uma técnica que é utilizada para balancear esses fatores e melhorar o desempenho dos sistemas computacionais é a hierarquia de memória. No cenário de novas tecnologias e propostas de novas organizações de processadores, um modelo que vem sendo adotada pelos projetistas de sistemas computacionais é o uso de MPSoCs (sistemas multiprocessados integrados em chip), que apresenta uma maior eficiência energética e computacional. Nesse cenário com muitos elementos de processamento, a utilização de redes em chip (NoC - *networks-on-chip*) se mostra mais eficiente que o uso de barramentos. Uma NoC consiste em um conjunto de roteadores e canais interligados formando uma rede chaveada. Os núcleos são conectados aos terminais da rede e a comunicação ocorre pela troca de pacotes. Essas NoCs foram tradicionalmente projetadas exclusivamente para a comunicação em SoCs. Entretanto, um projeto de uma arquitetura não convencional resolveu integrar processamento e comunicação em uma NoC. Essa arquitetura é conhecida por IPNoSys. A arquitetura IPNoSys (*Integrated Processing NoC System*) é um processador não convencional que utiliza redes em chip e implementa unidades de processamento e roteamento para tratar e processar instruções. Aproveita as características das NoCs, como escalabilidade e comunicação paralela, para implementar de maneira eficiente execuções de programas que exploram paralelismo em nível de threads. Atualmente, a arquitetura IPNoSys possui quatro memórias fisicamente distribuidas nos cantos da rede, mas que representam um endereçamento unificado. Cada módulo de memória é associado a uma unidade de acesso que se encarregam de gerenciá-la. Diante da atual organização de memórias da IPNoSys, esse trabalho desenvolveu um novo sistema de hierarquia de memórias para o IPNoSys e investigou os possíveis impactos sobre o desempenho e o modelo de programação.

Palavras-chave: Hierarquia de Memória, Arquitetura IPNoSys, Redes em Chip.

Abstract

Over the years, with the as technology advances, the search for improvements in the performance of computer systems is notable. The computer systems have evolved in both processing capacity and complexity of the implemented architectures. In such systems it is crucial to use memories since they are responsible for storing data to be processed. Considering an ideal environment, the memories should have a unlimited storage capacity, instant data access and the extremely low cost per bit. But in real systems the memories do not exhibit these characteristics. Storage capacity, speed and cost per bit are factors that increase in proportion to each other. One technique that is used to balance these factors and improve the performance of computer systems is the memory hierarchy. In the scenario of new technologies and proposals for new organizations of processors, a model that has been adopted by designers of computer systems is the use of MPSoCs (multiprocessor systems on chip), which has a higher energy and computational efficiency. In this scenario with many processing elements, networks using on-chip (NoC - *networks-on-chip*) is more efficient use of the buses. An NoC consists of a set of routers and interconnected channels forming a switched network. The cores are connected to network terminals and communication occurs through the exchange of packets. These NoCs have traditionally been exclusively designed for communication SoCs. However, a project of an unconventional architecture decided to integrate processing and communication in an NoC. This architecture is known for IPNoSys. The IPNoSys (*Integrated Processing NoC System*) architecture is an unconventional processor that uses networks on chip and implements processing units and routing to handle and process instructions. It takes advantage of the characteristics of NoC, such as scalability and parallel communication, for implement effectively runs programs that exploit parallelism-level threads. Currently, IPNoSys architecture has four memory physically distributed at the corners of the network, but represent a unified addressing. Each memory module is associated with an access unit in charge of managing it. Given the current organization of IPNoSys memories, this work proposes to develop a new memory hierarchy system for IPNoSys and investigate the possible impact on performance and the programming model.

Key-words: memory hierarchy, IPNoSys Architecture, networks on Chip.

Lista de Figuras

Lista de tabelas

Lista de abreviaturas e siglas

CU Control Unit

DRAM Dinamic Random Access Memory

EPROM Erasable Programmable Read-Only Memory

EEPROM Eletrically Erasable Programmable Read-Only Memory

IPNoSys Integrated Processing NoC System

MAU Memory Access Unit

MM Memory Manager

MP-SoC Multiprocessor System-on-Chip

NoC Network-on-Chip

RPU Routing and Processing Unit

SRAM Static Random Access Memory

SU Synchronization Unit

SoC System-on-Chip

SSD Solid State Drive

Sumário

1 INTRODUÇÃO

Atualmente, a busca por melhorias no desempenho dos sistemas computacionais é algo contínuo. Inúmeras técnicas são adotadas para atingir um nível de desempenho melhor. Desde o surgimento da arquitetura de Von Neumann esses sistemas passaram a possuir memórias que armazenam dados e instruções que são utilizados na execução de programas, estas memórias também tornaram-se alvo de várias pesquisas visando a melhor forma de organização para otimizar o desempenho dos sistemas, uma vez que o tempo necessário para que o processador obtenha um dado na memória afeta diretamente o seu desempenho. A técnica de organização de memórias mais conhecida é a hierarquia de memória que consiste em organizar em níveis, diferentes tipos de memórias, explorando e tirando proveito de suas melhores características, balanceando custo, tempo de acesso e capacidade de armazenamento.

Nos dias atuais, já é um fato a utilização de sistemas multiprocessados que exploram o paradigma de paralelismo de tarefas. Outro fato é a ascensão de sistemas fisicamente pequenos com um alto poder de processamento. Recentemente, já é possível a integração de muitos componentes de *hardware* dentro de um único *chip*, esses sistemas são chamados de SoCs (*System-on-Chip*). Nos sistemas dessa natureza, a distância física entre os componentes deixou de ser um problema, já que todos os componentes estão dentro do mesmo *chip*. Nos SoCs, o mecanismo de interconexão e comunicação ou subsistema de comunicação, é tão relevante quanto o próprio sistema formado pelos componentes. O subsistema de comunicação influencia no mecanismo de comunicação entre os componentes e, por consequência, o desempenho do sistema como um todo (ARAÚJO, 2008). Uma solução para os subsistemas de comunicação dos SoCs são as redes em *chip* ou NoCs (*Network-on-Chip*).

As NoCs são baseadas nos modelos de redes de computadores, com a diferença de estarem dentro do *chip* e, consequentemente, estarem restritas às características próprias desse ambiente. Assim, na literatura, as NoCs têm sido uma das soluções mais utilizadas para sistemas multiprocessadores em *chip* único ou MP-SoC (*Multiprocessor SoC*) (ARAÚJO, 2008).

Diante disso, esse trabalho tem como objetivo aliar as características das NOCs com as características da hierarquia de memórias, desenvolvendo propostas diferentes de aplicações dessa hierarquia para então verificar a sua capacidade de melhorar o desempenho de sistemas computacionais, aplicando-a em uma arquitetura não convencional. Para isso, foram desenvolvidos projetos de hierarquia de memórias para a arquitetura IPNoSys (*Integrated Processing NoC System*) desenvolvida por Araújo (2008) que já explora as vantagens existentes nas NoCs, como paralelismo, reusabilidade e escalabilidade, e analisar os impactos dessa integração no desempenho geral do

sistema.

Como objetivos específicos pode-se destacar:

- Desenvolver modelos de hierarquia de memória diferentes, destacando suas características e avaliando seus impactos sobre a arquitetura IPNoSys.
- Aplicar mecanismos de manutenção de coerência de memórias visando melhorar o modelo de programação da IPNoSys diminuindo a chance de erros por conta da possível desatenção do programador.

1.1 MOTIVAÇÃO

Na arquitetura IPNoSys existem quatro módulos de memórias, estes módulos são totalmente independentes, porém possuem replicação de dados de tal forma que todos os dados utilizados na execução de um programa estão disponíveis em todos os módulos. Além dessa replicação de dados, existem ainda algumas informações que são exclusivas de cada memória (os pacotes). No modelo de programação da IPNoSys, o programador deve identificar em que memória cada pacote deve ficar armazenado. Caberá a essa memória injetar tal pacote na rede para que seja executado. No início da execução do programa todos os dados são replicados em todas as memórias, já os pacotes são colocados apenas nas memórias para as quais foram identificadas pelo programador.

O modelo de programação desse sistema mostra-se mais complexo para o programador, já que o modelo de memória adotado é o modelo de memória distribuída. Nesse caso, o programador deve ter plena noção para onde endereçar cada pacote, sendo de sua total responsabilidade garantir a coerência de memória. Um exemplo dessa complexidade é quando ocorre um acesso ao mesmo dado (variável) em módulos de memória diferentes. Se esses dois acessos forem de leitura de dados, não há nenhum tipo de problema, mas basta que um dos acessos seja de escrita para que o dado fique incoerente, pois um mesmo dado apresentará valores diferentes em módulos de memórias diferentes. Portanto, atualmente o programador deve garantir que ao haver escrita de um dado em um determinado módulo de memória, todos os demais acessos àquele dado também serão feitos neste mesmo módulo.

Nessa arquitetura apenas duas instruções fazem acesso a dados na memória, a instrução LOAD(busca de dados na memória) e a instrução STORE(guarda dados na memória). Na execução da instrução LOAD é enviado para a MAU um pacote com a requisição do dado, o processamento fica parado até que a resposta da requisição com o dado chegue de volta à RPU que solicitou. Diante disso essa instrução é considerada o caminho crítico dessa arquitetura, uma vez que seu tratamento pode gerar grande latência na rede. Por conta disso, o modelo de programação da IPNoSys orienta ao programador utilizar a menor quantidade possível de instruções do tipo LOAD e

STORE. Isso é possível, graças a uma característica da arquitetura IPNoSys, que permite o envio de resultados intermediários das instruções executadas para serem usados como operandos de outras instruções sem a necessidade de acessar a memória.

Baseado em simulações, é possível observar que IPNoSys dedica até 88% do tempo total de execução das aplicações executando acesso à memória. A Tabela 1 mostra algumas aplicações e os resultados de tempo total de execução e tempo de acesso à memória obtidos.

Aplicação	**Tempo Total(ns)**	**Tempo de Acesso à Memória(ns)**	**Relação(%)**
Acumulador	14020	9330	65
DCT	390420	313190	80
RLE	51730	45850	88
Mult. Matriz	270330	212000	78

Tabela 1 – Comparativo entre o Tempo de Acesso à Memória em Relação ao Tempo Total - IPNOsys Original

A utilização de hierarquia de memória em sistemas computacionais é uma forma de reduzir o tempo de acesso aos dados ou instruções e consequentemente melhorar o desempenho de processamento. Na arquitetura IPNoSys, a hierarquia de memória também pode melhorar o modelo de programação de aplicações, mais detalhes serão apresentados nos próximos capítulos. Levando em consideração que o tempo de execução de algumas instruções da arquitetura IPNoSys depende de alguns fatores como: distância entre a RPU que solicita e a MAU solicitada que acessa o dado, o tráfego de pacotes de controle na rede nesse caminho entre a RPU e a MAU e o tráfego de acesso à memória onde o dado está sendo buscado. Utilizando hierarquia de memória que mantenha a coerência de dados nas memórias também é possível melhorar o desempenho total da arquitetura. Uma vez que a coerência de memória é garantida em todos os módulos de memória, os dados poderão ser acessados para leitura ou escrita em qualquer módulo. Logo o dado pode ser acessado a partir de um módulo mais próximo da RPU solicitante diminuindo tráfego de pacotes de controle e tempo de execução.

Neste trabalho foram desenvolvidos três modelos de hierarquia de memória com características organizacionais diferentes para verificar pontos positivos e negativos de suas aplicações sobre a arquitetura IPNoSys. Resultados mostram que foi possível melhorar o desempenho reduzindo o tempo de acesso à memória em até 3 vezes. O primeiro modelo ainda permitiu o desenvolvimento de uma funcionalidade nova que possibilita a busca de dados e/ou instruções no módulo de acesso mais próximo a RPU que requisita, diminuindo o tempo de espera e a quantidade de *hops* na rede. O segundo modelo apresenta um número maior de interfaces de comunicação entre as RPUs e as MAUs, diminuindo ainda mais a quantidade de *hops* na rede, consequentemente reduzindo o tempo de espera. O terceiro modelo apresenta uma organização das caches

associadas a cada RPU dentro da rede apresentando um latência na rede menor que nos outros modelos.

O texto está organizado da seguinte forma: o Capítulo 2 apresenta os fundamentos teóricos da hierarquia de memórias e a arquitetura IPNoSys. No Capítulo 3 são apresentados alguns trabalhos relacionados. O Capítulo 4 descreve as propostas de hierarquia de memória para serem aplicadas sobre a IPNoSys. O Capítulo 5 exibe os resultados obtidos. E, no Capítulo 6 são expostas as considerações finais e os trabalhos futuros que podem ser desenvolvidos a partir das contribuições desse trabalho.

2 REFERENCIAL TEÓRICO

Neste capítulo, são expostos conceitos gerais em relação à hierarquia de memórias, a arquitetura IPNoSys e seu modelo de distribuição de memórias, para um melhor embasamento sobre o assunto que será tratado neste trabalho.

2.1 Hierarquia de Memórias

Nos sistemas computacionais é crucial a utilização de memórias, uma vez que elas são responsáveis pelo armazenamento de dados que serão processados. Considerando um ambiente ideal, as memórias deveriam ter uma capacidade de armazenamento ilimitado, o acesso de dados imediato e o custo por bit extremamente baixo, porém nos sistemas reais as memórias não apresentam essas características. Capacidade de armazenamento, velocidade e custo por bit são fatores que crescem proporcionalmente entre si.

A diferença entre os diversos tipos de memórias se dá por causa do tipo de tecnologia de fabricação adotada para cada uma delas, onde essa tecnologia pode implicar em vantagens que reduzem o tempo de acesso ou garantem uma maior capacidade. As memórias podem ser divididas basicamente em dois tipos: Voláteis e Não-voláteis. As memórias voláteis só retém os dados gravados enquanto possui eletricidade, de modo que no momento em que deixam de ser alimentadas eletricamente os dados são perdidos. Já nas memórias não-voláteis isso não acontece. Entre as tecnologias de fabricação de memórias mais conhecidas, estão:

- Tecnologia SRAM (Static Random Access Memory) - volátil: utilizada para implementar as memórias caches, essa tecnologia possui circuitos eletrônicos baseados em semicondutores. Ela prioriza a velocidade, porém tem um custo por bit relativamente mais elevado se comparada com outras tecnologias.
- Tecnologia DRAM (Dinamic Random Access Memory) - volátil: utilizado para implementar a memória principal de um sistema, essa tecnologia apresenta um custo por bit menor que a SRAM, embora seja substancialmente mais lenta. A diferença de preço ocorre porque as DRAMs utilizam significativamente menos área por bit de memória, portanto têm maior capacidade para armazenar a mesma quantidade de silício.
- Tecnologia de meio magnético - não-volátil: possibilita a fabricação de dispositivos de modo a armazenar informações sob a forma de campos magnéticos. Por possuírem características magnéticas, as memórias fabricadas com essa tecnologia

são não-voláteis. Devido à natureza eletromecânica de seus componentes e à tecnologia de construção em comparação com memórias de semicondutores, esse tipo é mais barato e permite, assim, o armazenamento de grande quantidade de informação. O método de acesso às informações armazenadas em discos e fitas é diferente, resultando em tempos de acesso diversos e geralmente mais elevados que em outros tipos de tecnologias.

- Tecnologia EPROM (Erasable Programmable Read-only Memory) - não-volátil: uma memória que pode ter seus dados apagados e ser reutilizada. Para apagar dados deve-se irradiar luz ultravioleta intensa para dentro do chip da memória.

- Tecnologia EEPROM (Electrically Erasable Programmable Read-only Memory) - não-volátil: assim como as memórias EPROM as EEPROM, permitem que dados sejam gravados e apagados, a diferença consiste na técnica utilizada no processo de apagamento. Essas memórias podem ser reprogramadas rapidamente aplicando-se uma tensão maior que a normal. Entretanto esse processo apaga todo o conteúdo da memória e não ocorre de forma seletiva.

- Tecnologia Flash RAM - não-volátil: essa tecnologia é considerada não-volátil, porém isso se deve ao fato de ser continuamente alimentada por uma bateria. Memórias com essa tecnologia podem ser gravadas e apagadas em blocos de maneira mais rápida e utilizando tensões comuns de um computador.

- Tecnologia SSD (Solid State Drive) - não-volátil: essa tecnologia utilizam circuitos integrados semicondutores para armazenar dados, são mais rápidas que as tecnologias de meio magnético por não possuírem partes mecânicas.

Dessa forma, ter uma memória extremamente rápida e com uma grande capacidade de armazenamento implica em um custo muito alto. Para balancear esses fatores desenvolveu-se um sistema de organização de memórias.

A hierarquia de memória tem como um de seus objetivos aumentar a velocidade de acesso aos dados no sistema. Segundo Hennessy e Patterson (2003), essa necessidade vem da grande diferença de velocidade existente entre processadores e dispositivos de memória. A Figura 1 exibe um gráfico que mostra essa diferença ao longo dos anos.

A solução tradicional para armazenar grandes quantidades de dados é a Hierarquia de Memória(TANENBAUM, 2006), que é representada normalmente por uma pirâmide que faz relação entre vários tipos de memória. Essas memórias são categorizadas entre si de acordo com suas características (Velocidade de acesso, Capacidade de armazenamento, custo) conforme como mostra a Figura 2.

Observando a Figura 2 nota-se que as memórias que estão mais no topo da pirâmide possuem uma capacidade de armazenamento menor e alto custo, porém possuem uma alta velocidade de acesso aos dados, enquanto que as memórias mais

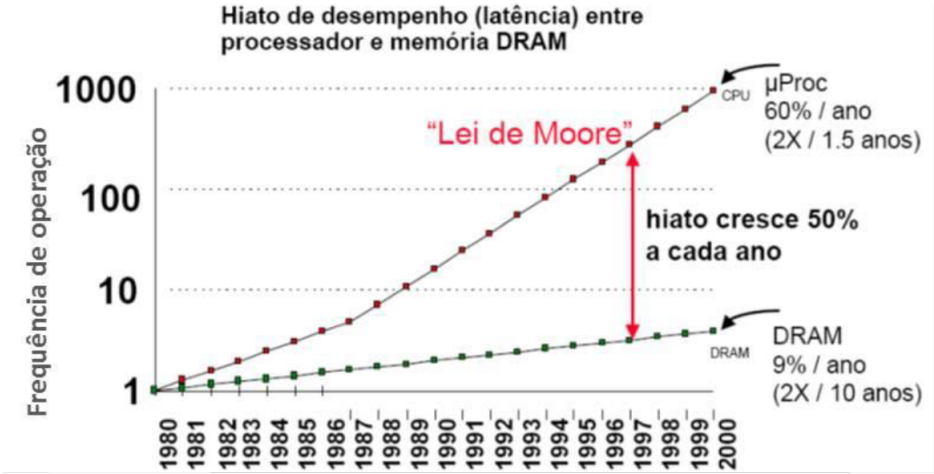

Figura 1 – Diferença de velocidade entre processadores e memórias

Fonte: (SILVA, 2009)

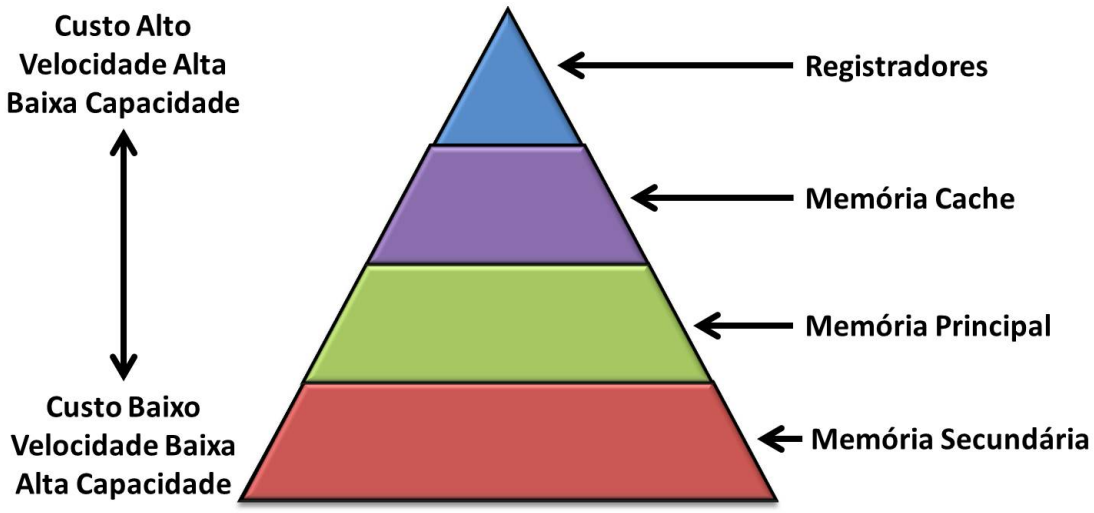

Figura 2 – Pirâmide da Hierarquia de Memória

Fonte: Próprio Autor

próximas da base da pirâmide apresentam uma capacidade de armazenamento maior e uma velocidade de acesso mais lenta, o que implica em um custo baixo.

Dessa forma ocorre um equilíbrio entre os três fatores (Armazenamento, Velocidade e Custo), uma vez que o cenário tecnológico atual impede a otimização dos três eixos para uma mesma solução de memória. As memórias rápidas são organizadas de modo a diminuírem a capacidade de armazenamento e o custo, em contra partida, para que o projeto possa armazenar grandes quantidades de dados são utilizadas memórias mais baratas (porém mais lentas), mas com uma alta capacidade de armazenamento.

Essa técnica se torna viável graças a dois princípios observados em todos as aplicações executadas pelos processadores. (HENNESSY; PATTERSON, 2003) afirma

que sistemas de hierarquia de memória com memória cache apresentam vantagem aproveitando-se do princípio de localidade. Esse princípio diz que os programas acessam uma parte relativamente pequena do seu espaço de endereçamento em qualquer instante do tempo. Existem dois tipos diferentes de localidade:

- Localidade temporal: se um item é referenciado, ele tenderá a ser referenciado novamente em breve.
- Localidade espacial: se um item é referenciado, os itens cujos endereços estão próximos tenderão a ser referenciados em breve.

A partir de então, quando o processador precisa de um dado ele faz a busca nas memórias que estão mais no topo (que são mais rápidas), caso ele não encontre o dado nelas a busca continua descendo na pirâmide hierárquica até encontrá-lo. Geralmente, quando o dado é encontrado, uma parte do contexto adjacente (os dados vizinhos) também é levada e distribuída nos níveis superiores da pirâmide hierárquica, de modo que quando o processador necessitar de outro dado ele esteja disponível nos níveis de memórias mais rápidos.

Como mencionado, a hierarquia de memória cria níveis de memórias diferentes de acordo com suas características, a solução mais adotada é a criação de um nível de memória muito próximo do processador, essa memória é muito rápida e é conhecida como memória cache. A memória cache ainda pode apresentar até três níveis diferentes que são chamados de nível L1, L2, L3 e atualmente o intel i7 Broadwell já conta com um nível L4 (INTEL, 2016).

O nível L1 surgiu quando algumas arquiteturas preferiram adicionar uma memória cache dentro do próprio chip do processador para melhorar ainda mais seu desempenho, de modo que o nível L1 estava dentro do chip e o nível L2 ficava fora do chip. Essa organização em dois níveis de cache funcionou bem durante a fase dos processadores single-core e dual-core, porém com a introdução dos processadores quad-core, passou a fazer mais sentido utilizar caches L1 e L2 menores que as do primeiro modelo e dentro do chip, criando um outro nível maior fora do chip chamado de nível L3, de modo que existe um bloco de cache L1 e L2 para cada núcleo e um bloco de cache L3 compartilhado. O nível L4 utiliza uma tecnologia conhecida como eDRAM que nada mais é que uma memória DRAM encapsulada dentro do processador.

Os níveis mais abaixo da cache são níveis de memória RAM que possuem maior capacidade, em sistemas multiprocessados esses níveis podem ser compartilhados ou distribuídos, de forma que passa a existir uma classificação de memórias quanto a sua organização física no sistema. A próxima seção discorre sobre esse assunto.

2.1.1 Classificação das memórias quanto a organização física

Nos sistemas computacionais atuais, principalmente em sistemas multiprocessados, é comum ter uma organização física de memórias mais adequado às suas necessidades. Os tipos mais comuns são apresentados nos próximos tópicos.

2.1.1.1 Memória Compartilhada

Memória compartilhada é uma memória que pode ser acessada simultaneamente por múltiplos processadores com a intenção de promover comunicação entre eles ou para evitar cópias redundantes. As vantagens de um sistema de memória compartilhada, além da simplicidade de ser programado uma vez que todos os processadores compartilham a mesma visão dos dados, é a comunicação entre processadores que pode ser tão rápida quanto o acesso à memória na mesma posição.

A principal desvantagem de sistemas com memória compartilhada é que várias CPUs necessitam de acesso rápido à memória simultaneamente e por isso utilizam sistemas de cache na própria CPU, o que possui duas complicações. A primeira é que a conexão da CPU para a memória se torna um gargalo no sistema reduzindo sua escalabilidade. A segunda, diz respeito à integridade da cache, uma vez que deve haver uma forma de manter a coerência dos dados compartilhados.

2.1.1.2 Memória Distribuída

Memória distribuída consiste em múltiplas unidades de processamento independentes com módulos de memória privados conectados por uma rede de comunicação. A principal vantagem destes sistemas é o fato de apresentarem alta escalabilidade permitindo o desenvolvimento de aplicações com um poder de computação muito alto. Possui uma maior complexidade de comunicação uma vez que esta é feita através de uma rede.

Geralmente, o acesso às informações é feito por meio de troca de mensagens (*sends* e *recives*), se o processador precisar de um dado que está na sua memória privada basta simplesmente acessá-lo, mas no caso de o dado estar na memória de outro processador ele deve requisitar através de uma mensagem e aguardar enquanto o dado é enviado.

2.1.2 Tipos de Memória

Como mencionado anteriormente, existem vários tipos de memórias que apresentam diferentes características devido às diferentes tecnologias utilizadas em suas fabricações. Nos próximos tópicos são apresentados os tipos mais comuns de memórias em uma hierarquia tradicional.

2.1.2.1 Registradores

Em um sistema de computação, a destinação final do conteúdo de qualquer tipo de memória é o processador, isto é, o objetivo final de cada uma das memórias é armazenar informações destinadas a serem, em algum momento, utilizadas pelo processador. Ele é o responsável pela execução das instruções, pela manipulação dos dados e pela produção dos resultados das operações.

No entanto, antes que a instrução seja interpretada e as unidades da CPU sejam acionadas, o processador necessita buscar a instrução de onde ela estiver armazenada (memória cache ou principal) e armazená-la em seu próprio interior, em um dispositivo de memória denominado registrador de instrução.

Após este armazenamento da instrução, o processador deverá, na maioria das vezes, buscar dados da memória para serem manipulados. Esses dados também precisam ser armazenados em algum local da CPU até serem efetivamente utilizados. Os resultados de um processamento (de uma soma, subtração, operação lógica, etc.) também precisam, às vezes, ser guardados temporariamente na CPU, ou para serem novamente manipulados por uma outra instrução, ou para serem transferidos para uma memória externa à CPU. Esses dados são armazenados em pequenas unidades de memória, denominadas registradores de dados (FILHO, 2001).

A estrutura interna de um registrador consistem em um grupo de *n flip-flops* de modo a poder armazenar em um mesmo instante *n bits*. A Figura 3 apresenta o circuito interno de um registrador de 4 *bits*.

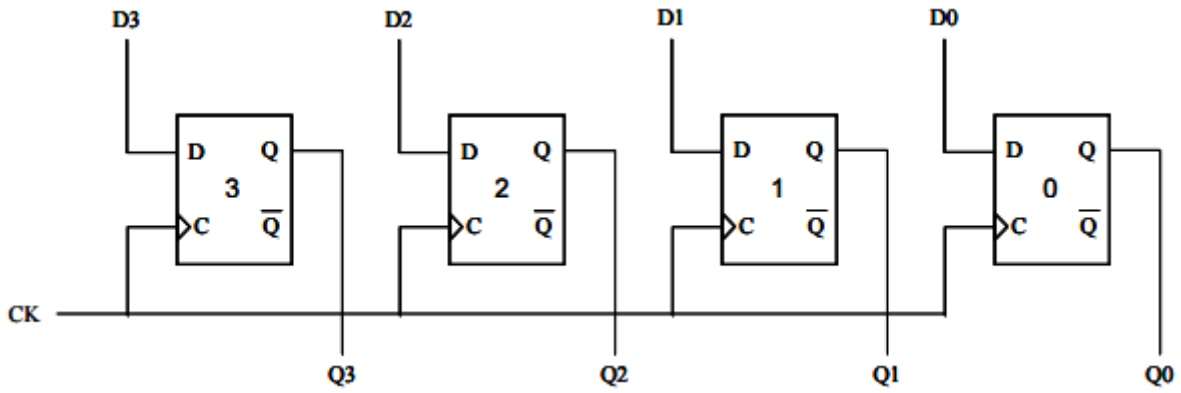

Figura 3 – Estrutura Interna de um Registrador

Fonte: (GUNTZEL, 2001)

Observe que cada *flip-flop* armazena um *bit* e que cada *bit* apresenta um caminho independente tanto para entrada quanto para saída. O registrador armazena os bits por um determinado tempo, nesse caso entre as bordas ascendentes do clock. Desse modo, os valores disponíveis em D0, D1, D2 e D3 só serão copiados quando o clock passar por uma borda ascendente.

Um registrador é, portanto, o elemento superior da pirâmide de memória, por possuir a maior velocidade de transferência dentro do sistema (menor tempo de acesso), menor capacidade de armazenamento e maior custo.

2.1.2.2 Memória Cache

A memória cache é uma pequena quantidade de memória rápida (STALLINGS, 2010), ela fica próxima ao processador e serve de intermédio entre o processador e a memória principal que possui maior capacidade de armazenamento, sua principal função é diminuir o tempo médio de acesso aos dados. A necessidade de diminuir esse tempo de acesso vem da grande diferença de velocidade entre processadores e dispositivos de memória (HENNESSY; PATTERSON, 2003).

As duas principais políticas de escritas em caches são a *write-through* (que ao realizar uma escrita, o dado alterado é imediatamente enviado para a memória para mantê-la atualizada) e a *write-back* (que é o caso do bloco alterado só ser atualizado na memória quando precisar ser retirado da cache). A utilização dessas políticas de escrita podem ocasionar a incoerência de cache, que é o fato de um mesmo dado possuir valores diferentes em diferentes caches.

Quando se utiliza compartilhamento de dados, a incoerência de cache pode acontecer da seguinte forma: considerando dois processadores, cada um com sua própria memória cache e compartilhando um mesmo módulo de memória principal, suponha-se que exista um dado X que é referenciado pelos dois processadores (Figura 4a). Imagina-se que ocorre uma escrita do tipo *write-through* por um dos processadores, a memória terá o dado atualizado (X'), mas a cache do outro processador ainda terá o dado com o valor antigo (Figura 4b). Caso utilize-se uma escrita do tipo *write-back* tanto a cache do outro processador quanto a própria memória ficarão desatualizadas (mas a memória será atualizada quando a cache que possui o dado modificado precisar substituir o bloco) (Figura 4c).

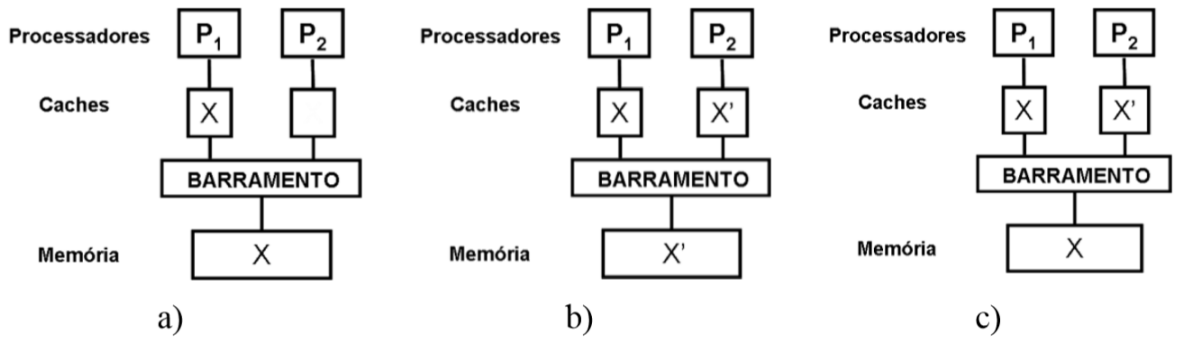

Figura 4 – Incoerência por dados compartilhados

Fonte: (HWANG, 1993)

A memória cache armazena pequenas partes (blocos) da memória principal, para isso é necessário fazer o mapeamento dos endereços da memória principal para a memória cache de modo que todos os endereços da memória principal tenham representatividade na cache. Existem três modos de mapeamento muito utilizados no projeto de memórias cache, são eles:

- Mapeamento Direto: nesse tipo de mapeamento um bloco só pode ser mapeado para uma determinada linha de cache. A Figura 5 exemplifica isso. Pode-se notar que os m primeiros blocos da memória principal são mapeados para linhas de cache diferentes. Em caso de uma memória cache muito pequena em relação a memória principal os conflitos de bloco acontecerão com maior frequência de modo que os blocos serão substituídos em cache mais vezes, isso caracteriza a maior desvantagem desse modelo. Em contrapartida, é mais simples de ser implementado e menos custoso verificar se o bloco está ou não em cache (já que ele só pode estar em um único lugar).

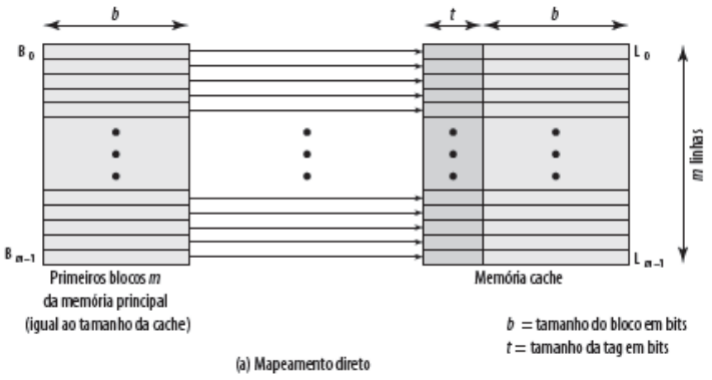

Figura 5 – Mapeamento Direto

Fonte: (STALLINGS, 2010)

- Mapeamento Associativo: esse mapeamento permite que qualquer bloco da memória principal seja mapeado para qualquer linha de cache, dessa forma diminui-se consideravelmente o conflito entre blocos e a substituição só ocorrerá quando a memória cache estiver completamente cheia. A Figura 6 mostra como se dá o mapeamento associativo. A principal desvantagem desse modelo é o custo para verificar se um bloco está ou não em cache uma vez que é necessário percorrer toda a memória (já que o bloco pode estar em qualquer linha).

- Mapeamento Associativo por Conjunto: esse modelo surgiu com o objetivo de unir as vantagens dos modelos anteriormente discutidos. Nele, a memória cache é dividida em conjuntos de modo que cada bloco da memória principal só pode ser mapeado para um único conjunto, porém, dentro desse conjunto, o bloco pode ser

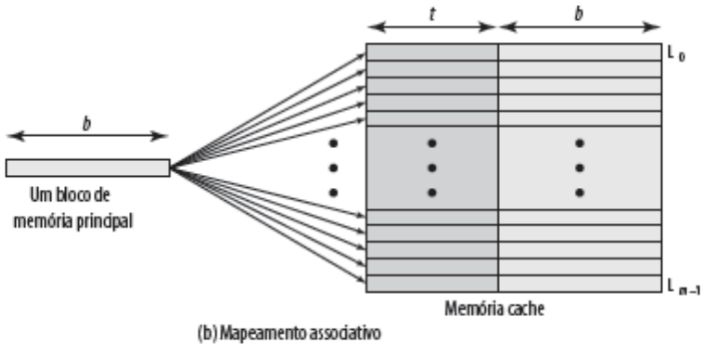

Figura 6 – Mapeamento Associativo

Fonte: (STALLINGS, 2010)

armazenado em qualquer linha. Esse comportamento pode ser visto na Figura 7. Nesse modelo de mapeamento diminui-se os conflitos entre blocos, uma vez que dentro de um conjunto eles podem ser mapeados para qualquer linha. E diminui o tempo para encontrar um bloco em cache, já que o bloco só pode estar em um determinado conjunto basta apenas percorrer todas as linhas do conjunto.

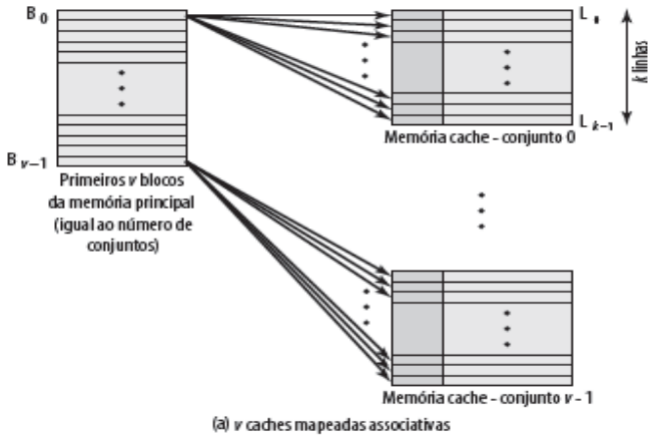

Figura 7 – Mapeamento Associativo por Conjunto

Fonte: (STALLINGS, 2010)

2.1.2.3 Memória Principal

A CPU pode acessar imediatamente uma instrução após a outra porque elas estão armazenadas no computador. Esta é a importância da memória. E, desde o princípio, a memória especificada para armazenar o programa (e os seus dados) a ser executado é a memória que atualmente chamamos de principal, para distingui-la da memória de discos e fitas (memória secundária).

A memória principal é, então, a memória básica de um sistema de computação

desde seus primórdios. É o dispositivo onde o programa (e seus dados) que vai ser executado é armazenado para que a CPU vá "buscando"instrução por instrução.

Essa memória é construída com elementos cuja velocidade operacional se situa abaixo das memórias cache, embora sejam muito mais rápidas que a memória secundária. Em geral, a capacidade da memória principal é bem maior que a da memória cache, os melhores sistemas de memória cache apresentam apenas capacidade de 12MB em contraste com sistemas simples que já apresentam memória principal com capacidade de 6GB.

2.1.2.4 Memória Secundária

A Memória Secundária, também conhecida por memória de massa ou memória auxiliar, tem por função armazenar grande quantidade de dados e evitar que estes se percam com o desligamento do computador. Essas memórias são fabricadas utilizando a tecnologia de meio magnético, EEPROM, Flash RAM e SSD que foram mencionadas anteriormente e apresentam um custo relativamente baixo se comparada às outras tecnologias de fabricação de memórias. Devido ao seu baixo custo, é comum existirem sistemas computacionais que apresentem memórias secundárias com grande capacidade de armazenamento.

São exemplos de memórias secundárias os Discos magnéticos e as Fitas magnéticas, muito utilizados nas técnicas de backups de grandes quantidades de dados. Além desses existem os HDs SSD que apresentam memórias flash RAM para armazenamento não-volátil de dados com acesso de alta velocidade.

2.2 Networks on Chip - NoC

No contexto de sistemas multiprocessados, segundo Silva (2009) a forma de comunicação é um fator crítico uma vez que o grau de paralelismo das aplicações em execução é, geralmente, dependente da capacidade de escalabilidade e baixa latência do mecanismo físico de comunicação. Uma das primeiras soluções de comunicação nesses sistemas é a utilização de barramentos. Apesar da simples implementação e do baixo custo, o barramento apresenta baixa escalabilidade tornando-se um gargalo à medida que a quantidade de dispositivos conectados aumenta.

Surgem então um novo elemento de interconexão conhecido como NoCs. São redes de interconexão baseadas em redes de computadores ou computadores paralelos. A principal diferença entre NoSCs e redes de computadores segundo Yang, Du e Han (2008) é a área limitada, dissipação de potência e os efeitos físicos na tecnologia submicrônica.

A definição de NoC por Concer, Iamundo e Bononi (2009) é a união de interface de rede, roteadores e enlace. Esses três componentes são responsáveis pela arquitetura

de comunicação de roteamento de pacotes que encapsulam informações que são transmitidas da origem até o destino através da infraestrutura de rede oferecida. Os enlaces são os canais físicos que interligam os roteadores, estes por sua vez são encarregados de encaminhar os pacotes até o destino, já a interface de rede provê a configuração de comunicação entre eles. Segundo Zeferino (2002) além do alto grau de escalabilidade, NoCs têm por principal característica a capacidade de comunicação paralela entre os elementos do sistema.

As NoCs seguem uma determinada topologia, segundo LEE e al. (2007) essa topologia de conexão pode ser configurável dando maior flexibilidade à estas redes. As topologias podem ser classificadas como diretas ou indiretas (CARARA, 2004). As topologias diretas são caracterizadas pelo fato de cada roteador estar diretamente ligado a um núcleo do sistema. Nas topologias indiretas alguns roteadores podem ser utilizados como intermediários entre roteadores que estão ligados aos núcleos. Muitos estudos foram feitos para comparar topologias em NoC visando destacar suas principais vantagens e desvantagens. As topologias diretas mais conhecidas são Grelha 2D (mesh), Torus 2D, Cubo 3D,Cubo 4D ou Hipercubo e Totalmente conectada. A Figura 8 representa essas topologias, onde os pares formados por um roteador ligado a um núcleo são apresentados como círculos.

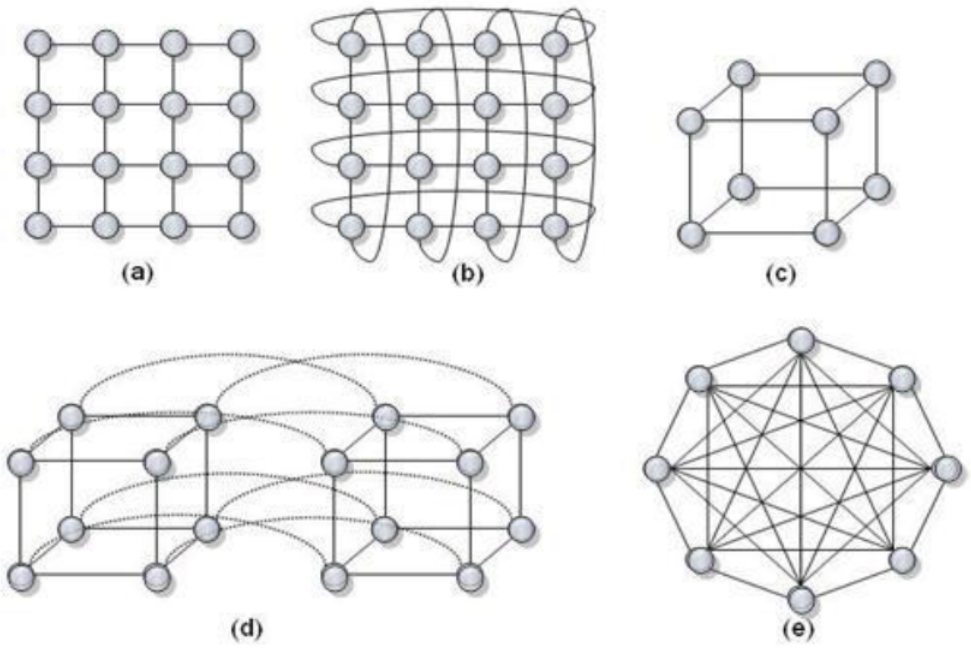

Figura 8 – Topologias Diretas: (a) Grelha 2D; (b) Torus 2D; (c) Cubo 3D; (d) Cubo 4D ou Hipercubo; (e) Totalmente Conectada

Fonte: (REGO, 2006)

2.3 Arquitetura IPNoSys

O IPNoSys é um processador de propósito geral baseado em redes em *chip* (NoCs) (ARAÚJO, 2012). As NoCs possibilitam transmissão paralela de dados e instruções em forma de pacotes, sendo processados durante seu trajeto. Esta arquitetura possibilita o armazenamento temporário de pacotes ou partes deles antes da transmissão, ocorrendo como em um pipeline de roteador em roteador até seu destino.

A Figura 9 ilustra a Arquitetura IPNoSys com topologia do tipo malha 2D de dimensão quadrada, ou seja, apresenta o mesmo número de linhas e colunas. Essa malha é formada por RPUs (*Routing and Processing Unit*) que são roteadores com capacidade de roteamento e de processamento. Os pacotes são divididos em partes e roteados através da malha de RPUs, de modo que cada parte é processada por uma RPU. Dessa forma, ao final do roteamento o pacote foi totalmente processado. Durante a execução da instrução, os pacotes passam a funcionar como sequências de instruções, onde a operação corrente é aquela que está no início do pacote e os resultados gerados podem ser inseridos em outras partes do pacote como operandos de outras instruções.

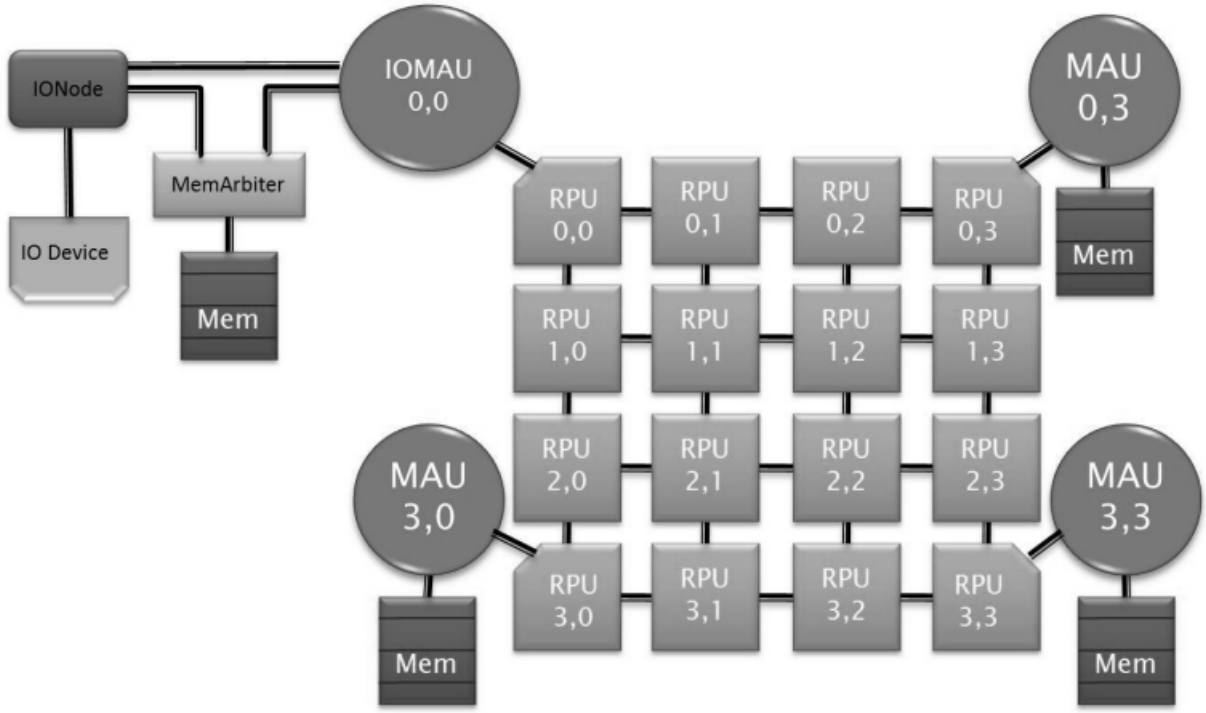

Figura 9 – Modelo da Arquitetura IPNoSys

Fonte: (ARAÚJO, 2012)

Ainda observando a Figura 9 é possível identificar o modelo de hierarquia de memória atual, onde verifica-se quatro módulos de memória distribuídos fisicamente, mas que representam um endereçamento unificado. Cada módulo de memória possui

uma Unidade de Acesso à Memória (MAU) que é responsável pelo gerenciamento da memória e dos pacotes requisitados. Existe também uma MAU especial, chamada de IO-MAU porque apresenta uma interface de comunicação com o IONode, esse componente é um mecanismo que realiza comunicação entre a IPNoSys e o mundo externo (operações de Entrada e Saída) e consiste em um módulo de acesso direto à memória. Como em algum momento pode haver concorrência entre o IONode e o módulo de memória associados a IOMAU, foi acrescentado um componente denominado MemArbiter que funciona como um árbitro que controla se está havendo um acesso à memória ou uma operação de E/S.

Como foi explicado nesta seção, a arquitetura IPNoSys processa pacotes, que podem conter instruções ou dados. Esses pacotes são armazenados na memória e injetados no sistema através das Unidades de acesso à Memória (MAU). Existem dois principais tipos de pacotes, os pacotes regulares que carregam as instruções que deverão ser executadas em qualquer elemento de processamento, e os pacotes de controle que são gerados quando uma instrução de MAU (LOAD e STORE) não pode ser executada por qualquer MAU e não é destinada a MAU corrente (a MAU que está associada a RPU que recebe a primeira instrução do pacote corrente), então a Unidade de Sincronismo (SU) cria esse pacote de controle contendo a instrução a ser executada e seus operandos, e injeta-o com destino a MAU que irá executar a instrução.

A hierarquia de memória existenta na IPNoSys original diferencia-se da hierarquia convencional por não apresentar registradores, entretando existem *buffers* que armazenam informações dentro das RPUs. A Figura 10 apresenta a estrutura da hierarquia original.

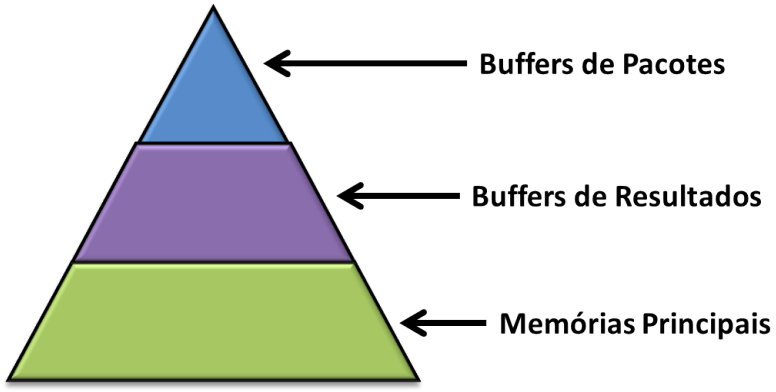

Figura 10 – Hierarquia de memória do IPNoSys Original

Fonte: Próprio Autor

O primeiro nível (mais próximo das RPUs) é formado por *buffers* de pacotes, internos à RPU, que armazenam pacotes com dados ou instruções que devem ser processadas. O segundo nível é formado pelos *buffers* de resultados que ficam dentro das MAUs, estes têm a função de armazenar dados que serão inseridos posteriormente

em algum pacote antes de ser injetado na rede. O terceiro e último nível é composto pelos módulos de memórias principal que estão associados às MAUs.

2.3.1 Unidade de Processamento e Roteamento

As unidades de processamento e roteamento, também conhecidas como RPUs (*Routing and Processing Unit*), possuem cinco portas de entrada, todas com *buffers*; possuem um módulo de roteamento que tem a função de direcionar os pacotes contidos em cada *buffer* de entrada para um saída, seguindo o algoritmo de roteamento; possuem também uma Unidade Lógica e Aritmética (ULA), onde são executadas a maior parte das instruções; cada porta de saída possui um árbitro que tem a função de controlar as disputas pelos canais de saída, requisitar e enviar dados necessários para a ULA executar uma operação.

A principal diferença entre a RPU do IPNoSys e um roteador convencional de redes-em-chip é a adição da ULA e a Unidade de Sincronismo (SU), que como mencionada anteriormente é responsável por criar pacotes de controle, os quais são usados para encaminhar instruções que serão executadas em uma MAU.

A Figura 11 mostra como está organizada a arquitetura de uma RPU, detalhando o módulo de roteamento e destacando a ULA e a SU. Observa-se a aplicação de um módulo *CrossBrar* ou módulo de roteamento utilizando os sinais de chaveamento de entrada/saída para que os árbitros decidam quem deve acessar o canal em determinado momento e enviem dados a serem processados pela ULA.

2.3.2 Unidade de Acesso à Memória

A Unidade de acesso à memória, também chamada de MAU (*Memory Access Unit*) possui a função de introduzir os pacotes na rede e controlar o acesso à memória, além de executar algumas instruções que leem ou escrevem dados na memória. Tais tarefas são executadas por MAUs através de instruções destinadas a elas nos pacotes de controle. A Figura 12 mostra como está organizada a MAU.

Observando a Figura 12 nota-se que o IPNoSys possui quatro MAUs, cada uma conectada a um módulo de memória próprio. Esses módulos de memória armazenam dados e pacotes. Existem dois tipos de pacotes, os regulares e os de controle. Pacotes regulares são os pacotes que contém instruções que serão executadas pelas RPUs, são os pacotes mais comuns no IPNoSys. Já os pacotes de controle são pacotes gerados a partir de eventos especiais como uma instrução de acesso à memória.

Durante a execução dos pacotes regulares injetados a partir de uma MAU, podem existir instruções que provocam o retorno do mesmo pacote ou de pacotes de controle para a MAU onde foram originalmente injetados ou para outra MAU. Quando isso acontece, um processo paralelo ao processo de envio da MAU, recebe o pacote. Em seguida, a MAU identifica qual o tipo de pacote recebido, que pode ser de controle, interrompido

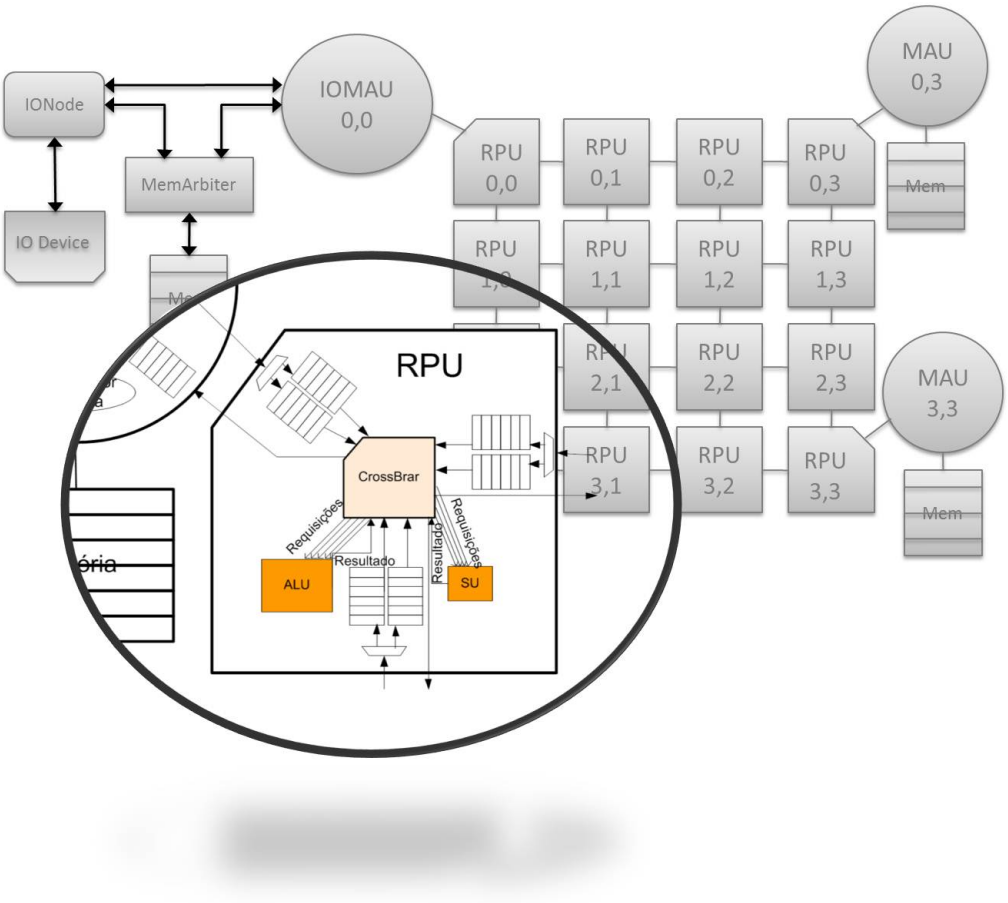

Figura 11 – Organização da RPU

Fonte: (ARAÚJO, 2012)

ou caller. O primeiro tipo, traz consigo uma instrução que deve ser executada pela MAU que o recebeu. Os outros dois tipos de pacotes são pacotes regulares que não podem continuar a execução, pois aguardam algum evento para continuarem. Deste modo, devem ser armazenados novamente na memória até que possam ser injetados mais uma vez no sistema. Assim, a MAU faz alocação de memória para esses pacotes.

Como existem quatro módulos de memórias separados no IPNoSys, o programador precisa ter noção de como os dados serão distribuídos pelas memórias. No momento em que surge uma instrução de acesso à memória essa instrução não é executada pelo RPU, ela é encapsulada em um pacote de controle e roteada para uma MAU, esta por sua vez, trata a instrução e se necessário envia algum dado para o RPU que enviou a instrução de acesso à memória.

2.3.2.1 Instruções do IPNoSys executadas pelas MAUs

O conjunto de instruções do IPNoSys é formado por 32 instruções, onde 4 são aritméticas, 4 lógicas, 2 de deslocamento, 3 de acesso à memória, 4 de sincronização,

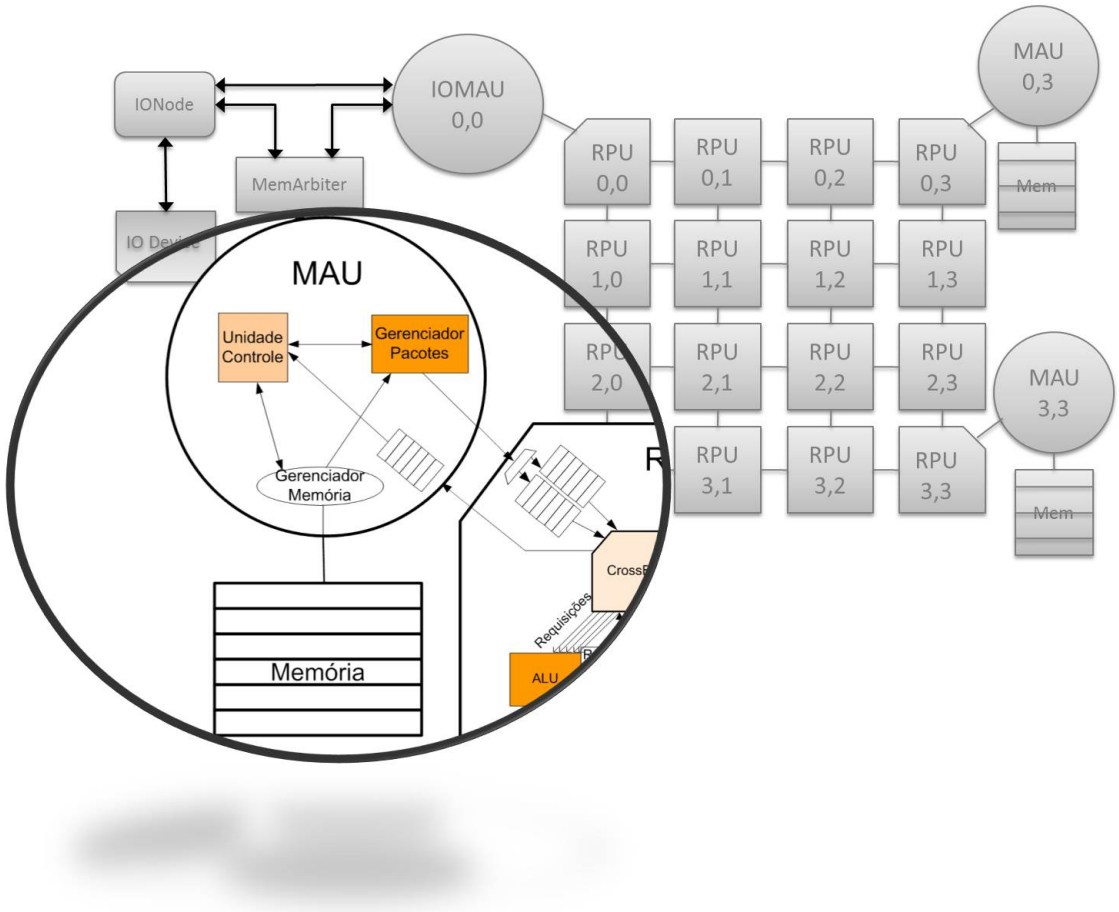

Figura 12 – Organização da MAU

Fonte: (ARAÚJO, 2012)

6 condicionais, 1 incondicional, 2 auxiliares, 2 de entrada/saída, 2 de sistemas e 2 de chamada de procedimento. Como mostra a Figura 13. Dessas instruções, 6 são executadas pelas MAUs, estas instruções são discutidas nos próximos tópicos.

- LOAD: essa instrução é enviada para a MAU para acessar um dado em uma determinada posição de memória e retornar este dado a RPU que solicitou. Quando um LOAD é decodificado por um RPU a execução é paralisada onde essa instrução foi encontrada e é gerado um pacote de controle que é encaminhado a uma MAU específica (identificada na própria instrução de LOAD), esta por sua vez, busca o dado na memória e devolve-o em outro pacote de controle para a RPU que está aguardando, só então a execução é retomada.
- STORE: essa instrução é enviada para a MAU para que um dado seja guardado na memória no endereço determinado pela instrução.
- EXEC: Quando uma MAU recebe essa instrução, ela injeta assim que possível o pacote armazenado na memória, indicado no único operando dessa instrução. Isso

Codop	Instrução	Tipo	# Operandos	Descrição
0	ADD	Aritmética	2	Soma 2 inteiros
1	SUB	Aritmética	2	Subtrai 2 inteiros
2	MUL	Aritmética	2	Multiplica 2 inteiros
3	DIV	Aritmética	2	Divide 2 inteiros
4	NOT	Lógica	1	Negação de 1 valor
5	AND	Lógica	2	Conjunção de 2 valores
6	OR	Lógica	2	Disjunção de 2 valores
7	XOR	Lógica	2	Ou-exclusivo de 2 valores
8	RSHIFT	Deslocamento	2	Desloca n bits de um valor à direita
9	LSHIFT	Deslocamento	2	Desloca n bits de um valor à esquerda
10	LOAD	Acesso à Memória	1	Solicita um valor da memória
11	STORE	Acesso à Memória	Vários	Armazena um valor na memória
12	EXEC	Sincronização	1	Ordena a injeção imediata de um pacote
13	SYNEXEC	Sincronização	Vários	Ordena a injeção de um pacote após sincronização
14	SYNC	Sincronização	1	Sinal de sincronização para um pacote
15	RELOAD	Acesso à Memória	1	Retorna um valor carregado da memória
16	BE	Condicional	2	Desvia se igual
17	BNE	Condicional	2	Desvia se diferente
18	BL	Condicional	2	Desvia se menor
19	BG	Condicional	2	Desvia se maior
20	BLE	Condicional	2	Desvia se menor ou igual
21	BGE	Condicional	2	Desvia se maior ou igual
22	JUMP	Incondicional	0	Desvia incondicionalmente
23	COPY	Auxiliar	1	Copia 1 valor para outra instrução no mesmo pac.
24	NOP	Auxiliar	0	Sem operação
25	SEND	Sincronização	2	Envia um valor para um ser inserido em um pac.
26	IN	Entrada/Saida	3	Recebe bytes do controlador de E/S
27	OUT	Entrada/Saida	3	Envia bytes ao controlador de E/S
28	WAKEUP	Sistema	1	Ordena a reinjetar um pacote antes interrompido
29	NOTIFY	Sistema	1	Notifica estado de um pacote

Figura 13 – Conjunto de Instruções do IPNoSys

Fonte: (ARAÚJO, 2012)

significa que é feita uma ordem de injeção de um pacote, a qual é colocada na fila de requisições da MAU.

- SYNEXEC: É possível que a injeção de um pacote esteja condicionada à sincronização da execução de outros pacotes. Este tipo de injeção é indicado através da instrução SYNEXEC. Essa instrução identifica o pacote que deve ser injetado, na primeira palavra de operando, e os números de todos os outros pacotes que devem enviar sinais de sincronização, nas palavras de operandos seguintes. A injeção do pacote, solicitada, acontecerá apenas depois que todos os sinais de sincronismo dos outros pacotes chegarem à MAU, que o injetará. Contudo, a MAU continua executando outras instruções que chegam até ela.

- SINC: Essa instrução é utilizada para enviar à MAU os sinais de sincronização dos pacotes, que são utilizados para saber em qual momento deverá ser injetado o pacote identificado por uma instrução SYNEXEC. De modo que, nessa instrução é identificada qual MAU deve receber o pacote e qual pacote está esperando pelos

sinais de sincronismo. Quando uma instrução SINC chega à MAU, o identificador do pacote que sinaliza sincronização é removido da lista de espera. Assim, quando a última SINC chega à MAU, todos os pacotes de sinalizadores já foram removidos e o pacote identificado pela instrução SYNEXEC é colocado na lista de requisições de envio da MAU para ser injetado.

- SEND: essa instrução é utilizada para colocar em outro pacote o valor presente no campo do segundo operando. De modo que a MAU guarda o valor enviado em uma memória específica chamada de buffer de resultados, e antes de injetar um pacote no sistema, ela verifica se há algum dado informado por uma instrução SEND anterior para inserir no pacote.

2.3.3 Modelo de Programação para IPNoSys

O modelo de programação do IPNoSys consiste em descrever as aplicações ou programas através de um ou mais pacotes, os quais são injetados e executados no sistema, na ordem de dependência dos dados entre as computações que cada pacote representa. Do mesmo modo, um pacote é um conjunto de instruções com seus respectivos dados, empilhados de acordo com a dependência de dados. Assim uma instrução só é executada quando estiver no topo dessa pilha, ou seja, no início do pacote já com todos seus operandos. Com isso, o resultado de uma instrução executada pode servir como operando para outras instruções a serem executadas posteriormente(ARAÚJO, 2012).

Para descrever as aplicações em formato de pacotes, foi desenvolvida a Linguagem de Descrição de Pacotes (PDL - Package Description Language) para ajudar o programador a desenvolver suas aplicações em um nível intermediário entre linguagem de alto nível e o formato de pacotes(ARAÚJO, 2012).

Todo programa PDL deve começar com a palavra reservada PROGRAM seguido pelo nome do programa definido pelo usuário. Em seguida deve ser feita a declaração das variáveis através da palavra reservada DATA. É possível não declarar variáveis, mas ainda assim é necessária a utilização da palavra DATA.

Depois de declarar as variáveis o programador deve declarar os pacotes, isso é feito com a palavra reservada PACKAGE ou FUNCTION seguida de um nome definido pelo usuário. Logo após vem a definição do endereço da MAU que injetará o pacote, o qual é definido através da palavra reservada ADDRESS seguida do identificador da MAU. Existem quatro MAUs, logo os identificadores delas são os seguintes: MAU_0 (MAU situada no canto superior esquerdo), MAU_1 (MAU situada no canto superior direito), MAU_2 (MAU situada no canto inferior esquerdo) e MAU_3 (MAU situada no canto inferior direito). Em seguida o programador escreve todas as instruções que definem o que o pacote deve fazer, e após todas as instruções, utilizar a palavra reservada END para marcar o fim do pacote.

No modelo de programação atual é de inteira responsabilidade do programador identificar qual MAU é responsável pela injeção de qual pacote, além de que, quando utilizar instruções de controle, deve identificar qual MAU deverá executar a instrução. Dessa forma, o programador deve ter total conhecimento sobre a arquitetura IPNoSys e como suas memórias estão distribuídas. A manutenção da coerência de dados nas memórias também é de competência do programador, de modo que ele deve garantir que após uma escrita de um determinado dado em uma das memórias as próximas leituras desse dado só poderá ser feita nessa mesma memória. Nota-se que esta forma de garantir a coerência de informações pode tornar-se um ponto crítico uma vez que a leitura do dado fica restrita a um único módulo de memória, dessa forma quando a RPU que solicita a leitura estiver fisicamente distante da MAU que gerencia essa memória é mais custoso para obter o dado uma vez que a latência na rede será maior. A nova hierarquia de memória apresenta uma solução baseada em garantia de coerência por protocolo snoop que possibilita a busca de dados sempre na MAU mais próxima, no capítulo 4 voltaremos a falar mais detalhadamente sobre esse assunto.

3 TRABALHOS RELACIONADOS

Nesse capítulo são expostos trabalhos relacionados que tratam de assuntos como hierarquia de memória em sistemas MPSoCs e máquinas multiprocessadas, simuladores de hierarquia de memória e técnicas de coerência e consistência de cache em multiprocessadores NoC, mostrando suas propostas e conclusões.

A melhor forma de armazenar coisas é algo que sempre chamou a atenção dos seres humanos, sempre buscou-se uma organização adequada que permitisse armazenar mais e poder encontrar coisas de formas mais rápida. Essa ideia foi introduzida também no mundo tecnológico com o advento dos computadores. Armazenar informações de forma organizada, além de garantir segurança facilita o acesso à essa informação e isso reflete diretamente no tempo de obtenção dessa informação. Quando tratamos de sistemas computacionais, obter informação mais rápido significa executar e processar dados mais rápido.

Considerando a organização da arquitetura de computadores, a forma de armazenar informação é guardando-a em memórias. Tais memórias evoluíram muito com o passar dos anos. A forma de organizar as memórias também sempre foi algo muito pesquisado, na literatura é possível encontrar trabalhos que desenvolvidos nos anos 1960, como o trabalho do Anderson e Glaser (1963) onde já era destacada a importância das memórias no desempenho das máquinas, ressaltando ainda que a velocidade é um derivado substancial da organização do sistema e mostrando que grande capacidade de armazenamento e alta velocidade implicavam em alto custo.

Seguindo na história dessas pesquisas, nos anos 1970 o trabalho de Patton (1970) mostra que essa preocupação em organizar as memórias se intensifica com a evolução das aplicações computacionais, destacando que essas novas aplicações necessitavam de uma hierarquia de memória que de alguma forma pudesse ser estruturada sob o controle do programa. Nos anos 1980 já são encontrados trabalhos que visam otimizar a hierarquia de memória, como o trabalho de Chanson e Sinha (1980) que apresenta técnicas de otimização em uma abordagem que tenta estimar as capacidades e velocidades ideais para os diferentes níveis da hierarquia de memória no meio multiprogramado. Ainda nos anos 1980 o trabalho de Patterson e Sequin (1980) apresenta uma estimação futurista visando as tendências gerais na evolução dos circuitos integrados. Já se discute a implementação de hierarquia de memória on-chip com várias caches homogêneas para um maior paralelismo de instruções. Os autores concluem afirmando que será possível ter um computador auto-suficiente em um único chip.

Nos anos 1990 o trabalho de Hake e Homberg (1990) pesquisa os impactos da hierarquia de memória sobre aplicações como a multiplicação de matrizes, enquanto o trabalho de Bellew, Hsu e Tam (1990) sugere um novo modelo de hierarquia de memória

chamado de distributed memory hierarchy (DMH), que é um sistema de memória consistindo em módulos de armazenamento distribuídos em rede local.

Nos dias atuais ainda existe muita pesquisa sendo desenvolvida sobre o tema hierarquia de memória, alguns desses trabalhos são melhor descritos nas seções a seguir.

3.1 Estudo sobre o Impacto da Hierarquia de Memória em MPSoCs baseados em NoC

Um estudo sobre o impacto da hierarquia de memória em sistemas multiprocessados baseados em NoC foi desenvolvido por Silva (2009). Dentro desse escopo foi desenvolvida uma nova organização de memória fisicamente centralizada com diferentes espaços de endereçamento denominada nDMA. Seu trabalho também compara o modelo nDMA com os modelos de memória distribuída, memória compartilhada e memória compartilhada distribuída. Os modelos de memória foram implementados na plataforma virtual SIMPLE (*SIMPLE Multiprocessor Plataform Environment*).

Foram feitos experimentos que testaram: performance, consumo de energia e tráfego na rede. Esses experimentos foram aplicados em quatro casos: aplicação de estimação de movimento, aplicação de multiplicação de matrizes, aplicação JPEG e aplicação Mergesort.

Os resultados obtidos mostram que o desempenho dos modelos de memória propostos apresentam uma forte dependência com relação à carga de comunicação gerada pelas aplicações. Onde o modelo de memória distribuída apresenta melhores resultados com carga de comunicação baixa. Por outro lado o nDMA apresenta melhores resultados com carga de comunicação alta.

Foram feitos experimentos que objetivam analisar o desempenho dos modelos de memória em situação de alta latência de comunicação da rede. Os melhores resultados do modelo de memória distribuída se deram quando a carga de comunicação era alta e, caso contrário, o modelo nDMA apresentou melhores resultados.

3.2 Escalonamento adaptativo ao uso da hierarquia de memória para máquinas multiprocessadas

Pillon e Richard (2004), propõem um sistema de controle adaptável (Figura 14) que visa maximizar a utilização dos recursos baseados no relacionamento entre o uso da memória e o desempenho das aplicações. A estratégia de escalonamento de DRAC (*aDaptative contRol system with hArdware performance Counters*) busca evitar a saturação no barramento de memória, permitindo o aumento de desempenho.

Para isso é realizado um estudo do relacionamento entre a utilização da hierarquia de memória e o *speed-up* em máquinas quadri-processadas.

Na Figura 14 observa-se que o protótipo do DRAC é composto de duas aplicações a nível de usuário. DRACsub, que se encarrega do lançamento e controle de submissão

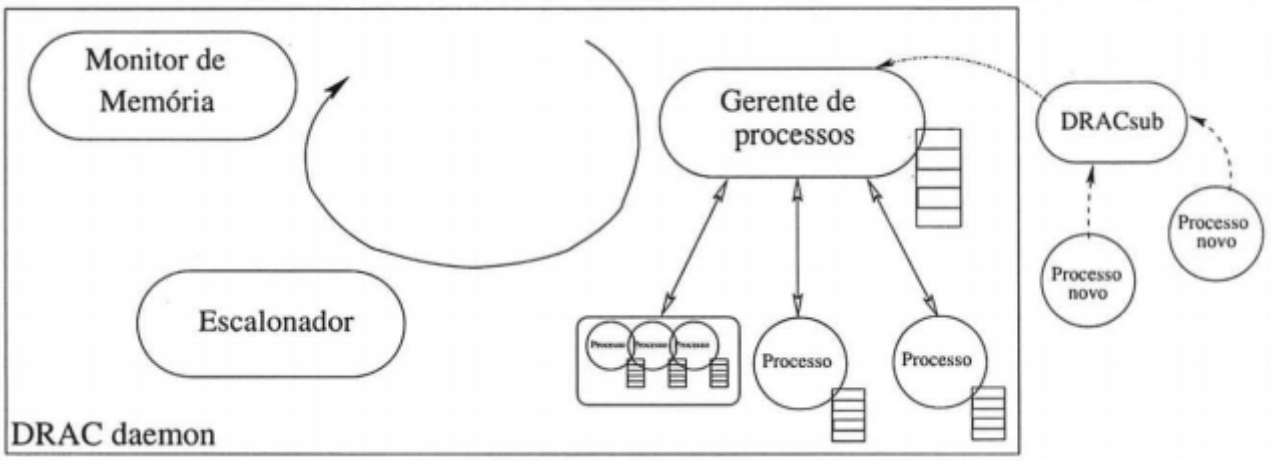

Figura 14 – Sistema de controle adaptável proposto pelo DRAC

Fonte: (PILLON; RICHARD, 2004)

de processos, e DRAC *deamon*, responsável pelo escalonamento e gerenciamento de processos e pela monitoração da memória. O *deamon* é um *loop* que possui uma função para cada um de seus módulos.

Os resultados obtidos mostram que o escalonamento DRAC tem melhor desempenho que o escalonamento aleatório para toda porcentagem de processos maior que 10%. O melhor escalonamento chega a 22% mais rápido que o escalonamento aleatório. O melhor desempenho obtido pelo escalonamento DRAC em relação ao escalonamento aleatório foi de 10%.

3.3 MH-TEDSim: Simulador de hierarquia de memória com simulação dirigida por execução ou por rastro

Silva et al. (2007), propõem e desenvolvem um simulador chamado MH-TEDSim, que é um simulador de hierarquia de memória que suporta simulações dirigidas por execução ou por trace memory.

Esse trabalho visa tornar o aprendizado mais didático e eficiente, permitindo ao usuário a simulação e a análise de problemas mais complexos e reais, através de simulações dirigidas por execução de códigos *Assembly*.

A Figura 15 mostra como é organizada a arquitetura do MH-TEDSim.

Observa-se que ele é dividido em três camadas independentes. A primeira é a camada de entrada, na qual o usuário fornece os dados (código *Assembly* por exemplo) e os parâmetros da hierarquia de memória a ser utilizada. Na segunda camada os dados são processados e o comportamento da hierarquia de memória é simulado. A terceira é a camada de saída, onde os resultados da simulação são apresentados ao usuário.

Os autores concluem que a utilização do simulador desenvolvido produz ganhos no aprendizado de hierarquia de memória e permite uma maior integração com outros

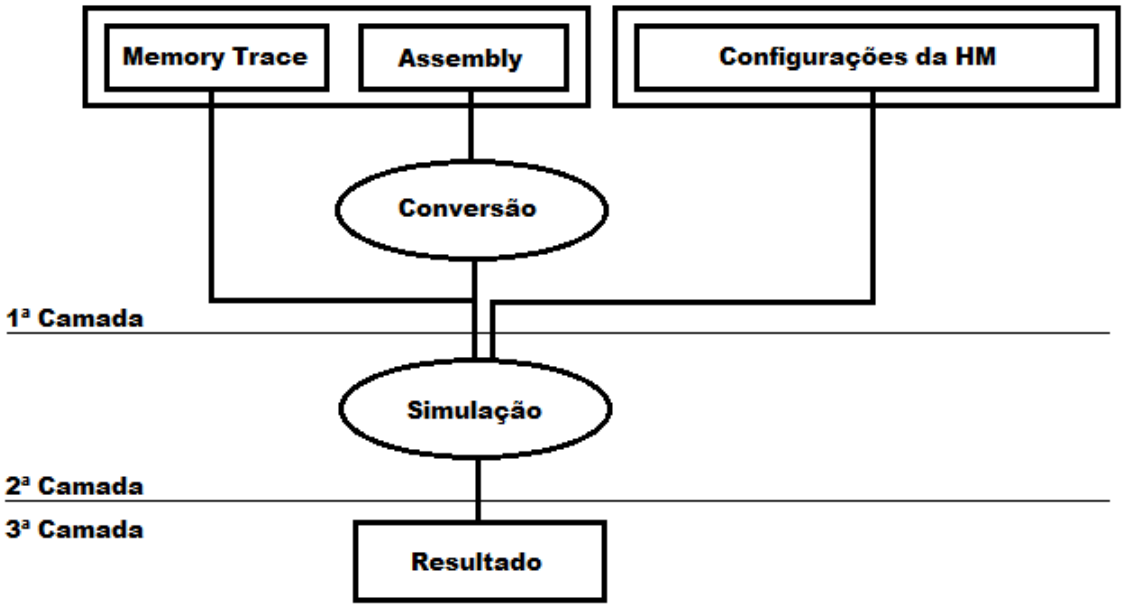

Figura 15 – Diagrama da arquitetura do MH-TEDSim

Fonte: Adaptado de (SILVA et al., 2007)

tópicos de arquitetura de computadores como o nível ISA (*Instruction Set Architecture*), programação em *Assembly* e avaliação de desempenho de sistemas computacionais.

3.4 Análise da hierarquia de memória em GPGPUs

Conrad (2010) desenvolveu um trabalho que teve como objetivo analisar o impacto de diferentes otimizações no acesso à memória da GPU, onde ele utiliza como caso de uso a arquitetura CUDA da empresa NVIDIA.

Essas otimizações são feitas a partir do uso de hierarquia de memória que, no seu trabalho recebeu uma gama de testes com carga de dados diferentes para fazer as devidas análises de memória compartilhada e global. Esses testes utilizaram como base uma aplicação de cópia coalescida de dados, ou seja, cópias onde foi feito apenas uma transação com a memória e todo o segmento foi recuperado, e transposição de matrizes.

Para a cópia direta de dados, percebeu-se uma ordem de magnitude de diferença entre a cópia direta coalescida e a cópia de dados com grandes espaçamentos entre seus elementos. Isto ocorre pois mesmo nas placas gráficas mais recentes, os acessos só podem ser coalescidos quando os dados ainda apresentam alguma localidade, isto é, quando se encontram em segmentos próximos. Devido à maior flexibilidade para coalescer os acessos nos modelos mais recentes da arquitetura, o impacto dos acessos desalinhados e com pequenos espaçamentos foi uma redução de aproximadamente 60% na taxa de transferência.

Através do exemplo de transposição de matrizes, foi possível estudar o impacto de acessos não coalescidos na transferência de dados, e como é possível utilizar a memória compartilhada para coalescer os acessos na memória global. Observou-se uma diferença de até 7x entre o acesso não coalescido e o acesso coalescido utilizando a memória global.

O autor conclui que as otimizações devem ser aplicadas em etapas, sendo primeiramente necessário otimizar o acesso à memória global, de modo a evitar acessos não coalescidos. Em seguida deve-se eliminar os conflitos no acesso à memória compartilhada. O autor também acredita que o trabalho pode ser estendido para outras arquiteturas como a ATI Stream.

3.5 On cache coerency and memory consistency inssues in NoC based shared memory multiprocessor SoC architecture

Pretot, Greiner e Gomez (2006) desenvolveram um trabalho que propõe um modelo de tratamento de coerência de cache puramente em software. Evidencia os principais problemas relacionados a coerência de cache.

Em seu modelo a memória é particionada em segmento local e segmento compartilhado, onde no segmento compartilhado estão guardadas as variáveis globais. Os autores destacam que um mapeamento de memória feito em software pode identificar a localização de dados locais e compartilhados. Foram executados testes utilizando como base a aplicação de um decodificador JPEG objetivando mensurar a vazão e o CPI de cada processador.

Os autores concluem que a utilização do software para mapear variáveis e a abordagem de segmentos locais e compartilhados consegue resolver de forma simples os problemas de coerência de cache e consistência de memória. Aplicado a um decodificador JPEG obteve de 50% a 100% de ganho em desempenho comparado a uma abordagem totalmente sem cache.

Todos esses trabalhos desenvolvem e analisam modelos de hierarquia de memória em arquiteturas multiprocessadas, o diferencial do trabalho descrito nesta dissertação é a aplicação e análise de um modelo de hierarquia de memória em uma arquitetura não-convencional com características que favorecem aplicações com forte princípio de localidade espacial e temporal.

4 APRESENTAÇÃO DOS MODELOS DE HIERARQUIA DE MEMÓRIA

Um dos principais aspectos no que diz respeito à memória em sistemas multiprocessados é a escolha do modelo a ser utilizado. Existem dois modelos principais: memória compartilhada e memória distribuída.

No modelo de memória compartilhada, os processadores disputam o acesso a um dispositivo de memória único no sistema. Neste modelo fica claro que a disputa pelo acesso à memória causa um grande gargalo no sistema (SILVA, 2009). No modelo de memória distribuída, é necessário que exista uma comunicação entre os processadores para que os dados na memória possam ser acessados por qualquer um dos processadores. No modelo com memória distribuída tem a vantagem de escalabilidade do sistema, enquanto no modelo com memória compartilhada existe a facilidade da programação.

No modelo de IPNoSys original os quatro módulos de memória são totalmente independentes e fatores como coerência de memória são de total responsabilidade do programador. No cenário do IPNoSys original, durante a execução de um programa algumas instruções como *LOAD* e *STORE* podem mudar o valor das variáveis replicadas nos módulos de memória, para evitar incoerência de dados essas instruções devem ser endereçadas aos módulos de memória específicos. Diante disso, deve ser considerada uma forma de garantir a coerência de dados nas memórias. Sendo que no modelo original da IPNoSys essa coerência depende, exclusivamente, do endereçamento correto das instruções de LOAD e STORE (instruções responsáveis pela modificação de dados), onde é especificado exatamente em qual módulo de memória essas instruções devem ser executadas. Existem duas técnicas principais muito utilizadas para garantir coerência de memórias em sistemas que apresentam módulos de memórias privados associados a um ou mais módulos de memória compartilhada: a técnica de *snoop* e a técnica de Diretório.

Na técnica *snoop* os módulos de memória devem verificar a todo instante as transações de dados que ocorrem no barramento, e na técnica de Diretório é implementado um módulo centralizador que contém todas as informações de utilização de dados e memórias.

O desenvolvimento desse trabalho levou em consideração a proposta de três modelos de hierarquia de memória diferentes. O primeiro modelo tem o objetivo de criar coerência de dados de forma transparente para o programador com o menor impacto possível na arquitetura e na organização da IPNoSys. O segundo modelo é uma evolução do primeiro, tem o objetivo de diminuir a quantidade de *hops* (latência) na rede. Já o terceiro modelo é uma evolução do segundo e tem o objetivo de otimizar a utilização da rede. As seções a seguir descrevem em detalhes os três modelos.

4.1 PRIMEIRO MODELO DE HIERARQUIA DE MEMÓRIA PROPOSTO

O primeiro modelo de hierarquia de memória foi implementado em C++ utilizando a biblioteca *System C*. Propõe a utilização do modelo de memória próprio que lembra o funcionamento de um modelo de memória compartilhada distribuída, porém as memórias caches é que estão fisicamente distribuídas e representam faixas de endereçamento da memória principal compartilhada. Diante disso, esse modelo é definido como compartilhado com caches distribuídas. Esse modelo híbrido faz com que a principal desvantagem do modelo de memória compartilhada, que é a geração de um gargalo no sistema ocasionado pela disputa pelo acesso à memória, seja amenizada com a utilização dos módulos de memória distribuída e ainda tira proveito da facilidade de programação do modelo de memória compartilhada.

Dessa forma toda estrutura existente foi preservada e, um nível abaixo na hierarquia de memória, foi adicionado um módulo de memória compartilhada, de modo que todos os pacotes são armazenados na memória principal (módulo de memória compartilhada) e à medida que vão sendo requisitados, são levados para as memórias caches (módulos de memória distribuída). Em caso de escrita na memória, os dados são guardados primeiramente na cache e logo em seguida são enviados para serem guardados na memória principal, isso caracteriza a política de escrita em memórias chamada *write-through*.

A escolha dessa política de escrita se deu pela sua eficiência com baixa complexidade de implementação e pelo fato dela adequar-se muito bem ao modelo de aplicações desenvolvidas para a arquitetura IPNoSys, uma vez que suas aplicações geralmente transmitem resultados parciais das instruções diretamente no mesmo pacote e apenas guardam na memória os resultados finais. No modelo proposto, é possível não especificar em qual dos quatro módulos de memória as instruções de busca de dados (LOAD e STORE) e as instruções de busca de instruções (EXEC e SYNEXEC) serão executadas. Nesse cenário proposto, estas instruções são executadas pela MAU mais próxima.

Para isso, deve-se garantir a coerência dos quatro módulos de memória cache (nível distribuído). Inicialmente, com pesquisas realizadas, pensou-se que a melhor técnica de coerência de memórias a ser aplicada seria a técnica de Diretório com um modo de implementação definida por (REGO, 2006) e modificada por (SILVA, 2009), já que a técnica de *snoop* não se aplica em arquiteturas baseadas em redes, uma vez que os módulos de memória ficarão impossibilitados de verificar toda a rede. Porém, verificou-se posteriormente que as memórias caches encontram-se em um nível fora da rede e podem comunicar-se por um canal único que liga todas elas (barramento), além disso a quantidade máxima de elementos ligados a este canal é 4 (4 módulos de memória). Desenvolvendo uma forma de comunicação mais rápida, uma vez que não é necessário utilizar a rede, não estando sujeita a oscilações de latência desta. Além

disso, com a utilização da política de escrita *write-through* favorece a utilização desse protocolo de coerência. A Figura 16 mostra o modelo de hierarquia de memória proposto.

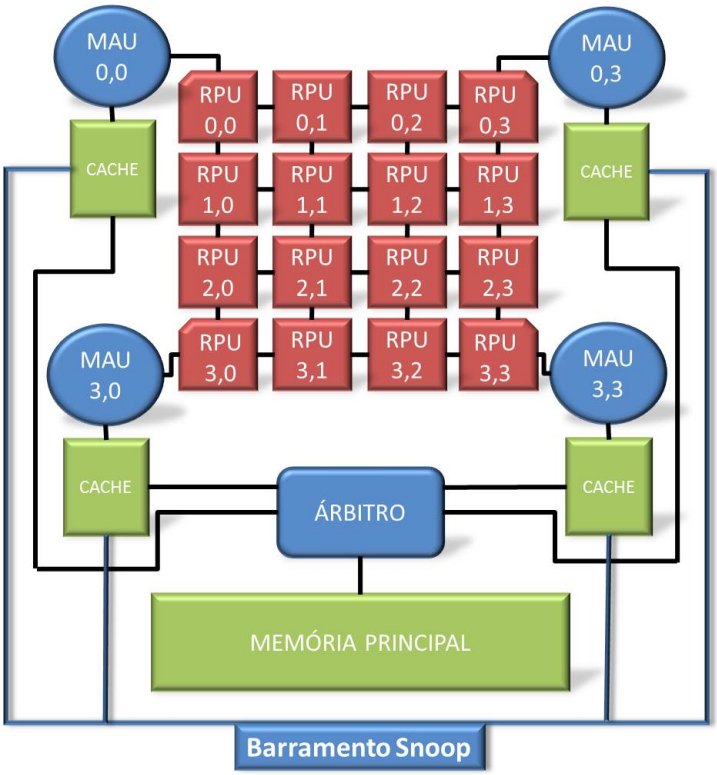

Figura 16 – Primeiro Modelo de hierarquia de memória proposto

Fonte: Próprio Autor

Observa-se que os quatro módulos de memória distribuída (memória cache) estão ligados a um árbitro e posteriormente ao módulo unificado de memória compartilhada (memória principal). Dessa forma, todos os dados estarão inicialmente na memória principal compartilhada, e a medida que forem requisitados serão copiados para as memórias caches distribuídas. O árbitro gerencia o acesso à memória principal, já que mais de uma cache pode requisitar a memória principal em um mesmo instante. A memória principal e as caches armazenam dados e instruções, os barramentos entre as caches, o árbitro e a memória principal possuem 36 *bits* de largura por causa do formato do pacote (32*bits* para a palavra + 4*bits* de controle para identificar o tipo de palavra). O controle da coerência das memórias cache é feito através do barramento *snoop*, onde cada cache verifica a cada ciclo se alguma outra cache está escrevendo em um dado compartilhado. Em caso positivo, a cache atualiza o dado modificado, em caso negativo, ela simplesmente ignora. Para ficar mais claro como a hierarquia de memória funciona, as Figuras 17, 19 e 20 apresentam o fluxograma de funcionamento da memória cache, do árbitro e da memória principal respectivamente.

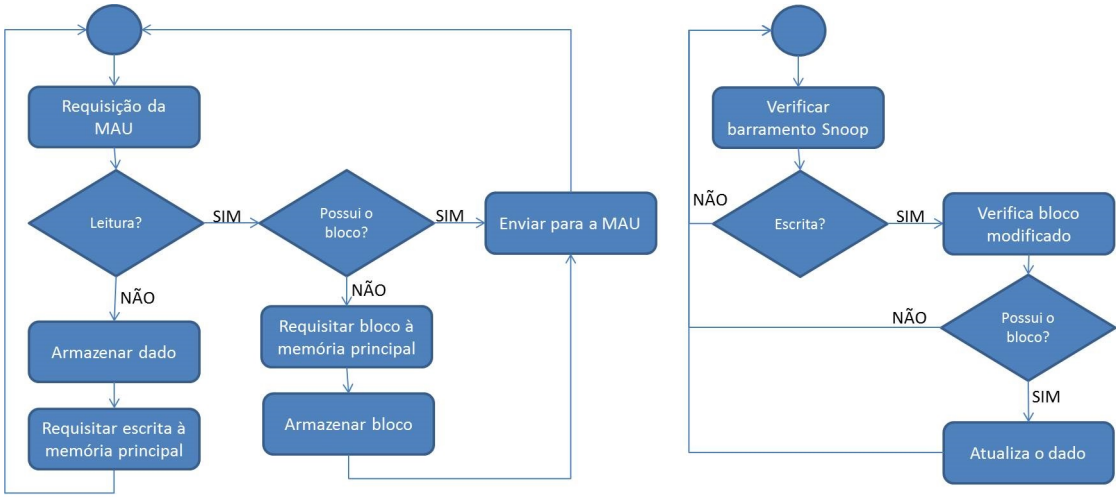

Figura 17 – Fluxograma do funcionamento da memória cache

Fonte: Próprio Autor

O funcionamento da memória cache se divide em duas etapas. A primeira, consiste em tratar requisições de leitura ou escrita feitas a partir da MAU, nessa etapa a cache verifica se há alguma requisição, caso seja uma leitura ela verifica se possui o dado e envia-o para a MAU. Se o dado não estiver armazenado, a cache envia uma requisição de leitura para a memória principal para que esta envie o bloco com o dado, o bloco é armazenado na cache e o dado repassado para a MAU. Caso a requisição não seja uma leitura, ela será uma escrita. Nessa situação a cache armazena o dado e envia-o para a memória principal para que ele também seja armazenado por ela. A segunda etapa do funcionamento da cache consiste em verificar o barramento *snoop* para manter a coerência, portanto ela verifica se alguma outra cache está fazendo escrita em algum dado: se ela possuir o dado que está sendo escrito, ele é atualizado, caso contrário, a cache ignora. A segunda etapa acontece a cada troca de estado da primeira, de modo que as duas ocorrem praticamente de forma paralela. A organização de estados da memória cache pode ser observada na Figura 18.

Observa-se que o funcionamento inicia no estado ”Verifica Escrita/Leitura”, ao receber um sinal de requisição de leitura passa para o estado ”Verifica Bloco” onde verifica se possui o bloco. Em caso positivo (*P.Bloco=True*), passa para o estado ”Entrega Dado” onde o dado é entregue à MAU para que possa ser injetado na rede retornando incondicionalmente para o estado inicial. Em caso negativo (*P.Bloco=False*), passa para o estado ”Requisita Dado” onde é feita a requisição da leitua à memória principal e incondicionalmente passa para o estado ”Aguarda Leitura” que vai aguardar a comunicação com a memória para recebimento do bloco. Quando a comunicação é estabelecida (READ.ready = True), passa para o estado ”Recebe Bloco” que recebe o bloco inteiro e passa incondicionalmente para o estado ”Entrega Dado” já mencionado

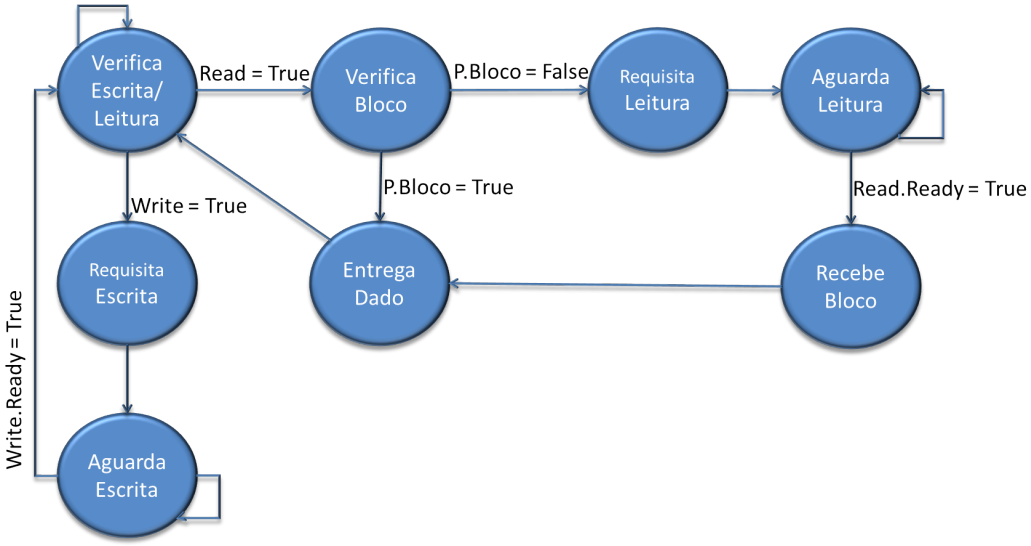

Figura 18 – Máquina de estados da Memória Cache

Fonte: Próprio Autor

anteriormente. Novamente, no estado "Verifica Escrita/Leitura" e recebendo o sinal de requisição de escrita, a cache passa para o estado "Requisita Escrita" e, após requisitar a escrita na memória principal, passa para o estado "Aguarda Escrita". Quando a escrita for realizada (Write.Ready=True), passa para o estado inicial.

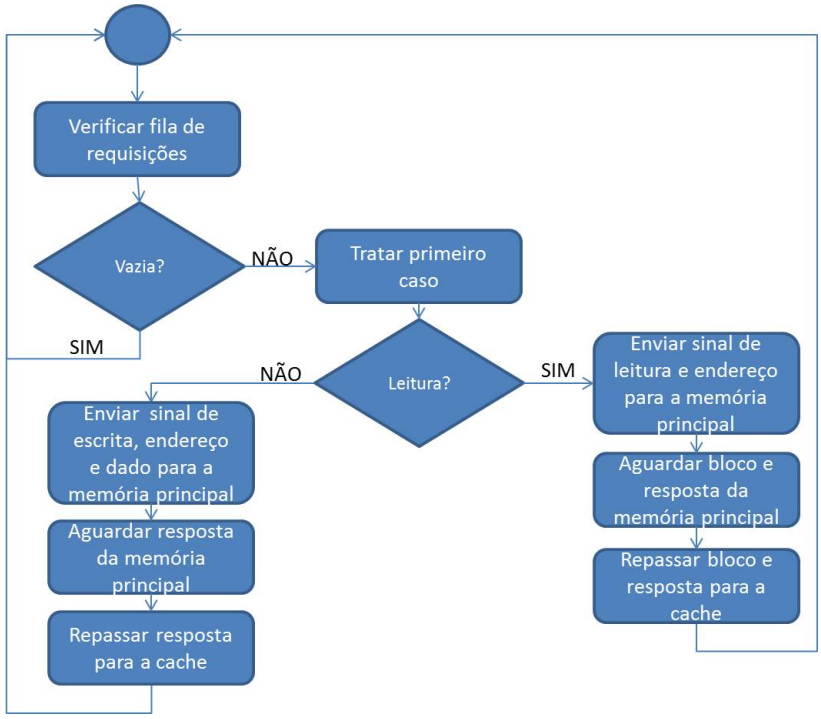

Figura 19 – Fluxograma do funcionamento do árbitro

Fonte: Próprio Autor

Como mencionado anteriormente e exibido na Figura 19, o árbitro funciona como um gerenciador de acesso à memória principal, uma vez que mais de uma cache podem acessá-la ao mesmo instante. Diante disso, seu funcionamento consiste em armazenar as requisições das caches em uma fila de requisições e tratar sempre a de maior prioridade (o primeiro elemento da fila). Em caso de leitura, o árbitro envia os sinais de leitura, assim como o endereço, que deve ser acessado para a memória principal, aguarda a resposta da memória que deve vir com o bloco onde o dado está, recebe e repassa o bloco para a cache e vai tratar a próxima requisição. Se for uma requisição de escrita, o árbitro passa os sinais de escrita, o endereço que deve ser modificado e o dado que vai ser armazenado na memória principal, aguarda e repassa a resposta da memória para a cache voltando novamente a verificar a fila de prioridades.

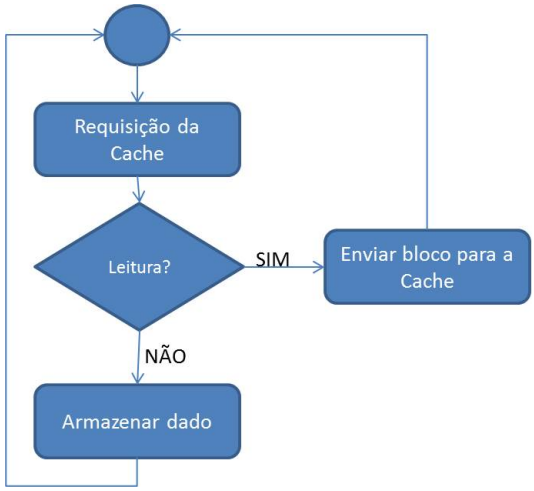

Figura 20 – Fluxograma do funcionamento da memória principal

Fonte: Próprio Autor

O funcionamento da memória principal apresentado na Figura 20, consiste apenas em verificar se há alguma requisição, caso seja uma leitura, ela envia o bloco inteiro para a cache. Caso seja uma escrita, ela armazena o dado e envia um sinal informando que foi realizado.

Com a coerência de memória cache garantida, foi possível implementar uma funcionalidade que facilita o modo de programar aplicações para a IPNoSys, essa funcionalidade permite que o programador tenha a possibilidade de não se preocupar com a identificação de em qual módulo de memória as instruções de busca de dados ou instruções serão executadas. Anteriormente, o programador deveria informar qual MAU deveria injetar qual pacote e também em qual MAU um dado deveria ser buscado, para isso, o programador utilizava as palavras reservadas ”MAU_0”, ”MAU_1”, ”MAU_2” e ”MAU_3”. Com a nova funcionalidade, simplesmente o programador pode utilizar a palavra reservada ”MAU_N”, indicando que a instrução pode ser executada pela MAU mais próxima. Vale ressaltar que mesmo com o desenvolvimento dessa funcionalidade o

programador ainda pode utilizar a forma de programar indicando onde cada instrução será executada, de modo que ambos os modelos de programação são compatíveis.

Com essa funcionalidade, as instruções LOAD, STORE, EXEC e SYNEXEC são sempre executadas pela MAU mais próxima. Entretanto, as instruções SYNC e SEND para serem executadas corretamente, devem ser endereçadas para a MAU que injetará o pacote, ou seja, a MAU que executará o EXEC ou o SYNEXEC. Essa dependência se dá pelo fato de que as instruções SYNC e SEND levam sinais ou dados para as instruções SYNEXEC e EXEC respectivamente, sendo que a MAU que executa a instrução EXEC deve executar também a instrução SEND associada a ela. Como as instruções podem ser executadas por qualquer MAU, e no momento da decodificação de um SYNC ou SEND não se sabe qual MAU executou o EXEC ou o SYNEXEC associado, essas instruções são enviadas em *multicast* para todas as MAUs, dessa forma a MAU que recebeu o EXEC ou o SYNEXEC também recebe o SYNC ou o SEND e as outras MAUs ignoram.

Além de facilitar a forma de programação, essa funcionalidade impacta diretamente no tempo de espera na rede (latência), uma vez que buscando na MAU mais próxima, o caminho percorrido através da rede é menor e consequentemente a ação é realizada em menos tempo. Para tornar mais clara a importância dessa funcionalidade, considere o seguinte exemplo: observando a Figura 16, suponhamos que a RPU 3,3 esteja requisitando um dado a MAU 0,0. Como o algoritmo de roteamento utilizado para pacotes de controle na IPNoSys é o XY tradicional, o caminho a ser feito passa pelas RPUs (2,3), (1,3), (0,3), (0,2), (0,1) e (0,0) totalizando 6 hops (salto de transmissão entre RPUs, sabendo que um *hop* leva em média 30ns para ser executado, o tempo de espera é de 180ns. Agora suponhamos que a RPU 3,2 requisite o dado a MAU 3,3, o caminho passaria apenas pela RPU 3,3 com 1 hop, de modo que o tempo de espera diminui para 30ns. Dessa forma, buscar na MAU mais próxima é, neste caso, claramente mais vantajoso.

Mesmo com a coerência dos dados garantida entre os módulos de memória cache, em aplicações onde há concorrência de dados, o programador ainda deve se responsabilizar por manter a consistência de dados. Na Arquitetura IPNoSys, ele pode utilizar instruções que já existem para fazer isso. A instrução SYNEXEC pode garantir que um pacote só será executado após um ou vários pacotes, que enviam sinais de sincronização através da instrução SYNC. Colocando instruções que devem manipular dados de forma exclusiva em pacotes diferentes e utilizando as instruções SYNEXEC e SYNC, o programador pode garantir a sequencialidade da execução dessas instruções, evitando a inconsistência dos dados manipulados. Por outro lado, pode-se ainda pensar no desenvolvimento de novas instruções que promovam o bloqueio e o desbloqueio de um determinado dado exclusivo. Uma instrução do tipo *LOCK*, que garante exclusividade do dado e, no IPNoSys, não geraria um novo pacote como acontece nas instruções SYNEXEC e SYNC.

A nova hierarquia de memória acrescenta um novo nível à pirâmide de hierarquia da IPNoSys original. A Figura 21 apresenta a nova organização. Observa-se que os demais níveis são mantidos, acrescentando-se o nível de memória cache distribuída.

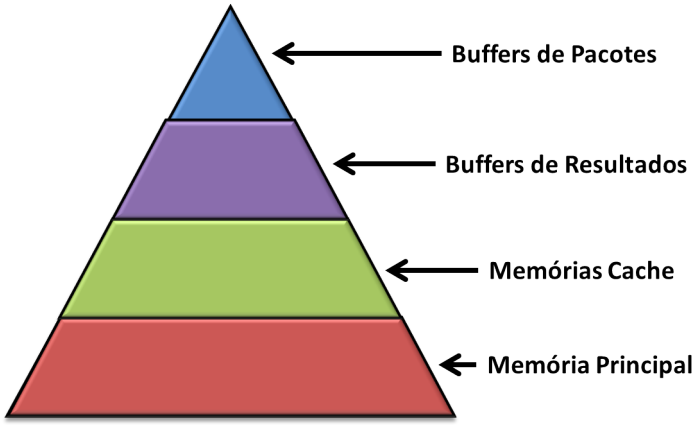

Figura 21 – Hierarquia de memória proposta

Fonte: Próprio Autor

Com esse primeiro modelo de hierarquia de memória, tornou-se possível o desenvolvimento de um mecanismo de manutenção de coerência das memórias. Esse fato implica diretamente em uma melhor forma de programar para IPNoSys, sem a necessidade de destacar a todo instante em qual MAU será executada uma determinada instrução. Além disso, essa possibilidade também impactou no desempenho da arquitetura, uma vez que é possível buscar dados ou instruções no módulo de memória mais próximo diminuindo a latência da rede. Com as informações sendo carregadas inicialmente em um único módulo de memória compartilhado (memória principal), afeta a forma de implementar esse sistema em hardware, já que é mais simples carregar tudo em um só lugar ao invés de carregar informações específicas para cada módulo de memória individualmente.

4.2 SEGUNDO MODELO DE HIERARQUIA PROPOSTO

A segunda proposta de organização de memórias, pode ser considerada a evolução mais natural do primeiro modelo. Vale ressaltar que este modelo não foi implementado, mas foi projetado com base nas pesquisas realizadas que permitem a descrição do seu funcionamento e a previsão dos resultados esperados. No primeiro modelo, entre as RPUs que compõem o quadrante de uma MAU (RPUs mais próximas da MAU), apenas uma está diretamente ligada à ela, ou seja, todas as outras RPUs tem que efetuar de 1 a 2 *hops* para se comunicar com a MAU por acesso. Como a intenção é diminuir o tempo de espera por informação, o segundo modelo consiste em construir

um canal de acesso direto entre a MAU e todas as RPUs do quadrante mais próximo a ela. A Figura 22, apresenta a estrutura do segundo modelo.

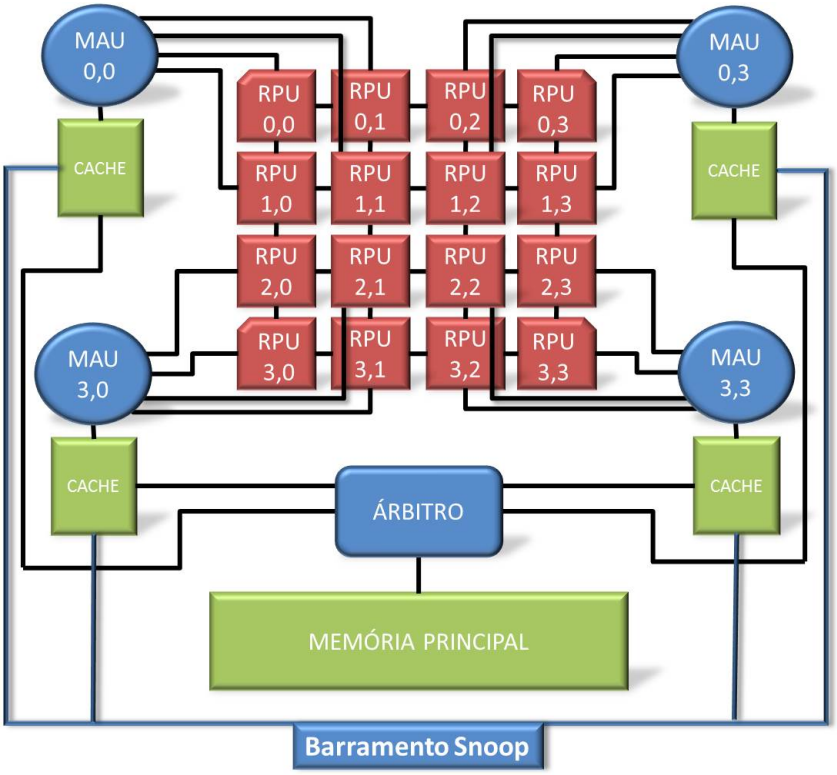

Figura 22 – Estrutura do segundo modelo de hierarquia de memória

Fonte: Próprio Autor

Observa-se que existem 4 interfaces de comunicação entre a MAU e as 4 RPUs que formam o seu quadrante. O restante da estrutura é semelhante ao primeiro modelo, contendo 4 caches associadas às MAUs e ligadas à memória principal por intermédio de um árbitro que gerencia os acessos. Para adicionar as novas interfaces de comunicação entre MAU e RPUs, algumas modificações estruturais devem ser feitas. Essas modificações são apresentadas pela Figura 23.

Como na MAU original, a MAU modificada apresenta um gerenciador de pacotes, um gerenciador de memória e uma unidade de controle. As mudanças ocorrem no gerenciador de pacotes, que agora pode injetar pacotes a partir de qualquer uma das 4 RPUs associadas a ele. Além disso, na interface de entrada, existe uma fila de requisições (onde cada requisição corresponde a um acesso à memória) para cada RPU, de modo que se faz necessário a utilização de um gerenciador de interfaces que implementa o algoritmo *round-robin*. Este, por sua vez, faz o escalonamento entre essas filas e passa informações para o gerenciador de pacotes com a finalidade de informá-lo para qual RPU deve ser encaminhado um determinado pacote.

Outra modificação deve ser feita no algoritmo de roteamente utilizado. Da forma

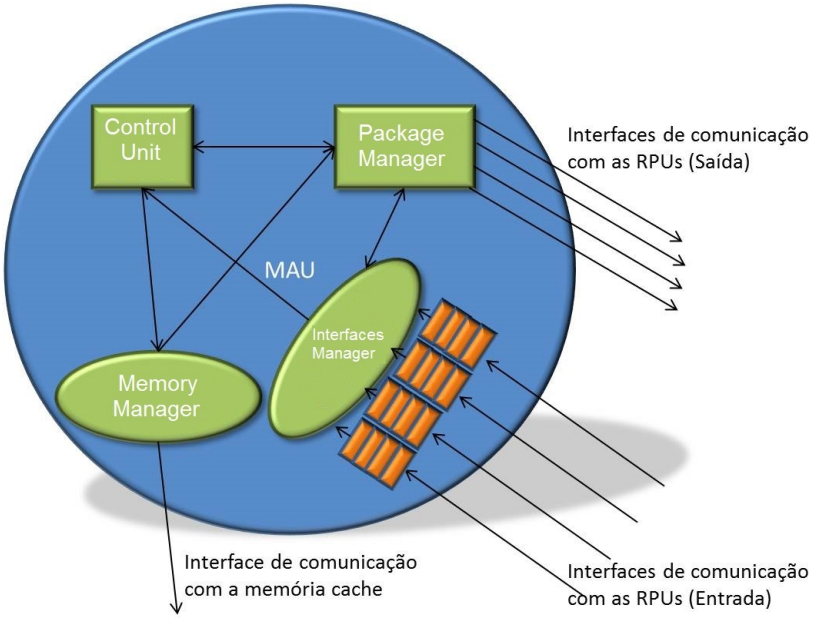

Figura 23 – Estrutura interna da MAU com 4 interfaces de comunicação com as RPUs

Fonte: Próprio Autor

como acontece no IPNoSys original e no primeiro modelo os pacotes são roteados a partir das RPUs dos cantos da rede e nunca serão roteador a partir de uma RPU do meio (ex.: RPU(1,0) e RPU(1,1)), se um pacote for injetado a partir dessas RPUs, o cálculo do destino que é feito não vai construir um caminho espiral proposto pelo algoritmo subutilizando a rede, sobrecarregando algumas RPUs. A Figura 24 apresenta o algoritmo de roteamento utilizado.

- Se início
  - Complemento do nó corrente
- Senão
  - Se o nó corrente estiver em um dos 4 cantos
    - Se o nó que enviou o pacote for o complemento do nó corrente
      - Calcula o complemento do nó corrente
      - Incremento ou decremento de X ou Y do complemento (alternado)
    - Senão
      - Complemento do nó corrente
  - Senão
    - Endereço do nó que enviou com incremento/decremento de X ou Y (alternado)

Figura 24 – Algoritmo para calcular novo endereço no Spiral Complement

Fonte: (ARAÚJO, 2012)

Para o cálculo de um novo destino, o algoritmo *spiral complement* leva em conta duas informações: o destino atual do pacote (nó corrente que calcula o novo destino); e a origem atual (nó que enviou o pacote ao nó corrente). Considerando o destino atual, o algoritmo verifica se este está em um dos quatro cantos da rede Se não estiver, o novo destino é calculado como sendo a origem atual com o incremento ou decremento da coordenada X ou coordenada Y em uma unidade. A operação de incremento ou decremento de uma das coordenadas, geralmente, é alternada e a coordenada afetada também alterna em cada novo cálculo. Se o destino atual (nó que calcula o novo destino) estiver em um dos cantos da rede, o algoritmo considera ainda a origem atual (nó que o enviou o pacote). Se a origem atual for um dos quatro cantos da rede, o novo destino é calculado em dois passos: no primeiro, é calculado o complemento do nó corrente e no segundo, acontece o incremento ou decremento, de uma unidade, de umas das duas coordenadas, X ou Y. Entretanto, se a origem atual não for um dos quatro cantos da rede o novo destino será calculado em um único passo, o complemento do nó corrente. A Figura 25 mostra como seria o roteamento dos pacotes feitos para o quadrante da MAU (0,0).

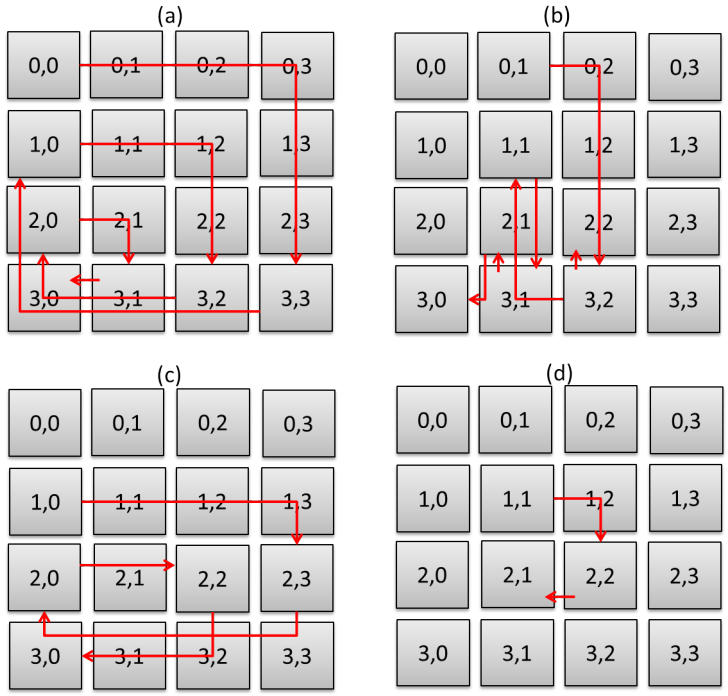

Figura 25 – Comportamento do Roteamento Spiral Complement sem modificação

Fonte: Próprio Autor

Caso (a): Pacote injetado pela RPU(0,0).
Caso (b): Pacote injetado pela RPU(0,1).
Caso (c): Pacote injetado pela RPU(1,0).
Caso (d): Pacote injetado pela RPU (1,1).

Observa-se que para os casos (b) e (d) o roteamento não forma a espiral. Ainda considerando o caso (d), ao chegar na RPU 2,1, o algoritmo calcula que o próximo endereço continua sendo a RPU 2,1 sobrecarregando essa RPU. Como esse não é o comportamento esperado para o roteamento Spiral Complement, uma modificação com o menor impacto possível sobre as RPUs foi adotada. Considera-se que quando um pacote for injetado a partir de qualquer RPU o endereço de origem será o endereço da MAU associada àquele quadrante. Desse modo, as RPUs (0,0), (0,1), (1,0) e (1,1), teriam o mesmo endereço (0,0) no momento da injeção do pacote, no decorrer do roteamento os endereços são calculados normalmento seguinto o *spiral complement*. O mesmo aconteceria para os demais quadrantes. A Figura 26 exibe como se comportaria o novo roteamento.

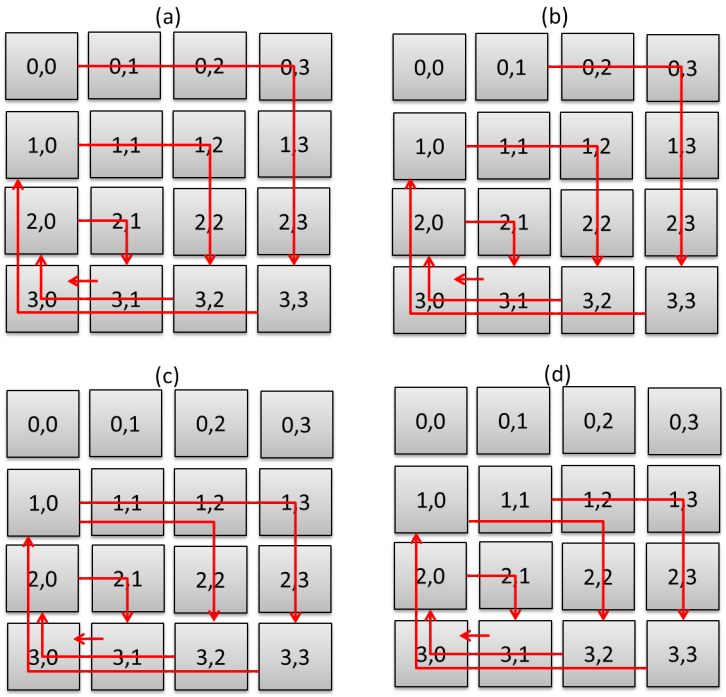

Figura 26 – Comportamento do Roteamento Spiral Complement modificado

Fonte: Próprio Autor

Caso (a): Pacote injetado pela RPU(0,0).

Caso (b): Pacote injetado pela RPU(0,1).

Caso (c): Pacote injetado pela RPU(1,0).

Caso (d): Pacote injetado pela RPU (1,1).

Observa-se que o roteamento de pacotes a partir de qualquer RPU do mesmo quadrante, tende a seguir o mesmo caminho, apenas as RPUs (1,0) e (1,1) inicialmente seguem um caminho diferente. Esse comportamento apresenta um modelo semelhante ao algoritmo Spiral Complement, e ainda possibilita a utilização de caminhos não explorados na IPNoSys original (RPU(1,2)->RPU(1,3), RPU(2,1)->RPU(2,0), RPU(1,1)-

>RPU(1,0), RPU(2,2)->RPU(2,3)), desafogando e melhor utilizando a rede.

O funcionamento de todas as instruções do IPNoSys, funciona da mesma forma que no primeiro modelo. A principal vantagem do segundo modelo é que no momento em que uma requisição (leitura ou escrita de dados, bem como injeção de pacotes) chegar no quadrante da MAU, a primeira RPU que recebê-la pode encaminhar diretamente para MAU.

Para entender melhor o funcionamento do segundo modelo, consideremos o caso em que uma instrução LOAD é decodificada pela RPU (0,1), essa RPU compõe o quadrante da MAU (0,0). Ao decodificar o LOAD, a RPU (0,1) encaminha o pacote de controle com o LOAD que será enviado diretamente para a MAU, em vez de passar pela RPU (0,0), como acontecia no primeiro modelo. O mesmo acontece para todas as outras instruções que necessitam acessar a MAU.

Nesse modelo, é possível injetar um pacote por cada RPU do quadrante, nesse cenário, totaliza-se 4 pacotes (4 fluxos de execução), sendo isso uma vantagem sobre o primeiro modelo. Além disso, o tempo de espera por informações diminui para o quadrante, já que todas as RPUs estão diretamente conectadas a MAU. O *overhead* gasto com as modificações feitas na MAU é praticamente insignificante, quando consideramos uma rede 4x4, mas deve ser levado em consideração quando essa rede aumentar, uma vez que a quantidade de RPUs associadas à uma MAU aumenta o tempo para tratar as requisições de todas elas pode ser muito alto, gerando um gargalo no desempenho da arquitetura.

Entretanto, como as 4 RPUs podem requisitar a memória cache no mesmo instante (sendo essas requisições gerenciadas por um gerenciador de requisições), o tempo de acesso à cache depende da quantidade de RPUs requisitando em um mesmo instante e da prioridade de requisição (quem requisitou primeiro). Ainda por causa dessa serialização de acessos à MAU, a injeção de pacotes na rede (que nesse modelo pode acontecer a partir de qualquer RPU) não pode ser feita simultaneamente em todas as RPUs. Na Seção 5.2 do Capítulo 5, serão apresentados com mais detalhes a forma como calcular o tempo de acesso à cache a partir de qualquer RPU do quadrante, e o impacto desse modelo sobre o desempenho da arquitetura.

4.3 TERCEIRO MODELO DE HIERARQUIA PROPOSTO

Outra proposta de modelo de hierarquia de memória para IPNoSys, consiste na implementação de um módulo de memória cache L1 privado para dados e pacotes associados a cada RPU do sistema. O restante da arquitetura é idêntica ao segundo modelo proposto, com as MAUs ligadas diretamente a todas as RPUs de seus quadrantes, mantendo quatro módulos de memória cache distribuídas nos cantos da rede (neste terceiro modelo esse nível de cache é considerado o nível L2) associados a uma memória principal através de um árbitro, como exibido na Figura 27.

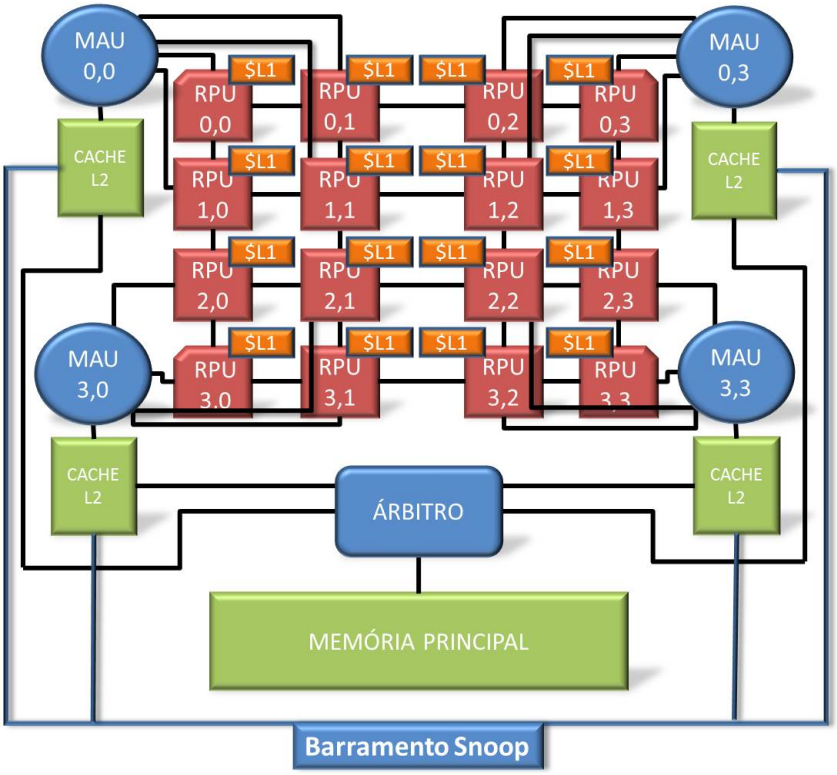

Figura 27 – Terceiro Modelo de hierarquia de memória proposto

Fonte: Próprio Autor

Para isso, os módulos de memória cache privados, ou módulos de memória cache L1(identificadas na Figura 27 por $L1), são responsáveis pelo armazenamento de dados e instruções (pacotes). Considerando o quadrante formado pelas RPUs mais próximas a uma determinada MAU, pode-se dizer que os módulos de cache L1 dessas RPUs representam faixas de endereços da cache L2 associada a esta MAU. As caches L1 podem ser descritas como conjuntos da memória cache L2, onde para cada conjunto é mapeada uma faixa de endereços. Para entender melhor observe a Figura 27. Considere a MAU 0-0, o quadrante dela seria as RPUs (0,0), (0,1), (1,0) e (1,1), e as caches L1 dessas RPUs armazenariam faixas diferentes de endereços referentes à cache L2 vinculada a MAU 0-0, de modo que quando um LOAD fosse decodificado e endereçado à MAU 0-0, a RPU que requisita saberia para qual RPU do quadrante enviar a instrução requisitando uma determinada informação (dado ou pacote). O cálculo para saber qual RPU pode ter a informação requisitada armazenada em cache pode ser feito de acordo com a Equação 4.1:

$$RPUaddress = (DATAaddress \pmod{QtdRPUsporQuadrante}) + Deslocamento \quad (4.1)$$

O endereço da RPU é o resto da divisão do endereço da informação pela quantidade de RPUs por quadrante somando o deslocamento que indica o endereço da

MAU, e consequentemente em que quadrante está a RPU.

No primeiro e no segundo modelo de hierarquia de memória, assim como no IPNoSys original, as RPUs têm acesso somente ao número do pacote. As MAUs possuem tabelas que mapeiam os pacotes identificados pelos números para os endereços onde estão armazenados na memória. Para que os pacotes possam ser endereçados para as caches L1 da mesma forma que os dados, é necessário a replicação dessa tabela nas RPUs (para que elas também possam identificar em qual endereço de memória o pacote está armazenado). Para gerenciar o tratamento dos pacotes nas RPUs, é necessário algumas funções extras à Unidade de Sincronização (SU) das RPUs, dando a este módulo a função de executar as instruções SEND, EXEC, SYNC e SYNEXEC, adicionando também toda a estrutura necessária para isso. Com isso, as RPUs conseguem injetar pacotes na rede a partir de suas caches L1, não sendo mais um trabalho exclusivo das MAUs. A Figura 28, mostra a estrutura interna acrescentada à SU.

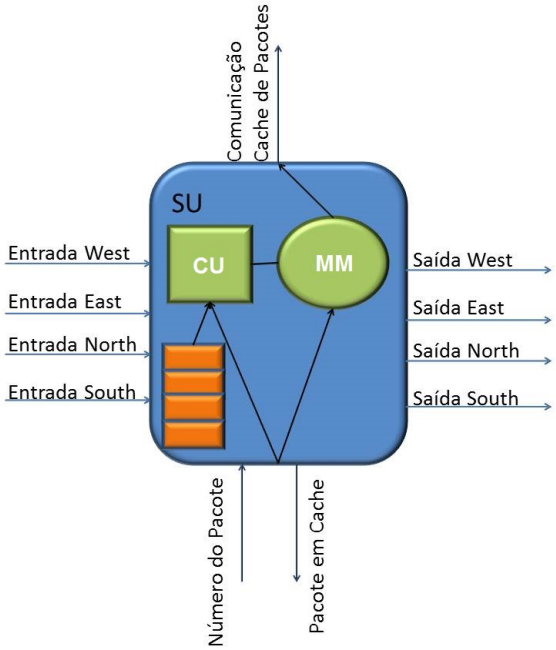

Figura 28 – SU modificada

Fonte: Próprio Autor

A SU possui conexão com as entradas e saídas *west*, *east*, *north* e *south* da RPU, isso é necessário para que ela possa ter controle sobre todos os pacotes que chegam na RPU e também possa encaminhar pacotes através delas. Os dois componentes básicos acrescentados à SU são a Control Unity (CU) e o Memory Manager (MM). O primeiro é responsável pelo controle de todas as entradas da SU, administra as quatro entradas verificando se o pacote que acabou de chegar deve ou não ser armazenado na cache de pacotes. Além disso, a CU é encarregada de executar as instruções de acesso aos pacotes e gerenciar o buffer de resultados. O MM tem a função de gerenciar o acesso à cache L1.

Cabe a ele converter o número de pacote para o endereço de armazenamento e verificar se ele está ou não na cache, permitindo tanto a leitura quanto a escrita de novos pacotes.

A principal diferença entre a SU e a MAU é que a primeira, além de tratar instruções de acesso à memória, tem ainda a função de verificar se o pacote que chega à RPU deve, ou não, ser armazenado na cache L1. A Figura 29, mostra o funcionamento da SU para tratamento de pacotes. A Figura 30 e a Figura 31, mostram o funcionamento para tratar instruções de manipulação de dados (LOAD e STORE).

O funcionamento da SU consiste em tratar as requisições feitas pela RPU, essas requisições são originadas pela chegada de instruções (EXEC, SYNEXEC, SYNC e SEND) que são encaminhadas à RPU, pois sua cache L1 pode já possuir um determinado pacote que deverá ser injetado. Diante disso, a RPU requisita a SU para executar a instrução e, caso necessário, injetar o pacote na rede.

A instrução SEND pode inserir valores em outros pacotes. Para tratá-la, a SU tem um buffer de resultados onde esses valores são armazenados e inseridos nos pacotes correspondentes no momento em que este é injetado na rede.

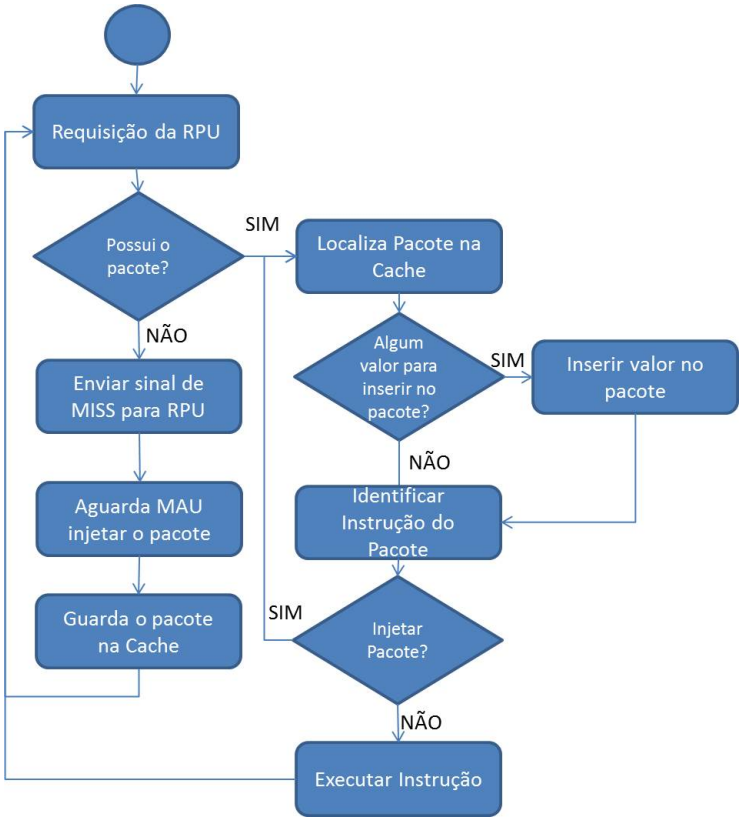

Figura 29 – Fluxograma do funcionamento da SU no tratamento de pacotes

Fonte: Próprio Autor

Para tornar mais claro o funcionamento do segundo modelo de hierarquia de memória proposto, a seguir é explicado o comportamento da arquitetura para todas as instruções que manipulam dados ou pacotes:

- LOAD: No momento em que uma RPU (RPU de origem) decodifica a instrução LOAD na rede, ela verifica qual o endereço do dado, calcula o endereço da RPU que pode ter o dado em sua cache L1 e gera um pacote de controle para essa RPU (RPU de destino). A RPU de destino requisita o dado a sua cache L1, que pode responder com o dado ou com um sinal de *miss*. Caso ela tenha o dado, a RPU gera um novo pacote de controle, agora com a instrução RELOAD contendo o dado, e encaminha para a RPU de origem. A RPU de origem recebe o RELOAD com o dado e segue a sua execução. Em caso de *miss*, a RPU de destino encaminha o LOAD para a MAU que tem o dado, mas modifica o campo de origem para o seu endereço (dessa forma ela vai receber a resposta do LOAD com o dado e pode armazená-lo em sua cache). Para conseguir enviar o dado de volta à RPU de origem que o requisitou, ela armazena o endereço dessa RPU de origem. A MAU trata o LOAD e responde com um pacote de controle endereçado para a RPU de destino, essa por sua vez guarda o dado em cache e encaminha o pacote com o RELOAD para a RPU de origem que requisitou o dado inicialmente. A Figura 30, exibe o algoritmo seguido pela RPU quando decodifica uma instrução LOAD.

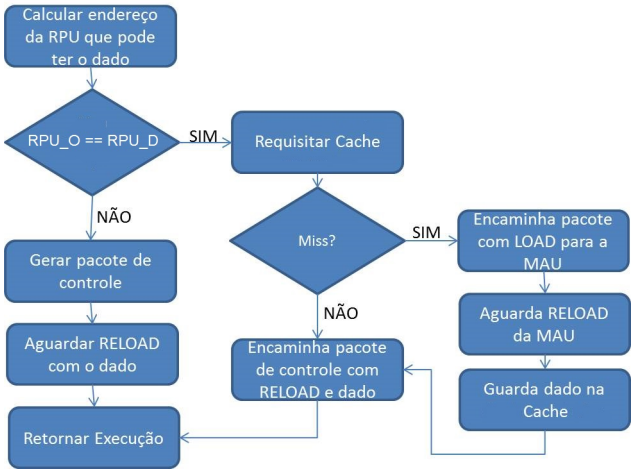

Figura 30 – Fluxograma do comportamento da RPU na decodificação de uma instrução LOAD

Fonte: Próprio Autor

- STORE: Ao decodificar uma instrução STORE, a RPU (RPU de origem) verifica o endereço onde o dado deve ser salvo, calcula o endereço da RPU que possui a cache L1 onde o dado deve ser salvo e gera um pacote de controle com o STORE encaminhando para essa RPU (RPU de destino). A RPU de destino recebe o pacote e salva o dado na sua cache. O dado só será atualizado/armazenado na cache L2, e consequentemente na memória principal quando houver uma substituição de bloco, esse comportamento caracteriza uma política de escrita *write-back*. Nesse caso, essa política foi escolhida para diminuir a transmissão de dados através da

rede. A Figura 31, exibe o algoritmo seguido pela RPU quando decodifica uma instrução STORE.

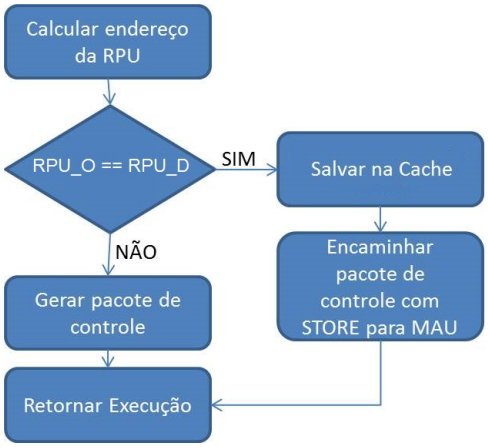

Figura 31 – Fluxograma do comportamento da RPU na decodificação de uma instrução STORE

Fonte: Próprio Autor

A Figura 32 mostra o algoritmo seguido pelas RPUs no caso de decodificação das instruções EXEC, SEND, SYNC e SYNEXEC descritas a seguir.

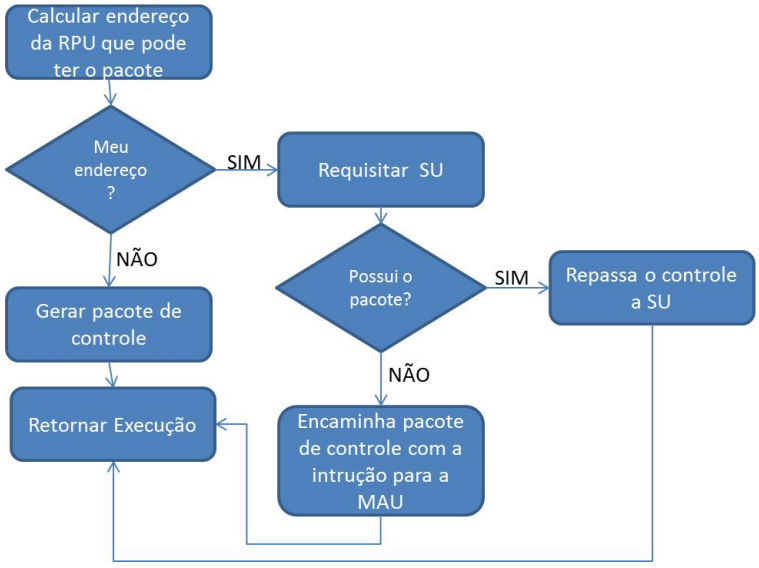

Figura 32 – Fluxograma do comportamento da RPU na decodificação dasinstruções EXEC, SEND, SYNC e SYNEXEC

Fonte: Próprio Autor

- EXEC: A RPU (RPU de origem) que decodifica uma instrução EXEC verifica o número do pacote, calcula o endereço da RPU que pode ter esse pacote em sua cache de pacotes e gera um pacote de controle com o EXEC para essa RPU (RPU

de destino). A RPU de destino requisita o pacote à SU, caso o pacote esteja na cache, a SU injeta-o na rede (coloca o pacote no buffer de saída e calcula o destino baseado no algoritmo de roteamento espiral). Caso o pacote não esteja na cache, a RPU de destino encaminha o pacote com o EXEC para a MAU que possui o pacote, esta por sua vez trata a instrução EXEC injetando o pacote na rede através da RPU de destino. No momento em que o pacote chega na RPU de destino, uma cópia inteira será armazenada na cache L1, de modo que quando houver uma nova requisição da injeção desse pacote ele já será injetado pela própria SU da RPU.

- SEND: A instrução SEND contém valores de resultados parciais que são inseridos nos operandos de instruções em pacotes que serão injetados na rede através de uma instrução EXEC, de modo que esses resultados parciais são armazenados em um buffer de resultados até que o pacote indicado pela EXEC chegue para ser injetado. No modelo proposto, quando uma RPU (RPU de origem) decodifica uma instrução de SEND, ela verifica o número do pacote onde o resultado parcial deve ser inserido, calcula o endereço da RPU que pode ter esse pacote e gera um pacote de controle com o SEND para essa RPU (RPU de destino). Na RPU de destino a SU verifica se tem o pacote onde o resultado deve ser inserido, em caso positivo, ela coloca o resultado no seu buffer de resultados e aguarda a chegada de uma instrução EXEC para injetar o pacote na rede. Em caso negativo, a RPU de destino encaminha o pacote com o SEND para a MAU que tem o pacote onde o resultado deve ser inserido.

- SYNEXEC: Quando uma RPU (RPU de origem) na rede decodifica uma instrução SYNEXEC, ela verifica o número do pacote, calcula o endereço da RPU que pode ter o pacote em sua cache e gera um pacote de controle com o SYNEXEC para essa RPU (RPU de destino). A RPU de destino requisita o pacote à SU, caso o pacote esteja na cache, a SU vai aguardar a chegada das instruções SYNC e para poder injetar o pacote na rede. Caso não esteja na cache, o pacote de controle com o SYNEXEC é encaminha para a MAU que tem o pacote. Quando o pacote foi injetado na rede pela MAU e através da RPU de destino, uma cópia será armazenada na cache L1.

- SYNC: Essa instrução envia sinais de sincronismo, para que a MAU ou a SU saiba o momento exato de injetar na rede um pacote determinado pela instrução SYNEXEC. Quando uma RPU na rede (RPU de origem) decodifica a instrução SYNC, ela verifica o número do pacote que deve receber esse sinal de sincronização, calcula o endereço da RPU que pode ter o pacote em sua cache de pacotes e gera um pacote de controle com o SYNC pra essa RPU (RPU de destino). A RPU de destino recebe o pacote de controle com o SYNC e a SU verifica se o pacote que deve receber o sinal está na cache, em caso positivo, o campo de sincronismo do

pacote indicado pelo SYNEXEC é decrementada e a SU injetará o pacote na rede no momento em que todos os sinais de sincronismos tiverem chegado. Caso o pacote não esteja na cache de pacotes, a RPU encaminha o pacote de controle com o SYNC para a MAU que possui o pacote, para que ela possa receber todos os sinais de sincronismo e injetar o pacote na rede.

Com base no funcionamento esperado desse modelo de hierarquia, gerou-se a Equação 4.2 que permite calcular o tempo de espera por um dado:

$$TempoEsp = 2 * (QtdHops * Th) + Tc \tag{4.2}$$

A variável *QtdHops*, representa a quantidade de *hops* entre a RPU de origem e a RPU de destino. A constante *Th*, é o tempo em nanosegundos gasto para realizar um *hop*(nesse cenário 30ns). A constante *Tc*, é o tempo em nanosegundos gasto para a cache L1 entregar o dado (nesse cenário 20ns). Essa equação leva em consideração que o dado sempre estará em cache (*hit*) na cache L1, no caso de *miss* a equação que deve ser utilizada para calcular o tempo de espera é a Equação 4.3:

$$TempoEsp = (2 * (QtdHops * Th) + Tc) + (Tm + To) \tag{4.3}$$

A primeira parcela da equação é idêntica a Equação 4.2, já a segunda parcela considera o tempo de acesso à memória a partir da MAU (Acesso à cache L2 e/ou à memória principal), onde a constante *To* refere-se ao tempo em nanosegundos gasto na comunicação entre a MAU e a RPU que está diretamente conectada a MAU (27ns), e a variável *Tm* refere-se ao tempo em nanosegundos que a memória cache L2 gasta para devolver o dado. Para calcular o tempo *Tm* deve utilizar a Equação 4.4 (para o caso de hit na cache L2) e Equação 4.5 (para caso de miss na cache L2).

$$Tm = TcL2 \tag{4.4}$$

$$Tm = TcL2 + Tmp \tag{4.5}$$

Na Equação 4.4, o tempo de acesso à memória é igual ao tempo de acesso à memória cache L2($TcL2$ = 20ns). Enquanto na Equação 4.5, o tempo de acesso à memória é igual ao tempo de acesso à memória cache L2 ($TcL2$ = 20ns) mais o tempo de acesso à memória principal (Tmp = 80ns).

Outro fator analisado neste modelo de hierarquia de memória é o tempo gasto para injeção de um pacote. Para realizar esse cálculo gerou-se a Equação 4.6:

$$Tempo_i = (QtdHops * Th) + Tmccp + Tc \tag{4.6}$$

A variável *QtdHops*, refere-se à quantidade de *hops* entre a RPU de origem e a RPU de destino, a constante *Th*, representa o tempo necessário para realizar um *hop*, a

constante *Tmccp*, representa o tempo em nanosegundos gasto pela SU para requisitar e injetar o pacote (nesse cenário 20ns), e a constante *Tc*, é o tempo em nanosegundos gasto pela cache de pacotes para devolver o pacote a SU (20ns). Essa equação considera que o pacote sempre estará na cache de pacotes (*hit*) na cache L1. Para calcular o tempo de injeção de pacotes em caso de *miss*, a equação a ser considerada é a Equação 4.7:

$$Tempo_i = ((QtdHops * Th) + Tmccp + Tc) + (Tm + To) \tag{4.7}$$

A primeira parcela é idêntica a Equação 4.6. A segunda, representa o tempo gasto para requisitar a MAU que possui o pacote, onde a constante *To* é o tempo em nanosegundos de comunicação entre MAU e RPU (27ns), e a variável *Tm* é o tempo em nanosegundos que a memória cache L2 gasta para devolver o dado à MAU e para que possa ser injetado na rede. Para calcular o tempo *Tm*, deve-se utilizar a Equação 4.4 (para o caso de hit na cache L2) e Equação 4.5 (para caso de miss na cache L2) apresentadas anteriormente.

A hierarquia de memória proposta no terceiro modelo, acrescenta mais um novo nível de memória à hierarquia original. A Figura 33, mostra a nova organização.

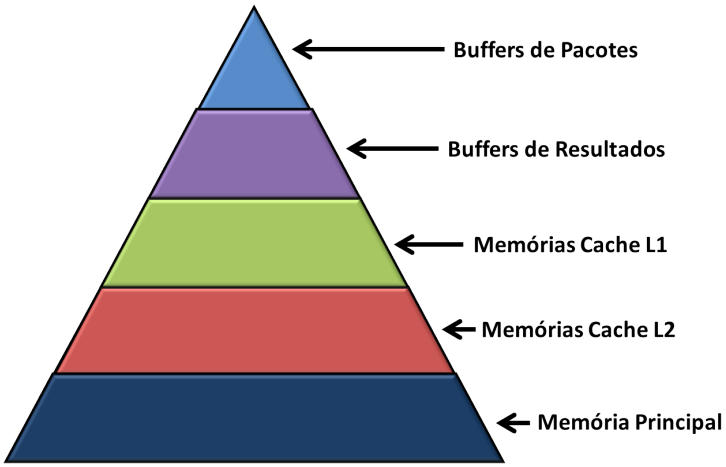

Figura 33 – Hierarquia de Memória - 3º modelo

Fonte: Próprio Autor

Nesse modelo de hierarquia proposto, dados e instruções(pacotes) são armazenados nas caches L1 conforme vão sendo requisitados. Como os pacotes e dados podem estar disponíveis dentro da rede, ocorre uma diminuição do tempo de espera, tanto para injetar pacotes na rede, quanto para receber um dado requisitado. Além disso, como cada RPU pode injetar um pacote a partir de sua cache L1, pode-se gerar um fluxo de execução por RPU (nesse cenário chega-se a um total de 4 fluxos por quadrante e até 16 fluxos na rede simultaneamente). Dessa forma, é possível aumentar o paralelismo na arquitetura IPNoSys.

Assim como no segundo modelo proposto, para que o algoritmo de roteamento

funcione adequadamente quando um pacote for injetado por uma RPU que não está no canto da rede é necessário que o endereço de origem seja o endereço da MAU associada ao quadrante desta RPU.

5 RESULTADOS

Neste capítulo são comparados o desempenho de aplicações na arquitetura IPNoSys original e na nova arquitetura IPNoSys com os modelos de hierarquia de memória propostos. Para tal, assumiu-se o cenário da IPNoSys com uma rede 4x4 (16 RPUs). As comparações levam em consideração o tempo de acesso à memória como medida de desempenho. Também é analisado o tempo de espera por informações ou latência da rede.

Os resultados de cada modelo são apresentados a seguir divididos em seções.

5.1 Resultados do primeiro modelo

Para a análise dos resultados do primeiro modelo foi usado o tempo de acesso à memória gasto por cada aplicação como medida de desempenho. Também é analisado o tempo de espera por uma informação ou latência da rede, para verificar a eficiência da funcionalidade que permite buscar na MAU mais próxima, além de confrontar o tempo de execução de aplicações que utilizam essa funcionalidade com o tempo de execução de aplicações que não a utilizam.

Para obter um valor de tempo de acesso à memória estimado, baseado em um sistema real, utilizou-se a ferramenta Cacti (JOUPPI; ALKHATIB, 1996). Para os testes a seguir, considerou-se memória principal de 32K fabricadas com tecnologia DRAM e memória cache de 4K com tecnologia SRAM ambas com 90nm. Em todos os testes, considerou-se a utilização de caches com modelo de mapeamento associativo por conjunto de 4 vias.

A Tabela 2, apresenta os resultados obtidos considerando a quantidade de 4 palavras por bloco (4 P/B), 8 palavras por bloco (8 P/B), 16 palavras por bloco (16 P/B), 32 palavras por bloco (32 P/B) e 64 palavras por bloco (64 P/B). O tempo de acesso à memória leva em consideração a busca de informações (dados e instruções).

Aplicação	Tempo de Acesso à Memória(ns)					
	Original	Prop.(4P/B)	Prop.(8P/B)	Prop.(16P/B)	Prop.(32P/B)	Prop.(64P/B)
Acumulador	9330	7430	7070	6710	6910	7020
DCT	313190	96460	94550	93570	93510	93910
RLE	51730	14640	13940	13820	13740	13940
Mult. Matriz	270330	67730	66440	65960	65980	66230

Tabela 2 – Comparação do Tempo de Acesso à Memória Considerando o IPNoSys Original e o Proposto em Função da Quantidade de Palavras por Bloco

Em todos os casos, o tempo de acesso à memória é menor que no IPNoSys

original (comparando-se as mesmas aplicações), chegando a ser cerca de 4 vezes menor nas aplicações DCT e multiplicação de matrizes (aplicações com forte localidade espacial e temporal). Ainda assim, também apresenta redução do tempo de acesso em aplicações com apenas localidade espacial como na aplicação "Acumulador"(ACC). A Figura 34, exibe o gráfico que mostra a diferença entre os tempos de acesso à memória comparando o melhor tempo das aplicações no IPNoSys proposto e o tempo obtido no IPNoSys original.

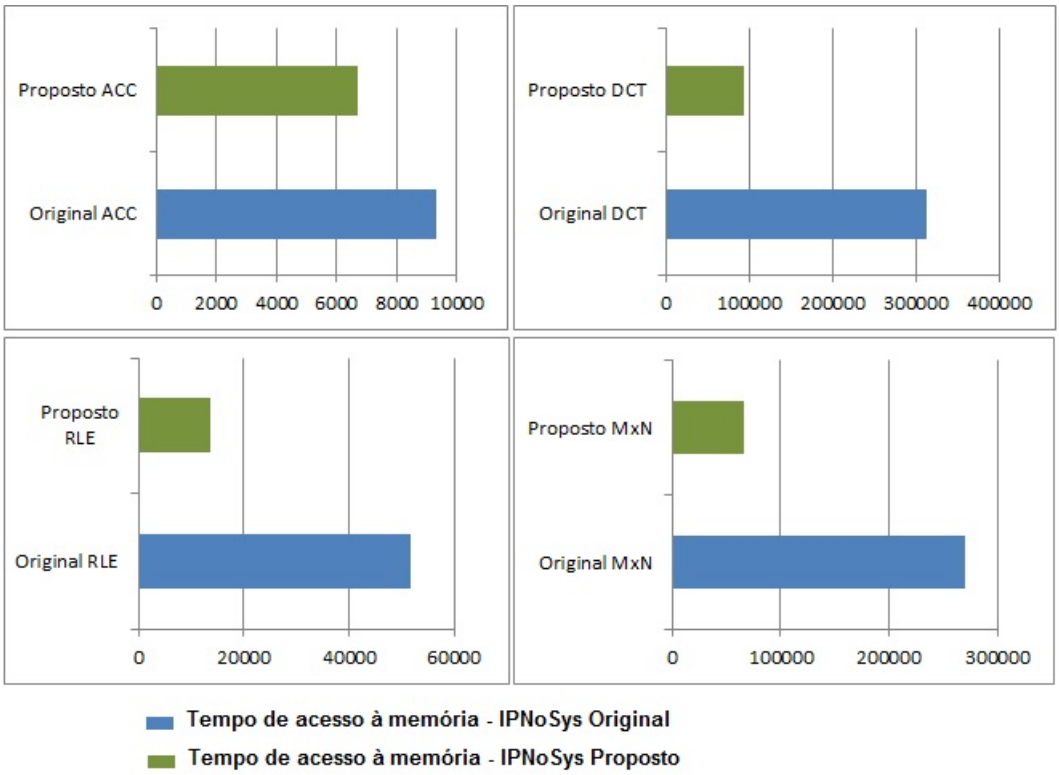

Figura 34 – Gráfico - Tempo IPNoSys Proposto x IPNoSys Original

Fonte: Próprio Autor

Analisando a Tabela 2, nota-se que o tempo gasto em acesso à memória diminui em função do tamanho do bloco até um determinado tamanho, isso ocorre porque aumentando o tamanho do bloco diminui-se a taxa de erros (*miss*) na cache (ver Tabela 3), de modo que se diminuem os acessos à memória principal que é mais lenta, porém com blocos de 32 palavras e 64 palavras o tempo de acesso à memória começa a aumentar. O Gráfico exibido pela Figura 35, mostra o comportamento do tempo de acesso à memória em função do tamanho do bloco, considerando uma média desse tempo das aplicações exibidas na Tabela 2.

Observa-se que, em geral, todas as aplicações apresentam uma diminuição no tempo de acesso à memória até o tamanho de bloco com 16 palavras, mas para blocos de tamanho superior a esse, o tempo começa a aumentar. Esse tipo de comportamento é esclarecido por Hennessy e Patterson (2003), explicando que quanto maior o bloco, mais

	Taxa de miss (%)				
Aplicação	**4P/B**	**8P/B**	**16P/B**	**32P/B**	**64P/B**
ACC	15,19	9,97	5,21	3,17	2,04
DCT	3,16	1,58	0,79	0,45	0,26
RLE	6,63	3,55	2,13	1,18	0,71
Mult. Matriz	2,59	1,32	0,71	0,44	0,27

Tabela 3 – Taxa de *miss*

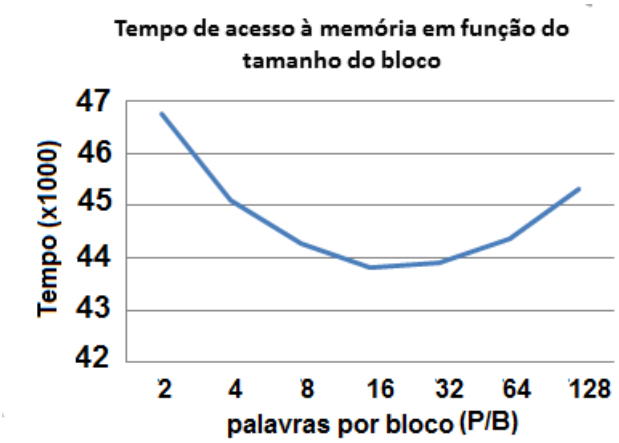

Figura 35 – Gráfico Tempo x Tamanho do Bloco

Fonte: Próprio Autor

custoso é para trazê-lo da memória principal até a memória cache. Os autores também afirmam que, é possível a partir de um determinado tamanho do bloco, considerando a concorrência entre blocos na cache, que o tempo gasto com acesso à memória volte a aumentar, já que esse bloco será substituído constantemente. O que acontece nos testes é exatamente o esperado, onde o tamanho exagerado do bloco aumenta o tempo para leva-lo à memória causando desvantagem. Por tanto, o tamanho de bloco ideal para a arquitetura em questão é 16 palavras por bloco.

A Tabela 4, mostra a análise comparativa entre o tempo total de execução (TE) e o tempo gasto em acessos à memória (TM) para o IPNoSys proposto, considerando o melhor caso (16 palavras por bloco) em contraste com os valores obtidos no IPNoSys original, a coluna (R) mostra a relação entre o tempo de acesso à memória e o tempo de execução.

	Original			Proposto		
Aplicação	**TE(ns)**	**TM(ns)**	**R(%)**	**TE(ns)**	**TM(ns)**	**R(%)**
Acumulador	14020	9330	65	11400	6710	58
DCT	390420	313190	80	170800	93570	54
RLE	51730	45850	88	19700	13820	70
Mult. Matrizes	270330	212000	78	124290	65960	53

Tabela 4 – Comparativo entre o Tempo de Acesso à Memória em Relação ao Tempo Total

Nota-se que no IPNoSys original gastava-se 80% do tempo acessando a memória na DCT, esse percentual diminui para 54% no IPNoSys proposto. As aplicações DCT e Multiplicação de Matrizes apresentam essa redução de até 26%, enquanto as outras aplicações obtiveram uma redução menor (até 18%). Isso acontece porque essas aplicações possuem forte localidade espacial e temporal, nelas os impactos são maiores, uma vez que as mesmas informações são utilizadas várias vezes, passam mais tempo em cache e consequentemente o tempo para acessá-las é menor.

Além de analisar o tempo de acesso à memória, também levou-se em consideração a taxa de *miss* em cada aplicação, ou seja, quantas vezes o dado foi requisitado e não estava em cache. A Tabela 3, e a Figura 36, mostram os valores obtidos variando o tamanho do bloco, mas mantendo o tamanho da palavra.

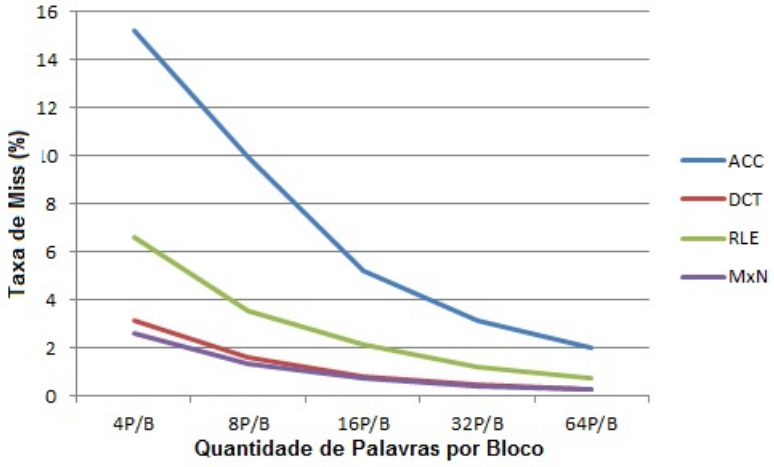

Figura 36 – Gráfico Taxa de Miss x Tamanho do Bloco

Fonte: Próprio Autor

Pode-se notar que a taxa de *miss* diminui em função da quantidade de palavras por bloco, de modo que quanto maior o bloco, menor a chance de o dado não estar em cache. A média de Taxa de *miss* do modelo de hierarquia proposto para blocos com 16 palavras é cerca de 2,19%.

Outros testes foram realizados para verificar o impacto da funcionalidade que permite buscar dados ou instruções sempre na MAU mais próxima da RPU que solicita. Para isso, foi levado em consideração os seguintes casos exibidos na Figura 37.

- Caso 1: Quando a RPU que solicita um dado/instrução está diretamente ligada a MAU, 0 *hops*.
- Caso 2: Quando a MAU que possui o dado/instrução está na distância média no caminha do pacote da RPU, 3 *hops*. (Ex.: RPU 0-3 requisita da MAU 0-0).

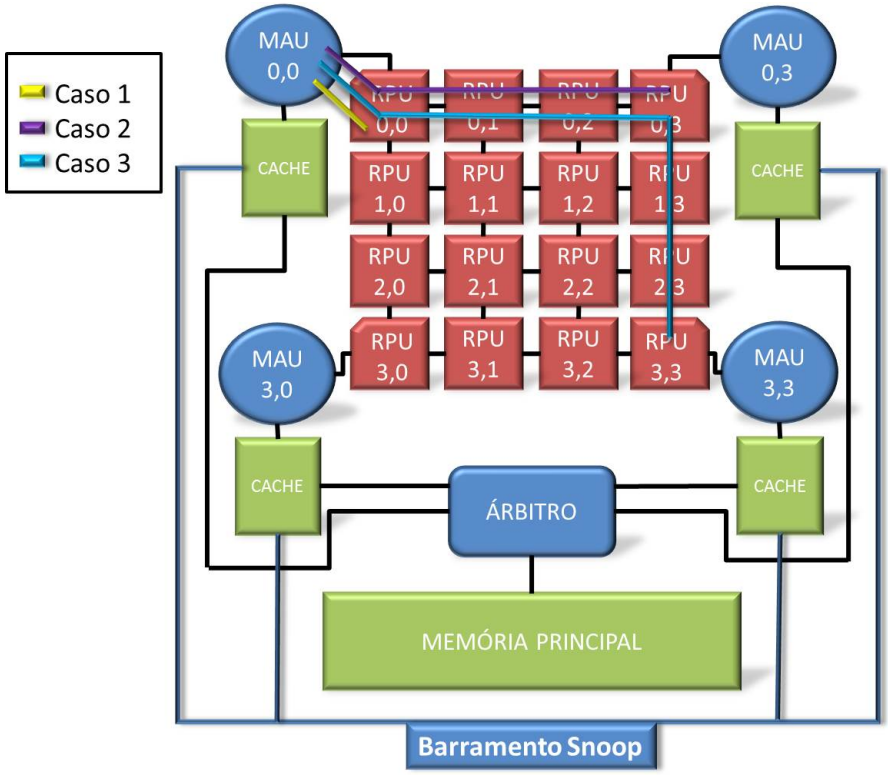

Figura 37 – Casos de Acesso à Memória a partir das RPUs

Fonte: Próprio Autor

- Caso 3: Quando a RPU que solicita um dado/instrução está na diagonal oposta da rede, 6 *hops*. (Ex.: RPU 3-3 requisita dado na MAU 0-0).

A Tabela 5, apresenta uma média de tempo de espera por dado obtido para blocos com 16 palavras.

	Tempo para Receber o Dado(ns)		
	Tempo de Espera da Memória(ns)	**Tempo de Espera da Rede(ns)**	**Tempo Total(ns)**
Caso 1	100	27	127
Caso 2	100	207	307
Caso 3	100	387	487

Tabela 5 – Relevância do Tempo de Espera na rede em função do Tempo Total

O tempo de espera da memória é o tempo gasto do momento em que a MAU requisita um dado à memória até a obtenção desse dado. Observa-se que na configuração apresentada no início desta seção esse tempo é uma constante de 100ns, sendo que desse valor 20ns é o tempo de acesso à memória cache, 60ns é o tempo de acesso à memória principal e 20ns é o tempo de gerenciamento do árbitro. No caso em que o dado já está na cache, esse tempo é de apenas 20ns. Considerou-se o pior caso para mostrar como o tempo de espera na rede é extremamente relevante diante do tempo total de espera por um dado.

Os resultados mostram que, como esperado, quanto menor a distância entre MAU e RPU, menor o tempo de espera na rede e, consequentemente, menor o tempo de espera pelo dado. Verifica-se ainda que o tempo gasto para realizar 1 *hop* é de 30ns, diante disso, gerou-se a Equação 5.2, que calcula quanto tempo médio de espera na rede será necessário para qualquer RPU:

$$TempoEspera = 2 * (QtdHops * Th) + TempoMem + To \tag{5.1}$$

A variável *QtdHops*, refere-se a quantidade de *hops* que deverão ser feitos entre a RPU e a MAU, a constante *Th*, indica o tempo gasto para realizar 1 *hop* (nesse cenário corresponde a 30ns), a variável *TempoMem*, indica o tempo de acesso à informação armazenada na hierarquia de memória que varia de 20ns, quando a informação já está na cache e 100ns quando a informação está na memória principal. A constante *To*, refere-se ao tempo(ns) de overhead necessário para a comunicação entre a RPU e a MAU diretamente ligadas, nesse cenário corresponde a 27ns. Com isso, podemos considerar as RPUs (2,0), (2,1), (3,0) e (3,1), que formam o quadrante de RPUs mais próximas da MAU (3,0), variando a quantidade de hops de 0 a 2, e analisar o tempo de espera por informações para a MAU (3,0). A Tabela 6, mostra essa variação de tempo de todas elas, esse comportamento aplica-se aos demais quadrantes da IPNoSys.

RPU	**Tempo de Espera(ns)**
2,0	127
2,1	160
3,0	160
3,1	220

Tabela 6 – Tempo de espera por uma informação para as RPUs mais próximas de uma MAU

Nota-se que ocorre uma variação do tempo de espera de acordo com a distância entre RPU de origem e a MAU mais próxima. Com os resultados obtidos, constatou-se que é mais vantajoso buscar dados/instruções na MAU mais próxima, chegando a obter um tempo cerca de 3 vezes menor que o pior caso (Caso 3), diminuindo assim, a latência de rede.

Além disso, o programador não é mais o responsável por manter a coerência dos dados na memória, melhorando consideravelmente o modelo de programação e diminuindo a chance de possíveis erros que poderiam ser ocasionados por desatenção.

A Tabela 7, faz uma comparação entre o tempo total de execução de aplicações executadas no primeiro modelo de hierarquia proposto, tanto com a funcionalidade de buscar/executar instruções na MAU mais próxima, quanto sem essa funcionalidade. A Figura 38, exibe as diferenças do tempo de execução de cada aplicação com e sem a funcionalidade ”MAU_N”.

Aplicação	Tempo de Execução(ns)	
	Indicando a MAU	**Com MAU_N**
DCT	14362	7192
RLE	1676	1681
Mult. Matriz	11692	10864
Acumulador	664	821

Tabela 7 – Comparação de Tempo de Execução Entre Aplicações Utilizando a Funcionalidade MAU_N e Aplicações Sem a Funcionalidade

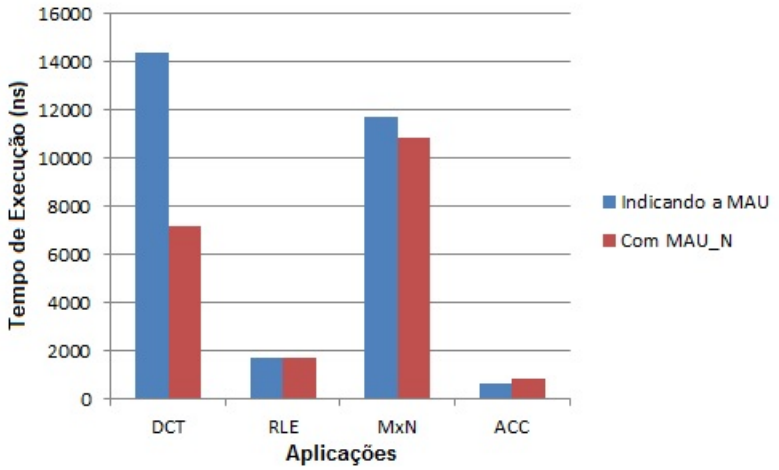

Figura 38 – Gráfico Tempo de execução x Aplicações - Comparação com e sem funcionalidade MAU_N

Fonte: Próprio Autor

Os resultados anteriores já mostraram que o primeiro modelo de hierarquia proposto apresentava mais eficiência, chegando a ser cerca de 5 vezes mais rápido que o modelo original. Com os resultados da Tabela 7, é possível ver que ainda melhora-se o tempo de execução em quase 2 vezes utilizando a funcionalidade que permite buscar informações na MAU mais próxima. Analisando os resultados obtidos com a aplicação "Acumulador", pode-se notar que o tempo de execução aumenta com o uso da funcionalidade MAU_N, isso ocorre porque essa aplicação inicia disparando, através de instruções EXEC, quatro pacotes destinados as 4 MAUs, de modo que existe 4 fluxos de execução ao mesmo instante explorando o paralelismo da arquitetura. Com a utilização da funcionalidade MAU_N, os pacotes serão injetados pela MAU mais próxima, dessa maneira o paralelismo não acontece e a aplicação é executada de forma sequencial. Esse resultado mostra que a utilização da funcionalidade MAU_N deve ser feita com cautela, principalmente quando se trata de aplicações com muitas instruções que controlam a injeção de pacotes de instrução, uma vez que ela pode destruir o paralelismo de execução desses pacotes.

Outro fato importante, que pode ser visto diante dos resultados apresentados na Tabela 7, é o aumento do tempo de execução da aplicação RLE, isso acontece por

conta do grande número de instruções do tipo SEND presentes nela. Como explicado na Seção 5.1, essas instruções são enviadas por *multicast* para todas as MAUs, isso aumenta a quantidade de pacotes com essas instruções gerando mais fluxo na rede e um *overhead* em cada MAU mesmo quando a instrução SEND não deve ser tratada por ela. Diante disso, percebe-se que a utilização da funcionalidade "MAU_N", mais uma vez, deve ser feita com cautela analisando cada caso.

O primeiro modelo de hierarquia proposto, apresenta melhor eficiência em aplicações onde a busca de dados é maior. Para que se entenda melhor o impacto desse modelo nessas aplicações, desenvolveu-se uma aplicação sintética que explora essa característica. A aplicação consiste em uma série de instruções LOADs sendo executadas sequencialmente (No mesmo pacote) considerando uma rede 4x4. Os resultados obtidos com os testes são exibidos na Tabela 8, e na Figura 39.

Número de LOADs	**Tempo de Execução(ns)**	
	Identificando a MAU	**Com MAU_N**
6	5290	5190
12	7240	6910
24	15260	11080

Tabela 8 – Resultados Aplicação Sintética Com a Funcionalidade MAU_N e Sem a Funcionalidade

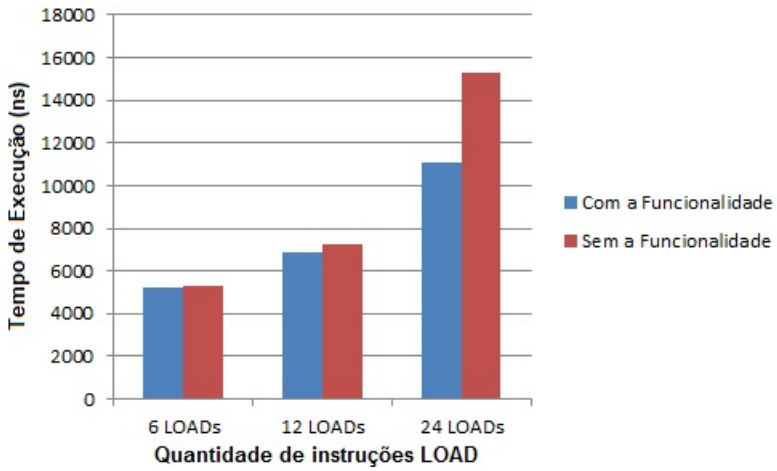

Figura 39 – Gráfico - Comparação do impacto da instrução LOAD com e sem funcionalidade MAU_N

Fonte: Próprio Autor

Verifica-se que como esperado, quanto maior o número de LOADs na aplicação, melhor o desempenho da arquitetura proposta e maior o impacto da funcionalidade que permite buscar na MAU mais próxima. Isso acontece porque, de maneira geral, LOADs são operações muito custosas para o sistema. No IPNoSys original esse tipo de aplicação deveria ser evitada por conta de diminuir o desempenho da arquitetura.

5.2 Resultados do segundo modelo

Como explicado na Seção 4.2 do Capítulo 4, o segundo modelo possui suas MAUs ligadas diretamente às RPUs que formam seus quadrantes. Essa configuração visa diminuir a quantidade de *hops* feita pelas RPUs, uma vez que no primeiro modelo apenas uma RPU estava diretamente ligada à MAU e as demais precisavam fazer de 1 a 2 *hops* para acessar a MAU. Diminuir a quantidade de *hops* implica diretamente em diminuir o tempo de espera por uma informação e a latência na rede.

Para medir os impactos do segundo modelo, considerou-se os seguintes casos:

- Caso 1: A RPU (3,3) busca informação na MAU (3,3) - 0 *hops*;
- Caso 2: A RPU (3,2) busca informação na MAU (3,3) - 0 *hops*;
- Caso 3: A RPU (3,3) busca informação na MAU (0,3) - 2 *hops*;
- Caso 4: A RPU (3,3) busca informação na MAU (0,0) - 5 *hops*.

Analisando os casos à cima, verificou-se que a quantidades de *hops* necessários para que uma RPU acesse a MAU diminui em todos os casos com exceção do caso 1, pois nesse caso a RPU já apresentava ligação direta com a MAU. Diante dessa redução de *hops*, pode-se utilizar a mesma equação gerada para o primeiro modelo, apresentada e explicada na seção 5.1, para calcular o tempo de espera para qualquer RPU da rede:

$$TempoEspera = 2 * (QtdHops * Th) + TempoMem + To \tag{5.2}$$

As constantes e variáveis apresentam os mesmos valores do primeiro modelo: Th = 30ns, To = 27ns e $TempoMem$ variando de 20ns em caso de *hit* e 100ns em caso de *miss* (nesses casos considerou-se sempre o tempo de *miss* para os dois modelos). O único fator alterado é a $QtdHops$ referente a quantidade de *hops* entre a RPU e a MAU requisitada. Com essas informações, foi calculado o tempo de espera por informação para os casos apresentados à cima. A Tabela 9, mostra os resultados obtidos em contraste com o primeiro modelo.

	Tempo de Espera (ns)	
Caso	**Primeiro modelo**	**Segundo modelo**
Caso 1	127	127
Caso 2	187	127
Caso 3	307	247
Caso 4	487	427

Tabela 9 – Tempo de Espera por informação para o segundo modelo

Observa-se que nos casos 2, 3 e 4, o tempo de espera é menor que no primeiro modelo. Além disso, pode-se verificar que nos casos 1 e 2 o tempo é igual, isso acontece porque as RPUs (3,3) e (3,2) fazem parte do quadrante da mesma MAU, podendo afirmar

que todas as RPUs de um quadrante apresentarão o mesmo tempo para acessar a MAU. Considerando que no segundo modelo também pode ser utilizada a funcionalidade de buscar na MAU mais próxima, os casos 3 e 4 jamais acontecerão e o tempo de espera se resumirá ao tempo obtido nos casos 1 e 2.

Uma questão que pode ser discutida é a possibilidade de concorrência pelo acesso à MAU, uma vez que esse acesso é serializado e gerenciado por um gerenciador de requisições. Com base nisso e considerando o fato de que as RPUs que formam o quadrante da MAU podem requisitar acesso em um mesmo instante, o tempo de acesso à memória vai depender da prioridade de requisição de cada RPU. Considerando esses fatores, o tempo de acesso à memória pode ser calculado conforme a Equação 5.3.

$$TempoMem = Prioridade * (Tempo_m + To) \tag{5.3}$$

Nessa equação, calcula-se o novo tempo de acesso à memória (*TempoMem*) em função da prioridade de requisição. A variável $Tempo_m$, corresponde ao tempo de acesso à memória a partir da MAU, que pode variar de 20ns (se a informação está em cache) a 100ns (se a informação não está na cache), a variável *To*, é o tempo de *overhead* necessário para estabelecer conexão entre a RPU e a MAU (nesse cenário corresponde a 27ns). A variável *Prioridade*, corresponde à prioridade de execução de cada requisição onde quando menor o seu valor maior a prioridade, de modo que a requisição com prioridade 2 será executada depois da requisição com prioridade 1 e assim por diante. Considerando isso, o tempo de espera para acessar à memória depende da prioridade de cada requisição, ou seja, o quão rápido ela será atendida.

Apesar dessa situação de concorrência ser possível, maiores estudos não foram realizados sobre ela, pois percebeu-se que a serialização de acessos à MAU já ocorria tanto no primeiro modelo quanto no IPNoSys original. Nesses dois, a serialização acontecia a partir das RPUs dos cantos da rede que são ligadas diretamente às MAUs já que todas as requisições deveriam ser encaminhadas a elas e posteriormente seriam repassadas para as MAUs. Portanto, considerou-se apenas os casos onde a serialização não ocorre.

5.3 Resultados do terceiro modelo

Nessa seção são mostrados os resultados referentes ao terceiro modelo. Estes resultados são frutos de uma análise estática deste modelo, estipulados com base em dados coletados a partir de experimentos com a IPNoSys original e o primeiro modelo proposto.

Considerando a organização do terceiro modelo, com caches L1 privadas associadas a cada RPU, pode-se estipular o tempo de espera por um dado a partir da decodificação de uma instrução LOAD. A seguir, são apresentados alguns casos que

podem ocorrer, dependendo da localização física da RPU que requisita o dado e a RPU que possui o dado em cache, além de considerar também a possibilidade de o dado não estar em cache. Para melhor entendimento, considerar o termo RPU de origem referente a RPU que decodificou a instrução LOAD e RPU de destino referente a RPU que pode ter o dado em cache:

- Caso 1: a RPU de destino é a RPU de origem e possui o dado na cache L1.
- Caso 2: a RPU de destino está na diagonal oposta da rede à RPU de origem e possui o dado na cache L1, 6 *hops*.
- Caso 3: a RPU de destino está na distância média do caminho do pacote da RPU de origem e possui o dado na cache L1, 3 *hops*.
- Caso 4: a RPU de destino está na distância média do caminho do pacote da RPU de origem e possui o dado na cache L2, 3 *hops*.
- Caso 5: a RPU de destino está na diagonal oposta da rede à RPU de origem e possui o dado na cache L2, 6 *hops*.
- Caso 6: a RPU de destino está na distância média do caminho do pacote da RPU de origem e possui o dado na memória principal, 3 *hops*.
- Caso 7: a RPU de destino está na diagonal oposta da rede à RPU de origem e possui o dado na memória principal, 6 *hops*.

Diante dos casos apresentados e utilizando a Equação 4.2, e a Equação 4.3 (apresentadas na seção 4.3), pode-se calcular o tempo de espera por dados para cada um deles. Os resultados esperados são exibidos na Tabela 10.

Caso	**Tempo de Espera por Dados (ns)**
Caso 1	20
Caso 2	380
Caso 3	200
Caso 4	247
Caso 5	427
Caso 6	327
Caso 7	487

Tabela 10 – Tempo de Espera por dados - terceiro modelo de hierarquia

Observa-se que, como esperado, os valores de tempo de espera por dados são menores que no primeiro modelo proposto, obtendo resultados inferiores apenas para os casos 6 e 7 onde ocorre *miss*, tanto na cache L1, quanto na cache L2. Isso acontece devido a localização física da cache, que agora está dentro da rede, diminuindo assim a latência. Nota-se ainda que no caso 2 o tempo de espera é maior que no caso 6, com

isso é possível verificar que dependendo da distância entre a RPU de origem e a RPU de destino será mais vantajoso buscar o dado diretamente na memória principal. Para tomar essa decisão, deve-se desenvolver uma função que calcula a distância entre a RPU de origem e a RPU de destino e compara com a distância entre a RPU de origem e a MAU associada a cache L2, de modo a fazer a busca sempre na mais próxima. O trabalho de Barcelos (2008), também explora esse tipo de comportamento, analisando a viabilidade de buscar informações com base no custo de deslocamento dentro da rede, concluindo que essa técnica pode reduzir o tempo de transferência em até 24%. Vale ressaltar que esses casos só ocorrem quando o programador não estiver utilizando a funcionalidade de buscar sempre na MAU mais próxima.

Outro fato levado em consideração é a questão relacionada à injeção de pacotes na rede, no IPNoSys original e no primeiro modelo proposto, os pacotes eram injetados apenas pelas MAUs nos cantos das redes. No segundo modelo, qualquer RPU pode injetar um pacote, desde que ele esteja em sua cache L1. A seguir são apresentados alguns casos que podem ocorrer considerando a requisição de um pacote na rede. Para melhor entendimento, o termo RPU de destino refere-se à RPU que pode ter o pacote em cache e RPU de origem refere-se à RPU que decodifica a instrução de busca por pacote.

- Caso 1: a RPU de destino é a RPU de origem e possui o pacote na cache L1.
- Caso 2: a RPU de destino está na diagonal oposta da rede à RPU de origem e possui o pacote na cache L1, 6 *hops*.
- Caso 3: a RPU de destino está na distância média do caminho do pacote da RPU de origem e possui o pacote na cache L1, 3 *hops*.
- Caso 4: a RPU de destino está na distância média do caminha do pacote da RPU de origem e possui o pacote na cache L2, 3 *hops*.
- Caso 5: a RPU de destino está na diagonal oposta da rede à RPU de origem e possui o pacote na cache L2, 3 *hops*.
- Caso 6: a RPU de destino está na distância média do caminha do pacote da RPU de origem e possui o pacote na memória principal, 3 *hops*.
- Caso 7: a RPU de destino está na diagonal oposta da rede à RPU de origem e possui o pacote na memória principal, 6 *hops*.

Diante dos casos apresentados e utilizando a Equação 4.6 e a Equação 4.7 (apresentadas na seção 4.3), é possível calcular o tempo de injeção de pacotes para os casos anteriormente apresentados, a Tabela 11 exibe os valores obtidos para cada um deles.

Caso	Tempo para Injeção de Pacotes na rede(ns)
Caso 1	40
Caso 2	220
Caso 3	130
Caso 4	177
Caso 5	267
Caso 6	237
Caso 7	327

Tabela 11 – Tempo para injeção de pacotes

Observa-se que o tempo para injeção de pacotes, assim como o tempo de espera por dados, depende da distância entre a RPU de origem e a RPU de destino, e também do fato de o pacote estar ou não em cache. Como o pacote pode estar na cache L1 da RPU, ou seja, dentro da rede favorece instruções do tipo EXEC e SYNEXEC (instruções que injetam pacotes). Supondo que uma aplicação injeta o mesmo pacote várias vezes e esse pacote já está na rede, o tempo para injetá-lo será menor e a execução será mais rápida.

Vale ressaltar que o terceiro modelo também permite a utilização da funcionalidade que busca na MAU mais próxima. Utilizando essa funcionalidade os casos 2, 3, 4, 5, 6 e 7 não acontecerão já que essas RPUs irão buscar as informações na MAU mais próxima (e nunca na MAU da diagonal oposta). Como as RPUs do quadrante estão diretamente ligadas à MAU, no pior caso ocorrerão 2 *hops* para consultar a cache L2 de outra RPU dentro do mesmo quadrante. O tempo para o pior caso é de 207 ns, referente ao caso onde ocorre *miss* tanto na cache L1 quanto na cache L2.

5.4 Considerações sobre os resultados

Com os resultados apresentados nesse Capítulo, pode-se notar como é possível melhorar o desempenho da arquitetura IPNoSys aplicando uma hierarquia de memória eficiente. Além de apresentar melhorias consideráveis no desempenho, o modelo de programação da arquitetura também foi aprimorado retirando parte da responsabilidade do programador sobre a manutenção da coerência de memória. Esta última melhoria também impactou diretamente na diminuição da latência da rede.

O segundo modelo de hierarquia proposto, apresenta uma evolução natural do primeiro, com a ligação direta entre MAUs e RPUs que formam o seu quadrante foi possível reduzir o tempo de espera por dado em até 20% em relação ao primeiro modelo.

Levando em consideração o terceiro modelo de hierarquia proposto, é visível como a nova forma de organizar as caches dentro da rede em cada RPU melhora o tempo de espera por um dado diminuindo ainda mais a latência da rede e melhorando o desempenho da arquitetura. O fato de cada RPU poder injetar um pacote a partir de sua própria cache L1 possibilita a injeção de até 4 fluxos de execução por quadrante

	Tempo de Espera (ns)		
Caso	**Primeiro modelo**	**Segundo modelo**	**Terceiro modelo**
Caso 1	107	47	20
Caso 2	187	127	147
Caso 3	347	287	260
Caso 4	427	367	387

Tabela 12 – Comparação direta entre os três modelos de hierarquia

simultaneamente, melhorando ainda mais a exploração do paralelismo da arquitetura.

Uma análise quantitativa dos impactos de um modelo sobre o outro só poderá ser feita após a implementação dos dois últimos modelos e a coleta de resultados mediante simulação. Entretanto, pode-se analisar os seguintes casos para observar o comportamento dos três modelos e contrastar seus desempenhos. Os resultados obtidos são apresentados na Tabela 12:

- Caso 1: RPU(3,2) busca informação na MAU(3,3) e essa informação está em cache;
- Caso 2: RPU(3,2) busca informação na MAU(3,3) e essa informação está na memória principal;
- Caso 3: RPU(3,2) busca informação na MAU(0,0) e essa informação está em cache;
- Caso 4: RPU(3,2) busca informação na MAU(0,0) e essa informação está na memória principal.

Considerando o caso 1, sabe-se que o primeiro modelo vai precisar executar 1 *hop* da RPU(3,2) para a RPU(3,3), e esta é quem vai se comunicar com a MAU(3,3) solicitando a informação. No segundo modelo a RPU(3,2) possui ligação direta com a MAU(3,3) e não necessita executar nenhum *hop*. Já o terceiro modelo não precisa executar nenhum *hop*, ou ainda comunicar-se com a MAU(3,3), já que o dado está na cache da própria RPU(3,2). Portanto, para esse caso, o modelo mais eficiente é o terceiro. Entretanto, se forem considerados os caso 2 e 4, o primeiro e o segundo modelo continuam com o mesmo comportamento, mas o terceiro modelo terá um *miss* na cache da RPU(3,2), e outro na cache associada a MAU(3,3) de modo que se torna mais custoso buscar na memória principal. Diante disso, pode-se concluir que se a taxa de *miss* for muito elevada, o terceiro modelo terá seu desempenho inferior aos demais. Além disso, verificou-se na seção 5.3, que este modelo pode apresentar um desempenho inferior se a informação buscada estiver em uma RPU muito afastada da RPU de origem, sendo mais vantajoso fazer a busca diretamente na memória principal. Com base nesses resultados, é possível afirmar que o segundo modelo apresenta melhor desempenho e maior estabilidade que os outros.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

O IPNoSys é um exemplo de arquitetura que utiliza redes-em-chip para interconectar suas unidades de processamento e roteamento. Muitas pesquisas estão sendo desenvolvidas sobre essa arquitetura para melhorar seu desempenho, porém a parte da arquitetura que trata da utilização de memórias ainda é pouco explorada. A proposta deste trabalho foi explorar formas de organizar a memória do IPNoSys em níveis hierárquicos, buscando melhorar seu desempenho e facilitar a sua programação até então desenvolvida sobre memórias distribuídas.

Alguns modelos de memórias foram estudados e três modelos foram escolhidos para serem testados sobre a arquitetura IPNoSys. O primeiro, utilizando memória compartilhada e distribuída, onde existe um módulo único compartilhado (memória principal) e, um nível acima, existem quatro módulos distribuídos (memórias caches) atrelados às quatro MAUs do sistema original. O segundo modelo, surge como uma evolução do primeiro, sendo que consiste em ligar todas as RPUs de um quadrante à sua MAU. Este modelo reduz a quantidade de *hops* e diminui a espera por dados e instruções. Apresentou-se ainda um outro modelo onde cada RPU do sistema possui sua própria memória cache L1 privada, de modo a explorar a possibilidade de consultar as caches durante o roteamento dos pacotes para que quando um dado for requisitado seja lido da memória cache mais próxima. Esse outro modelo pode reduzir consideravelmente a latência do sistema.

Alguns testes foram realizados, e com os resultados obtidos, observou-se que, de fato a aplicação do primeiro modelo de memórias proposto mostrou-se eficiente podendo ser até 4 vezes mais eficiente dependendo da aplicação. Verificou-se ainda que em aplicações com forte localidade espacial e temporal o impacto é ainda maior, mas também apresenta bons números em aplicações totalmente sequenciais apenas com localidade espacial. Por último, nota-se que o fator "quantidade de palavras por bloco" afeta diretamente o tempo de acesso à memória, uma vez que quanto maior o número de palavras em um bloco, maior a taxa de acertos na cache e menor a quantidade de acessos à memória principal que é mais lenta. Porém, existe um ponto em que aumentar a quantidade de palavras por bloco não implica vantagens no tempo de acesso à memória, considerando o bloco com 16 palavras o tamanho ideal para essa arquitetura. Outro fator analisado foi a taxa de *miss* que, para esse tamanho de bloco, apresenta uma média de 2,19%.

Verificou-se que o tempo de espera na rede é extremamente relevante, podendo aumentar consideravelmente o tempo total de espera por um dado. Diante disso, mostrou-se como a funcionalidade de buscar na MAU mais próxima pode diminuir esse

tempo em até 3 vezes, tornando ainda mais eficiente o modelo de hierarquia proposto que consegue ser até 2 vezes mais rápido que o mesmo modelo sem a utilização dessa funcionalidade em aplicações diversas.

Nota-se que o primeiro modelo de hierarquia de memórias proposto apresenta melhor eficiência em função da quantidade de instruções LOADs, mas que também é eficiente nos demais casos, já que diminui a quantidade de *hops* na rede diminuindo consequentemente a latência na mesma. Por fim, verifica-se que a funcionalidade de buscar na MAU mais próxima deve ser utilizada com cautela, uma vez que utilizada de forma imprudente, pode causar a eliminação do paralelismo de pacotes e instruções na rede, aumentando o tempo de execução de aplicações desse tipo.

Com os resultados obtidos a partir do segundo modelo de hierarquia de memória proposto, foi possível reduzir em até 20% o tempo de espera por informação em relação ao primeiro modelo.

Analisando o terceiro modelo de hierarquia de memória proposto, verificou-se que ele apresenta uma redução da latência na rede para busca de dados ainda mais eficiente que o segundo modelo. Como os pacotes podem ser injetados na rede a partir de uma RPU dentro da rede o tempo de espera para a injeção do pacote diminui, melhorando o tempo de execução de aplicações em que as instruções de EXEC e SYNXECE são mais comuns, de modo a favorecer aplicações com laços de repetição. Entretando, em caso de taxa de *miss* muito elevada o seu desempenho é inferior se comparado com o segundo modelo. O segundo e o terceiro modelo possuem seus resultados mais eficientes quando se utiliza a funcionalidade de buscar na MAU mais próxima.

Existem uma grande quantidade de trabalhos futuros que podem ser desenvolvidos a partir das conclusões desse trabalho. Alguns são listados a seguir:

6.1 Implementação do Segundo e do Terceiro Modelo

Nas seções 4.2 e 4.3 foram exibidos o segundo e o terceiro modelo de hierarquia de memória propostos para a IPNoSys. Para esses modelos, foram feitos estudos descritivos e analíticos possibilitando estipular resultados esperados.

Como trabalho futuro, pretende-se implementá-los e simulá-los verificando e confrontando os resultados que serão obtidos com os resultados esperados a partir deste trabalho. Resultados que indicam o quanto o tráfego aumenta na rede com a troca de pacotes para fazer LOAD e STORE a partir das RPUs e mensurar os impactos de um modelo sobre o outro só poderão ser analisados após implementação e simulação.

6.2 Busca/Acesso Inteligente

Levando em consideração que nos três modelos de hierarquia de memória apresentados, no caso de taxas de *miss* muito elevadas, os resultados serão seriamente

afetados ocasionando uma perda de eficiência, uma solução para manter a taxa de *miss* sempre baixa é desenvolver mecanismos que possam identificar a MAU ou RPU que já possui a informação que está sendo buscada, pois dessa forma o tempo de espera por essa informação será menor, uma vez que ela já encontra-se na cache. Com isso, as informações sempre seriam acessadas com mais eficiência e velocidade.

Ainda explorando a busca inteligente, outro trabalho que pode ser desenvolvido e muito bem se aplica ao primeiro modelo de hierarquia proposto é a busca/acesso na MAU mais ociosa. No primeiro modelo apresentou-se uma função que busca sempre na MAU mais próxima, porém essa MAU pode está sendo muito requisitada de modo que demora a atender todas as requisições. Uma solução para isso é desenvolver mecanismos que permitam saber o nível de ociosidade das MAUS e calcular a viabilidade de requisitar a MAU que atenderá a requisição mais rápido, mesmo que o caminho até ela seja maior.

6.3 Aprimorando a Consistência de dados

Existem casos em que garantir que todos os módulos de memória distribuídas tenham o dado mais atual não é suficiente para garantir coerência de dados. Em alguns momentos é necessário garantir que apenas um núcleo processante está manipulando um determinado dado e que só após ele terminar um outro poderá manipular o mesmo dado. Na IPNoSys existem algumas instruções que podem ser utilizadas para isso. A instrução SYNC e SYNEXEC garantem que um determinado pacote só será injetado na rede para execução quando outros forem injetados, porém ainda não existe a garantia de que o mesmo dado não está sendo manipulado por outra RPU. Para isso, uma solução poderia explorar o desenvolvimento de instruções novas para o IPNoSys com a intenção de prover a funcionalidade *LOCK*, que é útil em aplicações onde há concorrência de dados e ainda é necessário a intervenção do programador para garantir a coerência, desse modo o programador poderia identificar qual dado estaria bloqueado e só poderia ser manipulado por uma RPU.

REFERÊNCIAS

ANDERSON, J. P.; GLASER, E. L. Automatic utilization of hierarchical memories. *American Institute of Electrical Engineers, Part I: Communication and Electronics, Transactions of the*, v. 82, n. 2, p. 288–292, May 1963. ISSN 0097-2452. Citado na página 34.

ARAÚJO, S. R. F. de. *Estudo da Viabilidade do Desenvolvimento De Sistemas Integrados Baseados em Redes em Chip Sem Processadores: Sistema IPNoSys.* Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, 2008. Programa de Pós-Graduação em Sistemas e Informação. Citado na página 11.

ARAÚJO, S. R. F. de. *Projeto de Sistemas Integrados de Propósito Geral Baseados em Redes em Chip - Expandindo as Funcionalidades dos Roteadores para Execução de Operações: A plataforma IPNoSys.* Dissertação (Doutorado) — Universidade Federal do Rio Grande do Norte, 2012. Programa de Pós-Graduação em Sistemas e Informação. Citado 6 vezes nas páginas 26, 29, 30, 31, 32 e 48.

BARCELOS, D. *Modelo de Migração de Terefas para MPSoCs baseados em Redes-em-Chip*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, Porto Alegre, Março 2008. Citado na página 72.

BELLEW, M.; HSU, M.; TAM, V.-O. Update propagation in distributed memory hierarchy. In: *Data Engineering, 1990. Proceedings. Sixth International Conference on*. [S.l.: s.n.], 1990. p. 521–528. Citado na página 34.

CARARA, E. A. *Uma Exploração Arquitetural de Redes Intra-chip com Topologia Malha e Modo de Chaveamento Wormhole.* Dissertação (Trabalho de Conclusão II) — Pontifícia Universidade Católica do Rio Grande do Sul., Porto Alegre, 2004. Citado na página 25.

CHANSON, S.; SINHA, P. Optimization of memory hierarchies in multiprogrammed computer systems with fixed cost constraint. *Computers, IEEE Transactions on*, C-29, n. 7, p. 611–618, July 1980. ISSN 0018-9340. Citado na página 34.

CONCER, N.; IAMUNDO, S.; BONONI, L. aequalized: A novel routing algorithm for the spidergon network-on-chip. In: *Design, Automation & Test in Europe Conference & Exibition*. [S.l.: s.n.], 2009. p. 749–754. Citado na página 24.

CONRAD, D. F. *Análise da Hierarquia de Memória em GPGPUs*. Dissertação (Monografia) — Universidade Federal do Rio Grande do Sul - (UFRS), 2010. Citado na página 37.

FILHO, R. N. *Hierarquia de Memória*. 2001. Notas de Aula. Disponível em: <http://www.di.ufpb.br/raimundo/Hierarquia/Registradores.html>. Citado na página 20.

GUNTZEL, J. L. *Circuitos de Armazenamento*. 2001. Notas de Aula. Disponível em: <http://www.inf.ufsc.br/~guntzel/isd/isd5.pdf>. Citado na página 20.

HAKE, J.-F.; HOMBERG, W. The impact of memory organization on the performance of matrix multiplication. In: *Supercomputing '90., Proceedings of*. [S.l.: s.n.], 1990. p. 34–40. Citado na página 34.

HENNESSY, J. L.; PATTERSON, D. *Arquitetura de Computadores - Uma abordagem Quantitativa*. 3. ed. San Francisco: Morgan Kaufmann, 2003. Citado 4 vezes nas páginas 16, 17, 21 e 62.

HWANG, K. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. [S.l.]: [S1], 1993. Citado na página 21.

INTEL. *Intel Processor i7-6700K*. 2016. Especificacoes Tecnicas. Disponível em: <http://ark.intel.com/pt-br/products/88195/Intel-Core-i7-6700K-Processor-8M-Cache-up-to-4_20-GHz>. Citado na página 18.

JOUPPI, N. P.; ALKHATIB, H. S. Guest Editors' introduction: Hot chips and the microprocessor. v. 16, n. 2, p. 6–7, abr. 1996. ISSN 0272-1732 (print), 1937-4143 (electronic). Citado na página 61.

LEE, H. G.; AL. et. On-chip communication architechture exploration: a quantitative exploration of point-to-point, bus and network-on-chip architechtures. *ACM Transactions on Design Automation of Eletronic System*, v. 12, p. 21–40, 2007. Citado na página 25.

PATTERSON, D.; SEQUIN, C. Design considerations for single-chip computers of the future. *Solid-State Circuits, IEEE Journal of*, v. 15, n. 1, p. 44–52, Feb 1980. ISSN 0018-9200. Citado na página 34.

PATTON, P. Trends in data organization and access methods. *Computer*, v. 3, n. 6, p. 18–24, Nov 1970. ISSN 0018-9162. Citado na página 34.

PILLON, M.; RICHARD, O. Escalonamento adaptativo ao uso da hierarquia de memória para máquinas multiprocessadas. In: . Foz do Iguaçú - Brazil: SBC, 2004. Citado 2 vezes nas páginas 35 e 36.

PRETOT, F.; GREINER, A.; GOMEZ, P. On cache coerency and memory consistency inssues in noc based shared memory multiprocessor soc architecture. *EUROMICRO Conference on Digital System Design*, v. 9, 2006. Citado na página 38.

REGO, R. S. D. L. S. *Projeto de Implementação de uma Plataforma MP-SoC usando SystemC*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, Departamento de Informática e Matemática Aplicada, Natal., 2006. Citado 2 vezes nas páginas 25 e 40.

SILVA, G. da et al. Mh-tedsim: Simulador de hierarquia de memória com simulação dirigida por execução ou por rastro. In: *I Concurso de Trabalhos de Iniciação Científica em Arquitetura de Computadores e Computação de Alto Desempenho, WSCAD-CTIC 2007*. [S.l.: s.n.], 2007. Citado 2 vezes nas páginas 36 e 37.

SILVA, G. G. B. da. *Estudo sobre o impácto da Hierarquia de Memória em MPSoCs baseados em NoC*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, 2009. Programa de Pós-Graduação em Computação. Citado 5 vezes nas páginas 17, 24, 35, 39 e 40.

STALLINGS, W. *Arquitetura e Organização de Computadores*. 8. ed. [S.l.]: Pearson Education, 2010. Citado 3 vezes nas páginas 21, 22 e 23.

TANENBAUM, A. S. *Organização Estruturada de Computadores*. 5. ed. [S.l.]: Pearson Education, 2006. Citado na página 16.

YANG, X.; DU, H.; HAN, J. Research on node coding and routing algorithm for network-on-chip. In: COLLOQUIUM, I. I. (Ed.). *Computing, Communication, Control, and Management*. [S.l.: s.n.], 2008. p. 198–203. Citado na página 24.

ZEFERINO. A study on communication issues for systems-on-chip. *SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN*, Porto Alegre, Setembro 2002. Citado na página 25.